Parallel Clustered Low-rank Approximation of Graphs and Its Application to Link Prediction

Xin Sui[1], Tsung-Hsien Lee[2], Joyce Jiyoung Whang[1], Berkant Savas[3⋆], Saral Jain[1], Keshav Pingali[1], and Inderjit Dhillon[1]

[1] Department of Computer Science, The University of Texas at Austin
[2] Department of Electrical and Computer Engineering, The University of Texas at Austin
[3] Department of Mathematics, Linköping University

Abstract. Social network analysis has become a major research area that has impact in diverse applications ranging from search engines to product recommendation systems. A major problem in implementing social network analysis algorithms is the sheer size of many social networks, for example, the Facebook graph has more than 900 million vertices and even small networks may have tens of millions of vertices. One solution to dealing with these large graphs is dimensionality reduction using spectral or SVD analysis of the adjacency matrix of the network, but these global techniques do not necessarily take into account local structures or clusters of the network that are critical in network analysis. A more promising approach is *clustered* low-rank approximation: instead of computing a global low-rank approximation, the adjacency matrix is first clustered, and then a low-rank approximation of each cluster (*i.e.*, diagonal block) is computed. The resulting algorithm is challenging to parallelize not only because of the large size of the data sets in social network analysis, but also because it requires computing with very diverse data structures ranging from extremely sparse matrices to dense matrices. In this paper, we describe the first parallel implementation of a clustered low-rank approximation algorithm for large social network graphs, and use it to perform link prediction in parallel. Experimental results show that this implementation scales well on large distributed-memory machines; for example, on a Twitter graph with roughly 11 million vertices and 63 million edges, our implementation scales by a factor of 86 on 128 processes and takes less than 2300 seconds, while on a much larger Twitter graph with 41 million vertices and 1.2 billion edges, our implementation scales by a factor of 203 on 256 processes with a running time about 4800 seconds.

Keywords: Social network analysis, link prediction, parallel computing, graph computations, clustered low-rank approximation.

1 Introduction

Networks are increasingly used to model mechanisms and interactions in a wide range of application areas such as social network analysis, web search, product recommendation, and computational biology. Not surprisingly, the study of large, complex networks has attracted considerable attention from computer scientists, physicists, biologists, and social scientists. Networks are highly dynamic objects; they grow and change quickly over time through the additions of new vertices and edges, signifying the appearance of new interactions in the underlying structure. For example, a specific network analysis problem is link prediction, where the task is to predict the presence or absence of a link between certain pairs of vertices, based on observed links in other parts of the networks [15].

One of the most important issues in addressing such network analysis problems is the sheer size of the data sets; for example, Facebook has over 900 million monthly active users, and even relatively small networks may have millions of vertices and edges. One solution to deal with these large scale networks is dimensionality reduction, which aims to find more compact representations of data without much loss of information. Principal Component Analysis (PCA) and low-rank approximation

⋆ This work was started when Berkant Savas was a postdoctoral researcher in ICES, University of Texas at Austin.

by truncated Singular Value Decomposition (SVD) are well-known techniques for dimensionality reduction [14, 10]. ISOMAP [28] and locally linear embedding [24] are also widely used when we need to retain the non-linear property or manifold structure of the data.

However, these global dimensionality reduction techniques do not necessarily take into account *local structure* such as clusters in the network that are crucial for network analysis. Specifically, global techniques are likely to extract information from only the largest or a few dominant clusters, excluding information about smaller clusters. This is not desirable since different clusters usually have distinct meanings. We need to extract some information from every cluster regardless of its size to preserve important structural information of the original network in a low-dimensional representation. This is the motivation of a recently proposed method called *clustered low-rank approximation* [25], which reflects the clustering structure of the original network in the low-rank representation of the network. It extracts clusters, computes a low-rank approximation of each cluster, and then combines together the cluster approximations to approximate the entire network.

Unfortunately, the only available implementation of the clustered low-rank approximation is a sequential implementation, which precludes its use for processing large scale data sets for two reasons: (1) when the network size is huge, the network usually does not fit into the memory of a single machine, and (2) the running time can be substantial.

In this paper, we describe the first parallel implementation of clustered low-rank approximation, and show its application to link prediction on large scale social networks. It is a challenging problem to develop a parallel algorithm for the clustered low-rank approximation since it requires computing with very diverse data structures ranging from extremely sparse matrices to dense matrices. Experimental results show that our parallel implementation scales well on large distributed-memory machines; for example, on a Twitter graph, a standard data set in the social networks area with roughly 11 million vertices and 63 million edges, our implementation scales by a factor of 86 on 128 processes. The whole procedure, including link prediction, takes less than 2300 seconds on 128 processes. On a much larger Twitter graph with 41 million vertices and 1.2 billion edges, our current algorithm produces encouraging results with a scalability of 203 on 256 processes. In this case, the running time is about 4800 seconds.

The rest of this paper is organized as follows. In Section 2, we present the clustered low-rank approximation algorithm in detail, and introduce the link prediction problem. In Section 3, we describe our parallel algorithm. We present our experimental results in Section 4, and we briefly review some related work in Section 5. Finally, we state our conclusions in Section 6.

2 Preliminaries

In this section, we describe the clustered low-rank approximation method proposed in [25], and introduce the problem of link prediction in social network analysis. Throughout the paper, we use capital letters to represent matrices, lower-case bold letters to represent vectors, and lower-case italics to represent scalars. Note that the terms *graph* and *network* are used interchangeably.

2.1 Clustered Low-rank Approximation

A graph $G = (\mathcal{V}, \mathcal{E})$ is represented by a set of vertices $\mathcal{V} = \{1, \ldots, m\}$ and a set of edges $\mathcal{E} = \{e_{ij} | i, j \in \mathcal{V}\}$ where e_{ij} denotes an edge weight between vertices i and j. The corresponding adjacency matrix of G is represented by $A = [a_{ij}]$ such that $a_{ij} = e_{ij}$ if there is an edge between vertices i and j, and 0 otherwise. Note that A is an $m \times m$ matrix. For simplicity, we focus our discussion on undirected graphs, which implies that the adjacency matrix of the graph is symmetric.

One of the standard and very useful methods for dimensionality reduction is obtained by spectral or SVD analysis of the adjacency matrix A. For example, if the graph is undirected (A is symmetric), the rank-k spectral approximation of A can be computed by eigendecomposition as follows:

$$A \approx V \Lambda V^T, \tag{1}$$

where $V = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$, $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_k)$ is a $k \times k$ diagonal matrix, and $\lambda_1, \ldots, \lambda_k$ are the largest eigenvalues (in magnitude) of A, $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are the corresponding eigenvectors of A. One

benefit of the spectral approximation is that it gives a globally optimal low-rank approximation of A for a given rank. On the other hand, a drawback with spectral analysis and SVD is that they do not necessarily take into account local structures, such as clusters, of the network that are important for network analysis. These local structures (clusters) of the network are usually discovered by graph clustering which seeks to partition the graph into c disjoint clusters $\mathcal{V}_1, \ldots, \mathcal{V}_c$ such that $\bigcup_{i=1}^{c} \mathcal{V}_i = \mathcal{V}$. Suppose that A is an $m \times m$ adjacency matrix, and that we cluster the graph into c disjoint clusters. We use m_i to denote the number of vertices in the cluster i. By reordering vertices in order of their cluster affiliations, we can represent the $m \times m$ adjacency matrix A as follows:

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1c} \\ \vdots & \ddots & \vdots \\ A_{c1} & \cdots & A_{cc} \end{bmatrix}, \tag{2}$$

where each diagonal block A_{ii}, for $i = 1, \ldots, c$, is an $m_i \times m_i$ matrix that can be considered as a local adjacency matrix for cluster i. The off-diagonal blocks A_{ij} with $i \neq j$, represent the set of edges between vertices belonging to cluster i and cluster j. Note that A_{ij} is $m_i \times m_j$ matrix. Figure 1 shows the adjacency matrix of an arXiv network in the block form of (2). In the figure, a blue dot represents a non-zero entry of the matrix. Observe that the non-zeros (links) in the adjacency matrix are concentrated in the diagonal blocks A_{ii}, while the off-diagonal blocks are much more sparse.

