Parallel implementation of graph partitioning algorithms for VLSI-CAD applications

ME766: High Performance Scientific Computing, May 2021

Gaurav P. Kate, 183079018
Department of Electrical Engineering, IIT Bombay
183079026@iitb.ac.in

Punit Jain, 17307R016
Department of Electrical Engineering, IIT Bombay
punitjain@iitb.ac.in

Sandesh Goyal, 183079026
Department of Electrical Engineering, IIT Bombay
sandeshg@iitb.ac.in

Vidur Shah, 183079035
Department of Electrical Engineering, IIT Bombay
183079035@iitb.ac.in

***Abstract*: The problem of graph partitioning is NP-Complete in generating the result with minimum cutset. VLSI netlist can be represented as a graph with nodes/vertices representing circuit elements and edges representing the interconnections. We have implemented graph partitioning by heuristic based iterative algorithm (KL Algorithm) and Spectral algorithm for comparing their performance in terms of complexity and speed using VLSI netlist dataset. Serial implementation of the KL algorithm has a very high time complexity of $O(n^3)$, where N is the number of nodes in the bi-partition graph. We have done serial implementation of the KL algorithm and Spectral partitioning for a single thread and then parallelized it using OpenMP to run on multiple CPU threads and CUDA to run on multiple GPU threads and did a comparative analysis.**

***Keywords*: KL, Spectral, OpenMP, CUDA**

I. Introduction

Since balanced graph partitioning with minimum cut-set is NP-Complete problem, heuristics have to be applied to solve the problem for balanced partitions like KL heuristic. Another approach would be to use spectral methods. E R Barnes[12] has found a Heuristic based on linear programming transportation problems. The spectral methods represent the graph in the form of adjacency matrix and degree matrix. The minimum cut-set partition can be found from the Eigen-vectors of the matrix representation. This approach helps us explore parallelism that can be exploited and advantages gained in Matrix based computations using tools such as OpenMP or CUDA. Many such implementations are available in the literature[14]

II. Kernighan-Lin Approach

KL Partitioning algorithm is a bi-partition algorithm which takes the nodes(referred to as MODULES in code) and edges as input and creates an adjacency matrix. The two sets are initialized either with equal or with difference of one number of nodes which ensures that the two sets are balanced. The algorithm finally produces the two sets such that the cut-set is minimum. KL can be used to partition any graph or circuit which can be converted to an adjacency matrix using nodes and edges. The algorithm has a time complexity of $O(n^3)$ where n is the number of nodes in a set.

A. Algorithm

Algorithm starts with initializing the two balanced sets, in order to avoid the randomness in initialization the nodes are stored in the whole number format in increasing order, starting with 0 and goes upto n-1(n is number of nodes), therefore set1 is initialized with nodes from 0 to (n+1)/2 (taking care of odd/even n with set1 getting 1 node more than set2 if n is odd). Adjacency matrix for a set contains its nodes along with the interconnection value for every other node, both in set1 and set2. Every edge connecting one node to another node is given a weight of 1.

After the two sets are initialized with adjacency matrix, Internal cost **(Ix)** and External cost **(Ex)** for all nodes in both the sets are computed where Ix is the cost(number) of edges connecting node x within its own set and Ex is the cost of edges connecting node x with the other set. After this a difference is calculated for all the nodes in both the sets which is **Dx**(= Ex - Ix) where Dx represents the gain or reduction in cutset in moving node x from its current set to the other set.

Next step is to compute the gain benefit (**Gxy**) for exchanging or swapping a node pair between the 2 sets (one node from a set is taken), Gxy (= Dx + Dy - 2*Dxy) is calculated for all the possible node pairs in the two sets where **Dx** is the gain for node x in set1, **Dy** is the gain for node y in set2 and **Dxy** is the number of edges connecting the node x to y.

Next step is to get the node pair with maximum gain benefit (**Gxy max**), once obtained, the two nodes are exchanged between the two sets. After swapping, the two swapped nodes are locked and will not be considered further for swap in the algorithm. This swap leads to change in the Ex and Ix of not only the swapped nodes but all the other nodes as well.

Next step is to update the Ex, Ix and finally Dx for all the nodes(locked nodes can be skipped). **Dx updated (**for nodes in set1**)** = Dx(old) + 2*Dx1 - 2*Dx2 and **Dx updated (**for nodes in set2**)** = Dx(old) + 2*Dx2 - 2*Dx1 where **Dx1** is the number edges between node x and the swapped node from set1 and **Dx2** is the number edges between node x and the swapped node from set2. The swapped nodes and the gain benefit in swapping the two is stored and this marks the first iteration of the algorithm and here **G1** will be stored as the gain benefit for the first iteration.

