Final Report

15–618 (Fall '16)

Anant Subramanian
assubram@andrew.cmu.edu

Feroze Naina
fdheenmo@andrew.cmu.edu

HPSCG: High Performance Spectral Graph Clustering
An MPI + CUDA Parallel Library

1 Introduction

We designed a high performance library for graph clustering that exploits hybrid MPI + CUDA parallelism to perform distributed GPU clustering on large graph datasets of well over 1 million vertices. This document highlights the design choices involved in creating the library, the interface to the library, and experimental results that show the performance advantage gained in using the library.

1.1 The Problem/Opportunity

Given an undirected similarity graph $G = (V, E, W)$, a clustering of the graph is an assignment of vertices $v_i, \forall i \in \{1, 2, \cdots, |V|\}$ to clusters $c_i, \forall i \in \{1, 2, \cdots, C\}$ such that a particular *cost metric* on the similarities $W(v_i, v_j)$ over the clusters c_i is minimized.

A *normalized cut* of a graph is a partition of the graph into two partitions A and B such that the sum of the weights of the edges connecting vertices in A to those in B is the minimum of all possible partitions *and* the size of A and B are similar. Mathematically,

$$\text{Normalized Cut}\,(A, B) = \sum_{i \in A, j \in B} w_{ij} \left(\frac{1}{vol(A)} + \frac{1}{vol(B)} \right)$$

where

$$vol(A) = \sum_{i \in A} d_i$$

A *ratio cut* of a graph is similar to a normalized cut, except that the sizes of the subgraphs are compared using the number of vertices, instead of the number of edges (sum of degrees), as is done in the case of a normalized cut. Mathematically,

$$\text{Ratio Cut}\,(A, B) = \sum_{i \in A, j \in B} w_{ij} \left(\frac{1}{|A|} + \frac{1}{|B|} \right)$$

The *discrete* versions of both these problems, i.e., when a vertex is assigned to only one of the subgraphs, is NP-hard in nature. Spectral clustering methods attempt to relax these constraints,

and solve the *continuous* versions of these problems (vertices are affiliated with the partitions to some extent $\in [0, 1]$) using Graph Laplacians (one may refer to [1, 2, 3] for more details). In essence, the problem reduces to that of constructing the graph Laplacian (or the normalized graph Laplacian):

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \qquad (\mathbf{D} = \text{Diagonal degree matrix, } \mathbf{A} = \text{Affinity matrix})$$
$$\mathbf{L}^{norm} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$$

and computing the two (or more) partitions using the eigenvectors corresponding to one of the two Laplacian matrices. The eigenvectors of the unnormalized Laplacian help us solve the continuous version of the ratio-cut problem, and those of the normalized Laplacian help solve the continuous normalized cut problem.

One popular method for computing the eigenvectors of a matrix is using the Lanczos [4] algorithm to first reduce the size of the matrix, work on finding the eigenvectors of the reduced size matrix, and then recover the eigenvectors of the original matrix using the eigenvectors of the approximation matrix. All of these operations involve iterative matrix-vector and vector-vector multiplications. Therefore, from an engineering standpoint, spectral graph clustering algorithms reduce to steps of matrix operations.

Once the eigenvectors of the Laplacian matrix have been obtained, various techniques can be used to extract clusters from these eigenvectors. The simplest of these is to just look for ”spikes” of similar values in the eigenvectors. These spikes correspond to clusters in the graph (Figure 1 shows this for one of the eigenvectors for a graph). More sophisticated techniques such as K-Means clustering may also be used to obtain clusters from these eigenvectors.

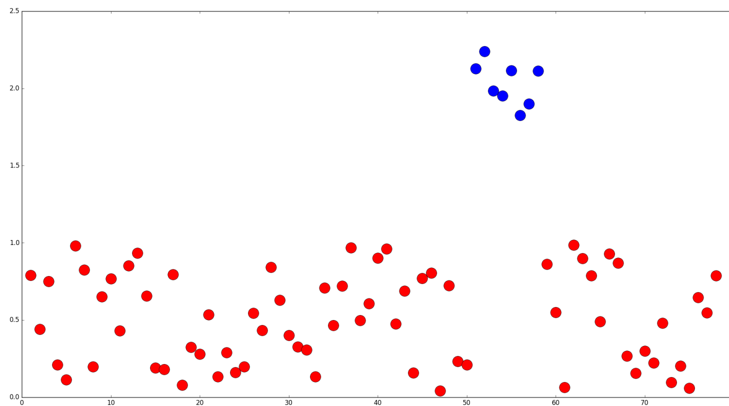

Figure 1: An eigenvector corresponding to one of the clusters in the graph. Notice the blue spike corresponding to the cluster.

