A Parallel Implementation of Approximate Semidefinite Programming for MAXCUT with Experimental Results

Jordi Petit i Silvestre
Fatos Xhafa

Report LSI-97-20-T
A Parallel Implementation of Approximate Semidefinite Programming for MAXCUT with Experimental Results*

Jordi Petit i Silvestre    Fatos Xhafa

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord – Mòdul C6
Jordi Girona Salgado, 1-3
08034 Barcelona, Spain
{jpetit,fatos}@lsi.upc.es

Abstract

We have implemented the approximate semidefinite programming for MAXCUT given by Klein and Lu (STOC'96). Our implementation for an IBM SP2 reveals the impractibility of this algorithm, basically because it hides the use of very large and very small floating-point numbers that can not be directly handled by the current computer hardware. On the other side, our experimental results show that the algorithm converges to a good approximation of the optimal, though oscillating.

1 Introduction and Motivation

Semidefinite Programming (SDP) is the problem of optimizing a linear function of a symmetric matrix subject to linear equality constraints and the condition that the matrix of variables be positive semidefinite. SDP is strong enough to represent combinatorial problems and very recently, it has been used to obtain striking approximation algorithms for several NP-hard problems, namely MAXCUT [GW94, GW95], Max 2SAT [FG95], Graph COLORING [KMS94], etc. In all these approximation results, the first step consists in obtaining a semidefinite relaxation of the problem at hand, then the semidefinite program is solved in linear time (by either Ellipsoid algorithms [GLS81] or Interior-Point algorithms [Ali95, HRVO93, VB96]), and finally, a feasible solution for the original problem is obtained from the solution of SDP, which is guaranteed to be within some bound of the optimum. The most celebrated approximation results based on semidefinite programming are those of Goemans and Williamson [GW95], and particularly the .878-approximation for MAXCUT problem (hereafter referred to as the GW algorithm).

In [GW95] the authors give also experimental result for several (random) type graphs evidencing the good quality of the solution found by their algorithm, where the code for solving the semidefinite program was based on (polynomial time) Interior-Point algorithms of Rendl et al. [RVW93]. Posteriorly, Peinado [Pei95] did a parallel implementation of GW algorithm on CM-5 with several interesting experimental results. His implementation

*This work has been supported by ESPRIT LTR Project no. 20244 — ALCOM-IT.
is based on unconstrained optimization. More precisely, he transforms equivalently the MAXCUT semidefinite program into a global unconstrained minimization problem and then applies the well-known First Derivative Methods (Gradient Descent or Conjugate Gradient).

It should be mentioned that the Interior-Point algorithms, on an instance of size $n$ take, roughly speaking, $O(n^{3.9})$ time. As for the gradient methods, any step of such methods takes $\Theta(n^2)$ time and they converge to the global optimum in a polynomial number of steps. Very recently, Klein & Lu [KL96] inspired by the scheme of Plotkin, Shmoys and Tardos [PST95] for solving a special class of Linear Programming, known as Fractional Covering/Packing Linear Programs, show that sometimes the semidefinite programming can be solved more fast to the price of finding a nearly-optimal solution. Specifically, they gave approximate semidefinite programs for MAXCUT (hereafter KL algorithm) and COLORING problems where in both cases they exploit the special structure of the semidefinite programs arisen from those problems. The interesting point of KL algorithm is that, for a given graph $G = (V, E)$ of $|V| = n$ nodes and $|E| = m$ edges and for any fixed $\varepsilon > 0$, it finds an $\varepsilon$-optimal solution in $\tilde{O}(nm)$ time (the $\tilde{O}$ notation suppresses the $\log n$ factors). Therefore, one would expect that their algorithm would run very fast on sparse or very sparse graphs, which indeed are the most difficult instances of the problem.