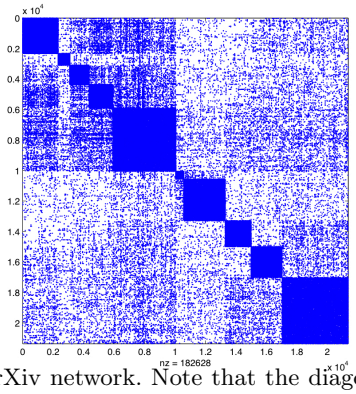

Fig. 1. Clustering structure of an arXiv network. Note that the diagonal blocks are much denser than the off-diagonal blocks.

In the clustered low-rank approximation framework, we first cluster a given network. Then, we independently compute a low-rank approximation of each cluster which corresponds to a diagonal block A_{ii}. With a symmetric matrix A as in (2), we can compute the best rank-k approximation of each A_{ii} as follows:

$$A_{ii} \approx V_i D_{ii} V_i^{\mathsf{T}}, \quad i = 1, \cdots, c, \tag{3}$$

where D_{ii} is a diagonal matrix with the k largest (in magnitude) eigenvalues of A_{ii}, and V_i is an orthogonal matrix with the corresponding eigenvectors.

Subsequently, the different cluster-wise approximations are combined together to obtain a low-rank approximation of the entire adjacency matrix. That is,

$$A \approx \begin{bmatrix} V_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & V_c \end{bmatrix} \begin{bmatrix} D_{11} & \cdots & D_{1c} \\ \vdots & \ddots & \vdots \\ D_{c1} & \cdots & D_{cc} \end{bmatrix} \begin{bmatrix} V_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & V_c \end{bmatrix}^{\mathsf{T}} \equiv \bar{V} \bar{D} \bar{V}^{\mathsf{T}}, \tag{4}$$

where $D_{ij} = V_i^{\mathsf{T}} A_{ij} V_j$, for $i, j = 1, \ldots, c$, which makes $\bar{D}$ optimal in the least squares sense.

With a rank-k approximation of each cluster A_{ii}, we can observe that the clustered low-rank approximation has rank ck. As a result, compared with a regular rank-k approximation of A, we

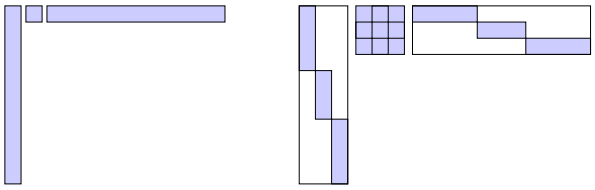

Fig. 2. Left panel shows an illustration of the regular spectral approximation $A \approx V \Lambda V^{\mathsf{T}}$. Right panel shows an illustration of the clustered low-rank approximation $A \approx \bar{V} \bar{D} \bar{V}^{\mathsf{T}}$ from (4). In both cases V and $\bar{V}$ are "long-thin" and they both use the same amount of memory as only the diagonal blocks of $\bar{V}$ are stored.

see that the rank in the clustered low-rank approximation is increased by a factor of c. However, a key observation is that $\bar{V}$ is a block diagonal matrix and uses exactly the same amount of memory as a regular rank-k approximation of A, as only the non-zero V_i blocks are stored while zero blocks of $\bar{V}$ are not stored. A pictorial representation of a regular rank-k approximation and the clustered low-rank approximation is given in Figure 2.

There are a number of benefits of clustered low rank approximation compared to spectral regular low-rank approximation: (1) the clustered low-rank approximation preserves important structural information of a network by extracting a certain amount of information from all of the clusters; (2) it has been shown that the clustered low-rank approximation achieves a lower relative error than the truncated SVD with the same amount of memory [25]; (3) it also has been shown that even a sequential implementation of clustered low rank approximation [25] is faster than state-of-the-art algorithms for low-rank matrix approximation [20]; (4) improved accuracy of clustered low-rank approximation contributes to improved performance of end tasks, e.g., prediction of new links in social networks [26] and group recommendation to community members [29].

2.2 Link Prediction in Social Networks

Link prediction [21] is one of the important tasks in social network analysis. Link prediction is the problem of predicting formation of new links in networks that evolve over time. This problem arises in applications such as friendship recommendation in social networks [26], affiliation recommendation [29], and prediction of author collaborations in scientific publications [23].

In social network analysis, the Katz measure [18] is a widely used proximity measure between the vertices (actors). In an undirected social network A, the Katz measure can be represented as a matrix function $Katz(A)$, where the (i,j)-th element represents the value of a proximity between actor i and actor j, as follows:

$$Katz(A) = \beta A + \beta^2 A^2 + \beta^3 A^3 + ... = \sum_{k=1}^{\infty} \beta^k A^k, \tag{5}$$

where $\beta < 1/\|A\|_2$ is a damping parameter.

Given a network, we sort all the pairs of vertices according to the Katz scores in descending order. By selecting top-k pairs which do not appear in the current network, we predict k links that are likely to be formed in the future. Note that this Katz computation is infeasible when the network size is very large, since it requires $O(m^3)$ time where m is the number of vertices in the network. However, the computation becomes feasible to approximate by leveraging the clustered low-rank approximation. Suppose that A is represented as (4) by using the clustered low-rank approximation. Then, the Katz measure can be approximated by:

$$Katz(A) \approx \hat{K} = \sum_{k=1}^{k_{\max}} \beta^k (\bar{V} \bar{D} \bar{V}^T)^k = \bar{V} (\sum_{k=1}^{k_{\max}} \beta^k \bar{D}^k) \bar{V}^T \equiv \bar{V} \bar{P} \bar{V}^T \tag{6}$$

3 Parallelization Strategy

In this section, we describe the parallelization strategy for each major phase of the clustered low-rank approximation algorithm: (i) graph clustering, (ii) approximation of diagonal blocks (clusters), and (iii) approximation of off-diagonal blocks (inter-cluster edges). We also describe how the parallel low-rank approximation algorithm can be used to solve a parallel link prediction problem on social networks.

On distributed memory machines, a graph is stored across different processes such that each process owns a subset of vertices and their adjacency lists. We use the term *local vertices* to designate the vertices each process owns.

