# Benchmarking Parallel Eigen Decomposition for Residuals Analysis of Very Large Graphs

Edward M. Rutledge, Benjamin A. Miller, and Michelle S. Beard
Lincoln Laboratory
Massachusetts Institute of Technology
Lexington, MA 02420
{rutledge, bamiller, michelle.beard}@ll.mit.edu

***Abstract*—Graph analysis is used in many domains, from the social sciences to physics and engineering. The computational driver for one important class of graph analysis algorithms is the computation of leading eigenvectors of matrix representations of a graph. This paper explores the computational implications of performing an eigen decomposition of a directed graph's symmetrized modularity matrix using commodity cluster hardware and freely available eigensolver software, for graphs with 1 million to 1 billion vertices, and 8 million to 8 billion edges. Working with graphs of these sizes, parallel eigensolvers are of particular interest. Our results suggest that graph analysis approaches based on eigen space analysis of graph residuals are feasible even for graphs of these sizes.**

## I. Introduction

A graph consists of a pair of sets: a set of vertices, $V$, denoting entities, and a set of edges, $E$, which connect the vertices. Graphs are used to model networks in many domains, from the social sciences to physics and engineering. A number of problems that arise in these domains can be cast as the problem of finding anomalous subgraphs within a larger graph. The eigen decomposition of a graph's "modularity matrix" is the computational driver for a class of algorithms that aim to find subgraphs whose structure differs significantly from that described by a random graph model, such as the community detection algorithms described in [1], and the anomaly detection algorithms described in [2], [3]. The utility of these algorithms has been demonstrated for graphs generated both synthetically and from real-world data [2]–[4].

As the ability to collect and store ever-increasing amounts of data results in ever-larger graphs, it is natural to ask how well these algorithms scale. For graphs with $|V|$ vertices and $|E|$ edges, iterative methods exist for finding the first $m$ eigenvectors in $O((m|E| + m^2|V| + m^3)h)$, where $h$ is the number of iterations of the method [5], but what performance do these algorithms have in practice, and what are the practical limits of applying these algorithms to large data sets using state-of-the-art technology? This paper explores that question, providing benchmarking results for computation of the eigen decomposition of the modularity matrix of graphs with one million to one billion nodes, using commodity cluster hardware and freely available eigensolver software.

This work is sponsored by the Intelligence Advanced Research Projects Activity (IARPA) under Air Force Contract FA8721-05-C-0002. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.



The remainder of this paper is organized as follows. Section II gives an overview of the algorithm we consider in this paper: the eigen decomposition of a directed graph's symmetrized modularity matrix. Section III describes the implementation of this algorithm for which we present benchmarking results. Section IV describes the benchmarking experiments. Section V presents some of our benchmarking results. Section VI concludes and presents potential future work.

## II. Algorithm Overview

Graph analysis algorithms such as those described in [1], [2] rely on finding eigenvectors of the graph's modularity matrix. The modularity matrix is essentially an expression of how a given graph's structure deviates from that expected under a random model in which the likelihood of a potential edge's existence is proportional to the product of the degrees of the vertices connected by the edge. The modularity matrix of an unweighted, directed graph is given by

$$\hat{B} = A - k_{out}k_{in}^T/|E|,$$

where $A$ is the graph adjacency matrix, $k_{out}$ is the out-degree vector (a vector whose $i$th entry is the out-degree of the $i$th vertex), $k_{in}$ is the in-degree vector (a vector whose $i$th entry is the in-degree of the $i$th vertex), and $|E|$ is the total number of edges in the graph. As in [6], this paper considers the "symmetrized" modularity matrix $B = (\hat{B} + \hat{B}^T)/2$, or

$$B = (A - k_{out}k_{in}^T + A^T - k_{in}k_{out}^T)/2|E|.$$

Note that for an undirected graph, in which an edge from vertex $u$ to vertex $v$ exists only if there is an edge from $v$ to $u$, $A$ is symmetric and $k_{out} = k_{in}$. Thus, $B = \hat{B}$ (and $\hat{B}$ is symmetric), so the problem of computing the eigen decomposition of $B$ reduces to the (easier) problem of computing the eigen decomposition of symmetric $\hat{B}$.