Our contributions are the following:

1. We have designed a C++ library that parallelizes all the intermediate operations involved in computing the clusters of a graph using spectral methods.
2. The library provides methods to efficiently construct the distributed representations of sparse graphs, and their corresponding Laplacians, from input files.
3. The library exposes methods to run a hybrid MPI+CUDA parallel version of the Lanczos algorithm on these sparse graph Laplacians to construct the reduced size approximation matrices.

4. A final routine is provided that recovers the eigenvectors of the original graph from the eigenvectors of the reduced size approximation matrix.

We begin by discussing related work in this area. Following this, we explain the parallel algorithms used in our library in detail, provide details about the experimental setup that can be used to reproduce the results, and describe our results. We conclude by discussing future work.

1.2 Related Work

There are currently two classes of free libraries that perform end-to-end spectral clustering of input graphs: numerical computing libraries and large scale I/O libraries.

The first class involve computing solutions where CPU performance is the limiting factor, but the graphs typically tend to fit in the main memory (memories) of a single machine (cluster of machines). We test the quality and performance of our library algorithms against one of the most popular libraries in this class, ARPACK [5] (used in conjunction with scipy [6]).

The second class of libraries (like MLlib in Apache Spark [7]) are solutions to the problem where the graphs are so large that they would not fit in the main memories of a cluster of reasonable size. Since these solutions are I/O bound, the methods are not entirely targeted at high performance computing, and so we do not compare against these class of libraries.

A CUDA-only parallel implementation of the Lanczos algorithm alone was designed by students in Spring 2015 [8]. It was focused on implementing native CUDA subroutines for performing sparse matrix operations and comparing the performance against the CUDA cuSparse library.

To the best of our knowledge, this is the first hybrid MPI+CUDA parallel library that performs end-to-end spectral clustering, as well as the first effort in the direction of combining MPI and CUDA parallelism to perform Lanczos tri-diagonalization, an intermediate step in the spectral clustering approach followed by many libraries.

1.3 Outline of Our Approach

Since GPUs are excellent candidates for speeding up matrix operations by performing independent computations in parallel, we use them to speed up steps of the spectral clustering algorithm. Our primary focus was on discovering methods to distribute and perform these computations on a cluster of nodes using MPI, while continuing to exploit the parallelism offered by the GPUs on each of these nodes. Figure 2 illustrates our parallelization strategy.

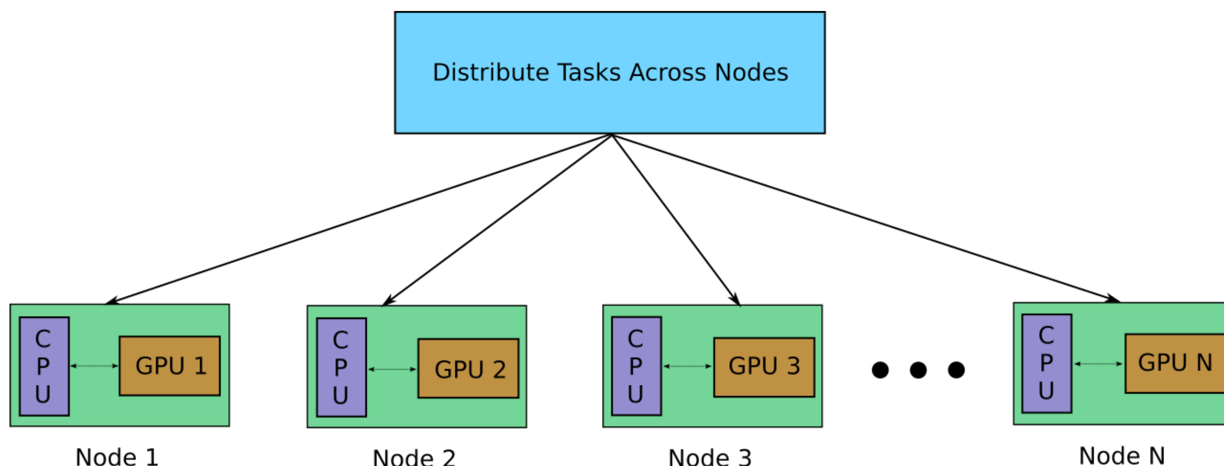

Figure 2: The hybrid parallelization strategy that is used by our algorithm.

2 Our Approach

Our library takes as input an edge list file (in network storage, accessible from all nodes) that contains the graph to be clustered, runs the algorithm in parallel on the cluster using MPI and CUDA, and outputs the eigenvectors of the graph to the desired output location. We discuss each of the steps in detail below.

