Design and Performance of Parallel and Distributed Approximation Algorithms for Maxcut*

Steven Homer
Department of Computer Science and Center for Computational Science
Boston University
111 Cummington Street
Boston, MA 02215
email: homer@cs.bu.edu

Marcus Peinado
Institute for Algorithms and Scientific Computing
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven
53754 Sankt Augustin, Germany
email: peinado@gmd.de

*Research partially supported by the National Science Foundation under grants CCR-9103055 and CCR-9400229.

Proposed running head:

Approximation Algorithms for Maxcut

Author to whom proofs should be sent:

Steven Homer
Department of Computer Science
Boston University
111 Cummington St
Boston, MA 02215
email: homer@cs.bu.edu
phone: (617) 353-8927
fax: (617) 6457

Abstract: We develop and experiment with a new parallel algorithm to approximate the maximum weight cut in a weighted undirected graph. Our implementation starts with the recent (serial) algorithm of Goemans and Williamson for this problem. We consider several different versions of this algorithm, varying the interior-point part of the algorithm in order to optimize the parallel efficiency of our method. Our work aims for an efficient, practical formulation of the algorithm with close-to-optimal parallelization. We analyze our parallel algorithm in the LogP model and predict linear speedup for a wide range of the parameters. We have implemented the algorithm using the message passing interface (MPI) and run it on several parallel machines. In particular, we present performance measurements on the IBM SP2, the Connection Machine CM5, and a cluster of workstations. We observe that the measured speedups are predicted well by our analysis in the LogP model. Finally, we test our implementation on several large graphs (up to 13,000 vertices), particularly on large instances of the Ising model.

List of Symbols

ϵ	epsilon
ξ	xi
λ	lambda
α	alpha
Θ	Theta
$\mathbb{N}$	N (the natural numbers)
$\mathbb{R}$	R (the real numbers)

1 Introduction

Given a weighted undirected graph $G = (V, E)$, a set $S \subseteq V$ of vertices defines the cut $(S, V \setminus S)$. The size of the cut is the sum of the weights w_{ij} of all edges which connect a vertex in S with a vertex in $V \setminus S$, i.e. $cut(S) = \sum_{i \in S, j \in V \setminus S} w_{ij}$. The MAXCUT problem is the problem of finding a cut of maximum size in the input graph G. MAXCUT has applications in VLSI design [9, 8, 29, 6] and statistical physics [6]. MAXCUT is known to be NP-complete [24]. However, it can be approximated to within a constant factor by polynomial time algorithms. Indeed, the greedy algorithm of Sahni and Gonzalez [37] finds a cut whose size is guaranteed to lie within a factor of 0.5 of the size of the maximum cut. For more than 20 years this factor of 0.5 was the best polynomial time *performance guarantee* known for MAXCUT. A recent algorithm by Goemans and Williamson (GW) [16] is guaranteed to come to within a factor of 0.878 of the optimum. This breakthrough in the design of approximation algorithms has lead to improved approximation algorithms for other NP-complete problems [21, 15, 3, 13, 10]. The basis for the significant improvement are the more sophisticated techniques of positive semidefinite programming and randomized rounding. However, solving semidefinite programs is computationally expensive. Previous implementations of the algorithm on serial machines could only handle relatively small inputs (200 to 500 vertices). See Poljak and Rendl [31] for related work.

Our goals are twofold. Firstly, we want to establish the practical possibility of using the GW algorithm for much larger input graphs of thousands of vertices within realistic amounts of time. In order to achieve this goal, we make use of parallel computation techniques and consider different versions of the GW algorithm where the semidefinite program is solved using gradient-descent and interior-point methods. We analyze our algorithm using the LogP model. This model, first proposed by Culler et al. [11], provides a realistic platform with which to evaluate the efficiency of algorithms on parallel distributed memory machines. The LogP model seems to be the most accurate and explanatory model for analyzing the algorithms which we consider here. It considers point-to-point communication of processors over a network, taking into account system parameters which include message overhead, communication latency and network bandwidth, as well as the number of processors. We define this model more precisely in Sect. 3. More details of and motivation for this model can be found in [11].

Secondly, we want to determine if the theoretically superior approximation quality of the GW algorithm is matched by an improved performance on typical test inputs. For this purpose, we have also implemented the greedy algorithm and simulated annealing. We observe that the GW algorithm's solution quality is significantly better than that of the greedy algorithm, while it does not match the solution quality of simulated annealing. We consider modifications to the GW algorithm which improve its solution quality. We are especially interested in inputs which are derived from practical applications. In particular, we investigate whether the GW algorithm can compete with the Monte Carlo methods which are predominant in applications of MAXCUT in statistical physics. Independently of the actual cuts, the GW algorithm also produces an upper bound on the size of the optimal solution. This information is not provided by any of the simpler heuristics but can be quite useful when evaluating their performances.

1.1 Previous Work

A randomized version of the greedy algorithm of Sahni and Gonzalez [37] can be stated as follows: "Given a graph, generate cuts according to the uniform distribution". By linearity of expectation, the expected cut size under this distribution is half the sum of all edge weights and, thus, at least half the size of the optimum cut.

Goemans and Williamson [16] improve this ratio from 1/2 to 0.878 by generating cuts from a more sophisticated distribution: Each vertex is represented by an n-dimensional unit vector v_i. The first step of their algorithm is to find a vector configuration (that is, a set of n vectors in $\mathbb{R}^n$) which solves

$$\max Z_v = \frac{1}{2} \sum_{i<j} w_{ij}(1 - \langle v_i, v_j \rangle) \ \text{ subject to } \|v_i\|_2 = 1 \quad \forall i \in V \tag{1}$$

where $\langle ., . \rangle$ is the inner product. In general, we denote constrained optimization problems in the form

$$\max F = f(x) \ \text{ subject to } \ \text{cond}(x)$$

meaning that the goal is to find a solution x which maximizes (minimizes) $F = f(x)$ and satisfies the condition cond(x). A near-optimal solution for (1) can be found in polynomial time using a semidefinite programming algorithm and incomplete Cholesky decomposition.

The second step of the GW algorithm is to uniformly generate random hyperplanes through the origin. A hyperplane – given by its normal vector r – separates the vectors into the sets $L = \{v_i : \langle v_i, r \rangle \geq 0\}$ and $R = \{v_i : \langle v_i, r \rangle < 0\}$. This defines a cut. The analysis of Goemans and Williamson shows that the expected cut size under this distribution is at least 0.878 times the size of the optimum cut. Intuitively, an edge $\{i, j\}$ has the tendency to increase the angle between v_i and v_j in the optimal configuration. This increases the probability that the edge is cut, as the probability that two vectors are on different sides of a random hyperplane is proportional to the angle between them. The algorithm can be derandomized by means of the method of conditional expectations [16].

1.2 Outline

The computationally expensive part of the GW algorithm is finding a near-optimal solution of (1). In Sect. 2 we describe an interior-point algorithm for solving (1). Sect. 2 contains some technical details about interior-point algorithms and might be skipped by expert readers familiar with the primal-dual procedure presented in Nesterov and Nemirovskii [28]. Our implementation is a relatively straightforward parallelization of their method. Sect. 3 is the central section of this work. An alternative parallel algorithm, which, in practice, is more efficient for sparse graphs, is described in this section. It includes a detailed analysis of the parallel algorithm in the LogP model which shows that linear speedup can be achieved on most modern massively parallel machines. Furthermore, it describes an MPI based parallel implementation, displays performance measurements on several parallel machines, and compares these measurements with our theoretical predictions. Finally, Sect. 4 describes an experimental comparison of the approximation quality of the GW algorithm and of simulated annealing for several classes of graphs.

2 An Interior-Point Algorithm

We use $Y \succeq 0$ to signify that the matrix Y is symmetric and positive semidefinite. Let y_{ij} be the ij-entry of a matrix Y. Goemans and Williamson [16] show that program (1) is equivalent to the following semidefinite program.

$$\max Z_y = \frac{1}{2} \sum_{i<j} w_{ij}(1 - y_{ij}) \text{ subject to } Y \succeq 0 \text{ and } y_{ii} = 1 \ (i, j \in V). \tag{2}$$

A feasible solution of (2) can be transformed into a feasible solution of (1) with the same value via an incomplete Cholesky decomposition.

Semidefinite programming is a special case of convex programming which can be solved in polynomial time with ϵ-error. One allows small errors as exact optimal solutions might be non-rational. In the following we will simply speak of solving semidefinite programs (without mentioning the small error). Nesterov and Nemirovskii [28] present a number of interior-point algorithms for convex programming and prove polynomial-time bounds for them. The first interior-point algorithms were designed for linear programming by Karmarkar [22]. The name *interior point* refers to the fact that, unlike the simplex algorithm, these algorithms approach the optimum from the interior of the polyhedron of feasible solutions. Alizadeh [2] generalizes Ye's algorithm for linear programming [41] to semidefinite programming.

