Parallel Graph-Embedding and the Mob Heuristic

John E. Savage
Markus G. Wloka

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-91-07
February 20, 1991

Parallel Graph-Embedding and the Mob Heuristic † ‡ §

John E. Savage
Markus G. Wloka

Brown University
Department of Computer Science, Box 1910
Providence, Rhode Island 02912
jes@cs.brown.edu
mgw@cs.brown.edu
February 20, 1991
PREPRINT

†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 and ARPA Order Nos. 4786, 6320.

‡The P-hardness results in this paper will be presented at the *Proceedings of the IMACS World Congress on Computation and Applied Mathematics, Special Session on Simulated Annealing*, Dublin, July 1991, [48].

§A report on the experiments with the *Mob* heuristic for grid and hypercube embeddings will appear at the *5th SIAM Conference on Parallel Processing for Scientific Computing*, March 1991, Houston [49].

Abstract

We show that important local search heuristics for graph embedding, such as simulated annealing, are inherently difficult to parallelize. We also present a new massively parallel heuristic that gives excellent results for large graphs on the Connection Machine.

Graph embedding is the NP-complete problem of mapping one graph into another while minimizing a cost function on the embedded edges of the graph. It finds application in VLSI placement and the minimization of data movement in parallel computers. An automatic graph-embedding tool optimizes communication resources, permits fault tolerance, and allows parallel programs to be divorced to some extent from the structure of the underlying communication network. Local search heuristics, of which steepest descent, Kernighan-Lin, and simulated annealing are well-known examples, have been established as the heuristics of choice for general graph-embedding problems. The recent availability of general-purpose parallel processing hardware and the need to solve very large problem instances have led to increasing interest in parallelizing local search heuristics.

We show that local search heuristics for grid and hypercube embeddings are P-hard (evidence of non-parallelizability) when the neighbors in the solution space are generated by swapping the embeddings of two vertices. Thus it is unlikely that a parallel algorithm exists that can find even a local minimum solution in polylogarithmic time in the worst case. This puts experimental results reported in the literature into perspective: attempts to construct the exact parallel equivalent of serial simulated-annealing-based heuristics for graph embedding have yielded disappointing parallel speedups.

Guided by the P-hardness results, we have developed a new massively parallel heuristic (which we call the *Mob* heuristic). Our heuristics swap large sets (mobs) of vertices across planes of a grid or hypercube. We report on an extensive series of experiments with our heuristics on the 32K-processor CM-2 Connection Machine for grid and hypercube embeddings that show impressive reductions in edge costs and run in less than 30 minutes on random graphs of 1 million edges. Due to excessive run times, previous heuristics reported in the literature were able to construct graph embeddings only for graphs that were 100 to 1000 times smaller than those used in our experiments. On small graphs, where simulated annealing and other heuristics have been extensively tested, our heuristic was able to find solutions whose quality was at least as good as simulated annealing.

Contents

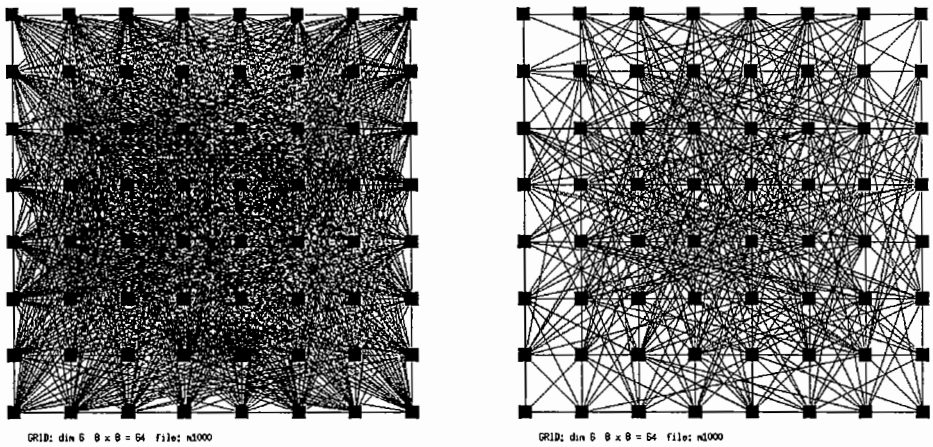

Figure 1: 16-to-1 grid embedding of a random graph: (a) Randomly generated solution (b) *Mob* heuristic solution

1 Introduction

In this paper we address the complexity of parallel local search algorithms for graph embedding, and present a new parallel heuristic, the *Mob* heuristic for embedding of graphs into grids and hypercubes. Here the problem is to embed a graph $G = (V, E)$ into other graphs and structures. A cost function assigns a value to each embedding; the objective is to *minimize* that cost function, as illustrated in Figure 1 and Figure 2. For clarity, we will call the target structure a *network*, and we will call the vertices of the target structure *nodes*.

Definition 1 Consider 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 $\nu : V_1 \mapsto V_2$ is a mapping of the vertices V_1 to the network nodes V_2 and ϵ maps edges in G_1 to paths in G_2. Let $\mathcal{S}$ be the set of all graph embeddings S of G_1 in G_2. Let the *cost function* $f : \mathcal{S} \mapsto \mathcal{R}$ assign a cost to each embedding in $\mathcal{S}$. The *graph-embedding problem* is to find a graph embedding S with minimum cost $f(S)$.

Graph partitioning is a special case of graph embedding, where the network G_2 consists of just two nodes. The goal of the *graph-partitioning problem* is to partition the vertices of an undirected graph $G = (V, E)$ ($|V|$ even) into two sets of equal size such that the number of edges between them is minimized. The minimum number of edges linking two sets of a partition is called the *bisection width* of the graph.

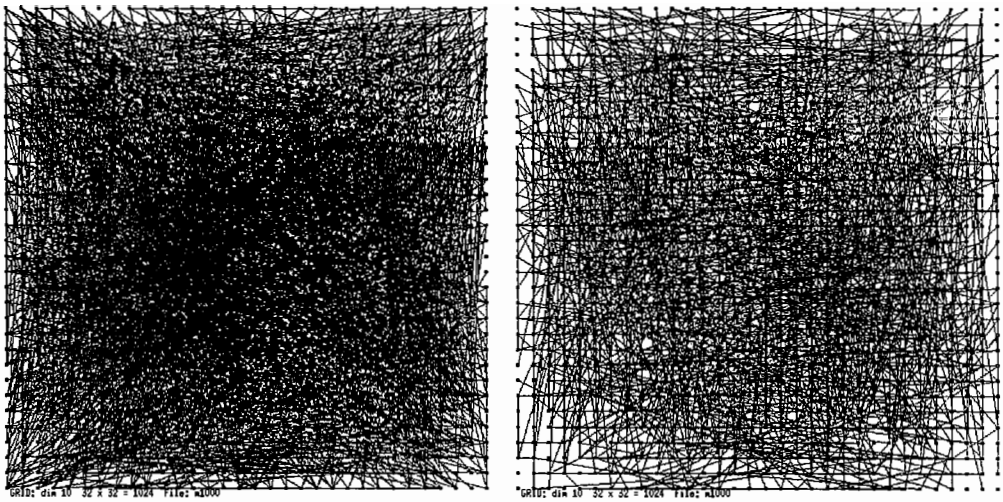

Figure 2: 1-to-1 grid embedding of a random graph: (a) Randomly generated solution (b) *Mob* heuristic solution

1.1 VLSI Placement and Minimization of Data Movement

Graph embedding finds application in VLSI placement and the minimization of data movement in parallel computers. The *VLSI placement problem* can be modeled by a grid embedding which is restricted to allow only one logic element location per grid node. *Hypercube embeddings* and *grid embeddings* are used to embed a data-structure graph on the topology of a parallel machine. An automatic graph-embedding tool optimizes communication resources, permits fault tolerance, and allows parallel programs to be divorced to some extent from the structure of the underlying communication network. SIMD-style parallel machines, such as the Connection Machine and the MasPar Machines, are characterized by small processors (one to four bits per processing element) that operate in lockstep (when activated) and a communication network that has a high latency unless nearest neighbors are accessed. Good code that makes efficient use of the processors can be written with traditional editors, compilers and debuggers. However, the cost of communication is so high on these machines, it must be severely controlled in order to obtain peak performance. One way to do this is to place tasks on nodes in such a way that the cost of communication between nodes is minimized. Instead of always using message-passing primitives for data access, data can be referenced across an edge by direct addressing if the connected vertex resides on the same processor. This will dramatically cut down on communication costs. The graph embeddings considered in this paper have the potential to do this.

1.2 Bulk Parallelism

An important paradigm for programming parallel machines is the *bulk parallelism model* introduced by Valiant [52]. In this model, v *virtual processors* are simulated on a computing network with (roughly) $v/\log v$ real processors. The Connection Machine software provides support for virtual processors. The advantages of the bulk parallelism model arise both from theoretical and practical observations.

There exist a number of important parallel algorithms used as atomic programming primitives which for problems of size n run in optimal asymptotic time on $n/logn$ processors. Increasing the number of processors past the $n/logn$ processor threshold yields diminishing returns because increasing the number of processors to n yields a decrease of a factor of at most two in the running time of the algorithm. This holds true for the *parallel prefix* of n numbers which can be computed in time $O(\log n)$ with $n/\log n$ processors.

Other algorithms where it is not advantageous to have more than $v/\log v$ processors include merging, sorting, memory access by hashing, and the simulation of FFT, hypercube, and Omega networks. For software development, the bulk parallelism model permits the program to be written independently of the actual number of available processors. The mechanism of simulating virtual processors and balancing computing loads need not be visible to the application program, and can be well matched to the underlying hardware. Should the hardware change, it is possible to alter the mechanism of simulating virtual processors without having to alter the applications program.

Current state-of-the-art hardware technology also favors the bulk parallel model. Floating-point processors make use of *pipelining*, a form of parallelism where an operation such as floating point multiplication or instruction decode, is subdivided into several hardware stages. Pipeline lengths can range from 2 to 64 stages. Once a data item has passed through a stage, the hardware is free to work on the next data item. Pipeline processors make efficient use of silicon real estate and have been very successful in conventional supercomputers. Since a processor in the bulk parallel model has to simulate $\log v$ virtual processors, it can keep its pipeline filled and work close to peak speed.

1.3 Local Search

Local search heuristics, of which steepest descent, Kernighan-Lin, and simulated annealing are well-known examples, have been established as the heuristics of choice for general graph-embedding problems. Using local search to produce an embedding of a problem into an architecture is an attractive technique. It simplifies porting a program to a new machine with a possibly esoteric network because one need only

change the cost functions and apply the same or a similar local search algorithm to produce a good embedding. This is vastly preferable to developing a completely new heuristic for each computing network. Also, concern for the quality of a heuristics is mitigated somewhat if it has an excellent track record on other problems.

The recent availability of general-purpose parallel processing hardware and the need to solve very large problem instances have led to increasing interest in parallelizing local search heuristics. Graph embedding heuristics are known to require large amount of processor time. We can expect the communication graph of a parallel supercomputer to be so large that it will not fit into the memory of a workstation. It is thus natural and elegant to consider graph embedding heuristics that run on the same parallel machine whose communication patterns they are ultimately supposed to optimize.

1.4 Cost Functions

We now define several cost functions on graph embeddings that are commonly used to estimate VLSI placement costs and interprocessor communication costs in parallel computers. These cost functions will measure a quantity equivalent to the length of embedded edges of G_1 in G_2, and can be computed very efficiently on a parallel machine. We note that none of the cost functions investigated in this paper measure routing congestion.

We begin by extending the concept of graph partitioning from a pair of sets to multiple sets. The $CROSS$ cost function counts the number of edges (v, w) that have their vertices in different sets of a partition:

Definition 2 Let $S : V \mapsto \{0..k-1\}$ be a partition of V into k sets. The cost of the partition under the CROSS cost function is defined as $\sum_{(v,w)\in E} CROSS(S(v), S(w))$, where

$$CROSS(a, b) := \begin{cases} 0 & a = b \\ 1 & a \neq b \end{cases}$$

The PERIMETER cost function is used to estimate the wire costs in a VLSI layout and to estimate communication costs in a grid of parallel processors.

Definition 3 Let $S : V \mapsto \{0..k-1\} \times \{0..l-1\}$ be an embedding of G into a $k \times l$-node grid. The cost of the embedded edges under the PERIMETER cost function is defined to be $\sum_{(v,w)\in E} PERIMETER(S(v), S(w))$, 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 whose x and y coordinates are a_x, a_y and b_x, b_y, respectively.

The HCUBE cost function is used to estimate communication costs in a hypercube network.

Definition 4 Let $S : V \mapsto \{0..2^k - 1\}$ be an embedding of G into a 2^k-node hypercube. The cost of the embedded edges under the $HCUBE$ cost function is defined as $\sum_{(v,w)\in E} HCUBE(S(v), S(w))$, where $HCUBE(a, b)$ is the Hamming distance of two nodes in the hypercube, when the integers a and b are represented as binary k-tuples.

1.5 Hypergraph Embeddings

The *hypergraph embedding problem* is a generalization of the graph-embedding problem [8,17]. Edges are replaced by sets of vertices, called hypergraph edges, or *nets* in VLSI terminology. Nets denote pins on cells that must be connected electrically in a physical layout.

The CROSS and PERIMETER cost functions have natural extensions to hypergraph edges. For the problem of partitioning a graph into k sets, the function CROSS$((a, b, \ldots, z))$ for an embedded net $(a, b, \ldots, z)$ is defined to be the number of different sets intersected by the net $(a, b, \ldots, z)$. The PERIMETER cost of an embedded net $(a, b, \ldots, z)$ is the perimeter of the bounding rectangle of the net:

$$\begin{aligned} perimeter((a, b, \ldots, z)) \;:=\; & |\max((a_x, b_x, \ldots, z_x)) - \min((a_x, b_x, \ldots, z_x))| + \\ & |\max((a_y, b_y, \ldots, z_y)) - \min((a_y, b_y, \ldots, z_y))| \end{aligned}$$

Since graph edges can be modeled as two-terminal nets, the completeness results presented in section 3 extend to local search heuristics applied to hypergraph embeddings.

1.6 Summary of Contributions

We show that local search heuristics for grid and hypercube embeddings are P-hard (evidence of non-parallelizability) when the neighbors in the solution space are generated by swapping the embeddings of two vertices. We have shown in previous papers [45,46,47] that for graph partitioning, local search under the SWAP neighborhood is P-complete. We give two logspace constructions that map the graph used in the graph-partitioning completeness proof into initial embeddings in the grid and hypercube. Thus it is unlikely that a parallel algorithm exists that can find even a local minimum solution in polylogarithmic time in the worst case. This result puts experimental results reported in the literature into perspective: attempts to construct the exact parallel equivalent of serial simulated-annealing-based heuristics for graph embedding have yielded disappointing parallel speedups.

We have developed a new massively parallel heuristic, which we call the *Mob* heuristic. The heuristic is closely related to both Kernighan-Lin and simulated annealing. The algorithm uses a *mob-selection rule* to swap large sets, *mobs*, of vertices across planes of a grid or hypercube. The mob-selection rule searches for an approximation to the subset that causes the largest improvement in the embedding cost, and is designed to be computed very quickly in parallel.

