Heuristics for Parallel Graph-Partitioning

John E. Savage
Markus G. Wloka

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-89-41
Revised Version
January 29, 1991

Heuristics for Parallel Graph-Partitioning * † ‡

John E. Savage
Markus G. Wloka

Brown University
Department of Computer Science, Box 1910
Providence, Rhode Island 02912
Tel: 401-863-7600
jes@cs.brown.edu
mgw@cs.brown.edu

Abstract

Graph partitioning is an important NP-complete problem with applications in VLSI CAD, processor allocation, and many other areas. The problem is to partition vertices of a graph into two equal-sized sets so that the number of edges joining the sets is minimum. In this paper we show that the Kernighan-Lin heuristic is P-complete and the simulated annealing heuristic is P-hard, which suggests that they are both hard to parallelize. We also describe a new parallel heuristic (which we call the *Mob* heuristic) that on the 32K-processor CM-2 Connection Machine handles graphs with more than two million edges and gives in nine minutes partitions that are within 2% of the best ever found.

---

*This work was supported in part by National Science Foundation under Grant CDA 87-22809 and the Office of Naval Research under contract N00014-83-K-0146 and ARPA Order No. 4786.

†An abbreviated version of this paper appeared in the *Proceedings of the ICALP'90*, Lecture Notes in Computer Science, pp. 476-489, no. 443, Springer-Verlag [31].

‡This paper is to appear in the *Journal of Parallel and Distributed Computing*, Academic Press

1 Introduction

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 number of edges linking two sets of a partition is called the *bisection width* of the graph. A partition yielding a minimum bisection width is called a *min-cut* or a *minimum bisection*. Graph-partitioning is important in VLSI placement and routing [4,14, 15,26,29], memory segmentation to minimize paging [24], and processor allocation, among other areas.

Since graph partitioning is NP-complete [16], heuristics are needed to approximate the minimum bisection width. Two widely used heuristics for this problem are the Kernighan-Lin (KL) algorithm [24] and the simulated annealing (SA) algorithm introduced by Kirkpatrick *et al.* [25].

KL and SA are local search algorithms [23]. Each starts at some initial partition, usually chosen at random, and searches the neighborhood of the current partition for a new partition. Both heuristics give excellent results on serial machines for random graphs of small degree and moderate size [22,28]. 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.* [22]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.

We introduce a new partitioning heuristic, the *Mob* heuristic, that 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 between two sets but uses randomization to select which of a large number of equally good candidates will be swapped.

Mob was implemented on a CM-2 Connection Machine, a parallel machine with 32K processors. We ran *Mob* on random graphs with degrees ranging from 3 to 16. On random graphs we observed that *Mob*'s running time grows very slowly (logarithmically) with graph size if enough processors are available. The graph sizes were limited by available memory; we can currently handle graphs with up to 2 million edges, corresponding to an effective utilization of 4 million simulated processors. 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 and where a comparison with serial heuristics was feasible) and found that *Mob* gives excellent results.

An important reason for introducing a new heuristic is that both KL and SA are believed to be hard to parallelize. We show that KL is P-complete under logspace reductions on graphs

with unit-weight edges. A randomized local search algorithm, the zero-temperature version of SA, is shown to be P-hard under logspace reductions. We also show that *Mob* is P-hard under some restrictions, which occur only infrequently in practice.

In their study of local search, Johnson, Papadimitriou and Yannakakis [23] have shown that a search of a single KL neighborhood is P-complete when graphs have edge weights exponential in the size of a graph and the bisection size is defined to be the sum of the weights of edges in a bisection. Our P-completeness result is obtained for unit-weight edges. Schäffer and Yannakakis have independently shown that local search with the SWAP heuristic on unit-weight edge graphs is P-complete [34]. From this they deduce that a complete local search with KL of unit-weight-edge graphs is P-complete. Their result is implied by our result of completeness of KL for search of just one neighborhood. Our proof of P-hardness of zero-temperature SA implies P-completeness for deterministic local search heuristics using the SWAP neighborhood.

Other heuristics for graph partitioning have been developed. Fiduccia and Mattheyses [15] have modified the basic Kernighan-Lin algorithm to increase its flexibility and efficiency but keep its essential structure. Bui, Chaudhuri, Leighton and Sipser [5,6] give a min-cut, max-flow-based algorithm to partition a graph with unit edge weights. This algorithm works well with graphs of a particular structure having small expected bisection widths. It would be interesting to know if it can be parallelized. Bui, Heigham, Jones and Leighton [7] use a maximum random matching algorithm to coalesce nodes into pairs, thus forming a smaller graph of higher average degree, and then run either KL or SA on this graph to obtain a partition. Vertex pairs are then separated to create a partition for the original graph that is then used as an initial partition for either KL or SA.

The *hypergraph partitioning problem* is a generalization of the graph-partitioning problem [4]. 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. Since graph edges can be modeled as two-terminal nets, the logspace completeness results extend to local search heuristics applied to hypergraph partitioning, such as the Kernighan-Dunlop [14] and SA algorithms (at zero temperature).

This work is part of a larger effort to exploit parallelism in VLSI synthesis. Recent research in this area has produced new algorithmic methods, since traditional methods are essentially serial. Progress has been made on channel routing [32], river routing [11], knock-knee routing [12], mask verification [9], compaction [33]. Parallel simulated annealing heuristics for placement have been proposed by Casotto and Sangiovanni-Vincentelli [10] and Wong and Fiebrich [35] on the Connection Machine, by Banerjee *et al.* [2]on an Intel Hypercube, and by Darema *et al.* [13]on an IBM 3081 multiprocessor simulating virtual processors. Greening [20] gives a general survey in the field of parallel simulated annealing.

In the following section we define the KL, SA and *Mob* local search heuristics. In section 3 we demonstrate that KL and SA at zero temperature are P-complete and P-hard, respectively. In section 4 we describe the essential features of the CM-2 Connection Machine, the machine for which we developed our new heuristic. The implementation of our heuristic is described in detail section 5 and the experimental results obtained with it are described in section 6. The last section presents conclusions. An abbreviated version of this paper was presented in [31].

2 Local Search Heuristics

The goal of the graph partitioning problem is to find a partition of the vertices of a graph into disjoint, equal-size sets such that the weight of the edges between them is minimum.

Definition 1 Consider a graph $G = (V, E)$, with vertices V, $|V|$ even, and edges E, $E \subset V \times V$, and a function, $w : E \rightarrow Z$, that assigns integer weights to edges. The *graph partitioning problem* is to find a partition $P = (X, Y)$ of V into two disjoint, equal-size sets X and Y, so that the *bisection width*, $bw(P)$, the sum of all weights of edges joining the two sets, is minimum.

The *hypergraph-partitioning problem* is a generalization of the graph-partitioning problem to sets of vertices. In VLSI applications such a set is called a "net", and denotes pins on cells that must be connected electrically in a physical layout.

Definition 2 Consider a hypergraph $G = (V, N)$, with vertices V, $|V|$ even, and hypergraph edges $N \subset 2^V$ which are subsets of V. The *hypergraph partitioning problem* is to find a partition $P = (X, Y)$ of V into two disjoint, equal-size subsets X and Y, so that the number of hypergraph edges in N intersecting both sets is minimum.

We now define the KL, SA, and *Mob* local search algorithms and their *neighborhood* structures. Each heuristic is given some starting partition and chooses a partition from the neighborhood of the current partition as the starting point for another search. The rules of termination differ. The reader should refer to [15,24,25] for more detailed descriptions and examples of how the KL, FM, and SA heuristics work.

2.1 The KL Heuristic

Definition 3 The *KL-neighborhood* KL-N(P_0) of a partition $P_0 = (X_0, Y_0)$ is a set of partitions $\{P_i = (X_i, Y_i)\}$, where X_i and Y_i are each unions of two disjoint sets: $X_i = XF_i \cup XR_i$, $Y_i = YF_i \cup YR_i$. Let $a \in XR_{i-1}$ and $b \in YR_{i-1}$ be vertices which, if swapped, cause the largest reduction (or smallest increase, if no reduction is possible) in the bisection width $bw(P_{i-1})$. P_i is produced by swapping and freezing a and b:

$$\begin{array}{llll} XF_0 & = \emptyset & YF_0 & = \emptyset \\ XR_0 & = X_0 & YR_0 & = Y_0 \\ XF_i & = XF_{i-1} \cup \{b\} & YF_i & = YF_{i-1} \cup \{a\} \\ XR_i & = XR_{i-1} - \{a\} & YR_i & = YR_{i-1} - \{b\} \end{array}$$

The number of partitions in the neighborhood KL-N(P) is $|V|/2$. The KL heuristic is described by the following procedure.

```
KernighanLin(P)
    Let P                          be the initial partition;
    Q = best_partition_in_KL-N(P);
    while( bw(Q) < bw(P) ) {
        P = Q;
        Q = best_partition_in_KL-N(P);
    }
    return P;
```

The function best_partition_in_KL-N(P) returns the partition Q in the KL-N(P) neighborhood of a partition P that has the smallest bisection width. It uses the new partition Q for another iteration step until no improving partition Q can be found. We shall see that performing even one neighborhood search in the KL heuristic is P-complete. Additionally, the above algorithm remains P-complete if KL-N is replaced with a simpler neighborhood such as SWAP-N, which is defined below.