The algorithm described in this section is based on an interior point algorithm of Nesterov and Nemirovskii [28] which belongs to the class of *primal-dual potential reduction algorithms* [41]. Primal-dual algorithms solve the original (primal) problem and a second problem which is dual to it. Given a semidefinite program P, one can define a second semidefinite program D by means of a simple transformation and call it the *dual* program of P. In order to make such a definition useful, some facts similar to those known about duality in linear programming like *strong duality* or *complementary slackness* should be shown. In linear programming, a primal-dual pair of programs consists of a maximization and a minimization problem. Intuitively, strong duality states that the value of the objective function of the maximization problem is not larger than the value of the objective function of the minimization problem at any pair of feasible solutions of the respective problems. Furthermore, the primal and dual solutions are optimal if and only if the values of the two objective functions are equal. Complementary slackness states that the inner product $\langle x, s \rangle$ of any feasible solution x of the primal problem P and any feasible solution s of the dual problem D becomes zero if and only if x is optimal for P and s is optimal for D. These notions have been generalized to semidefinite programming by several authors [28, 2].

At each moment, the primal-dual algorithm maintains a strictly feasible solution x of P and a strictly feasible solution s of D. A *step* of the algorithm updates either the primal solution (primal step) or the dual solution (dual step) in a way which tends to reduce $\langle x, s \rangle$. When the inner product becomes sufficiently small, the primal and the dual solution are close to optimal.

In order to guarantee polynomial running time, the algorithm tries to minimize a potential function rather than the inner product itself. The potential function is chosen such that minimizing it implies driving complementary slackness to zero. In addition, it should be such that for any strictly feasible pair of solutions, either a primal step or a dual step will reduce its value by a constant amount. If the difference between the initial value of the potential function and a value which makes complementary slackness sufficiently small, is polynomially bounded, the algorithm will only need a polynomial number of steps.

After this outline of the general principle, we present now the details of the algorithm. The first step is to formulate a dual problem. Consider the space of symmetric $n \times n$ matrices (which we can identify with $\mathbb{R}^{\binom{n}{2}}$), and choose $\langle x, y \rangle = \mathrm{Tr}\{xy\}$ as the inner product on this space [28, Sect. 6.4.1], where $\mathrm{Tr}\{x\}$ is the trace of the matrix x. We consider the following pair of primal (P) and dual (D) problems [28, Def. 4.2.1]:

$$
\begin{array}{llllll}
(P): & \text{minimize } \langle I_n, x \rangle & \text{subject to} & w + x \succeq 0 \ , & x_{ij} = 0 \text{ for all } i \neq j \\
(D): & \text{minimize } \langle s, w \rangle & \text{subject to} & s \succeq 0 \ , & s_{ii} = 1 \text{ for all } 1 \leq i \leq n \ ,
\end{array}
$$

where $w = (w_{ij})_{ij}$ is the matrix of the edge weights of G, and I_n is the $n \times n$ identity matrix. Strong duality and complementary slackness for (P) and (D) follow from [28, Thm. 4.2.1]. Furthermore, it is easily seen that strictly feasible solutions for both (P) and (D) exist. Observe that minimizing $\langle s, w \rangle = \sum w_{ij} s_{ij}$ is the same as maximizing $\sum w_{ij}(1 - s_{ij})$. *Our algorithm will solve (D) and this will provide us with a solution of (2).* Interpretations of (P) as eigenvalue minimization problems can be found in [28, Sect. 6.6.4.3] and [16]. Similarly, Poljak and Rendl [32] have shown strong duality between (2) and an eigenvalue minimization problem which admits a semidefinite formulation (also cf. [28, 2, 16]). For reasons of consistency, we have stayed within the framework of Nesterov and Nemirovskii [28].

We have chosen the primal-dual potential reduction algorithm outlined in [28] to solve (P) and (D). The algorithm is summarized in Fig. 1. It maintains a pair (x, s), where the matrix x is a strictly feasible solution of (P) and the matrix s is a strictly feasible solution of (D).

The function $v(u)$ is derived from the potential function. The components of $v(u)$ are the logarithmic barrier $F(u) = -\ln \det u$ and a second term $\alpha(\langle s, u \rangle / \langle s, x \rangle - 1)$ which drives (x, s) towards a point at which the duality gap $\langle x, s \rangle$ is zero. The logarithmic barrier $F(u)$ ensures that the Newton steps which are used to update x and s will always remain in the interior of the positive semidefinite

Algorithm: primal-dual
Input: x (a strictly feasible solution for (P)) s (a strictly feasible solution for (D)) ϵ (the required accuracy)
WHILE $\langle x, s\rangle > \epsilon$ $\xi =$ argmin$\{\langle v'(x), h\rangle + \frac{1}{2}\langle v''(x)h, h\rangle \vert h \in L\}$ $\lambda = \sqrt{\langle v''(x)\xi, \xi\rangle}$ **IF** $\lambda > 0.34$ **THEN** $x := x + \frac{\xi}{1+\lambda}$ **ELSE** $s := \alpha^{-1}\langle s, x\rangle(-F'(x) - F''(x)\xi)$
where $v(u) = F(u) + \alpha\left(\frac{\langle s,u\rangle}{\langle s,x\rangle} - 1\right)$, $F(x) = -\ln\det x, \qquad \alpha = n + 1.05\sqrt{n}.$

Figure 1: The basic primal-dual procedure from [28, p. 140]. $F'(X)$ and $F''(x)$ denote the vector of first partial derivatives (gradient) and the Hessian matrix of second partial derivatives of F at point x. The notation for v is similar. The algorithm stops after $O(\sqrt{n})$ iterations.

cone. In other words, incorporating $F(u)$ (which approaches infinity near the boundary of the positive semidefinite cone) into the objective function of the unconstrained optimization method used to determine the next solution, ensures that this solution will always remain a positive definite matrix.

The algorithm uses Newton's method to minimize $v(u)$. In particular, ξ is the direction in which the second-order Taylor expansion of $v(u)$ at x (projected to the feasible plane) is minimized. The decrease of v and of the potential function is at least the Newton decrement λ. Nesterov and Nemirovskii show [28, Proposition 4.5.2] that as long as λ is large enough ($\lambda > 0.34$), a primal step in the direction of ξ is guaranteed to decrease the potential function by at least a constant amount. Otherwise ($\lambda \leq 0.34$), a dual step is guaranteed to achieve this goal. After $O(\sqrt{n})$ steps the potential function and, therefore, the duality gap are sufficiently small.

The computational cost of each step is dominated by the cost of computing the Hessian matrix of second derivatives $v''(x) = F''(x) = x^{-1} \circ x^{-1}$ and inverting it [28, Sect. 6.4.1], where $a \circ b$ denotes the Hadamard (componentwise) product of two matrices a and b. The matrix inversions are needed to compute ξ and the direction of the dual step. Note that it is not necessary to compute the entire Hessian matrix because all but the n diagonal entries of x are constant. The matrix inversions involved lead to a complexity of $\Theta(n^3)$ operations per step and $\Theta(n^{3.5})$ operations for the entire algorithm.

We have implemented the same interior-point algorithm in serial on a workstation and in parallel on the Connection Machine CM5. The serial implementation is written in C and uses LAPACK routines [4] for the time critical matrix inversions. Due to the fairly high complexity of the algorithm, the running times become quite large for graphs with more than 500 vertices (Tab. 1). The running time for a 1000 vertex graph is about one day on an SGI Indy (100MHz MIPS R4000 processor, IRIX 5.0, gcc -O2). However, similar behavior is displayed by other existing codes. Goemans and Williamson [16] report running times of 4692 seconds $\approx$ 77 minutes on a SUN Sparc 1 for the widely used code of Rendl et al. [34] and 200 vertex random graphs. The corresponding time for our algorithm on an SGI Indi is about 3.3 minutes (Tab. 1). For comparison, the MIPS R4000 processor is only about 5.5 times faster than the Sparc IPC processor (as measured by the SPECfp-92 benchmark: 61 for R4000 vs. 11.1 for Sparc IPC). While this comparison omits some detail, it does indicate that our serial implementation – which is the basis for evaluating the performance of our parallel implementations – is quite efficient. We have used random graphs here not for their intrinsic interest but rather to provide a comparison with the running times reported in [16].

The parallel program was implemented on the following CM5: 32 processors, CMOST version 7.4; 32MBytes and four vector units per node; nodes running at 32MHz. The same machine was used for the implementations described in Sect. 3. The parallel implementation is written in C* [38]. The running time of the algorithm is completely dominated by the cost of the matrix inversions. Very efficient solutions for this problem exist [20]. Using the LU routines from the CMSSL library [39] (version 3.2.1), our algorithm achieves speedups of more than 31 (over one-node CM5) on a 32-node CM5. Table 1 compares the running times of the parallel and serial implementations.

3 First Derivative Methods

The interior-point algorithm described in the previous chapter is well suited for dense graphs. However, its complexity of $\Theta(n^3)$ operations per step leads to long running times for large graphs. This is the case even if the input graph is sparse. Alizadeh [2] notes that the known interior-point algorithms for semidefinite programming do not take advantage of sparse inputs. (Recently Klein and Lu [25] have developed a different technique to handle sparse graphs using these methods.) Driven by our goal to test the quality of the approximation scheme of Goemans and Williamson on large instances of graphs derived from practical applications, we considered alternative ways of obtaining an optimal vector configuration. The first derivative method described in this section involves $O(en + n^2)$ operations per step – which is $O(n^2)$ for sparse graphs. Thus, we trade the worst-case guarantee of a polynomial running time for a method which can take full advantage of the sparseness of the graphs which are derived from typical applications like VLSI design or statistical physics. In practice, this method is fast and efficient and allows us to solve very large problem instances. We have tested algorithms based on different first derivative methods on a variety of different graphs with up to 13,000 vertices. In every single run we have observed fast convergence to within the required accuracy.

