Fast Spectral Graph Layout on Multicore Platforms

Ashirbad Mishra
Pennsylvania State University
University Park, Pennsylvania
amishra@psu.edu

Shad Kirmani
eBay Inc.
San Jose, California
skirmani@ebay.com

Kamesh Madduri
Pennsylvania State University
University Park, Pennsylvania
madduri@psu.edu

ABSTRACT

We present ParHDE, a shared-memory parallelization of the High-Dimensional Embedding (HDE) graph algorithm. Originally proposed as a graph drawing algorithm, HDE characterizes the global structure of a graph and is closely related to spectral graph computations such as computing the eigenvectors of the graph Laplacian. We identify compute- and memory-intensive steps in HDE and parallelize these steps for efficient execution on shared-memory multicore platforms. ParHDE can process graphs with billions of edges in minutes, is up to 18× faster than a prior parallel implementation of HDE, and achieves up to a 24× relative speedup on a 28-core system. We also implement several extensions of ParHDE and demonstrate its utility in diverse graph computation-related applications.

CCS CONCEPTS

• **Human-centered computing** → *Graph drawings*; • **Computing methodologies** → **Spectral methods**; • **Theory of computation** → **Shared memory algorithms**.

KEYWORDS

graph layout, graph embedding, breadth-first search, sparse matrix vector multiplication, orthogonalization

ACM Reference Format:
Ashirbad Mishra, Shad Kirmani, and Kamesh Madduri. 2020. Fast Spectral Graph Layout on Multicore Platforms. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404471

1 INTRODUCTION

The area of spectral graph theory devises elegant solutions to combinatorial graph problems by relating them to eigenvalues and eigenvectors of matrices associated with the underlying graph. Efficient spectral methods are known for several graph problems, including connectivity, partitioning, and vertex ordering formulations. In this work, we study parallelization of a fast spectral method for *graph layout*. The objective of graph layout is to assign coordinates to graph vertices such that a pre-specified cost function is optimized. The coordinates can also be used to generate a graph *drawing*.

The algorithm we study is called High-Dimensional Embedding (HDE) [31] and is considered one of the fastest algorithms for graph layout. HDE is closely related to classical spectral graph drawing algorithms. However, unlike the classical algorithms, HDE does not solve an eigenproblem on the entire graph. Instead, it uses the solution to a fixed-size eigenproblem in a principled manner to generate an approximate solution to the original problem. A sample HDE layout is shown in Figure 1, where the top figure, generated using ParHDE—our implementation of HDE—captures the global structure seen in the bottom figure. The main advantage of HDE is that it can be orders-of-magnitude faster than the spectral algorithm used to generate the bottom figure.

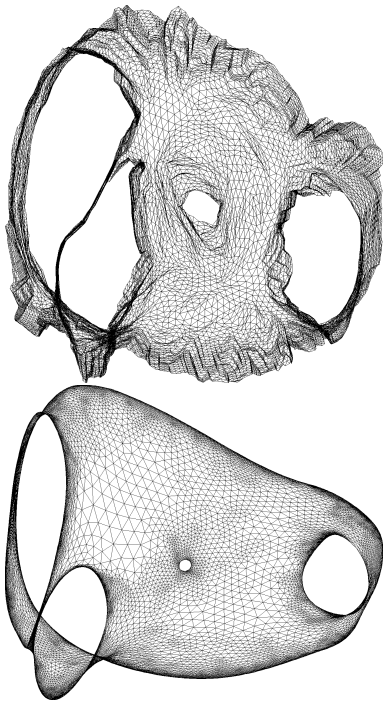

Figure 1: Drawings of the barth5 graph [10] (15606 vertices and 45878 edges) using ParHDE (top) and using dominant eigenvectors of the normalized adjacency matrix (bottom).

The primary contribution of this work is the new ParHDE implementation targeting shared-memory platforms. We identify three main compute- and memory-intensive phases in HDE, and show that ParHDE is significantly faster than a prior implementation of HDE [27, 33]. We also extend ParHDE and demonstrate its utility for several related graph computations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '20, August 17–20, 2020, Edmonton, AB, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404471

2 BACKGROUND AND PRIOR WORK

2.1 Preliminaries

We denote a graph as $G(V, E, W)$, where $V = \{1, \ldots, n\}$ is the set of vertices and E is the set of edges. $m = |E|$. We assume the graph is undirected and has no self loops or parallel edges. An edge $\langle i, j \rangle$ can have a non-negative real edge weight $W(i, j)$ representing the similarity of vertices i and j. Note the interpretation of an edge weight: heavier edges indicate greater levels of similarity. If two vertices are not connected, then the similarity value is assumed to be 0, i.e., the pair of vertices are dissimilar. For unweighted graphs, the edge weights are all assumed to be the same and equal to one. The weighted degree of a vertex is the sum of the weights of its attached edges.

Let A denote the symmetric adjacency matrix corresponding to the graph, with $A(i, j) = W(i, j)$ if $\langle i, j \rangle \in E$, and 0 otherwise. Let D denote the degrees matrix, a diagonal matrix with $D(i, i)$ set to the weighted degree of vertex i. The degree normalized matrix $D^{-1}A$ is called the transition matrix or the walk matrix. The Laplacian L is a symmetric matrix given by $D - A$. The properties of the Laplacian are well studied. We will state some of its properties relevant to graph drawing in this section, but please refer to Koren's paper [31] and references therein for more details. Spielman's book chapter on spectral graph theory [38] is another good reference.

The p-dimensional layout of a graph is defined by p vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p \in \mathbb{R}^n$, where $\boldsymbol{x}_k(i)$ corresponds to the coordinate of vertex i in the k^{th} dimension. In practice, p is chosen to be 2 or 3 for screen layouts, and we assume $p = 2$ in this paper. To produce a node-link graph drawing, the two endpoints of an edge can be connected by straight line. The Euclidean distance d_{ij} between vertices i and j in the p-dimensional layout is given by $\sqrt{\sum_{k=1}^{p} (\boldsymbol{x}_k(i) - \boldsymbol{x}_k(j))^2}$.

Given any vector $\boldsymbol{y} \in \mathbb{R}^n$, it can be shown that $\boldsymbol{y}^T L \boldsymbol{y}$ equals $\sum_{\langle i,j \rangle \in E} W(i, j)\,(\boldsymbol{y}(i) - \boldsymbol{y}(j))^2$. This fact is directly related to an *optimal* layout of a graph. Additionally, the Laplacian satisfies many interesting properties. Since L is symmetric and positive semidefinite, all its n eigenvalues are real and nonnegative. $\mathbf{1}_n$ is an eigenvector of L with a corresponding eigenvalue of 0. If the graph is connected (as we assume in this paper), the multiplicity of the eigenvalue 0 is exactly 1. The eigenvalues of the Laplacian are by convention ordered from low to high, $0 = \lambda_1 < \lambda_2 \leq \cdots \leq \lambda_n$.

Given D and L, let the eigenpair $(\boldsymbol{u}, \mu)$ be a solution to the *generalized eigenproblem* $L\boldsymbol{u} = \mu D\boldsymbol{u}$. D-normalization is a way to uniquely specify the $\boldsymbol{u}_k$ eigenvectors: $\boldsymbol{u}_k^T D \boldsymbol{u}_k = 1,\ k = 1, \ldots, n$. The vectors $\boldsymbol{u}_k$ are called *degree-normalized* eigenvectors. It can be easily shown that the degree-normalized eigenvectors are also the (non-generalized) eigenvectors of the transition matrix $D^{-1}A$, using the fact that $L = D - A$. The eigenvalues of this matrix are in reverse order. Koren shows that the *degree-normalized eigenvectors* are the *optimal solution* to a constrained minimization problem that is closely related to graph drawing aesthetics:

$$\begin{aligned} \underset{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p}{\text{minimize}} \quad & \frac{\sum_{k=1}^{p} \boldsymbol{x}_k^T L \boldsymbol{x}_k}{\sum_{k=1}^{p} \boldsymbol{x}_k^T D \boldsymbol{x}_k} \\ \text{subject to} \quad & \boldsymbol{x}_k^T D \boldsymbol{x}_l = \delta_{kl},\ k, l = 1, \ldots, p \\ & \boldsymbol{x}_k^T D \mathbf{1}_n = 0. \end{aligned} \tag{1}$$

In the above equation, δ_{kl} is the Kronecker delta function, which is 1 when $k = l$ and 0 otherwise. Minimizing the numerator in the above expression has the effect of placing similar vertices close to each other, whereas maximizing the denominator scatters vertices. The constraints impose D-orthogonality to the unit vector and to each other. Eigen-projection or Hall's algorithm [22] refers to the use of p non-degenerate eigenvectors of the Laplacian for layout. In contrast, Koren recommends using p degree-normalized eigenvectors [31]. Degree-normalized vectors are also used for image segmentation [36] tasks, but with a different motivation. Using degree-normalized eigenvectors instead of the Laplacian eigenvectors has the effect of treating a heavy edge (say, weight 50) connecting two high-degree vertices (say, degree 500) in the same manner as a light edge (say, weight 1) connecting two low-degree vertices (say, degree 10). Both are drawn with equal length. However, when using the Laplacian's eigenvectors, the heavier edge is assigned a 50× shorter length and this pushes high-degree vertices towards each other, which is undesirable. The objective function reduces to $\sum_{k=1}^{p} \mu_k$, where μ_k is the k^{th} eigenvalue corresponding to the degree-normalized eigenvector $\boldsymbol{u}_k$.