The *Fiduccia-Mattheyses* (FM) heuristic is similar to the KL heuristic but differs in how vertices are moved and in the data structures it uses. As a serial algorithm it runs faster in practice than KL. In FM a balance parameter r is specified and the smaller of two sets in a partition must have a size that is an approximate fraction r of the total number of vertices. The FM neighborhood consists of partitions obtained by moving and freezing individual vertices. A vertex is a candidate to be moved if it is not frozen and if changing its set causes the most improvement (or least degradation) in the bisection width without violating the balance condition. We will show that FM is also P-complete.

2.2 The SA Heuristic

The behavior of SA is determined by the SWAP neighborhood and a "cooling schedule:"

Definition 4 The *SWAP* neighborhood SWAP-N(P_0) of a partition $P_0 = (X_0, Y_0)$ is the set of partitions $\{P_i = (X_i, Y_i)\}$ obtained by swapping one element from X_0 with one from Y_0.

The number of partitions in the neighborhood SWAP-N(P) is $(|V|/2)^2$.

Definition 5 A *cooling schedule* CS of length L is a sequence $[CS_1, CS_2, \ldots, CS_L]$ of probability distribution functions where $CS_t : Z \rightarrow (0, 1)$.

Many implementations for SA use the cooling schedule defined by $CS_t(x) = k_1 e^{-x/k_2 T(t)}$ where $T(t)$ is the current "temperature" at which the heuristic operates and k_1 and k_2 are constants. In order to properly define SA, we must impose some restrictions on the cooling schedule functions CS. In general, CS must be computable by a uniform family of circuits generated in logspace and the circuits must have polylogarithmic depth so that no hidden serial bottlenecks are created.

We can now explicitly define the SA heuristic. We assume that it executes a number of steps, $q(n)$, polynomial in the number n of vertices. The probabilistic nature of SA is reflected in sets RX, RY containing $q(n)$ random index variables each, chosen uniformly and independently from the interval $[1, \ldots, n/2]$, and by a set RV of $q(n)$ random variables that are real numbers selected uniformly and independently from the interval $(0, 1)$. A random variable uniformly distributed over the interval $(0, 1)$ is less than or equal to p with probability p, assuming $0 \leq p \leq 1$.

```
SimulatedAnnealing(P)
    Let P                          be the initial partition;
    Let RX[1..q(n)], RY[1..q(n)]   be random indices in [1,n/2];
    Let RV[1..q(n)]                be random reals in (0,1);
    Let CS[1..q(n)]()              be cooling schedule functions;
    t = 1;
    while( t <= q(n) ) {
        Q = SWAP(P, RX[t], RY[t]);
        delta = bw(Q) - bw(P);
        if( delta < 0  OR   RV[t] < CS[t](delta) )
            P = Q;
        t = t + 1;
    }
    return P ;
```

SA randomly selects a partition Q by exchanging vertices $RX[t]$ and $RY[t]$ with the function SWAP$(P, RX[t], RY[t])$. The change Δ in the bisection width is computed and Q replaces the current working partition P if the bisection width decreases, or if the current probability distribution $RV_t < CS_t(\Delta)$. The SA heuristic keeps exploring the SWAP neighborhood of its working partition P until the time limit $q(n)$ is exceeded.

We show in section 3.4 that a version of SA called the "zero-temperature" version, ZT-SA, is P-hard. We now define this problem.

Definition 6 The *zero-temperature* version of SA (ZT-SA) uses a cooling schedule CS for which $RV_t < CS_t(\Delta)$ is always false.

In this version of SA only improving swaps are made. When the standard cooling schedule $CS_t(x) = k_1 e^{-x/k_2 T(t)}$ is used, this is equivalent to operating at zero temperature. While there is no uniform distribution on infinite sets, namely the integers Z, note that CS_t has a finite domain because the difference in bisection widths between any two partitions is bounded. ZT-SA is a form of probabilistic steepest-descent algorithm. Note that in practice, SA always enters a low temperature phase, which implies that if it enters this phase with a partition that is not locally optimal, the problem is believed to be very hard to parallelize.

2.3 The *Mob* Heuristic

Definition 7 The *mob-neighborhood* Mob-N(P_0, m) of a partition $P_0 = (X_0, Y_0)$ is the set of partitions obtained from P_0 by swapping a *mob* of m vertices from X_0 with an equal number of vertices from Y_0.

The number of partitions in the neighborhood Mob-N(P, m) is $\binom{|V|/2}{m}^2$. A mob is selected from the vertices in each set whose exchange would cause the largest individual change in the bisection width. The following rule $\rho(P, m, r_1, r_2)$ is used to select a partition from Mob-N(P, m):

Definition 8 The *rule* $\rho(P, m, r_1, r_2)$ selects a mob of *size* m from Mob-N(P, m) as follows: Let r_1, r_2 be random real numbers drawn uniformly from the interval $(0, 1)$. For each vertex v, let $gain(v)$ be the change in the bisection width if v alone changes sides, and let $gain(v)$ be positive if the bisection width decreases. Let $SX(g) = \{v \in X \mid gain(v) \geq g\}$. 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.

The procedure is applied to both sides of the partition. This mob-selection rule is designed to be computed very quickly in parallel. The rule ρ looks for a rough approximation to the subset that causes the largest improvement in the bisection width. 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.

A schedule of mob sizes is used to control the searches:

Definition 9 A *mob schedule* MS of length L is a sequence $[MS_1, MS_2, \ldots, MS_L]$ of integers from the interval $[1, \ldots, n/2]$.

In section 6 we describe experiments that we have performed with *Mob* on random graphs with small degrees. In these experiments, we usually let the starting mob size MS_1 be 10% of the size of each set in a partition and decrease the mob size linearly in about 50 steps. This sequence is then repeated until *Mob* terminates.

We now explicitly define the *Mob* heuristic. It searchs a mob-neighborhood of a partition with a given mob size, and if the new partition found has a smaller bisection width, the search is repeated on the neighborhood of the new partition with the same mob size. If the bisection width increases, the neighborhood of the new partition 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 $(0, 1)$. Our heuristic is outlined below.

```
Mob(P)
    Let P                          be the initial partition;
    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 = RULE(P, MS[s], R1[t], R2[t]);
        if( bw(Q) > bw(P) )
            s = s + 1;
        P = Q;
        t = t + 1;
    }
    return P ;
```

We shall show in section 3.4 that when the mob size is fixed at 1 by the mob schedule, the execution of a sequence of moves in the neighborhood Mob-N(P, 1) up to a locally minimal partition is P-hard. The *Mob* heuristic based on exchange of subsets of the partition is in the class of heuristics described in general terms by Kernighan and Lin [24] but is different, of course, from their "two-way" heuristic described above. Bui [8] describes a "block" heuristic in which the gains of all vertices are computed in advance. Then vertices in each set of a partition with the largest gains are repeatedly selected and frozen without updating gains. This defines a neighborhood and the partition within this neighborhood with the smallest bisection width is selected. One pass is taken. Bui reports good results on graphs with large degree, graphs for which most heuristics give fairly good results.

3 Parallelizability of KL and SA

We have introduced the *Mob* heuristic because the KL and SA heuristics are expected to be hard to parallelize. We formalize this fact in this section by showing that KL is P-complete and ZT-SA is P-hard under logspace reductions. 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 10 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 11 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 12 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.