3.1 Parallel Graph Clustering

Recall that for a given graph $G = (\mathcal{V}, \mathcal{E})$, graph clustering (also called graph partitioning) seeks to partition the graph into c disjoint clusters $\mathcal{V}_1, \ldots, \mathcal{V}_c$ such that $\bigcup_{i=1}^{c} \mathcal{V}_i = \mathcal{V}$. We use the term *clusters* and *partitions*, interchangeably.

Parallelization of graph clustering algorithms has long been recognized as a difficult problem. The state-of-the-art parallel library for graph clustering and partitioning is ParMetis [17], which is designed to deal with large scale graphs on distributed memory machines. However, ParMetis was designed for clustering and partitioning graphs that arise in computational science applications, and it does not perform well on social network graphs, which have a very different structure. For example, ParMetis could not cluster one of our data sets which has 40 million vertices and 1 billion edges due to lack of memory. Therefore, as one variation of the algorithm proposed in [30], we developed a custom parallel graph clustering algorithm which (i) scales well for social network graphs, and (ii) produces comparable quality clusters with ParMetis for graphs on which ParMetis is successful. We name our new parallel graph clustering algorithm PEK[4]. PEK consists of four phases: (1) extraction of a representative subgraph, (2) initial partitioning, (3) partitioning propagation and refinement, and (4) recursive partitioning.

Extraction of a Representative Subgraph. From a given graph, we first select vertices whose degrees are greater than a certain threshold. These vertices induce a representative subgraph of the original graph, which is constructed with these selected vertices and the edges between them. Note that we can typically designate a degree threshold so that a desired number of vertices is included in the subgraph.

This step is easy to parallelize. First of all, each process scans its local vertices to select vertices and then communicates with other processes to decide the location of the selected vertices. According to the location information, a subgraph is created and distributed across different processes.

Initial Partitioning. The extracted subgraph is clustered using ParMetis. The runtime of this initial partitioning step only takes a small fraction of the total runtime if the extracted subgraph is very small compared to the original graph. When a network follows a power-law degree distribution, which is a well-known property of social networks, a very small number of high-degree vertices cover a large portion of the edges in the entire network. So, we usually extract a small number of vertices from the original graph, which are likely to govern the overall structure of the entire network, and then cluster this small network using ParMetis.

Partitioning Propagation and Refinement. At this point, the vertices of the extracted subgraph have been assigned to clusters. These vertices are considered to be the "seeds" for clustering the entire graph. Starting from vertices of the extracted subgraph, we visit the rest of the vertices in the original graph in a breadth-first order. To reduce communication among processes, each process only considers its local vertices when doing a breadth-first traversal. When we visit a vertex, we

[4] The abbreviation PEK represents two key concepts of our Parallel graph clustering algorithm: Extraction of graph, and weighted Kernel k-means.

assign the vertex to some cluster by applying a weighted kernel k-means (WKKM) algorithm. We will explain the WKKM algorithm in detail below. Once we assign all the vertices of the original graph to some clusters, we refine the clustering using the WKKM algorithm repeatedly.

It has been shown that a general weighted kernel k-means objective is mathematically equivalent to a weighted graph clustering objective [13]. Therefore, we can optimize a weighted graph clustering objective by running the WKKM algorithm. At a high level, this algorithm computes the distance between a vertex and the centroid of each of the clusters, and assigns each vertex to its closest cluster.

To describe the WKKM algorithm in detail, we introduce some notation. Recall that for a given graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \ldots, m\}$ and $\mathcal{E} = \{e_{ij} | i, j \in \mathcal{V}\}$, the corresponding adjacency matrix of G is represented by $A = [a_{ij}]$ such that $a_{ij} = e_{ij}$, the edge weight between i and j, if there is an edge between i and j, and 0 otherwise. Now, let us define $links(\mathcal{V}_p, \mathcal{V}_p)$ to be the sum of the edge weights between vertices in $\mathcal{V}_p$ for $p = 1, ..., c$, i.e., $links(\mathcal{V}_p, \mathcal{V}_p) = \sum_{i \in \mathcal{V}_p, j \in \mathcal{V}_p} a_{ij}$. Similarly, $links(\hat{x}, \mathcal{V}_p)$ denotes the sum of the edge weights between a vertex $\hat{x}$ and the vertices in $\mathcal{V}_p$. Also, we define $degree(\mathcal{V}_p)$ to be the sum of the edge weights of vertices in $\mathcal{V}_p$, i.e., $degree(\mathcal{V}_p) = links(\mathcal{V}_p, \mathcal{V})$. Finally, we use $\hat{x}$ to denote a vertex, and $\hat{w}$ to denote the degree of the corresponding vertex.

There can be many variations of the WKKM algorithm when applying it to a graph clustering problem. In our experiments, we measure the distance between a vertex $\hat{x}$ and a cluster $\mathcal{V}_p$, denoted by $dist(\hat{x}, \mathcal{V}_p)$, using the following expressions (detailed explanation about how this distance measure is derived is stated in [30]):

$$dist(\hat{x}, \mathcal{V}_p) = \begin{cases} \dfrac{\hat{w} \cdot degree(\mathcal{V}_p)}{(degree(\mathcal{V}_p) - \hat{w})} \left(\dfrac{links(\mathcal{V}_p, \mathcal{V}_p)}{degree(\mathcal{V}_p)^2} - \dfrac{2links(\hat{x}, \mathcal{V}_p)}{\hat{w} \cdot degree(\mathcal{V}_p)} \right), & \text{if } \hat{x} \in \mathcal{V}_p, \\ \dfrac{\hat{w} \cdot degree(\mathcal{V}_p)}{(degree(\mathcal{V}_p) + \hat{w})} \left(\dfrac{links(\mathcal{V}_p, \mathcal{V}_p)}{degree(\mathcal{V}_p)^2} - \dfrac{2links(\hat{x}, \mathcal{V}_p)}{\hat{w} \cdot degree(\mathcal{V}_p)} \right), & \text{if } \hat{x} \notin \mathcal{V}_p. \end{cases} \tag{7}$$

Once we compute the distance between a vertex and the clusters, we assign the vertex to the closest cluster. If a vertex moves from its current cluster to another cluster, the centroids of the current cluster and the new cluster need to be updated immediately. However, since the cluster centroids are globally shared, the updates will serialize the algorithm. To avoid this serialization, we synchronize the cluster centroids less frequently. In our experiments, we synchronize the centroids of clusters once all of the processes finish considering their local vertices.

In summary, given the current cluster information, each process assigns its local vertices to their closest clusters. After this, the cluster information is updated. This procedure is repeated until the change in the WKKM objective value is sufficiently small or the maximum number of iterations is reached.

Recursive Partitioning. If we observe very large clusters, we can further partition the clusters by recursively applying the above procedures until all the clusters are small enough. To do the recursive partitioning on the large clusters, we need to extract the large clusters from the original graph. Usually, each extracted cluster is not necessarily a single connected component. Therefore, we first find all components in the extracted clusters. If the size of a component is larger than a certain threshold, we recursively partition it using the WKKM procedure. If the size of a component is near the threshold (*i.e.*, a moderate-sized component), we just leave it as a new cluster. Finally, we form new clusters by merging small components. At the end, each cluster contains a reasonably large number of vertices.