With updated values of Dx, the previous 3 steps are repeated till there are no more unlocked nodes left to swap. The number of iterations is equal to the number of nodes in set2(n/2). In every iteration 2 nodes(1 in each set) are locked and therefore the number of unlocked nodes to be considered for gain benefit in the next iteration reduces. Also for every iteration **Gx** is stored which is the gain benefit for swapped nodes in iteration x.

After all the iterations (n/2) are completed, maximum cumulative gain benefit (**Gt**) is calculated in the order of iteration. First t swap pairs are considered for the final result which gives maximum cumulative gain or minimum final cut-set size. Gt is subtracted from the initial cut-set value to get the final cut-set value.

B. Global Run

The current final cut-set obtained is best for the particular set initialization used in the above description, however there can be a random initialization for which final results can be even better but instead of searching or guessing for that random initialization we continue with our initialization scheme but run the complete algorithm until **non-positive** Gt is obtained, this is termed as Global run and for every next global run the set initialization will be the swapped or updated version obtained from the previous run. By using this approach we get absolute maximum gain or absolute minimum cut-set. If Global run is used, the algorithm will always run one extra loop which will verify that no further improvement is possible.

C. Implementation

Different design blocks implemented (Fig. 1 and Fig. 2):

1. Normal KL:
   a. Parser: reads data from .net file and creates adjacency matrix for set1 and set2 with details of nodes and edges, it also add a unique set_id to all nodes
   b. Initial gain: calculate Ex, Ix and Dx for all nodes in both the sets
   c. Swap gain benefit with maximum gain: calculate swap gain benefit for all possible node pair swap and also find the maximum of them (used in OpenMP)
   d. Swap gain benefit: calculate only swap gain benefit for all possible node pari swap (used in CUDA)
   e. Maximum swap gain: calculate only maximum swap gain (used in CUDA)
      - V1: square root method is used to calculate number of threads that run in parallel, this version performs better(less time) for dataset with less number of nodes
      - V2: square root multiplied with 2 is used to calculate number of threads that run in parallel, this version performs better(less time) for dataset with more number of nodes
   f. Swap nodes: swap node pair with maximum gain benefit and their related parameters like set_id
   g. Update gain: update Dx for all nodes in both the sets after node pair swap is completed
   h. Maximum cumulative gain(MCG): calculate MCG that can be obtained from all node pair swaps done in sequence
2. Full KL:
   a. Update set: update the two sets for adjacency values and set_id in sequence where maximum cumulative gain is obtained
   b. Total gain: calculate total for all the full run of the algorithm, this is the sum of all cumulative gains

Single thread execution of the OpenMP version can be treated as serial execution.

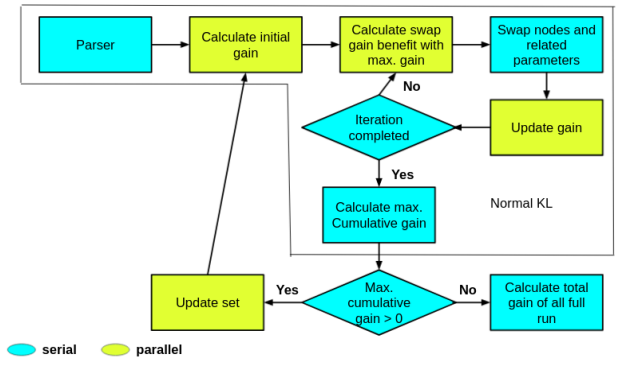


Fig. 1 Block diagram for OpenMP based implementation of full KL

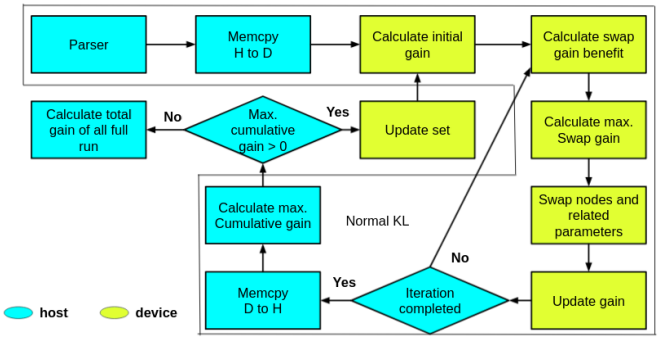


Fig. 2 Block diagram for CUDA based implementation of full KL

D. Results:

KL implementation for OpenMP and Cuda is tested on a platform with Intel(R) Xeon(R) CPU E5-2640 v4 (20 physical cores) @ 2.40GHz, 32 GB RAM and CentOS Linux 7 (Core) 64-bit and GPU Quadro K2200 with 4GB of global memory.