We evaluated the performance of the *Mob* hypercube and grid-embedding algorithms by conducting an extensive series of experiments on the Connection Machine CM-2. 1-to-1 mappings of graph vertices to network nodes were used to model VLSI placement problems and standard embedding problems; and 16-to-1 mappings were chosen to model bulk parallelism. We conducted experiments on randomly generated graphs of small ($d = 3 \ldots 16$) degrees with up to 500K vertices and 1M edges. On random graphs, the parallel complexity of the *Mob* heuristic on the CM-2 was found to be time $O(\log |E|)$ with $2|E|$ processors. The absolute speed at which an embedding is produced shows that *Mob* can be implemented very efficiently on a SIMD-style machine. Due to excessive run times, previous heuristics reported in the literature were able to construct graph embeddings only for graphs that were 100 to 1000 times smaller than those used in our experiments.

In section 2 we describe local-search heuristics and give an overview of graph-embedding heuristics. In section 3 we give our P-completeness results for local search heuristics. We describe the Connection machine CM-2, define the *Mob* heuristic, and address special implementation issues on the CM-2 in section 4. Our experimental results for embedding random and random geometric graph onto the hypercube and the grid are given in section 5. In section 6 we give our conclusions and outline future research.

2 Graph Embeddings and Local Search

The graph embedding problem is very difficult to solve even in restricted cases, which justifies the study of heuristics or approximation algorithms. It can be shown by simple reductions that the problem of finding a minimum-cost graph embedding is at least as hard as the GRAPH-ISOMORPHISM, SUBGRAPH-ISOMORPHISM, HAMILTONIAN CIRCUIT, and CLIQUE problems. GRAPH-ISOMORPHISM is in NP but is not known to be NP-complete [14,20]. The SUBGRAPH-ISOMORPHISM, HAMILTONIAN CIRCUIT, and CLIQUE problems are NP-complete [20]. Restricted cases of the graph-embedding problem remain hard to solve: Wagner and Corneil [53] have shown that determining whether an arbitrary tree is a subgraph of a hypercube is NP-complete, and Afrati *et al.* [2] have shown that determining whether a graph is

a subgraph of a hypercube is NP-complete. Graph partitioning is NP-complete [20].

2.1 Constructive Heuristics

A *constructive heuristic* exploits structure and regularity present in a graph G_1 and a network G_2 to construct a good embedding of G_1 into G_2. Constructive heuristics work well in restricted cases and lend themselves to the derivation of bounds on the runtime and the quality of the approximate solution.

Algorithms for constructing a tree embedding into a hypercube have been studied by Afrati *et al.* [2], Monien and Sudborough [38], and Wu [55]. Chan [11], Bettayeb *et al.* [4], and Ho and Johnsson [27] show that grids can be embedded into a hypercube with maximum edge length 2.

Sadayappan and Ercal [42] and Sadayappan *et al.* [43] embed grid subgraphs obtained from *Finite Element Modeling (FEM)* into a processor grid by *strip partitioning*, a method that cuts G_1 into narrow strips and preserves locality. Sadayappan and Ercal [43] use the same technique to embed the FEM graphs into hypercubes, since a hypercube contains a grid as a subgraph. The authors also give a clustering algorithm for embedding arbitrary graphs into hypercubes. Fukunaga *et al.* [19] use a *force-directed-relaxation* approach for the FEM problem.

2.2 Local Search Heuristics

In *local search algorithms*, an initial solution is constructed, usually by some random procedure, and the cost function f is computed. Changes are made to the current solution, and the new solution, which we say is in the *neighborhood* of the current solution, replaces the current solution. The improvement in the cost function f obtained from changing from solution S_1 to solution S_2 is given by the *gain function* $\Delta f := f(S_1) - f(S_2)$. A positive gain change is one that decreases cost.

The procedure that generates changes to the current solution defines the neighborhood. Two neighborhoods that are commonly used by graph embedding heuristics are SWAP and 1-MOVE. In the definitions we give here, both neighborhoods do not determine the order in which steps are taken; they only require that an improving swap be made, not necessarily the best improving swap.

Definition 5 The SWAP neighborhood SWAP-N(S_0) of an embedding S_0 is the set of embeddings S_i obtained by swapping the embedding of two vertices.

Definition 6 The 1-MOVE neighborhood 1-MOVE-N(S_0) of an embedding S_0 is the set of embeddings S_i obtained by changing the embedding of exactly one vertex.

A local search algorithm based on the 1-MOVE neighborhood operates similar to algorithms based on the SWAP neighborhood, except that a balance condition is added so that the sets created become neither too large nor too small. In practice, vertex moves in the 1-MOVE neighborhood can be simulated by vertex swaps in the SWAP neighborhood. This is done by adding "solitary vertices", vertices with no edges attached to them, to the embedded graph. A move of a vertex v is then equivalent to swapping v and a solitary vertex w. These solitary vertices can serve to balance the graph embedding so that every node contains the same number of embedded vertices. Classical local search heuristics for graph embedding are steepest descent, the Kernighan-Lin heuristic, and simulated annealing.

Steepest Descent The *steepest descent* heuristic selects the pair of vertices that gives the largest decrease in cost if swapped. If this decrease is positive, the vertices are swapped, otherwise the heuristic halts. The steepest descent heuristic tends to become trapped in bad local minima, and is not widely used since better local search heuristics are available.

Kernighan-Lin The *Kernighan-Lin neighborhood* of a given embedding consists of the sequence of embeddings obtained by selecting one previously unselected vertex pair that gives the largest decrease (or smallest increase if no decrease is possible) in cost if swapped. The Kernighan-Lin heuristic searches the neighborhood of an embedding and if it finds a sequence of vertex swaps that gives a smaller cost, replaces the current embedding by the new one and continues. If not, it halts. The Kernighan-Lin neighborhood is larger than the SWAP neighborhood obtained by simply swapping two vertices which accounts for the high quality of the results obtained with this heuristic. The Kernighan-Lin heuristic [29] and the extensions by Fiduccia-Mattheyses [18] were initially developed for graph partitioning.

Simulated Annealing *Simulated annealing (SA)* for graph embedding selects one vertex pair at random and swaps them if this reduces the cost. If not, the swap is accepted with a probability that decreases (usually exponentially) with the size of the increase in the cost. This probability function also decreases with a "temperature" parameter of the annealing process which decreases as time elapses. The high-quality results given by this algorithm are explained by observing that it appears to find the region containing a global minimum quickly and then homes in on the minimum as the temperature decreases.

The simulated annealing algorithm was introduced by Kirkpatrick *et al.*[30] and is based on the work in statistical mechanics by Metropolis *et al.* [36]. The first applications of SA were to variations of the grid-embedding problem problem occurring

in VLSI design. Simulated annealing has been established as the method of choice for many optimization problems whenever very good solutions are desired.

Convergence of SA Geman and Geman [21], Lundy and Mees [35], and Mitra *et al.* [37] have shown by Markov chain analysis that SA with a logarithmic cooling schedule will converge to the optimal solution in time $O(a^n), a > 1$. Worst case examples can be constructed which put a lower bound of $\Omega(a^n)$ on SA's running time. SA can actually run slower than the naive algorithm that simply searches the whole solution space.

The result by Geman and Geman was initially derived for an application of SA to image processing, whereas the other two papers address the general case. For the analysis of Lundy and Mees and Mitra *et al.* to hold, certain weak conditions must be met, namely:

(a) The initial solution can be generated in polynomial time.

(b) The transformation of one solution into a neighboring solution can be done in polynomial time.

(c) A polynomial number of transformations suffices to transform one solution into any other solution.

(d) The cost function f and the gain function Δf must be computable in polynomial time.

There is overwhelming empirical evidence that SA provides excellent solutions for most problem instances in polynomial time, but no theoretical work exists to back up these results. Often the above cited results showing convergence in $O(a^n)$ time are invoked to justify convergence in polynomial time, although no such proofs are known. More research needs to be done to explain SA's convergence behavior with fast cooling schedules. An encouraging result is Sorkin's showing that SA with a fast cooling schedule will converge on a fractal landscape [51]. He conjectures that the neighborhood structure of VLSI placement problems may have fractal properties.

2.3 Application of Local Search to Graph Embedding

For the simpler graph partitioning problem, both Kernighan-Lin and simulated annealing give excellent results on serial machines for random graphs of small degree and moderate size [28,34]. The time for one run of KL from an initial partition is much smaller than that for SA. However, KL generally gives bisection widths that are about 10% larger than those provided by SA. If the running times of both are

equalized by making many more runs of KL with randomly chosen initial partitions and choosing the best results of all runs, Johnson *et al.*[28] show that KL gives a best bisection width typically within a few percent of SA's best bisection width, although SA continues to provide better results.

Bokari [6] gave a local search algorithm to embed grid subgraphs obtained from Finite Element Modeling into a processor grid augmented by communication links to diagonal processors that tries to minimize the number of graph edges not mapped to network edges. The algorithm is based on steepest descent with random perturbations; it exchanges the best of $n \times n$ vertex pairs at every iteration and thus has a total running time of $O(n^3)$. The algorithm was tested on 20 graphs with 9 to 49 vertices which were mapped onto processor meshes of size 4×4 to 7×7 and produced solutions of good quality.

Bollinger and Midkiff [7] used simulated annealing to map trees into hypercubes and hypercubes onto themselves. The size of the hypercubes ranges from 8-512 nodes, and the authors report that SA performs very well, and was able to find optimal mappings for hypercubes of size 128 or less.

Chen and Stallmann [13], and Chen, Stallmann, and Gehringer [12] give a survey of hypercube embedding algorithms. Reports are given on experiments to embed into a 128 node hypercube various types of graphs, such as random graphs, geometrical random graphs, trees, hypercubes and hypercubes with randomly selected missing edges. Algorithms for which running times and performance are reported are SA, SA restricted to moves along hypercube axis, Kernighan-Lin, steepest descent, and constructive heuristics. The authors report that the restricted version of SA consistently found the best solutions for all types of graphs, and was thus the most versatile heuristic. SA and the more restricted version of SA were also the most time-consuming heuristics. The time versus quality trade-off still favored restricted SA on random graphs, where the heuristic performed at its best. On more regular graphs a simple constructive heuristic might be a better choice, at least as a preprocessing step to generate a good initial solution which can then be further improved by a local search heuristic.

2.4 Parallel Simulated Annealing

Since graph embedding requires considerable computational resources, recent work has addressed parallel methods. Most of this work has focused on parallel simulated annealing. Greening [25] gives a survey of parallel simulated annealing techniques, and classifies parallel local search algorithms into two categories: *perfect convergence* algorithms, that emulate the execution of a serial algorithm, and *altered convergence* algorithms, that allow for errors when accepting moves.

The simplicity of the SA algorithm seems to make an implementation on a massively parallel computer almost trivial. It is therefore surprising that parallelizing SA by straightforward or elaborate means has eluded researchers, and is in fact, as we will show later, not possible in the worst case.

A parallel local search algorithm for graph embedding will compute cost and gain functions once per iteration, independently of how many vertices are selected for moves or swaps. It is therefore desirable for computational efficiency to move a set of vertices on each iteration. Every vertex or vertex pair can decide whether to move or not based on its gain function Δf. However, the sum of the gain functions is only an approximation to the quality of a resulting embedding. The gains do not take fully into account that other vertices are moving at the same time. To summarize, parallel moves introduce errors in the cost and gain functions, and thus can produce results not produced by the serial local search heuristic.

Perfect Convergence Algorithms Perfect convergence algorithms attempt to emulate the execution of a serial algorithm, usually by trying to make exact the parallel computations of gain functions. A parallel local search algorithm that behaves differently from its serial equivalent cannot rely on experimental work showing good convergence in polynomial time with one processor. The proofs of SA's convergence (with the SWAP neighborhood) to an optimal solution in time $O(a^n)$, a somewhat inconclusive result at best, do not hold either. Note that randomness of serial algorithms is no inherent barrier to their exact parallelization because the output of a random number generator can be replaced by a collection of random numbers accessed in parallel. We will show below that several local-search algorithms for graph embedding are P-complete or P-hard (these terms are defined in section 3) and therefore unlikely to run in polylogarithmic time. Consequently, any attempt to implement a perfect convergence algorithm introduces massive serial bottlenecks.

Kravitz and Rutenbar [31] propose a parallel SA algorithm for VLSI placement. Among all parallel moves, only the *serializable subset*, i.e. all non-interacting moves, are accepted. Unfortunately the serializeable sets tend to be rather small and do not increase with the number of processors. Also, the method prefers to accept certain moves and thus introduces artifacts which degrade convergence behavior.

Roussel-Ragout and Dreyfus [41] 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. This technique allows the authors to show the equivalence to serial SA, and thus the convergence in time $O(a^n)$, but the degree of parallelism is very small. However, the technique results in a serial bottleneck at high temperature, when a large fraction of proposed moves

could be accepted, but the master processor can pick only one move and then has to update all data structures.

Chamberlain *et al.* [10] use a dynamically balanced tree to find the correct sequence of moves accepted by simulated annealing. The authors are able to obtain a logarithmic speedup with this method, and state that an improved balancing algorithm will achieve linear speedup. Neither proof nor experimental evidence is given to support this claim.

Altered Convergence Algorithms Altered convergence algorithms propose and accept a change in the solution without correcting for the influence of other changes that are occurring simultaneously. In some cases the parallel local search algorithms perform better than their serial counterparts since their criteria for accepting moves have been altered. This has led to claims of superlinear speedups in the literature. Since it is always possible to efficiently simulate many parallel processors by one processor, these superlinear speedups vanish once the alterations are incorporated in the serial algorithm.

Parallel simulated annealing heuristics for VLSI placement have been proposed by Casotto and Sangiovanni-Vincentelli [9], Wong and Fiebrich [54] on the Connection Machine, by Darema *et al.* [16] on an IBM 3081 multiprocessor simulating virtual processors, and by Banerjee *et al.* [3] on an Intel Hypercube. Experiments done by these researchers indicate that that no large benefits in solution quality are derived from perfect convergence algorithms, and that the gain-correcting procedures consume vast amounts of time.

Parallel simulated annealing has been applied by Dahl [15] to reduce communication costs on the hypercube. Costs are computed in the HCUBE metric, and vertex swaps are restricted to hypercube edges. For each vertex, the move generation scheme chooses a neighboring hypercube node at random. The change in the cost function is computed, and the move is either accepted or rejected by the simulated annealing algorithm. Since single vertex moves cause unbalanced embeddings, a vertex move is only executed when two neighboring nodes want to swap positions. This approach works well for mappings of $G_1 = (V_1, E_1)$ into $G_2 = (V_2, E_2)$, where $|V_1| = |V_2| \log |V_1|$. The balancing requirement serializes the algorithm for 1-to-1 mappings ($|V_1| = |V_2|$), and for many-to-1 mappings ($|V_1| \gg |V_2|$), for instance graph partitioning ($|V_2| = 2$). A package incorporating the parallel simulated annealing algorithm for hypercube embedding has been implemented on the Connection Machine 2.