2.2 High-dimensional embedding (HDE)

The HDE algorithm is motivated by the high computational cost of eigenvector calculations. For layout, iterative algorithms are typically used to determine a few eigenvectors, and the convergence rate of these approaches is heavily dependent on the graph structure and the distribution of eigenvalues. The Laplacian eigenvectors or the degree-normalized eigenvectors are chosen to be the axes in eigen-projection schemes. Note that these could be arbitrary vectors in $\mathbb{R}^n$.

The key idea behind HDE is to *constrain* the axes to lie in a subspace $\mathbb{S} \subseteq \mathbb{R}^n$. The subspace $\mathbb{S}$ can be defined by a matrix $B \in \mathbb{R}^{n \times s}$ whose columns span $\mathbb{S}$. Koren recommends using graph-theoretic distances to construct such a matrix B. Consider s *pivot vertices* or *viewpoints* in the graph. s is chosen to be a small constant (e.g., 50). The shortest path lengths from one of these vertices, say v, to all other vertices (i.e., the single-source shortest paths or SSSP problem) can be used as an axis to map the rest of the vertices. Given s such vertices, we may be able to reduce vertex occlusion. We would also want to pick vertices such that the shortest path vectors are not correlated. One way to do this is to consider the set of vertices that are a 2-approximation to the k-centers problem [19]. In the k-centers problem, we are asked to choose k vertices in a graph such that the longest shortest path length from any vertex to the k centers is minimized. The *farthest-first* 2-approximation algorithm begins by randomly picking a vertex and then, in each

Algorithm 1 Eigen-projection in a subspace: high-dimensional embedding (HDE).

Input: G, L (Laplacian), s (subspace dimension)
Output: $\boldsymbol{x}$, $\boldsymbol{y}$

Initialize $S \in \mathbb{R}^{n \times (s+1)}$
Initialize $B \in \mathbb{R}^{n \times s}$
$\boldsymbol{s}_0 \leftarrow \mathbf{1}$ ▷ Columns of S are indexed from 0
Normalize $\boldsymbol{s}_0$
$v \leftarrow$ randomly-chosen start vertex
Initialize $\boldsymbol{d} \in \mathbb{R}^n$
for $i \leftarrow 1, s$ **do**
  $\boldsymbol{b}_i \leftarrow \text{SSSP}(v)$ ▷ $\boldsymbol{b}_i$ is i^{th} column of B
  $\boldsymbol{s}_i \leftarrow \boldsymbol{b}_i$
  Normalize $\boldsymbol{s}_i$
  for $l \leftarrow 0, i-1$ **do**
    $\boldsymbol{s}_i \leftarrow \boldsymbol{s}_i - (\boldsymbol{s}_l^T \boldsymbol{s}_i)\boldsymbol{s}_l$ ▷ Orthogonalize
  for $j \leftarrow 1, n$ **do**
    $\boldsymbol{d}(j) \leftarrow \min(\boldsymbol{d}(j), \boldsymbol{b}_i(j))$
  $sv \leftarrow$ farthest vertex from previous sources
Drop 0^{th} column of S (degenerate vector)
$Y_{s\times 2} \leftarrow$ Top two eigenvectors of $S^T L S$
$[\boldsymbol{x}, \boldsymbol{y}] \leftarrow BY$

iteration, adding to the set of centers the vertex that is farthest from the currently chosen centers. Ties are arbitrarily broken. Once the shortest path vectors are determined, they are made linearly independent using a Gram-Schmidt-like procedure and vectors that are linearly dependent are discarded. Further, the resulting vectors are ensured to be orthogonal to a unit vector. Denote this matrix as S.

Koren then shows that by modifying Equation 1, the eigenvectors corresponding to the $S^T LS$ matrix can be considered as approximations to the eigenvectors of L. Since $S^T LS$ is a very small matrix (if we choose $s = 50$), the running time for computing its eigenvectors will be negligible. The steps of HDE are given in Algorithm 1, and HDE is best thought of as Eigen-projection in a subspace.

2.3 Related Work

The HDE algorithm given in Algorithm 1 is a refined version and successor to another algorithm with the same name (HDE), designed by Harel and Koren [23, 24]. To avoid this confusion, the older HDE algorithm will be referred to as PHDE. Its pseudocode is given in Algorithm 2. The interpretation of weights is the opposite of HDE, ie., lower weights indicate closer vertices. Like HDE, PHDE also uses graph-theoretic distances from s pivots to compute a shortest paths matrix. However, in the next step, the principal components analysis (PCA) dimensionality reduction strategy is applied to a column-centered version of the path length matrix. Column centering implies subtracting the mean of each column from the entries of the column, and this has the effect of making the column means zero. Next, the two dominant eigenvectors of $C^T C$ are used as the drawing axes. Harel and Koren show that this algorithm has the effect of maximizing the scatter of the nodes, i.e., the denominator of Equation 1 (without the D-normalization).

Algorithm 2 PCA-based high-dimensional embedding (PHDE).

Input: G, s (subspace dimension)
Output: $\boldsymbol{x}$, $\boldsymbol{y}$

Initialize $B \in \mathbb{R}^{n \times s}$
$v \leftarrow$ randomly-chosen start vertex
Initialize $\boldsymbol{d} \in \mathbb{R}^n$
for $i \leftarrow 1, s$ **do**
  $\boldsymbol{b}_i \leftarrow \text{SSSP}(v)$ ▷ $\boldsymbol{b}_i$ is i^{th} column of B
  for $j \leftarrow 1, n$ **do**
    $\boldsymbol{d}(j) \leftarrow \min(\boldsymbol{d}(j), \boldsymbol{b}_i(j))$
  $sv \leftarrow$ farthest vertex from previous sources
$C \leftarrow B$ after column centering ▷ (mean of each column is 0)
$Y_{s\times 2} \leftarrow$ Top two eigenvectors of $C^T C$
$[\boldsymbol{x}, \boldsymbol{y}] \leftarrow CY$

HDE can be considered a more refined version of PHDE because the information from the graph Laplacian is also captured, and the link to eigenvectors of the Laplacian is clear.

PivotMDS [5] is another fast algorithm that is closely related to PHDE. PivotMDS can be considered a fast approximation of the classical multidimensional scaling (MDS)-based [39] drawing algorithm. The computational costs of PivotMDS and PHDE are identical, but they differ in their derivation. SDE [8] is another method marrying graph-theoretic distances and Laplacian eigenvectors. However, the computational cost and memory requirements are quadratic and comparable to classical MDS. SSDE [9] is a linear-time variant of SDE and based on sampling.

The *multilevel* paradigm is a common speedup heuristic for graph computations. The class of force-directed layout algorithms, exemplified by the Fruchterman-Reingold algorithm [15], are very popular. Multilevel force-directed layout can also lead to linear-time approaches [25]. Parallelizations of multilevel approaches are also well studied [1, 35, 43], especially on Graphics Processing Units [7, 13, 14, 18, 26].

In prior work, Kirmani and Madduri implemented HDE along with several other spectral algorithms in a multilevel setup [27, 33]. They noted that HDE was significantly faster than other algorithms and was also compatible with the multilevel approach.

