A Distributed Block Chebyshev-Davidson Algorithm for Parallel Spectral Clustering

Qiyuan Pang[1] and Haizhao Yang *[2]

[1]Department of Mathematics, Purdue University, West Lafayette
[2]Department of Mathematics, University of Maryland, College Park

Abstract

We develop a distributed Block Chebyshev-Davidson algorithm to solve large-scale leading eigenvalue problems for spectral analysis in spectral clustering. First, the efficiency of the Chebyshev-Davidson algorithm relies on the prior knowledge of the eigenvalue spectrum, which could be expensive to estimate. This issue can be lessened by the analytic spectrum estimation of the Laplacian or normalized Laplacian matrices in spectral clustering, making the proposed algorithm very efficient for spectral clustering. Second, to make the proposed algorithm capable of analyzing big data, a distributed and parallel version has been developed with attractive scalability. The speedup by parallel computing is approximately equivalent to $\sqrt{p}$, where p denotes the number of processes. Numerical results will be provided to demonstrate its efficiency and advantage over existing algorithms in both sequential and parallel computing.

Keywords. Sparse symmetric matrices, distributed memory, spectral analysis, spectral clustering.

1 Introduction

Spectral clustering has a long history [1, 2, 3, 4, 5, 6, 7] and it was popularized as a machine learning model by Shi & Malik [1] and Ng, Jordan, & Weiss [7]. Spectral clustering makes use of the spectrum of the similarity matrix of the data to perform dimensionality reduction before clustering in fewer dimensions. The basic algorithm is summarized as follows:

1. Calculate the Laplacian or normalized Laplacian A from data.

2. Compute the first k eigenvectors corresponding to the smallest k eigenvalues of A.

3. Consider the feature matrix formed by the first k eigenvectors, where the i-th row defines the features of the i-th graph node.

4. Cluster the graph based on the features using clustering methods like k-means.

This paper focuses on solving the large-scale leading eigenvalue problem, which is the second step of spectral clustering. Given a symmetric matrix $A \in \mathbb{R}^{N\times N}$, the leading eigenvalue problem is to find

$$AV = V\Lambda_k, \tag{1}$$

where $\Lambda_k \in \mathbb{R}^{k\times k}$ is a diagonal matrix with $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_k$ as the k smallest eigenvalues of A along the diagonal, and $V \in \mathbb{R}^{N\times k}$ is a tall-skinny ($k << N$) matrix consisting of the corresponding eigenvectors. We assume A is sparse and a Laplacian or normalized Laplacian matrix throughout the paper.

The Chebyshev-Davidson method is a good candidate to solve the eigenproblem with fast convergence for spectral clustering, since the spectrum bounds of the Laplacian and normalized Laplacian matrices in

*corresponding author, email: hzyang@umd.edu

spectral clustering are known explicitly [2]. The Chebyshev-Davidson method was first introduced [8] as an eigensolver for quantum chemistry systems [9, 10, 11], which applies a Chebyshev polynomial filter to accelerate the convergence of the Davidson method [12]. The Davidson method can augment the searching subspace for eigenvectors by a potentially better new vector than the one based on a strict Krylov subspace structure, resulting in faster convergence. The augmentation vector added to the subspace at each step requires solving a correction equation that is not affordable for big data, even though the Jacobi-Davidson method [13] has been designed to favor the efficient use of modern iterative techniques for the correction-equation, based on preconditioning and Krylov subspace acceleration. Compared to all other types of Davidson-type methods, there is no need to form or solve any correction equations in the Chebyshev-Davidson method. Instead, interval-wise filtering based on Chebyshev polynomials is utilized, making it very suitable for large-scale sparse matrices with known bounds of the spectrum. The Chebyshev filter can enhance the eigensubspace of interest and dampen the eigensubspace undesired, making the Chebyshev-Davidson method efficiently applicable to general matrices, including the sparse symmetric matrices in spectral analysis. The Chebyshev-Davidson method could be extended to the Block Chebyshev-Davidson method [14] with an inner-outer restart technique to reduce total CPU time and a progressive polynomial filter to take advantage of suitable initial vectors when they are available. Good initials are available and essential in many data science scenarios. For example, when partitioning a streaming graph changing over time using spectral clustering, eigenpairs computed for the previous graph are good initials for evaluating the eigenpairs of the current graph.

Problems in data science areas like spectral clustering are usually large-scale, and parallel algorithms are a common practice to reduce CPU time. The multi-threading implementation for a single shared-memory node and the multi-processing/distributed implementation for distributed-memory nodes are standard parallelization techniques. Multi-threading algorithms are easy to implement, but their ability to handle large-scale problems and accelerate solutions is still restricted by the limited memory and number of threads in one single node. Multi-process algorithms are difficult to implement and scale due to the communication cost among processes; however, they could deal with large-scale matrices that cannot be stored or processed in a single node. Besides, with careful design, distributed algorithms could accelerate solutions much further.

Since matrices in problem (1) are usually too large for one single node, even though they are sparse in spectral clustering in practice, we propose a novel multi-processing Block Chebyshev-Davidson method to solve the leading eigenvalue problem. There are three essential components in the Block Chebyshev-Davidson method to be parallelized in distributed memory: sparse times tall-skinny matrix multiplication (SpMM), Chebyshev polynomial filter to tall-skinny matrices, and tall-skinny matrix orthonormalization. A multi-processing implementation of the Block Chebyshev-Davidson method could be found in PARSEC package [15] and it is used to solve the Kohn-Sham eigenvalue problem [16, 17, 18]. However, it is not scalable to large concurrencies due to its 1D parallel SpMM algorithm in the Chebyshev polynomial filters and the parallel DGKS [19] for orthonormalization. Unlike the well-studied parallel algorithms for dense matrix-dense matrix multiplication (GEMM) [20, 21, 22, 23] and sparse matrix-sparse matrix multiplication (SpGEMM) [24, 25, 26], the research of parallel SpMM algorithms has not flourished until recent years due to the needs in data science areas. The flops in parallel GEMM scale with N^3 for multiplying two $N \times N$ matrices, whereas the communication costs scale with N^2. This benefit of parallel GEMM is called the surface-to-volume ratio. Parallel SpGEMM and SpMM do not benefit from this favorable computation-to-communication scaling. In fact, it is easier to scale SpGEMM than to scale SpMM to large concurrencies [27]. For example, the 1D SpMM algorithm scales poorly to dozens of concurrencies. Oguz Selvitopi et al. [27] compare A-Stationary 1.5D [28], A-Stationary 2D, and C-Stationary 2D algorithms [26] and conclude that, unlike for GEMM and SpGEMM, 2D algorithms are not strongly scalable to large process counts for SpMM; instead, 1.5D achieves much better scaling. We adopt the A-Stationary 1.5D algorithm for SpMM and Chebyshev polynomial filter. In the A-Stationary 1.5D algorithm, the sparse matrix is partitioned in 2D while the tall-skinny matrix is partitioned in 1D. The resulting tall-skinny matrix is partitioned in 1D but in a different schema. Hence, the 1.5D algorithm generally does not apply to a Chebyshev polynomial filter which conducts multiple SpMMs and matrix-matrix additions sequentially. When the sparse matrix A is symmetric and square, we could solve the issue by transposing the 2D process grid and re-distributing the tall-skinny matrices.

Parallel orthonormalization is another issue because a parallel QR seldom scales due to communication costs. James Demmel et al. [29] compare multiple parallel QR algorithms and present a parallel Tall Skinny

QR (TSQR) algorithm, which attains known communication lower bounds and communicates as little as possible only up to polylogarithmic factors. The parallel TSQR algorithm leads to significant speedups in practice over some of the existing algorithms, including DGEQRF in LAPACK [30] and PFDGEQRF in ScaLAPACK [31]. We adopt the parallel TSQR for orthonormalization, which is also more efficient than the parallel DGKS. Though orthonormalization hardly scales to large concurrencies, orthonormalization only takes a small amount of computation cost compared to Chebyshev polynomial filters, especially when the ratio of the degree of the filter to the number of desired eigenvectors is large. Note that a higher ratio results in faster convergence as well. Therefore, the distributed Block Chebyshev-Davidson method is practically scalable and efficient for data science applications like spectral clustering. Table 1 summarizes the flops and communication costs of our distributed algorithm and the components.