TABLE I
Speedup data for Multiple CPU & GPU threads compared to single CPU thread

Dataset	No. of Nodes	OpenMP					CUDA	
		1	2	4	8	16	v1	v2
soc-karate	34	1.00	5.67	5.67	4.53	4.00	0.02	0.02
bn-mouse_visual-cortex_2	194	1.00	4.52	4.91	3.05	2.90	0.03	0.03
bn-macaque-rhesus_brain_1	242	1.00	5.32	6.41	5.74	4.64	0.07	0.07
ia-crime-moreno	829	1.00	1.94	2.92	3.62	3.44	0.18	0.17
bn-mouse-kasthuri_graph_v4	1029	1.00	1.83	2.42	3.70	3.48	0.24	0.24
bn-fly-drosophila_medulla_1	1781	1.00	2.11	3.20	4.50	4.13	0.88	0.85
bio-DM-HT	2989	1.00	2.00	3.03	3.65	3.82	2.42	2.59
ca-GrQc	4158	1.00	1.89	2.79	2.97	3.62	3.46	3.69
Ca-Erdos992	6100	1.00	1.68	2.03	2.14	2.94	4.32	4.52
bio-dmela	7393	1.00	1.84	2.78	3.20	3.54	4.63	4.74
CL-10000-2d1-trial2	10000	1.00	1.75	2.29	2.33	2.59	5.09	5.30
ibm01	12752	1.00	1.66	1.96	3.03	3.51	4.51	4.78

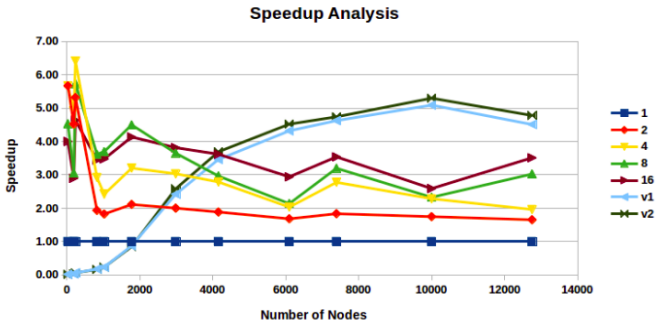


Fig. 3 Speedup analysis of OpenMP and CUDA implementation for different number of nodes

TABLE II
Cut-set result for Normal and Full KL

Dataset	No. of Nodes	Initial Cut-set	No. of run for Full	Final Cut-set	
				Normal	Full
soc-karate	34	20	2	10	10
bn-mouse_visual-cortex_2	194	115	3	13	11
bn-macaque-rhesus_brain_1	242	1065	4	864	862
ia-crime-moreno	829	480	4	253	227
bn-mouse-kasthuri_graph_v4	1029	674	4	276	245
bn-fly-drosophila_medulla_1	1781	18646	3	2628	2200
Bio-DM-HT	2989	1645	4	424	388
ca-GrQc	4158	6726	5	1053	934
Ca-Erdos992	6100	3734	4	1130	1070
bio-dmela	7393	9431	6	4235	4139
CL-10000-2d1-trial2	10000	14989	4	6002	4865
ibm01	12752	70821	4	14795	12156

III. Spectral Methods

B. Theory

Spectral method for partitioning involves obtaining the second smallest eigenvalue and the corresponding eigenvector(Fiedler vector) of the Laplacian matrix obtained from the input circuit netlist. From the Fiedler vector, we can obtain partitions by checking the sign of indices of the eigenvectors. For multi-way partitioning, multiple eigenvectors can be used. Individual components involved in the process are described below.