First derivative methods for unconstrained optimization (e.g. gradient-descent or conjugate-gradient) [27, 30, 33] try to minimize a differentiable objective function $f : \mathbb{R}^n \to \mathbb{R}$ by taking a sequence of steps based only on the value of the function and its gradient. This process terminates when the value of the objective function is sufficiently small. The simplest example is the gradient-descent method:

Algorithm: gradient descent$_\alpha(f)$
choose initial feasible solution x
WHILE $f(x)$ too large
$\quad x := x + \alpha \bigtriangledown f(x)$

where $\bigtriangledown f(x)$ is the gradient of f at point x, and $\alpha < 0$ is an appropriately chosen constant. The conjugate-gradient method [33, 26] is another example. For simplicity, we will describe the algorithm in terms of gradient descent. However, as we will parallelize the evaluation of the objective function and the gradient, our parallelization and analysis do not depend on any particular method for unconstrained optimization. Using the strong duality property of our problem, we can test the stopping condition by computing the duality gap between $f(x)$ and the corresponding value for the dual problem.

3.1 The objective function and the gradient

Problem (1), which is a constrained optimization problem, must be reformulated as an unconstrained optimization problem if the methods just stated are to be applied. Maximizing Z_v is equivalent to minimizing $\bar{Z}_v = \sum_{i<j} w_{ij}\langle v_i, v_j\rangle$ since $Z_v = (\sum w_{ij} - \bar{Z}_v)/2$. Given a vector x, let $\|x\| = \sqrt{\langle x, x\rangle}$ be its length. We choose the function

$$g(v_1, \ldots, v_n) = \sum_{i<j} w_{ij} \frac{\langle v_i, v_j\rangle}{\|v_i\|\|v_j\|} \tag{3}$$

as our objective function. Clearly, g is identical with $\bar{Z}_v$ on the unit sphere. Furthermore, g is independent of the norms of the v_i. This allows us to drop the side condition $v_i \in S_n$. We prove in the appendix that g has no local minima. The potential problem of saddle points (which do exist) can be alleviated by small perturbations.

Computing gradients is straightforward and we give only the final result. Let v_i^k denote the k-th component of the i-th vector. Then, assuming $\|v_i\| = 1$ ($\forall i \in V$):

$$\frac{\partial g}{\partial v_i^k} = v_i^k\langle v_i, C_i\rangle - C_{ik} \tag{4}$$

where C is the matrix formed by the column vectors

$$C_i = \sum_{j \in \mathrm{adj}(i)} w_{ij} v_j, \tag{5}$$

where $\mathrm{adj}(i) = \{j \in [n] : w_{ij} \neq 0\}$, $[n] = \{1, \ldots, n\}$, and C_{ik} denotes the k-th component of C_i.

The value of g can be computed as follows:

Algorithm: g	operations
1. Compute C as in (5).	$2en$
2. $T_i = \langle v_i, C_i \rangle$ for all $i \in [n]$.	$2n^2$
3. $g = \sum_i T_i$	$n - 1$

The second column (operations) counts the number of floating-point operations. The computation of the gradient $\bigtriangledown g$ is quite similar:

Algorithm: $\bigtriangledown g$	operations
1. Compute T_i as before	$2en + 2n^2$
2. $d_i^k = v_i^k T_i - C_{ik}$	$2n^2$

Apart from requiring evaluations of g and $\bigtriangledown g$, the usual unconstrained optimization methods update the current solution V by multiplying a correcting term C by a constant α and adding the result to V. Namely, let $V := V + \alpha C$. This update operation requires $2n^2$ floating-point operations as V and C are $n \times n$ matrices.

Finally, after $V = (v_i)$ has been updated, each v_i must be normalized ($v_i := v_i / \|v_i\|$) at a total cost of $3n^2$ floating-point operations. This normalization is unrelated to the optimization algorithm and is merely a result of our attempt to save operations in the computation of the gradient. The normalization relieves us from having to compute norms there. The total number of operations per step is $\Theta(en + n^2)$, and the space requirement is $\Theta(n^2)$.

3.2 The Parallelization and its Analysis

As already mentioned, we parallelize the basic operations used by the unconstrained optimization algorithms. In particular, we will present and analyze parallel algorithms for evaluating the objective function and gradient.

The rest of this section will focus on sparse graphs, graphs with $e = o(n^2)$ edges. This case presents obstacles since no efficient general off-the-shelf solutions for sparse graphs are available (which are better than for arbitrary graphs). No single type of operation dominates the running time – as did the matrix inversion operation in the previous section. Instead, a number of different $\Theta(n^2)$ operations must be efficiently parallelized. The practically relevant applications for MAXCUT (graphs derived from VLSI applications [40, 6] and the Ising model) are typically extremely sparse (for example, $|E| \approx 2|V|$ for most of the VLSI graphs in Tab. 5).

The first question to be addressed when parallelizing the computation of the gradient on a distributed memory parallel machine is the data layout, that is the distribution of the data among the local memories of the processors. Since for sparse graphs the memory requirements are dominated by the matrices $V = (v_i)_{i\in[n]}$ and C (size $\Theta(n^2)$), we focus on these two matrices. The other variables used by the algorithm have size $O(e + n)$ which is small compared to the space required for V and C.

We choose to partition the matrices along the rows. In terms of the algorithm, this means that each processor q is assigned a set S_q of components of the n-dimensional vectors v_i. Each processor q stores locally the components S_q of all vectors v_i and, similarly, all C_{ik} for $k \in S_q$. Thus, only the reduction operations $\langle .,. \rangle$ (inner product) and $\|.\|$ (Euclidean norm) require communication.

Figure 2 shows the parallelized components of the gradient-descent algorithm based on row partitioning. The parallel running time t_p with p processors can be divided into the time E_p spent on performing computations and the communication time C_p such that $t_p = E_p + C_p$. Table 2 compares the number of serial operations from Sect. 1.1 with the number of parallel operations from Fig. 2. Clearly, in all cases $E_p \leq E_1/p$. Therefore,

$$t_p = E_p + C_p \leq \frac{E_1}{p} + C_p \tag{6}$$

3.3 The Communication Cost

All communication operations of our algorithm are of the same type: In the language of Karp et al. [23], they are sequences of n “broadcast-with-combining” operations – one for each v_i ($i \in [n]$). Recall that broadcast-with-combining is a communication operation in which each processor contributes an initial value x_i. At the end of the operation each processor receives the result of combining (or reducing) all x_i by a simple operation (e.g. addition). We require this operation to be associative.

Assign to each processor q a set S_q of components such that $|S_{q_i}| \geq |S_{q_j}| - 1$ for all processors q_i, q_j.

	operations per processor
Algorithm: parallel g	
1. **For all** q **in parallel**:	
$\forall i \in [n], k \in S_q$: $C_{ik} = \sum_{j \in \mathrm{adj}(i)} w_{ij} v_j^k$	$2en/p$
$\forall i \in [n]$: $T_i^{(q)} = \sum_{k \in S_q} v_i^k C_{ik}$	$2n^2/p - n$
2. Compute $T = \sum_q T^{(q)}$ using global communication.	GLOBAL COMM.
3. Return $\sum_i T_i$	$n - 1$
Algorithm: parallel $\bigtriangledown g$:	
1. Execute steps 1 and 2 of *parallel g*	$\frac{2(en+n^2)}{p} - n$ + GL. COMM.
2. **For all** q **in parallel**:	
$\forall i \in [n], k \in S_q$: $d_i^k = v_i^k T_i - C_{ik}$	$2n^2/p$
Algorithm: parallel normalize:	
1. **For all** q **in parallel**:	
$\forall i \in [n]$: $T_i^{(q)} = \sum_{k \in S_q} v_i^k v_i^k$	$2n^2/p - n$
2. Compute $T = \sum_q T^{(q)}$ using global comm.	GLOBAL COMM.
3. $\forall i \in [n], k \in S_q$: $v_i^k = v_i^k / \sqrt{T_i}$	$n^2/p + n$
Algorithm: parallel update:	
4. **For all** q **in parallel**:	
$\forall i \in [n], k \in S_q$: $v_i^k = v_i^k + \alpha d_i^k$	$2n^2/p$

Figure 2: The parallel versions of the operations. Every processor q stores two vectors T and $T^{(q)}$. T_i and $T_i^{(q)}$ denote the i-th components of these vectors.

Karp et al. [23] present an algorithm for a single broadcast-with-combining operation which is optimal in the LogP model. The LogP model [11] uses only four parameters to describe the network: an upper bound L (*latency*) on the time needed by a small message to travel across the network from its source to its destination; the time (*overhead*) o a processor is engaged in sending or receiving a message; the minimum time interval between consecutive message transmissions or receptions (*gap*) g, and the number of processors p. The inverse of g is the per-processor bandwidth for short messages. On many machines, the per processor bandwidth for long messages is several orders of magnitude higher than $1/g$. The LogGP model [1] extends the LogP model by introducing G, the *gap per byte* for long messages as an additional parameter. The LogP model reduces to the postal model of Bar-Noy and Kipnis [5] (also see Karp [23]) if $o = 0$ and the time is rescaled such that $g = 1$.