In this paper, we consider the computation of the largest magnitude eigenvectors of the symmetrized modularity matrix

$B$ (i.e., the space of largest positive residuals) for very large randomly generated directed graphs.

## III. Implementation

In practically all efficient algorithms for eigenvector computation, the matrix being decomposed is only accessed through matrix-vector multiplication. As described in [1], [4], multiplication of a vector by the symmetrized modularity matrix can be implemented efficiently as

$$\begin{aligned} Bx = Ax - k_{out}((k_{in}^T x)/2|E|) \\ +A^T x - k_{in}((k_{out}^T x)/2|E|). \end{aligned} \quad (1)$$

The dense matrix $B$ does not need to be stored to compute $Bx$. Only the sparse adjacency matrix $A$, the degree vectors $k_{out}$ and $k_{in}$, and the count of edges $|E|$ must be stored. The operation can be implemented as 2 sparse matrix-vector multiplications, 2 vector dot products, 2 scalar-vector multiplications, and 3 vector additions.

There are over a dozen freely available eigensolver software packages. A survey is given in [7]. For our application, we desire a parallel eigensolver that can utilize commodity cluster hardware. Using a cluster allows us to solve larger problems, and allows us to solve problems more quickly, than using a single workstation. To achieve good performance and to scale to large problem sizes in our application, the eigensolver package must allow us to implement efficient multiplication of a vector by the symmetrized modularity matrix as in (1). One eigensolver package that meets these criteria is the Scalable Library for Eigenvalue Problem Computations (SLEPc), a C library developed by the High Performance Networking and Computing Group (GRyCAP) of Universitat Politècnica de València (Spain) [8]. We used SLEPc for our initial implementation, which we describe and for which we present benchmarking results in this paper. Other suitable eigensolver packages exist as well, such as Anasazi [9], and we may evaluate and benchmark some of these other packages for our application in the future.

SLEPc is layered on top of the Portable, Extensible Toolkit for Scientific Computation (PETSc), developed by Argonne National Laboratory [10]. SLEPc supports any PETSc matrix type, including sparse distributed matrices and "matrix shells." PETSc sparse distributed matrices may be row-distributed across multiple processes of an MPI program. (MPI is the Message Passing Interface standard [11].) Computations on these matrices are distributed among the processes according to which data are local. PETSc "matrix shells" allow users of the library to define matrices with non-standard storage and operations. In our case, we used the matrix shell facility to define a modularity matrix type that stores the sparse, distributed graph adjacency matrix $A$, the distributed degree vectors $k_{in}$ and $k_{out}$, and the count of edges in the graph $|E|$; and implements matrix-vector multiplication as defined in (1). When we apply a SLEPc eigensolver to a modularity matrix, SLEPc invokes the matrix-vector multiplication operation as part of the eigensolver method, and the computation is distributed across the processes of the MPI program according to the distribution of $A$, $k_{in}$, and $k_{out}$. As we show in the experimental results, this distribution of storage and data allows us to solve larger problems and solve problems faster on a cluster than a single-workstation implementation.

In our implementation, we used the default PETSc and SLEPc settings. In particular, we used the default SLEPc algorithm for computing eigenvectors: the Krylov-Shur method [12]. We also used SLEPc's default stopping condition of

$$\|Bx - \lambda x\|_2 / \|\lambda x\|_2 \leq 10^{-8},$$

where $B$ is the symmetrized modularity matrix for which we are computing eigenvectors, $x$ is a computed eigenvector, and $\lambda$ is the corresponding eigenvalue. (The condition must hold for each computed eigenvector and its corresponding eigenvalue).

## IV. Experiments

We benchmarked the modularity matrix eigen decomposition implementation described above on Lincoln Laboratory's LLGrid cluster for random graphs of various sizes. The computing platform, data sets, and parameter space of the experiments are described below.