Graph Laplacian Matrix: A level-1 heading must be in Small Caps, centered and numbered using uppercase Roman numerals. For example, see heading “III. Page Style” of this document. The two level-1 heading

Basic Lanczos Algorithm: Finding the eigenvectors of the Laplacian matrix is a very costly process in terms of time. Lanczos Algorithm is used to transform the sparse symmetric Laplacian matrix into a symmetric tridiagonal matrix T. The eigenvalues and eigenvectors of matrix T (Ritz vectors and Ritz values) approximate smallest and largest eigenvalues and corresponding eigenvectors of original Laplacian matrix. Such that Q^TAQ is a tridiagonal matrix and Q called Lanczos vectors matrix is orthogonal. The single Lanczos iteration is requires k*n operations for matrix vector or vector-vector multiplication or even lower O(n) for sparse matrices however the value of k increases linearly with number of iterations. If the Lanczos algorithm converges at the jth iteration, the QR algorithm further requires $O(j^2)$ for tridiagonal matrix T_{jxj}. As the Lanczos process proceeds in each iteration, the Q vector loses its orthogonality and has to be re-orthogonalized. Re-orthogonalization is an operation with higher algorithmic complexity $n^2*k/2$ operations increase linearly with iterations. Steps for basic Lanczos algorithm are given below for reference where A is the Laplacian matrix and \alpha and \beta are diagonal and off-diagonal of the matrix T. Lanczos algorithm finds many applications in fields of structural analysis, graph theory, and other problems where approximate eigenvalues and eigenvectors are useful.

ALGORITHM I

BASIC LANCZOS ALGORITHM

	Initial $r0$ = random,
0	$q_0=0, q1 = r_0/\|\|r_0\|\|$
	for $k = 1 \rightarrow n$
1	$r_k = Aq_k$
2	$\alpha_k = r^T q_k$
3	$r_k = r_k - \alpha_k q_k - \beta_{k-1} q_{k-1}$
4	$\beta_k = \|\|r_k\|\|$
5	*Reorthogonalize*(q_k)
6	$q_{k+1} = r_k \beta_{k-1}$
	$\beta_k = 0$? break; else continue to (1)

QR Algorithm: QR algorithm is also used in eigendecomposition for tridiagonal matrices. Here, it is used iteratively to converge the off-diagonal elements to zero, giving a diagonal matrix. The entries on the diagonal give all eigenvalues of the input matrix. A basic iteration is shown below:

ALGORITHM II

BASIC QR ALGORITHM

1	$A^{(k)} - \mu_k I_k = Q_k R_k$
2	$A^{(k+1)} = R_k Q_k + \mu_k I_k$
3	$Q_k = Q_{k-1} G_k$

The rate of convergence is slow in case of directly applying this method. However, most eigendecomposition problems require a few eigenvalues and not the complete set. In such scenarios, the shifted version of this algorithm is commonly used.

The Shifted-QR algorithm incorporates a shift amount, which can speed up convergence. This shift amount can be specified explicitly(explicit shift QR method) or calculated implicitly using elements of the matrix(implicit shift QR method). Here, explicit method is advantageous if an approximation of required eigenvalue or range of eigenvalues needed, is known apriori. Another way is the use of implicit shift amounts calculated using Wilkinson's method or Rayleigh's heuristic.

The general flow is as follows. The shift amount is derived for the last diagonal element. Then, starting from the first element $A_{(1,1)}$ of input matrix A, the shift is applied and the elements $A_{(1,2)}$ and $A_{(2,1)}$ are eliminated using Given's rotation. Given's rotations are continued for all elements along the diagonal, to eliminate off-diagonal elements. This process is repeated for shift amounts derived from all diagonal elements except $A_{(1,1)}$. Alongside this, the corresponding Given's rotations are right-multiplied to obtain the corresponding eigenvector matrix.

C. Our Implementation

1) *Lanczos with Partial Reorthogonalization(PRO):* The partial reorthogonalization scheme implemented (similar to [13]) finds the w vector for each Lanczos iteration, the values in w vector represents the loss of orthogonality of each Lanczos vector. Recall that each column of Q represents a Lanczos vector. Since finding $||Q^TQ||$ is expensive, an approximate formula is used to find w_{k+1} from w_k, w_{k-1} and β,α. Based on the value of w_i, we orthogonalize against q_i if $w_i > \epsilon^{\eta}$, where ϵ is the machine precision and η is a constant depending upon the precision we expect to obtain. Partial reorthogonalization avoids reorthogonalization against all Q vectors and reduces the run time by mn^2 where as, it's kn^2 where k is k^{th} iteration and n is the maximum dimension of Laplacian The scheme implemented also predicts the convergence of second smallest eigenvalue. If the convergence has occurred, the Lanczos iteration is stopped. For finding whether the eigenvalues are converged, we have to run the QR algorithm on the T matrix which is costly. To reduce the number of such QR steps during the Lanczos iteration, the number of steps required from the current step is speculated and next QR step is run only at the speculated Lanczos step. For speculating, we need two consecutive QR steps to get eigenvalue for two consecutive iterations. Algorithm III describes our implementation.