If a P-complete problem is in NC, the problems solvable by a uniform class of circuits with polynomial size and polylogarithmic depth, then P is contained in NC [27], 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 [17] 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 $f(n)$ 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 13 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 $NP \subseteq PP$ [17]. 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 14 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 $P \subseteq \begin{matrix} BPP \\ NP \end{matrix} \subseteq PP$ [17]. 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 [30]. 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 15 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 [27].*

Restricted versions of CVP are also P-complete, which will simplify the proof of P-completeness of KL. A *monotone circuit* uses only the operations AND and OR. The *monotone circuit value problem* (MCVP) is P-complete [18]. The *planar circuit value problem* is also P-complete [18], but the *planar monotone circuit value problem* is in NC [19]. 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.

The reduction in the following section will use a P-complete circuit value problem where some of the above restrictions are combined: a dual-rail monotone circuit with fan-in/fanout of 2. We will use these facts to show that if local search in the Kernighan-Lin neighborhood can be done in polylogarithmic parallel time, then CVP can be also be done in polylogarithmic parallel time. The same is true for the ZT-SA.

KL, 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-Completeness of the KL Heuristic

In this section we demonstrate that KL is P-complete for graphs with unit-weight edges, the case of interest for many practical applications. Our proof consists of a logspace procedure to translate a given *fan-in/fan-out 2, dual-rail monotone circuit* DR(C) into a graph G. When we apply the KL heuristic to G, it returns the best partition P_1 in the neighborhood KL-N(P_0) of the given initial partition P_0 which uniquely determines the value of DR(C). The actual value of the circuit DR(C) can then be determined from P_1 in logarithmic space. Thus the KL language is P-complete. If the partition P_1 can be computed by a deterministic

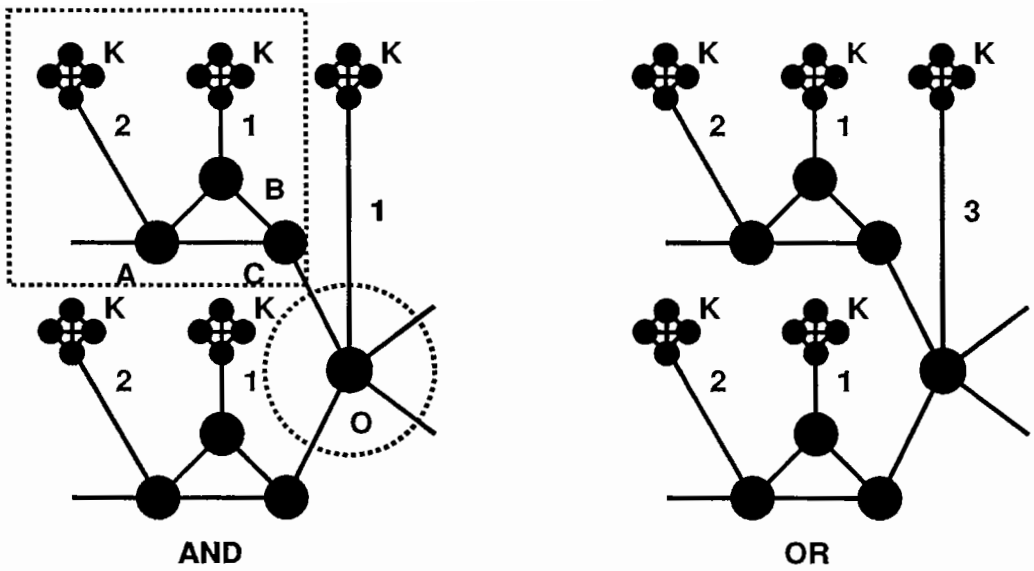


Figure 1: AND and OR Equivalent Subgraphs

circuit of polynomial width and polylogarithmic depth, so can the value of any circuit DR(C). This would imply that P $\subseteq$ NC.

Theorem 2 *The Kernighan-Lin local search algorithm for graph bisection is P-complete.*

Proof We reduce DRMCVP to KL by constructing graphs so that the early steps in the local search by KL mimic the computation by a monotone circuit when all inputs initially have value 0 and then some inputs are assigned value 1. We are given a fan-in/fan-out 2, dual-rail monotone circuit DR(C). AND and OR gates are represented by subgraphs, as shown in Figure 1. A gate subgraph has two *input arms*, with three vertices that we label A, B, and C, and a *central* vertex labeled O. Note that edges with missing vertices are connected to other gate subgraphs or input cliques. A composite graph is created from a circuit by interconnecting such gate subgraphs, as shown in the example of Figure 2. Extra subgraphs are added to direct the action of KL, as in Figure 3.

The vertices are partitioned into two sets, denoted X and Y, which correspond to circuit values of 0 and 1, as suggested in Figure 3. The initial sets are X_0 and Y_0. KL then searches its neighborhood swapping vertices. This search process moves some gate subgraphs from X to Y, thereby denoting that a value of 1 has propagated to their outputs. Gate subgraphs are moved in an order consistent with the propagation of input values to outputs in a circuit

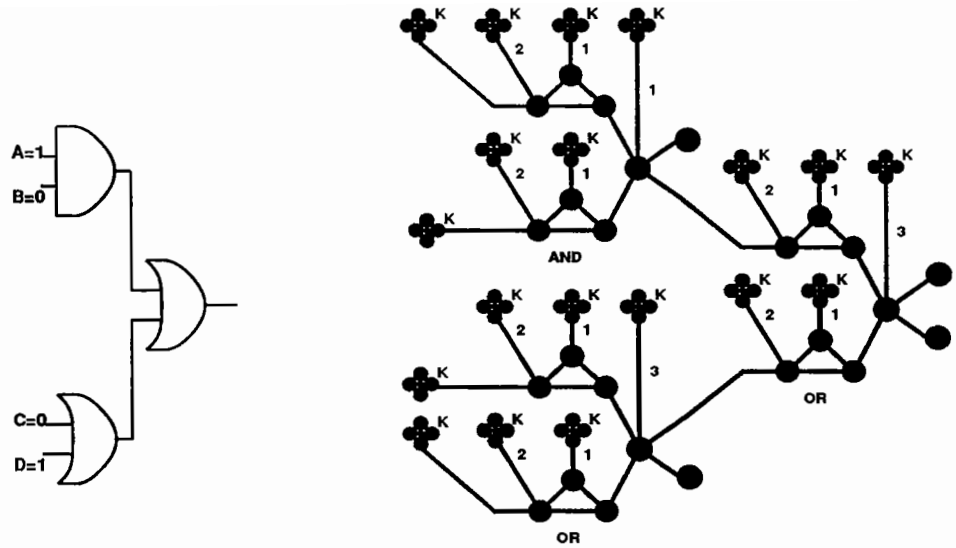


Figure 2: A Monotone Circuit and Its Equivalent Subgraph

when all inputs are initially zero and some are given the value 1. We now describe this construction in more detail.

We attach cliques, called *pull-up cliques*, of size K (K = 10) to certain gate vertices and put these cliques in the set Y. A weight w on an edge attached to a K-clique is a shorthand for w unit-weight edges each attached to different K-cliques. K is chosen so large that KL will only swap clique vertices after all gate vertices have been swapped. We attach zero, one or two vertices by a unit-weight edge to each output vertex of a gate subgraph so that each gate has outdegree two.

Since DR(C) is a monotone dual-rail circuit, DR(C) contains an equal number of AND and OR gates and exactly half of the circuit and gate inputs and gate outputs have value 1 when a computation is complete. Inputs, gates and outputs appear in complementary pairs in DR(C). We let $|C|$ be the number of paired gates in DR(C), and we let I be the number of paired circuit inputs. Let L ($L = 14$) be the number of vertices internal to a gate pair, i.e. not in attached cliques, and let D ($D = 16$) be the number of K-cliques attached to the internal vertices of a pair of AND and OR gates in the graph of DR(C). Let F be the number of paired vertices attached to outputs of gates in DR(C) so that the outdegree of a gate is always two.