3 SHARED-MEMORY PARALLELIZATION

We now give a high-level overview of ParHDE, explain design choices, and mention key implementation details. Algorithm 3 gives the pseudocode. We identify three compute-intensive phases in ParHDE: the breadth-first search (BFS) phase where we perform s traversals and compute distance vectors, the D-Orthogonalization phase (also referred to as DOrtho in short) where we construct an orthonormal matrix S given the s distance vectors, and finally the step where we compute $S^T LS$, which we refer to as TripleProd. There are additionally some initialization steps and the eigensolve on the $s \times s$ matrix, which take negligible time.

Comparing Algorithms 1 and 3, we can notice three main changes. First, the BFS and the orthogonalization stages are decoupled in ParHDE. This permits replacing the current approach of choosing vertices (approximate solution to the k-centers problem) with alternatives (such as random selection). Second, notice that we perform

Algorithm 3 ParHDE: high-level overview.

Input: G, s (subspace dimension)
Output: $\boldsymbol{x}, \boldsymbol{y}$

1: Initialize $S \in \mathbb{R}^{n \times (s+1)}$
2: Initialize $B \in \mathbb{R}^{n \times s}$ ▷ Column-major format
3: $\boldsymbol{s}_0 \leftarrow \mathbf{1}/\sqrt{n}$ ▷ Columns of S are indexed from 0
4: $v \leftarrow$ randomly-chosen start vertex
5: **for** $i \leftarrow 1, s$ **do** ▷ **BFS** phase
6: $\quad \boldsymbol{b}_i \leftarrow$ ParallelBFS(v)
7: $\quad \boldsymbol{s}_i \leftarrow \boldsymbol{b}_i$
8: $\quad sv \leftarrow$ farthest vertex from previous sources
9: **for** $i \leftarrow 1, s$ **do** ▷ **DOrtho** phase
10: $\quad$ **for** $j \leftarrow 0, i-1$ **do**
11: $\quad\quad \boldsymbol{s}_i \leftarrow \boldsymbol{s}_i - \left(\frac{\boldsymbol{s}_j' D \boldsymbol{s}_i}{\boldsymbol{s}_j' D \boldsymbol{s}_j}\right) \boldsymbol{s}_j$ ▷ Vector ops
12: $\quad$ **if** $\|\boldsymbol{s}_i\| \leq 10^{-3}$ **then**
13: $\quad\quad$ drop $\boldsymbol{s}_i$
14: $\quad$ **else**
15: $\quad\quad \boldsymbol{s}_i \leftarrow \frac{\boldsymbol{s}_i}{\|\boldsymbol{s}_i\|}$
16: Drop 0^{th} column of S (degenerate vector)
17: $P \leftarrow LS$ ▷ Step 1 of **TripleProd** phase
18: $Z \leftarrow S^T P$ ▷ Step 2 of TripleProd phase
19: $Y_{s \times 2} \leftarrow$ Top two eigenvectors of Z ▷ Eigensolve
20: $[\boldsymbol{x}, \boldsymbol{y}] \leftarrow BY$

Table 1: Asymptotic analysis of various steps of ParHDE. $d_{\max}$ denotes the graph diameter and $\frac{n}{m} \leq \gamma \leq 1$ indicates work reduction with direction-optimizing BFS. ‡ : assuming $\frac{m}{n} \gg s$, $\mathbf{O}(1)$ otherwise.

Phase	Work	Depth	Arith. Int.	Extra Mem.
ParallelBFS	$s(d_{\max} n + \gamma m)$	$s \max(d_{\max}, \log n)$	1	sn
BFS: Other	sn	$s \log n$	1	n
DOrtho	$s^2 n$	$s^2 \log n$	1	n
TripleProd: LS	$s(m+n)$	$\log n$	$s^{\ddagger}$	sn
TripleProd: matmul	$s^2 n$	$\log n$	s	s^2

D-orthogonalization instead of orthogonalization. The resulting vectors can be considered to be approximations to the result of the generalized eigenproblem $L\boldsymbol{x} = \mu D\boldsymbol{x}$, where D is a diagonal matrix of weighted vertex degrees. D-orthogonalization instead of orthogonalization requires a very minor change. Third, instead of SSSP calculations, the pseudocode and subsequent discussion assume an unweighted graph and we use a parallel BFS. We also support parallel SSSP for weighted graphs, but since most publicly available and large-scale real-world graphs are unweighted, we discuss this special case first.

Table 1 gives asymptotic bounds in the work-depth shared memory machine model for various steps, as well as additional memory required for the step, and an estimate of the *arithmetic intensity*. Arithmetic intensity is defined as the ratio of the operation count to the sum of the sizes of input and output. An O(1) arithmetic intensity indicates that there is not much opportunity to exploit temporal locality, and higher arithmetic intensities mean that it may be possible to improve data reuse by optimizations such as tiling.

We use a level-synchronous parallel BFS algorithm for the BFS step. Specifically, we use the direction-optimizing BFS [3, 42], an approach developed for traversing low-diameter graphs with skewed degree distributions. Since there is a dependency between iterations of the BFS phase loop, multiple BFSes cannot be performed in parallel, and so the there is a multiplicative s term in the depth (or span) bound. Also, the level-synchronous algorithm has a worst-case $O(n)$ depth (consider a linear chain of vertices). This has not prevented its current widespread use. In future work, we will augment this step with a low diameter decomposition [11, 12, 37] to improve the depth bounds. There is not much opportunity for reuse if the searches are performed iteratively, but if the source selection is decoupled from the traversal step, there could be possible opportunities for reuse and reducing memory traffic. Also, note the significant $O(sn)$ memory requirements, as the s distance vectors need to be stored until the final step.

The "BFS: Other" step in Table 1 refers to the source selection calculations. The multiplicative factor of s appears in both the work and depth expressions. Since we need to determine the farthest vertex from all visited sources, we currently use a loop similar to lines 13-14 of Algorithm 1. This is not shown in Algorithm 3, but the bounds correspond to the reduction with the max operator.

The DOrtho phase requires a Gram-Schmidt-like orthogonalization procedure where we operate on $O(n)$-sized vectors in the inner loop. There is an s^2 multiplicative term in both the work and depth expressions because of the loop-carried dependencies. We parallelize the vector dot product and addition operations required for line 11. The $\log n$ depth bound is because of the dot product summation.

The TripleProd phase can be separated into two steps, calculating $P = LS$ first, and then computing $Z = S^T P$. Alternately, we can compute $S^T L$ first and then compute $(S^T L)S$. The second step is a product of two dense matrices of dimensions $s \times n$ and $n \times s$, for which library routines are readily available. The arithmetic intensity is s for this step. Assuming $m/n \gg s$, the first step will perform more work. This step can be viewed as performing s sparse matrix and dense vector multiplications (SpMVs), and so the corresponding work and depth bounds are listed in Table 1. This step also requires additional space to store the temporary product. The arithmetic intensity of this step is generally O(1), but can be $O(s)$ when $m/n \gg s$. Notice that the depth of both the TripleProd phase steps have no dependence on s, which is a desirable feature.

3.1 Implementation Details

We store the graph in a compressed sparse row (CSR)-like format. Because we consider only unweighted graphs, we do not store weights or explicitly construct the Laplacian. We use double precision arithmetic for the eigenvector-related calculations.

We modify the direction-optimizing BFS [3] in the GAP Benchmark Suite [2, 4] for our purpose. Level-synchronous approaches are typically comprised of *top-down* search phases, where unexplored adjacencies of vertices in the current frontier are marked

and queued up for exploration. The direction-optimizing BFS additionally proposes a bottom-up traversal phase, where unexplored vertices scan their adjacency lists to identify a possible parent vertex and set them. Heuristically switching between top-down and bottom-up strategies for different levels is shown to significantly reduce the number of edges traversed for low-diameter graphs with highly skewed degree distributions. While the GAP BFS maintains a BFS tree by storing parents of reachable vertices, we further need distances from the source vertex. We thus modify the GAP code to compute distances in an atomic-free manner. Note that GAP already uses the compare-and-swap atomic primitive for tracking parent pointers and we do not introduce additional overhead.