2.5 The *Mob* Heuristic for Graph Partitioning

In previous papers [45,46,47], we developed a parallel heuristic for graph partitioning, called the *Mob* heuristic. The heuristic has some of the features of the KL and SA heuristics but is better at exploiting available parallelism. *Mob* deterministically swaps large numbers of elements (mobs) between the two sides of the partition but uses randomization to select which of a large number of equally good candidates will be swapped. The running time for one iteration (one mob swap) is $O(\log m)$ with $2m$ processors, where m is the number of edges. We determined empirically that for random graphs the number of iterations required to obtain a very good solution is around 2000-4000 and independent of graph size. On the 32K-processor CM-2 Connection Machine the *Mob* heuristic was able to find graph partitions for random graphs with more than 2 million edges and 1 million vertices. We compared the performance of *Mob* against that of KL and SA on small random graphs with up to 1,000 vertices, the largest graphs for which we have independent data [28] and where a comparison with serial heuristics was feasible, and found that our *Mob* graph-partitioning heuristic gave better solutions than KL and solutions comparable to SA. In this paper we will apply a modified version of the *Mob* heuristic to graph-embedding problems.

3 The Parallel Complexity of Graph Embeddings

We have introduced the *Mob* heuristic because local search heuristics for graph embedding are expected to be hard to parallelize. For the graph partitioning problem, we have shown in [45,46,47] that local search under the Kernighan-Lin neighborhood is P-complete. In [45,46,47] and independently by Yannakakis and Schäffer [50] it is shown that local search under the SWAP neighborhood for graph partitioning is P-complete, and local search under the SWAP neighborhood is P-hard if the local search algorithm is randomized.

We begin with definitions of P-complete and P-hard problems, the randomized decision class BPP, and the boolean circuit value problem CVP, which is the canonical P-complete problem.

3.1 Parallel Complexity

Definition 7 A *decision problem* is a subset of $\{0,1\}^*$. A decision problem A is in P, the class of polynomial time problems if there is a a deterministic polynomial-time Turing machine T such that if $x \in A$ then T accepts x and if $x \notin A$ then T rejects x.

Definition 8 A decision problem A is *logspace-reducible* to a problem B if there is a function $g : \{0,1\}^* \rightarrow \{0,1\}^*$ computable in logarithmic space (logspace) by a deterministic Turing machine such that $x \in A$ if and only if $g(x) \in B$.

Definition 9 A decision problem is *P-complete* if it is in P and every problem in P is logspace-reducible to it. A decision problem is *P-hard* if every problem in P is logspace-reducible to it.

Definition 10 A decision problem is in *NC* if it is solvable by a uniform class of circuits with polynomial size and polylogarithmic depth. A class of circuits is uniform if there is a Turing machine that for each integer n generates the nth circuit in polynomial time.

If a P-complete problem is in NC, then P is contained in NC [32], a highly unlikely result. Problems in NC can be solved on a parallel machine with polynomially many processors in polylogarithmic time. Since logspace-reducibility is transitive, if a problem A is P-complete and we can find a logspace reduction of it to another problem B in P, then B is also P-complete. The definition of P-hardness does not require that the decision problem be in P. A P-hard problem is at least as hard to solve as a P-complete problem.

A "randomized Turing machine" RT [22] is a machine with an input tape and a read-once, one-way binary "guess tape." If a transition for a given state and input-tape symbol is probabilistic, RT will read a string of k binary symbols from the guess tape, treat it as an approximation to a real number in the interval (0,1) and use it to select a transition. The probability that a string x of length n is accepted (rejected) by a randomized Turing machine making $f(n)$ transitions is the fraction of strings of length $f(n)$ that could be put on the guess tape that causes x to be accepted (rejected). PP and BPP are classes of languages defined below recognized by polynomial-time RTs.

Definition 11 A decision problem $A \in$ PP if there is a a randomized polynomial-time Turing machine RT such that if $x \in A$ then $Pr(RT\ accepts\ x) > 1/2$ and if $x \notin A$ then $Pr(RT\ rejects\ x) > 1/2$.

The class PP is not very useful for our purposes. It can be shown that $NP \subseteq PP$ [22]. We are interested in a higher degree of confidence in the outcome of the RT, that characterized by the class of polynomial-time RTs with *bounded error probability* defined below.

Definition 12 A decision problem $A \in$ BPP if there is a a randomized polynomial-time Turing machine RT and an $\epsilon > 0$ such that if $x \in A$ then $Pr(RT\ accepts\ x) > (1/2 + \epsilon)$ and if $x \notin A$ then $Pr(RT\ rejects\ x) > (1/2 + \epsilon)$.

It is known that $P \subseteq \begin{matrix} BPP \\ NP \end{matrix} \subseteq PP$ [22]. Every language in BPP can be recognized by a polynomial-time randomized turing machine RT with probability arbitrarily close to 1 by repeating the computation and taking the majority of the outcomes as the result. It can be seen that this construction decreases the error probability exponentially with the number of repetitions. It is unknown whether there are problems logspace-complete for BPP.

3.2 Circuits

A *Boolean circuit* is described by a straight-line program over a set of Boolean operations [44]. One output generated by the circuit is identified as significant and is called the "value" of the circuit. The free variables are called "inputs" and are assigned values. A circuit corresponds to a directed acyclic graph (DAG) with Boolean function labels on internal vertices.

Definition 13 The *circuit value problem* (CVP) is the problem of computing the value of a Boolean circuit from a description of the circuit and values for its inputs.

Theorem 1 *CVP is P-complete [32].*

Restricted versions of CVP are also P-complete. A *monotone circuit* uses only the operations AND and OR. The *monotone circuit value problem* (MCVP) is P-complete [23]. The *planar circuit value problem* is also P-complete [23], but the *planar monotone circuit value problem* is in NC [24]. The *fan-in/fan-out 2 circuit value problem* is P-complete, since any circuit can be translated in logspace into a circuit where both fan-in and fan-out of internal vertices are at most 2.

The *dual-rail circuit* DR(C) of a monotone circuit C consists of two subcircuits, C and $\overline{C}$, containing only ANDs and ORs. The subcircuit $\overline{C}$ is obtained by replacing each AND in C with an OR and vice versa, and complementing the inputs. A very convenient feature of the dual-rail circuit DR(C) is that exactly half of the inputs and half of the circuit elements in DR(C) have value 1. It follows from the above construction that the *dual-rail monotone circuit value problem* (DRMCVP) is P-complete.

SA and *Mob* evaluate functions that map partitions onto partitions. To convert these functions to decision problems it is sufficient to mark one node in the graph and to accept partitions in which the marked node changes sets under the heuristic mapping. When we speak of the P-completeness or P-hardness of KL, SA and *Mob* we refer to the decision-based versions of these problems.

3.3 P-Complete Graph Embedding Problems

We now demonstrate that traditional graph-embedding heuristics for embedding graphs into the two-dimensional grid and the hybercube are P-hard. This is done by reducing them to P-hard problems for graph partitioning under the SWAP neighborhood and the $CROSS$ cost function.

Theorem 2 *Finding a locally minimum graph partition under the SWAP neighborhood is P-hard.*

We give the proof of this theorem in [45,46,47]. The proof can be summarized as follows: A graph G is constructed from the dual-rail circuit DR(C) of a monotone circuit C. Computing the value of these circuits is a P-complete problem. The graph G contains a graph corresponding to DR(C) as well as auxiliary subgraphs. A partition $P_0 = (X_0, Y_0)$ of G into equal-sized sets is given as a starting point. By applying improving swaps to P_0 until a local minimum is reached, the value of the circuit DR(C) and the circuit C can be computed directly from the resulting partition.

The proof of Theorem 2 holds for any heuristic H that will only accept vertex swaps with positive gain. H can be a deterministic heuristic such as Steepest Descent, which selects the vertex pair with the highest positive gain for swapping, and stops if no more positive-gain swaps are possible. To make Steepest Descent completely deterministic, a rule has to be provided to arbitrate among swaps with the same gain. Theorem 2 also holds for randomized heuristics H such as randomized Steepest Descent and *zero-temperature simulated annealing* (ZT-SA). Randomized Steepest Descent has a randomized rule to select the vertex swap of highest positive gain among swaps that have equal gain. Thus the heuristic has to run on a randomized turing machine RT, but the probability that RT accepts or rejects its input string in polynomial time is always 1. In SA the mechanism that generates swaps is randomized. ZT-SA, the zero-temperature version of SA, accepts only swaps of positive gain. It can be shown that ZT-SA is in BPP [45,46,47], so while ZT-SA is not a polynomial-time heuristic, the probability that the RT running ZT-SA will accept or reject its input string in polynomial time can be make arbitrarily small.

The following lemma is very useful in the proofs below since it provides a connection between previous results showing that graph partitioning under the SWAP neighborhood is P-hard and local search algorithms discussed in this paper.

Lemma 1 *REDUCTION LEMMA. Let* H *be a local search algorithm for graph embedding which uses the SWAP neighborhood.* H *is P-hard if every swap accepted by* H *corresponds to a positive gain swap in the local search algorithm for the graph-partitioning problem based on the SWAP neighborhood.*

The proof of this lemma follows immediately from Theorem 2.

There are certain graph-embedding problems whose solution can be computed in NC. For example, an n-vertex graph can be partitioned into $n/2$ sets under the CROSS cost function in NC, since this corresponds to computing a maximum matching, which has been shown to be no harder than matrix inversion by Mulmuley *et al.* [39]. Unfortunately, known randomized and deterministic algorithms for maximum matching use matrix inversion as a subroutine, which makes these algorithms impractical for large graphs. The local search heuristics are interesting possible alternatives for these problems.

3.4 Grid Embeddings

We now consider a *family* of local search algorithms for the two-dimensional grid in which the PERIMETER cost function is used.

The PERIMETER cost function is an approximation to wiring cost and is widely used in practice. It should be noted that this cost function does not measure wire congestion in the spaces between objects but that reduction of wire lengths reduces congestion as a secondary effect.

Theorem 3 *Let H be a local search algorithm for grid embedding using the PERIMETER cost function and the SWAP neighborhood. Let the grid embedding be constrained to allow only one graph vertex per network node: $S : V \mapsto \{0..k-1\} \times \{0..l-1\}$ with $S(i) \neq S(j)$ when $i \neq j$. If H accepts only vertex swaps that improve the cost, then H is P-hard.*

Proof We give a construction to convert the graph G used in the graph-partitioning proof in [45,46,47] into a graph G' used here. Let (X_0, Y_0) be the initial partition of G. Every vertex in G has degree at most $K = 9$. The construction can be done in logspace and gives an initial embedding of G' in the grid, as shown in Figure 3, in which every positive-gain swap accepted by H corresponds to a positive gain swap in the associated graph-partitioning problem. Thus, by the reduction lemma, H is P-hard.

The construction consists of a grid with two vertical lines L_0, L_1 which have unit horizontal separation. The line L_0 represents a logical value of 0 while the other line represents a logical value of 1. Along two lines L_0 and L_1 the vertices of G are placed. The vertices in the set X_0 of G are placed on L_0 as shown in Figure 3 and above them is an equal number of solitary vertices. The vertices in Y_0 are placed on L_1 above an equal number of solitary vertices. These vertices on the vertical lines L_0 and L_1 are spaced D vertical grid points apart, where $D = K + 1$, to leave enough room for the

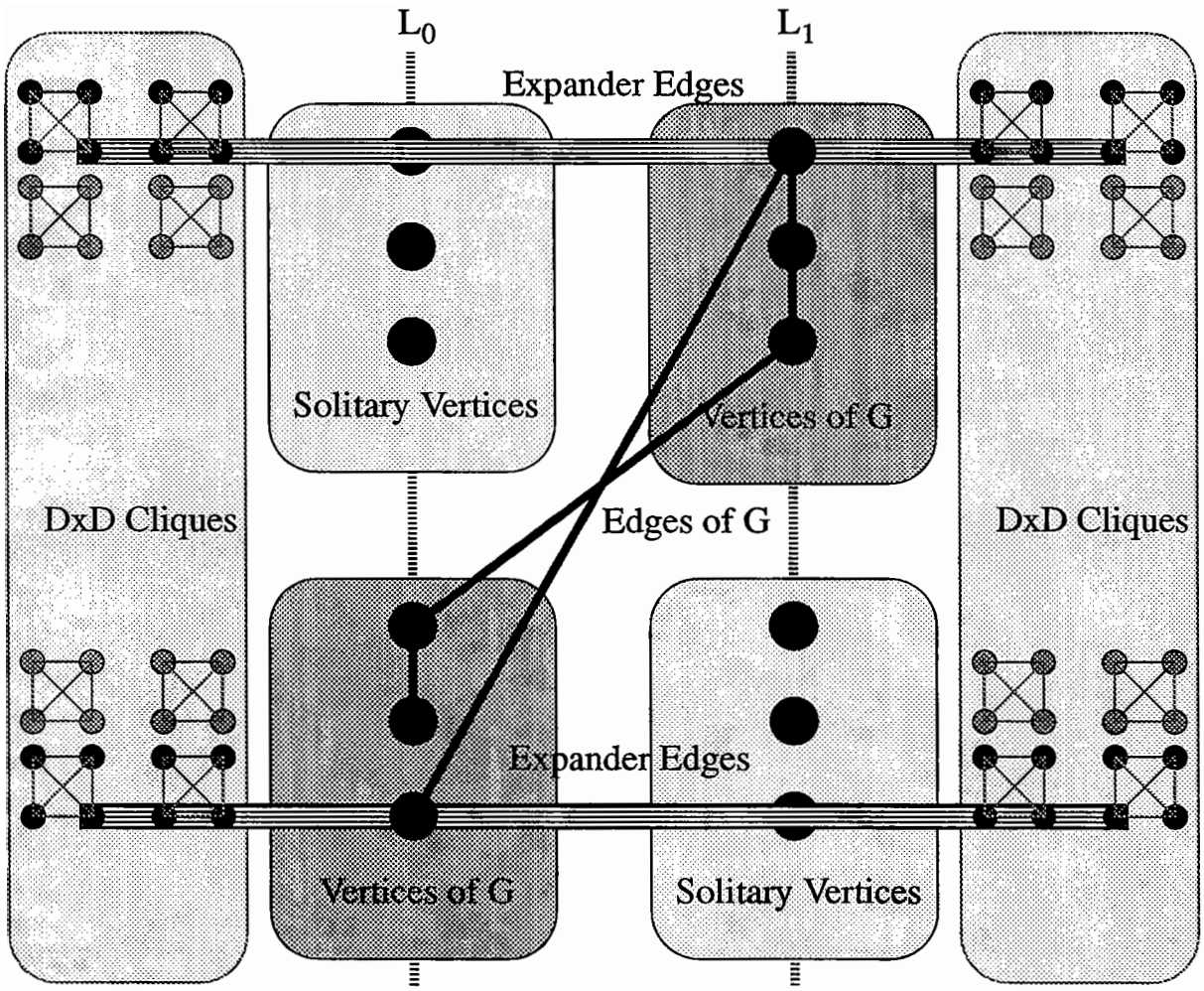

Figure 3: A grid embedding corresponding to a graph-partitioning problem.