Bar-Noy and Kipnis [5] describe several algorithms for sequences of $m \in \mathbb{N}$ broadcast (without combining) operations in the postal model. One algorithm (PACK) reduces this problem to the simple broadcast problem (by considering all m data items as a single large data item), and uses algorithm BCAST of [5] for the single broadcast. The running time of this algorithm is at most (cf. Bar-Noy and Kipnis [5, Cor. 8]):

$$\frac{2(m+\lambda-1)\log p}{\log(2+(\lambda-1)/m)} + 2(m+\lambda+1) < 2(m+\lambda)(1+\log p) \tag{7}$$

where λ, the latency parameter, is the ratio of the time it takes for a message to reach its destination and the time needed by the originator of the message to send it. In the following, we will work with the simpler term on the right-hand side of (7). As m will be large, the right-hand side is a close approximation of the left-hand side. We have selected algorithm PACK because it gathers data in long messages. Bar-Noy and Kipnis [5] describe alternative algorithms which send only short messages and which are faster in the postal model (which does not distinguish between bandwidths for short and long messages). However, the LogGP model (and in practice) algorithm PACK is clearly superior for realistic parameter choices.

The algorithm is easily generalized to broadcast-with-combining by first combining the values by executing the broadcast algorithm `in reverse' [23]. The running time is twice the value of (7). Thus, combining (7) and (6), noting that in each gradient-descent step, there are two calls to the communication algorithm (in `gradient' and in `normalize'), and that the data elements are eight byte floating point numbers, one obtains:

Fact 1 *In the LogGP model with $o = 0$, and $p \leq n$, the running time of each step of the gradient-descent algorithm is at most*

$$t_p = \frac{E_1}{p} + 8(8Gm + L)(1 + \lceil \log p \rceil) \ . \tag{8}$$

The postal model and the LogP model ignore the network structure. A few words about this assumption with respect to our application are in order. The requirements of the algorithm just described are clear: The network must be able to simulate the degree-d broadcast tree, where in our application $d > 1$ should be a small constant. Most common network architectures (e.g. binary tree, hypercube, mesh, cube connected cycles, butterfly) are easily verified to have degree-d spanning trees and, thus, satisfy the requirement.

The situation is quite different if the processors (workstations) are connected by a flat bus (ethernet). While a single message trivially solves the broadcast problem, the combination problem requires $p - 1$ sequential messages as the root processor needs information from each of the other $p - 1$ processors and only one message can be transmitted at any given time. Thus, there is an upper and lower bound of $C_p = pgm$ for the time needed for a sequence of m broadcast-with-combine operations on p processors where g is the time a single message blocks the network.

Fact 2 *For a flat bus (ethernet) network, the running time of each step of the gradient-descent algorithm is at most*

$$t_p = \frac{E_1}{p} + 2pgm \tag{9}$$

Figure 3 displays the resulting predicted speedups for reasonable parameter choices (compare Sect. 3.4 in this work and Sect. 5.2 in [11]).

To illustrate the difference between the two cases, we derive expressions for the number of processors p that can be used before the efficiency $X(p) = t_1/(pt_p)$ falls below a given constant $\alpha \in (0, 1)$, that is we consider $p_\alpha(n) = \max\{p \in \mathbb{N} : \mathrm{X(p)} \geq 1 - \alpha\}$. By (6) $X(p) \geq \frac{E_1}{E_1 + pC_p}$, which is required to be at least $1 - \alpha$. Thus, substituting the expressions for E_1 and C_p derived above and assuming $e = \Theta(n)$ we obtain:

Fact 3 *For our algorithm, $p_\alpha = \Theta(n)$ in the LogP model. For a flat bus/ethernet, $p_\alpha = \Theta(\sqrt{n})$.*

These results show that the algorithm scales linearly (in n) for the more sophisticated network types like binary trees and hypercubes. For the flat bus, however, p may only grow proportional to $\sqrt{n}$.

Table 1: Running times (in minutes) of the serial (on an SGI Indy) and the parallel implementation (on a 32-node CM5) of the interior-point method on random graphs with n vertices and edge probability 0.5.

n	200	400	600	800	1000
serial	3.3	35	154	450	982
parallel	1.5	5	12	21	40

Table 2: The cost of the serial and of the parallel versions. The term `bc-with-comb' stands for broadcast-with-combining and is defined in Sect. 3.3.

	objective fct	gradient	normalize	update
E_1 (serial)	$2(en+n^2)+n-1$	$2(en+2n^2)$	$3n^2$	$2n^2$
E_p (par.)	$2(en+n^2)/p-1$	$2(en+2n^2)/p-n$	$3n^2/p$	$2n^2/p$
C_p (comm.)	n bc-with-comb.	n bc-with-comb.	n bc-with-comb.	0

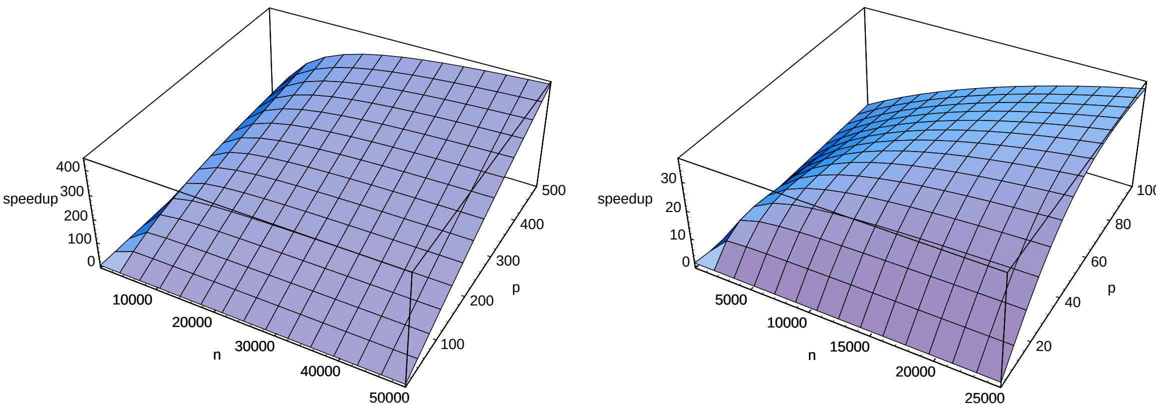

Figure 3: The speedups predicted by our performance model for larger graphs and machines. Left: LogGP with parameters $G = 2.8 \cdot 10^{-8}$ sec/byte $= 2.26 \cdot 10^{-7}$ sec/double, $L = 4.4 \cdot 10^{-5}$ sec, $d = 2$, $E_1 = (2ne + 7n^2) \cdot 10^{-7}$ sec. Right: ethernet with parameters $g = 1/50,000$ sec/double and $E_1 = (3ne + 11n^2) \cdot 10^{-6}$ sec.

3.4 Implementation

We have implemented and run the algorithm on a variety of parallel machines, in particular a Connection Machine CM5, an IBM SP2, and a cluster of workstations. The CM5 is described in Sect. 2. The cluster of workstations consisted of 16 Sparc 2 stations (40MHz clock rate), running Solaris, and linked by an ethernet (which was free from outside traffic). The IBM SP2 (IBM RS/6000 SP) had 34 nodes. The nodes were RS/6000 based POWER2 thin nodes, running at 66.7 MHz. Each node was equipped with a data cache of 32 kBtyes, an instruction cache of 64 kBytes, and 128 MBytes of random access memory. The nodes were running A.I.X. 4.1.4 and the IBM AIX Parallel Environment (PE) Version 2. An implementation of the Message Passing Interface (MPI) forms part of the PE. The nodes were linked by IBMs SP Switch.

The program was written in C and directly based on the parallel algorithm of Fig. 2. We used the message passing interface (MPI) [14] for the global communication operations. In addition, we optimized our implementation for the CM5, hand coding the instructions for the vector units and using the CMMD library [38] (version 3.3) instead of MPI. This allowed us to treat each of the four vector units in each CM5 node as a separate processor. Finally, we used UNIX sockets to implement the communication operations on the cluster of workstations.

Table 3 displays measured running times for the CM5 and the SP2. The inputs are two dimensional Ising grid graphs. We have measured the running times for only one class of graphs because the parallel performance (i.e. the ratio between t_1 and t_p ($p > 1$)) depends only on the numbers of vertices and edges, but not on the particular structure of the graph. The structure of the graph influences only the number of gradient-descent steps – which is the same for both the parallel and the serial version. A technical problem arises when trying to compute the speedup.The larger graphs do not fit into the memory of a single processor. Thus, we cannot measure t_1 directly. We estimate this time by running the program on a small number of processors, measuring the execution time for the parallelized parts of the program (i.e. ignoring communication and non-parallel code) and multiplying this time by the number of processors used. The result is a lower bound on t_1 which yields a lower bound on the speedup.

Figure 4 compares the measured speedups on the different machines with those predicted by the LogP model. As predicted by the analysis in the previous section, the speedups for the SP2 and

Table 3: Performance data for the CM5 and SP2 implementations: $|V|$ and $|E|$ are the number of vertices and edges in the input graph. The running times t_p with p processors (vector units for the CM5) are measured in seconds per gradient-descent step.