This recursive partitioning is required since subsequently each cluster is approximated using eigendecomposition of the corresponding submatrix (described in Section 3.2). If a cluster is too large, the memory consumption increases significantly in the eigendecomposition step. Therefore we partition each cluster until every cluster is small enough to be handled by a single process. In our experiments, we recursively partition the graph until each cluster contains less than 100,000 vertices.

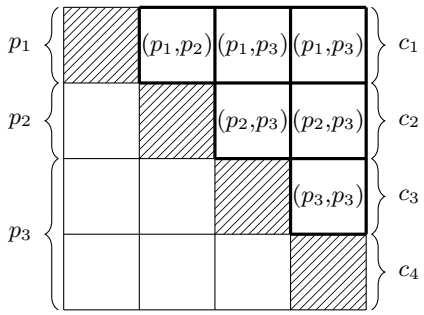

Fig. 3. An example where a pair (p_i, p_j) represents a job shared by processes p_i and p_j in approximation of off-diagonal blocks.

3.2 Approximation of Diagonal Blocks

The clustering step is followed by a reordering of the vertices so vertices in the same cluster are contiguously numbered. The adjacency matrix of the resulting graph G has a block structure as in (2). Each A_{ii} is the local adjacency matrix of cluster i, and off-diagonal blocks A_{ij} contain the edges between cluster i and cluster j. Then each process computes the rank-k eigendecomposition approximation for each of the diagonal matrices A_{ii} it owns according to (3); for example, $A_{11} \approx V_1 D_{11} V_1^\mathsf{T}$.

Since each process can only access a limited amount of memory and computing the eigendecomposition requires fairly large amount of memory, it is better to compromise the balance of the memory usage and the computation time across processes. Therefore, we assign clusters to processes using a static list scheduling approach. All the clusters are put into an ordered list where the priority of each cluster A_{ii} is determined by (the number of non-zero entries in A_{ii}) $\times\, k$. Each process is associated with a weight. The clusters are repeatedly extracted from the list and assigned to the process with the current minimum weight. Whenever a cluster is assigned to a process, its weight is increased by an amount equal to the cluster's priority. Since this computation is so small that every process can simultaneously compute the assignment. After the assignment, we redistribute the graph so that the vertices belonging to the same cluster are aggregated into the same process.

The matrices V_i in the low-rank approximation (see Figure 2) will, in general, be dense matrices. They are typically small enough so that they fit on any node of the distributed-memory machine, so we do not distribute individual V_i's across processes, reducing communication further.

3.3 Approximating Off-Diagonal Blocks

The approximations of the off-diagonal blocks A_{ij} is given by $A_{ij} \approx V_i D_{ij} V_j^\mathsf{T}$ where $D_{ij} = V_i^\mathsf{T} A_{ij} V_j$. Given that all V_i are computed in the previous step, what remains is to compute D_{ij} for $i, j = 1, 2, \ldots, c$ and $i \neq j$. Recall that all V_i are dense matrices and off-diagonal blocks A_{ij} are sparse matrices. It follows that matrix products of the type $V_i^\mathsf{T} A_{ij}$ or $A_{ij} V_j$ result in dense matrices. Consequently, computation of each D_{ij} involves two multiplications: one between a dense matrix and a sparse matrix, and the other between two dense matrices.

Since the graph G is undirected, we can exploit the symmetry of its adjacency matrix representation. In this case, we only need to compute D_{ij} or D_{ji}, as the other can be easily obtained with a transpose operation. We define job(i, j) as computing D_{ij} $(i < j)$, so the total number of jobs is $\frac{c(c-1)}{2}$. From Section 3.2, it is easy to see that each process contains A_{ij} and V_i for the clusters it owns. In order to compute D_{ij}, we require the matrices V_i, A_{ij}, and V_j. In the easy case, where a process which owns A_{ij} and V_i, also owns the matrix V_j as well, D_{ij} can be computed without any communication. Otherwise, some communication is required between the process owning V_i (and A_{ij}) and the process owning V_j. Therefore, the jobs can be categorized as the ones which can be performed independently and those which need some communication between processes. For the

former type, all of V_i, V_j, A_{ij}, and A_{ji} are located in the same process and we call them *private jobs* of the owner process. The latter is a more complicated scenario in which A_{ij} and V_i are co-located in one process while A_{ji} and V_j are co-located in another process. Therefore, either process can execute job(i, j) by fetching the required matrices from the other process. We call these *shared jobs* between the two processes. Since jobs take different amounts of time, a challenging problem that arises here is to evenly divide the shared jobs between every pair of processes, with the aim of minimizing communication and achieving an ideal load balance.

Figure 3 is an example to illustrate this problem. Figure 3 shows a matrix representing the block view of A. Each entry of this matrix represents A_{ij}. In this example, graph A is partitioned into four clusters. Each row of the matrix can be regarded as a cluster including edges that connect to other clusters, denoted as c_i. There are three processes $p_1, p_2,$ and p_3. Process p_1 owns cluster c_1, p_2 owns cluster c_2, and p_3 owns clusters c_3 and c_4. The pair (p_i,p_j) inside each entry in Figure 3 represents that the job (approximation of this entry) can be computed by either process i or process j.

We build a dynamic load balancing framework for performing the jobs. For each pair of processes, we create a *job queue* storing the shared jobs between them. A job queue is a priority queue where jobs are ordered by the amount of work. Each process first works on its private jobs. Once its private jobs are finished, the process starts to ask for jobs from a master process p_{master} dedicated to scheduling jobs. Whenever a process p asks for a new job, p_{master} goes through all job lists that involve p, picks the job queue with the most jobs, extracts the largest job in that job list, and then hands it to the process p. When the process p receives the job, it fetches the corresponding V_j from the process owning V_j using RDMA (Remote Direct Memory Access).

Once this stage is complete, we have computed all necessary factors to approximate A, i.e., $A \approx \hat{A} = \bar{V}\bar{D}\bar{V}^{\mathsf{T}}$, as in (4). In particular, the approximation of each block A_{ij} in (2) is given by $A_{ij} \approx \hat{A}_{ij} = V_i D_{ij} V_j^{\mathsf{T}}$, for $i, j = 1, \ldots, c$.

3.4 Parallel Computation of the Katz Measure and Link Prediction

Now, we describe how we compute the Katz measure in parallel, and perform parallel link prediction. Recall (6). The computation of the term $\sum_{k=1}^{k_{\max}} \beta^k \bar{D}^k$ requires a distributed dense matrix multiplication since $\bar{D}$ is a dense matrix. Parallel dense matrix multiplication has been fairly well understood and there are several efficient libraries available. We use the Elemental Matrix class library [2] to perform this step and we set $k_{\max} = 6$ in the experiments. In terms of the block-wise view as in (4), we can rewrite (6) as follows:

$$\hat{K} = \bar{V}\bar{P}\bar{V}^{\mathsf{T}} \equiv \begin{bmatrix} V_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & V_c \end{bmatrix} \begin{bmatrix} P_{11} & \cdots & P_{1c} \\ \vdots & \ddots & \vdots \\ P_{c1} & \cdots & P_{cc} \end{bmatrix} \begin{bmatrix} V_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & V_c \end{bmatrix}^{\mathsf{T}}, \tag{8}$$

Then, $\hat{K}_{ij}$ is computed as follows: $\hat{K}_{ij} = V_i P_{ij} V_j^{\mathsf{T}}$, for $i, j = 1, \ldots, c$. We distribute P_{ij} to the same process as A_{ij}. Due to the symmetry of $\bar{P}$, computing $\hat{K}$ is very similar to the approximation of off-diagonal matrices in Section 3.3, so we adopt a similar parallelization stratgy. Since $\hat{K}$ is a $m \times m$ matrix, it is infeasible to compute the whole matrix if m is very large. Therefore, we only compute a subset of $\hat{K}$ and predict links based on the sampled subset. The details of our sampling method is stated in Section 4.