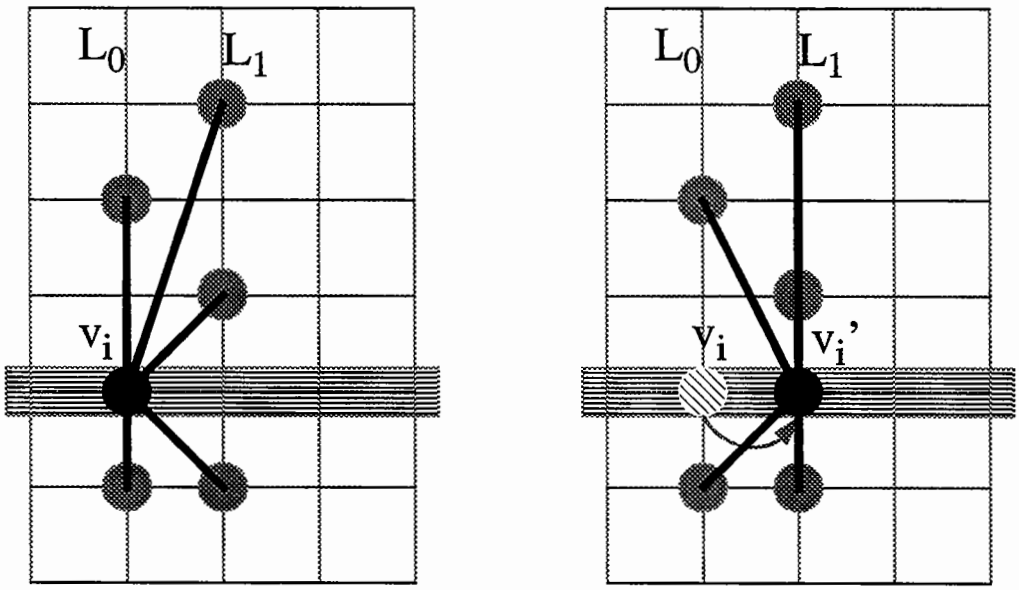

Figure 4: a) Vertex v_0 before swap. b) Regular swap with positive gain.

rest of the construction. (We will show later that the only swaps accepted by H are between a vertex of G and the opposing solitary vertex.)

To limit the movement of a vertex v in G, we place L $(L = 2K + 2)$ $D \times D$ *anchor cliques* to the left and another L anchor cliques to the right of the two vertical lines, and we connect the closest vertex of every clique to v with an *expander edge*, of which there are $2L$. The solitary vertices are not anchored, and can thus move freely. We shall show that the expander edges and cliques constrain v to move along the x-axis between the two vertical lines.

The above construction can be done in logspace, and leaves us with a graph G' embedded on the grid. In the following, we shall compute the change in the cost function, or gain, caused by a vertex swap:

$$gain = cost(old) - cost(new)$$

The PERIMETER cost function allows us to decompose costs and gains into x and y components;

$$gain = gain_x + gain_y$$

We now have to show that all cost-improving swaps executed on G' correspond to cost improving swaps in the associated graph-partitioning problem.

We first examine swaps involving anchor-clique vertices and show that such swaps have negative gain. All other possible swaps involve vertices of G. We examine what

we call *regular swaps*, where a vertex v is exchanged with the opposing solitary vertex. We then examine *irregular swaps*, where v moves to some other grid location. Our induction hypothesis is that regular swaps will only have positive gain when they can be interpreted as positive-gain graph-partitioning swaps, and irregular swaps have only negative gain.

Swaps with Anchor-Clique Vertices Vertex swaps inside $D \times D$ anchor cliques have gain 0 and are not accepted. A swap involving a vertex in a $D \times D$-clique and an outside vertex will have negative gain. A clique vertex has degree at least $D \times D - 1 = (K+1)^2 - 1$, whereas all other vertices have degree at most $K + 2L = 5K + 4$, and at most one edge in common with a clique vertex. Consequently, the swaps of two vertices from different cliques or a vertex in a clique with a non-clique vertex both have large negative gain.

Regular Swaps Figure 4 shows a swap in which vertex v_i at (x_i, y_i) is swapped with a solitary vertex v_i' on line L_0 to $(x_i + 1, y_i)$. A symmetrical argument holds when vertex v_i at (x_i, y_i) is swapped with a solitary vertex v_i' on line L_1 to $(x_i - 1, y_i)$.

The movement of the solitary vertex has no effect on the cost function. Since the vertex v_i only moves along the x dimension, we have

$$gain_y = 0$$

Let $d(v_i)$ be the number of graph vertices to which v_i is attached. By induction, these vertices are placed on the two vertical lines. Let $d_0(v_i)$ be the number of attached vertices in set L_0 and let $d_1(v_i)$ be the number of attached vertices in set L_1. The gain of a regular vertex swap is

$$gain = d_1(v_i) - d_0(v_i)$$

Assume a positive-gain swap in the associated graph-partitioning problem. We have $d_1(v_i) > d_0(v_i)$, so $gain > 0$, and the swap will be accepted. Assume a negative-gain swap in the associated graph-partitioning problem. We have $d_1(v_i) < d_0(v_i)$, so $gain < 0$, and the swap is not accepted.

Irregular Swaps It is of course possible that the grid-embedding heuristic considers a swap of v_i to an undesired location. Figure 5 (a) and Figure 5 (b) show how v_i moves to a irregular position. We now demonstrate that such a swap can only have negative gain.

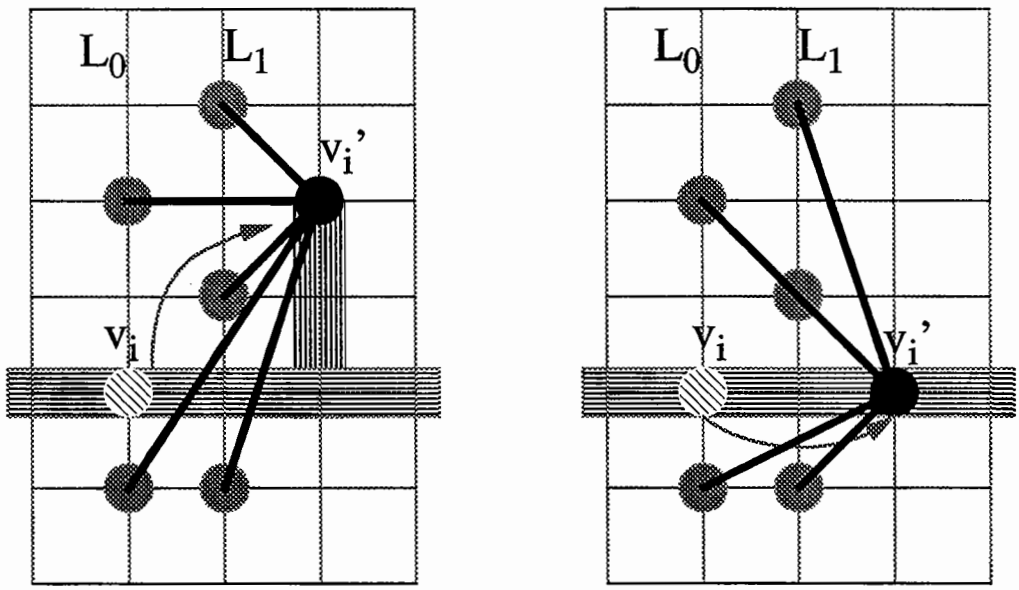

Figure 5: a) Irregular swap, $\Delta y \neq 0$. b) Irregular swap, $\Delta y = 0, \Delta x > 1$.

Case (a) The vertex v_i moves from (x_i, y_i) on line L_0 to $(x_i + \Delta x, y_i + \Delta y)$ and we assume $\Delta y \geq 1$, as shown in Figure 5(a). A symmetrical argument holds when vertex v_i moves from (x_i, y_i) on line L_1 to $(x_i + \Delta x, y_i + \Delta y)$, $\Delta y \leq -1$.

We have

$$gain = gain_x + gain_y$$

We compute the gain along the x dimension. Vertex v_i is attached to $d(v_i)$ vertices. By induction, the vertices of G are on the two vertical lines. Therefore, for each edge the initial edge length in the x dimension is either 0 or 1:

$$\begin{aligned} gain_x &= cost_x(old) - cost_x(new) \\ &\leq cost_x(old) \\ &= \sum_{j=1..d(v_i)} |x_j - x_i| \\ &\leq d(v_i) \end{aligned}$$

The gain along the y dimension depends both on the the attached vertices in G and on the L expander edges attached to their cliques. Then,

$$gain_y = cost_y(old) - cost_y(new)$$

$$= \sum_{j=1..d(v_i)} |y_j - y_i| - [L\Delta y + \sum_{j=1..d(v_i)} |y_j - (y_i + \Delta y)|]$$
$$= \sum_{j=1..d(v_i)} |y_j - y_i| - L\Delta y - \sum_{j=1..d(v_i)} |y_j - y_i - \Delta y|)$$

Since we have $|a| - |b| \leq |a - b|$

$$\begin{aligned} gain_y &\leq \sum_{j=1..d(v_i)} |y_j - y_i| - L|\Delta y| - \sum_{j=1..d(v_i)} |y_j - y_i| - \sum_{j=1..d(v_i)} -|\Delta y| \\ &= -L|\Delta y| + d(v_i)|\Delta y| \\ &= (d(v_i) - L)|\Delta y| \end{aligned}$$

We assumed above that $|\Delta y| \geq 1$. In the following, let $K = \max_{v_i \in G} d(v_i)$ be the maximum degree of G. We shall set the parameter L so that $L = 2K + 2 \geq 2d(v_i) + 2$.

$$\begin{aligned} gain &\leq d(v_i) + (d(v_i) - L)|\Delta y| \\ &\leq (d(v_i) + d(v_i) - L)|\Delta y| \\ &< 0 \end{aligned}$$

Case (b) Vertex v_i moves from (x_i, y_i) on line L_0 to $(x_i + \Delta x, y_i)$, $\Delta x > 1$, as shown in Figure 5(b). A symmetrical argument holds for vertex moves from (x_i, y_i) on line L_1 to $(x_i - \Delta x, y_i)$, $\Delta x > 1$.

As the vertex v_i moves only along the x dimension, we have $gain_y = 0$. By induction, all vertices in G attached to v_i are placed on the two vertical lines. Thus, before v_i moves, the length of all edges attached to v_i in the x dimension is either 0 or 1. After v_i moves to $(x_i + \Delta x, y_i)$, $\Delta x > 1$, every edge that attaches v_i to other vertices in G has length of at least 1. Since every edge length remains the same or gets stretched along the x dimension, this vertex move can only increase the cost. This move will not be accepted by H, as it does not have positive gain.

Case (c) Vertex v_i moves from (x_i, y_i) on line L_0 to $(x_i - \Delta x, y_i)$, $\Delta x > 0$. A symmetrical argument holds for vertex moves from (x_i, y_i) on line L_1 to $(x_i + \Delta x, y_i)$, $\Delta x > 0$.

As the vertex v_i moves only along the x dimension, we have $gain_y = 0$. This vertex move can only increase the cost, since every edge gets stretched along the x dimension. This move will not be accepted by H, as it has negative gain.

To summarize, the conditions of the reduction lemma hold for a regular swap. Other swaps cannot be accepted by H since they have negative gain. No swaps are accepted that violate our inductive assumption that the vertices of G are placed on the two vertical lines. □

3.5 Hypercube Embeddings

Theorem 4 *Let H be a local search algorithm for hypercube embedding using the HCUBE cost function and the SWAP neighborhood. Let the hypercube embedding be constrained to allow only one graph vertex per network node: $S : V \mapsto \{0..2^k - 1\}$ with $S(i) \neq S(j), i \neq j$. If H accepts only vertex swaps that improve the cost, then H is P-hard.*

Proof We give a construction to convert the graph $G = (V_1, E_1)$ used in the graph-partitioning proof in [46,47] into a graph G' used here. Let (X_0, Y_0) be the initial partition of G. Every vertex in G has degree at most $K = 9$. The construction can be done in logspace and gives an initial embedding of G' in the hypercube in which every positive-gain swap accepted by H corresponds to a positive gain swap in the associated graph-partitioning problem. Thus, by the reduction lemma, H is P-hard.

To construct G', let the tuple (a_1, a_2, a_3, a_4) represent the k-bit address of a hypercube node, where the fields a_1, a_2, a_3, a_4 are binary numbers with constant k_1, k_2, k_3, k_4 number of bits, with $k = k_1 + k_2 + k_3 + k_4$. We can associate the individual fields a_1, a_2, a_3, a_4 with collections of hyperplanes. As a shorthand, 0 represents a binary number of arbitrary length where all bits are 0.

Field a_1 has length $k_1 = 1$. Graph vertices of G and their associated solitary vertices are placed on either side of the hyperplane a_1. All addresses $(0, a_2, a_3, a_4)$ denote a logical value of 0 while addresses $(1, a_2, a_3, a_4)$ denote a logical value of 1.

Field a_2 has length $k_2 = \lceil \log |V_1| \rceil$, where $|V_1|$ is the number of vertices in G. If vertex v_i is in the set X_0 of G, v_i is placed at $(0, i, 0, 0)$ and a solitary vertex is placed at $(1, i, 0, 0)$. If vertex v_i is in the set X_1 of G, v_i is placed at $(1, i, 0, 0)$ and a solitary vertex is placed at $(0, i, 0, 0)$. The solitary vertices are not anchored, and can thus move freely.

To limit the movement of a vertex v_i in G, we place L $(L = K + 5)$ *anchor vertices* for every vertex v_i. Field a_3 with length $k_3 = L$ is used for the placement of the anchor vertices. Let e_l be a binary number of length k_3 where the lth bit is 1 and all other bits are 0. *Anchor vertices* are placed at addresses $(0, i, e_1, 0)$, ..., $(0, i, e_L, 0)$ for every vertex v_i. Another set of anchor vertices are placed at addresses $(1, i, e_1, 0)$, ..., $(1, i, e_L, 0)$ for every vertex v_i. Let E_0 be the set of anchor vertices at $(0, i, e_1, 0)$, ..., $(0, i, e_L, 0)$, let E_1 be the set of anchor vertices at $(1, i, e_1, 0)$, ...,$(1, i, e_L, 0)$. Since we do not want the anchor vertex to move itself, every anchor vertex $(0, i, e_j, 0)$ is part of a $D \times D$-clique, with $(D = K + 1)$. We use field a_4 with length $k_4 = \lceil \log D \times D \rceil$ for the cliques, and place the kth vertex of the $D \times D$-clique at $(0, i, e_j, k)$.

We connect each anchor vertex to v_i with an *expander edge*. The purpose of expander edges is to constrain v_i to move only between the planes $(0, i, 0, 0)$ and

$(1, i, 0, 0)$. Whether a vertex v_i is at $(0, i, 0, 0)$ or $(1, i, 0, 0)$, it is connected by expander edges to the set of anchor vertices at $(0, i, e_1, 0)$, ..., $(0, i, e_L, 0)$, and to the set of anchor vertices at $(1, i, e_1, 0)$, ..., $(1, i, e_L, 0)$. One set of expander edges has cost L and the other set has cost $2L$. As we shall see later, the expander edges drastically add to the cost when v_i moves to some other location in the hypercube. The only swaps accepted by H are between a vertex of G and the opposing solitary vertex.