2.1 The Algorithm

The working of the library (after abstracting out parallelization details), is depicted in the following flowchart:

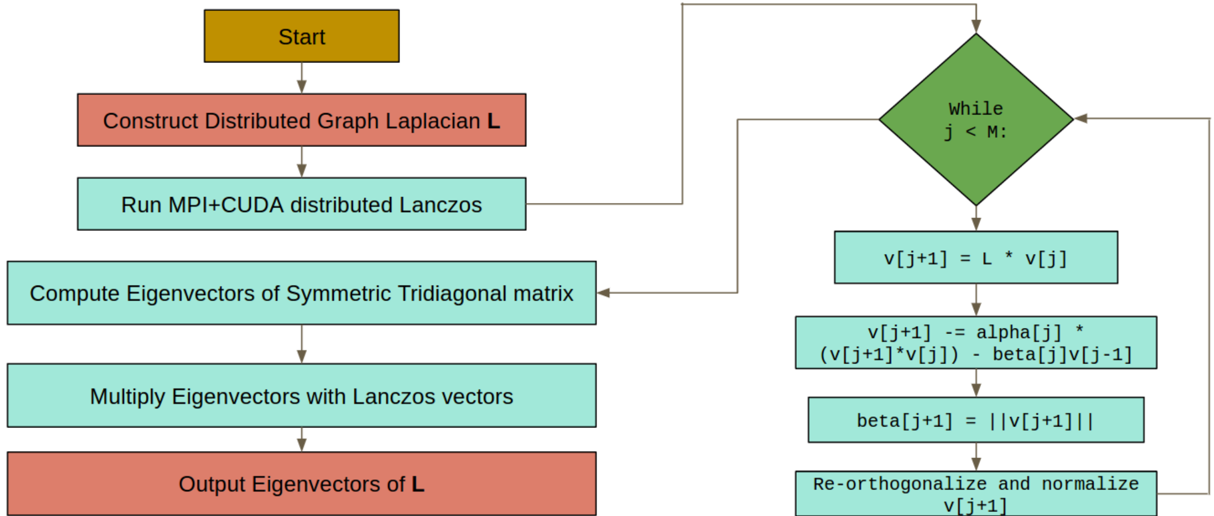

Figure 3: Algorithm

In our approach,

- Each cluster node loads a portion of the graph, and constructs the unnormalized Laplacian L in parallel. Note that we can efficiently scale to larger graphs as there is **no** synchronization overhead involved in the construction step. We use properties of the graph to infer the necessary Laplacian values in parallel. The module provided in the library currently supports **CSR** and **CSC** representations of these sparse graphs. Each has its advantages, which are discussed in the Lanczos algorithm section.
- Next, we call the distributed MPI + CUDA hybrid parallel Lanczos submodule of the library to reduce this symmetric real matrix into tri-diagonal form.
- Finally, the library interfaces with LAPACK [9] to compute the eigenvectors of the reduced size symmetric, real, tri-diagonal matrix, using a variant of the QR decomposition algorithm.
- The computed eigenvectors are multiplied with the intermediate Lanczos vectors, and these eigenvectors (which are good approximations of the eigenvectors of L) are written to the desired output location.

2.2 Lanczos with MPI + CUDA

The key area where our library is able to benefit from parallelization is in the Lanczos tri-diagonalization step, as this is the most time consuming step out of all the steps outlined in the

previous section. Just to give you a sense of the importance of parallelizing this step, we start out by providing these numbers: for a YouTube community graph [10] of **1,134,890** vertices and **5,975,248** edges, to obtain **800** eigenvectors,

- Graph and Laplacian construction takes ~3 seconds.
- The sequential Lanczos tri-diagonalization step takes ~**600** seconds.
- Computing the eigenvalues of the reduced matrix takes ~5 seconds.
- Obtaining the eigenvalues of the Laplacian using the approximations takes ~2 seconds.

It is clear that the Lanczos step is the most time consuming step in the library, and this difference becomes more stark for graphs of larger sizes, or when requesting more eigenvectors from a graph. Thus, we chose to spend most of our time optimizing this step of the clustering process.