		CM5			SP2		
$\lvert V\rvert$	$\lvert E\rvert$	t_1	t_{128}	speedup	t_1	t_{32}	speedup
1600	3200	27.8	0.26	107	2.3	0.09	26
2500	5000	68	0.60	113	6.8	0.25	28
4900	9800	260	2.21	118	30.5	1.01	30
6400	12800	445	3.67	121	52	1.72	30
10000	20000	out of memory			127	4.11	31
12100	24200	out of memory			186	5.98	31

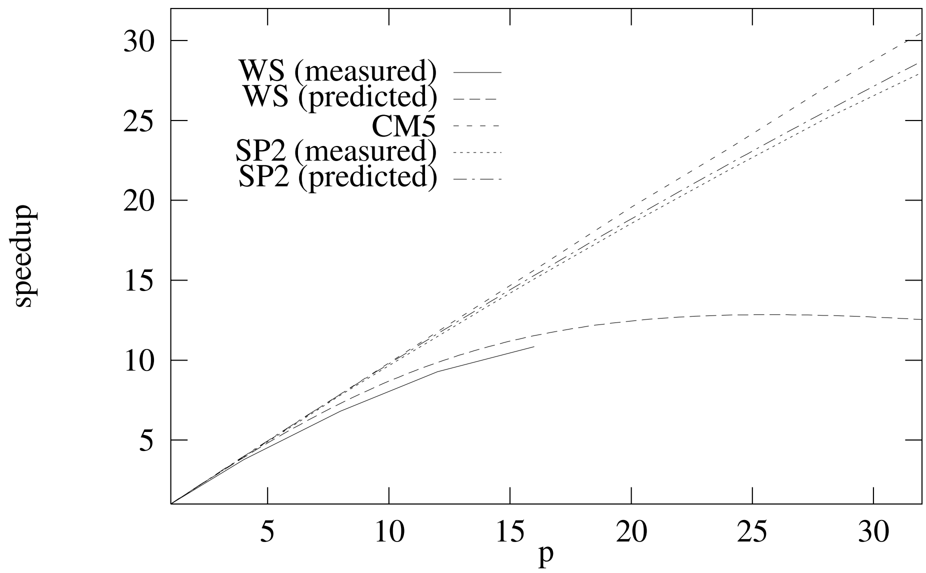

Figure 4: Speedups of our parallelization of the minimization algorithm on different machines. $|V| = 2500$, $|E| = 5000$

CM5 are far better than for the workstations (WS) linked by an ethernet. The predicted speedup for the workstations (WS predicted) is based on (9), except that we used a communication term of $4pgm$ (instead of $2pgm$) to account for the suboptimal broadcast routine in our implementation. We measured $E_1 = 110$ seconds and estimate $g = 1/50,000$ seconds per double precision float. (We estimate that about one third of the 10 Megabits per second capacity of a standard ethernet is available at the level of UNIX socket read/write operations, which corresponds to approximately $50,000$ double precision floats per second.) Despite this rather crude estimate, the figure shows a good match between the measured curve and the prediction.

We have estimated the LogGP parameters for MPI on the SP2 by timing the relevant communication operations. Figure 3 displays our parameter estimates and the predicted speedups based on (8). The curve `SP2 predicted' in Fig. 4 was obtained using the parameters of Fig. 3. The difference between this prediction and the measured speedups is less than three percent. This coincidence provides some evidence for the validity of our model. Figure 5 shows how the speedups improve as the graph size increases. In this figure, the term *processor* refers to a CM5 vector unit.

Table 4 compares the numbers of iterations and total running times on sparse graphs of the interior-point method of Sect. 2 and the algorithm described in this section. Both programs were run on a 32-node CM5 until the duality gap was smaller than 0.1%. The inputs were sparse random graphs with edge probability $10/n$. The gradient-descent program is several orders of magnitude faster. This is mostly due to the shorter time per iteration. However, it is interesting to observe that in spite of its simpler (faster) computations, the first derivative method uses less iterations than the algorithm of Sect. 2. In light of the remarks at the end of Sect. 2, we observe furthermore, that this favorable comparison is not due to potential inefficiencies of our interior-point algorithm and implementation.

The step size α (cf. page 12) was determined by an adaptive scheme which decreases α whenever a step would lead to a deterioration and otherwise increases α by a factor $1 + \epsilon$ with $\epsilon = 0.02$. In practice, this scheme kept the algorithm from spending much time near saddle points. Testing the stopping condition by computing the gap between the current value of the objective function and the corresponding value for the dual problem involves computation of the largest eigenvalue of a dense matrix. As this operation is fairly expensive, we did not test the duality gap during the running time of the algorithm. Instead, we fixed the number of steps to between 200 and 700 (depending on the graph

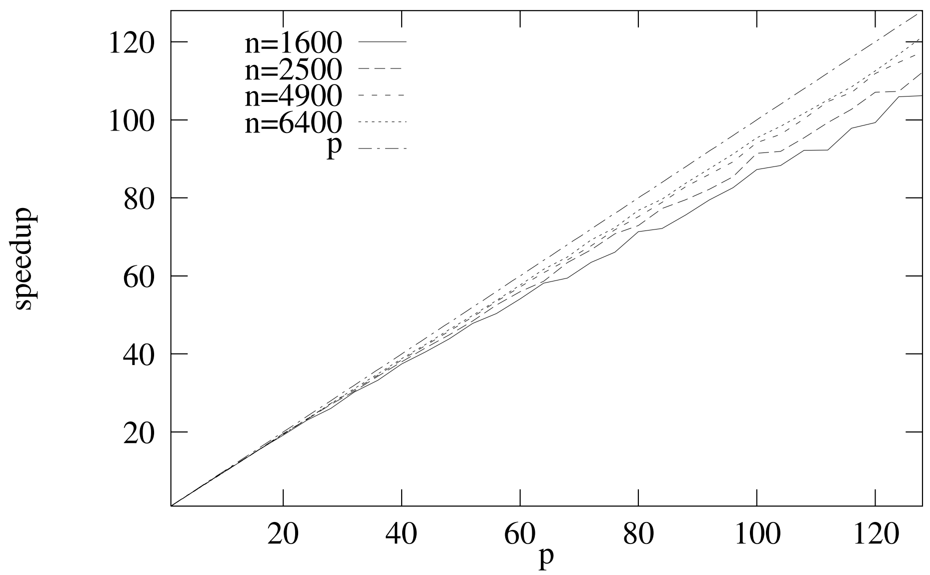

Figure 5: Speedups on the CM5: $n = |V|$ is the number of vertices in the input graph; p is the number of processors (vector units).

Table 4: Total running times and iteration counts on sparse random graphs (cf. Sect. 4) on a 32-node CM5 of the interior-point algorithm of Sect. 2 and the first derivative method described in this section.

	interior-point		gradient-descent	
$\lvert V\rvert$	iterations	time	iterations	time
200	111	2 min.	40	< 1 sec.
400	161	6 min.	45	1 sec.
600	201	15 min.	50	5 sec.
800	231	22 min.	55	9 sec.
1000	251	54 min.	60	15 sec.

size) and stored the information necessary to compute the eigenvalue gap for the solution returned by the algorithm. This allowed us to verify that the solution was within the required accuracy after the algorithm had terminated. We always observed that the gap was far smaller than the 0.1% target we had set and that usually even a far smaller number of iterations would have sufficed.

The vector units required special consideration. As far as the parallelization is concerned, we have treated the vector units as normal processors. In particular, the components of the v_i are split evenly among the $4p$ vector units and each vector unit works on its components as described in Fig. 2. All steps of the algorithm except the first step in computing the objective function involve operations on long regular vectors. They can be directly adapted to the vector units. Step 1 in the computation of the objective function and the gradient (Fig. 2), the only step which depends on the input graph, is different. When the input graph is sparse, we assume that it is given in adjacency list representation. The input graphs can be arbitrary and so can the lengths of the adjacency lists of the individual vertices. In particular, many of the adjacency lists can be so short (e.g. length = 2) that much of the benefit of the vector units is lost. One way to solve this problem is to preprocess the input graph and transform it into edge representation where the graph is given by the list of all its edges. Now, the vector units can operate on one long vector, the vector of all edges. However, this approach leads to read-after-write pipeline hazards if the distance in the edge list between any two edges which are incident on the same vertex is smaller than the vector length. This problem can be solved for most edges by preprocessing the edge list. Depending on the input graph, there can be edges for which such a `schedule' does not exist. However, these edges belong to a set of adjacency lists whose size is bounded by the vector length which is a small constant. These adjacency lists are processed individually.

The last step in the GW algorithm is to convert the vector configuration into an actual cut. This is achieved by generating a random vector r on the sphere ($\Theta(n)$ operations) and computing $\langle r, v_i \rangle$ for all i ($\Theta(n^2)$ operations). Given the data layout used above, these steps are easily parallelized.

4 Solution Quality

In this last section, we examine how well the GW algorithm approximates the maximum cut size for a variety of graphs with several thousand vertices. While 0.878 is the *worst case* performance guarantee, we are now interested in the approximation quality one can expect in practice when running the algorithm on typical input graphs. Furthermore, we provide an experimental evaluation of the algorithm on graphs with negative edge weights – a case which is not covered by the 0.878 performance guarantee of the algorithm. By way of comparison, we have also implemented simulated annealing and the randomized greedy algorithm.