Free software implementing spectral clustering is available in large open-source projects like Scikit-learn using LOBPCG [32] or ARPACK [33], and MLlib for pseudo-eigenvector clustering using the power iteration method [34]. Parallel versions of the eigensolvers mentioned are used in different parallel spectral clustering methods [35, 36, 37, 38]. Parallel LOBPCG is used in [35] for parallel spectral graph partitioning, and the speedup increases slowly as the number of processes becomes large. Numerical results in [36, 38] using up to 256 processes show that the parallel ARPACK used in parallel spectral clustering is accelerated at a rate proportional to the square root $\sqrt{p}$ of the number of processes p. The parallel power iteration method in parallel Power Iteration Clustering (p-PIC) [37] achieves linear speedups in one single node when the number of processes is smaller than 70, but then the speedups drop when the number of processes is larger. No numerical results are provided in [37] to demonstrate the scalability when the number of processes is larger than 90. The speedup of a parallel eigensolver is usually high with dozens of processes but becomes less significant when the number of processes keeps growing. In most of the works mentioned above, the scalabilities of the parallel eigensolvers are tested with less than 256 processes. In our work, besides theoretical proof, we will test the scalability of our algorithm with up to 1000 processes to demonstrate that the speedup is approximately the square root $\sqrt{p}$ when the number of processes p is large.

The contributions of this paper are summarized as follows:

- Besides LOBPCG, ARPACK, and the Power Iteration method, we propose to use the Block Chebyshev-Davidson method as an efficient eigensolver for spectral clustering due to the known analytic spectrum bounds.
- We develop a distributed Block Chebyshev-Davidson method which achieves approximately $\sqrt{p}$ speedup when the number of processes p is large.

Table 1: **Summary of the complexity per iteration of components in our distributed Block Chebyshev-Davidson algorithm for solving the problem (1) with p processes. k_b and act_{max} ($k_b < act_{max}$) are constants related to k. $nnz(A)$ is the number of nonzero entries in A. See Section 3 for a detailed analysis.**

Components	**#Flops**	**#Messages**	**#Words**
Filter (deg m)	$O(\frac{nnz(A)mk_b}{p})$	$O(m \log p)$	$O(\frac{2mNk_b}{\sqrt{p}})$
SpMM	$O(\frac{nnz(A)k_b}{p})$	$O(\log p)$	$O(\frac{2Nk_b}{\sqrt{p}})$
Orthonormalization	$O(\frac{3Nact_{max}^2}{p} + 3act_{max}^3 \log p)$	$O(\log p)$	$O(act_{max}^2 \log p)$
Update Rayleigh-quotient	$O(\frac{Nk_b act_{max}}{p})$	$O(\log p)$	$O(act_{max} k_b \log p)$
Evaluate residual	$O(\frac{nnz(A)k_b + Nk_b^2}{p})$	$O(\log p)$	$O(\frac{2Nk_b}{\sqrt{p}})$
Totals	$O(\frac{nnz(A)mk_b + Nact_{max}^2}{p} + 3act_{max}^3 \log p)$	$O(m \log p)$	$O(\frac{2mNk_b}{\sqrt{p}} + act_{max}^2 \log p)$

The rest of this paper is organized as follows. In Section 2, we will briefly introduce the Block Chebyshev-Davidson method. In Section 3, we will describe the distributed Block Chebyshev-Davidson. Numerical results are provided in Section 4 to demonstrate the effectiveness of our algorithm. Section 5 concludes the paper.

2 Block Chebyshev-Davidson Method

Chebyshev-Davidson [8] and Block Chebyshev-Davidson [14] algorithms employ Chebyshev polynomial filters to accelerate the convergence of the Davidson method. The latter applies an inner-outer restart technique inside an iteration that reduces the computational costs of using a large dimension subspace and a progressive filtering technique to take advantage of suitable initial vectors if available. The sequential Block Chebyshev-Davidson algorithm is summarized in Algorithm 1, which is Algorithm 3.1 in [14].

To understand the algorithm, we first introduce some input variables. k_{want} denotes the number of eigenpairs one wants to evaluate. k_b is the number of vectors added to the projection basis per iteration in Step 5 of Algorithm 1. k_{sub} is the dimension of the current subspace V, k_c is the number of converged eigenvectors and k_{act} is the dimension of the active subspace in V deflated by the converged eigenvectors $V(:, 1 : k_c)$. It always holds that $k_c + k_{act} = k_{sub}$. act_{max} and dim_{max} denote the maximum dimensions of the active subspace, and the subspace spanned by V, respectively. We will briefly introduce the essential components of the algorithm and refer the readers to the original paper [14] for more details.

Algorithm 1 Block Chebyshev-Davidson method with inner-outer restart

1: Compute or assign *upperb*, *lowb*, and *low_nwb*.
2: Set $V_{tmp} = V_{init}(:, 1 : k_b)$, (construct $k_b - k_{init}$ random vectors if $k_{init} < k_b$); set $k_i = k_b$(k_i counts the number of used vectors in V_{init}).
3: Set $k_c = 0, k_{sub} = 0$ (k_{sub} counts the dimension of the current subspace); set $k_{act} = 0$ (k_{act} counts the dimension of the active subspace.)
4: **while** *itmax* is not exceeded **do**
5: $V(:, k_{sub} + 1 : k_{sub} + k_b) = ChebyshevFilter(V_{tmp}, m, lowb, upperb, low\_nwb)$.
6: Orthonormalize $V(:, k_{sub} + 1 : k_{sub} + k_b)$ against $V(:, 1 : k_{sub})$.
7: Compute $W(:, k_{act}+1 : k_{act}+k_b) = AV(:, k_{sub}+1 : k_{sub}+k_b)$; set $k_{act} = k_{act}+k_b$; set $k_{sub} = k_{sub}+k_b$.
8: Compute the last k_b columns of the Rayleigh-quotient matrix H: $H(1 : k_{act}, k_{act} - k_b + 1 : k_{act}) = V(:, k_c + 1 : k_{sub})^T W(:, k_{act} - k_b + 1 : k_{act})$, then symmetrize H.
9: Compute eigen-decomposition of $H(1 : k_{act}, 1 : k_{act})$ as $HY = YD$, where $diag(D)$ is in non-increasing order. Set $k_{old} = k_{act}$.
10: If $k_{act} + k_b > act_{max}$, then do inner restart as: $k_{act} = k_{ri}, k_{sub} = k_{act} + k_c$.
11: Do subspace rotation (final step of Rayleigh-Ritz refinement) as: $V(:, k_c+1 : k_c+k_{act}) = V(:, k_c+1 : k_{old})Y(1 : k_{old}, 1 : k_{act}), W(:, 1 : k_{act}) = W(:, 1 : k_{old})Y(1 : k_{old}, 1 : k_{act})$.
12: Compute residual $r = AV(:, k_c+1 : k_c+k_b) - V(:, k_c+1 : k_c+k_b)D(k_c+1 : k_c+k_b, k_c+1 : k_c+k_b)$ and test for convergence of the k_b vectors in $V(:, k_c + 1, k_c + k_b)$, denote the number of newly converged Ritz pairs at this step as e_c. If $e_c > 0$, then update $k_c = k_c + e_c$, save converged Ritz values in $eval$(sort $eval(:)$ in non-increasing order), and deflate/lock converged Ritz vectors (only need to sort $V(:, 1 : k_c)$ according to $eval(1 : k_c)$).
13: If $k_c \geq k_{want}$, then return $eval(1 : k_c)$ and $V(:, 1 : k_c)$, exit.
14: If $e_c > 0$, set $W(:, 1 : k_{act} - e_c) = W(:, e_c + 1 : k_{act}), k_{act} = k_{act} - e_c$.
15: Update H as the diagonal matrix containing non-converged Ritz values $H = D(e_c+1 : e_c+k_{act}, e_c+1 : e_c + k_{act})$.
16: If $k_{sub} + k_b > dim_{max}$, do outer restart as: $k_{sub} = k_c + k_{ro}, k_{act} = k_{ro}$.
17: Get new vectors for the next filtering: Set $V_{tmp} = [V_{init}(k_i+1 : k_i+e_c), V(:, k_c+1 : k_c+k_b-e_c)]; k_i = k_i + e_c$.
18: Set *low_nwb* as the median of non-converged Ritz value in D.
19: **end while**

Chebyshev polynomial filter (Step 5 of Algorithm 1). Given a full eigendecomposition $A = V\Lambda V^T$ of the symmetric matrix in the leading eigenvalue problem (1), for any polynomial $\phi(x) : \mathbb{R} \hookrightarrow \mathbb{R}$, it holds

$$\phi(A) = V\phi(\Lambda)V^T. \tag{2}$$

The Chebyshev polynomial of degree m is defined as

$$C_m(x) = \begin{cases} \cos(m\cos^{-1}(x)), & -1 \leq x \leq 1, \\ \cosh(m\cosh^{-1}(x)), & |x| > 1, \end{cases} \tag{3}$$

