Using Graph Theory to Improve Some Algorithms in Scientific Computing

Suely Oliveira

Department of Computer Science, The University of Iowa, Iowa City, Iowa 52242, USA

1. Introduction

Lately graph theory along with data structures have played a special role in the development of algorithms for various problems in computational science. In this short paper I describe briefly two applications in which the representation of the problem in graphical language was insightful for the development of numerical method. The first problem (Sections 1 through 3) is about the tridigonalization of arrowhead matrices. The operations for transforming a matrix from one form to another are commonly referred to as chasing steps. In a previous paper [11] we represented the chasing steps as graphical operations, which allowed us to design a more parallelized algorithm for achieving our tridiagonal form matrix. The second problem (Sections 4 and 5) involves the use of graphical information for developing a preconditioner for a subspace eigensolver. The eigensolver is used here for finding a heuristic for the graph partitioning problem. We use the nature of the problem solved to design an efficient preconditioner for our eigensolver method. This work was published in [7].

2. A New Parallel Method for A Chasing Transformation Algorithm

The problem of finding eigenvalues of arrowhead matrices arise in molecular physics [9] and other applications. One approach for solving these problems is to reduce the arrowhead matrices to tridiagonal form while still preserving the eigenvalues and symmetry of the matrix. Then, one can apply the well-known symmetric QR algorithm to the resulting tridiagonal matrices. Gragg and Harrod [5] developed a chasing algorithm with complexity $6n^2$. Zha [13] proposed a two-way algorithm to reduce the complexity further to $3n^2$. In [11] the author presented a new algorithm, which has the same complexity as Gragg and Harrod's algorithm, but can be more efficiently implemented on parallel architectures. This algorithm was developed by using a graphical representation of the chasing method for matrix transformation. In Sections 2 and 3 of this paper we describe the old and new methods for transforming arrowhead matrices into tridiagonal form and show how we were able to design the faster new pipelined algorithm by looking at the problem graphically.

3. Algorithm

Chasing algorithms are standard ways of implementing the QR algorithm for eigenvalue computations. These are done by using Givens rotations, which are matrices that equal

to the identity matrix except for 2×2 submartix. Givens rotation have the form:

$$\begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c & \cdots & s & & \\ & & & \vdots & \ddots & \vdots & & \\ & & & -s & \cdots & c & & \\ & & & & & & 1 & \\ & & & & & & & \ddots \\ & & & & & & & & 1 \end{bmatrix}$$

where $c^2 + s^2 = 1$. Givens matrices G are orthogonal ($G^T G = I$), and so algorithms based on Givens matrices usually have good numerical stability properties. A Givens operation on a matrix A is the operation of computing $A' = G^T AG$ for a suitable Givens matrix G. Since $G^T = G^{-1} AG$ is a similarity transformation and consequently preserves the symmetry and the eigenvalues of A.

To compute the eigenvalues of a symmetric matrix, the matrix is first transformed to tridiagonal form. Then using appropriate Givens operations, a new-zero entry is introduced into the matrix, outside the tridiagonal band. This additional entry is then chasing along, and parallel to, the tridiagonal band by applying a sequence of Givens rotation to the matrix. Chasing algorithms have also been used for transforming matrices, such as banded and arrowhead matrices, into tridiagonal matrices. However, all these algorithms so far had used the standard chasing step. In our work we concentrated on the implementation and performance issues for a new chasing algorithm for transforming arrowhead matrices to tridiagonal matrices. The existence of our new chasing algorithm was discovered by analyzing the graph structure of the standard chasing step.

The numerical computations performed in the *standard chasing step* are,

$$\begin{bmatrix} \alpha' & \theta' & 0 & 0 \\ \theta' & \beta' & \lambda' & \nu' \\ 0 & \lambda' & \delta' & \eta' \\ 0 & \nu' & \eta' & \gamma' \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & c & s & \\ & -s & c & \\ & & & 1 \end{bmatrix} \begin{bmatrix} \alpha & \theta & \mu & 0 \\ \theta & \beta & \lambda & 0 \\ \mu & \lambda & \delta & \eta \\ 0 & 0 & \eta & \gamma \end{bmatrix} \begin{bmatrix} 1 & & & \\ & c & -s & \\ & s & c & \\ & & & 1 \end{bmatrix}. \tag{3.1}$$