For the vector operations in DOrtho phase, we experiment with Basic Linear Algebra Subroutines (BLAS) routines in the Intel Math Kernel Library (MKL) [41], as well as relevant functions in Eigen C++ linear algebra library [20], but found our implementations to be generally faster. So we use our own code with OpenMP pragmas for portability and simplicity. In future work, we will look at eliminating the s^2 term in the depth using alternate parallel formulations of the D-orthogonalization phase [32].

In the TripleProd phase, the *LS* calculation dominates running time. We use the BLAS *dgemm* routine in MKL for the less expensive $S^T(LS)$ calculation. We experimented with sparse matrix vector multiplication routines in MKL, but again found our OpenMP code with loop collapse pragmas to be faster. We note that performance can be further improved for special cases such as $m/n \gg s$ or $s \gg 1$.

3.2 Parallelization of PHDE and PivotMDS

PHDE is very similar to ParHDE, but does not have the *LS* product. There is a column centering step which requires subtracting the mean of every column from the column entries. We implement this in a two-phase manner, computing the column means in the first phase and performing the subtraction in the second phase. PivotMDS requires double-centering of the distance matrix [5], which is computationally similar to column centering. The matrix multiplication and eigenvector computation steps in PivotMDS are similar to PHDE.

3.3 Extension to weighted graphs

We also extend our work to weighted graphs by performing SSSP instead of parallel BFS. We use the SSSP implementation from GAP, which implements the Δ-stepping algorithm [34]. This algorithm uses a bucketing data structure parameterized by the value Δ and partitions edges into two groups, heavy and light.

The GAP implementation creates two types of buckets, shared buckets and thread-local buckets. Each iteration proceeds in two phases. In the first phase each thread picks a vertex out of the current shared bucket and tries to relax its neighbours. If they are updated, the vertices are added to the thread-local bucket. In the next phase, the threads add vertices in their local bucket to the corresponding shared bucket. The implementation does not recycle the buckets and ignores settled vertices to improve performance. We modify GAP to work with our CSR-like data structure and to retrieve the settled distances of the vertices.

4 EVALUATION

In this section, we evaluate the scalability and efficiency of ParHDE. We also discuss simple extensions and applications beyond graph drawing in Section 4.5.

4.1 Experimental Setup

Input. Table 2 gives details of the test graphs used. The number of edges and the number of vertices are listed. We use the synthetic graph generators in the GAP Benchmark Suite [2, 4] to generate the urand27 and kron27 synthetic random graphs. The rest of the graphs are based on matrices from the SuiteSparse matrix collection [10]. Since ParHDE expects a connected undirected simple graph as input, we preprocess the matrices and graphs to remove self loops and parallel edges. We also ignore edge direction for directed graphs and extract the largest connected component. Because of this preprocessing, the vertex and edge counts might differ from the ones reported in the original sources. We attempt to retain the original vertex ordering as far as possible. When extracting the largest connected component, we remove vertices not in the component and renumber the vertices to be contiguous, but preserving the original implied ordering.

Table 2: A collection of undirected graphs used for evaluating ParHDE. The first two graphs are generated using the GAP Benchmark Suite [2], and the remaining graphs listed are based on sparse matrices from the SuiteSparse matrix collection [10]. We preprocess the graphs to extract the largest connected component and relabel vertex identifiers. The number of edges (m) and vertices (n) after preprocessing are given.

Graph	m	n
urand27	2 147 483 376	134 217 728
kron27	2 111 622 405	63 045 458
sk-2005	1 810 050 743	50 634 118
twitter7	1 202 513 046	41 652 230
road_usa	28 854 312	23 947 347
cage14	12 812 282	1 505 785
CurlCurl_4	12 067 676	2 380 515
kkt_power	6 482 320	2 063 494
ecology1	1 998 000	1 000 000
pa2010	1 029 231	421 545

To shed more insight into performance results obtained, we show in Figure 2 the distribution of *adjacency list gaps* using the Fibonacci binning [40] technique. Suppose a vertex u has four adjacencies v_1, v_2, v_3, and v_4. Assume that the adjacencies are stored in sorted order, i.e., $v_1 < v_2$, $v_2 < v_3$, and $v_3 < v_4$. Then, we term $v_2 - v_1$, $v_3 - v_2$, and $v_4 - v_3$ as the adjacency list gaps, or gaps in short. Gaps are an indicator of memory locality: low values mean that nearby memory locations are accessed when performing accesses of the type $S[v]$, where $v \in \text{Adj}(u)$ and S is an array of size n. Figure 2 is a histogram of gaps, with the histogram bin widths set to numbers in the Fibonacci sequence. A point $[x_i, c]$ on the plot means that there are c occurrences of gaps in the range $[x_{i-1}, x_i)$. $x_0 = 0$, $x_1 = 1$,

$x_i = x_{i-1} + x_{i-2}$ for $i \geq 2$. Note that $\sum c = 2m - n$. Further, note that both the axes use a logarithmic scale. Use the trend line for the uniform random graph urand27 as a guide to read the chart. Ideally, we would like low gaps to occur more frequently. A linear chain of n vertices with a linear vertex ordering would have a gap of just 2 occurring $n - 2$ times and is an example of an ideal case. The urand27 ordering would show poor locality of memory reference (again, for accesses of the type $S[v]$, where $v \in \text{Adj}(u)$). The trend for the kron27 graph is also as expected and similar to the urand27 trend, because the vertex identifiers are random shuffled in the graph generator. For the remaining three real-world graphs, we retain the ordering given in the source collection. Observe that the distribution for sk-2005 has a favorable trend for enhancing memory locality. If we randomly shuffle the vertex identifiers, we lose this locality and the trend line for sk-2005 will nearly coincide with the urand27 and kron27 lines.

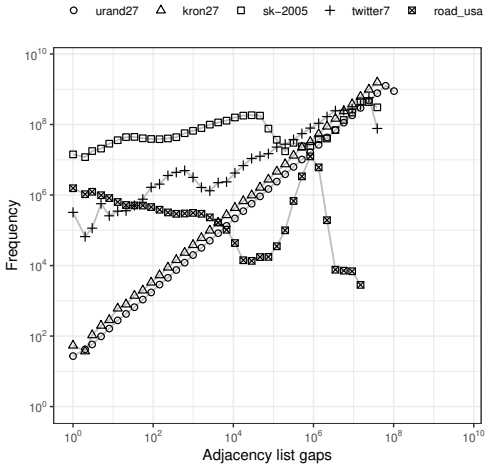

Figure 2: Distribution of adjacency list gaps for five test graphs.

Hardware. We report performance results primarily on a 28-core compute node of the Pittsburgh Supercomputing Center Bridges supercomputer. This node from the regular shared memory partition has two 14-core Intel Xeon E5-2695 v3 (Haswell) processors. Each processor has 35 MB last level cache and each node has 128 GB DDR4-2133 memory. In addition, we report some results on 80 cores of a 288-core compute node from the extreme shared memory partition of Bridges. This node has 16 18-core Intel Xeon E7-8880 v3 (Haswell) processors. Each processor has 45 MB last level cache and the node has 12 TB memory. Due to allocation limits, we could not get dedicated access to this large memory node, and care should be taken while comparing results on this system to the 28-core node results.

Software. ParHDE is developed in C++ and OpenMP. We use the Eigen library [20], GAP, and the Intel Math Kernel Library (MKL) [41]. We use version 3.3.7 of Eigen for computing eigenvalues and eigenvectors of the small $s \times s$ matrix. Our BFS code is based on the direction-optimizing BFS in GAP (version 1.2). We use Intel MKL version 2018 Update 4 for the double-precision generalized matrix multiply *dgemm*. To compile the code, we use the Intel C++ compiler version 18.0.4 with O2 optimization. OpenMP threads are bound to cores using a *compact* thread pinning scheme and hyperthreading is disabled. With these settings, we observed a STREAM Triad bandwidth of 112 GB/s on the 28-core system. After obtaining coordinates, we use an open-source Portable Network Graphics (PNG) format file writer to create the drawings. Edges are drawn as straight lines of fixed thickness and we have not experimented with color. Note that the PNG file writing step is untimed.