Algorithm 2 $[W] = ChebyshevFilter(V, m, a, b, a_0)$

1: **variables**: A the sparse matrix; V the input matrix; m the degree of a Chebyshev polynomial; a the lower bound of unwanted eigenvalues of A; b the upper bound of all eigenvalues; a_0 the lower bound of all eigenvalues;
2: $c = (a + b)/2$; $e = (b - a)/2$;
3: $\sigma = e/(a_0 - c)$;
4: $t = 2/\sigma$;
5: $U = (AV - cV)\sigma/e$;
6: **for** $i = 2 : m$ **do**
7: $\quad \sigma_1 = 1/(\tau - \sigma)$;
8: $\quad W = 2\sigma_1(AU - cU)/e - \sigma\sigma_1 v$;
9: $\quad V = U$;
10: $\quad U = W$;
11: $\quad \sigma = \sigma_1$;
12: **end for**

which rapidly grows outside the interval $[-1, 1]$. By (2), if $\phi(x) = C_m(sx+t)$ with appropriate s and t, then the smallest k eigenvalues of A become the largest k eigenvalues of $\phi(A)$ which are significantly much larger than other eigenvalues, making they well separated from others, which means that it is relatively much easier to identify the leading k eigenpairs of $\phi(A)$ using the Davidson method without correction-equation. Note that the eigenvectors of A and $\phi(A)$ remain unchanged by (2). Therefore, it is sufficient to find the k largest eigenpairs of $\phi(A)$. Another advantage of Chebyshev polynomials is that they admit a three-term recurrence relation such that $\phi(A)$ can be efficiently applied to an arbitrary vector v as long as the fast matvec of A is available. For example, $\phi(A) = C_m(sA + tI_N)$ can be applied to an arbitrary vector v efficiently via the following recursive computation:

$$C_{k+1}(sA + tI_N)v = 2(sA + tI_N)C_k(sA + tI_N)v - C_{k-1}(sA + tI_N)v, \tag{4}$$

for $k = 1, ..., m - 1$, where $C_0(sA + tI_N) = I_N$, $C_1(sA + tI_N) = sA + tI_N$, and I_N is the identify matrix of size N. The computation above only requires a fast algorithm for the matvec of A, which is available since A is sparse. Algorithm 2 is an example of Chebyshev polynomial filter which projects $[a_0, b]$ to $[-1, 1]$ and $[a, a_0)$ to $(-\infty, -1)$ respectively, where a and b are the lower and upper bounds of the spectrum respectively, and a_0 is the lower bound of the unwanted eigenvalues. When the bounds a, a_0, and b are unknown, they could be estimated by standard Lanczos decomposition via some matrix-vector products [14]. Note that a and b are 0 and 2 respectively when A is the normalized Laplacian matrix of a graph, and in this case, a_0 could be roughly estimated, e.g., $a_0 = a + (b - a)k_{want}/N$. Though the first estimation of a_0 might not be satisfactory, it will be approximated by Ritz values computed in Step 18 in Algorithm 1.

Orthonormalization (Step 6 of Algorithm 1). The orthonormalization step is initially performed by the DGKS method [19]; random vectors are used to replace any vectors in $V(:, k_{sub} + 1 : k_{sub} + k_b)$ that may become numerically linearly dependent to the current basis vectors in V. However, a parallel DGKS is inefficient and scales poorly to large concurrencies. We, therefore, replace parallel DGKS with the parallel TSQR [29] for orthonormalization, which attains known communication lower bounds and communicates as little as possible only up to polylogarithmic factors. We leave the description of the parallel TSQR in Section 3.3.

Inner-outer restart (Steps 10 and 16 of Algorithm 1). When the dimension of the active space is larger than act_{max} (Step 10, $k_{act} + k_b > act_{max}$), inner-restart would be applied to reduce the dimension of the active subspace to a smaller value k_{ri}. When the dimension of the current subspace is larger than dim_{max} (Step 16, $k_{sub} + k_b > dim_{max}$), outer-restart would be used to reduce the dimension of the subspace to a smaller value k_{ro}. The default values of the optional parameters act_{max}, k_{ro} and k_{ri} can be readily determined by k_{want}, k_b and the matrix dimension; e.g. $k_{ri} = max([\frac{act_{max}}{2}], act_{max} - 3k_b)$ and $k_{ro} = dim_{max} - 2k_b - k_c$. Though the outer restart is a standard technique in eigensolvers, the inner restart is applied to reduce the cost of the orthonormalization in Step 6 and the Rayleigh-Ritz refinement in Steps 8, 9, and 11. The orthonormalization cost can be high since orthogonalizing k vectors is of $O(Nk^2)$ complexity. A larger

dim_{max} may incur more orthonormalization cost at each iteration. In contrast, if we perform an inner restart, the number of non-converged vectors involved in the orthonormalization per iteration is less than act_{max}. A Rayleigh-Ritz refinement includes computing the last column of the Rayleigh-quotient matrix H (Step 8), computing an eigendecomposition of H (Step 9), and refining the basis V (Step 11). When k_{want} is large and only standard outer restart is applied, the size of H would become too large, making the last two steps expensive. Instead of waiting until the size of H exceeds dim_{max} to perform the standard restart, the algorithm performs an inner restart to restrict the active subspace $V(:, k_c + 1 : k_c + k_{act})$ and hence the size of H. The reduced refinement and reorthogonalization cost per iteration induced by the inner-outer restart may require more iterations to reach convergence; however, the total CPU time for the approach can be much smaller than that of only using standard restart [14].

Progressive filtering technique to utilize initial vectors (Step 17 of Algorithm 1). Instead of using subspace iteration on the entire available initial vectors at once, the algorithm progressively iterates over blocks of the initial vectors [14]. By assuming that the initial vectors are ordered so that their corresponding Rayleigh quotients are in non-decreasing order, the algorithm progressively filters all initial vectors in the natural order. First, it filters the first k_b initial vectors when starting the iteration. Then, for each iteration afterward, it filters e_c number of the leftmost unused initial vectors together with $k_b - e_c$ number of the current best non-converged Ritz vectors, then augment the k_b filtered vectors into the projection basis. One advantage of this approach is that it can augment potentially better new basis vectors during the iteration process instead of only resorting to the initials [14].

3 Distributed Block Chebyshev-Davidson Algorithm

With a basic understanding of the sequential algorithm, we are now ready to describe our distributed algorithm. Algorithm 3 summarizes the framework of our distributed Block Chebyshev-Davidson method and Table 1 summarizes the complexity of each component of the distributed algorithm. To describe the algorithm and complexities in detail, we will progressively explain the following essential steps: distribution of matrices like A and V, communication among processes, distributed SpMM, distributed Chebyshev polynomial filter, local matrix-matrix multiplication, and orthonormalization of tall-skinny matrices.

We assume that p processes partitioning in parallel computation are organized into either a 2D $\sqrt{p}\times\sqrt{p} = p$ grid or a 1D grid. We use $P(i, j)$ where $0 \le i, j < \sqrt{p}$ or $P(\ell)$ where $0 \le \ell < p$ to indicate a process at the corresponding location, respectively. For the former, the set of processes at row i and column j are respectively indicated with $P(i, :)$ and $P(:, j)$. Note that $P(i, j)$ and $P(j\sqrt{p} + i)$ refer to the same process. A matrix M is partitioned in a 2D $\sqrt{p} \times \sqrt{p}$ block structure or a 1D p row block structure. For the former, $M[i, j]$ denotes the submatrix associated with the $P(i, j)$ process, and for the latter, $M[i\sqrt{p}+j]$ denotes the row block associated with $P(i\sqrt{p} + j)$ or $P(j\sqrt{p} + i)$ depending on the context.

Collective communications play an important role in the algorithm. For communication cost analysis, we assume the cost of sending a message of size w from one process to another is given by $\alpha + \beta w$, where α is the latency or message setup time, and β is the reciprocal bandwidth or per-word transfer time. Five collective communication operations are applied to our work: MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Allgather, and MPI_Reduce_scatter. If implemented with a tree algorithm, the MPI_Bcast collective takes $O(\alpha \log p + \beta w \log p)$ cost to broadcast w words to all processes in a communicator. The MPI_Reduce collective reduces w words from all processes at a single process, which has $O(\alpha \log p + \beta w \log p)$ cost with a tree implementation. The MPI_Allreduce collective combines MPI_Reduce and MPI_Bast, which consequently has $O(\alpha \log p + \beta w \log p)$ cost as well. The MPI_Allgather collective gathers w words from all processes at each process, which has $O(\alpha \log p + \beta wp)$ cost if implemented with a recursive doubling algorithm. The MPI_Reduce_scatter collective collects w words from each process and then scatters w/p words to each process, which has $O(\alpha \log p + \beta w)$ cost assuming a recursive halving implementation. For details of these algorithms, refer to a survey by Chan et al. [39].