The above construction can be done in logspace, and leaves us with a graph G' embedded on the hypercube. We now have to show that all cost-improving swaps executed on G' correspond to cost improving swaps in the associated graph-partitioning problem. We first examine swaps involving anchor-clique vertices and show that such swaps have negative gain. All other possible swaps involve vertices of G. We examine what we call *regular swaps*, where a vertex v is exchanged with the opposing solitary vertex. We then examine *irregular swaps*, where v moves to some other hypercube location. Our induction hypothesis is that regular swaps will only have positive gain when they can be interpreted as positive-gain graph-partitioning swaps, and irregular swaps have only negative gain.

Swaps with Anchor-Clique Vertices Vertex swaps inside $D \times D$ anchor cliques have gain 0 and are not accepted. A swap involving a vertex in a $D \times D$-clique and an outside vertex will have negative gain. A clique vertex has degree at least $D \times D - 1 = (K+1)^2 - 1$, whereas all other vertices have degree at most $K + 2L = 5K + 4$, and at most one edge in common with a clique vertex. Consequently, the swaps of two vertices from different cliques or a vertex in a clique with a non-clique vertex both have large negative gain.

Regular Swaps Assume v_i moves from $(0, i, 0, 0)$ to $(1, i, 0, 0)$. A symmetrical argument holds when v_i moves from $(1, i, 0, 0)$ to $(0, i, 0, 0)$.

Let $d(v_i)$ be the number of graph vertices to which v_i is attached. By induction, these vertices are placed at addresses $(0, j, 0, 0)$ or $(1, j, 0, 0)$, $j \neq i$. Let $d_0(v_i)$ be the number of attached vertices in L_0, and let $d_1(v_i)$ be the number of attached vertices in L_1. In a regular move, the expander edges do not contribute to the gain, since L edges belonging to the anchor vertices at $(0, i, e_1, 0)$, ..., $(0, i, e_L, 0)$ get stretched by 1 and L edges belonging to the anchor vertices at $(1, i, e_1, 0)$, ..., $(1, i, e_L, 0)$ get shortened by 1. The change in edge length is either 0 or 1,

$$gain = d_1(v_i) - d_0(v_i)$$

Assume a positive-gain swap in the associated graph-partitioning problem. We have $d_1(v_i) > d_0(v_i)$, so $gain > 0$, and the swap will be accepted. Assume a negative-

gain swap in the associated graph-partitioning problem. We have $d_1(v_i) < d_0(v_i)$, so $gain < 0$, and the swap is not accepted.

Irregular Swaps The hypercube-embedding heuristic may of course consider a swap of v_i to an undesired location. We now show that such a swap can only have negative gain, and is thus not accepted by H. Let us assume that v_i moves from $(0, i, 0, 0)$ to $v_i' = (a_1, a_2, a_3, a_4)$. A symmetrical argument holds when v_i moves from $(1, i, 0, 0)$ to (a_1, a_2, a_3, a_4). Let $h = HCUBE(v_i, v_i')$ be the distance that v_i moves, and let $cost(expander(v_i))$ be the cost of the expander edges attached to v_i. The gain of the vertex swap is dependent on the attached edges of the graph G and the expander edges.

$$\begin{aligned} gain &= \sum_{j=1..d(v_i)} HCUBE(v_i, v_j) + cost(expander(v_i)) \\ &\quad -[\sum_{j=1..d(v_i)} HCUBE(v_i', v_j) + cost(expander(v_i'))] \end{aligned}$$

$HCUBE()$ is a metric, so we can use the triangular inequality to get an upper bound on the summation:

$$HCUBE(v_i, v_j) - HCUBE(v_i', v_j) \leq HCUBE(v_i, v_i') = h$$

$$gain \leq d(v_i)h + cost(expander(v_i)) - cost(expander(v_i'))$$

It remains to consider the effect of the move from v_i to v_i' on the expander edges. By induction, v_i is at $(0, i, 0, 0)$. Recall that the anchor vertex set E_0 is located at $(0, i, e_1, 0), \ldots, (0, i, e_L, 0)$, the anchor vertex set E_1 is at $(1, i, e_1, 0), \ldots, (1, i, e_L, 0)$. The address $(0, i, 0, 0)$ and anchor vertices in E_0 have a Hamming distance of 1, and $(0, i, 0, 0)$ and anchor vertices in E_1 have a Hamming distance of 2, so the cost of the expander edges before the move is:

$$cost(expander(v_i)) = L + 2L$$

Let us compute $cost(expander(v_i'))$, the cost of the $2L$ expander edges that connect v_i' at (a_1, a_2, a_3, a_4) to E_0 and E_1. Let b_1, b_2, b_3, b_4 be the number of bits that change

in a_1, a_2, a_3, a_4, respectively. We have $h = b_1 + b_2 + b_3 + b_4$. We now examine how edge costs are affected by a change of bits in the address fields.
(a_1): The contribution of a_1 to the cost of the $2L$ expander edges is L, both when $a_1 = 0$ and when $a_1 = 1$.
(a_2): A change of b_2 bits in a_2 causes a cost increase of b_2 for every edge.
(a_3): Consider the change of b_3 bits in the third field. For every bit j that changes in a_3, exactly one pair of expander edges, namely the pair attached to $(0, i, e_j, 0)$ and $(1, i, e_j, 0)$ will contribute a cost of 0, whereas the $2L - 2$ other expander edges contribute a cost of 1. $L - b_3$ bits do not change in a_3. For every bit j that does not change in a_3, exactly one pair of expander edges, namely the pair attached to $(0, i, e_j, 0)$ and $(1, i, e_j, 0)$ will contribute a cost of 1, whereas all other expander edges contribute a cost of 0. Therefore the contribution of a_3 to the cost of the $2L$ expander edges is $b_3(2L - 2) + (L - b_3)2$.
(a_4): A change of b_4 bits in a_4 causes a cost increase of b_4 for every edge.

The total cost of the expander edges that join v_i' to E_0 and E_1 is given below:

$$\begin{aligned} cost(expander(v_i')) &= L + b_2 2L + b_3(2L - 2) + (L - b_3)2 + b_4 2L \\ &= (L + 2L) + 2L(b_2 + b_3 + b_4) - 4b_3 \end{aligned}$$

In the following, let $K = \max_{v_i \in G} d(v_i)$ be the maximum degree of G. We shall set the parameter L so that $L = K + 5 > d(v_i) + 3$. Given the above equation, we consider first the special case where $h = 1$.

Case ($h = 1$) When $h = 1$, we have $b_1 = 0$, otherwise v_i' would be at $(1, i, 0, 0)$, a regular position. We have $b_2 + b_3 + b_4 = 1$, and $b_3 \leq 1$.

$$cost(expander(v_i')) \geq (L + 2L) + 2L - 4$$

and so

$$gain \leq d(v_i) - 2L + 4$$

Since $L > d(v_i) + 5$, we have $gain < 0$

Case ($h > 1$) When $h > 1$, we have $b_2 + b_3 + b_4 \geq h - 1$, and $b_3 \leq h$.

$$cost(expander(v_i')) \geq (L + 2L) + 2L(h - 1) - 4h$$

and so

$$gain \leq d(v_i)h - 2L(h - 1) + 4h$$

$$\begin{aligned} &\leq d(v_i)h - Lh + 4h \\ &\leq (d(v_i) + 4)h - Lh \end{aligned}$$

Since $L > d(v_i) + 5$, we have $gain < 0$

To summarize, the conditions of the reduction lemma hold for a regular swap. Other swaps cannot be accepted by H since they have negative gain. □

4 *Mob* Heuristics for Graph Embedding

In this section, we describe the Connection Machine CM-2, define the *Mob* heuristic, and address special implementation issues on the Connection Machine.

4.1 The Connection Machine

The Connection Machine 2 (CM-2) is a massively parallel computer consisting of up to 64K moderately slow one-bit processors [1,26]. It is organized as a 12-dimensional hypercube with sixteen nodes at each corner of the hypercube. The CM-2 supports virtual processors which are simulated very efficiently in microcode. A user allocates virtual processors that are then bound to physical processors for execution. The CM-2 usually obtains its peak performance when the ratio of virtual to real processors is more than one. Additionally, 2K pipelined floating-point processors are available for numerical computations.

Each one-bit processor has an associated memory of up to 8K bits, which is shared by passing messages among processors. The CM-2 supports message combining on the hardware level to avoid network congestion. In contrast to shared-memory machines, concurrent writes are much faster than concurrent reads. The CM-2 also permits communication along hypercube and multidimensional grid axes, which is substantially faster than the general router.

The CM-2 is a SIMD machine: each processor executes the same instruction at the same time unless it has been disabled. In practice, the SIMD approach simplifies debugging, permits an elegant programming style, and does not limit the expressiveness of algorithms. The processors have unique IDs, and can switch themselves on or off depending on the outcome of boolean tests. The CM-2 is programmed using parallel variables *(Pvars)* with the same name in each processor. System software supports embeddings of multidimensional Pvars onto the hypercube. The CM-2 has local arithmetic and boolean operations to act on Pvars as well as operations to send and receive messages to grid neighbors or arbitrary locations. Some of these operations set condition bits which determine whether or not a processor is involved in subsequent operations.

Higher-level primitives are provided on the Connection Machine. They include sorting, reduction operations that combine values from each active processor, such as a global OR operation, and the powerful scan operations. A *scan* operation [5], also known as *parallel prefix* operation [33], applies an associative operator $*$ to a linear array and produces a second linear array. If the first array has as $a[i]$ as its ith element and the second has $b[j]$ as its jth element, then a scan of the array a produces the array b of partial products with $b[j] = a[1] * \cdots * a[j]$. With addition as the associative operator, it computes a running sum, whereas with the copy operation it spreads a value into an array of locations. The CM-2 also has *segmented scan* operations in which a series of scans is performed on contiguous regions of a linear array. The scan operation runs in time logarithmic in the number of virtual processors on the CM-2.

Our *Mob* graph-embedding heuristics were implemented in the C language with calls to Paris (Release 6.0), the high-level assembly language of the CM-2.

MasPar has introduced a family of machines with four-bit processors and integrated floating point hardware for which results comparable to those obtained with the CM-2 can be expected. The MasPar machines have two networks, a two-dimensional grid and a three-stage crossbar switch.

4.2 The *Mob* Neighborhood

The *Mob* heuristic for graph partitioning is the basis for our graph embedding heuristics. We report on two new *Mob* heuristics for embedding graphs into the two-dimensional grid and the hypercube.

We now define the *Mob* local search algorithm and its *neighborhood* structure. The algorithm searches a mob neighborhood of an embedding with a given mob size. If the new embedding found has a smaller cost, the search is repeated on the neighborhood of the new embedding with the same mob size. If the cost increases, the neighborhood of the new embedding is searched with the next scheduled mob size. We assume that *Mob* executes a number of steps that does not exceed $q(n)$, a polynomial in the number n of vertices. The probabilistic nature of *Mob* is reflected by sets R1, R2 of $q(n)$ random index variables chosen uniformly and independently from the interval $(0, 1)$. A "schedule" determines the rate at which the variable *mob* decreases. Our heuristic is outlined below.

```
Mob(S)
    Let S                          be the initial embedding;
    Let R1[1..q(n)], R2[1..q(n)]   be random reals in (0,1);
    Let MS[1..q(n)]                be a mob schedule in [1,n/2];
    t = 1; s = 1;
    while( t <= q(n) )
```

```
{
    Q = MOB-NR(S, MS[s], R1[t], R2[t]);
    if( bw(Q) > bw(S) )
        s = s + 1;
    S = Q;
    t = t + 1;
}
return S ;
```

We now define the terms used in the above algorithm. A schedule of mob sizes is used to control the searches:

Definition 14 A *mob schedule* MS of length L is a sequence $[MS_1, MS_2, \ldots, MS_L]$ of integers from the interval $[1, \ldots, n/2]$.

We now define the mob-neighborhood MOB-N(S, m) structure.

Definition 15 The *mob-neighborhood* MOB-N(S, m) of an embedding S is the set of embeddings obtained from S by selecting m pairs of vertices, and swapping these vertices.

The size of MOB-N(S, m) is $\frac{n!}{(n-2m)!m!}$. Clearly, it is not feasible to compute the gains of all embeddings in the neighborhood MOB-N(S, m) of a solution S. If we were to pick an embedding S at random from MOB-N(S, m), we should expect only a very small change in the cost function on average. Therefore, for grid and hypercube embeddings we use the neighborhood MOB-NR(S, m), a restriction of the mob-neighborhood MOB-N(S, m), which uses the mob-selection rule MOB-R given below to look for a rough approximation to the subset that causes the largest improvement in the embedding cost.

Definition 16 The *mob-neighborhood* MOB-NR(S, m) of an embedding S is the set of embeddings obtained from S by selecting m pairs of vertices with the MOB-R rule, and swapping these vertices.

Definition 17 The rule MOB-R(S, m, r_1, r_2) selects a mob of *size* m as follows: Let r_1, r_2 be random real numbers drawn uniformly from the interval $(0, 1)$. For each vertex pair (v, w), let $gain(v, w)$ be the change in the embedding cost if v and w are swapped, and let $gain(v, w)$ be positive if the embedding cost decreases. Let $SX(g) = \{v \in X \mid gain(v, w) \geq g\}$. Choose g_X such that $|SX(g_X + 1)| < m \leq |SX(g_X)|$. Let $m_p = |SX(g_X)|$ be the size of the *pre-mob* $SX(g_X)$. Each element in $SX(g_X)$ is indexed from 0 to $m_p - 1$. A *mob* of size m is selected from $SX(g_X)$ by adding $\lfloor r_1 m_p \rfloor$ to each index modulo m_p and selecting those with index less than m.

MOB-R is designed to be computed very quickly in parallel. On the hypercube, a hypercube axis is chosen at random; vertices can be swapped between hypercube neighbors across the chosen hypercube axis. On the grid, a distance d of $\pm 1, 2, 4, 8, 16, \ldots$ on either the X or Y axis is chosen at random; vertices can be swapped between grid neighbors that are a distance d apart. A gain is computed for every vertex to find the vertex pairs whose exchange would cause the largest individual change in the embedding cost. The rule MOB-R(S, m, r_1, r_2) selects a group of high-gain vertices of size *mob* to move. To avoid sorting by gain, we use an adaptive binary search algorithm to identify such a "pre-mob" of vertices with gain g or larger in each set of the embedding where g is the smallest gain such that at least *mob* vertices have a gain greater than or equal to g. We select *mob* vertices at random from this pre-mob.

We also implemented routines to choose *exactly* m vertices whose individual gains are largest, but neither sorting or randomized methods provided adequate performance. Instead we chose vertices at random among all vertices in $SX(g_X)$. While in so doing we may miss some vertices with large gain, our heuristic is very effective.

4.3 Implementation of *Mob*

We now discuss how the *Mob* heuristic was implemented to make full use of the CM-2's parallel hardware and instruction set. We describe a collection of primitive operations that are used to build both hypercube and grid *Mob* heuristics. These operations are common to any parallel graph-embedding heuristic that uses local search, and can be used to design variations of the *Mob* or SA heuristic. We show how to construct edge and vertex data structures that can exchange vertices and compute cost and gain functions with minimal communication overhead. Cost and gain functions are unique to the hypercube and grid embedding problems and can be easily changed to solve variations of the graph-embedding problem. The move generation routine limits the size of the neighborhood of an embedding in order to use a reasonable number of processors while still allowing the *Mob* heuristic to quickly converge to a good solution. We describe how the mob selection rule Mob-R is implemented without a call to an expensive sorting routine. The advantages of parallel move generation hold also for serial algorithms. The *Mob* heuristic gets maximum use out of cost and gain computations. No queues or bucket data structures would be necessary in a serial implementation.