In this paper, we implement sequential and parallel versions of the KL algorithm in order to evidence the speed up of the parallel algorithm. The paper is organized as follows. In Section 2, we give some definitions and preliminaries together with an overview on Klein & Lu's algorithm. In Section 3 we present our implementation of this algorithm. Some experimental results are given in Section 4. We end the paper with a summary of our conclusions and observations.

## 2 Definition and Preliminaries

In this section, we give some useful definitions in order to fix the terminology and review Klein & Lu's algorithm for MAXCUT.

**$\varepsilon$-optimality.** Given a problem and an instance $x$ for it, a feasible solution $s$ to $x$ is called $\varepsilon$-optimal if the cost of $s$ is between $\text{Opt}(x)/(1+\varepsilon)$ and $(1+\varepsilon)\text{Opt}(x)$, where $\text{Opt}(x)$ is the optimal cost to instance $x$.

**MAXCUT Problem.** Given a graph $G = (V, E)$ and positive weights $c_{ij}$, for any edge $(i, j) \in E$, the MAXCUT problem is to find a partition of the vertex set $V$ into two subsets $V_1$ and $V_2$ that maximizes the sum of the weights of the edges with one endpoint in $V_1$ and the other in $V_2$.

It is already known that the algorithm of Sahni & Gonzalez [SG76] finds a cut of cardinality that is at least half of the optimal cut, i.e. it is 1-optimal.

**Fractional Packing Problem.** Given a set of $m$ inequalities $Ax \leq b$, where $A$ is $\mathbb{R}^n$ $m \times n$ matrix, $b$ a vector of positive components and a convex set $P \subseteq \mathbb{R}^n$ such that $Ax \geq 0$ for any $x \in P$, decide if there is a point $x \in P$ that satisfies $Ax \leq b$.

For this class of linear programming, there exist sequential algorithms (as the scheme of Plotkin, Shmoys and Tardos [PST95]) that solve the problem much more efficiently than the traditional algorithms for Linear Programming. [PST95] assumes that there exists a...
**fast subroutine** to solve the following optimization problem for a given convex set \( P \) and a matrix \( A \):

\[
\text{Given an } m\text{-dimensional vector } y \geq 0, \\
\text{find } \tilde{x} \in P \text{ such that } c\tilde{x} = \min(cx : x \in P) \text{ where } c = y^t A.
\]  

\( (1) \)

**Positive Semidefinite Matrix.** A real symmetric matrix \( X \) is positive semidefinite if and only if there exists a matrix \( U \) such that \( X = U^t U \).

**The Eigenvalue/Eigenvector Problem.** Let \( \lambda \) be the maximum eigenvalue of a positive semidefinite matrix \( A \). The Eigenvalue/Eigenvector Problem consists in finding a normalized vector \( x \) such that \( x^t Ax = A \cdot (xx^t) \geq (1 + \varepsilon) \lambda \), where the operation \( \cdot \) is defined by \( A \cdot B = \sum_{i,j} a_{ij} b_{ij} \). This problem can be solved by the well known *Power Method* (see, e.g., [IK66, pages 147–149]).

**The Semidefinite Program for MAXCUT.** Given a graph \( G \) with weight matrix \( C \), the semidefinite program for MAXCUT [GW95] is:

\[
\text{maximize} \quad \frac{1}{2} \sum_{i<j} c_{ij} (1 - X_{ij}) \\
\text{subject to} \quad X_{ii} = 1, \quad i \in V \\
X \text{ symmetric positive semidefinite.}
\]

\((SD)\)

**The Semidefinite Program (SD) in Terms of Matrices.** Let \( \tilde{L} \) be the *Laplacian* of \( G \) defined by \( \tilde{L}_{ij} = -c_{ij} \) if \( i \neq j \) and \( \tilde{L}_{ii} = \sum_{k=1}^{n} c_{ik} \). Then (SD) is written equivalently as follows [KL96]:

\[
\text{minimize} \quad 1/4 \ \tilde{L} \cdot X \\
\text{subject to} \quad X_{ii} = 1, \quad i \in V \\
X \text{ symmetric positive semidefinite.}
\]