3.1 Distributed SpMM

Sparse times tall-skinny matrix multiplication (SpMM) is computing

$$U = AV \tag{5}$$

Algorithm 3 Distributed Block Chebyshev-Davidson method

1: Set up communicators of a 2D grid of p processes and set $upperb$, $lowb$, and low_nwb.
2: Set and distribute $V_{tmp} = V_{init}(:, 1 : k_b)$, (construct $k_b - k_{init}$ random vectors if $k_{init} < k_b$); set $k_i = k_b (k_i$ counts the number of used vectors in $V_{init})$.
3: Set $k_c = 0, k_{sub} = 0$ (k_{sub} counts the dimension of the current subspace); set $k_{act} = 0$ (k_{act} counts the dimension of the active subspace.)
4: **while** $itmax$ is not exceeded **do**
5: $V(:, k_{sub} + 1 : k_{sub} + k_b) = DistributedChebyshevFilter(V_{tmp}, m, lowb, upperb, low\_nwb)$.
6: Orthonormalize $V(:, k_{sub} + 1 : k_{sub} + k_b)$ against $V(:, 1 : k_{sub})$ using parallel TSQR.
7: Compute $W(:, k_{act} + 1 : k_{act} + k_b) = AV(:, k_{sub} + 1 : k_{sub} + k_b)$ using distributed SpMM; set $k_{act} = k_{act} + k_b$; set $k_{sub} = k_{sub} + k_b$.
8: Parallelly compute the last k_b columns of the Rayleigh-quotient matrix H: $H(1 : k_{act}, k_{act} - k_b + 1 : k_{act}) = V(:, k_c + 1 : k_{sub})^T W(:, k_{act} - k_b + 1 : k_{act})$, then symmetrize H.
9: Compute eigen-decomposition of $H(1 : k_{act}, 1 : k_{act})$ as $HY = YD$ locally, where $diag(D)$ is in non-increasing order. Set $k_{old} = k_{act}$.
10: If $k_{act} + k_b > act_{max}$, then do inner restart as: $k_{act} = k_{ri}, k_{sub} = k_{act} + k_c$.
11: Do subspace rotation locally (final step of Rayleigh-Ritz refinement) as: $V(:, k_c + 1 : k_c + k_{act}) = V(:, k_c + 1 : k_{old})Y(1 : k_{old}, 1 : k_{act}), W(:, 1 : k_{act}) = W(:, 1 : k_{old})Y(1 : k_{old}, 1 : k_{act})$.
12: Compute residual using distributed SpMM $r = AV(:, k_c+1 : k_c+k_b) - V(:, k_c+1 : k_c+k_b)D(k_c+1 : k_c+k_b, k_c+1 : k_c+k_b)$ and test for convergence of the k_b vectors in $V(:, k_c+1, k_c+k_b)$, denote the number of newly converged Ritz pairs at this step as e_c. If $e_c > 0$, then update $k_c = k_c + e_c$, save converged Ritz values in $eval$(sort $eval(:)$ in non-increasing order), and deflate/lock converged Ritz vectors (only need to sort $V(:, 1 : k_c)$ according to $eval(1 : k_c)$).
13: If $k_c \geq k_{want}$, then return $eval(1 : k_c)$ and $V(:, 1 : k_c)$, exit.
14: If $e_c > 0$, set $W(:, 1 : k_{act} - e_c) = W(:, e_c + 1 : k_{act}), k_{act} = k_{act} - e_c$.
15: Update H as the diagonal matrix containing non-converged Ritz values $H = D(e_c+1 : e_c+k_{act}, e_c+1 : e_c + k_{act})$.
16: If $k_{sub} + k_b > dim_{max}$, do outer restart as: $k_{sub} = k_c + k_{ro}, k_{act} = k_{ro}$.
17: Get new vectors for the next filtering: Set $V_{tmp} = [V_{init}(k_i+1 : k_i+e_c), V(:, k_c+1 : k_c+k_b-e_c)]; k_i = k_i + e_c$.
18: Set low_nwb as the median of non-converged Ritz value in D.
19: **end while**

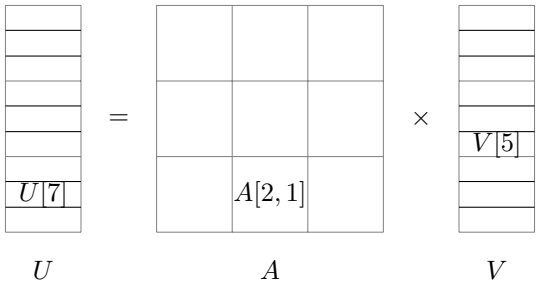

Figure 1: **Illustration of A-Stationary 1.5D SpMM $U = AV$ when the number of processes is $p = 9$. Process $P(2, 1)$ owns the submatrices $U[7], V[5]$, and $A[2, 1]$.**

where $V, U \in \mathbb{R}^{N \times k_b}$ and $k_b << N$. Such computations widely appear in Algorithm 3 including Steps 5, 7, and 12, so a fast distributed SpMM is of essential importance for distributed data.

In the A-Stationary 1.5D algorithm [27], A is partitioned in 2D such that $P(i,j)$ process has $A[i,j]$; V and U are partitioned in 1D (by rows) such that $P(i,j)$ process has $V[j\sqrt{p}+i]$ and $U[i\sqrt{p}+j]$. See Figure 1 for illustration. V is first replicated among $\sqrt{p}$ processes in each column of the process grid, and after this operation, $P(i,j)$ has $\sqrt{p}$ blocks of V, which are given by $V[j\sqrt{p}+\ell]$ for $0 \leq \ell < \sqrt{p}$. Next, the processes perform local SpMM of form $U^j[i\sqrt{p}+\ell] = A[i,j]B[j\sqrt{p}+\ell]$ for $0 \leq \ell < \sqrt{p}$, where U^j denotes the partial dense resulting matrix evaluated by the process at $j-th$ column of the grid. These partial dense matrices are then summed up to get the final result matrix at each process with $U[i\sqrt{p}+j] = \Sigma_{\ell=0}^{\sqrt{p}} U^{\ell}[i\sqrt{p}+j]$. The memory requirement for each process is $nnz(A)/p + 2Nk_b/\sqrt{p}$ where $nnz(A)$ denotes the number of nonzeros in A, and the local computation takes $O((nnz(A)/p)k_b)$ flops.

We use an MPI_Allgather collective to realize the replication among $\sqrt{p}$ processes in each column of the process grid. The replication begins at each process with Nk_b/p words, and thus the respective MPI_Allgather has a cost of $O(\alpha \log p + \beta Nk_b/\sqrt{p})$. We use an MPI_Reduce_scatter collective with a summation operator to realize the reduction of partial dense matrices among $\sqrt{p}$ processes in each row of the process grid. The reduction begins at each process with $Nk_b/\sqrt{p}$ words, and thus the respective MPI_Reduce_scatter has a cost of $O(\alpha \log p + \beta Nk_b/\sqrt{p})$. Therefore, the total communication cost is

$$O(\alpha \log p + \beta \frac{2Nk_b}{\sqrt{p}}). \tag{6}$$

In our distributed implementation Algorithm 3, A is partitioned in 2D, and V, V_{init}, V_{tmp}, W are partitioned in 1D (by rows) in the same way. Each process maintains identical copies of other small dense matrices D, Y, H. The distributed SpMM is applied to Steps 5, 7, and 12. Applying Chebyshev polynomial filters in Step 5 requires additional treatments other than direct A-Stationary 1.5D.

In the distributed implementation in PARSEC, $A, V, V_{init}, V_{tmp}, W$ are all partitioned in 1D (by rows) in the same way, and then a straightforward 1D SpMM is applied. Though local computation in the 1D SpMM takes the same amount of flops, the communication cost

$$O(\alpha \log p + \beta Nk_b) \tag{7}$$

is much more expensive and not scalable because MPI_Allgather is used for all p processes simultaneously.

3.2 Distributed Chebyshev Polynomial Filter

To apply a polynomial of degree m of a distributed sparse A to a distributed dense matrix V, we must evaluate the distributed SpMM AV m times. Imagine applying a simple polynomial filter x^m with $m = 2$:

$$U_2 = AAV, \quad V \in \mathbb{R}^{N \times k_b}, \tag{8}$$

we have to first compute $U_1 = AV$ and then the filtering matrix $U_2 = AU_1$. In the A-Stationary 1.5D algorithm, U_1 and V are partitioned in 1D but in different ways: the $P(i,j)$ process owns $V[j\sqrt{p}+i]$ and $U_1[i\sqrt{p}+j]$. Computing AU_1 directly via the A-Stationary 1.5D algorithm leads to the wrong filtering matrix when A is a general sparse matrix. When A is symmetric, $P(j,i)$ owns $U_1[j\sqrt{p}+i]$ and the submatrices owned by $P(i,j)$ and $P(j,i)$ are the same. As a consequence, transposing the process grid and then applying the A-Stationary 1.5D algorithm to A and U_1 gives the correct filtering matrix U_2 because the operations are equal to computing $A^T U_1$. Note that V and U_2 need not be two separate variables in implementation. However, this only works for the case when m is even. When m is odd, the final filtering matrix U_m after applying a polynomial of degree m to V is still partitioned differently (like U_1). There are two remedies.