Previous experimental work with the GW algorithm [16, 7, 31, 12] has been limited to much smaller inputs. Our parallel implementation makes it possible to compare the trends reported previously with results obtained on much larger graphs.

4.1 The Graphs and the Algorithms

We have tested the algorithm on the following graphs:

1. **Graphs derived from circuit design problems:** It has been observed [9, 8, 29, 6] that one of the phases (the layer assignment problem) in the design process for VLSI chips and printed circuit boards can be reduced to the MAXCUT problem. We have run the GW algorithm on ten graphs derived from VLSI problems [40].

2. **Graphs derived from the Ising model:** The Ising model is used in statistical physics to describe magnetism. A reduction from the problem of finding the ground states (states of minimal energy) in an Ising spin glass system to MAXCUT is well known. We model each elementary magnet as a $\{0,1\}$-variable and consider only nearest neighbor interactions and interactions with an external magnetic field. The reduction transforms such a system into a graph which has one vertex for each elementary magnet. There is one further vertex for the external magnetic field. By means of the reduction, the cut size found by the algorithm translates into an upper bound on the energy of the ground states and the size of the positive semidefinite solution translates into a lower bound.

3. **Sparse random graphs:** We have tested the algorithm on sparse random graphs in which the edge probability is set to $p = 10/n$, where $n = |V|$. This corresponds to random graph class C in Goemans and Williamson [16].

The first two graph classes are derived from real world applications of the MAXCUT problem. The third class is a popular test instance.

We have implemented simulated annealing and the randomized greedy algorithm as a basis for comparisons. The randomized greedy algorithm of Sahni [37] has been mentioned in Sect. 1.1. In addition to what is described there, we do greedy improvements of the random cuts. The application to MAXCUT is straightforward. The set of states is the set of cuts. The objective function is the cut size. The possible transitions are those which change the cut by at most one vertex (cf. [19] for a comprehensive study of simulated annealing and its application to the MAXCUT problem).

4.2 Experimental Results

In this section, we report on the results of running the algorithms on the graphs described in the previous section. The results for the GW algorithm as well as all running times we report were obtained by running our gradient-descent implementation on a 32-node Connection Machine CM5. All running times are reported in minutes. Simulated annealing and the randomized greedy algorithm were run on SGI Indy workstations. We took care to give approximately the same computational resources to the three algorithms – making adjustments in the running times for the different number of processors and processor speeds. Our basis was the running time on the CM5 of GW – which was dominated by the time needed to find an optimal vector configuration. This time was multiplied by the number of processors and by a factor accounting for the relative speed of the CM5 Sparc processors compared to the SGI Indy workstation.

Space limitations have led us to include only three data tables here. The conclusions we draw from them are consistent with many other experiments we have run and could not include here. Table 5 shows the results for the VLSI graphs. The table displays the cut sizes found by simulated annealing ($\mathrm{cut}_{\mathrm{SA}}$), the randomized greedy algorithm ($\mathrm{cut}_{\mathrm{RG}}$), and the GW algorithm. The results for simulated annealing are the best cuts found over five runs of 10^7 annealing steps each. The results for randomized greedy are the maximum cuts found over $20,000$ independent runs. The column $\mathbf{E}\mathrm{cut}_{\mathrm{GW}}$

contains the expected size of a single random cut as described in Goemans and Williamson [16]. Two observations are apparent: On one hand, the random experiment of Goemans and Williamson produces cuts larger than the randomized greedy algorithm is likely to find even after a large number of iterations (cf. Sect. 1.1 for the relationship between the two algorithms). On the other hand, the cuts found by simulated annealing are somewhat larger than those found by the GW algorithm. These two observations hold for a wide range of different graph classes (we did not find a single counter example), and are also supported by observations made previously about simulated annealing and the randomized greedy algorithm. Johnson et al. [19] observe for the related GRAPH PARTITIONING problem, that the randomized greedy algorithm performs far worse than simulated annealing. Berry and Goldberg [7] perform a similar experimental comparison of several heuristics – including simulated annealing.

While it seems clear that the simplest version of the GW algorithm cannot compete with simulated annealing, it is also clear that there are simple modifications to it which will improve its performance in practice. Probably the simplest such scheme is to find an optimal vector configuration and then generate many independent cuts by generating independently many hyperplanes [16]. This is an analogue to the iterated randomized greedy algorithm. While the iterated randomized greedy algorithm samples cuts from the uniform distribution, the iterated GW algorithm samples cuts from the distribution induced by the semidefinite program which gives larger weight to larger cuts.

Just as taking independent samples under the uniform distribution (iterated randomized greedy) is not the best way to generate large cuts, one might suspect that there are more effective schemes than generating cuts by independent random hyperplanes. One possibility is to add the same kind of Metropolis filter that is used in standard simulated annealing to the procedure which generates random hyperplanes. The result is a simulated annealing version of the basic random experiment of Goemans and Williamson. The moves (neighborhood) of such an annealing algorithm could be defined as small random perturbations of a current hyperplane. Objective functions and cooling schedules could be the same as in normal simulated annealing.

Column $\mathrm{cut}_{\mathrm{GWSA}}$ in Tab. 5 shows the results for the simulated annealing version of the GW algorithm. Now the algorithm does not only clearly win against simulated annealing – it even finds the (known) optimal cuts for each graph. The time needed to solve the positive semidefinite program

Table 5: Results for graphs (V, E) derived from circuit layout problems. UB is the upper bound of the GW algorithm, and $\mathrm{cut}_{\mathrm{RG}}$, $\mathrm{cut}_{\mathrm{SA}}$, $\mathrm{cut}_{\mathrm{GW}}$, and cut_{GWSA} are the cut sizes found by randomized greedy, simulated annealing, GW, and an extension to GW, respectively. The column t_{conv} displays the time spent to find a near optimal vector configuration and t_{cut} is the time spent by GWSA deriving cuts from it. See text for details.

graph	$\lvert V\rvert$	$\lvert E\rvert$	UB	$\mathbf{E}\mathrm{cut}_{\mathrm{GW}}$	$\mathrm{cut}_{\mathrm{RG}}$	$\mathrm{cut}_{\mathrm{SA}}$	$\mathrm{cut}_{\mathrm{GWSA}}$	t_{conv}	t_{cut}
via.c1n	828	1446	6183	5969	4110	6048	6150	0.75	0.17
via.c2n	980	1776	7118	6860	4968	7037	7098	1.00	0.25
via.c3n	1327	2482	6943	6590	4186	6844	6898	1.67	0.42
via.c4n	1366	2608	10111	9860	6156	9900	10098	1.80	0.33
via.c5n	1202	2235	8002	7657	4902	7920	7956	1.48	3.00
via.c1y	829	1750	7798	7535	6102	7746	7746	0.27	0.08
via.c2y	981	4206	8279	7985	6252	8226	8226	0.35	0.10
via.c3y	1328	2845	9585	9110	6550	9502	9502	0.58	0.12
via.c4y	1367	2916	12563	12332	8850	12516	12516	0.62	0.12
via.c5y	1203	2558	10333	9980	6954	10248	10248	0.50	0.12

is displayed under t_{conv}. The time given to the GW simulated annealing scheme is displayed under t_{cut}.

In addition to finding cuts, the GW algorithm provides one piece of information which none of the simple heuristics is able to find: an upper bound on the maximum cut size in the graph. Column UB displays the upper bounds which were derived from the dual solutions. Our corresponding primal and dual approximations of the optimum are within 0.05% of each other and therefore within 0.05% of the true upper bound.

The advantage our simulated annealing version of the GW algorithm had over standard simulated annealing on the ten VLSI graphs is not universal. Tables 6 and 7 show examples of graph distributions on which simulated annealing wins. Both distributions are quite similar to the distributions on which Johnson et al. [19] have also observed simulated annealing to outperform the Kernighan-Lin heuristic: sparse random graphs.

Table 6 shows the results for the second graph class described in the previous section (3D Ising graphs). Each graph corresponds to a regular three dimensional $20 \times 20 \times 20$ grid whose nearest neighbor interactions are drawn from the standard normal distribution. The parameter h which corresponds to the strength of an external magnetic field is varied between 0 and 4. Even though the 0.878 performance guarantee does not apply to these graphs due to the presence of negative edge weights, most of the gaps we observe are within that bound. However, this is not the case for small values of h.

The results shown in Tab. 7 are for sparse random graphs. The gap size increases with the graph size and comes close to the 0.878 guarantee. This observation is consistent with the trend visible in Goemans and Williamson [16].