\((SD1)\)

The above program is further written in the form of Fractional Packing Problem in order to match the Scheme of Plotkin *et al.* Let the matrix \( L = \tilde{L} + I/n \). Klein and Lu show that solving (SD1) is equivalent to solve the following program:

\[
\text{minimize} \quad \lambda \\
\text{subject to} \quad X_{ii} \leq \lambda, \ i = 1, \ldots, n \\
X \in P.
\]

\((SD2)\)
where \( P \) is the convex set
\[
P = \{ X \mid L \cdot X = 1; \; X \text{ positive semidefinite} \}.
\]
The equivalence between (SD1) and (SD2) means that any optimal (resp. \( \varepsilon \)-optimal) solution of one program can be transformed into an optimal (resp. \( \varepsilon \)-optimal) solution to the other. Moreover, the value of the CUT can be computed from
\[
1/4 \cdot L \cdot X = 1/4 (1/\lambda - 1).
\]
The fast subroutine of Klein and Lu solves the following problem (the analog of (1)): Given a vector \( y \) of positive components (and the convex set \( P \) of (SD2)), find the minimizer of
\[
\min \left\{ \sum_{i=1}^{n} y_i X_{ii} : \sum_{i,j} L_{ij} X_{ij} = 1, X \text{ positive semidefinite} \right\}
\]
Solving this last problem can be reduced to solving an eigenvector problem and therefore the Power Method, mentioned previously, can be employed.

**Review of Klein & Lu’s Algorithm for MAXCUT.** The procedure \( \text{MaxCut}(C, \varepsilon) \) first finds an initial solution that is 1-optimal. This solution is found by the greedy procedure of Sahni & Gonzalez [SG76]. Then, the algorithm starts with \( \varepsilon' = 1 \) and runs in scaling phases. In each phase, the \( \varepsilon' \) is halved and a procedure \( \text{Improve}(X, \lambda, \varepsilon'/7) \) is called repeatedly using the previous output as the input until \( \varepsilon' \leq \varepsilon \).

**procedure** \( \text{MaxCut}(C, \varepsilon) \) is
\[
\begin{align*}
(X, \lambda) &:= \text{Initial}(C) \\
\varepsilon' &:= 1 \\
\text{while } \varepsilon' > \varepsilon \text{ do} \\
\varepsilon' &:= \varepsilon'/2 \\
(X, \lambda) &:= \text{Improve}(X, \lambda, \varepsilon'/7) \\
\text{end while} \\
cutWeight &:= (1/\lambda - 1)/4 \\
\text{return } (X, \lambda, \text{cutWeight})
\end{align*}
\]

The procedure \( \text{Improve}, \) on its turn, uses a vector \( y \) with positive components defined as a function of the current solution \( (X, \lambda) \), and it proceeds iteratively to update (i.e. improve) the current solution. More precisely, it finds a feasible matrix solution \( \bar{X} \) and the current solution is moved towards \( \bar{X} \) by a small amount \( \sigma \), i.e. \( X := (1 - \sigma)X + \sigma \bar{X} \) and \( \lambda \) is let to be the maximum diagonal element of the new matrix solution \( X \).

**procedure** \( \text{Improve}(X, \lambda_0, \varepsilon) \) is
\[
\begin{align*}
\lambda &:= \lambda_0 \\
\alpha &:= \lfloor (12/\varepsilon) \ln(2n/\varepsilon) \rfloor \\
\sigma &:= \varepsilon/(4\alpha n) \\
\text{repeat} \\
\forall i = 1, n : \; y_i &:= e^{\alpha X_{ii}} \\
\bar{X} &:= \text{Direction}(y, \varepsilon) \\
\text{if } \left( \sum_{i=1}^{n} y_i X_{ii} - \sum_{i=1}^{n} y_i \bar{X}_{ii} \right) &\leq \varepsilon \cdot \left( \sum_{i=1}^{n} y_i X_{ii} + \lambda \sum_{i=1}^{n} y_i \right) \text{ then} \\
\text{return } (X, \lambda) \\
\text{else}
\end{align*}
\]

4
\[ X := (1 - \sigma)X + \sigma \bar{X} \]
\[ \lambda := \max\{X_{11}, \ldots, X_{nn}\} \]
end if
for ever
end

The procedure Direction is in fact the fast subroutine used in the scheme of Plotkin et al. to solve (2).