- a) When m is odd, we partition an identity matrix I in the same way as we partition A, transpose the process grid, and then apply A-Stationary 1.5D to I and the filtering matrix U_m to get the final matrix which is partitioned in the same way as V;
- b) After each distributed SpMM, we apply a) to the resulting matrix to get a re-distributed matrix partitioned in the same way as V. It is clear that this remedy is more expensive than the first one, but the total complexities are still in the same order.

Note that applying a distributed SpMM to the identity matrix I and a matrix U is essentially equal to moving around parts of the matrix U without any local computations. Our distributed Chebyshev polynomial filter of degree m is implemented in the second way because every intermediate matrix $U_i (i <= m)$ and V should be partitioned in the same way due to the addition operations in Step 5 and 8 in Algorithm 2. Algorithm 4 summarizes the distributed Chebyshev polynomial filter algorithm. The local computation takes $O(nnz(A)mk_b/p)$ because each process averagely own $nnz(A)/p$ nonzeros of A, and the total communication cost is

$$O(m\alpha \log p + \beta \frac{2mNk_b}{\sqrt{p}}). \tag{9}$$

Despite applying the 1D SpMM in PARSEC to compute the filtering matrices is straightforward, which avoids the issues mentioned above, it requires much more communication cost

$$O(m\alpha \log p + \beta mNk_b), \tag{10}$$

which is not scalable.

Algorithm 4 $[W] = DistributedChebyshevFilter(V, m, a, b, a_0)$

1: **variables**: Ω the 2D grid of p processes; A the distributed sparse matrix; I the distributed identity matrix; V the distributed input matrix; m the degree of a Chebyshev polynomial; a the lower bound of unwanted eigenvalues of A; b the upper bound of all eigenvalues; a_0 the lower bound of all eigenvalues;
2: $c = (a + b)/2$; $e = (b - a)/2$;
3: $\sigma = e/(a_0 - c)$;
4: $t = 2/\sigma$;
5: Compute $U = (AV - cV)\sigma/e$ using distributed SpMM;
6: Transpose the 2D grid Ω;
7: Update $U \leftarrow IU$ using distributed SpMM;
8: **for** $i = 2 : m$ **do**
9: $\quad \sigma_1 = 1/(\tau - \sigma)$;
10: $\quad$ Transpose the 2D grid Ω;
11: $\quad$ Compute $W = 2\sigma_1(AU - cU)/e - \sigma\sigma_1 v$ using distributed SpMM;
12: $\quad$ Transpose the 2D grid Ω;
13: $\quad$ Update $W \leftarrow IW$ using distributed SpMM;
14: $\quad V = U$;
15: $\quad U = W$;
16: $\quad \sigma = \sigma_1$;
17: **end for**

3.3 Distributed Orthonormalization

For orthonormalization, we adopt the parallel TSQR in [29]. We begin by illustrating the algorithm for the case of $p = 4$ processes and then state the general version of the algorithm and its performance model. Suppose that the $N \times n$ matrix V is divided into four row blocks $V = [V[0]; V[1]; V[2]; V[3]]$, where $V[i]$ is $N/4 \times n$ associated with $P(i)$ process. First, process i computes the QR factorization of its row block $V[i] = Q_i R_i$, see eq. (11).

$$V = \begin{pmatrix} Q_0R_0 \\ Q_1R_1 \\ Q_2R_2 \\ Q_3R_3 \end{pmatrix}, \begin{pmatrix} \begin{pmatrix} R_0 \\ R_1 \end{pmatrix} \\ \begin{pmatrix} R_2 \\ R_3 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} Q_{01}R_{01} \\ Q_{23}R_{23} \end{pmatrix}, \begin{pmatrix} R_{01} \\ R_{23} \end{pmatrix} = Q_{0123}R_{0123}. \tag{11}$$

Then, processes work in pairs, combining their local R factors by computing the QR factorization of the $2n \times n$ matrix $[R_0; R_1]$, $[R_2; R_3]$, respectively. Thus, $[R_0; R_1]$ is replaced by R_{01} and $[R_2; R_3]$ is replaced by R_{23}. Here and later, the subscripts on a matrix like R_{ij} refer to the original row blocks $V[i]$ and $V[j]$ on which they depend. Finally, the $2n \times n$ QR factorization $[R_{01}; R_{23}] = Q_{0123}R_{0123}$ is computed. It is easy

to verify that R_{0123} is the R factor in the QR factorization of the original matrix V. We could combine all steps above into eq. (12), which expresses the entire computation as a product of intermediate orthonormal factors. Note that the dimensions of the intermediate Q factors are chosen consistently for the product to make sense. The Q factor of the QR decomposition is then the product of the intermediate orthonormal factors. Note that the Q factor is computed locally without communication cost because each process owns all the corresponding intermediate orthonormal factors.

$$V = \begin{pmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{pmatrix} \begin{pmatrix} Q_{01} & \\ & Q_{23} \end{pmatrix} Q_{0123} R_{0123}. \tag{12}$$

The parallel TSQR for general cases is summarized in Algorithm 5 (Algorithm 1 in [29]). In the implementation, the communication between process i and its $q-1$ neighbor in Step 5, 6, and 7 are achieved by MPI_Allgather, which take $O(\alpha \log q + \beta n^2 q)$ time each level. Note that q is a constant. The local QR decomposition at the leaf level and a non-leaf level take $O(2Nn^2/p - 2n^3/3)$ and $O(2qn^3 - 2n^3/3)$ flops, respectively. The local update of Q at the leaf level and a non-leaf level take $O(Nn^2/p)$ and $O(qn^3)$, respectively. When orthonormalizing V in Step 6 of Algorithm 3, the second dimension n of V is bounded by act_{max}. Therefore, in our distributed orthonormalization with TSQR, it takes totally

$$O(3Nact_{max}^2/p + 3act_{max}^3 \log p) \tag{13}$$

flops for local computation and

$$O(\alpha \log p + \beta act_{max}^2 \log p) \tag{14}$$

time for communication.

Unlike the parallel TSQR, to orthonormalize $V(:, k_{sub}+1 : k_{sub}+k_b)$ against $V(:, 1 : k_{sub})$ in Step 6 of Algorithm 3, the parallel DGKS first orthonormalizes $V(:, k_{sub}+1)$ against $V(:, 1 : k_{sub})$, then orthonormalizes $V(:, k_{sub}+2)$ against $V(:, 1 : k_{sub}+1)$, and repeats the procedures until all vectors are orthonormal. Despite simple, the local computation takes $O(Nact_{max}k_b^2/p)$ flops and communication takes

$$O(\alpha k_b \log p + \beta \frac{Nk_b}{p} \log p) \tag{15}$$

time because MPI_Allreduce is applied to normalization. Note that the communication cost is much higher than that of the parallel TSQR because Nk_b/p is much larger than the constant act_{max}^2 when the problem dimension N is large.

Algorithm 5 parallel TSQR

1: **Require**: set Π of p processes; tree with p leaves and height $L = \log_q p$, describing communication pattern; $N \times n$ matrix V distributed in 1D; current process index i.
2: Compute QR factorization $A[i] = Q_{i,0}R_{i,0}$
3: **for** $k = 1 : L$ **do**
4: **if** the current process i has $q-1$ neighbors **then**
5: Send $R_{i,k-1}$ to each neighbor
6: Receive $R_{j,k-1}$ from each neighbor j
7: Stack the $R_{j,k-1}$ from all neighbors (incl. $R_{i,k-1}$), in j order, into the $qn \times n$ matrix $C_{i,k}$, and factor $C_{i,k} = Q_{i,k}R_{i,k}$
8: **else**
9: $R_{i,k} := R_{i,k-1}$, and $Q_{i,k} := I_{n \times n}$
10: **end if**
11: **end for**
12: **Ensure**: $R_{i,L}$ is the R factor of A, for all processes.
13: **Ensure**: Q factor could be evaluated locally and top-down using the tree of intermediate Q factors $Q_{i,k} : i \in \Pi, k \in 0, 1, ..., L$.

3.4 Other Steps

In our algorithm, each process maintains the same Rayleigh-quotient Matrix H which is updated in Step 8. To update $H(1:k_{act}, k_{act}-k_b+1:k_{act})$, we first perform the local computation of the transpose of the corresponding parts of $V[j\sqrt{p}+i]$ and $W[j\sqrt{p}+i]$ on process $P(i,j)$, then reduce all results among $\sqrt{p}$ processes in each row by MPI_Allreduce with summation operator. Next, we perform a similar reduction for all resulting among $\sqrt{p}$ processes in each column to get the updated H. This step takes at most $O(Nk_b act_{max}/p)$ flops for local computation and

$$O(\alpha \log p + \beta act_{max} k_b \log p) \tag{16}$$

for communication, where act_{max} and k_b are small constants.