2.2.1 The sequential Lanczos algorithm

The basic Lanczos algorithm converts a **real, symmetric** matrix A into a **tri-diagonal** matrix T that has the following format.

$$T_{mm} = \begin{pmatrix} \alpha_1 & \beta_2 & & & & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & & \\ & \beta_3 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & \beta_{m-1} & \\ & & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ 0 & & & & \beta_m & \alpha_m \end{pmatrix}$$

The motivation for using the Lanczos algorithm in the context of spectral clustering is threefold:

1. It is easier to find the eigenvalues and vectors of a tri-diagonal matrix efficiently.
2. The tri-diagonal matrix typically has a **much** smaller size than the original matrix, making it easier to operate on.
3. The eigenvectors of the tri-diagonal matrix can be used to approximate the extremal eigenvalues of the original matrix quite accurately.

Algorithm 1: A basic version of the Lanczos algorithm

Input : Matrix A, iterations M
Output: Tri-diagonal matrix elements α_1, β_1, α_2, β_2, ..., α_M, β_M.

1 Let $v_0 \leftarrow \mathbf{0}, v_1 \leftarrow \hat{p}$ for a unit vector $\hat{p}$;
2 Let $\beta_0 \leftarrow 0, \beta_1 \leftarrow 0$;
3 **for** $j \leftarrow 1, 2, \ldots, M-1$ **do**
4 $\quad w'_j \leftarrow Av_j$;
5 $\quad \alpha_j \leftarrow w_j'^T v_j$;
6 $\quad v_{j+1} \leftarrow w'_j - \alpha_j v_j - \beta_j v_{j-1}$;
7 $\quad \beta_{j+1} \leftarrow v_{j+1}^T v_{j+1}$;
8 $\quad v_{j+1} \leftarrow v_{j+1}/\beta_{j+1}$;
9 **end**
10 $w_M \leftarrow Av_M$;
11 $\alpha_M \leftarrow w_M^T v_M$;
12 **return** $\alpha_i, \beta_i, \forall i = 1, 2, ..., M$;

2.2.2 Partial re-orthogonalization

For our approximation to be good, it is important to maintain orthogonality between the intermediate vectors (called Lanczos vectors) produced in the Lanczos algorithm. The Lanczos algorithm would work perfectly well in exact arithmetic. However, as shown in [11], while working with limited precision arithmetic, the intermediate Lanczos vectors quickly lose orthogonality and spurious results begin to appear in the tri-diagonal matrix.

Ideally, we would have to compute the orthogonality between the vector v_{j+1} produced in the current iteration against *all* the j previously generated Lanczos vectors in order to check whether orthogonality is maintained. This is an $O(M \times N)$ complexity increase in *each iteration* of the algorithm, as we need to compute $O(M)$ dot products, and each dot product takes $O(N)$ amount of time to compute. Not only does this increase the complexity of the sequential algorithm, it **drastically increases the synchronization costs** in a parallel algorithm if each of the vectors are distributed and stored (imagine $O(M)$ more communication operations per loop to determine the orthogonality of the currently produced vector v_{j+1} against all the previous distributed vectors $v_{1 \ldots j}$.)

We eliminate the need to do this by *estimating* the dot product of the newly generated Lanczos vector against the previously generated Lanczos vectors using the following recurrence relation described in [11]:

$$
\begin{aligned}
\omega_{j\,j} &= 1 \\
\omega_{j\,j-1} &= \psi_j \\
\omega_{j+1\,k} &= \frac{1}{\beta_{j+1}} \left[\beta_{k+1}\omega_{j\,k+1} + (\alpha_k - \alpha_j)\omega_{j\,k} + \beta_k \omega_{j\,k-1} - \beta_j \omega_{j-1\,k} \right] + \Theta_{j+1\,k}
\end{aligned}
$$

Here, $\omega_{j\,k}$ is the *estimated* orthogonality between the Lanczos vectors v_j and v_k. ψ_j and $\Theta_{j\,k}$ are numbers that are drawn from specific normal distributions and approximate the loss in orthogonality due to machine precision. While the details of the relation are not terribly important in the context of this report (one may refer to [11] for more details), it is easy to see that the recurrence relation possesses *optimal substructure*, lending itself to efficient computation using a **dynamic programming** algorithm. Further, the recurrence relation **makes no use of the Lanczos vectors themselves**! Thus, we don't need to communicate a large number of vectors, each having a large size, in order to estimate the dot product between any two of them.

Once we are able to efficiently estimate the orthogonality between the new Lanczos vector v_{j+1} and all previously generated Lancozs vectors $k \in \{1 \ldots j\}$, we can selectively re-orthogonalize against the necessary Lanczos vectors only (those that fall below a certain precision threshold). Further, these estimates can be computed **without any parallel synchronization overhead**.