The standard chasing step and the new chasing algorithm are illustrated graphically in Figure 1. The Greek letters at the nodes of the graph indicates the values of the diagonal entries in matrix while the Greek letters at the edges of the graph correspond to the off-diagonal entries. The value at node i represents a_{ii}, and the value associated with edge i j is $a_{ij} = a_{ji}$. A dark edge between nodes i and j indicates that the Givens operation is applied to rows and columns i and j. A dashed line between node i and j indicated that before the Givens rotation, $a_{ij} \neq 0$, but after the operation. If the edge between nodes i and j is marked with a cross (x) then it indicates that $a_{ii} \neq 0$ before the Given operation, but after $a'_{ii} = 0$ the Givens operation.

If the chasing steps are presented in matrix form, applying new chasing algorithm to a 6×6 matrix will correspond to the following steps shown in Figure 2. In this figure, "*" indicates a non-zero entry, "+" indicates a newly created non-zero entry, and "0" indicates an entry that has just been made zero.

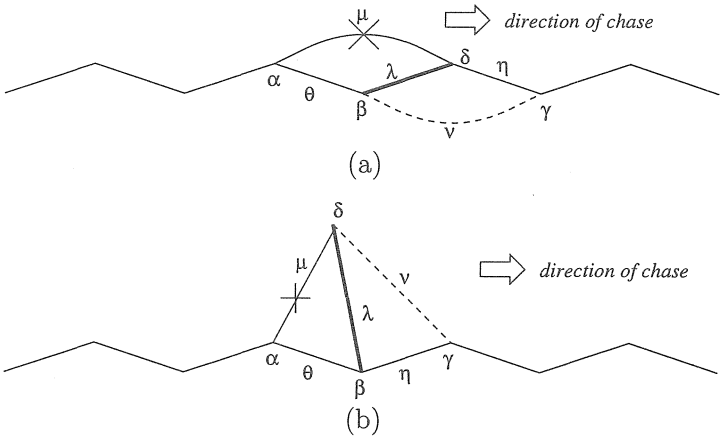
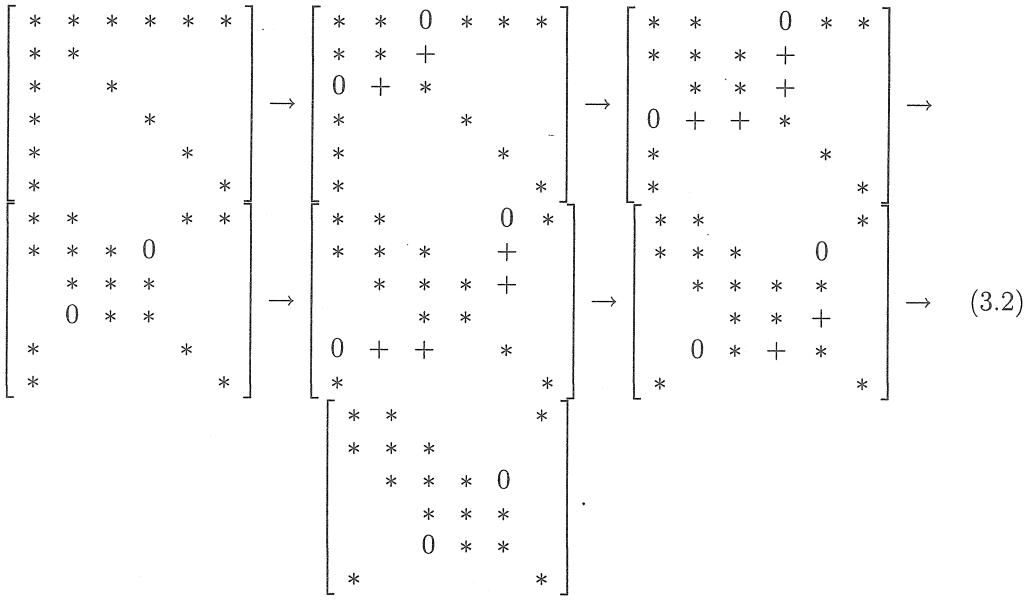


FIGURE 1. Chasing algorithms: (a) standard (b) new



(3.2)

FIGURE 2. Description of new chasing algorithm for a 6×6 matrix

Similarly to the standard chasing step operations in the new chasing step requires 29 flops plus a square root [11]. From Figure 1 we can also see that the new algorithm can be thought of as chasing a node along the path, while the standard algorithm chases an edge along the path. The main feature of the new chasing step is that it allows the chasings to have less interaction with the main path thus the density of bulges (triangles on Figure 1)) on the main path is higher. This benefits pipelined parallel algorithms.