4 Experimental Results

In this section, we present and analyze experimental results on a large-scale parallel platform at the Texas Advanced Computing Center (TACC), Ranger [5]. Ranger has 3,936 nodes, and each node is equipped with a 16-core AMD Opteron 2.2GHz CPU and 32GB memory. Ranger uses InfiniBand networks with 5GB/s point-to-point bandwidth. The MPI library on Ranger is Open MPI 1.3. Our implementation is written in C++. We use ARPACK++ [1] for the eigendecompositions of diagonal blocks, GotoBLAS 1.30 [3] for the dense matrix multiplications involving the off-diagonal blocks, and Elemental Matrix class library [2] for the dense matrix multiplications for the Katz measure.

4.1 Data Sets

We use three different social graphs which are summarized in Table 1. LiveJournal is a free online community with almost 10 million members, and it allows members to select other members as their friends. Twitter is an online social networking website where members *follow* other members they are interested in. In our experiments, we extract the largest connected component from each of the network. Originally, each of these networks were directed networks. So, we transformed them into undirected graphs by adding additional edges. In the experiments, the degree thresholds for PEK are 42 (Soc-LiveJournal), 200 (Twitter-10M) and 2500 (Twitter-40M). The number of vertices of the extracted subgraph is less than 5% of the original graph in all cases.

Table 1. Detailed information of the graph data sets.

Data set	#Vertices	#Edges	Description
soc-LiveJournal	3,828,682	39,870,459	LiveJournal on social network [6].
Twitter-10M	11,316,799	63,555,738	Crawled Twitter graph from [7].
Twitter-40M	41,652,230	1,202,513,046	Crawled Twitter graph from [19].

4.2 Parallel Performance Evaluation

We use one process as the scheduling server and the other processes as the workers in the phases of computing the approximation of off-diagonal blocks and link prediction. In other phases, the scheduling process stays idle and does not participate in computation in any phase. The graphs are initially randomly distributed among all the processes other than the scheduling process.

Figure 4 shows the performance of our parallel implementation of the clustered low-rank approximation, on the soc-LiveJournal, Twitter-10M and Twitter-40M graphs. All these social network graphs are too large to be processed in a single node of Ranger. Therefore, we run each graph on the smallest number of nodes on which the program finishes successfully, and then measure the performance as the number of nodes increases. We use only one MPI process on each node to enable us to measure performance without interference from other processes in the same node. From Figure 4, we see that our implementation scales very well on the three different sizes of real social graphs. A speedup of 68 is achieved on 64 processes for soc-LiveJournal, a speedup of 86 is achieved on 128 processes for Twitter-10M and 203 on 256 processes for Twitter-40M. The super-linear speedup is mainly due to the cache effects of matrix multiplication. For soc-LiveJournal and Twitter-10M, performance levels off after 64 nodes, but this is mainly because the problem size is relatively small compared to the number of processes. This is clear from the performance of Twitter-40M: it consistently scales up to 256 processes.

Figure 5 shows how much time is spent on each of the major phases of the algorithm: (i) partitioning, (ii) computing diagonal blocks, (iii) computing the off-diagonal blocks D_{ij}'s, and (iv) link prediction. We divide the link prediction phase into two steps: (a) computing the $\bar{P}$ matrix in (6) which requires dense matrix multiplication, and (b) computing $\hat{K}_{ij} = V_i P_{ij} V_j^{\mathsf{T}}$ in (8). We label (a) as MatrixPower, and label (b) as ScoreComputation in Figure 5. The MatrixPower dominates the running time since this step involves large dense matrix multiplication (square matrix, each dimension is $c \times k$ where c is the number of clusters and k is the number of eigenvalues). To make the runtime of other phases visible in the graph, we show MatrixPower time in a separate figure from other phases.

From Figure 5, we see that most of the phases scale well with increasing numbers of processes, especially with our new parallel graph partitioning algorithm. For soc-LiveJournal and Twitter-10M, the runtime of the diagonal phase does not decrease beyond 32 processes. There are two reasons: first, the clustering algorithm is not deterministic, so running with different number of processes may cluster the graph differently. Cluster sizes will affect the runtime significantly since the complexity of eigendecomposition in the diagonal phase does not increase linearly with cluster size. Second, since the number of eigendecompositions is equal to the number of clusters, the average number of clusters

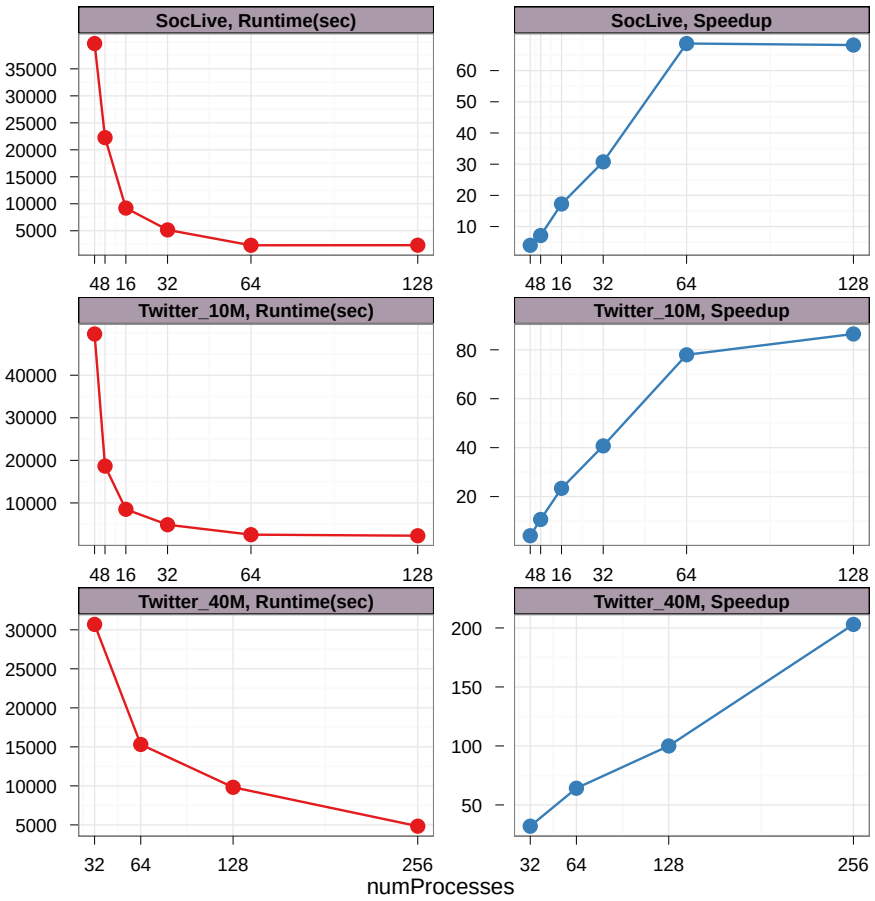

Fig. 4. Runtime and Speedup for soc-LiveJournal, Twitter-10M, and Twitter-40M graphs, where $c = 500$ and $k = 100$ for soc-LiveJournal and Twitter-10M, and $c = 1000$ and $k = 100$ for Twitter-40M.