Move Generation For every embedding, $\binom{|V_1|}{2}$ individual vertex swaps are possible. Clearly, the processor-time cost of proposing every possible vertex swap, computing the gain of the swap, and then using the mob procedure to pick the swaps with the best gain, while ensuring that swaps do not share vertices, would be prohibitive.

We want to generate an $O(|V_1|)$-size subset of all possible vertex swaps. A subset of size $O(|V_1|)$ picked at random will probably miss a very large number of high-gain swaps that are possible. Therefore, on the hypercube, a hypercube axis is chosen at random, and vertices are swapped between hypercube neighbors across the chosen hypercube axis. On the grid, a distance d of $\pm 1, 2, 4, 8, 16, \ldots$ on either the X or Y axis is chosen at random, and vertices are swapped between grid neighbors that are a distance d apart. The effect of this move set is that while a vertex may not be able to move to its optimal position in one step, it can approach this position in a very small number of steps. This move set is geared to the communication bottlenecks of the underlying computing network.

When more than one vertex is embedded on a network node, ordering the vertices by their gain will generate swaps with higher added gains. We have found that this significantly speeds up the convergence behavior of the *Mob* heuristic. We experimented with several move-generating procedures and found that the cost of sorting vertex gains, segmented by the network nodes, is prohibitive on the CM-2. We therefore implemented a compromise where every network node finds the vertex with highest gain and swaps it into the first location of the node segment in the vertex array. We believe that this max-gain shuffle offers an excellent trade-off between computation time and convergence behavior.

Edge Data Structure Our algorithms use a data structure in which each edge is repeated twice, e.g. (a, b) and (b, a). These edges are stored in a linear array and each stores the array address of the other. One virtual processor is used for each edge and the edges are sorted by their left vertex so that groups of vertices with the same left vertex are formed that represent all edges going into one vertex. We let the first edge processor in a group also serve as a vertex processor. This data structure supports the computation of the cost of an embedding as well as the gain of individual vertices. We use scan operations for all but one non-local communication step on the edge data structure. For this we use a Send operation. Blelloch [5] reports that this data structure also works very well for other graph algorithms.

Vertex Data Structure After every vertex has computed its gain, a separate data structure is used to swap vertices. The vertices are stored in a linear array, and the edge data structure maintains link pointers to the vertex processors. The vertex data structure is designed to facilitate rapid vertex exchanges. Vertices embedded on the same network node are adjacent in the vertex array. Filler vertices are added at the initialization stage of the *Mob* heuristic to ensure that every network node has the same number of embedded vertices. Let V_1 be the number of vertices to be embedded, let $|V_2|$ the number of network nodes, and let $k = |V_1|/|V_2|$ be the number of embedded

vertices per node. Every network node can communicate and swap vertices with a network node a distance d away by simply adding kd modulo $|V_2|$ to its own address in the vertex array. Such monotonic communication patterns are asymptotically easier to route than general patterns on most architectures. For instance, the pattern can be implemented by a chain of nearest-neighbor communication operations on the hypercube network of the Connection Machine.

We experimented with an alternative method of generating vertex swaps, which requires only the edge data structure. Each vertex processor, represented by the first edge processor in the edge data structure, computes the address of the network node to which it will move. Computing the address of this target node is a simple arithmetic operation. To form a pair of vertices to be swapped, the vertex at the target node has to be located in the edge data structure. This is an operation not easily supported by the edge data structure. One easy solution is to sort the vertices by their target addresses. We concluded that the cost of transferring information to and from the vertex data structure was an order of magnitude smaller than the cost of sorting node addresses, and gave us a more flexible data structure.

Cost Every edge computes its length. A reduction operation sums the edge costs. Since every edge has a twin, this number is divided by two to obtain the cost of the embedding.

Gain A "gain" is computed for each vertex that measures the effect of its move to a new location on the embedding cost. The gain is computed by summing the contribution of each edge adjacent to the vertex. Every edge computes its length. The gain of an edge is the difference in edge lengths between the old and new vertex positions. We use a segmented additive scan in reverse order in the array to sum up the (adjacent) edge gains into a vertex gain. The segments are the groups of edge processors with the same left component. The gain of a swap is the sum of its two vertex gains.

Selecting A Mob Adaptive binary search is used to form the smallest pre-mob of at least *mob* vertices with the highest gains, where *mob* is provided by a mob schedule. This avoids the cost of a sorting step. An integer randomly chosen from the interval $(0,mob-1)$ is then broadcast to all active processors (the simulated vertex processors), which then add this value to their rank modulo *mob* to compute a new rank. Finally, all active processors with new rank less than *mob* are selected for the mob and a change of position.

Exchange After every vertex has decided whether or not to move, all edge processors become active and the first edge processor in a group (the simulated vertex processors) communicates its new position to all processors in its group by a segmented Copy-Scan. Edge processors then use Sends to their twin edges to notify the right vertex of each edge of their position.

5 Experimental Results

5.1 Random Graphs

We evaluated the performance of the *Mob* hypercube and grid-embedding algorithms by conducting experiments on the CM-2. 1-to-1 mappings of graph vertices to network nodes were used to model VLSI placement problems and standard embedding problems. We also wanted to model bulk-parallelism, where n virtual processors are embedded into a computing network of size $n/logn$, and chose 16-to-1 mappings as a rough approximation of $\log n$-to-1 mappings. The random graphs studied by us are more than 1000 times larger than graphs previously studied in the literature. Serial algorithms such as SA or KL would have prohibitive running times for such large graphs. We measured the solution quality, convergence as a function of time, and running times with a varying number of processors, and compared *Mob* to other algorithms. We show that the *Mob* embedding heuristics produce excellent solutions, converge quickly, run in time $O(\log |E|)$ with $2|E|$ processors, and are well matched to the parallel 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, removing multiple edges and self-loops, to yield a total of $Vd/2$ edges. We did not use the standard approach of generating edges by flipping a coin with the appropriate probability for all $V(V-1)/2$ potential edges in a graph because the number of trials would have been far too large. The graphs are of small average degree d, ranging from $d = 3 \ldots 16$, since this is the kind found in VLSI problems and processor mapping problems.

The exact 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. At least 5 runs were performed for each embedding.

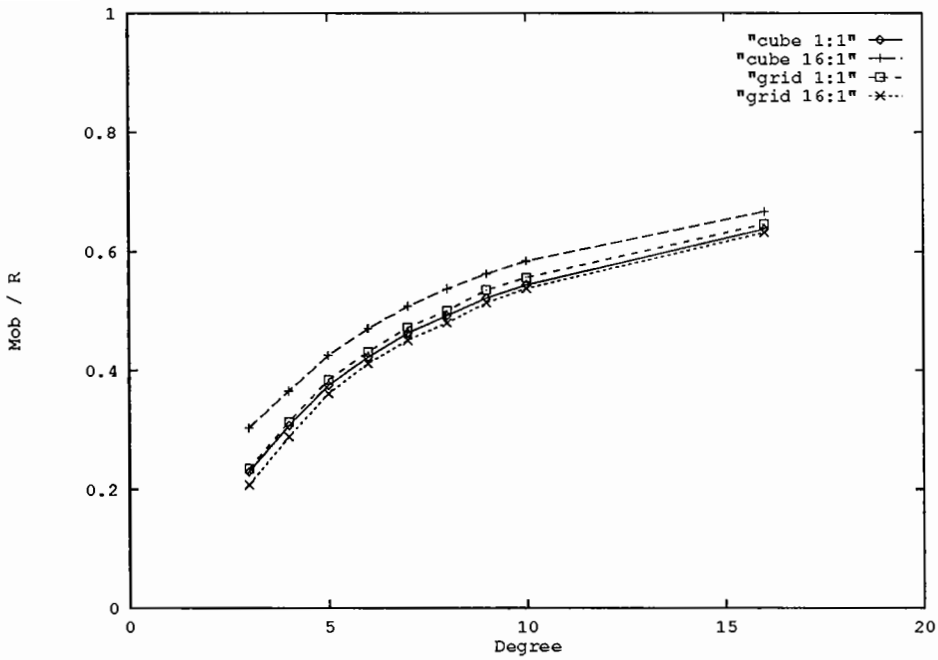

Figure 6: Mob / R embedding cost ratio for *Mob* as a function of degree

Mob Schedule The mob schedule specifies how many element pairs can swap at one iteration, and thus influences how the mob heuristic converges. The mob schedule used for random graphs was constructed as follows: The maximum mob size of the schedule is set to the number of vertex pairs divided by 8. The mob size is then decremented in 16 steps. The process is repeated with the number of steps required to decrement the mob size from the maximum value doubling every time. *Mob* was always stopped after 8000 iterations.

The 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 pairs of maximum gain at each hypercube node, but also moves other vertex pairs. We experimented with various schedules and concluded that this schedule combined initial rapid convergence and very good results at the later stages of the algorithm, when the improvements that can be realized get smaller and smaller. We found that this schedule worked well with 16-to-1 and 1-to-1 mappings, and with both grid and hypercube embeddings.

Solution Quality of *Mob* The quality of the solutions produced by *Mob*is shown in Table 2 and Table 3 for both 1-to-1 and 16-to-1 embeddings. Table 2 gives results

for hypercube embeddings, Table 3 gives results for grid embeddings. The results in these tables are expressed as average edge lengths, and have to be multiplied by the number of edges $\frac{nd}{2}$ to give total embedding costs. The columns labeled D give the dimension of the hypercube that the graph is embedded in. For the grid embeddings, D is the dimension of the hypercube containing a $2^{\lfloor D/2 \rfloor} \times 2^{\lceil D/2 \rceil}$ grid. The columns labeled R give the average edge length produced by a random embedding. The columns labeled *Mob* give the average edge length in an embedding produced by *Mob*. We can see that the *Mob* heuristic offers impressive reductions in embedding costs. The columns labeled Mob/R give the ratio of improvement produced by *Mob* over a random solution.

Our experiments show:

(a) For fixed degree d, Mob/R is largely independent of graph size.

(b) 16-to-1 mappings give slightly better Mob/R ratios than 1-to-1 mappings. If we model communication costs by edge lengths, this shows the advantage of the bulk-parallel model in reducing overall network communication bandwidth.

(c) The grid-embedding *Mob* heuristic achieves lower Mob/R ratios than the hypercube *Mob* heuristic.

(d) The ratio Mob/R rises with increasing average graph degree toward an asymptote of 1, as shown in Figure 6. The differences between grid and hypercube embeddings and between 1-to-1 and 16-to-1 embeddings become smaller with increasing graph degree.

Rates of Convergence of *Mob* Table 2 and Table 3 also report the convergence of *Mob* for an increasing number of iterations. The columns labeled *Iterations* show the average embedding cost as percentages above the best-ever embedding cost, produced after 100, 1000 and 4000 iterations of *Mob*, respectively. Our experiments indicate that the number of iterations needed to reach a fixed percentage above the best-ever embedding cost appears to be approximately constant, and therefore independent of graph size or network size. This important observation holds for hypercube and grid embeddings, and for both 1-to-1 and 16-to-1 mappings. The behavior is surprising since vertices have to travel larger distances, especially in grid networks, to reduce average edge costs by a fixed ratio; and underlines the importance and appropriateness of *Mob*'s set of moves along hypercube axes. The fact that *Mob*'s rate of convergence is independent of graph size was also observed for graph partitioning in [45,46,47].

We would like to stress that these results describing *Mob*'s behavior were obtained for random graphs. In fact, our work on the P-completeness of local search heuristics

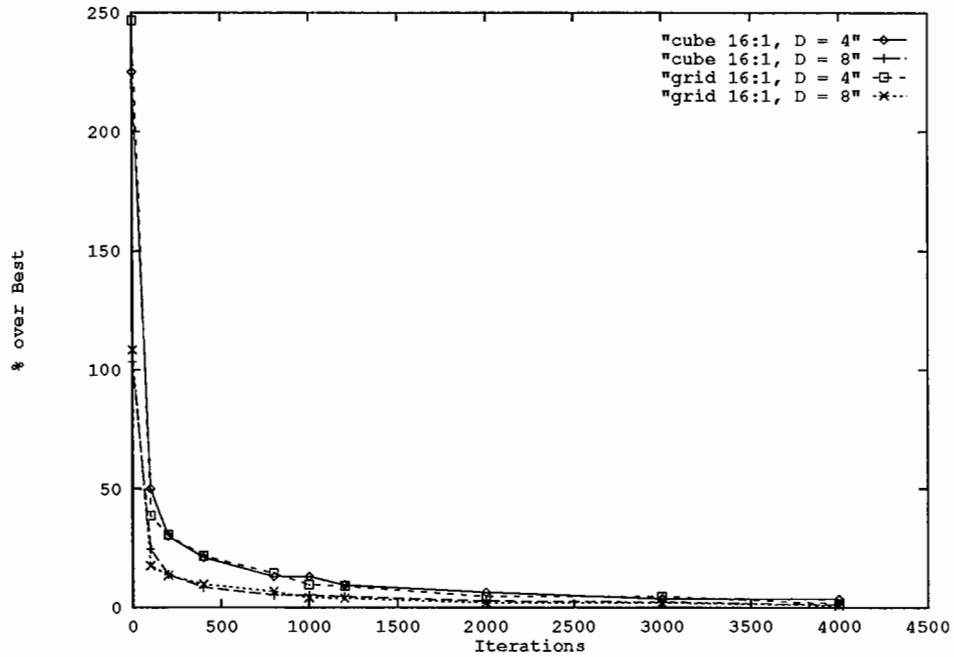

Figure 7: Convergence of *Mob*

for hypercube and grid embeddings described in section 3 should serve as a basis for constructing worst case examples, where at each *Mob* iteration a small and constant improvement in the cost function is made, thus forcing *Mob* to slowly "creep" towards a local minimum.

The *Iterations* columns in Table 2 and Table 3 show that the reductions in embedding costs decrease rapidly as the total number of iterations increase. The rate of convergence is shown in Figure 7 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 increases, which can be seen in Figure 7. Thus, fewer iterations are needed on high-degree graphs.

Computation Time The edge and vertex data structures are never used at the same time in the *Mob* heuristic. The edge data structure, the larger of the two, requires $2|E|$ processors. The scan operations are the most complex operations in an iteration of *Mob*. Thus, one iteration of *Mob* should run in time $O(\log |E|)$. This was tested by experiment on the CM-2 with graphs of different sizes, where 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 Table 5 for grid embeddings. We find that the time per iteration normalized by the number of virtual processors grows logarithmically on the Connection Machine. 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 can be computed.

As we 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 complexity of the *Mob* heuristic is time $O(\log |E|)$ with $2|E|$ processors.

Table 4 and Table 5 also indicate that one iteration of the *Mob* grid-embedding heuristic is approximately 40-50% slower than the corresponding iteration of the *Mob* hypercube-embedding heuristic, since 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* has to select a vertex with maximum gain for every node and shuffles it to the head of the node's vertex list, as described in detail in the move generation stage in section 4.3.