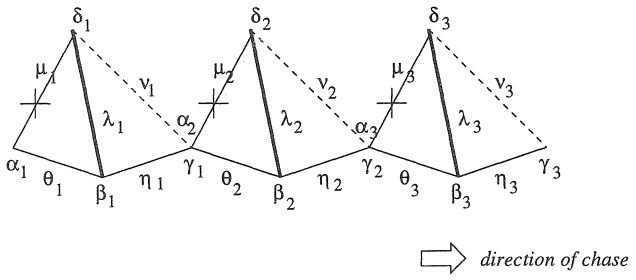


FIGURE 3. Simultaneous steps for the new chasing algorithm

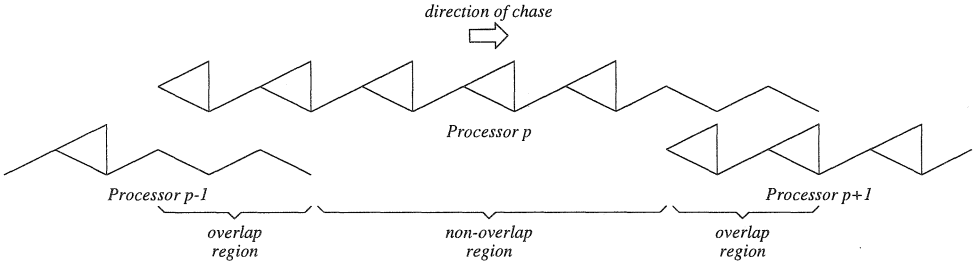


FIGURE 4. Partitioning of data between processors

We use the pipelined parallel algorithm presented in [11], for this new chasing scheme on $O(n)$ processors with $O(n)$ time complexity for tridiagonalizing an arrowhead matrix. The basis for the pipeline technique is the ability to chase multiple bulges along a path simultaneously, as illustrated in Figure 3.

The new chasing step can be performed simultaneously since the only matrix entries that might be affected by more than one of the parallel chasing steps are those on the main path and they are not changed by a chasing step (as $\alpha' = \alpha$ and $\gamma' = \gamma$), the chasing steps shown can be done simultaneously.

Parallel algorithms implementations demands attention to problems of overhead in message passing. This is particularly true for this pipeline algorithm since the natural message size (six entries per bulge) is quite small. Thus the block version implementation of the algorithm is used and implemented with a message passing library software (MPI). The block region version idea for this algorithm is represented in Figure 4.

The sequence of operations on processor p_i is as follows:

Repeat steps 2-5 until there are no bulges to chase:

1. Chase $\lceil l/2 \rceil$ bulges from the left-hand overlap region into the non-overlap region, and from the non-overlapping region into the right-hand overlap region, in processor p_i.
2. Send $\lceil l/2 \rceil$ bulges from the right-hand overlap region in the processor p_i to the left-hand overlap region in processor p_{i+1}.
3. Chase off $\lceil l/2 \rceil$ bulges from the right hand overlap region passing the right-most edge of this region, treating the end of this overlap region as the end of the matrix. The node in the bulges can be ignored once they are chased past the end of the right-most overlap region, since down stream entries do not affect upstream entries.

4. Receive $\lceil l/2 \rceil$ bulges from the right-hand overlap region in processor p_{i-1} into the left -hand overlap region in processor p_i.

4. Implementation

Three one-dimensional arrays are used to represent an arrowhead matrix. One for row one (or column one) and two for the tridiagonal entries since the matrix is symmetric.

Processors are connected like an assembly line as shown in Figure 4. The load is partitioned across the processors as indicated in Figure 5.