Each process also maintains the same copy of D and Y at each process because they are of dimension at most act_{max}. Computing residual in Step 12 involves a distributed SpMM, a local matrix-matrix multiplication, and communication, which in total takes $O(nnz(A)k_b/p + Nk_b^2/p)$ flops for local computation and

$$O(\alpha \log p + \beta \frac{2Nk_b}{\sqrt{p}}). \tag{17}$$

time for communication.

Steps 9, 11, and 15 are performed locally without any communication cost and together take at most $O(act_{max}^3 + Nact_{max}^2/p)$ flops. All other steps take only $O(1)$ time.

4 Numerical Results

We present the numerical results of four experiments conducted on the Zaratan cluster operated by the University of Maryland. It features 360 compute nodes, each with dual 2.45 GHz AMD 7763 64-core CPUs. The cluster has HDR-100 (100 Gbit) Infiniband interconnects between the nodes, with storage and service nodes connected with full HDR (200 Gbit). The theoretical peak floating-point rate is 3.5 Pflops. The code[1] is written in Julia 1.7.3 using MPI.jl [40] with OpenMPI [41]. The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

In the first test, we apply our algorithm as the eigensolver in spectral clustering to partition graphs with known truth from IEEE HPEC 2022 Graph Challenge [2]. We compare the sequential algorithm with an eigensolver LOBPCG on matrices from the graph challenge in the second test. In the third test, we test the scaling of the distributed algorithm and its components to the number of processes on various matrices. In the fourth test, we show the superiority of our distributed implementation to the distributed implementation in PARSEC. Standard variable settings used in the numerical results are as follows:

- p: the number of processes or cores;
- k: the number of eigenvectors to compute;
- N: the dimension of a matrix or the number of nodes in a graph;
- m: the degree of a Chebyshev polynomial filter;
- k_b: the number of vectors added to the projection basis per iteration;
- act_{max}: the maximum dimension of the active subspace $act_{max} = max(5k_b, 30)$ throughout all experiments;
- dim_{max}: the maximum dimension of the subspace $dim_{max} = max(act_{max} + 2k_b, k + 30)$ throughout all experiments;
- tol: the stopping criterion or tolerance.

[1] https://github.com/qiyuanpang/DistributedLEVP.jl
[2] http://graphchallenge.mit.edu

Table 2: **Evaluation metrics for clustering results on the smallest 32 eigenvectors of the static graphs from GraphChallenge. The eigenvectors are computed by the Block Chebyshev Davidson method with degree $m = 15$, $k_b = 4$, and tolerance $tol = 10^{-3}$.**

Graph size N	Graph type	RI	ARI	NMI	VI
5K	LBOLBSV	97.51%	79.74%	94.31%	3.22e-01
5K	LBOHBSV	83.39%	39.14%	79.26%	9.91e-01
5K	HBOLBSV	97.46%	80.59%	94.17%	3.25e-01
5K	HBOHBSV	82.73%	44.65%	74.03%	1.17e+00
20K	LBOLBSV	99.74%	96.12%	99.14%	5.87e-02
20K	LBOHBSV	96.16%	73.29%	94.64%	3.24e-01
20K	HBOLBSV	99.54%	93.44%	98.23%	1.20e-01
20K	HBOHBSV	99.47%	94.84%	97.76%	1.42e-01
50K	LBOLBSV	99.25%	85.70%	95.45%	3.37e-01
50K	LBOHBSV	99.41%	90.45%	95.15%	3.47e-01
50K	HBOLBSV	98.36%	70.16%	91.67%	6.15e-01
50K	HBOHBSV	98.26%	78.39%	91.95%	5.57e-01
200K	LBOLBSV	97.15%	37.96%	77.26%	1.84e+00
200K	LBOHBSV	98.41%	66.09%	84.62%	1.23e+00
200K	HBOLBSV	98.03%	44.85%	69.22%	2.54e+00
200K	HBOHBSV	98.20%	54.99%	78.01%	1.79e+00
1M	LBOLBSV	98.29%	28.05%	58.36%	3.87e+00
1M	LBOHBSV	98.10%	33.82%	62.59%	3.39e+00
1M	HBOLBSV	98.18%	22.06%	50.09%	4.63e+00
1M	HBOHBSV	98.50%	42.05%	58.90%	3.75e+00
5M	LBOLBSV	98.46%	98.53%	39.69%	6.16e+00
5M	LBOHBSV	98.50%	98.58%	45.56%	5.49e+00
5M	HBOLBSV	98.87%	98.92%	34.08%	6.87e+00
5M	HBOHBSV	98.90%	98.95%	43.41%	5.82e+00

4.1 Spectral Clustering

The exact bounds of the Laplacian or normalized Laplacian A are known, e.g., the smallest and largest eigenvalues of the normalized Laplacian are 0 and 2, respectively. The Block Chebyshev-Davidson method, therefore, converges fast with the optimal bounds. Here we consider the normalized Laplacian A of static graphs from the graph challenge. There are fours categories of graphs: low block overlap and low block size variation (LBOLBSV), low block overlap and high block size variation (LBOHBSV), high block overlap and low block size variation (HBOLBSV), and high block overlap and high block size variation (HBOHBSV). Table 2 summarizes the evaluation metrics, including Rand Index (RI), Adjusted Rand Index (ARI), Normalized Mutual Information (NMI), and Variation of Information (VI), of the clustering results of the graph above. For every type of graph, k-means clustering based on the computed eigenvectors reaches a high rand index which is one of the major metrics measuring clustering accuracy. Note that k-means is not guaranteed to find the optimal clusters [42] and the task is indeed NP-hard in general Euclidean space even for two clusters [43, 44].

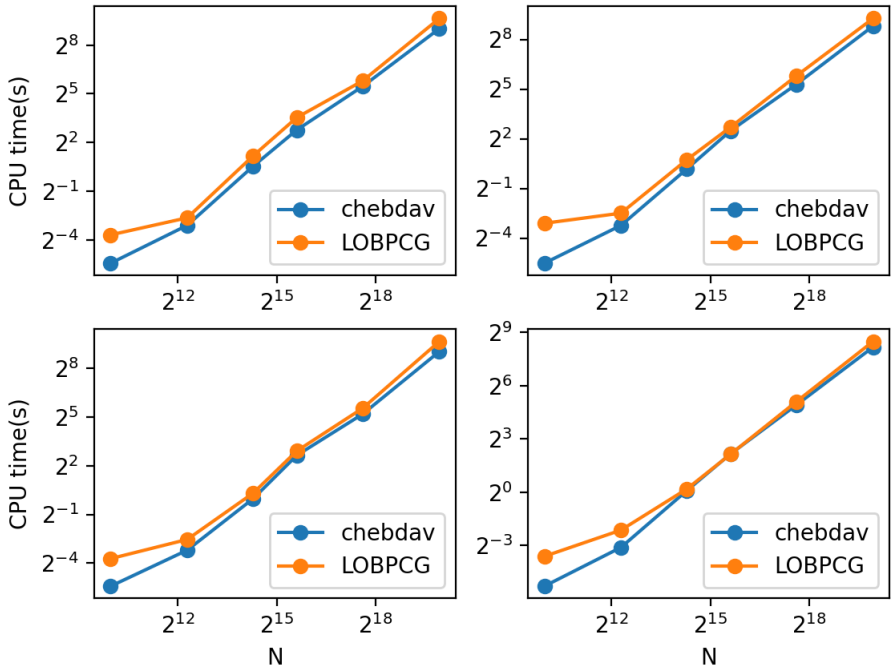

Figure 2: **CPU time comparison between chebdav and LOBPCG for converging** 16 **eigenpairs of matrices in Tables 2: LBOLBSV (upper left), LBOHBSV (upper right), HBOLBSV (lower left), and HBOHBSV (lower right). The stopping criteria (tolerance) is** 10^{-6}**. For chebdav, the degree** $m = 11$ **and** $k_b = 4$**. The preconditioner for LOBPCG is the identity matrix because no suitable preconditioners are known.**

4.2 Comparison of Sequential Eigensolvers

We compare the sequential Block Chevyshev-Davidson method with a block preconditioned eigensolver LOBPCG [32] on the Laplacian matrices in Tables 2. The preconditioner of LOBPCG is the identity matrix because we do not know any other effective preconditioners, and the design for preconditioners of LOBPCG is an entirely different topic. Figure 2 presents that the Block Chebyshev-Davidson method is comparable to LOBPCG while computing eigenpairs of the normalized Laplacian matrices of the graphs. Furthermore,

numerical results in [14] show that the Block Chebyshev-Davidson method is an order of magnitude faster than LOBPCG when computing the smallest eigenpairs for specific matrices from quantum chemistry.