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 (29 minutes) to find an embedding of a 500K-vertex, 1M-edge graph into a 15-dimensional hypercube, and approximately 1264 seconds (21 minutes) to embed that graph into a 19-dimensional hypercube. It takes approximately 2370 seconds (40 minutes) to find an embedding of a 500K-vertex, 1M-edge graph into a 256×64 grid, and approximately 2157 seconds (36 minutes) to embed that graph into a 1024×512 grid. All these embeddings are within 5 percent of best-ever.

Comparison to Graph-partitioning Algorithms To calibrate our results, we measured the graph partitions produced by the hypercube and grid embeddings. A graph embedding S is a mapping of graph vertices to network nodes. S is completely described by storing for every vertex v the network node on which v is embedded. In the implementation of *Mob*, this node field contains the hypercube address of the network node, which is used directly for hypercube embeddings, and from which x,y-coordinates are computed for grid embeddings. A graph partition $P = (X, Y)$ can be generated from an embedding S by cutting the embedding in half along a hyperplane. A bit position is selected in the node field, and placing all vertices with

a 0 in that bit position in set X, and all vertices with a 1 in that bit position in set Y of the partition P. A d-dimensional hypercube network has d hyperplanes and thus d projected partitions, from which we can choose the partition with minimum cost. For hypercube embeddings, we found that the cost of the projected partitions, measured by the number of edges between the two sets, are about the same for all cutting hyperplanes. For grid embeddings, the two hyperplanes corresponding to vertical and horizontal lines cutting the grid in half gave the best partitions.

Table 6 shows how the *Mob* hypercube and grid embedding algorithms performed 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 [45,46,47]. The cost of the graph embeddings $P = (X, Y)$ is given in Table 6 by the percentage of all edges that cross the cut between X and Y. We found that both 16-to-1 *Mob* embedding algorithms produced graph partitions comparable to the *Mob* graph-partitioning algorithm. The KL graph-partitioning algorithm, which we ran on graphs with only 32K (or less) vertices due to running time considerations, was significantly outperformed by both 16-to-1 *Mob* embedding algorithms. KL was slightly worse than the 1-to-1 *Mob* grid-embedding algorithm and slightly better than the 1-to-1 *Mob* hypercube-embedding algorithm.

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

Comparison to Simulated Annealing Chen *et al.* [12,13] evaluated the performance of 1-to-1 hypercube-embedding heuristics. The graphs to be embedded 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 tested algorithms, simulated annealing with a move set limited to swapping vertex pairs along hypercube edges produced the best solutions.

We obtained the 10 random graphs used by Chen *et al.* and also generated 10 random graphs with our own generator. For each graph 5 runs were performed, and results were averaged over these runs. Each run of *Mob* was limited by 8000 iterations, and a schedule as described above was used. Chen *et al.* report that on 10 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 our performance was

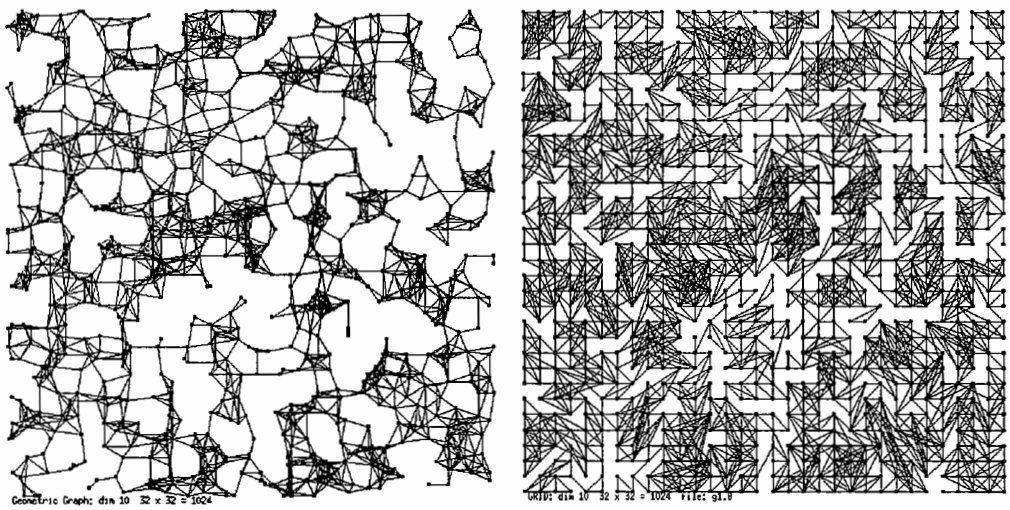

Figure 8: (a) Random geometric graph (b) Slice heuristic solution

about equal to the performance of a tuned version of SA.

Our experiments on much larger graphs, shown in Table 2, indicate that the *Mob* achieves a 51.7% reduction for degree 7 graphs, with very little variation between individual runs. Thus, we believe that the difference between 58.1% and 51.7% is an artifact of using very small graphs. It would be interesting to see how SA performs on larger examples, unless excessive running time prohibits experiments.

5.2 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 harder to find for a local search heuristic. 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 $[0,1) \times [0,1)$. These points are the vertices of $G_{n,d}$. For all vertex pairs, the geometric graph $G_{n,d}$ contains an edge if the pair of vertices are a distance of r or less apart, measured by a distance metric. To obtain graphs of degree d, the distance parameter r is set to the value r_d. The following three metrics have been used in the literature:

(a) *Manhattan metric* $r = |x_1 - x_2| + |y_1 - y_2|$, $r_d = \sqrt{\frac{d}{2n}}$

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

(c) *Infinity metric* $r = \max(|x_1 - x_2|, |y_1 - y_2|)$, $r_d = \sqrt{\frac{d}{4n}}$

For our experiments we used the Manhattan metric, since it matches the cost function used for grid embeddings. The graph-partitioning experiments done by Johnson *et al.* [28] were performed on random geometric graphs generated with the euclidean metric. Chen *et al.* [12,13]used random geometric graphs generated with the infinity metric to test hypercube embedding algorithms.

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 $O(n^2)$ work algorithm. This approach was perfectly adequate in previous studies, which concerned themselves with the embeddings of small graphs. A more sophisticated method is required for graph sizes of $1,000,000$ vertices or more.

Our approach was to divide the unit square into $1/r_d \times 1/r_d$ cells. It follows that for every vertex, all the vertices a distance of r_d or less apart must be located in the same cell or in one of the 8 neighboring cells. This holds for any of the above metrics. Every vertex computes into which cell it belongs, and the vertices are sorted by their cell value. 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 8 neighboring cells.

The number of cells on the unit square is $1/r_d \times 1/r_d = 2n/d$ for the Manhattan metric. n vertices were distributed randomly over the unit square. Thus every cell will contain an average of $d/2$ vertices. An average total of $9nd/2$ distance computations are done to generate $nd/2$ edges. The total computational work is $O(n \log n + nd)$. Under the realistic assumption that sorting n vertices by their cell value will take 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 $O(n)$ processors. Our experiments show that the constants are very small.

Note that the above search structure only works for points 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 above much simpler algorithm exists.

We can derive a simple upper bound for the bisection width b of a geometric graph $G_{n,d}$. Assume a vertical line divides the unit square into two sets containing

$n/2$ vertices each. We can place $1/r_d$ cells along both sides of the line. The cells will contain an average of $d/2$ vertices when the Manhattan metric is used. The edges $G_{n,d}$ are of length r_d or less, so any edge that crosses the vertical line has to have its two vertices in the cells along the vertical line. A vertex in one cell can be connected by an edge to all vertices in the 3 neighboring cells across the vertical line. Thus b_{min}, the average number of edges crossing the vertical line is

$$b_{min} \leq 3\frac{1}{r_d}\left(\frac{d}{2}\right)^2 = 3\sqrt{n}\left(\frac{d}{2}\right)^{3/2} = O(\sqrt{n})$$

whereas the cost of a random graph partitioning of $G_{n,d}$ can be estimated by

$$b_r \approx \frac{nd}{4}$$

We can get an estimate of g_{min}, the minimum cost grid embedding, by superimposing a $\sqrt{n} \times \sqrt{n}$ grid on the unit square, and assuming that the vertices of $G_{n,d}$ get mapped to grid vertices close to their locations on the unit square. Thus the total edge cost g_{min} under the Manhattan metric is approximately

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

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

$$g_r \approx \frac{nd}{4}\sqrt{n}$$

Since we can fold the $\sqrt{n} \times \sqrt{n}$ grid into a hypercube of dimension $\log n$, the same estimate as given above holds for h_{min}, the minimum cost hypercube embedding

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

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

$$h_r \approx \frac{nd}{4}\log n$$

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 columns labeled *Slice* in Table 9 and Table 8 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 have to be multiplied by the number of edges $\frac{nd}{2}$ to give total embedding costs.)

The Slice Heuristic Intuitively, if every randomly generated vertex of $G_{n,d}$ on the unit square was 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 that we present 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 [40] for VLSI floorplan design. The vertices are sorted along the x or the y-dimensions. 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 vertex set and the grid node set, is repeated until the sets are of size one. At this point every vertex will have one unique grid node assigned to it.

Johnson *et al.* [28] 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 the results produced by Slice here since we suspect it produces solutions very close to the optimal embedding. This allows 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 to do so is a grid-embedding heuristic.

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 5 runs were performed for each embedding. The mob schedule used was the same as for random graphs, and is given in subsection 5.1. *Mob* was always stopped after 8000 iterations.

The computation time needed for one iteration of Mob is the same as for random graphs, so Table 4 and Table 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.

Solution Quality of *Mob* The quality of the solutions produced by *Mob*is shown in Table 8 and Table 9 for both 1-to-1 and 16-to-1 embeddings. Table 8 gives results for hypercube embeddings, Table 9 gives results for grid embeddings. The results in these tables are expressed as average edge lengths, and have to be multiplied by the number of edges $\frac{nd}{2}$ to give total embedding costs. The columns labeled D give the dimension of the hypercube that the graph is embedded in. For the grid embeddings, D is the dimension of the hypercube containing a $2^{\lfloor D/2 \rfloor} \times 2^{\lceil D/2 \rceil}$ grid. The columns labeled R give the average edge length produced by a random embedding. The columns labeled *Slice* give the average edge length produced by the Slice heuristic. The columns labeled *Mob* give the average edge length in an embedding produced by *Mob* from an initial random embedding. The columns labeled $SLICE/R$ and Mob/R give the ratio of improvement produced by Slice and *Mob* over a random solution.

Our experiments show:

(a) The Slice heuristic produces slightly better results than *Mob* for hypercube embeddings, and considerably better results than *Mob* for grid embeddings.

(b) For fixed degree d, Mob/R is largely independent of graph size, but $Slice/R$ becomes smaller with increasing graph size. This means that the gap between a near-optimal solution and *Mob* will widen as graphs become larger.

(c) 16-to-1 mappings give better Mob/R ratios than 1-to-1 mappings.

(d) The grid-embedding *Mob* heuristic achieves lower Mob/R ratios than the hypercube *Mob* heuristic.

(e) The ratio Mob/R rises with increasing average graph degree toward an asymptote of 1. The differences between grid and hypercube embeddings and between 1-to-1 and 16-to-1 embeddings become smaller with increasing graph degree.

(f) For geometric graphs, the ratio Mob/R is smaller than for random graphs. (Compare Table 2, 3) to Table 8, 9) For 16-to-1 and 1-to-1 grid embeddings and for

1-to-1 hypercube embeddings, Mob/R is almost 10 times smaller for degree 4 graphs, and almost 5 times smaller for degree 8 graphs. For 1-to-1 hypercube embeddings, Mob/R of geometric graphs is about 2 times smaller than Mob/R of random graphs. So 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.

Rates of Convergence of *Mob* Table 8 and Table 9 also report the convergence of *Mob* for an increasing number of iterations. The columns labeled *Iterations* show the average embedding cost as percentages above the 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 best solution (produced by Slice) increases with increasing graph size.

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 the corresponding paragraph in subsection 5.1. Table 10 shows how the *Mob* hypercube and grid embedding algorithms performed as graph-partitioning algorithms, compared to the *Mob* graph-partitioning algorithm, and the KL graph-partitioning algorithm .

The cost of the graph embeddings $P = (X, Y)$ is given in Table 10 as the percentage of all edges that cross the cut between X and Y. The *Mob* graph-partitioning heuristic produced better results that the hypercube and grid embedding algorithms, but does not approach the partitions produced by Slice. At least with a constant number of iterations, the *Mob* heuristics produce embeddings where the average edge length as a function of graph size is constant or decreases slowly, whereas the average edge lengths produced by Slice as a function of graph size decrease quickly, about $O(\frac{1}{\sqrt{n}})$.

We found that both 16-to-1 embedding algorithms and the grid 1-to-1 embedding algorithm produced good graph partitions which were roughly larger by a factor of 1.5 to 4 than the *Mob* graph-partitioning algorithm. The KL graph-partitioning algorithm, which we only ran on graphs with only 32K (or less) vertices due to running time considerations, was slightly worse than these three graph-embedding heuristics. The hypercube 1-to-1 algorithm produced graph partitions which were roughly larger by a factor of 10, an order of magnitude.

As we noted for random graphs, we find the performance of the *Mob* embedding algorithms used as graph-partitioning algorithms remarkable, considering that *Mob*

is optimizing the "wrong" cost function. None of the local-search algorithms that we implemented was able to get close to the results produced by Slice.

Comparison to Simulated Annealing Analogously 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.* [12, 13]. 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 $r_d = \sqrt{\frac{d}{4n}} = 0.117386$ to obtain graphs of degree 7.

For each graph 5 runs were performed, and 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 are given in Table 11, expressed as % reduction in edge lengths of a random embedding. Chen *et al.* report that on 10 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*.

6 Conclusions

We have shown here that some local search heuristics based on the SWAP neighborhood, such as simulated annealing for cube-embedding and grid-embedding problems are P-hard. This suggests there is (probably) no parallel SA algorithm that does exactly what the serial algorithm does.

We have developed a new graph-embedding algorithm and run it on the CM-2 Connection Machine to show that it is a massively parallel algorithm, 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) circuits.

We believe that it is possible to show that many P-complete or P-hard local search heuristics for NP-complete problems

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] F. Afrati, C. H. Papadimitriou and G. Papadimitriou, "The Complexity of Cubical Graphs," in *Information and Control*, pp. 53–60, 1985.

[3] 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*, vol. PDS-1, no. 1, pp. 91–106, January 1990.

[4] S. Bettayeb, Z. Miller and I. H. Sudborough, "Embedding Grids onto Hypercubes," in *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing* , pp. 201–211, 1988.

[5] G. E. Blelloch, "Scans as Primitive Parallel Operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[6] S. H. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, Mar. 1981.

[7] S. W. Bollinger and S. F. Midkiff, "Processor and Link Assignment in Multicomputers using Simulated Annealing," *Proceedings International Conference on Parallel Processing*, vol. 1, pp. 1–6, 1988.

[8] M. A. Breuer, "Min-Cut Placement," *Design Automation and Fault-Tolerant Computing*, vol. 1, no. 4, pp. 343–362, Aug. 1977.