### A. Computing Platform

All experiments were run on Lincoln Laboratory's LLGrid TX-2500 cluster [13]. This cluster has about 550 Dell PowerEdge 2850 compute nodes, each with dual 3.2 GHz Intel Xeon processors and 8 GB RAM. However, the system scheduler limits jobs to no more than 64 nodes. To maximize the amount of memory available for our jobs, we scheduled our experiments so that a single MPI process would be mapped to each node, and so that no other user jobs were allowed to run on the nodes we used. LLGrid's gigabit Ethernet network was used by MPI for inter-node communications.

### B. Data Sets

The random graphs for which which we performed eigen decomposition of the modularity matrix were generated using the R-MAT algorithm [14]. R-MAT produces graphs that are, in many respects, similar to those that arise in real-world situations. The R-MAT probability matrix used was

$$\begin{pmatrix} 0.5 & 0.125 \\ 0.125 & 0.25 \end{pmatrix}.$$

The average in- (and out-) degree was slightly less than 8. (It was not exactly 8 because we discarded edges that collided with existing edges instead of trying to place them again.) The number of vertices ranged from $2^{20}$ to $2^{30}$ (or about one million to one billion). To simplify load balancing in the eigenvector computation, we randomly permuted the vertex labels of the generated graphs. This has the effect of more uniformly distributing edges across the rows and columns of the graph's adjacency matrix, but has no effect on the graph's topology.

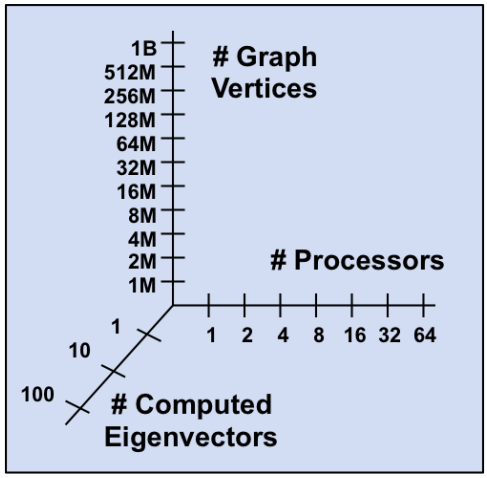


Fig. 1. Experiment Parameter Space

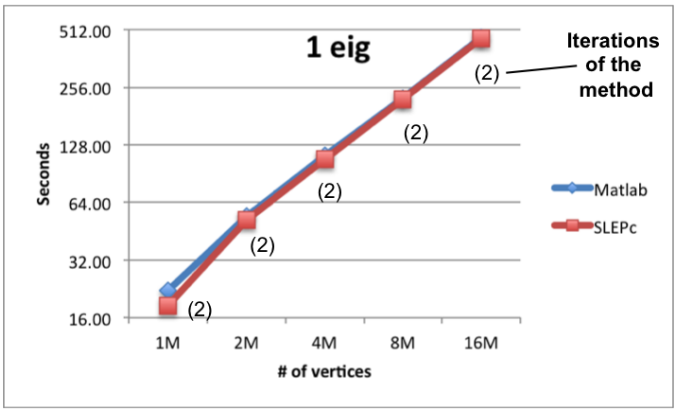


Fig. 2. Running time of SLEPc and Matlab implementations computing a single eigenvector of various size graphs using a single compute node

### C. Parameter Space

Figure 1 shows the parameter space explored in the experiments. The number of processors ranged from 1 to 64 in powers of 2. As mentioned earlier, we were limited to using 64 processors by LLGrid's scheduler. The number of graph vertices ranged from $2^{20}$ (approximately one million) to $2^{30}$ (approximately one billion) in powers of 2. For each number of processors and number of vertices, we attempted to compute the eigenvectors corresponding to the 1, 10, and 100 largest magnitude eigenvalues. Although the number of eigenvectors computed is a small fraction of the total, computing even a small number of leading eigenvectors can be useful in graph analysis. A number of the experiments failed due to insufficient memory on the computing hardware. For example, we were only able to process the billion-vertex data set on 64 processors, and were unable to compute more than 2 eigenvectors in that case. Again, the average in- (and out-) degree of the vertices was approximately 8, so the number of edges in the graphs in our experiments was approximately 8 times the number of vertices.