Table 3: **Properties of matrices used in our evaluations. The values under the "load imb." column present the load imbalance in terms of the sparse matrix elements for 121 processes (i.e., 11 $\times$ 11 2D partition).**

Sparse Matrix	N	avg degree	nnz(A)	load imb.
LBOLBSV	5M	48.5	242M	1.21
HBOLBSV	20M	48.5	970M	1.21
MAWI-Graph-1	18M	3.0	56M	8.8
Graph500-scale24-ef16	16M	31.6	529M	7.15

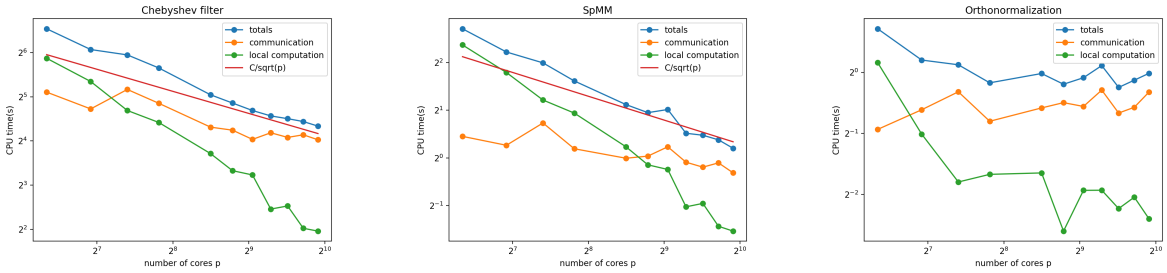

Figure 3: **Scaling of local computation and communication in a distributed Chebyshev filter, SpMM, and TSQR, on the HBOLBSV matrix in Table 3. The degree of the filter is $m = 11$ and the number of vectors is $k = 8$.**

4.3 Scaling of the Distributed Algorithm

We consider four sparse normalized Laplacian matrices input, including traffic data from the MAWI Project (MAWI-Graph-1 [45]), synthetic data at various scales generated using the scalable Graph500 Kronecker generator (Graph500-scale24-ef16 [46]), and two matrices from the Graph Challenge (LBOLBSV and HBOLBSV). Various properties of these matrices are presented in Table 3. Load imbalance is defined as the ratio of the maximum number of nonzeros assigned to a process to the average number of nonzeros in each process:

$$\frac{p * \max_{i,j} nnz(A[i,j])}{nnz(A)}. \tag{18}$$

We test the scaling of our distributed algorithm and its components on the four matrices. Figure 3 illustrates how local computation and communication in a Chebyshev polynomial filter, an SpMM, and an orthonormalization using TSQR scale to large concurrencies. The speedup of the filter and SpMM is roughly proportional to $\sqrt{p}$ because communication is more costly in the two components and accelerated at a rate roughly proportional to $\sqrt{p}$. The orthonormalization using distributed TSQR does not scale well due to communication costs. Figure 4 presents the scaling of the distributed algorithm and its components and shows the speedup of the Chebyshev polynomial filter and hence the entire algorithm is roughly proportional to $\sqrt{p}$. Though the distributed orthonormalization and other components, including updating the Rayleigh-quotient matrix and evaluating residuals, do not scale well to the numbers of cores, Figure 5 shows that they are minor, and the filters are dominant in terms of CPU time. Note that a higher degree of a Chebyshev polynomial filter tends to result in faster convergence and more dominance among other components. Therefore, the distributed algorithm is efficient and practically scalable to a large number of cores.

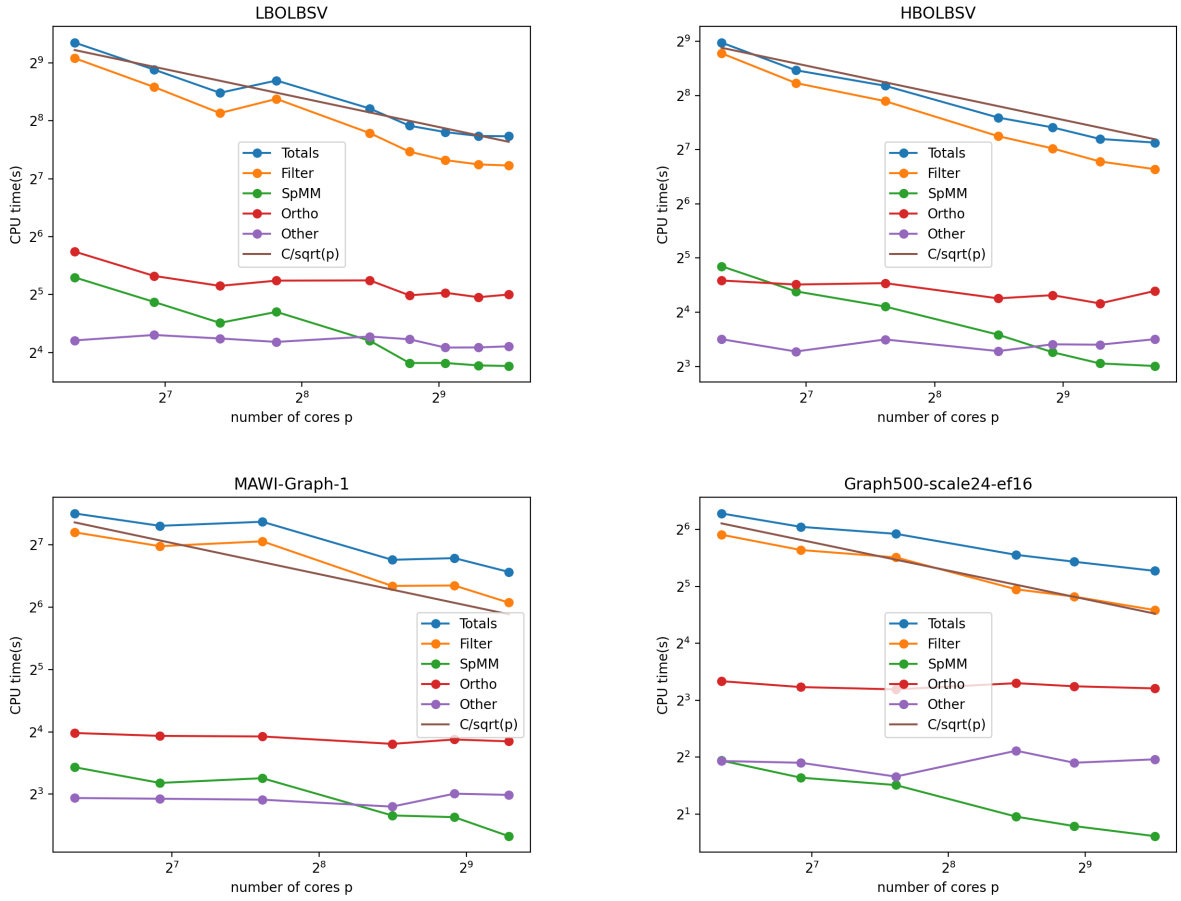

Figure 4: **Scaling of the distributed Block Chebyshev-Davison algorithm and its components to numbers of cores. The stopping criteria are** $tol = 10^{-3}$ **and the degrees are** $m = 15$**. The number of eigenvectors** k **and the number of vectors added per iteration** k_b **vary for matrices. LBOLBSV:** $k = 16, k_b = 16$**; HBOHBSV:** $k = 4, k_b = 4$**; MAWI-Graph-1:** $k = 4, k_b = 4$**; Graph500-scale24-ef16:** $k = 4, k_b = 4$**.**

4.4 Comparison of Distributed Implementations

We compare our distributed implementation of the Block Chebyshev-Davidson method with the distributed implementation in PARSEC. Comparing the corresponding implementations of SpMM, Chebyshev polynomial filters, and orthonormalization is sufficient. Figure 6 shows our implementations consistently outperform the implementations in PARSEC in efficiency and scalability. Indeed, the implementations in PARSEC do not scale to relatively large concurrencies.