4.2 Speedup over Prior Work

In Table 3, we report the speedup achieved over the fastest parallel HDE implementation [27, 33] known to us. This approach uses the Eigen library extensively and does not use parallel BFS. Further, in the prior approach, the use of an Eigen function for constructing the Laplacian matrix leads to a significant increase in the peak memory footprint. Because of this, we were unable to execute this code for the largest graphs in our collection on the 28-core 128 GB system, and instead use 80 cores on the large memory node to get a full set of results. Further, note that the subspace dimension value s is set to 10 for this set of results. Unless noted otherwise, we will use 10 as the default setting for obtaining running time results.

Table 3: Execution times of ParHDE and the prior parallel implementation from [33], and speedup achieved by ParHDE. $s = 10$. Results on large mem node.

Graph	Time (s)		Speedup
	ParHDE	Prior Par.	
urand27	72	1301	18.0 ×
kron27	47	688	14.7 ×
sk-2005	18	131	7.3 ×
twitter7	34	372	10.9 ×
road_usa	13	36	2.9 ×

We observe speedup to be correlated with the graph size, with larger graphs resulting in a higher speedup. The road_usa graph has a much lower average vertex degree than the rest of the graphs and a relatively higher diameter, and is not a good instance for the direction-optimizing BFS. We do not expect substantial improvement over the prior sequential BFS, and this explains the comparatively lower 2.9× speedup observed.

Since $s = 50$ is a common choice in HDE, ParHDE is able to process billion-edge graphs in the order of a few minutes. Because of HDE's low work complexity, ParHDE is significantly faster than recent parallelizations. For instance, MulMent [35] reports a running time of 27 seconds for a graph with a million vertices and 3 million edges on a large shared-memory server, whereas ParHDE is two orders of magnitude faster for similar-sized graphs. The running time of the ForceAtlas2 GPU implementation of Brinkmann et al. [7] is in the order of several minutes for large graphs that fit in GPU memory, and we estimate ParHDE to be an order-of-magnitude faster.

4.3 Running time Breakdown and Parallel Scaling

Both ParHDE and the prior approach share three main stages: performing s BFSes, computing the $S^T LS$ product, and orthogonalization of the distance vectors obtained from BFS using a Gram Schmidt-like procedure. In Figure 3, we show the percentage of time spent in each of these stages for three different scenarios: parallel 28-core execution of ParHDE, sequential execution of ParHDE, and parallel execution of the prior approach. Comparing these charts lets us understand the reasons for the overall speedup. In all three cases, the time for the BFS and the triple product steps dominate the time for the D-orthogonalization. Further, the remainder of time (eigenvector computation on the small matrix) is negligible. The ratio of time spent in various stages varies across graph instances, and depends on the edge count, vertex count, vertex ordering, and vertex degree distribution. Comparing the left and middle charts, it is apparent that TripleProd scales better than BFS for most graphs. Comparing the right and left charts, it is clear that ParHDE benefits from a much faster parallel BFS. Across all inputs, we note that DOrtho constitutes a greater fraction of overall time for road_usa and sk-2005. Also, looking for trends across all charts, we note that urand27, kron27, and twitter7 have similar breakdown plots, whereas sk-2005 and road_usa are somewhat different from these. sk-2005 is the odd one out because we would expect its breakdown profile to be similar to the other three low-diameter graphs. We will look at the reason in the next subsection.

In Figure 4, we look at overall parallel scaling and the scaling of individual stages for the five large inputs on the 28-core system. The overall speedups are as expected. The uniform random graph instance urand27 achieves the best speedup for all the steps that are graph structure-dependent. This is because of good overall parallel load balance (regular vertex degree distribution) and the latency-bound nature of the computation (because vertex ordering and other locality-enhancing optimizations will not help by design). The *LS* step in the TripleProd phase is less structure-dependent than BFS, and so the TripleProd phase shows better scaling than BFS on all instances. urand27's near-linear scaling for TripleProd is due to the large number of random reads (i.e., random cache line fetches) each thread makes, and the good overall load balance, making it latency-bound. For the other graphs, the vertex ordering and degree distributions affect overall scaling.

DOrtho becomes memory bandwidth-bound quickly and does not show much improvement beyond 7 threads. Recall that DOrtho performs $O(s^2)$ dot products, and each dot product of size $O(n)$ is parallelized across p threads. The need to orthogonalize against previous vectors introduces a loop dependence and prevents distributing the $O(s^2 n)$ work across threads.

Table 4 reports ParHDE execution times and relative speedup (i.e., speedup over single-threaded run) for all the test graphs. The speedups are also previously visualized in Figure 4 (left). Note the relatively low running time for sk-2005 in comparison to twitter7. Since the graphs are ordered by edge counts, it is expected that the running time of sk-2005 to be higher than twitter7, and closer to kron27. We note this trend even on the large memory node (see running times in Table 3). In Table 5, we give the 28-core running times and relative speedup of PHDE and PivotMDS for the five largest graphs. In combination with Figure 6, it is clear that the overall performance is dominated by the time taken for parallel BFS.

Table 4: ParHDE execution time on 28-core system and relative speedup (speedup over 1-core time).

Graph	Time (s)	Rel. Speedup
urand27	52.5	24.5×
kron27	34.3	14.8×
sk-2005	9.9	11.3×
twitter7	23.8	11.0×
road_usa	4.6	7.1×
CurlCurl_4	0.6	5.8×
kkt_power	0.5	8.1×
cage14	0.3	9.1×
ecology1	0.3	4.2×
pa2010	0.1	4.2×

Table 5: PHDE and PivotMDS execution times on 28-core system and relative speedup (speedup over 1-core time).

Graph	PHDE		PivotMDS	
	Time (s)	Rel. Speedup	Time (s)	Rel. Speedup
urand27	12.5	23.7×	13.9	23.4×
kron27	4.8	12.4×	4.6	20.1×
sk-2005	4.6	9.2×	4.9	11.6×
twitter7	5.7	6.5×	5.8	9.1×
road_usa	3.1	6.1×	3.1	7.9×

4.4 Additional Performance Analysis

We next consider the impact of increasing the subspace dimension (or the number of source vertices) s. Recall that the BFS and TripleProd phases scale linearly with s, whereas DOrtho phase work scales quadratically. This is apparent in the breakdown chart of Figure 5 (left), where DOrtho takes considerable longer across all instances in comparison to Figure 3 (left).

Next, in Figure 5 (middle), we show the breakdown of time spent in the BFS phase into actual traversal and other steps (such as finding the farthest source). It is clear that the traversal time dominates. In Figure 5 (right), we show the breakdown of the TripleProd phase into the two constituent steps, the *LS* computation, followed by the *dgemm* $S^T(LS)$ calculation. We note that urand27, kron27, and twitter7 have a similar profile (negligible *dgemm* time), whereas in sk-2005 and road_usa, the *dgemm* time is higher (or alternately viewing this as *LS* time being lower).

We also run some experiments to understand the performance of the SSSP-based implementation. When using unit weights for road_usa, the SSSP approach is only 18% slower than the BFS-based approach. However, for real or random integer weights, the performance is dependent on the setting for Δ and the slowdown compared to unweighted BFS is 3.66× or more for road_usa. A detailed analysis is left for future work.

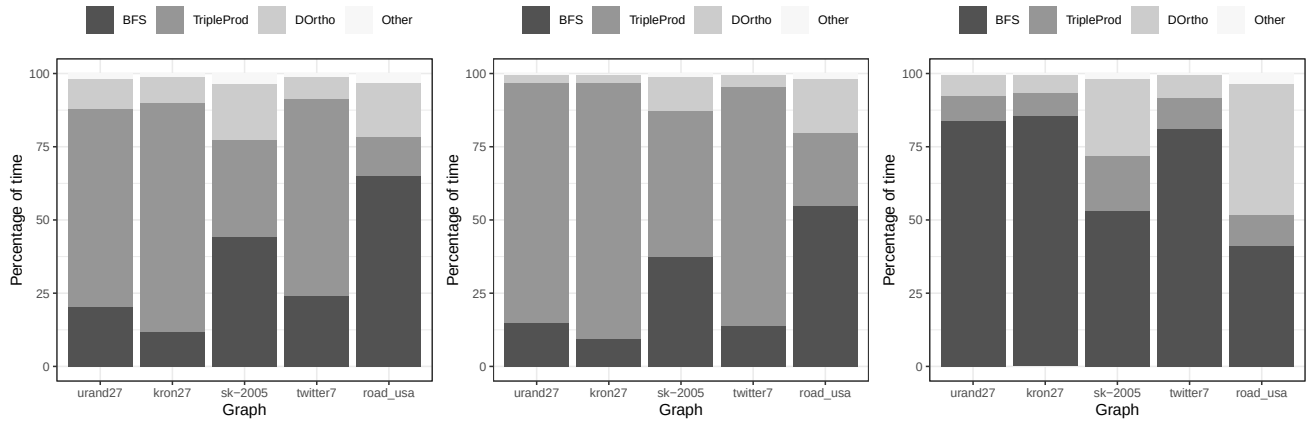

Figure 3: Component-wise breakdown of execution time: ParHDE on 28-core system (left), ParHDE on 1 core of 28-core system (middle), prior work on large memory node.