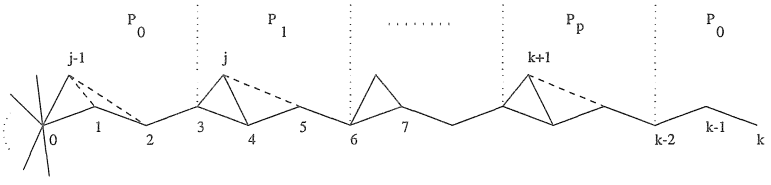


FIGURE 5. Parallel Data Distribution in Graph Notation

Each processor will start passing bulges to next the processor when it starts processing the maximum number of bulges ($\lceil k/2 \rceil$) allowed per processor. The processor will receive new bulges from the left processor as it passes bulges to the next processor on the right.

There are ($\lceil m/2 \rceil$) overlap bulges (chasing step) between two processors. To keep data consistency in the overlap region (in circles), the processor p_{i+1} should pass his overlap information back to processor p_i. Nevertheless in practice this backward passing of information can be avoided by making processor p_i chase the bulges to the end of its own path even though it has sent it to next processor in the beginning of the overlap area. This extra chasing to the edge of the processor main path ensures consistency of the information in the overlap region.

We can partition load such that we maximize processor utilization and minimize processor communication. To achieve these goals we implemented three versions of the algorithm to analyze the effectiveness of techniques such as wrap around and message group passing.

Implementation 1: The matrix is evenly divided by the number of processors being used. Each processor is responsible for a particular portion of matrix during the calculation.

Implementation 2: To maximize the processor utilization and to minimize processor idle time, a wrap around of paths (cyclic filling of the pipe) is used. This way processors waiting to receive their first bulge will become active earlier.

Implementation 3: To reduce communication time between processors, the information from several bulges is combined for passing them to other processors.

Chasing cycle: In a chasing cycle two entries will be chased off to the tridiagonal, one along the row and the other along the column. This is repeated $n-2$ times for a matrix of size $n \times n$. Each chasing cycle with have a different number of chasing steps. The whole process starts with one chasing step for the first chasing cycle and ends with $n-2$ chasing steps for the last one. A global view of the chasing cycle is shown in in Fig. 7. Each processor is responsible for certain columns of matrix. Initially there is only one processor which is active since there is only one chasing step in the first chasing cycle. In

the following subsequent chasing cycles the number of chasing steps for each new chasing cycle considered increases one by one. The processor will become active whenever the chasing cycle length reaches that processor.

5. Spectral Partitioning and Eigensolvers

Many algorithms have been developed to partition a graph into k parts such that the number of edges cut is small. This problem arises in many areas including finding fill-in reducing permutations of sparse matrices and mapping irregular data structures to nodes of a parallel computer.

The graph Laplacian of graph G is $L = D - A$, where A is G's adjacency matrix and D is a diagonal matrix where $d_{i,i}$ equals to the degree of vertex v_i of the graph. One property of L is that its smallest eigenvalue is 0 and the corresponding eigenvector is $(1, 1, \ldots, 1)$. If G is connected, all other eigenvalues are greater than 0. Fiedler [3, 4] explored the properties of the eigenvector associated with the second smallest eigenvalue. (These are now known as the "Fiedler vector" and "Fiedler value," respectively.) Spectral methods partition G based on the Fiedler vector of L. It is well known that for any vector x

$$x^T L x = \sum_{(i,j) \in E} (x_i - x_j)^2,$$

holds (see for example Pothen et al. [12]).
Note that there is one term in the sum for each edge in the graph. Consider a vector x whose construction is based upon a partition of the graph into subgraphs P_1 and P_2. Assign $+1$ to x_i if vertex v_i is in partition P_1 and -1 if v_i is in partition P_2. Using this assignment of values to x_i and x_j, if an edge connects two vertices in the same partition then $x_i = x_j$ and the corresponding term in the Dirichlet sum will be 0. The only non-zero terms in the Dirichlet sum are those corresponding to edges with end points in separate partitions. Since the Dirichlet sum has one term for each edge and the only non-zero terms are those corresponding to edges between P_1 and P_2, it follows that $x^T L x = 4 *$ (number of edges between P_1 and P_2).