An aside: Efficient parallel, agreeable random number generation:

Notice that in order for all the nodes to independently agree on which Lanczos vectors to re-orthogonalize against, it is important for all of them to agree on *all* the random numbers ψ_j and $\Theta_{j+1\,k}$, otherwise the estimates would no longer agree, and there would be a deadlock in the re-orthogonalization step. We counter this problem by using pseudo random number generators and using the same seed value for all nodes at the start of the algorithm (using an `MPI_Bcast` to broadcast the seed), and having them agree on the number of calls to the generator. The rationale is that the library is not security-critical and pseudo random number generators work perfectly well in this setting. Further, as shown in the results section, our estimates for the loss of orthogonality do a good job with pseudo random number generators.

2.2.3 MPI-only parallel Lanczos

Given the partial re-orthogonalization version of the Lanczos algorithm, we first designed the MPI only parallel version. We discuss the CSR case below. The CSC case is similar, but the intermediate vectors produced are slightly different. We avoid discussing the CSC case in the interest of brevity (one may refer to the source code — which is well documented — in order to understand the CSC version of the algorithm).

1. The Laplacian matrix is **distributed, partitioned, and stored** in the main memories of the nodes using an efficient sparse representation (either CSR or CSC). This is shown below.

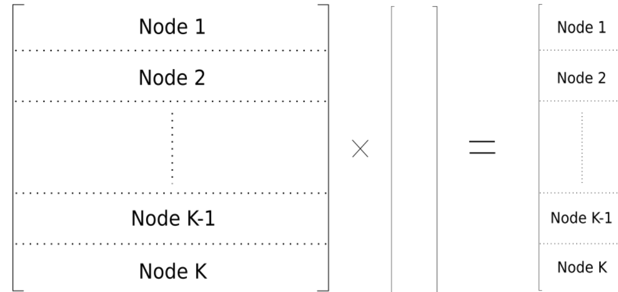

Figure 4: The distributed matrix-vector produced performed on CSR matrices

2. The matrix-vector product in step 4 of the sequential algorithm 1 produces a subpart of the vector $v_{j+1\,i}$ on each node i.

3. All dot product operations, including distributed re-orthogonalization, can be done on the distributed vector. This is a **big win**, as the matrix-vector product and re-orthogonalization steps are the most time consuming steps of the Lanczos algorithm.

4. We compute the dot products of the subparts of the vectors against the subparts of the necessary previous Lanczos vectors and store this in an array. We can then perform a single all-reduce on this array to obtain the global dot products for *all* the vectors in one shot. This strategy, in combination with the partial re-orthogonalization estimates using the recurrence relation, helped us **significantly reduce** the communication overhead of the algorithm. The scaling properties of the algorithm were *much* worse when we implemented a naïve version of Lanczos with full re-orthogonalization, and synchronization operations after every dot product.

5. Extra all-reduce operations are performed to compute α_j and β_{j+1} in each iteration. Notice that there is no way to reduce the synchronization overhead of these steps. However, we do not lose much time here, as the dot product operations are performed on vectors of **reduced size due to partitioning**, which results in significant gain per-processor for larger graphs, effectively subsuming a large part of the communication cost when compared to a sequential version.

Thus, at this point we have a working version of an MPI-parallel Lanczos algorithm that provides a decent speedup (detailed numbers are in the Experiments section). Notice that even within nodes, we are performing local submatrix-vector product and subvector-subvector product operations in the MPI parallel version. We can speedup these operations further using GPGPUs, which are specifically designed to perform such operations. The next section describes the integration of NVIDIA CUDA operations into the MPI-parallel Lanczos algorithm.

2.2.4 MPI + CUDA hybrid parallel Lanczos

Given the setting of the MPI-only parallel Lanczos algorithm, we can examine what is happening within each node locally and look at parallelizing these operations using a GPU and NVIDIA CUDA.

We make use of the cuSPARSE and cuBLAS library routines to compute sparse matrix-vector products and vector-vector products, as well as `axpy` and `ax` type operations. The rationale for choosing to use these routines is twofold:

1. We want our library to be adopted by a large community of users. For this, each part of the library must be studied and understood well. Writing our own GPU routines would have demanded an explanation of its own, since there are perfectly good libraries routines available from NVIDIA that have been well studied in the performance context. We do not perform any operations in the library that warrant such an investigation, so we avoid writing our own GPU routines. That said, the structure of the library is very modular, and replacing these routines with custom routines would be as simple as replacing a header file.
2. We want the library to be future proof. One less custom written module in the library is one less component that requires independent maintenance. Further, our library will automatically benefit from any upstream updates to the cuSPARSE and cuBLAS libraries, only making it better over time.