## V. Results

This section briefly presents a few of the experimental results, focusing on comparison of our SLEPc implementation to a Matlab implementation on a single compute node, how performance scales as compute nodes are added, and how our implementation performs when running on 64 compute nodes, the maximum allowed by our cluster's scheduler.

### A. Comparison to Matlab

To determine whether our SLEPc implementation has reasonable performance in the single-processor case, we compared its performance to Matlab's on a single processor. Our Matlab implementation makes use of the eigs function, passing a function implementing efficient multiplication by the symmetrized modularity matrix, as in (1), to eigs instead of computing and passing the modularity matrix to eigs directly. We also set the tolerance to $10^{-8}$ to mirror our SLEPc stopping condition. Thus, our Matlab implementation is computationally similar to our SLEPc implementation. We also used this Matlab implementation as a reference to verify that our SLEPc implementation was outputting the correct results.

A comparison of the performance of our Matlab and SLEPc implementations for computing a single eigenvector on a single processor is shown in Figure 2. Running time averaged across number of trials is on the vertical axis, and number of vertices is on the horizontal axis. For the SLEPc implementation, the number of iterations of the Krylov-Shur method needed to converge is indicated by the parenthetical numbers next to the data points. (Number of iterations was not reported by the Matlab implementation.) Both axes have a $\log_2$ scale. Neither implementation was able process a graph with greater than 16M nodes on a single processor. The performance of the SLEPc and Matlab implementations is nearly identical; SLEPc is typically a few seconds faster. This holds true for 10 and 100 eigenvalues as well. The main takeaway from this experiment is that the Matlab and SLEPc implementations perform similarly for the single processor case.

### B. Scaling the Number of Processors

Figure 3 shows how running on different numbers of compute nodes affects the time required to compute the leading eigenvector of a 16M vertex graph. As one might expect, nearly linear speedup is achieved compared to the single processor case when the processor count is low, but speedup begins to tail off for greater numbers of processors as the fraction of running time required for inter-processor communication increases. For graphs with $2^{23}$ or more vertices, we found that the best performance was always obtained running on 64 nodes (the maximum allowed in our system). For smaller graphs, better performance was often achieved running on fewer nodes, depending on the number of eigenvectors computed. For example, when computing only the leading eigenvector of a $2^{20}$ vertex graph, we observed the best performance running on 4 compute nodes. This is because the savings from distributing the computation among more nodes

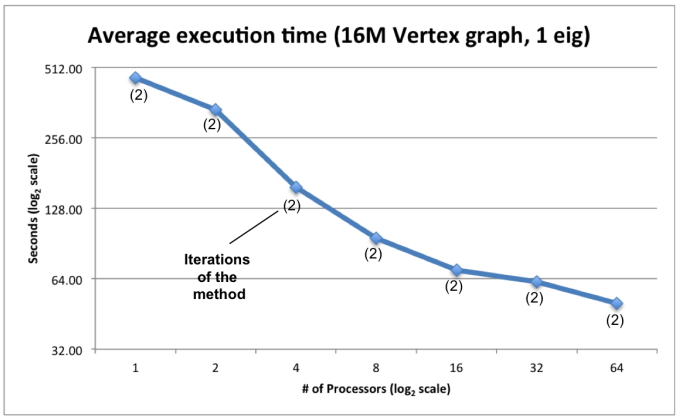


Fig. 3. Running time for calculating a single eigenvector of a 16M vertex graph using different numbers of compute nodes