Tables 6 and 7 show that simulated annealing outperforms the simulated annealing version of the GW algorithm on sparse random graphs. One reason is the generally observed good performance of simulated annealing on sparse random graphs. However, the results are also influenced by some low-level implementation issues. Simulated annealing, due to its simplicity can easily be implemented such that it takes advantage of the sparseness of a graph. In particular, each annealing step can be executed in time proportional to the degree of the vertex being moved across the cut. However, each step of our GWSA implementation takes time $\Theta(n)$ as the inner product of two n-dimensional vectors

Table 6: Results of running the algorithm on graphs derived from three dimensional $20 \times 20 \times 20$ Ising systems as described in the text. $|V| = 8001$, $t_{\mathrm{conv}} \approx 50$min., $t_{\mathrm{cut}} \approx 15$min.

h	UB	$\mathrm{cut}_{\mathrm{gw}}$	$\mathrm{cut}_{\mathrm{SA}}$
0.0	7488	6252.4	6727.46
0.4	9360	8053.0	8537.75
0.8	11485	9996.2	10612.70
1.2	13872	12304.9	12898.70
1.6	16215	14804.1	15339.80
2.0	18764	17509.8	17920.70
2.4	21412	20309.0	20632.20
2.8	24151	23232.0	23462.00
3.2	26967	26157.6	26366.60
3.6	29850	29221.2	29343.80
4.0	32792	32300.7	32381.30

Table 7: Results of running the algorithm on sparse random graphs (edge probability $p = 10/n$).

$\|V\|$	$\|E\|$	UB	$\mathrm{cut}_{\mathrm{GW}}$	$\mathrm{cut}_{\mathrm{SA}}$	t_{cut}
1000	5048	3934.4	3603	3686	1.32
2000	9964	7819.8	7114	7307	2.83
3000	14984	11790.2	10625	10994	4.92
4000	19948	15728.7	14162	14684	6.83
5000	24816	19586.5	17578	18225	9.17
6000	29880	23601.5	21196	21937	11.75
7000	35120	27728.6	24904	25763	14.80
8000	39675	31427.4	28210	29068	17.98

must be computed. Thus, if both algorithms are given the same amount of time, simulated annealing can perform $\Theta(n)$ times more steps than GWSA. If, on the other hand, both algorithms are run for the same number of steps, GWSA finds larger cuts than standard simulated annealing. Similarly, if the input graphs become dense, the advantage of simulated annealing evaporates. Overall, we draw the following conclusions from our experiments:

- The simplest version of GW is a clear improvement over the randomized greedy algorithm. However, it cannot compete with standard heuristics like simulated annealing.
- We only begin to address a more fundamental question about the average case performance of the GW algorithm: Can the basic random experiment which underlies the GW algorithm improve performance on commonly studied graph distributions when used as a primitive of a more complex heuristic? A definitive answer lies outside the scope of this paper. Our results for the VLSI graphs show that there is some potential.

5 Conclusions

We have studied ways to parallelize the approximation algorithm for MAXCUT by Goemans and Williamson. An interior-point method is quite efficient for dense input graphs. Parallelizing it amounts to parallelizing matrix inversion. The first derivative method we have studied is, in practice, significantly faster for sparse input graphs. We have presented a detailed analysis of the resulting parallel algorithm in a distributed memory model. We prove linear speedup in the LogP model. We compare the behavior of our parallel implementations on different machines with the results of our analysis. We show that given an efficient communication algorithm and implementation as well as high performance parallel machines, the algorithm can be used to solve very large problems with thousands of vertices.

Finally, we have used the implementations to compare the approximation quality of the GW algorithm with two other standard algorithms: simulated annealing and the randomized greedy algorithm. We conclude that, overall, simulated annealing appears to find larger cuts given comparable amounts of time. However, we have identified one class of graphs on which the GW algorithm works significantly better than simulated annealing.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, 1995.

[2] F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. To appear in *SIAM Journal on Optimization*. Preliminary version in *Proceedings of the 2nd Mathematical Programming Society Conference on Integer Programming and Combinatorial Optimization*. 1992.

[3] N. Alon and N. Kahale. Approximating the independence number via the θ-function. Manuscript, August 1994.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 1.0*. SIAM, 1992.

[5] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message passing systems. In *Proc. of SPAA*, pages 13–22, 1992.

[6] F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt. An application of combinatorial optimization to statistical optimization and circuit layout design. *Operations Research*, 36(3):493–513, 1988.

[7] J. Berry and M. Goldberg. Path optimization and near-greedy analysis for graph partitioning: An empirical study. Manuscript. Preliminary version in Proceedings of SODA'95, 1995.

[8] K. C. Chang and D. H. Du. Efficient algorithms for layer assignment problems. *IEEE Transactions on Computer-Aided Design*, CAD-6(1):67–78, 1987.

[9] R. Chen, Y. Kajitani, and S. Chan. A graph-theoretic via minimization algorithm for two-layer printed circuit boards. *IEEE Transactions on Circuits and Systems*, CAS-30(5):284–299, 1983.

[10] B. Chor and M. Sudan. A geometric approach to betweenness. In *Proceedings of the Third Annual European Symposium on Algorithms*, pages 227–237, Corfu, Greece, Sept. 1995.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, 1993.

[12] C. Delorme and S. Poljak. The performance of an eigenvalue bound on the max-cut problem in some classes of graphs. *Discrete Mathematics*, 111:145–156, 1993.

[13] U. Feige and M. Goemans. Approximating the value of two prover proof systems, with applications to MAX 2SAT and MAX DICUT. In *Proceedings of the Third Isreal Symposium on Theory and Computing Systems*, 1995.

[14] M. P. I. Forum. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[15] A. Frieze and M. Jerrum. Improved approximation algorithms for MAX-k-CUT and MAX BISECTION. *Algorithmica*, 1996. (to appear). Preliminary version in Proc. of the Sixth IPCO Conference, Springer-Verlag LNCS 920, pages 1–13.

[16] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, Nov. 1995. Preliminary version in Proc. 26th ACM Symposium on the Theory of Computing. 1994.

[17] P. M. Gruber and J. M. Wills, editors. *Convex Geometry*, chapter Mathematical Programming and Convex Geometry. North-Holland, 1993.

[18] http://www.mcs.anl.gov:80/home/lusk/mpich.

[19] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; Part I, graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[20] S. L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4:133–172, 1987.

[21] D. Karger, R. Motwani, and M. Sudan. Approximate graph coloring by semidefinite programming. In *Proceedings 35th IEEE Symposium on the Foundations of Computer Science*, pages 2–13, 1994.

[22] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[23] R. Karp, A. Sahay, and E. Santos. Optimal broadcast and summation in the LogP model. Technical Report CSD-92-721, University of California, Berkeley, 1992.

[24] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[25] P. Klein and H.-I. Lu. Efficient approximation algorithms for semidefinite programs arising from maxcut and coloring. In *Proc. 28th ACM Symposium on the Theory of Computing*, pages 338–347, 1996.

[26] J. Kowalik and M. R. Osborne. *Methods for Unconstrained Optimization Problems*. Elsevier, 1968.

[27] W. Murray. *Numerical Methods for Unconstrained Optimization*. Academic Press, 1972.

[28] Y. Nesterov and A. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. SIAM, 1994.

[29] R. Y. Pinter. Optimal layer assignment for interconnect. *J. VLSI Comput. Syst.*, 1:123–137, 1984.

[30] E. Polak. *Computational Methods in Optimization*. Academic Press, 1971.

[31] S. Poljak and F. Rendl. Solving the max-cut problem using eigenvalues. Technical Report 91735-OR, Universität Bonn, 1991.

[32] S. Poljak and F. Rendl. Nonpolyhedral relaxations of graph-bisection problems. Technical Report 92–55, DIMACS, 1992.

[33] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in Fortran*. Cambridge University Press, second edition, 1992.

[34] F. Rendl, R. Vanderbei, and H. Wolkowicz. Interior point methods for max-min eigenvalue problems. Technical Report 264, Technische Universität Graz, 1993.

[35] M. Rosenlicht. *Introduction to Analysis*. Scott, Foresman & Co., 1968.

[36] H. L. Royden. *Real Analysis*. Macmillan, 1968.

[37] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23:555–565, 1976.

[38] Thinking Machines Corporation. *Connection Machine CM5 technical summary*, November 1993.

[39] Thinking Machines Corporation. *CMSSL for CM Fortran*, April 1994. Version 3.2.

[40] D. P. Williamson. Personal Communication.

[41] Y. Ye. An $O(n^3L)$ potential reduction algorithm for linear programming. *Mathematical Programming*, 50:239–258, 1991.

6 Appendix

In general, the main danger of using a gradient-descent method is that of converging to a suboptimal solution (that is, a local minimum). In this section, we prove that any local minimum that exists has to be a global minimum, a point in which the objective function has its optimum value. The main idea of the proof is to map the space of vector configurations into a space of positive semidefinite matrices and to show for each point V in the vectors' world that it can only be a local minimizer if its corresponding positive semidefinite matrix is also a local minimizer. The step from vectors to positive semidefinite matrices is the first step in the polynomial time algorithm to find a maximum of (1) described in Goemans and Williamson [16]. For $i, j \in V$, let $y_{ij} = \langle v_i, v_j \rangle$, $Y = (y_{ij})_{i,j \in V}$ and

$$Z_y = \frac{1}{2} \sum_{i<j} w_{ij}(1 - y_{ij}). \tag{10}$$

The symmetric matrix Y is positive semidefinite. It is easy to see that $\max Z_v$ (subject to $v_i \in S_n$) is equal to $\max Z_y$ (subject to Y symmetric and positive semidefinite and $y_{ii} = 1$ for $i \in V$). Any local minimizer in the positive semidefinite matrix world is in fact a global minimizer and we can show that the same is true in the vectors' world. The proof requires mappings with certain properties between the vectors' world and positive semidefinite matrices. We begin by constructing the required mappings.