assigned to each process will become smaller when the number of processes becomes large. So, the space available for load balancing among processes to hide unbalanced clustering effects will be less. This problem may be alleviated by dynamic load balancing which we will explore in the future. On the other hand, in the MatrixPower step, the Elemental library [2] we use for parallel dense matrix multiplication does not scale well after 64 processes for small matrix sizes ($50,000 \times 50,000$ for soc-LiveJournal and Twitter-10M), while it can scale consistently to 256 processes for larger sizes ($100,000 \times 100,000$ for Twitter-40M). This phase dominates the total runtime, and it is the main reason why the running time of soc-LiveJounrnal is not improved after 128 nodes.

Load Balancing. Figure 6 shows the load balancing results of three phases: diagonal, off-diagonal and link prediction. The red and green bars denote the maximum and minimum time processes take, respectively. The dynamic load balancing framework is effective in most of the cases for approximating off-diagonal blocks and link prediction. For small number of processes, the diagonal phase is also balanced. When the number of processes increases, the load among processes starts to become unbalanced. As mentioned before, this is mainly due to the very unbalanced partitions.

4.3 Evaluation of Clustering Algorithm

Figure 7 compares our clustering algorithm (PEK) with ParMetis on soc-LiveJournal and Twitter-10M, using two measures: (i) the quality of the partition, and (ii) the running time of the algorithm.

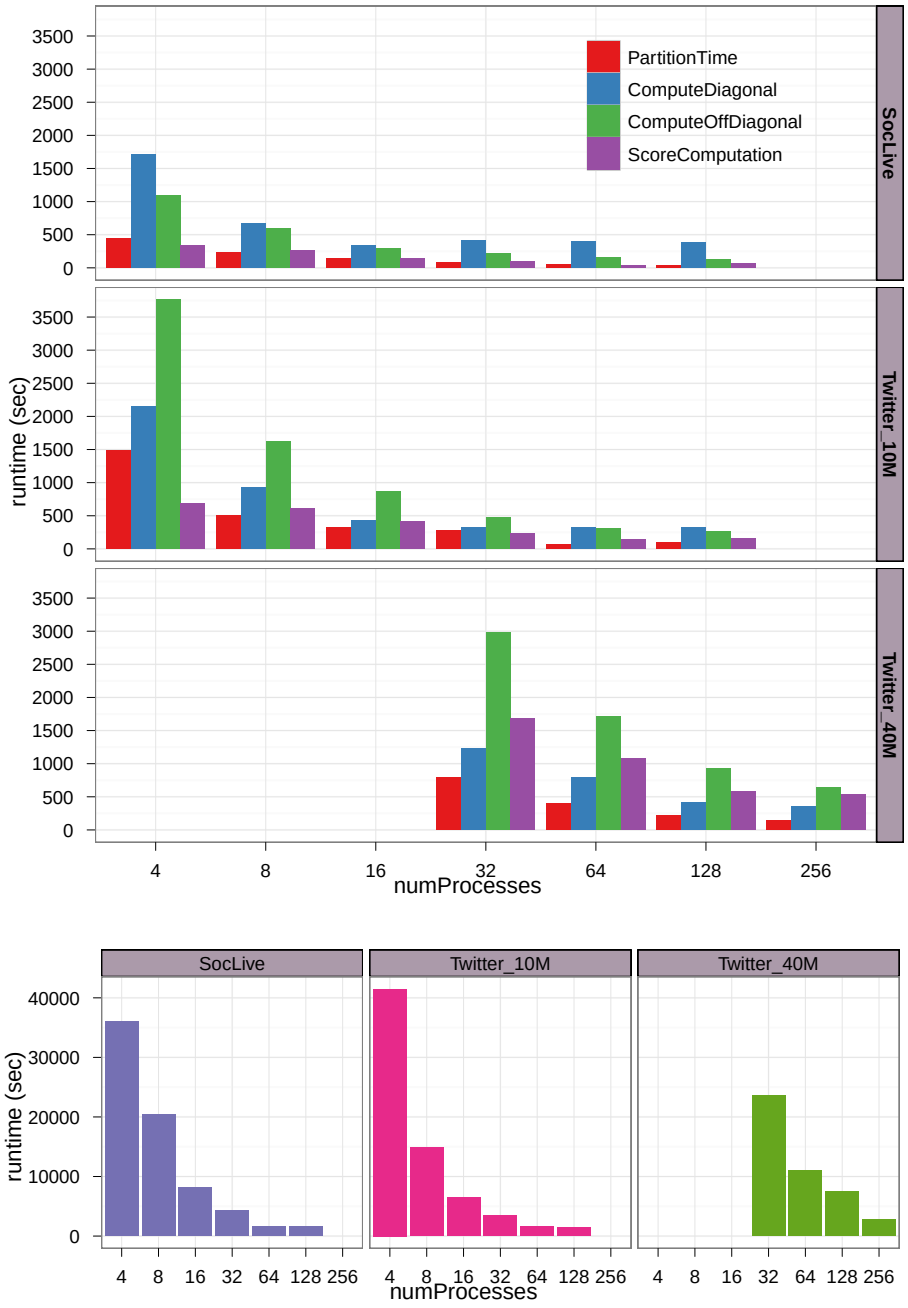

Fig. 5. Time spent in each phase for soc-LiveJournal, Twitter-10M, and Twitter-40M graphs, where $c = 500$ and $k = 100$ for soc-LiveJournal and Twitter-10M, and $c = 1000$ and $k = 100$ for Twitter-40M. The bottom figure shows the time for MatrixPower step. The MatrixPower dominates the running time since this step involves large dense matrix multiplication (square matrix, each dimension is $c \times k$ where c is the number of clusters and k is the number of eigenvalues). To make the runtime of other phases visible in the graph, we show MatrixPower time in a separate figure from other phases.