5 Conclusion

We propose a distributed Block Chebyshev-Davidson method for spectral analysis in spectral clustering. One of the advantages of applying the Block Chebyshev-Davidson method in such a spectral analysis is that the theoretically optimal lower and upper bounds are known, so the method needs not to estimate the bounds and converges fast. Numerical results show that the sequential method is at least comparable to LOBPCG. The distributed Block Chebyshev-Davidson algorithm is developed based on an A-Stationary 1.5D SpMM algorithm and a parallel TSQR. Though the distributed orthonormalization using TSQR does not scale well,

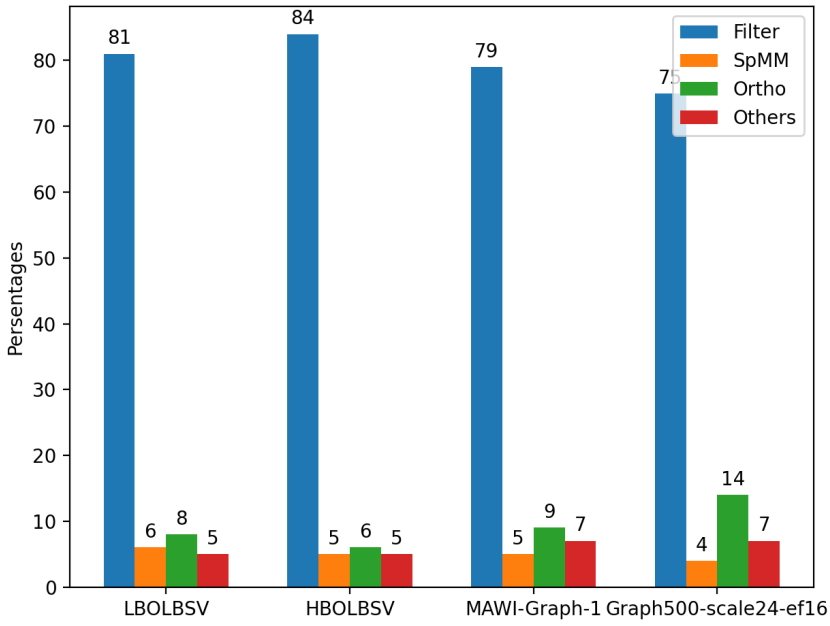

Figure 5: **Percentages of the CPU time spent on components in the experiments shown in Figure 4 when the number of cores $p = 121$.**

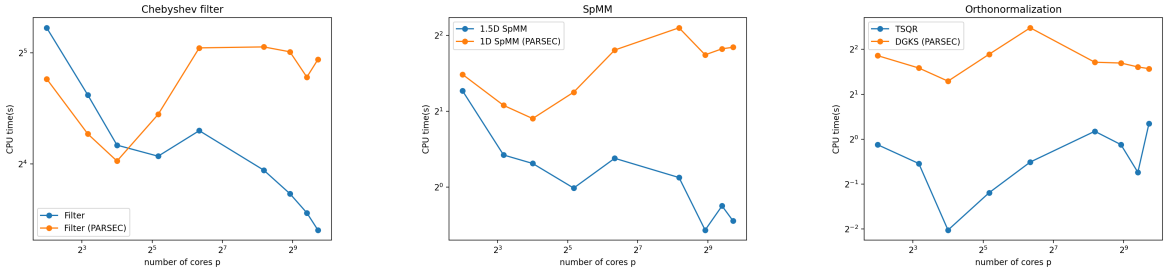

Figure 6: **Comparison of our implementation and PARSEC's implementation of Chebyshev polynomial filters, SpMM, and orthonormalization. The experiment is conducted on the LBOLBSV matrix in Table 3 with problem size $N = 5 \times 10^6$, the number of vectors $k = 16$, and the degree of the filter $m = 11$.**

the distributed Chebyshev polynomial filters and SpMMs scale well and dominate the entire algorithm in terms of running time. The SpMM, filters, and hence the entire algorithm are practically scalable to a large number of processes and accelerated approximately at a rate $\sqrt{p}$. Our new distributed algorithm consistently outperforms its counterpart in PARSEC in both efficiency and scalability.

6 Acknowledgments

We thank Aleksey Urmanov for helpful discussion and comments, and thank Oracle Labs, Oracle Corporation, Austin, TX, for providing funding that supported research in the area of scalable spectral clustering and distributed eigensolvers.

References

[1] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[2] Jeff Cheeger. A lower bound for the smallest eigenvalue of the laplacian. In *Problems in analysis*, pages 195–200. Princeton University Press, 2015.

[3] William E Donath and Alan J Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.

[4] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak mathematical journal*, 23(2):298–305, 1973.

[5] Stephen Guattery and Gary L Miller. On the performance of spectral graph partitioning methods. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994.

[6] Daniel A Spielman and Shang-Hua Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *Proceedings of 37th conference on foundations of computer science*, pages 96–105. IEEE, 1996.

[7] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.

[8] Yunkai Zhou and Yousef Saad. A chebyshev–davidson algorithm for large symmetric eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, 29(3):954–971, 2007.

[9] Attila Szabo and Neil S Ostlund. *Modern quantum chemistry: introduction to advanced electronic structure theory*. Courier Corporation, 2012.

[10] Jianfeng Lu and Haizhao Yang. Preconditioning orbital minimization method for planewave discretization. *Multiscale Modeling & Simulation*, 15(1):254–273, 2017.

[11] Yingzhou Li and Haizhao Yang. Interior eigensolver for sparse hermitian definite matrices based on zolotarev's functions. *Communications in Mathematical Sciences*, 19(4):1113 – 1135, 2021.

[12] Michel Crouzeix, Bernard Philippe, and Miloud Sadkane. The davidson method. *SIAM Journal on Scientific Computing*, 15(1):62–76, 1994.

[13] Gerard LG Sleijpen and Henk A Van der Vorst. A jacobi–davidson iteration method for linear eigenvalue problems. *SIAM review*, 42(2):267–293, 2000.

[14] Yunkai Zhou. A block chebyshev–davidson method with inner–outer restart for large eigenvalue problems. *Journal of Computational Physics*, 229(24):9188–9200, 2010.

[15] Yunkai Zhou, Yousef Saad, Murilo L Tiago, and James R Chelikowsky. Parallel self-consistent-field calculations via chebyshev-filtered subspace acceleration. *Physical Review E*, 74(6):066704, 2006.

[16] Walter Kohn and Lu Jeu Sham. Self-consistent equations including exchange and correlation effects. *Physical review*, 140(4A):A1133, 1965.

[17] Victor Wen-zhe Yu, Fabiano Corsetti, Alberto García, William P Huhn, Mathias Jacquelin, Weile Jia, Björn Lange, Lin Lin, Jianfeng Lu, Wenhui Mi, et al. Elsi: A unified software interface for kohn–sham electronic structure solvers. *Computer Physics Communications*, 222:267–285, 2018.

[18] Jianfeng Lu and Haizhao Yang. A cubic scaling algorithm for excited states calculations in particle–particle random phase approximation. *Journal of Computational Physics*, 340:297–308, 2017.

[19] James W Daniel, Walter Bill Gragg, Linda Kaufman, and Gilbert W Stewart. Reorthogonalization and stable algorithms for updating the gram-schmidt qr factorization. *Mathematics of Computation*, 30(136):772–795, 1976.

[20] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. Montana State University, 1969.

[21] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[22] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[23] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011.

[24] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[25] Aydin Buluç and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

[26] Martin D Schatz, Robert A Van de Geijn, and Jack Poulson. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing*, 38(6):C748–C781, 2016.

[27] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*, pages 431–442, 2021.

[28] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. Mpi-faun: An mpi-based framework for alternating-updating nonnegative matrix factorization. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):544–558, 2017.

[29] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[30] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[31] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[32] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

[33] Richard B Lehoucq, Danny C Sorensen, and Chao Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM, 1998.

[34] Frank Lin and William W Cohen. Power iteration clustering. In *ICML*, 2010.

[35] Maxim Naumov and Timothy Moon. Parallel spectral graph partitioning. *NVIDIA, Santa Clara, CA, USA, Tech. Rep., NVR-2016-001*, 2016.

[36] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y Chang. Parallel spectral clustering in distributed systems. *IEEE transactions on pattern analysis and machine intelligence*, 33(3):568–586, 2010.

[37] Weizhong Yan, Umang Brahmakshatriya, Ya Xue, Mark Gilder, and Bowden Wise. p-pic: parallel power iteration clustering for big data. *Journal of Parallel and Distributed computing*, 73(3):352–359, 2013.

[38] Zenan Huo, Gang Mei, Giampaolo Casolla, and Fabio Giampaolo. Designing an efficient parallel spectral clustering algorithm on multi-core processors in julia. *Journal of Parallel and Distributed Computing*, 138:211–221, 2020.

[39] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[40] Simon Byrne, Lucas C Wilcox, and Valentin Churavy. Mpi. jl: Julia bindings for the message passing interface. In *Proceedings of the JuliaCon Conferences*, volume 1, page 68, 2021.

[41] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[42] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[43] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75(2):245–248, 2009.

[44] Sanjoy Dasgupta and Yoav Freund. Random projection trees for vector quantization. *IEEE Transactions on Information Theory*, 55(7):3229–3242, 2009.

[45] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the {WIDE} project. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*, 2000.

[46] Jeremy Kepner, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Tim Davis, Vijay Gadepally, Michael Houle, Matthew Hubbell, Hayden Jananthan, et al. Design, generation, and validation of extreme scale power-law graphs. *arXiv preprint arXiv:1803.01281*, 2018.