The Mapping

We begin with a basic fact about metric spaces.

Fact 4 (Royden [36, p.136]) *Let (X, d_1), (Y, d_2) be metric spaces with Y complete. Let f be a uniformly continuous function from a subset S of X into Y. Then there is a unique continuous extension $\bar{f}$ of f from S to $\bar{S}$ (the closure of S); that is, there is a unique continuous function $\bar{f} : \bar{S} \to Y$ such that $\bar{f}(x) = f(x)$ for all $x \in S$. Moreover, $\bar{f}$ is uniformly continuous.*

Fact 5 *Under the same conditions as in the previous fact, let f and $g : Y \to X$ be functions such that g is continuous and $g(f(x)) = x$ for all $x \in S$. Then $g(\bar{f}(x)) = x$ for all $x \in \bar{S}$.*

Proof: For $x \in S$ there is nothing to prove. Given $x \in \bar{S} \setminus S$, consider any sequence (x_i) in S with limit x. Then:

$$x = \lim x_i = \lim g(f(x_i)) = g(\lim f(x_i)) = g(\bar{f}(x)) \quad \square$$

Now, we will apply these facts to positive semidefinite matrices. Let PD_n be the set of positive definite $n \times n$ matrices all of whose diagonal elements are 1 and let PSD_n be the set of positive semidefinite $n \times n$ matrices with ones on the diagonal. Furthermore, let VCT_n be the set of $n \times n$ matrices whose column vectors have length 1, (i.e. lie on the unit sphere). Let $\partial P_n = PSD_n \setminus PD_n$. We can consider all these sets as metric subspaces of $(\mathbb{R}^{n^2}, \mathrm{d})$ where d is the Euclidean metric.

Fact 6 *VCT_n is closed. PSD_n is bounded and closed. PSD_n is the closure of PD_n.*

Let $g : VCT_n \to PSD_n$ be given by $g(V) = V^T V$. Note that g is continuous. The Cholesky decomposition c (see Press, Flannery, Teukolsky, and Vetterling [33] for details) decomposes a positive definite matrix Y into a lower-triangular matrix $V = c(Y) \in VCT_n$ such that $V^T V = Y = g(V)$. The Cholesky decomposition is continuous because it is the composition of continuous functions [33].

Fact 7 *The function $c : PD_n \to VCT_n$ computed by the Cholesky decomposition has a continuous extension $f : PSD_n \to S$, where $S = f(PSD_n) \subseteq VCT_n$ is the range of f. Furthermore, f is a homeomorphism. Its inverse is the restriction of g to S.*

Proof: VCT_n is closed (Fact 6) and thus *complete*, since it is a closed subspace of the complete space $\mathbb{R}^{n^2}$ (Rosenlicht [35, p.52,53]). PSD_n is a closed, bounded subspace of $\mathbb{R}^{n^2}$ (Fact 6). Therefore, it is compact ([35, p.58]) and, hence, since f is continuous, it is *uniformly* continuous ([35, p.58]). Thus, by Fact 4, and since PSD_n is the closure of PD_n (Fact 6), c has a unique continuous extension $f : PSD_n \to VCT_n$.

By the definition of the Cholesky decomposition, $g(f(Y)) = Y$ for all $Y \in PD_n$ and by Fact 5 and since g is continuous, this is true even for all $Y \in PSD_n$. □

The Correspondence between local Minima

Definition 1 *Given a function $f : X \to \mathbb{R}$, a point $x \in X$ is a* local minimizer *of f if*

$$\exists \epsilon > 0 \forall y \in \mathcal{B}_\epsilon(x) : f(y) \geq f(x) \tag{11}$$

where $\mathcal{B}_\epsilon(x)$ be the open ϵ-ball around x.

Recall from the description of the algorithm the definition of Z_v (the objective function in the unit vector world) and Z_y (the objective function in the positive semidefinite world).

We want to show that if V is a local minimizer for Z_v then $g(V)$ is a local minimizer for Z_y. The next Fact is the main step in the proof.

Fact 8 *Let (X, d_1) and (Y, d_2) be metric spaces and $h : Y \to S \subseteq X$ a homeomorphism (where $S = h(Y)$ is the range of h). Let $Z_x : X \to \mathbb{R}$ and $Z_y : Y \to \mathbb{R}$ be functions such that $Z_x(x) = Z_y(h^{-1}(x))$ for all $x \in S$. If $x \in S$ is a local minimizer of Z_x then $h^{-1}(x)$ is a local minimizer of Z_y.*

Proof: Let $x \in S$ be a local minimizer of Z_x. That is, let there be $\epsilon > 0$ such that $Z_x(z) \geq Z_x(x)$ for all $z \in \mathcal{B}_\epsilon(x)$. Since h is continuous, there is a $\delta > 0$ such that for all z in a δ-ball around $h^{-1}(x)$, $h(z)$ is in the ϵ-ball around $h(h^{-1}(x)) = x$. Therefore, for all such z:

$$Z_y(z) = Z_y(h^{-1}(h(z))) = Z_x(h(z)) \geq Z_x(x) = Z_y(h^{-1}(x)) \quad \Box$$

One last step is needed before Fact 8 and Fact 7 can be combined to yield the desired result. Note that these two facts cover only the case in which the local minimizer of Z_v lies in $f(PSD_n)$. We cover all other $V \in VCT_n$ by mapping them into $f(PSD_n)$ by means of an appropriate homeomorphism.

Fact 9 *For every $V \in VCT_n$ there is a $\bar{V} \in f(PSD_n) \subseteq VCT_n$ and a homeomorphism $k_V : f(PSD_n) \rightarrow k_V(f(PSD_n))$ such that $k_V(\bar{V}) = V$ and $Z_v(W) = Z_v(k_V^{-1}(W))$ for all $W \in k(f(PSD_n))$.*

Proof: Given a description of the Cholesky decomposition [33], it is not hard to see that

$$f(PSD_n) = \{(a_{ij}) \in VCT_n : a_{11} = 1, a_{ii} \geq 0 \ \ (i \geq 2) \text{ and } a_{ij} = 0 \text{ for } j > i\}. \qquad (12)$$

We can interpret the columns of $V \in VCT_n$ as coordinates of unit vectors w.r.t. the standard orthonormal basis of IR^{n}.

For each matrix $V \in VCT_n$ there exists an orthonormal basis B_V such that the coordinates of $\bar{V}$ with respect to B_V of the column vectors of V have the form of (12). In other words, there exists a basis B_V and $\bar{V} \in VCT_n$ such that $V = B_V^T \bar{V}$ and $\bar{V} \in f(PSD_n)$. If the columns of V are linearly independent, B_V can be found by applying Gram-Schmidt orthonormalization to V. As Gram-Schmidt orthonormalization defines a continuous function (because it is the composition of continuous functions), the construction can be extended to all of VCT_n using Fact 4.

We can take k_V as the restriction to $f(PSD_n)$ of the linear map defined by B_V^T. Clearly, B_V is invertible, k_V and its inverse are continuous, and $k_V(\bar{V}) = B^T \bar{V} = V$. Furthermore, as k_v corresponds to a change of basis, the lengths and angles of the vectors are unchanged. That is, $Z_v(W) = Z_v(k_V^{-1}(W))$ for all $W \in k(f(PSD_n))$. Thus, k_V is an appropriate homeomorphism. $\Box$

Fact 10 *Each local minimizer V of Z_v is in fact a global minimizer. Furthermore, $Z_v(V) \leq Z_y(Y)$ for all $Y \in PSD_n$.*

Proof: Any local minimizer $V \in VCT_n$ is also a local minimizer in $k_V(f(PSD_n))$, and $k_V \circ f$ is a homeomorphism with the properties required in Fact 8. Now, by Fact 8 with $X = VCT_n$, $S = k_V(f(PSD_n))$, $Y = PSD_n$, and $h = k_V \circ f$, we have that $(k_V \circ f)^{-1}(V)$ is a local minimizer of Z_y. But as PSD_n is convex and Z_y is a convex function, $(k \circ f)^{-1}(V)$ is indeed a global minimizer of Z_y (Gruber and Wills [17, p.632]). Finally, every $V \in VCT_n$ can be mapped via g to a $Y \in PSD_n$ such that $Z_y(Y) = Z_v(V)$ and vice versa (via f). $\square$

Biographies

Steven Homer has a 1978 Ph.D. in mathematics from the M.I.T. He is a Professor at Boston University where he has been on the faculty since 1982. Dr. Homer has been a Fulbright Scholar in Heidelberg (1988-89) and Visiting Professor in Oxford (1996). His research interests include complexity theory, parallel and randomized algorithms, mathematical logic and computational learning theory. From 1994 to 1997 he served as chair of the the annual IEEE Conference on Computational Complexity.

Marcus Peinado graduated in computer science from the Technical University of Berlin in 1991. He received his Ph.D. in computer science from Boston University in 1995. Dr. Peinado has been a visiting scholar at the International Computer Science Institute at the University of California, Berkeley (1996,1997). Currently. he is a research scientist at the German National Research Center for Information Technology. His research interests include parallel algorithms, high-performance scientific computing, randomized algorithms, and cryptography.