All operations involving matrices and vectors within a given node are performed using calls to the appropriate cuBLAS and cuSPARSE routines. Notice that this a naïve version of this involves a significant amount of data transfers between host memory and device memory on each node. Thus, simply replacing matrix-vector and vector-vector products with the corresponding calls to cuSPARSE and cuBLAS would not result in good performance. Instead, we implement the local parts of the algorithm as follows.

1. The sparse partition of the Laplacian is transferred only **once** to the device memory at the start of the algorithm, and repeatedly reused in each iteration of the algorithm. Contrary to what regular performance evaluations do, we even **account for this one time data transfer** in our results. We see that this can consume a significant amount of time for large graphs, but the cost is easily amortized over the large number of Lanczos iterations, and the overall gains are positive. Also note that we store a **sparse, partitioned** part of the matrix in each CUDA device, so the matrix does not occupy a significant amount of the device memory on each node, which helps us scale to very large graphs.

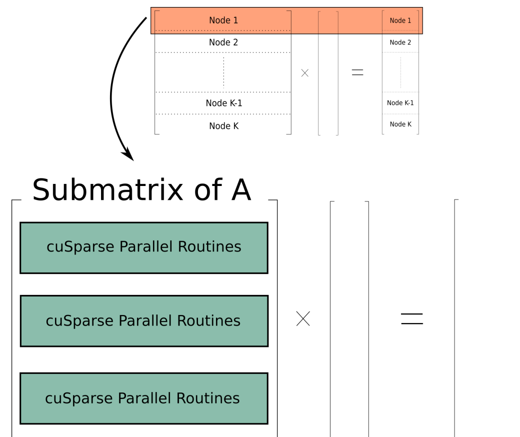

Figure 5: The working of the cuSparse routines on a partition of the entire matrix, locally on each node.

2. cuSPARSE routines are used to perform the matrix-vector product against the Lanczos vector generated in the previous step. This is shown below.

3. cuBLAS routines are used to compute the new Lanczos vector v_{j+1} using `axpy` operations, as well as to compute the normalization constants $\beta_k, \forall k \in 2, ..., M$ using u `dot` v operations.

4. Finally, apart from the sparse matrix-vector product, the re-orthogonalization sub procedure benefits a *lot* from CUDA operations. Since all of the intermediate Lanczos vectors are stored in device memory, we can very easily compute the dot products against all of the necessary vectors (determined using the recurrence relation explained in previous sections), storing them in a single array of moderate size (since it is bounded by $M << N$, the number of requested eigenvectors). These are $O(M)$ dot products and `axpy` operations, all of which stand to benefit from the used of cuBLAS operations.

5. We only transfer the final Lanczos vector v_{j+1} **once** per iteration, once all of the matrix-vector products, dot products, `axpy` operations and re-orthogonalization has taken place, thus minimizing our overhead per iteration (and bounding it by the total number of iterations M). This also amortizes the per-iteration transfer cost over all the cuBLAS and cuSPARSE operations that are performed on the vectors.

Given this MPI + CUDA hybrid parallel version of the Lanczos algorithm, we can look at putting it all together and the philosophical considerations involved in designing the library: the modules, interfaces and general structure.

2.3 Library Structure

Our source code is divided into three main components:

- A **driver program**, that parses arguments, the locations of input files, and calls the corresponding submodules of the library to perform the necessary tasks. Any **new algorithm additions or changes in high-level steps and ordering** will only require changing this file.

- A **distributed graph construction module**, that has data structures representing the various formats that a graph can be stored in, interfaces to access members of these objects, and routines to construct these partitioned graphs and their Laplacians in a distributed manner. Any **new graph storage formats** that need to be integrated into the library will only require changing the code associated with this module.

- The hybrid parallel CUDA + MPI **Lanczos** algorithm module. Changes specific to the Lanczos algorithm, different normalization procedures to be tried, and variants for other graph formats will be contained within the files associated with this module.

- The **CUDA interface module**, that contains thin wrappers around the CUDA cuBLAS and cuSPARSE routines that are to be called by each node during the Lanczos algorithm. Once may choose to replace these calls with custom calls or custom kernels. Any such changes will be contained within these files, and won't affect the rest of the library code.

- A **utilites** module, that performs CPU specific routines, and provides a thin wrapper C++ around LAPACK routines (originally written in Fortran, with different data types and memory layout schemes), that are used to efficiently computed the eigenvectors of the reduced size symmetric tri-diagonal matrix.

The main driver program is in the file `spectral_clustering.cpp` which currently takes as arguments the input file, the matrix representation format to use, whether to use CUDA or not, and the number of eigenvectors that are to be computed.