We have run two consecutive QR runs parallely using OpenMP sections clause. **Please note that the code for the QR algorithm is taken from the source[10] for Lanczos PRO**. The code implemented is mostly iterative and not much scope for parallelism. Also parallelism has been exploited for matrix-matrix and matrix-vector computations using OpenMP parallel for. Parallel-for is used to make the computations for vector-vector computations but for higher scalability, more number of processor cores have to be used. We have run the implementation on 4-core(8 hyper-threaded) on a standard dataset available at source[10].

ALGORITHM III

LANCZOS ALGORITHM WITH PRO AND HOUSEHOLDER SHIFT QR

	Step	Comments
0	*Parse input netlist to obtain Laplacian (A)*	serial code
1	*Initialize Lanczos Algorithm*	
2	*Basic Lanczos steps*	*M*V operations parallel*
3	*if near convergence: QR consecutive steps*	*(less frequent speculation)*
4	*Estimate loss of orthogonality*	*not costly*
5	*Re-orthogonalize against bad q vectors*	*M*M*V most costly*
6	*Check Convergence: goto(2) if not converged*	*condition to exit*
7	*Run QR estimate Fiedler vector*	*serial implementation*
8	*Partition nodes and calculate cost*	*search/sort type*

2) *Lanczos with Selective Reorthogonalization(SRO):* Selective reorthogonalization involves computing an approximate error, and if it is beyond a certain threshold, complete re-orthogonalization is performed for two newest vectors with all previous vectors. Instead of deriving non-orthogonalization from individual dot-products between vectors, parameters $\omega_{(j,k)}$ $\omega_{(j,k+1)}$ are derived and used. If the maximum value of ω is greater than the error threshold, then complete re-orthogonalization is done.

A reorthogonalization is the costliest step in the entire procedure(requires matrix-matrix multiplication), we use parallel computation methods using OpenMP and CUDA C for

re-orthogonalization. Parallel computation is also deployed for vector calculation and approximate error computation and in QR iteration. A reference to the error approximation method used in SRO is explained in [7]. We have used the same method in our implementation.

For our application, the input matrix is a symmetric tridiagonal. We have used both, QR without shift as well as Implicitly shifted QR method with Wilkinson shift[11]. We implemented the QR algorithm for SRO according to the above described algorithm.

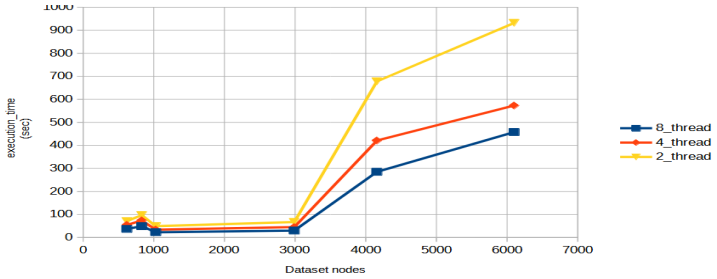
ALGORITHM IV

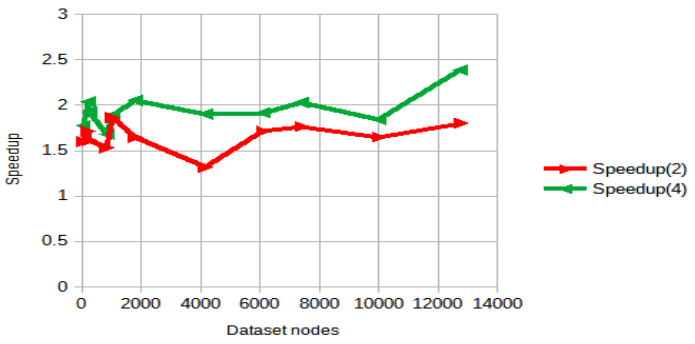
LANCZOS ALGORITHM WITH SRO AND IMPLICIT QR

	Step	Comments
0	r_0 = *Initialized vector* $\beta_0=\|\|r_0\|\|$	*Random initialization*
	for $k = 0 \rightarrow n$	
1	$q_{k+1} = r_k \beta_{k-1}$	
2	$r_k = Aq_k$	*Matrix-vector product*
3	$\alpha_k = r^T q_k$	
4	$r_k = r_k - \alpha_k q_k - \beta_{k-1} q_{k-1}$	*Obtain next r*
5	$\beta_{k+1} = \|\|r_k\|\|$	*Normalize* r_k *for* β_k
6	*Reorthogonalize(* q_k *) and modify* β_{k+1}, r_k	*Matrix-matrix product*
7	*Implicit QR and check eigenvalues*	*QR algorithm*
	β_{k+1} *< err? break; else continue to (1)*	*Exit condition*
	Check partition from eigenvector	*Check and separate nodes*