An x which minimizes the above expression corresponds to a partition which minimizes the number of edges between the partitions. The graph partitioning problem has been transformed into a discrete optimization problem with the goal of

- minimizing $\frac{1}{4} x^T L x$
- such that
  1. $e^T x = 0$ where $e = (1, 1, \ldots, 1)^T$
  2. $x^T x = n$.
  3. $x_i = \pm 1$

Condition (1) stipulates that the number of vertices in each partition be equal, and condition (2) stipulates that every vertex be assigned to one of the partitions. If we remove condition (3) the above problem can be solved using Lagrange multipliers.

We seek to minimize $f(x)$ subject to $g_1(x) = 0$ and $g_2(x) = 0$. This involves finding Lagrange multipliers λ_1 and λ_2 such that

$$\nabla f(x) - \lambda_1 \nabla g_1(x) - \lambda_2 \nabla g_2(x) = 0.$$

For this discrete optimization problem, $f(x) = \frac{1}{2}x^T Lx$, $g_1(x) = e^T x$, and $g_2(x) = \frac{1}{2}(x^T x - n)$. The solution must satisfy

$$Lx - \lambda_1 e - \lambda_2 x = 0.$$

That is, $(L - \lambda_2 I)x = \lambda_1 e$. Premultiplying by e^T gives

$$e^T Lx - \lambda_2 e^T x = e^T e\, \lambda_1 = n\, \lambda_1.$$

But $e^T L = 0$, $e^T x = 0$ so $\lambda_1 = 0$. Thus

$$(L - \lambda_2 I)x = 0$$

and x is an eigenvector of L.

The above development has shown how finding a partition of a graph which minimizes the number of edges cut can be transformed into an eigenvalue problem involving the graph Laplacian. This is the foundation of spectral methods.

6. Our Graph Based Eigensolver

We used the Davidson algorithm [2, 1, 10] as our eigensolver. The Davidson algorithm is a subspace algorithm which iteratively builds a sequence of nested subspaces. A Rayleigh-Ritz process finds a vector in each subspace which approximates the desired eigenvector. If the Ritz vector is not sufficiently close to an eigenvector then the subspace is augmented by adding a new dimension and the process repeats. The Davidson algorithm allows the incorporation of a preconditioner. We used the structure of the graph in the development of our Davidson preconditioner. More details about this algorithm is given in [7, 6]. Here we outline the parts of the algorithm which used graph theory.

6.1. Graphical Preconditioner We developed a new preconditioner to improve the rate at which the Davidson algorithm converges to the Fiedler vector for that graph. We will refer to our new preconditioned Davidson Algorithm as PDA. It operates in a manner similar to multigrid methods for solving discretizations of PDE's [8]. However, our preconditioner differs from these methods in that we do not rely on obtaining coarser problems by decimating a regular discretization. Our method works with irregular or unknown discretizations because it uses coarser graphs for the multilevel framework. The multilevel representation of the input graph G_0 consists of a series of graphs $\{G_0, G_1, \ldots, G_n\}$ obtained by successive coarsening. Coarsening G_i is accomplished by finding a maximum matching of its vertices and then combining each pair of matched vertices to form a new vertex for G_{i+1}; unmatched vertices are replicated in G_{i+1} without modification. Connectivity of G_i is maintained by connecting two vertices in G_{i+1} with an edge if, and only if, their constituent vertices in G_i were connected by an edge. The coarsening process concludes when a graph G_n is obtained with sufficiently few vertices. The next subsection describes another aspect of our preconditioned Davidson algorithm which resourced to graphical ideas [7].

6.2. Locality of Memory References In our method, no assumption was made about the regularity of the input graphs. Consequently, the manner in which the coarse graphs are constructed results in data structures with irregular storage in memory. The irregular storage of data structures has the potential of reducing locality of memory accesses and thereby reducing the effectiveness of cache memories.