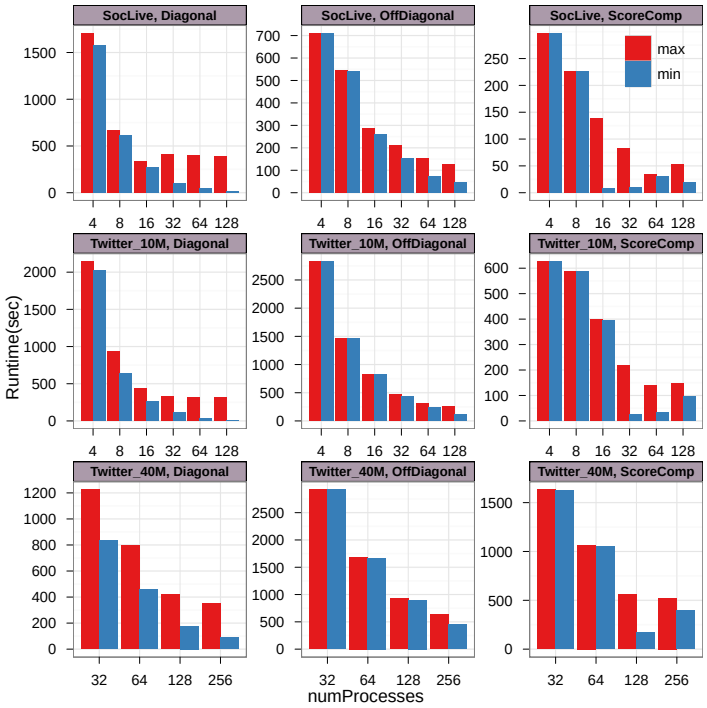

Fig. 6. Load balancing statistics in diagonal, off-diagonal, and link prediction phases for soc-LiveJournal, Twitter-10M, and Twitter-40M graphs, where $c = 500$ and $k = 100$ for soc-LiveJournal and Twitter-10M, and $c = 1000$ and $k = 100$ for Twitter-40M.

ParMetis fails to cluster Twitter-40M graph on Ranger because the memory at each node is not enough. To evaluate the quality of clusters, we use two standard measures: the normalized cut measure and the cut-size measure. These measures are defined as:

$$\text{NormCut} = \sum_{k=1}^{c} \frac{links(\mathcal{V}_k, \mathcal{V}\backslash\mathcal{V}_k)}{degree(\mathcal{V}_k)}, \text{Cut-Size} = \sum_{k=1}^{c} links(\mathcal{V}_k, \mathcal{V}\backslash\mathcal{V}_k). \tag{9}$$

where c is the number of clusters, A is the adjacency matrix of a graph $G = (\mathcal{V}, \mathcal{E})$, $links(\mathcal{V}_k, \mathcal{V}\backslash\mathcal{V}_k) = \sum_{i\in\mathcal{V}_k, j\in\{\mathcal{V}\backslash\mathcal{V}_k\}} a_{ij}$, and $degree(\mathcal{V}_k) = links(\mathcal{V}_k, \mathcal{V})$ for $k = 1, 2, ..., c$. By definition, the normalized cut is upper-bounded by the number of clusters. Lower normalized cut value indicates better quality of clusters. In Figure 7, we divide the normalized cut by the total number of clusters since PEK probably makes more clusters than the designated number of clusters due to its recursive partitioning phase. We see that PEK performs a little better than ParMetis on both of soc-LiveJournal and Twitter-10M in terms of the normalized cut. We also divide the cut-size by the total number of clusters, and present the results in Figure 7. Note that lower cut-size indicates better quality of clusters. We can see that the cluster quality of PEK and ParMetis are comparable in terms of the cut-size.

We see that PEK is much faster than ParMetis. For soc-LiveJournal, PEK is two times faster than ParMetis on 128 processes. For Twitter-10M, PEK is about seven times faster than ParMetis on 128 processes. Overall, PEK achieves similar quality as ParMetis but can scale better than ParMetis to larger number of processes and larger graphs.

4.4 Evaluation of Link Prediction

We perform our experiments as follows. Given a network $G = (V, E)$, a network $G' = (V, E')$ is obtained by randomly removing 30% of the links of G. We call the removed links *test links*. Then, we compute the Katz scores on the network G'. Since the networks we use are very large, we cannot compute the Katz scores for all the vertex pairs. So, we sample a set of vertex pairs from G',

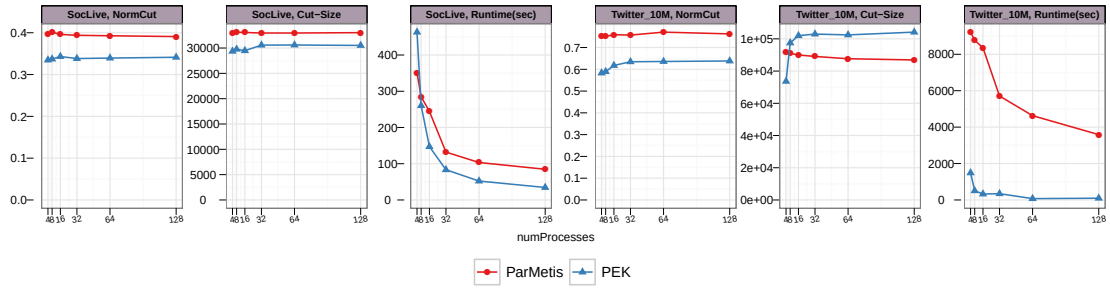

Fig. 7. Comparison between our clustering algorithm (PEK) with ParMetis on soc-LiveJournal and Twitter-10M. The number of clusters is 500. Since our implementation needs one scheduling server for later phases, we leave one process to handle that for both clustering algorithms in the experiments.

Graph	Top-k	Rank	Precision	Recall(UB)	Graph	Top-k	Rank	Precision	Recall(UB)
SocLive	10	c500r50	100	0.002(0.002)	Twitter-10M	10	c500r50	0	0(0.39)
		c500r100	100	0.002(0.002)			c500r100	100	0.389(0.389)
		c500r200	0	0(0.002)			c500r200	100	0.431(0.431)
	100	c500r50	100	0.024(0.024)		100	c500r50	10	0.431(4.310)
		c500r100	90	0.019(0.021)			c500r100	30	1.167(3.891)
		c500r200	30	0.006(0.021)			c500r200	10	0.431(4.310)
	1000	c500r50	98	0.234(0.239)		1000	c500r50	3	1.293(43.103)
		c500r100	46	0.096(0.209)			c500r100	10	3.891(38.910)
		c500r200	40	0.083(0.208)			c500r200	6	2.586(43.103)

Table 2. Link prediction evaluation on soc-LiveJournal and Twitter-10M. We compare the precision and recall with different ranks of approximations. The graph is always partitioned into 500 clusters. Three different rows represent different ranks of each cluster, which results in different rank approximations of the whole graph. UB is the upper bound for the recall. Note the upper bound differs in the experiments since we use a new sample set for each case.

and compute Katz scores on these sampled links. Let S denote the set of these sampled links. We randomly sample a subset of the links which are incident to vertices that are in the same cluster and whose degrees are larger than some threshold. The justification for this sampling method is as follows: new links are more likely to be formed within the same cluster and are more likely to be formed between vertices whose degrees are larger than a certain threshold.