Finally, the running time required by the algorithm MaxCut(\(C, \varepsilon\)) is \(\tilde{O}(mn/\varepsilon^3)\) for a graph with \(n\) vertices and \(m\) edges and error parameter \(\varepsilon\).

3 The Implementation

3.1 Considerations on the implementation of theoretical algorithms

Due to the nature of theoretical computer science, it is often the case that the implementation of a concrete algorithm on a real language for a real machine is a difficult matter. In general, this happens because practical issues are frequently ignored by theoretical algorithm designers, simply because these considerations do not matter in the mathematical model. As a consequence, the implementation of a theoretical algorithm (what is called algorithm engineering) is in many cases an interesting area by itself. Moreover, the implementation and testing processes can sometimes report new insights on the problem or algorithm under consideration, notably on its practical side.

As an example, consider the large family of parallel algorithms designed for the PRAM model. Most of these algorithms have never been implemented on real parallel machines, and when they were, implementors reported a considerably effort to code them and some discrepancies between the predicted and measured running times [GiS97]. To remedy this problem, some more realistic models of parallel computing have been proposed (such as LogP or BSP).

Among other reasons, difficulties for practical implementations of theoretical algorithms appear because of:

- The use of high level algorithms (for instance, the algorithm of Luby and Nisan for positive linear programming).
- The use of idealized models (like PRAMs).
- Hidden factors and constants (as is the case in the MaxCut algorithm, where the time complexity does not measure internal matrix operations).
- The use of unspecified subroutines and unspecified constant parameters (like defining parameters in big-oh notation).
- The use of theoretically good algorithms with bad practical performance (for example the Ellipsoid algorithm versus the Simplex or the Interior Point method).
- The use of algorithms that are existential in nature (for several problems no explicit algorithms are given, but instead their existence is proved).

In order to have an insight on this respect, we will consider a parallel implementation of the approximate semidefinite program for MaxCut [KL96].

5
3.2 Faced problems

The implementation of this algorithm seemed straightforward at the very beginning. The algorithm mainly uses linear algebra operations on matrices and these should have an easy parallelization using standard techniques. In fact, many traps were hidden and could only be seen as the implementation was progressing and extensive tests were run.

The first problem we had to deal with was the use of floating-point numbers. More precisely, our program has to use very large and very small numbers, which cannot be handled with the standard double data type. The use of large numbers appears while computing exponential values such as \( \exp(600) \) and the use of small numbers when computing the biggest eigenvector. We overcame this problem using a multiple precision library [Gra96]. Since the use of multiple precision numbers implies a slowdown, we have avoided their use whenever possible.

The most costly as well as the most frequently used part of the program corresponds to compute the biggest eigenvector of a matrix derived from the graph. In principle, the dimensions of such matrix can be very large. The Power Method is a frequently used method to find the biggest eigenvector, and it is also adapted in the [KL96] algorithm. We recall that there is no standard implementation of this method and practitioners do adapt it to their needs (profiting the structure of the problem in mind). Moreover, the few existing implementations [DPW96] compute all the eigenvectors of a given general matrix (e.g., sparse matrices are not considered). In our case, we are concerned with the precision issue because the entries of our matrix are (very) small numbers. Therefore we did our own implementation, which is fast enough. We have used a scaling technique to speedup this computation.

Finally, the executions of our program for moderated size inputs resulted very time consuming. This is due to the large number of calls to Power Method. For this reason we had to limit ourselves to do a relatively small number of trials.