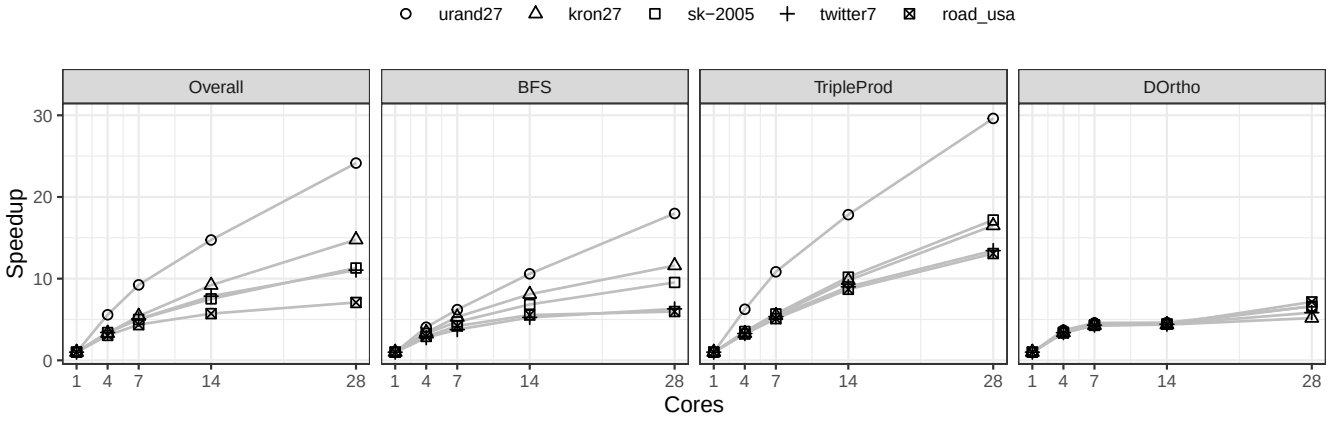

Figure 4: Relative scaling of ParHDE and constituent steps on the 28-core system.

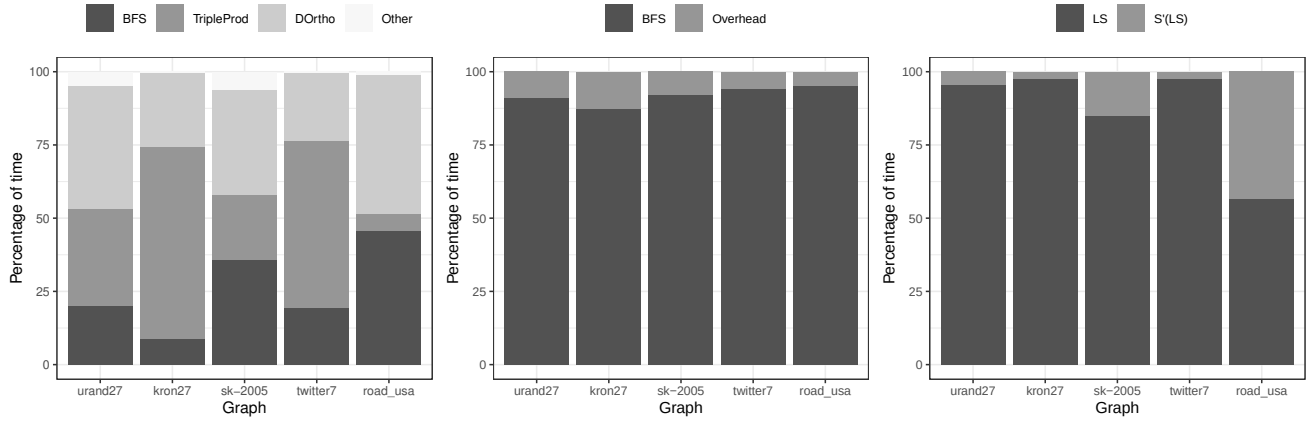

Figure 5: Analyzing ParHDE performance: execution time breakdown with 50 sources (left), breakdown of BFS stage into traversal time and other steps (middle), breakdown of TripleProd step into constituent steps (right).

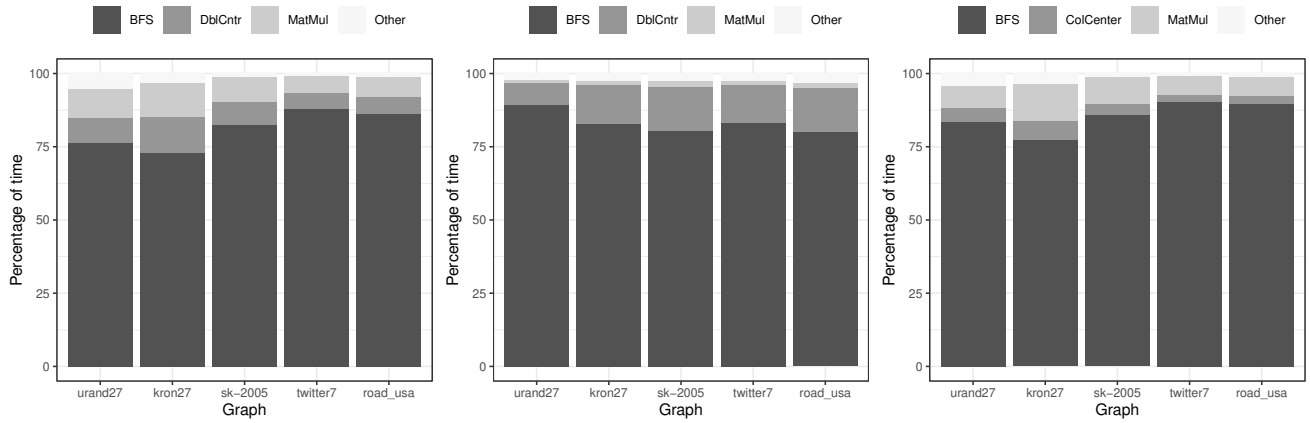

Figure 6: Analyzing PivotMDS and PHDE performance: PivotMDS execution time breakdown on 28 cores (left), PivotMDS breakdown on 1 core (middle), PHDE breakdown on 28 cores (right).

We resolve the seemingly anomalous running time and scaling behavior of sk-2005, and to some extent, road_usa. The overall time of sk-2005 is lower than twitter7 because the *LS* calculation step is considerably faster. *LS* can be viewed as s SpMVs. In a compressed sparse row-based implementation of SpMV, possibly irregular accesses to the vector dominate running time. If there is good locality, components of the vector are cached and the memory access time is amortized. The accesses to the vector are dependent on the adjacency list gap distribution. This is the reason we gave Figure 2. Because of the locality-enhancing vertex ordering of sk-2005, the *LS* step is much faster than expected. This is seen to some extent in road_usa as well. If we use a randomly permuted ordering of vertices, the time for *LS* increases by a factor of 6.8×, and the overall time increases by a factor of 3.5×. This observation highlights the benefits of locality-enhancing vertex orderings, and the need to conduct performance evaluations with consistent baselines. Additionally, we note that our *LS* implementation is consistently faster than MKL's *mkl_sparse_d_mm*() routine, with an average speedup of 2.50×. Further, MKL requires allocating a sparse Laplacian matrix (an untimed step), which our implementation avoids by using a dense degrees array to calculate the diagonal entry.

In addition to the approximate solution to the k-centers problem for pivot selection, we experiment with a random pivot selection strategy. Here, pivots are chosen uniformly at random without repetition, and threads concurrently perform multiple BFSes. On the other hand, for the default strategy, each BFS is parallelized. The random pivots strategy has lower overhead for smaller graphs and when the number of pivots is greater than the number of threads. In Table 6, we compare the execution time of the BFS phase using these two strategies when using 30 sources on the five smallest graphs in our collection. We observe higher speedups with random pivots for high-diameter graphs and for smaller graphs.