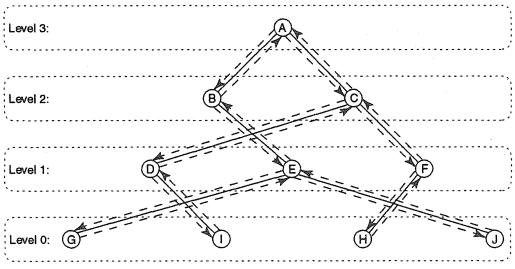


FIGURE 6. Storage of data structures before reordering

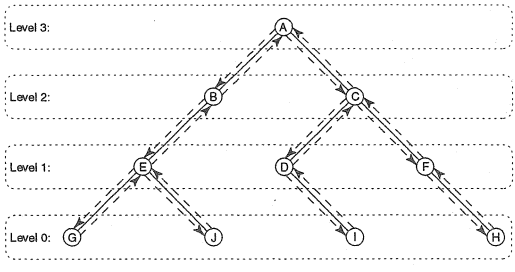


FIGURE 7. Storage of data structures after reordering

We developed a method called Multilevel Graph Reordering (MGR) which uses the multilevel representation of G_0 to reorder the data structures in memory to improve locality of reference. Intuitively, we permute the graph Laplacian matrices to increase the concentration of non-zero elements along the diagonal. This improved locality of reference during relaxation operations which represented a major portion of the time required to compute the Fiedler vector.

The relabeling of the graph is accomplished by imposing a tree on the vertices of the graphs $\{G_0, G_1, \ldots, G_n\}$. This tree was traversed in a depth-first manner. The vertices were relabeled in the order in which they were visited. After the relabelling was complete the data structures were rearranged in memory such that the data structures for the i^{th} vertex are stored at lower memory addresses than the data structures for the $i + 1^{\text{st}}$ vertex.

An example of reordering by MGR is shown in Figures 6 and 7. The vertices are shown as well as the tree overlain on the vertices. (The edges of the graphs are not shown.) If a vertex lies to the right of another in the figure, then its data structures occupy higher memory addresses. Notice that after reordering, the vertices with a common parent are placed next to each other in memory. This indicates that they are connected by an edge and will be referenced together during relaxation operations. Relaxation operations represented a large fraction of the total work done by our algorithms. This reordering has a positive effect of locality of reference during such operations.

The numerical results of [7] showed that our strategies were successful in improving purely spectral partitioning algorithms.

References

[1] L. Borges and S. Oliveira, *A parallel Davidson-type algorithm for several eigenvalues*, Journal of Computational Physics, 144 (1998), pp. 727–748.

[2] E. Davidson, *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices*, Journal of Computational Physics, 17 (1975), pp. 87–94.

[3] M. Fiedler, *Algebraic connectivity of graphs*, Czechoslovak Mathematical Journal, 23 (1973), pp. 298–305.

[4] ——, *A property of eigenvectors of non-negative symmetric matrices and its application to graph theory*, Czechoslovak Mathematical Journal, 25 (1975), pp. 619–632.

[5] W. Gragg and W. Harrrod, *The numerically stable reconstruction of jacobi matrices from spectral data*, Numer. Math., 44 (1984), pp. 317–335.

[6] M. Holzrichter and S. Oliveira, *New graph partitioning algorithms*, (1998). The University of Iowa TR-120.

[7] M. Holzrichter and S. Oliveira, *A graph based Davidson algorithm for the graph partitioning problem*, International Journal of Foundations of Computer Science, 10 (1999), pp. 225–246.

[8] S. McCormick, *Multilevel Adaptive Methods for Partial Differential Equations*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1989.

[9] D. O'Leary and G. Stewart, *Computing the eigenvalues and eigenvectors of symmetric arrowhead matrices*, J. Comp. Phys., 90 (1990), pp. 497–505.

[10] S. Oliveira, *A convergence proof of an iterative subspace method for eigenvalues problem*, in Foundations of Computational Mathematics Selected Papers, F. Cucker and M. Shub, eds., Springer, January 1997, pp. 316–325.

[11] ——, *A new parallel chasing algorithm for transforming arrowhead matrices to tridiagonal form*, Mathematics of Computation, 67 (1998), pp. 221–235.

[12] A. Pothen, H. D. Simon, and K. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430–452. Sparse matrices (Gleneden Beach, OR, 1989).

[13] H. Zha, *A two-way chasing scheme for reducing an arrowhead matrix to tridiagonal form*, J. Numer. Lin. Alg. Appl., 1 (1992), pp. 49–57.