In Figure 3 we show the initial partition for KL. All vertices of the graph associated with DR(C) are placed initially in X_0 except for the K-cliques. All of the attached K-cliques

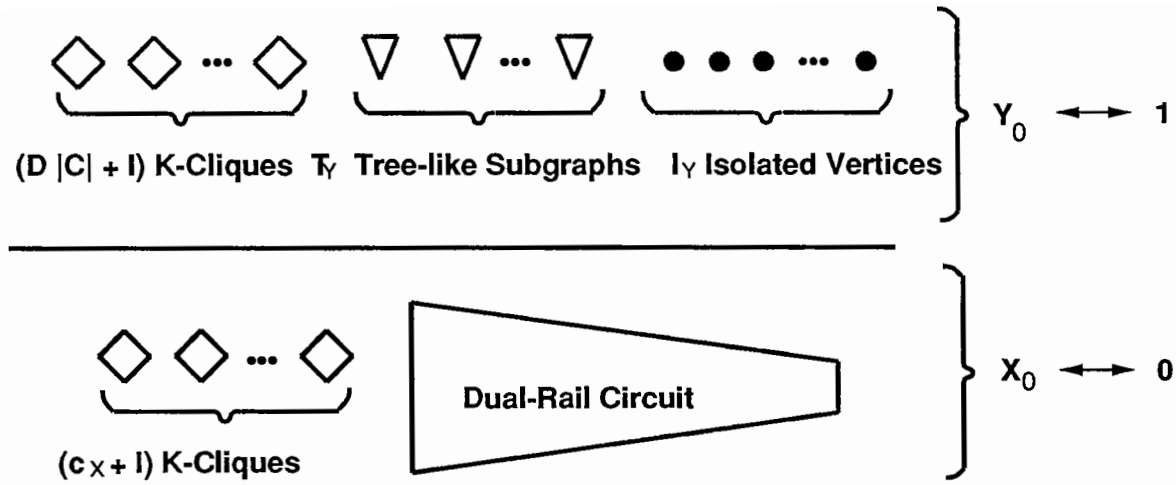


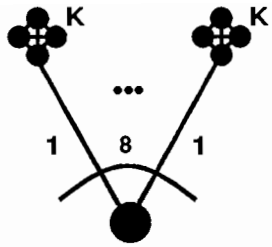
Figure 3: Schematic Representation of G

except those connected to inputs of DR(C) are placed in Y_0. Those connected to inputs that are assigned value 1 are placed in Y_0 and the rest are placed in X_0. The graph G also contains i_Y isolated vertices, c_X isolated K-cliques, and t_Y tree-like subgraphs. Each tree-like subgraph (shown in Figure 4) consists of one vertex connected to eight K-cliques. Initially the i_Y isolated vertices, the t_Y tree-like subgraphs, and the c_X balancing K-cliques are placed in Y_0, as shown in Figure 3. Here diamonds represent cliques, triangles represent the tree-like subgraphs, solid circles represent isolated vertices and the remaining object DR(C) contains the graph associated with dual-rail monotone circuit. Edges between these elements are not shown. The role of the various subgraphs will be described in more detail as we explain the action of KL.
We choose the following values for i_Y, t_Y, and c_X.

$$\begin{aligned} i_Y &= (L|C| + F)/2 \\ t_Y &= (L|C| + F)/2 \\ c_X &= D|C| + 8t_Y \end{aligned}$$

These assignments were chosen to result in equal numbers of vertices in each half of the partition.

The execution of KL on the graph G proceeds through three stages. Stage 0 corresponds to the initial partition. During Stage 1 subgraphs corresponding to gates in DR(C) are moved to Y if the gates of DR(C) assume value 1 on the given pattern of inputs. Isolated



(b)

Figure 4: Tree-Like Balancing Subgraphs

vertices from Y move to X to maintain a balance between the two sets. During Stage 2 all remaining vertices in the graph corresponding to DR(C) move to Y as the roots of tree subgraphs move in parallel. Finally, during Stage 3 all remaining vertices move to the other side. We show that the bisection width is locally minimal at the end of Stage 1.

Stage 0 We compute $bw_0(G)$, the initial bisection width of G, by using the fact that DR(C) is a dual-rail circuit. Ignoring K-cliques attached to inputs, each gate and its dual are connected to D K-cliques by unit-weight edges. Each of these edges contributes one to the initial bisection width $bw_0(G)$. I edges connected to K-cliques representing inputs of value 1 in DR(C) also contribute to $bw_0(G)$. No other edges cross the partition. Consequently,

$$bw_0(G) = D|C| + I$$

Stage 1 As described in section 2.1, KL swaps and freezes the pair of vertices on opposite sides of the partition that causes the largest reduction (or smallest increase) in the width of the bisection. The only vertices in X whose swapping can reduce the width of the bisection initially are those connected to inputs whose value is 1. Moving one of these by exchanging it with one of the i_Y isolated vertices in Y decreases the bisection width by one. This may change the conditions at an adjacent vertex so that swapping this vertex causes yet another reduction in the bisection width. Vertex swapping with isolated vertices continues until all gates that should eventually have value 1 do have value 1. Since DR(C) is a dual-rail circuit, at this time exactly half of the non-clique vertices in DR(C) have been swapped with i_Y isolated vertices from Y. At this point no more isolated vertices are available; any further swapping of a vertex in $DR(C) \cap X$ will cause the bisection width to increase.

At the end of Stage 1 the value of the circuit can be computed from the partition P by determining on which side of the partition the output vertex falls. We show later that the OR and AND gate subgraphs have been designed correctly; i.e gate subgraphs mimic the operation of gates and compute the correct circuit value of every gate.

Let us look now at the the bisection width $bw_1(G)$ of the graph at the end of Stage 1. Consider an OR gate in DR(C) and its dual AND gate. When both inputs to the OR gate are 0, the contribution of the gate to the bisection width is 9 while nothing is contributed by its dual AND gate. When the inputs to both gates are 0 and 1, their combined contribution to the bisection width is 9. When the two inputs to the OR gate are 1, all vertices in the equivalent subgraph have moved to Y and there is no contribution to the bisection width. Its dual AND gate has both inputs 0 and its contribution to the bisection width is 7. It follows that

$$bw_1(G) = 9|C| - 2n_{1,1}^{OR}$$

where $n_{1,1}^{OR}$ is the number OR gates in DR(C) that have both inputs 1.

Stage 2 The next gates in X to be moved are also from DR(C). To see this, observe that no vertex in DR(C) has degree more than 7; it cannot cause an increase of more than 7 in the bisection width. All other vertices in X are in the K-cliques, with K = 10, and would cause an increase of at least 9 in the bisection width. Each vertex in DR(C) is swapped with a vertex of degree 8 in a tree-like subgraph because they cause the smallest increase in the bisection width. Consequently, each swap of a remaining vertex with one from Y causes an increase in the bisection width of at least one. The bisection width bw_2 after all vertices have been moved, satisfies

$$bw_2(G) = 8t_Y + I$$

since t_Y tree vertices of degree eight have changed sides, I input edges in DR(C) are connected to K-cliques in X and no other edges cross the cut.

Stage 3 The only vertices available for moving at Stage 3 are the K-cliques. A clique is attached to at most one vertex in the rest of the graph. As a clique moves, the bisection width first increases and then decreases but the clique contributes positively to the bisection width until the clique has changed sides completely. Thus, we can guarantee that the bisection width during these moves never falls below bw_0.

Summarizing, the steps taken by KL cause the bisection width to decrease uniformly from bw_0 to bw_1, then to increase uniformly to bw_2, with $bw_2 > bw_0$, and then to decrease back to bw_0. We conclude that the best partition in the KL-neighborhood corresponds exactly

to the point when the value of DR(C) has been computed. The gate values of DR(C) can be computed in logspace from this partition by looping through all vertices of G, checking whether they are output vertices, and using the side of the partition as gate value.

Operation of Gate Subgraphs We now verify the correct operation of the gate subgraphs during Stage 1 of the KL algorithm. We demonstrate that gate subgraphs mimic the operation of gates.

In the KL heuristic, vertices are frozen once they have moved. Therefore, once the input arm vertices A, B, and C and the central vertex O have moved to X, they are unavailable for further swaps. As we have previously seen, the locally minimal bisection is assumed at the end of Stage 1. During Stage 1, we need only consider swaps of gate vertices in X with isolated vertices in Y, and only swaps that improve the partition are accepted. The various attached pull-up cliques will not move during Stage 1.

The *gate state* of a gate subgraph is the current assignment of its vertices to either X or Y. Once the local search algorithm starts operating on the initial partition, it will change vertex assignments by moving vertices between sets X and Y, thus causing *transitions* from one gate state to another. Special gate states are the *initial states*, where vertices are on the X side of the partition, with the exception of the pull-up cliques, and the *minimal states*, where no more positive *gain* (bisection-width improving) moves can be executed by the local search algorithm. The minimal states are reached at the end of Stage 1.

We show that a sequence of transitions starting from the initial state of a gate subgraph will eventually lead to a minimal state. This is done by examining all possible states and all possible state transitions that can occur during the course of the local search algorithm.