[9] A. Casotto and A. Sangiovanni-Vincentelli, "Placement of Standard Cells using Simulated Annealing on the Connection Machine," in *ICCAD*, pp. 350–453, Nov. 1987.

[10] R. D. Chamberlain, M. N. Edelman, M. A. Franklin and E. E. Witte, "Simulated Annealing on a Multiprocessor," in *Proceedings of the International Conference on Computer Design*, pp. 540–544, 1988.

[11] M. -Y. Chan, "Dilation-2 Embeddings of Grids into Hypercubes," *Proceedings International Conference on Parallel Processing*, vol. 3, pp. 295–298, 1988.

[12] W. -K. Chen, E. F. Gehringer and M. F. M. Stallmann, "Hypercube Embedding Heuristics: An Evaluation," in *International Journal of Parallel Programming*, to appear, 1990.

[13] W. -K. Chen and M. F. M. Stallmann, "Local Search Variants for Hypercube Embedding," in *Proceedings 5th Distributed Memory Computer Conference*, to appear, 1990.

[14] C. Corneil and R. C. Read, "The Graph Isomorphism Disease," *Journal of Graph Theory*, vol. 1, no. 4, pp. 339–363, 1977.

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

[16] F. Darema, S. Kirkpatrick and V. A. Norton, "Parallel Algorithms for Chip Placement by Simulated Annealing," *IBM Journal of Research and Development*, vol. 31, no. 3, pp. 391–401, May 1987.

[17] A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Transaction on Computer-Aided Design*, vol. CAD-4, no. 1, pp. 92–98, Jan. 1985.

[18] C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," in *19th IEEE Design Automation Conference*, pp. 175–181, 1982.

[19] K. Fukunaga, S. Yamada and T. Kasai, "Assignment of Job Modules onto Array Processors," *IEEE Transactions on Computers*, vol. c-36, no. 7, pp. 888–891, July 1987 .

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

[21] S. Geman and D. Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images," in *Neurocomputing: Foundations of Research*, J. A. Anderson and E. Rosenfeld, Eds. MIT Press, 1988.

[22] J. Gill, "Computational Complexity of Probabilistic Turing Machines," *SIAM Journal on Computing*, vol. 6, no. 4, pp. 675–695, 1977.

[23] L. M. Goldschlager, "The Monotone and Planar Circuit Value Problems," *ACM Sigact News*, vol. 9, no. 2, pp. 25–29, 1977.

[24] L. M. Goldschlager, "A Space Efficient Algorithm for the Monotone Planar Circuit Value Problem," *Information Processing Letters*, vol. 10, no. 1, pp. 25–27, 1980.

[25] D. R. Greening, "A Taxonomy of Parallel Simulated Annealing Techniques," IBM, Technical Report No. RC 14884, 1989.

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

[27] C. -T. Ho and S. L. Johnsson, "On the Embedding of Arbitrary Meshes in Boolean Cubes with Expansion Two Dilation Two," in *Proceedings International Conference on Parallel Processing*, pp. 188–191, 1987.

[28] D. S. Johnson, C. A. Aragon, L. A. McGeoch and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation (Part 1)," *Operations Research*, vol. 37, no. 6, pp. 865–892, Nov.-Dec. 1989.

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

[30] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[31] S. Kravitz and R. Rutenbar, "Multiprocessor-based Placement by Simulated Annealing," in *23rd IEEE Design Automation Conference*, pp. 567–573, 1986.

[32] R. E. Ladner, "The Circuit Value Problem is Log Space Complete for P," *ACM SIGACT News*, vol. 7, no. 1, pp. 18–20, 1975.

[33] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, vol. 27, pp. 831–838, 1980.

[34] J. Lam and J. -M. Delosme, "Simulated Annealing: a Fast Heuristic for Some Generic Layout Problems," in *ICCAD*, pp. 510–513, 1988.

[35] M. Lundy and A. Mees, "Convergence of an Annealing Algorithm," *Mathematical Programming*, vol. 34, no. 1, pp. 111–124, 1986.

[36] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, "Equations of State Calculations by Fast Computing Machines," *Journal of Chemical Physics*, vol. 6, no. 21, pp. 1087–1091, June 1953.

[37] D. Mitra, F. Romeo and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behaviour of Simulated Annealing," *Advances in Applied Probability*, vol. 18, no. 3, pp. 747–771, Sept. 1986.

[38] B. Monien and I. H. Sudborough, "Simulating Binary Trees on Hypercubes," in *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing* , pp. 170–180, 1988.

[39] K. Mulmuley, U. V. Vazirani and V. V. Vazirani, "Matching is as Easy as Matrix Inversion," in *19th Annual ACM Symposium on Theory of Computing*, pp. 345–354, 1987.

[40] R. H. J. M. Otten, "Automatic Floorplan Design," in *Proc. 19th IEEE Design Automation Conference*, pp. 261–267, 1982.

[41] P. Roussel-Ragot and G. Dreyfus, "A Problem Independent Parallel Implementation of Simulated Annealing: Models and Experiments," *IEEE Transaction on Computer-Aided Design*, vol. 9, no. 8, pp. 827–835, Aug. 1990.

[42] P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1408–1424, Dec. 1987.

[43] P. Sadayappan and F. Ercal, "Cluster-Partitioning Approaches to Mapping Parallel Programs onto a Hypercube," in *Supercomputing: 1st International Conference*, pp. 475–497, June 1987.

[44] J. E. Savage, *The Complexity of Computing*. John Wiley and Sons, 1976.

[45] J. E. Savage and M. G. Wloka, "Parallelism in Graph-Partitioning," in *Journal of Parallel and Distributed Computing*, to appear, 1991.

[46] J. E. Savage and M. G. Wloka, "Heuristics for Parallel Graph Partitioning," Department of Computer Science, Brown University, Technical Report No. CS-89-41, Dec. 1989.

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

[48] J. E. Savage and M. G. Wloka, "Parallelizing SA For Graph Embedding Is Hard," in *Proceedings of the IMACS World Congress on Computation and Applied Mathematics, Special Session on Simulated Annealing*, Dublin, to appear, July 1991.

[49] J. E. Savage and M. G. Wloka, "Parallel Graph-Embedding Heuristics," in *5th SIAM Conference on Parallel Processing for Scientific Computing*, Houston, to appear, Mar. 1991.

[50] A. A. Schäffer and M. Yannakakis, "Simple Local Search Problems That Are Hard to Solve," preprint, 1990.

[51] G. B. Sorkin, "Simulated Annealing on Fractals: Theoretical Analysis and Relevance for Combinatorial Optimization," in *Advanced Research in VLSI, Proceedings of the Sixth MIT Conference*, pp. 331–351, 1990.

[52] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[53] A. Wagner and D. G. Corneil, "Embedding Trees in a Hypercube is NP-Complete," *SIAM Journal on Computing*, vol. 19, no. 4, pp. 570–590, June 1990.

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

[55] A. Y. Wu, "Embedding of Tree Networks into Hypercubes," in *Journal of Parallel and Distributed Computing*, pp. 238–249, 1985.

Table 1: Large random graphs

Graph	V	E	d
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 embeddings: Average edge lengths

	16-to-1 mappings						
					Iterations		
Graph	D	R	Mob	Mob/R	100	1000	4000
Degree 4							
4.16K	10	5.0	1.538	.3076	50.0	13.1	3.6
4.32K	11	5.5	1.720	.3127	51.3	10.6	2.2
4.64K	12	6.0	1.861	.3102	55.8	13.9	3.0
4.128K	13	6.5	2.022	.3110	68.7	12.3	4.0
4.256K	14	7.0	2.175	.3107	72.7	13.3	3.9
4.512K	15	7.5	2.369	.3159	76.6	12.6	1.8
Degree 8							
8.8K	9	4.5	2.207	.4904	20.6	4.9	1.0
8.16K	10	5.0	2.460	.4920	24.6	5.3	1.2
8.32K	11	5.5	2.712	.4931	25.4	4.7	0.9
8.64K	12	6.0	2.955	.4925	31.4	5.7	1.8
8.128K	13	6.5	3.201	.4925	29.6	6.6	1.7
8.256K	14	7.0	3.473	.4961	31.7	5.0	0.8
Variable Degree							
3.16K	10	5.0	1.148	.2296	79.8	19.4	5.1
4.16K	10	5.0	1.538	.3076	50.0	13.1	3.6
5.16K	10	5.0	1.874	.3748	34.9	9.4	1.7
6.16K	10	5.0	2.111	.4222	28.7	7.6	1.3
7.16K	10	5.0	2.312	.4624	25.4	6.0	1.1
8.16K	10	5.0	2.460	.4920	24.6	5.3	1.2
9.16K	10	5.0	2.607	.5214	23.8	4.6	1.1
10.16K	10	5.0	2.719	.5434	21.3	4.3	1.1
16.16K	10	5.0	3.186	.6372	13.5	2.7	0.5

	1-to-1 mappings						
					Iterations		
Graph	D	R	Mob	Mob/R	100	1000	4000
Degree 4							
4.16K	14	7.0	2.559	.3655	62.7	16.1	6.6
4.32K	15	7.5	2.783	.3711	62.5	16.0	3.2
4.64K	16	8.0	2.924	.3655	66.2	16.0	2.7
4.128K	17	8.5	3.111	.3660	90.8	16.2	5.0
4.256K	18	9.0	3.251	.3612	97.7	17.3	4.5
4.512K	19	9.5	3.398	.3577	98.9	19.8	4.2
Degree 8							
8.8K	13	6.5	3.497	.5380	28.5	6.9	1.2
8.16K	14	7.0	3.759	.5370	30.3	6.6	1.4
8.32K	15	7.5	4.007	.5343	31.8	7.1	1.6
8.64K	16	8.0	4.261	.5326	32.9	7.6	2.5
8.128K	17	8.5	4.517	.5314	33.8	8.5	2.7
8.256K	18	9.0	4.781	.5312	35.1	8.3	2.4
Variable Degree							
3.16K	14	7.0	2.123	.3033	80.3	24.2	4.5
4.16K	14	7.0	2.559	.3655	62.7	16.1	6.6
5.16K	14	7.0	2.975	.4250	47.9	11.6	4.5
6.16K	14	7.0	3.291	.4701	40.0	9.1	2.2
7.16K	14	7.0	3.553	.5075	34.5	7.6	1.5
8.16K	14	7.0	3.759	.5370	30.3	6.6	1.4
9.16K	14	7.0	3.933	.5619	27.4	6.4	1.7
10.16K	14	7.0	4.085	.5836	24.3	5.7	1.6
16.16K	14	7.0	4.666	.6666	17.0	4.5	1.0

Table 3: Grid embeddings: Average edge lengths

	16-to-1 mappings						
					Iterations		
Graph	D	R	Mob	Mob/R	100	1000	4000
Degree 4							
4.16K	10	21.3	6.142	.2884	38.7	9.6	1.7
4.32K	11	32.0	9.312	.2910	40.2	11.8	2.5
4.64K	12	42.6	12.486	.2931	41.5	12.9	2.9
4.128K	13	64.0	18.917	.2956	43.4	13.0	2.8
4.256K	14	85.3	25.186	.2953	49.6	12.4	1.3
4.512K	15	128.0	37.787	.2952	52.8	12.2	3.3
Degree 8							
8.8K	9	15.9	7.694	.4839	16.8	3.4	0.3
8.16K	10	21.3	10.226	.4801	17.7	4.1	0.9
8.32K	11	32.1	15.525	.4836	18.1	4.4	0.7
8.64K	12	42.7	20.737	.4856	18.5	4.8	1.1
8.128K	13	64.0	31.063	.4854	18.5	5.2	1.3
8.256K	14	85.3	41.672	.4885	18.7	5.3	1.2
Variable Degree							
3.16K	10	21.2	4.417	.2074	61.8	15.7	3.4
4.16K	10	21.3	6.142	.2884	38.7	9.6	1.7
5.16K	10	21.3	7.688	.3609	29.2	6.8	1.0
6.16K	10	21.2	8.727	.4117	24.2	6.0	1.3
7.16K	10	21.3	9.597	.4506	20.3	4.7	1.0
8.16K	10	21.3	10.226	.4801	17.7	4.1	0.9
9.16K	10	21.3	10.934	.5133	15.7	3.6	0.7
10.16K	10	21.3	11.450	.5376	14.7	3.3	0.7
16.16K	10	21.3	13.449	.6314	9.8	2.2	0.5

	1-to-1 mappings						
					Iterations		
Graph	D	R	Mob	Mob/R	100	1000	4000
Degree 4							
4.16K	14	85.1	26.702	.3138	54.1	16.9	4.6
4.32K	15	128.3	40.387	.3148	55.9	16.4	4.3
4.64K	16	170.7	55.386	.3245	59.9	17.2	4.5
4.128K	17	256.1	81.779	.3193	59.9	18.1	3.9
4.256K	18	341.1	108.484	.3180	65.6	17.4	4.0
4.512K	19	512.2	163.316	3189	68.4	18.7	4.2
Degree 8							
8.8K	13	64.1	31.987	.4990	24.9	7.9	2.2
8.16K	14	85.5	42.779	.5003	24.8	7.8	1.9
8.32K	15	127.9	64.102	.5012	28.4	7.9	1.7
8.64K	16	170.7	85.097	.4985	27.8	8.1	2.0
8.128K	17	256.5	127.866	.4985	29.7	9.5	4.9
8.256K	18	341.4	170.619	.4998	29.7	8.0	1.7
Variable Degree							
3.16K	14	85.1	19.946	.2344	77.6	23.7	6.3
4.16K	14	85.1	26.702	.3138	54.1	16.9	4.6
5.16K	14	85.2	32.750	.3844	39.8	12.3	3.1
6.16K	14	85.8	36.978	.4310	33.7	10.1	2.3
7.16K	14	85.3	40.232	.4717	28.4	8.6	2.3
8.16K	14	85.5	42.779	.5003	24.8	7.8	1.9
9.16K	14	85.1	45.511	.5348	22.4	6.2	1.5
10.16K	14	85.3	47.416	.5559	19.8	6.1	1.4
16.16K	14	85.3	55.061	.6455	14.8	4.2	1.0

Table 4: Hypercube embedding: CM-2 execution times (secs) for 1 Mob iteration

	16-to-1 mappings				1 to 1 mappings			
Degree 4								
Graph	D	8K CM-2	16K CM-2	32K CM-2	D	8K CM-2	16K CM-2	32K CM-2
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: CM-2 execution times (secs) for 1 Mob iteration

	16-to-1 mappings				1-to-1 mappings			
Degree 4								
Graph	D	8K CM-2	16K CM-2	32K CM-2	D	8K CM-2	16K CM-2	32K CM-2
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: Random Graph Edge Lengths: Comparison to Graph Partition

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

Graph	V	E	d
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 embeddings of geometric graphs: Average edge lengths

	16-to-1 mappings								
							Iterations		
Graph	D	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	D	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	0.2539	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 embeddings of geometric graphs: Average edge lengths

	16-to-1 mappings								
							Iterations		
Graph	D	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	D	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: Geometric graph edge lengths: Comparison to graph partition

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: Random geometric graphs

Heuristic	Min	Avg. Cost	Avg. 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