For orthogonalization, we use Modified Gram Schmidt (MGS) with only Level 1 BLAS operations as the default option. However, we also experiment with a Classical Gram Schmidt procedure using Level 2 BLAS operations (i.e., matrix-vector operations) and summarize the results in Table 7. The CGS approach is consistently faster. However, the use of CGS requires all distance vectors to be precomputed before the orthogonalization procedure is performed, whereas the default procedure can also be executed with a coupled BFS and D-orthogonalization steps. We did not observe any significant change in drawing quality with this alternate procedure.

Table 6: Performance Impact of using randomly-chosen pivots on execution time of the BFS phase. Comparison on 28-core system and with 30 sources.

Graph	Default Alg. Time (s)	Rand. Pivots Time (s)	Rel. Speedup
CurlCurl_4	0.91	0.33	2.8×
kkt_power	1.10	0.66	1.7×
cage14	0.66	0.47	1.4×
ecology1	0.88	0.09	10.1×
pa2010	0.42	0.05	9.1×

Table 7: Performance Impact of using Classical Gram Schmidt on execution time of the D-Orthogonalization phase. Comparison on 28-core system.

Graph	Default Alg. (MGS) Time(s)	CGS Time (s)	Rel. Speedup
urand27	5.9	2.7	2.2×
kron27	3.0	1.1	2.8×
sk-2005	2.0	0.8	2.5×
twitter7	1.8	0.7	2.5×
road_usa	0.8	0.4	2.1×

4.5 ParHDE Extensions and Applications

4.5.1 Other graph drawing algorithm variants. At least three other algorithms can be viewed as special cases or trivial extensions of HDE/ParHDE. If we just perform orthogonalization instead of

D-orthogonalization (as shown in Algorithm 1), we get vectors that are approximations to the eigenvectors of the Laplacian. For graphs with uniform degree distributions, the results are more or less identical. We can obtain the coordinates for this case with a trivial change and a minor impact on running time. We discussed performance of PHDE and PivotMDS in the prior sections. Because the HDE algorithm designed by Koren [30, 31] (and as used in this paper) was proposed after the original PHDE algorithm [23, 24], nearly all papers mean PHDE when they refer to HDE (e.g., [6, 17, 21]). We do not include actual drawings in this submission, because they have been comprehensively evaluated in prior work [6, 17, 21], and we get similar drawings with our code. To provide a comparison to the barth5 drawings shown in Figure 1, we give in Figure 7 drawings generated with ParHDE and randomly-chosen pivots (top), with PHDE (middle), and with PivotMDS (bottom). All the drawings capture global structure with four "holes".

4.5.2 A "Zoom" feature for multilevel interactive visualization. Since we have the capability of real-time layout for million-edge graphs, we implemented a "zoom" feature in ParHDE. The idea is that the user can select a vertex in the global layout, and a zoomed-in visualization of the neighborhood can be shown. See Figure 8 for an example of a visualization of the 10-hop neighborhood of a random vertex in the barth5 graph. This would be useful for future browser-based interactive graph visualization.

4.5.3 Use as preprocessing step for iterative eigensolver. It was noted in prior work by Kirmani et al. [27] that HDE followed by a light-weight *weighted centroid* refinement step can closely approximate the eigenvectors (i.e., one could go from the top drawing to the bottom drawing in Figure 1). Further, in Table 6 of [27], it is shown that this scheme is 22× to 131× than the power iteration across a collection of test graphs. These results indicate that ParHDE could be used as a preprocessing step for modern eigensolvers such as LOBPCG [29].

4.5.4 Miscellaneous extensions. It is known that PHDE's layout serves as a good initialization for layout using stress majorization [16]. We could consider replacing PHDE by ParHDE to see if this speeds up this optimization problem. The vertex coordinates from ParHDE can be used by geometric graph partitioners. The ScalaPart [28] partitioner uses a force-directed layout to compute coordinates. We can use ParHDE instead. Vertex coordinates can also be used to reduce the work performed in the Kernighan-Lin based refinement stages of graph partitioners.

We have used the layouts to visualize output of graph partitioning and clustering algorithms, by using different colors for intra- and inter-partition edges. These visualizations shed insights into the inner workings of partitioning/clustering algorithms.

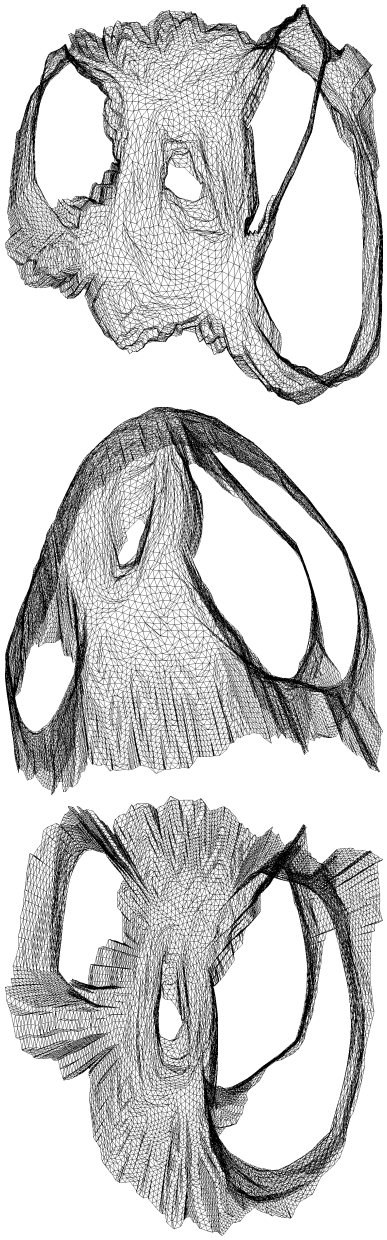

Figure 7: Various drawings of the barth5 graph (also see Figure 1) using ParHDE with randomly-chosen sources (top), PHDE (middle), and PivotMDS (bottom).

5 CONCLUSIONS AND FUTURE WORK

We summarize key conclusions from the evaluation: (i) ParHDE's execution time is in the order of minutes for billion-edge graphs: in specific, for four graphs with edge counts between 1.2 billion to 2.1 billion edges (and vertex counts between 42 million to 134 million), the running time ranged from 10 seconds to 53 seconds. (ii) We can organize ParHDE into three phases: a graph traversal phase, a sparse matrix vector multiplication (SpMV) phase, and an orthogonalization phase. We observe that, as expected, the graph traversal and the matrix multiplication phases have a linear dependence on the number of source vertices, and the orthogonalization phase has a quadratic dependence. (iii) We noted that the initial ordering of vertex identifiers has a significant performance impact on the SpMV step. In future work, we will adapt ParHDE to be compatible with the multilevel approach and use ParHDE as a preprocessing step for eigensolvers and geometric graph partitioning methods.

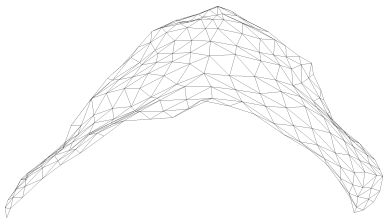

Figure 8: A zoomed drawing of the 10-hop neighborhood of a vertex in the barth5 graph.

ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation (NSF) grants CCF-1439057 and OAC-1253881. The work also used the Extreme Science and Engineering Discovery Environment (XSEDE) Bridges supercomputer at the Pittsburgh Supercomputing Center through allocations TG-SEE180003 and TG-SEE180004. XSEDE is supported by NSF grant number ACI-1548562.