For simplicity, the AND and OR gate subgraphs are decomposed. As we mentioned above, every gate has two *input arms*, with three vertices that we label A, B, and C, as shown in Figure 5, columns 1 and 2. The gate also has a *central* vertex labeled O, as shown in Figure 5, columns 3, 4, and 5. Vertex C1 (C2) is the vertex C of the attached first (second) input arm, and vertex A1 (A2) is the vertex A of the attached first (second) output arm, or a single vertex, if the gate has less than two outputs. Depending on whether this gate is an AND or an OR gate, there are either one or three edges attaching the central vertex to its pull-up cliques.

Figure 5, column 1 shows the operation of the input arm when the central vertex is still in X, and an input I moves to Y. First vertex A, then B, and then C move to Y, all with a gain of 1. Figure 5, column 2 handles the special case where the input arm is connected to the central vertex O of an OR gate, whose other input and central vertex have already moved to Y. The vertex C of the input arm will stay in X, unless vertex A moves from X to

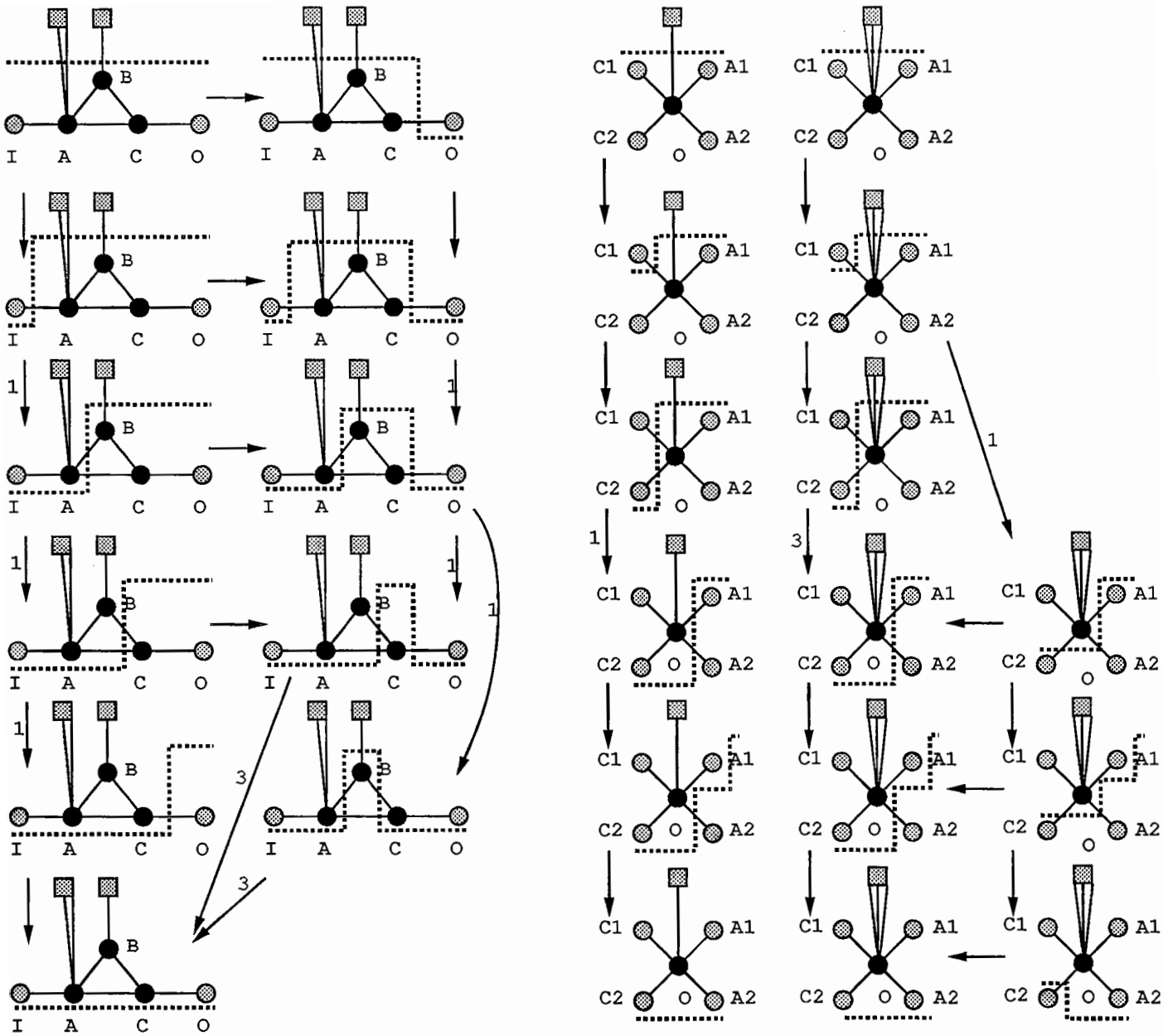


Figure 5: Operation of Gate Subgraphs

Y, because all possible moves have negative gain. As a consequence, gate subgraphs mimic the operation of gates in that a change in output does not cause the change of an input. In Figure 5, column 2, Vertex C in the 4th state, and vertex B in the 5th state move with gain 3 to the 6th state in column 1. All other vertices move with gain 1. Figure 5, column 3 shows the states of the central vertex of an AND gate. Note that it has only one pull-up clique attached to it. Vertex O moves to Y with gain 1 only when both vertices C1 and C2 have moved to Y. Note that the isolating effect of vertex C in the input arm now assures that the vertices A1 and A2, will never move before the central vertex 0 moves. Figure 5, columns 4 and 5 shows the states of the central vertex of an OR gate. Vertex O moves to Y with gain 1 when either C1 or C2 have moved to Y, and moves to Y with gain 3 when both vertices C1 and C2 have previously moved to Y.

All states but the minimal states allow moves with positive gain. Consequently improving swaps will be executed until a minimal state is reached. Thus our gate subgraphs operate correctly during Stage 1 of the KL local search algorithm. □

The above reduction remains correct when the move set is altered to allow single vertices be moved to the other side of the partition and then frozen. It follows that local search under the FM-N neighborhood is P-complete.

Theorem 3 *FM is P-complete.*

3.4 P-Hardness of the SA Heuristic

The "zero-temperature" version of graph partitioning by SA, ZT-SA, defined in section 2.2, is a type of probabilistic steepest-descent algorithm. We first show that ZT-SA is in BPP, so the error probability in finding a local minimum can be made arbitrarily small. We then show that any heuristic that accepts only swaps with positive gain is P-hard; it can solve the circuit value problem and is unlikely to be in NC. Instances of such heuristics are ZT-SA, *Mob* with mob size 1, and deterministic swap.

Theorem 4 *The ZT-SA heuristic is in BPP.*

Proof In the proof of Theorem 2 a boolean circuit C with n gates was transformed into a graph G with $|V| = O(n)$ vertices and $|E| = O(n)$ edges. The initial bisection width of G is therefore at most $|E|$. Let P_E be the probability that ZT-SA does not reach a local minimum after execution of a polynomial number $q(n)$ steps. We have to show that $P_E \leq 1/2 - \epsilon$, i.e. that the error probability is bounded away from $1/2$ by a constant.

ZT-SA has to execute at most $|E|$ moves that improve the bisection width. Divide the $q(n)$ time steps of the procedure into $|E|$ intervals of $m(n) = q(n)/|E|$ of time steps each. If the procedure fails to find the local minimum in time $q(n)$, then by the "pigeon-hole principle" at least one of these intervals does not contain a swap that improves the bisection width. Thus, the probability of an unsuccessful execution, P_E, is bounded above by the probability that a pair of vertices is not swapped in one or more of these $|E|$ intervals.

Since at each step until the local minimum is reached there is at least one vertex in each half of the partition which can be selected and swapped, it follows that the conditional probability of failing to make a successful swap on any given time step, conditioned on all prior actions, is at most $1 - 4/|V|^2$ since there are $[|V|/2]^2$ possible pairs to swap. The probability of failing to find a successful swap in a set of $m(n)$ contiguous time steps is thus at most $(1 - 4/|V|^2)^{m(n)}$. The probability of this event happening in one or more of the E intervals is an upper bound to P_E and satisfies the following inequality:

$$P_E \leq |E|(1 - 4/|V|^2)^{m(n)} \leq |E|e^{-4m(n)/|V|^2}$$

Thus, if we choose $q(n)$ for some constant ϵ so that

$$q(n) \geq \frac{|E|}{4}|V|^2[(-\ln(1/2 - \epsilon)) + \ln|E|]$$

then we have $P_E \leq 1/2 - \epsilon$. Since both $|V|$ and $|E|$ in our constructed graph G are proportional to n, it follows that $q(n)$ need grow only slightly faster than n^3. ZT-SA computes with bounded error probability the correct circuit value for C. □

Lemma 1 *Let H be a graph partitioning algorithm operating on the SWAP neighborhood. If H accepts only vertex swaps that improve the bisection width whenever such a swap is available, then H is P-hard.*

Proof We reduce DRMCVP to H by constructing graphs so that the steps in the local search mimic the computation by a monotone circuit when all inputs initially have value 0 and then some inputs are assigned value 1. We are given a fan-in/fan-out 2, dual-rail monotone circuit DR(C). The construction of the graph G and its initial partition is exactly the same as in the KL reduction, and is in logspace and deterministic.