The following section explains some of the general decisions we took to solve these problems. Details can directly be found in the code.

3.3 Language and libraries used

The goal was to efficiently run the algorithm for MaxCut on a parallel computer with distributed memory, namely an IBM SP2 with 8 Power2 processors under the AIX operating system. This computer has an interconnection network that provides high performance communication between the processors. A picture of this machine can be found in Figure 1. The SP2 follows the current trend in parallel computing industry, which consists in offering a modest number of powerful processors with distributed memory linked by a high bandwidth network. Remark that this kind of hardware is far from the PRAM model, that presupposes a large number of processors using a unique shared memory.

In order to code our program we decided to use ANSI C++ [Str93] (a high level object oriented version of the C language) with the help of some additional libraries. Besides portability and efficiency, the use of C++ gave us the possibility to have a very structured and readable code, mainly due to the use of polymorphism and operator overloading to hide matrix operations details. Another interesting possibility of C++ is the use of references, which decreases the usual pointer mess in C.

In order to have a parallel version of our program, we used the MPI library [For94, Mis96]. This library is quickly becoming an standard among message passing libraries, due to its advantages in portability, efficiency, availability and easy of use. Moreover, the MPI implementation for the IBM SP2 fully profits its high performance network. The MPI
model consists of a collection of processes (in the Unix sense) running on (usually) different processors. Each process has its private area of memory and can communicate with the rest of processors through message-passing, i.e. sending and receiving messages. No primitive mechanism of shared memory is provided within MPI, but instead many usual primitives such as broadcasting, gathering and scattering are provided.

We used the GNU MP library [Gra96] to have arbitrary precision arithmetic on floating-point numbers. This library aims to provide the fastest possible arithmetic for applications that need higher precision than is directly supported by the basic C or C++ types (float or double). The speed of MP is achieved by using fullwords as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code, and by a general emphasis on speed (as opposed to simplicity or elegance). Because this library is written in C, we wrote a wrapper in order to use it comfortably from C++ and use its big floating-point numbers (that we call real) as any other predefined simple type. In order to send and receive reals with MPI, we had to do a bit of hacking, because the GNU MP hides the representation of its abstract data types.

3.4 Modules of the code

Our code has the following modules:

Arbitrary precision arithmetic module. This module is a C++ wrapper to the data types and functions offered by the GNU MP library. It offers a class called real that corresponds to a floating number with arbitrary precision with its usual operations as *, exp(), ..., automatic conversion and input/output support. The use of this class is straightforward, see for example the following example:

double d=3.1416;
real x,y,z;

x=1.2; y=d; z=x+y*exp(800);
cout << x;

**Vector module.** In order to comfortably handle vectors of any type, we built a generic vector class with its usual operations. For example one can write:

```c++
vector<double> v=vector<double>(n);
vector<double> w=vector<double>(n);
vector<real> x=vector<real>(n);

v=3; v(6)=4; x=2*v+w-v(1);
```

**Matrix module.** As with the vector module, a generic class matrix offers operations for symmetric matrices.

**Graph module.** We implemented a module to encapsulate the usual operations on sparse graphs, as defining it, getting its adjacency lists, degree information...

**Communications module.** This module offers an interface to the MPI library in order to efficiently communicate objects of the classes offered by the previous modules. The more usual operations on our code were broadcast and gather communications and these were optimized in order to reduce the number of MPI calls by packing.