REFERENCES

[1] Alessio Arleo, Walter Didimo, Giuseppe Liotta, and Fabrizio Montecchiani. 2017. Large graph visualizations using a distributed computing platform. *Information Sciences* 381 (2017), 124 – 141.
[2] Scott Beamer. 2017. The GAP Benchmark Suite. https://github.com/sbeamer/gapbs, last accessed June 2020.
[3] Scott Beamer, Krste Asanović, and David Patterson. 2013. Direction-Optimizing Breadth-First Search. *Scientific Programming* 21, 3-4 (2013), 137–148.
[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. https://arxiv.org/abs/1312.3749, last accessed June 2020.
[5] Ulrik Brandes and Christian Pich. 2007. Eigensolver Methods for Progressive Multidimensional Scaling of Large Data. In *Proc. Graph Drawing (GD)*, M. Kaufmann and D. Wagner (Eds.). Springer, Berlin, Heidelberg, 42–53.
[6] Ulrik Brandes and Christian Pich. 2009. An Experimental Study on Distance-Based Graph Drawing. In *Proc. Graph Drawing (GD)*, I. G. Tollis and M. Patrignani (Eds.). Springer, Berlin, Heidelberg, 218–229.
[7] Govert G. Brinkmann, Kristian F.D. Rietveld, and Frank W. Takes. 2017. Exploiting GPUs for Fast Force-Directed Visualization of Large-Scale Networks. In *Proc. Int'l. Conf. on Parallel Processing (ICPP)*. IEEE, Piscataway, NJ, 382–391.
[8] Ali Civril, Malik Magdon-Ismail, and Eli Bocek-Rivele. 2006. SDE: Graph Drawing Using Spectral Distance Embedding. In *Proc. Graph Drawing (GD)*, P. Healy and N. S. Nikolov (Eds.). Springer, Berlin, Heidelberg, 512–513.
[9] Ali Çivril, Malik Magdon-Ismail, and Eli Bocek-Rivele. 2007. SSDE: Fast Graph Drawing Using Sampled Spectral Distance Embedding. In *Proc. Graph Drawing (GD)*, M. Kaufmann and D. Wagner (Eds.). Springer, Berlin, Heidelberg, 30–41.
[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1–25. https://sparse.tamu.edu/, last accessed June 2020.
[11] Laxman Dhulipala. 2018. GBBS: Graph Based Benchmark Suite. https://github.com/ldhulipala/gbbs, last accessed June 2020.
[12] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *Proc. Symp. on Parallelism in Algorithms and Architectures (SPAA)*. ACM, New York, NY, 393–404.
[13] Niklas Elmqvist, Thanh-Nghi Do, Howard Goodell, Nathalie Henry, and Jean-Daniel Fekete. 2008. ZAME: Interactive Large-Scale Graph Visualization. In *Proc. Pacific Visualization Symposium*. IEEE, Piscataway, NJ, 215–222.
[14] Yaniv Frishman and Ayellet Tal. 2007. Multi-Level Graph Layout on the GPU. *IEEE Trans. on Visualization and Computer Graphics* 13, 6 (Nov 2007), 1310–1319.
[15] Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph drawing by force-directed placement. *Software: Practice and Experience* 21, 11 (1991), 1129–1164.
[16] Emden R. Gansner, Yehuda Koren, and Stephen North. 2005. Graph Drawing by Stress Majorization. In *Proc. Graph Drawing (GD)*, J. Pach (Ed.). Springer, Berlin, Heidelberg, 239–250.
[17] Helen Gibson, Joe Faith, and Paul Vickers. 2013. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization* 12, 3-4 (2013), 324–357.
[18] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C. Hart. 2009. Rapid Multipole Graph Drawing on the GPU. In *Proc. Graph Drawing (GD)*, I. G. Tollis and M. Patrignani (Eds.). Springer, Berlin, Heidelberg, 90–101.
[19] Teofilo F. Gonzalez. 1985. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science* 38 (1985), 293–306.
[20] Gaël Guennebaud, Benoit Jacob, Philip Avery, Abraham Bachrach, and Sebastien Barthelemy. 2010. Eigen v3. http://eigen.tuxfamily.org, last accessed June 2020.
[21] Stefan Hachul and Michael Jünger. 2006. An Experimental Comparison of Fast Algorithms for Drawing General Large Graphs. In *Proc. Graph Drawing (GD)*, P. Healy and N. S. Nikolov (Eds.). Springer, Berlin, Heidelberg, 235–250.
[22] Kenneth M Hall. 1970. An r-dimensional quadratic placement algorithm. *Management science* 17, 3 (1970), 219–229.
[23] David Harel and Yehuda Koren. 2002. Graph Drawing by High-Dimensional Embedding. In *Proc. Graph Drawing (GD)*, M. T. Goodrich and S. G. Kobourov (Eds.). Springer, Berlin, Heidelberg, 207–219.
[24] David Harel and Yehuda Koren. 2004. Graph Drawing by High-Dimensional Embedding. *Journal of Graph Algorithms and Applications* 8, 2 (2004), 195–214.
[25] Yifan Hu and Lei Shi. 2015. Visualizing large graphs. *Wiley Interdisciplinary Reviews: Computational Statistics* 7, 2 (2015), 115–136.
[26] Stephen Ingram, Tamara Munzner, and Marc Olano. 2009. Glimmer: Multilevel MDS on the GPU. *IEEE Trans. on Visualization and Computer Graphics* 15, 2 (March 2009), 249–261.
[27] Shad Kirmani and Kames Madduri. 2018. Spectral Graph Drawing: Building Blocks and Performance Analysis. In *Proc. Workshop on Graph Algorithm Building Blocks (GABB)*. IEEE, Piscataway, NJ, 269–277.
[28] Shad Kirmani and Padma Raghavan. 2013. Scalable parallel graph partitioning. In *Proc. Int'l. Conf. on high performance computing, networking, storage and analysis (SC)*. ACM, New York, NY, 1–10.
[29] Andrew V. Knyazev. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing (SISC)* 23, 2 (2001), 517–541.
[30] Yehuda Koren. 2004. Graph Drawing by Subspace Optimization. In *Proc. Sixth Joint Eurographics - IEEE TCVG Conf. on Visualization (VISSYM)*. Eurographics Association, Goslar, DEU, 65–74.
[31] Yehuda Koren. 2005. Drawing graphs by eigenvectors: theory and practice. *Computers & Mathematics with Applications* 49, 11 (2005), 1867–1888.
[32] Steven J. Leon, Åke Björck, and Walter Gander. 2013. Gram-Schmidt orthogonalization: 100 years and more. *Numerical Linear Algebra with Applications* 20, 3 (2013), 492–532.
[33] Kamesh Madduri and Shad Kirmani. 2019. SpectralGraphDrawing. https://github.com/kmadduri/SpectralGraphDrawing, last accessed June 2020.
[34] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114 – 152.
[35] Henning Meyerhenke, Martin Nöllenburg, and Christian Schulz. 2018. Drawing Large Graphs by Multilevel Maxent-Stress Optimization. *IEEE Trans. on Visualization and Computer Graphics* 24, 5 (May 2018), 1814–1827.
[36] Jianbo Shi and Jitendra Malik. 2000. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22, 8 (2000), 888–905.
[37] Julian Shun, Laxman Dhulipala, and Guy Blelloch. 2014. A Simple and Practical Linear-work Parallel Algorithm for Connectivity. In *Proc. Symp. on Parallelism in Algorithms and Architectures (SPAA)*. ACM, New York, NY, 143–153.
[38] Daniel Spielman. 2012. Spectral Graph Theory. In *Combinatorial Scientific Computing*, Uwe Naumann and Olaf Schenk (Eds.). CRC Press, Boca Raton, FL, Chapter 18, 495–524.
[39] Warren S. Torgerson. 1965. Multidimensional scaling of similarity. *Psychometrika* 30, 4 (1965), 379–393.
[40] Sebastiano Vigna. 2013. Fibonacci Binning. https://arxiv.org/abs/1312.3749, last accessed June 2020.
[41] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Springer International Publishing, Cham, 167–188.
[42] Yuichiro Yasui and Katsuki Fujisawa. 2017. Fast, Scalable, and Energy-Efficient Parallel Breadth-First Search. In *The Role and Importance of Mathematics in Innovation*, B. Anderssen, P. Broadbridge, Y. Fukumoto, N. Kamiyama, Y. Mizoguchi, K. Polthier, and O. Saeki (Eds.). Springer, Singapore, 61–75.
[43] Enas Yunis, Rio Yokota, and Aron Ahmadia. 2012. Scalable Force Directed Graph Layout Algorithms Using Fast Multipole Methods. In *Proc. Int'l. Symp. on Parallel and Distributed Computing (ISPDC)*. IEEE, Piscataway, NJ, 180–187.