As in the KL reduction, we have to show that a sequence of transitions starting from the initial state of a gate subgraph will eventually lead to a minimal state. Correct values for all gates in DR(C) correspond to a local minimum that H reaches. H cannot leave this local minimum.

The main difference with the KL reduction is that vertices are not frozen; they can move back and forth across the partition boundary. A vertex swap consists of a vertex move from X to Y and a vertex move from Y to X.

We know that H accepts only positive gain or bisection width improving swaps, but the exact rule for selecting vertices to swap is left unspecified. We have to show that the gate subgraphs continue to operate correctly as they move.

By correct operation we mean a) that at all instances of time the gate states are identical to those of the KL algorithm shown in Figure 5, b) that vertex moves from X to Y have gain 0, 1 or 3, c) that vertex moves from Y to X have gain 0 or -1, and d) that only positive gain swaps are accepted by H.

Initially this assumption holds. We again examine all possible states and all possible state transitions in Figure 5 that can occur during the execution of the local search algorithm. Vertices belonging to the cliques and tree-like subgraphs will not move because they cause large negative gains and there are no vertices to be moved with even larger positive gains to cause a vertex swap to be accepted. Vertex moves with gain 0 are due to solitary vertices and do not influence the gate subgraphs. The gate states and transitions of a move of a vertex in DR(C) from X to Y with gain 1 or 3 are those assumed by the KL algorithm and shown in Figure 5. Once in one of these states, the only vertex moves from Y to X have gain -1, and are the inverse of moves with gain 1.

We find that all successor states in Figure 5 are the ones in Figure 5. No other gate states and transitions are possible. Therefore our assumption that gate states are identical to the gate states of the KL algorithm and that vertex moves from X to Y have gain 0, 1 or 3, and vertex moves from Y to X have gain 0 or -1 continues to hold. All states but the minimal states allow moves with positive gain. Consequently improving swaps will be executed until a minimal state is reached. Thus our gate subgraphs operate correctly until the H algorithm reaches a local minimum. □

Note that we have implicitly assumed in the above proof that the gain of a vertex swap is the sum of of the vertex move gains, and shown correct operation of the subgraphs under this assumption. If the vertex moving from X to Y and the vertex moving from Y to X are connected by an edge, the gain of the swap is the sum of the gains minus 2. This cannot cause the acceptance of a swap that violates the assumption of correct gate subgraph operation. Thus the difference in how the gain of a swap is computed has no effect on the correct operation of the subgraphs.

Let us now apply lemma 1 to a number of heuristics operating on the SWAP neighborhood. These heuristics can be distinguished by the rule that selects swaps. As long as the heuristic accepts only swaps with positive gain, the heuristic will be P-hard by the above

lemma 1. In ZT-SA the vertices to be selected are chosen at random. The *Mob* heuristic chooses randomly only among those vertices with highest gain. When the mob size is fixed at 1 throughout an execution of *Mob*, the above lemma applies to *Mob*. Lemma 1 also applies to heuristics in P, where a deterministic rule, like lexicographic-first, guides the selection of swaps.

Theorem 5 *ZT-SA is P-hard.* Mob *with mob size fixed at 1 is P-hard. Deterministic swap is P-complete.*

The results that local search under the SWAP and FM neighborhoods are P-complete were obtained independently by Schäffer and Yannakakis [34].

4 The Connection Machine

The Connection Machine 2 (CM-2) is a massively parallel computer consisting of up to 64K moderately slow one-bit processors [1,21]. 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 [3] 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* heuristic was implemented in the C language with calls to Paris (Release 5.1), the high-level assembly language of the CM-2.

5 Implementation of *Mob*

Our heuristic starts with an initial random partition of the vertices of a graph. As explained above, it then forms two "pre-mobs" of vertices from each set of the partition on the basis of the effect of individual vertices on the bisection width as measured by the "gain" of a vertex. From each pre-mob a "mob" is selected, of a size determined by a "mob schedule." The vertices in mobs then swap sets. If the cost of the new bisection decreases, another mob of the same size is selected in the same manner and swapped. If the bisection width increases, the mob size is decreased. When the mob size reaches its minimum value, it is increased to the maximum value again and the process repeated.

Data Structures We experimented with several ways of implementing our heuristic on the Connection Machine. Initially we used one virtual processor per vertex and one per edge. However, we found that the network became badly congested when edge processors had to send messages simultaneously to the corresponding vertex processors so that the latter could compute gains. We also created a bottleneck when vertex processors sent messages to adjacent edges telling them the side of the partition their endpoints were now on. To avoid this congestion, we eliminated vertex processors and simulated them with edge processors.

Our algorithm is based on 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 bisection width of a partition as well as the gain of individual vertices, and also permits us to select and swap mobs easily. Blelloch reports that this data structure also works very well for other graph algorithms [3].

Computing Addresses of Twins A presorting phase is used to compute the addresses of edge duplicates, so that we need not make this computation on every exchange operation.

Bisection Width Every edge computes whether it crosses the cut. A reduction operation counts the number of crossing edges. Since every edge has a twin, this number is divided by two to obtain the bisection width.

Gain A "gain" is computed for each vertex that measures its effect on the bisection width if it changes side. The gain of a vertex is computed by summing the contribution of each edge adjacent to the vertex. Every edge computes whether or not it crosses the cut. A crossing edge contributes a gain of 1 to each vertex to which it is connected, a non-crossing edge contributes a gain of -1. 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.

Selecting A Mob Vertices that join the pre-mob are those with the largest gains. Since there are generally many vertices with the smallest gain in a pre-mob, we need a method to select between these vertices. All edge processors except the first in each group (which also serves as a vertex processor) are then idled and a forward additive scan operation on these vertices is used to rank vertices. Adaptive binary search then 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 side.

Exchange After every vertex has decided whether or not to swap sides, all edge processors become active and the first edge processor in a group (the simulated vertex processors)

communicates its side of the partition 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 correct side.

6 Results of Experiments

Much effort has been expended on developing good serial heuristics for the graph partitioning problem, and thus new heuristics should be calibrated against them. We have done this by running the *Mob* heuristic on two small graphs used by Johnson *et al.* [22]in their study of KL and SA. Any new heuristic should also provide internally consistent results over a variety of graphs of varying degrees and sizes. Our data confirm the internal consistency of the results provided by *Mob*.

No experimental results on large graphs have been published in the literature. We conjecture that the queue data structures that give the KL and FM algorithms asymptotically optimal serial running times should cause massive paging on a supercomputer with a carefully balanced memory hierarchy, such as a Cray. We expect *Mob* to perform very well on a serial supercomputer, since the algorithm minimizes global data movement. Our experiments include running times and bisection widths for KL. KL was partially parallelized on the CM-2: Cost and gain functions are computed in parallel, but the sequence of moves is left serial. We believe partitioning large graphs on a serial machine is an open research problem. We hope that the data that we provide will serve as a basis for the comparative evaluation of partitioning algorithms for other researchers.

Comparisons between KL, SA and *Mob* David Johnson supplied us with the two graphs, *r500* and *m1000*, used in [22]. Data on these graphs are given in Table 1. Reported there are the degrees, number of vertices and edges in these graphs and their bisection widths for random partitions and best-ever partitions over many runs with KL, SA and *Mob*. Also shown is the *ratio* B/R of the bisection width of the best-ever *Mob* partition to that of a random partition. The bisection width data for KL and SA are those reported by Johnson *et al.* The results for *Mob* were obtained by running our *Mob* heuristic on the Connection Machine. Clearly *Mob* gives results as good or better than SA and much better than KL when compared by the best bisection width ever recorded.

Table 2 compares the quality of the bisections obtained with KL to *Mob* on the basis of their execution times. We implemented KL on the Connection Machine by modifying our implementation for *Mob*, and gave KL an advantage by computing the bisection widths and vertex pair selected for a swap in parallel.