### 3.5 The structure of the main program

The code is available under

[http://www.lsi.upc.es/~fatos/Implementations/maxcut](http://www.lsi.upc.es/~fatos/Implementations/maxcut)

on the WWW. In order to show the structure of our program and its main data structures, let us present the main class MaxCut, which represents the algorithm.

```cpp
typedef vector<double> vectord;
typedef vector<real> vectorr;
typedef matrix<double> matrixd;
typedef matrix<real> matrixr;

class MaxCut {
private:
    int n;     // Number of nodes
    graph G;   // Input graph
    matrixd w,L,Lh; // Matrices w, L and L(bar)
    double epsilon; // Approximation factor
    vectord x0; // Private variable
    int nbp;   // Number of processors
    int pid;   // Processors identifier

    // Create an initial solution with the Sahni-Gonzalez // greedy algorithm
    void Initial (matrixr& X, double& lambda);

    // Find the appropriate direction for the next improve
    void Direction (const vectorr& y, matrixr& X, double epsilon);

    // Compute the new solution of improve cost
    void Improve (matrixr& X, double& lambda, double epsilon);
```
public:
    MaxCut (int argc, char **argv); // Constructor
    double Approximate (void); // Starter, returns the cut

The main function, that we reproduce below, is MaxCut::Approximate. We remark its similarity with the high level algorithm shown in [KL96].

double MaxCut::Approximate (void) {
    matrixx X = matrixx(n,n);
    double lambda, wcut = 0;
    Initial(X, lambda);
    double epsilonP = 1;
    while (epsilonP > epsilon) {
        epsilonP *= 0.5;
        Improve(X, lambda, epsilonP / 7);
        wcut = 0.25 * (1 / lambda - 1);
    }
    return wcut;
}

4 Experimental Results

In this section we present some of the empiric results we obtained with our implementation. Let us first explain the type of graphs we use: the popular \(G_{n,p}\) model. In this model, graphs have \(n\) nodes and each of their possible edges appears with probability \(p\). Moreover the weights of the edges are drawn from a uniform distribution in some arbitrary interval and are then normalized. In order to focus on sparse graphs, we had to take a low probability \(p\).

We used as benchmarking graphs one instance of \(G_{50,0.1}\) and one instance of \(G_{100,0.1}\). We would have like to use larger graphs, but the long execution times needed by the implementation of Klein & Lu's algorithm have restricted it. We have preferred using small values of the error parameter (\(\varepsilon = 0.00625\)) than larger graphs. On the other hand, our experimental results evidence two remarkable facts.

First of all, from Figures 2 and 3 we can observe that the weight of the cut has an increasing tendency. We point that this tendency is however not continuous, but oscillates while growing. This fact is in concordance with the analysis of the original algorithm of Plotkin et al for Packing Linear problems, who theoretically predicted such oscillations.

The second fact is that the execution times are very slow. We explain this slowness by the large numbers of calls to the Power Method that the program does. We notice that a simple call of this procedure is quite fast, because we optimized it having into account the sparseness of the matrix. But this is not sufficient to cope with a so large number of calls.

Besides this high cost, we also have to take into account the extra overhead due to the use of multiple precision numbers. Further, even if the Power Method did not use these big numbers, it deals with a matrix whose entries are very small. This forces the Power Method procedure to perform a large number of iterations in order to converge to the biggest eigenvector of the matrix.
Figure 2: Trace of the cut in function of time on a random sparse graph with 50 nodes, where each edge appears probability 10%. The top figure shows the complete run; the bottom one focuses on the interval [0..600]. Each unit of time corresponds to 500 iterations of Improve.
Figure 3: Trace of the cut in function of time on a random sparse graph with 100 nodes, where each edge appears probability 10%. The top figure shows the complete run; the bottom one focuses on the interval [0..200]. Each unit of time corresponds to 500 iterations of Improve.
5 Conclusions

This work was motivated by the increasing interest in implementing new theoretical algorithms on sequential and parallel computers. In order to obtain approximation results for MAXCUT problem we addressed the implementation of Klein & Lu's approximate semidefinite program. Though the algorithm seems quite explicit at first glance, there are several points which make difficult its implementation. For example, the algorithm hides the use of large numbers that cannot be handled directly by the current hardware. This fact evidences a gap between the theoretical algorithms (based on idealized models) and their efficient implementation (on real machines). However, allowing inordinate running times, we have seen the convergence of the algorithm under consideration.

References