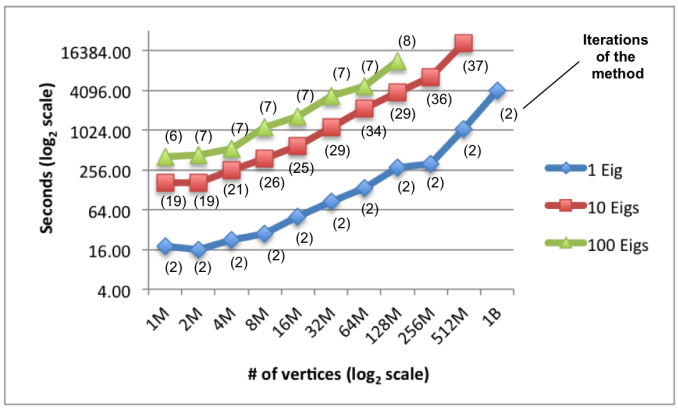


Fig. 4. Running time for computing 1, 10, and 100 eigenvectors on various size graphs using 64 compute nodes

does not offset the additional time required for inter-processor communications in this case.

### C. Results on 64 Compute Nodes

Figure 4 shows average running times when running on 64 compute nodes and computing 1, 10, and 100 eigenvectors. Recall that the eigenvector computation has complexity $O((m|E| + m^2|V| + m^3)h)$, where $m$ is the number of eigenvectors and $h$ is the number of iterations of the method required to converge. In our experiments $|E| = 8|V|$, so one would expect the running time to scale approximately linearly with increasing graph size for a given number of eigenvectors, (depending on the number of iterations required). When running on 64 compute nodes, we observe that the average running time scales approximately linearly across much of the range of graph sizes we tested, but the average running time scales less than linearly for the smallest graphs, and greater than linearly for the largest graphs. Most likely, running time scales less than linearly for the smaller graphs because the ratio of computation to communication is relatively high for the smaller graph sizes. (Note that we do not see this effect in the single processor case.) More investigation is required to determine why the running time scales greater than linearly for the larger graphs; one potential explanation is cache effects.

We also observe that execution time increases much more dramatically when moving from 1 eigenvector to 10 eigenvectors than when moving from 10 eigenvectors to 100 eigenvectors. This is because, for our data sets, the number of iterations of the method required when computing 10 eigenvectors was between 19 and 37, compared to 2 iterations required to compute 1 eigenvector, and 6 to 8 iterations required to compute 100 eigenvectors. It is difficult to make any general conclusions about how our implementation scales with the number of eigenvectors, given that we only gathered a few data points in this dimension, and given that the number of eigenvectors was small compared to the graph sizes. The number of iterations is dependent on the eigenvalues of the matrix; specifically, the gap between consecutive eigenvalues. Thus, a formula for the iteration count is too complicated to express in terms of numbers of vertices and edges. The number of iterations increased when computing 10 eigenvectors versus 1 because for our data sets, the difference in magnitude of the first and second eigenvalue was much greater than the difference in magnitude of subsequent eigenvalues. For example, the 10 leading eigenvalues in the 64M vertex case were:

$$\begin{aligned}
\lambda_1 &= 85.403845 \\
\lambda_2 &= 41.146193 \\
\lambda_3 &= 41.093851 \\
\lambda_4 &= 40.993092 \\
\lambda_5 &= 40.963347 \\
\lambda_6 &= 40.907482 \\
\lambda_7 &= 40.854489 \\
\lambda_8 &= 40.824815 \\
\lambda_9 &= 40.765026 \\
\lambda_{10} &= 40.735158.
\end{aligned}$$

One plausible explanation for the number of iterations decreasing when computing 100 eigenvectors versus 10 is that the number of matrix-vector multiplications scales as $O(mh)$. So, even though there are fewer iterations, there are more multiplications, which drive the algorithm toward convergence.

Although not shown in Figure 4, we were also able to compute 2 eigenvectors of a 1 billion vertex graph when running on 64 compute nodes. This computation required 49 iterations of the method, and took about 32,500 seconds (or about 9 hours). Running on more than 64 nodes, we would have been able to compute more eigenvectors, and running times would have been lower for the larger graph sizes.

## VI. Conclusion and Future Work

In this paper we have described an implementation of eigen decomposition of a directed graph modularity matrix that uses the freely available SLEPc library, and presented benchmarking results for that implementation. We have shown that, using this implementation, it is feasible to apply the algorithms described in [1]–[3] to very large graphs, at least

up to 1 billion vertices and 8 billion edges, on commodity cluster hardware. We have demonstrated that our SLEPc-based implementation scales fairly well, and believe that we could scale to larger problems than the experiments we describe here by running on more than 64 compute nodes.