Let R denote the set of top-k scoring links (top-k recommended links). Then, we evaluate our link prediction results by computing *precision* and *recall* which are defined as follows:

$$\text{Precision} = \frac{\text{number of correctly predicted links}}{\text{top-}k\text{ recommendations}} = \frac{|(E - E') \bigcap R|}{|R|}, \tag{10}$$

$$\text{Recall} = \frac{\text{number of correctly predicted links}}{\text{number of overlapped links between test and sampled links}} = \frac{|(E - E') \bigcap R|}{|(E - E') \bigcap S|}. \tag{11}$$

By definition, the upper bound of the recall measure is obtained by setting the numerator as $|R|$ (i.e., k). Higher precision and recall indicate better performance. Table 2 shows precision and recall for the soc-LiveJournal and Twitter-10M graphs with different ranks of approximations. We partition each graph into 500 clusters. The only difference among the three rows is the rank of each cluster. For soc-LiveJournal, we achieve 100% precision for predicting the top-10 and top-100 links and 98% precision for predicting the top-1000 links using rank 500×50 (500 indicates the number of clusters, 50 indicates the rank of each cluster). The recall is also very close to the upper bound. For Twitter-10M graph, precision and recall measures are not as good as for soc-LiveJournal. A contributing factor to this is that the overlap between $E - E'$ and S for the Twitter-10M graph is very small, around 2600 overlapping links with a sample size of 70 millions. (The overlap for soc-LiveJournal is around 500,000.) Even on this setting, we can see that the precision for top-10 is still

100%, and top-100 is 30% for rank 500×100. This reflects the effectiveness of clustered low-rank approximation approach.

5 Related Work

The study of parallel graph partitioning, which is a key component of parallel clustered low-rank approximation, has a long history. The most commonly used library is ParMetis [17]. However, ParMetis is not suitable for social networks because ParMetis utilizes a multilevel coarsening approach which is not effective in social networks. This multilevel coarsening is primarily designed for graphs in scientific computing (e.g. finite element meshes) [8, 27]. In order to overcome this problem, we develop PEK which is described in Section 3. While PEK is closely related to PGEM which is presented in [30], PEK is a custom clustering algorithm for clustered low-rank approximation framework. PEK includes a recursive partitioning step which allows us to proceed to the next phase of clustering phase. Furthermore, PEK utilizes ParMetis to cluster an extracted graph while PGEM uses weighted kernel k-means.

Cong et al. [12] studied the problem of parallel connected components implemented in UPC for distributed-memory systems. They started from a PRAM-based algorithm and applied several optimizations for sparse graphs. We implemented similar algorithm using MPI in our clustering algorithm. It takes very small fraction of the total clustering time so we do not report it.

Low-rank approximation has been applied to the task of link prediction and has been shown to be successful in practice [22]. Parallel eigendecomposition for dense matrices on multiple machines has been well studied in, e.g., [11, 9]. But they are not suitable for large social networks because the adjacency matrices for them are too large to be represented as dense matrices. Recently, [4, 16] studied the problem of large scale sparse eigensovler based on Hadoop.

Yoo et al. [31] studied level-synchronized breadth-first search on the Blue-Gene machine. They performed 2D partitions. By optimizing message buffer size and utilizing the processor topology, they achieved scalability of $\sqrt[2]{p}$ with p processors on a very large graph. Their work indicates that exploring processor topology information may be an important aspect for efficiency that we have not explored in this paper.

6 Conclusions

In this paper, we present the first parallel implementation of clustered low-rank approximation. We conduct experiments on distributed-memory machines, and the experimental results show that this parallel implementation is effective in processing large-scale social network graphs with tens of millions of vertices and hundreds of millions of edges. In particular, our parallel implementation scales well according to the number of processes.

Our parallel implementation of clustered low rank approximation provides a critical routine that is a key enabler for efficient analyses of social network graphs. Presently, such analyses are performed in a brute-force manner on the entire graph by using parallel processing in large datacenters; in contrast, low rank approximations of these graphs enable analyses to be performed more efficiently on a smaller graph that distills the essence of the original graph. However, most current low rank approximation techniques compute a global approximation of the graph, and ignore local structure, such as clusters, that must be preserved in the low rank approximation for accurate analysis. Fortunately, the clustered low rank approximation permits the computation of a structure-preserving low rank approximation. Our parallel implementation of this algorithm enables the full power of clustered low rank approximation to be brought to bear on huge social networks for the first time.

Acknowledgments: This research was supported by NSF grants CCF-1117055 and CCF-0916309.

References

1. ARPACK++. `http://www.ime.unicamp.br/~chico/arpack++/`.

2. Elemental. http://elemental.googlecode.com/hg/doc/build/html/core/matrix.html.
3. GotoBLAS. http://www.tacc.utexas.edu/tacc-projects/gotoblas2/.
4. Mahout. http://lucene.apache.org/mahout/.
5. Ranger. http://services.tacc.utexas.edu/index.php/ranger-user-guide.
6. SNAP - Stanford Network Analysis Package. http://snap.stanford.edu/snap/.
7. Social Computing Data Repository. http://socialcomputing.asu.edu/datasets/Twitter.
8. A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. IPDPS, 2006.
9. P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, and Y.-J. J. Wu. Plapack: parallel linear algebra package design overview. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, Supercomputing '97, pages 1–16. ACM, 1997.
10. C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
11. L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
12. G. Cong, G. Almasi, and V. Saraswat. Fast pgas connected components algorithms. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, 2009.
13. I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(11):1944–1957, 2007.
14. G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3 edition, 1996.
15. Z. Huang. Link prediction based on graph topology: The predictive value of the generalized clustering coefficient. In *Workshop on Link Analysis (KDD)*, 2006.
16. U. Kang, B. Meeder, and C. Faloutsos. Spectral analysis for billion-scale graphs: discoveries and implementation. In *Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining - Volume Part II*, PAKDD'11, pages 13–25, Berlin, Heidelberg, 2011. Springer-Verlag.
17. G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proceedings of SIAM International Conference on Parallel Processing for Scientific Computing*, 1997.
18. L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18:39–43, 1953.
19. H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, New York, NY, USA, 2010. ACM.
20. R. Lehoucq, D. Sorensen, and C. Yang. *Arpack Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, 1998.
21. D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, 2007.
22. D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7):1019–1031, May 2007.
23. Z. Lu, B. Savas, W. Tang, and I. S. Dhillon. Link prediction using multiple sources of information. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 923–928, 2010.
24. S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, 2000.
25. B. Savas and I. S. Dhillon. Clustered low rank approximation of graphs in information science applications. In *SIAM Data Mining Conference*, pages 164–175, 2011.
26. H. H. Song, B. Savas, T. W. Cho, V. Dave, Z. Lu, I. S. Dhillon, Y. Zhang, and L. Qiu. Clustered embedding of massive social networks. In *SIGMETRICS*, 2012.
27. X. Sui, D. Nguyen, M. Burtscher, and K. Pingali. Parallel graph partitioning on multicore architectures. In *LCPC*, pages 246–260, Berlin, Heidelberg, 2011. Springer-Verlag.
28. J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, December 2000.
29. V. Vasuki, N. Natarajan, Z. Lu, B. Savas, and I. S. Dhillon. Scalable affiliation recommendation using auxiliary networks. *ACM Transactions on Intelligent Systems and Technology*, 3:3:1–3:20, October 2011.
30. J. Whang, X. Sui, and I. Dhillon. Scalable and memory-efficient clustering of large-scale social networks. In *Proceedings of the IEEE International Conference on Data Mining*, 2012.
31. A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 25–43, 2005.