The graph construction module consists of the header file `graph.hpp` and the source file `graph.cpp`. It parses an input edge list file, and returns a pointer to the distributed graph data structure. Distributed graph data structures expose **interfaces** to construct the Laplacians in parallel, in CSR or CSC format.

The Lanczos module consists of the `lanczos.cpp`, `lanczos_cuda.cpp` and `lanczos_cuda.hpp` files, and has routines to compute the tri-diagonal approximation matrix of input graphs in various supported formats, for various MPI + CUDA combinations. It calls the cuBLAS and cuSPARSE routines using wrappers in `cu_utils.cpp`.

The `utils.hpp` and `utils.cpp` files contain CPU methods for BLAS routines, as well as wrappers to use LAPACK.

The following figure illustrates the flow of control within the library.

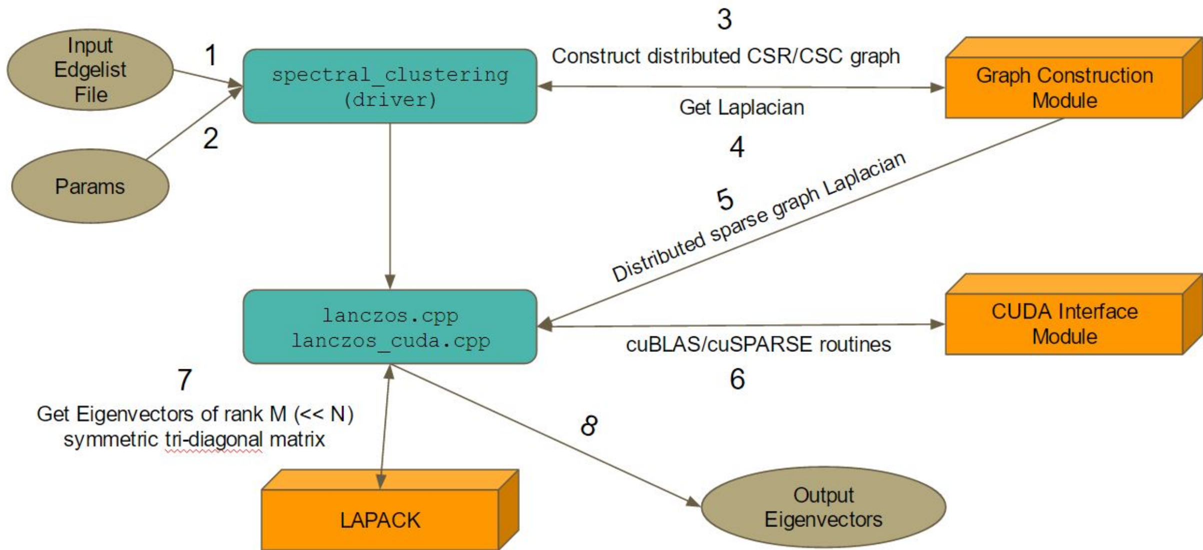

Figure 6: Code flow diagram of our project

2.4 Sparse Matrix Representations

Our target data sets are real world graphs, whose degree distributions typically tend to align well with **powerlaw** distributions, the result being that these graphs are sparse graphs with sparse adjacency matrices. If these matrices were represented as full $N \times N$ arrays in C++, for larger graphs, we would quickly run out of even cluster node memory, let alone device memory! Furthermore, a majority of the elements in these matrices will be zero, reducing computational density and increasing workload imbalance.

This problem is alleviated by using sparse matrix representations, which occupy far lesser space than their corresponding full representations. Sparse graphs are commonly represented in three formats:

- Compressed Sparse Row (CSR)
- Compressed Sparse Column (CSC)

- Coordinate List (COO)

The **CSR** format has the best locality for row-major access, which is the access pattern commonly followed in matrix-vector multiplications. Here, the matrix is decomposed into 3 arrays - data values, column indices and row pointers.

The **CSC** format is useful for fast column slicing, however matrix-vector operations are more expensive (due to non-locality of memory access). Further, cuSparse routines don't directly operate on CSC matrices, so they must be transposed into CSR matrices before they are operated on. The advantage to using the CSC format is that there is less memory overhead, as the intermediate Lanczos vectors are also stored in a distributed manner. So it is better at handling larger graphs in device memory, but this comes at the cost of performance.

COO format is less frequently used, as it does not directly support arithmetic operations or slicing, and must first be converted to either CSC or CSR format. We have avoided using it in our library for these reasons.

One may refer to [12] for a better understanding of sparse matrix representations.

We now describe our experimental setup and results.