D. Results

The figures below summarizes the runtimes and speed up obtained for OpenMP implementation on a single processor system with: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz max frequency, 4 Cores, 8 Logical Processors, 8.0GB main memory and Windows- 10, 64 bit OS for (PRO implementation). Intel(R) Core(TM) i3-4030H CPU @ 1.90GHz max frequency, 2 Cores, 4 Logical Processors, 8.0GB main memory and Windows- 10, 64 bit OS for (SRO implementation).





IV. CONCLUSIONS

Since the KL algorithm time complexity is of the order of O(n3), serial runtime increases exponentially with the number of nodes, parallelism is extracted in multiple blocks of implementation to get time complexity of O(n) in some blocks and O(n2) in others, as observed in Fig. 3 and Table 1, OpenMP gives speedup but not exactly equal to the number of threads because of scheduling and memory access overheads, 8 threads execution performs better than 16 threads execution for less number of nodes and vice versa. V2 of Cuda implementation performs better for higher number of nodes and vice versa. Cuda performs better than OpenMP because of more threads and efficient scheduling. Reduction in cut-set is repeatable in every execution.

It can be clearly seen that the SRO based Lanczos algorithm needs very less time compared to PRO, this is because of less re-orthogonalization in Lanczos SRO. This is the trade-off between accuracy and speed. Open MP implementation of the algorithm further improves the speed to some extent by parallelizing the matrix-vector operations. However, not much parallelism could be exploited in the QR algorithm due to its iterative nature and non-deterministic time. The CUDA implementation shows speed-up especially for re-orthogonalization. However, the GPU global memory limitation might become the bottleneck when matrix sizes are scaled. This will require more transfers and thus bring down the speed. The processes can be further improved by taking advantage of the sparse nature of the graph Laplacian matrix and performing sparse operations. The memory utilization grows significantly as the problem scales and restart based Lanczos methods have to be used in that case.

ACKNOWLEDGMENT

We are thankful to the authors of the document E. Biegert and N. Konopliv from which we obtained the QR Algorithm for Lanczos PRO to compare the performance with our QR Algorithm implementation from web source[10].

REFERENCES

[1] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," in *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291-307, Feb. 1970, doi: 10.1002/j.1538-7305.1970.tb01770.x.

[2] A K Rajan and D Bhaiya, "VLSI partitioning using parallel kernighan lin algorithm," (ICCSP), India, 2017, doi: 10.1109/ICCSP.2017.8286727.

[3] Lars Hagen, A B Kahng, "New Spectral Methods for ratio-cut partitioning and clustering", IEEE Transactions on CAD .

[4] E. Biegert and N. Konopliv, "Spectral decomposition using the Lanczos method", Thesis

[5] H D Simon "The Lanczos Algorithm With Partial Reorthogonalization", mathematics of computation volume 42, number 165 january 1984,

[6] H. Simon. "Analysis of the symmetric Lanczos algorithm with reorthogonalization methods" Linear Algebra Appl., 61:101-132, 1984.

[7] http://www.netlib.org/utk/people/JackDongarra/etemplates/node110.htm

[8] http://networkrepository.com/soc-karate.php

[9] https://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html

[10] https://sites.cs.ucsb.edu/~gilbert/cs290hSpr2014/Projects/BiegertKonoplivProject.pdf

[11] https://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter4.pdf

[12] Earl R. Barnes "An algorithm for partitioning the nodes of a graph", SIAM J. Alg. Disc. Meth. Vol. 3, No. 4, December 1982.

[13] Lanczos, C."An iteration method for the solution of the eigenvalue problem of linear differential and integral operators" JRNBS (1950).

[14] S.K.KimA.T.Chronopoulos, "A class of Lanczos-like algorithms implemented on parallel computers", Parallel Computing Volume 17, 1991