The first column labeled KL in Table 2 reports the percentage by which the *average* width of a bisection found by our implementation of KL exceeds the best-ever bisection size (with any heuristic) on *r500* in 100 runs with randomly chosen initial partitions and on *m1000* in 400 runs. The columns labeled 100, 1000, 2000 and 3000 report the same data for *Mob* when the number of iterations (one iteration is the swapping of two mobs) is that indicated by column header. It is a characteristic of *Mob* that the bisection width it discovers improves as the number of swaps increases while the bisection width discovered by KL improves only by choosing increasing numbers of random initial partitions, thereby increasing the odds that a good partition will be found.

Table 2 also reports the average execution time for one iteration of KL and for one and 2,000 iterations of *Mob* on an 8K-node CM-2 machine. The run times reflect the fact that the graphs are too small to fully use the Connection Machine.

Large Random Graphs To calibrate *Mob* further, we conducted experiments on randomly generated graphs with up to 1M vertices and 2M 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. SA and KL have prohibitive running times for such large graphs. Consequently, we measure the quality of our results primarily by comparing them among themselves, although we do report results given by KL on 16K-vertex graphs and one 32K-vertex graph.

The exact number of edges, vertices and average degree of the random graphs used in our experiments are given in Table 3. We used seven graphs of degree 4, and seven graphs of 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 degrees ranging from 3 to 16 to examine the effect of graph degree on the behavior of the *Mob* algorithm. The number of runs over which this data was averaged was at least 10 in every case.

Mob Schedule The mob schedule specifies how many vertices 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 10% of the number of vertices in the graph. The mob size is then decremented to 0 in 40 steps. The process is then repeated. *Mob* was always stopped after 4000 iterations. We found that this schedule worked well with graphs of varying size and degree.

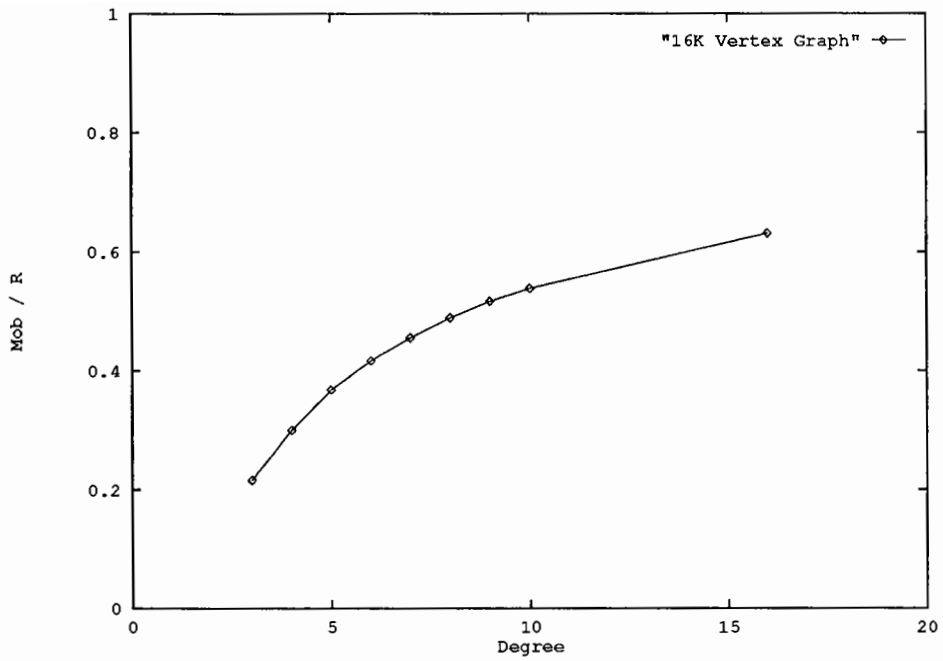


Figure 6: Best/random bisection widths for *Mob* as a function of degree

Summary of Results on Large Graphs Our experiments show that *Mob* has the following desirable properties:

a) The quality of solutions is consistently good.

b) The algorithm can produce good solutions in a short time.

c) The algorithm can produce increasingly better solutions with increasing time.

d) The running time of the algorithm grows logarithmically if the number of processors available is proportional to the graph size.

e) The algorithm uses the parallel CM-2 hardware well.

Consistent Results Table 4 shows the bisection widths produced by a random partition and the KL and *Mob* algorithms. The Mob/R column in this table reports the ratio of the best *Mob* bisection width to the average random bisection width. We see that the variation

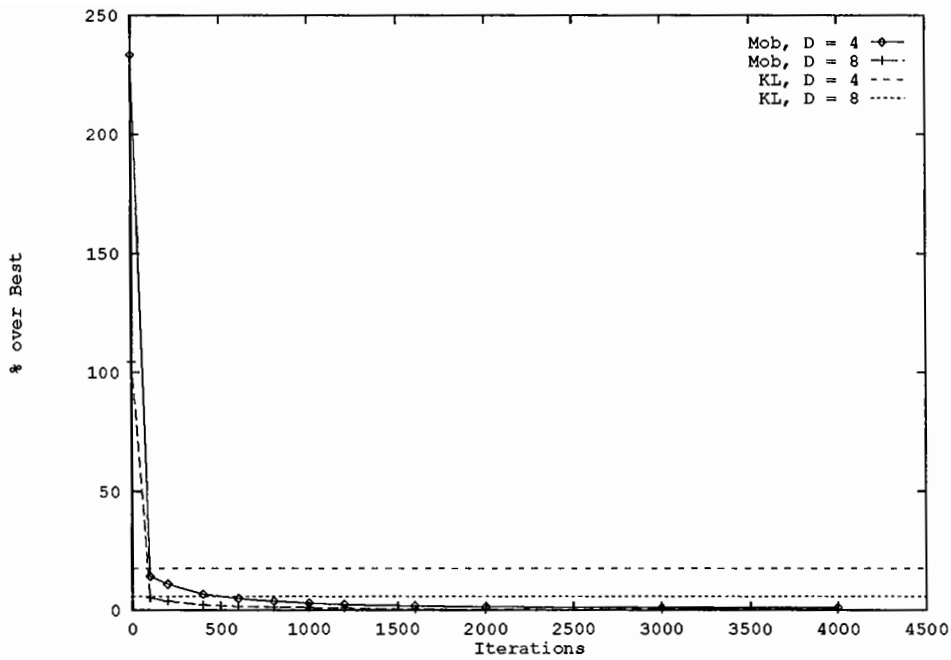


Figure 7: Convergence of *Mob*

in the ratio of best-ever bisection width to average bisection width is just about constant for graphs of constant degree independent of size. This is consistent with the theoretical bounds summarized by Bui [8]. As shown in Figure 6, these ratios rise with increasing degree toward an asymptote of 1.

Rates of Convergence of *Mob* Table 4 also reports the convergence of KL and *Mob* for an increasing number of iterations. The data in this table are bisection widths as percentages above the best-ever bisection width for these graphs. Again, *Mob* improves with the number of iterations. This data and the average bisection widths with KL are plotted in Figure 7.

Another measure of the consistency of *Mob*'s results is the number of iterations needed to reach a fixed percentage above the best-ever bisection width. As shown in Table 4, this number of iterations appears to be approximately constant over a wide range of graph widths for graphs of a fixed degree. Also, the number of iterations to achieve a given percentage decreases as the degree increases. Thus, very few iterations are needed on high-degree graphs.

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 for graph

partitioning described in Section 3 should serve as a base 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.

Computation Time We also experimented with graphs of different sizes on machines with varying numbers of real processors. The results of these experiments are reported in Table 5. We find that the time per iteration normalized by the number of virtual processors is nearly constant on the Connection Machine. That is, our heuristic runs in nearly constant time per edge. For the largest graphs with 2M edges we used 4M virtual processors. 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 bisections of increasingly larger graphs can be computed.

The absolute speed at which a bisection is produced by *Mob* is remarkable. On a 32K CM-2 it takes 520 seconds (less than 9 minutes) to find a bisection of a 1M-vertex, 2M-edge random graph that is within 2 percent of optimal.

7 Conclusions

We have shown here that the Kernighan-Lin graph-partitioning heuristic is P-complete. We have also shown that the zero-temperature version of simulated annealing is P-hard. Thus, it is unlikely that either heuristic can be parallelized. We have run a new graph-partitioning algorithm on the CM-2 Connection Machine to show that it is a massively parallel algorithm, is fast, can handle very large graphs, and gives results of a quality comparable to simulated annealing.