There are a number of potential directions for future work. As mentioned in Section III, SLEPc is just one of many freely available eigensolvers. Evaluating the performance of some other eigensolver libraries—in particular the Anasazi library—is a potential next step. Running our experiments on larger clusters so that we can scale to larger problem sizes is another potential next step. To do this, we would have to change our implementation somewhat, since we currently use 32-bit numbers to represent node vertex labels, and we are approaching graph sizes where the number of vertices cannot be represented in a 32-bit number. Another potential direction for future work is to further tailor our implementation to application. For example, our current implementation uses SLEPc's standard sparse matrix data structure to represent the adjacency matrix. This data structure stores a double-precision value for each edge in the graph, even though that value is 1 for every edge (since our graph is unweighted). Using a specialized data structure to represent our adjacency matrix would allow us to eliminate this unnecessary storage and potentially scale to larger problem sizes. Other possible avenues of investigation include incorporating preconditioners for applying these methods to other graph problems, such as finding the eigenvectors with smallest magnitude in the graph Laplacian, and considering more efficient partitioning schemes.

## Acknowledgments

The authors thank Nadya Bliss for her guidance and support as the Lincoln Laboratory manager of this effort. We also thank the LLGrid team for their support in running our experiments.

## References

[1] M. E. J. Newman, "Finding community structure in networks using the eigenvectors of matrices," *Phys. Rev. E*, vol. 74, no. 3, pp. 036 104–(1–19), Sep 2006.

[2] B. Miller, N. Bliss, and P. Wolfe, "Toward signal processing theory for graphs and non-Euclidean data," in *Proc. ICASSP*, 2010, pp. 5414–5417.

[3] B. A. Miller, N. T. Bliss, and P. J. Wolfe, "Subgraph detection using eigenvector L1 norms," in *Advances in NIPS*, J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., 2010, pp. 1633–1641.

[4] B. A. Miller, N. Arcolano, M. S. Beard, J. Kepner, M. C. Schmidt, N. T. Bliss, and P. J. Wolfe, "A scalable signal processing architecture for massive graph analysis," in *Proc. ICASSP*, 2012.

[5] S. White and P. Smyth, "A spectral clustering approach to finding communities in graphs," in *Proc. SIAM Int. Conf. Data Mining*, 2005.

[6] E. A. Leicht and M. E. J. Newman, "Community structure in directed networks," *Phys. Rev. Lett.*, vol. 100, no. 11, pp. 118 703–(1–4), Mar 2008.

[7] C. Campos, J. E. Roman, E. Romero, and A. Tomas, "A survey of software for sparse eigenvalue problems," Universitat Politècnica de València, Tech. Rep. STR-6, 2009, available at http://www.grycap.upv.es/slepc.

[8] V. Hernandez, J. E. Roman, and V. Vidal, "SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems," *ACM Trans. Math. Software*, vol. 31, no. 3, pp. 351–362, 2005.

[9] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist, "Anasazi software for the numerical solution of large-scale eigenvalue problems," *ACM Trans. Math. Software*, vol. 36, no. 3, pp. 13:1–13:23, Jul. 2009.

[10] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Web page," 2012, http://www.mcs.anl.gov/petsc.

[11] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable parallel programming with the message-passing interface*, 2nd ed. Cambridge, MA, USA: MIT Press, 1999.

[12] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal, "Krylov-Schur methods in SLEPc," Universitat Politècnica de València, Tech. Rep. STR-7, 2007, available at http://www.grycap.upv.es/slepc.

[13] A. Reuther, B. Arc, T. Currie, A. Funk, J. Kepner, M. Hubbell, A. McCabe, and P. Michaleas, "TX-2500 – An interactive, on-demand rapid-prototyping HPC system," in *Proc. High Performance Embedded Computing Worhshop*, 2007.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004.