We believe that our logspace reductions can be extended to show that certain graph embedding algorithms used in VLSI placement and logic simulation are hard to parallelize and plan to adapt the *Mob* heuristic to replace these algorithms. The speed of the *Mob* heuristic should be adequate for the optimized placement of large (100,000 to 1,000,000 gates) circuits.

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. The authors would also like to thank David Johnson for supplying graphs against which the performance of our new heuristic was compared.

References

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

[2] P. Banerjee, M. H. Jones and J. S. Sargent, "Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. PDS-1, no. 1, pp. 91–106, January 1990.

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

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

[5] T. Bui, S. Chaudhuri, F. T. Leighton and M. Sipser, "Graph Bisection Algorithms with Good Average Case Behavior," in *25th Annual Symposium on Foundations of Computer Science*, pp. 181–192, 1984.

[6] T. Bui, S. Chaudhuri, F. T. Leighton and M. Sipser, "Graph Bisection Algorithms with Good Average Case Behavior," *Combinatorica*, vol. 7, no. 3, pp. 171–191, 1987.

[7] T. Bui, C. Heigham, C. Jones and T. Leighton, "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms," in *26th IEEE Design Automation Conference*, pp. 775–778, 1989.

[8] T. N. Bui, "On Bisecting Random Graphs," Dept. of Electrical Engineering and Computer Science, M.I.T., MS Thesis MIT/LCS/TR-287, Feb. 1983.

[9] E. C. Carlson and R. A. Rutenbar, "Mask Verification on the Connection Machine," in *25th IEEE Design Automation Conference*, pp. 134–140, 1988.

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

[11] S. -C. Chang and J. Ja'Ja', "Optimal Parallel Algorithms for River Routing," in *Proceedings International Conference on Parallel Processing*, pp. 9–13, 1988.

[12] S. -C. Chang and J. Ja'Ja', "Parallel Algorithms for Channel Routing in the Knock-Knee Model," in *Proceedings International Conference on Parallel Processing*, pp. 18–25, 1988.

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

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

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

[22] 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.

[23] D. S. Johnson, C. H. Papadimitriou and M. Yannakakis, "How Easy is Local Search," in *Journal of Computer and Systems Sciences*, pp. 79–100, 1988.

[24] 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.

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

[26] B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks," *IEEE Trans. Computers*, vol. 33, no. 5, pp. 438–446, May 1984.

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

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

[29] F. T. Leighton, "A Layout Strategy for VLSI Which is Provably Good," in *14th Annual ACM Symposium on Theory of Computing*, San Francisco, pp. 85–98, May 1982.

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

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

[32] J. E. Savage and M. G. Wloka, "A Parallel Algorithm for Channel Routing," in *Graph-Theoretic Concepts in Computer Science*, no. 344. Amsterdam: Lecture Notes in Computer Science, Springer-Verlag, pp. 288–301, June 1988.

[33] J. E. Savage and M. G. Wloka, "Parallel Constraint Graph Generation," in *Advanced Research in VSLI: Proceedings of the 1989 Decennial Caltech Conference* , Cambridge, MA, pp. 241–259, Mar. 1989.

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

[35] 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.

Table 1: Small random graphs and their bisection widths

Graph	V	E	d	Random	KL	SA	Mob	Mob/R
r500	500	1,196	4.784	594	222	206	206	.3468
m1000	1,000	2,496	4.992	1,246	475	451	450	.3616

Table 2: Convergence and execution time for KL and Mob on *r500*, *m1000*

	Quality in Relation to Iterations					Execution Times (Sec)		
	Bisection Quality (% Over Best)					8K CM-2		
Graph	KL	100	1000	2000	3000	KL	1	2000
r500	16.5	14.4	5.6	4.3	4.1	2.61	.00455	9.1
m1000	14.0	13.3	4.8	3.8	3.4	4.92	.00467	9.3

Table 3: 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
4.1M	1,048,576	2,082,998	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
8.512K	524,288	2,079,997	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 4: Behavior of Mob on large random graphs

	Graph Bisections				Bisection Quality (% Over Best)				
Graph	Random	KL	Mob	Mob/R	KL	100	1000	2000	3000
Degree 4									
4.16K	16,130	5,685	4,838	.2999	17.5	14.3	3.0	1.4	1.1
4.32K	32,224	11,477	9,824	.3049	16.8	13.2	3.2	2.8	1.0
4.64K	65,547	-	19,504	.2976	-	14.5	3.7	1.8	1.2
4.128K	131,173	-	39,010	.2974	-	14.7	4.2	1.8	0.9
4.256K	259,843	-	80,725	.3107	-	14.0	4.2	2.0	1.0
4.512K	520,538	-	156,294	.3003	-	14.4	4.8	2.1	1.3
4.1M	1,041,426	-	314,463	.3020	-	13.8	4.7	1.9	1.0
Degree 8									
8.8K	16,013	8,100	7,716	.4819	5.0	5.8	1.5	1.0	0.9
8.16K	32,479	16,778	15,873	.4887	5.7	5.2	1.1	0.4	0.3
8.32K	64,991	33,924	31,670	.4873	7.1	5.4	1.3	0.7	0.5
8.64K	129,876	-	63,539	.4892	-	5.0	1.1	0.4	0.2
8.128K	260,034	-	127,003	.4884	-	5.1	1.2	0.5	0.2
8.256K	520,041	-	253,804	.4880	-	5.2	1.4	0.6	0.3
8.512K	1,039,073	-	507,824	.4887	-	5.1	1.4	0.6	0.2
Variable degree									
3.16K	12,290	3,402	2,653	.2159	28.2	22.8	5.3	3.0	2.1
4.16K	16,130	5,685	4,838	.2999	17.5	14.3	3.0	1.4	1.1
5.16K	20,443	8,482	7,520	.3679	12.9	10.3	2.1	1.1	0.9
6.16K	24,554	11,246	10,234	.4170	9.9	7.3	1.4	0.6	0.4
7.16K	28,608	14,212	13,030	.4555	9.1	6.4	1.6	0.8	0.7
8.16K	32,479	16,778	15,873	.4887	5.7	5.2	1.1	0.4	0.3
9.16K	36,865	20,123	19,046	.5166	5.7	4.4	1.0	0.5	0.3
10.16K	40,971	23,319	22,068	.5386	5.7	4.3	1.3	0.8	0.7
16.16K	65,517	42,722	41,324	.6307	3.4	2.7	0.7	0.4	0.3

Table 5: CM-2 execution times (secs) for KL and Mob

	8K CM-2			16K CM-2		32K CM-2	
Degree 4							
Graph	1 KL	1	2000	1	2000	1	2000
4.16K	-	.0175	35.0	.0106	21.2	-	-
4.32K	-	.0331	70.2	.0188	37.6	.0100	20.0
4.64K	-	.0650	130.0	.0347	69.4	.0220	44.0
4.128K	-	.1262	252.4	.0660	132.0	.0361	72.2
4.256K	-	.2469	493.8	.1277	255.4	.0669	133.8
4.512K	-	-	-	.2674	534.8	.1316	267.2
4.1M	-	-	-	-	-	.2597	519.4
Degree 8							
8.8K	433	.0185	37.0	.0105	21.0	.0065	13
8.16K	1445	.0320	64.0	.0186	37.2	.0103	20.6
8.32K	4000	.0601	120.2	.0348	69.6	.0199	39.8
8.64K	-	.1230	246.0	.0730	146.0	.0360	72.0
8.128K	-	.2448	489.6	.1295	259.0	.0674	134.8
8.256K	-	-	-	.2610	522.0	.1298	259.6
8.512K	-	-	-	-	-	.2581	516.2
Variable Degree							
3.16K	-	.0173	34.6	.0103	20.6	-	-
4.16K	-	.0175	35.0	.0106	21.2	-	-
5.16K	983	.0330	66.0	.0190	38.0	.0120	24.0
6.16K	961	.0362	72.4	.0179	35.8	.0101	20.2
7.16K	1266	.0340	68.0	.0183	36.6	.0102	20.4
8.16K	1444	.0320	64.0	.0186	37.2	.0103	20.6
9.16K	940	.0610	122.0	.0331	66.2	.0191	38.2
10.16K	1741	.0660	132.0	.0329	65.8	.0192	38.4
16.16K	1998	.0637	127.4	.0351	70.2	.0203	40.6