3 Experimental Setup

- Our code was written from scratch using C++. We used some cuBLAS (7.5) and cuSPARSE (7.5) helper routines for linear algebra operations.
- All results in subsequent sections are from tests run on the Latedays cluster. Each cluster node has 2 x Xeon E5-2620 v3 CPUs and an NVIDIA Tesla K40 GPU.
- The program was compiled using `nvcc` (CUDA 7.5 release) linked with OpenMPI 1.6.2.

3.1 Datasets

- We used datasets of undirected graphs obtained from the Stanford Network Analysis Project [10]. These files are available as adjacency lists. We provide helper Python scripts to convert these edge lists into symmetric edge lists (where $A \to B$ and $B \to A$ both exist in the edge list), which is the input format currently supported by the distributed graph construction module.
- Correctness tests were run on the Autonomous Systems graph dataset (`as20000102.txt`), which has 6,474 nodes and 13,895 edges.
- Performance tuning was done on the Gowalla Social Networks dataset (`loc-gowalla_edges.txt`), which has 196,591 nodes and 950,327 edges.
- Scaling tests were run on the YouTube Communities dataset (`com-youtube.ungraph.txt`), which has 1,134,890 nodes and 5,975,248 edges.

3.1.1 Correctness testing methodology

In order to ensure the correctness of the output of our library, we employed two strategies:

1. When the graph was small enough to use ARPACK/LAPACK to compute the eigenvalues and eigenvectors in reasonable time, we directly compared them with a tolerance threshold (to account for randomness in the algorithm).

2. For larger graphs where the sequential algorithm failed to terminate in reasonable time, we checked that the generated Lanczos vector were all orthogonal to each other, that the different formats and versions produced the same eigenvalues (within a tolerance threshold), and checked against the large I/O library version outputs where applicable.

4 Experimental Evaluation

We compared our implementation against the sequential baseline of ARPACK - a collection of Fortran 77 subroutines to solve large scale eignevalue problems efficiently. For symmetric matrices, it uses an algorithmic variant of the Lanczos tridiagonalization called the Implicitly Restarted Lanczos Method (IRLM).

In figure 7, we have plotted the computation time for the baseline, our sequential partial re-orthogonalization strategy (using the dynamic programming algorithm to estimate vectors to re-orthogonalize against), and our hybrid MPI + CUDA implementation. We observe that our hybrid MPI + CUDA partial re-orthogonalization based Lanczos algorithm outperforms the baseline ARPACK method (with a speedup of ∼**10x**), as well as the sequential implementation of the Lanczos algorithm with partial-reorthogonalization.

Note: For a very large number of eigenvectors M, the ARPACK process exceeds the 20 minute time limit that we set, on the Latedays cluster.

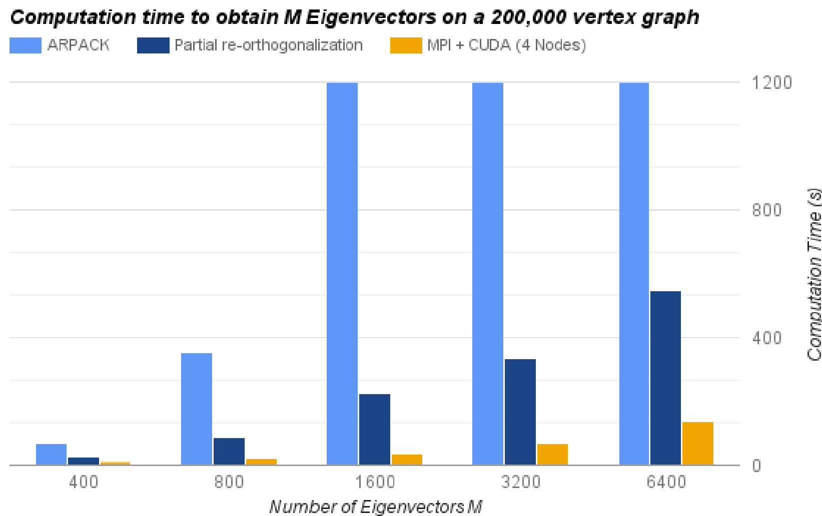

Figure 7: Computation time for different approaches

Table 1: The overall time in obtaining M eigenvectors using sequential ARPACK, sequential Lanczos with Partial Re-Orthogonalization (SPRO), and the hybrid MPI + CUDA parallel version (with 4 nodes).

M	ARPACK	SPRO	MPI+CUDA (s)
100	5.78	3.01	5.76
200	16.12	8.54	7.34
400	71.32	25.98	11.47
800	354.34	85.55	20.98
1600	1200+	223.82	37.25
3200	1200+	336.51	67.30
6400	1200+	547.87	138.26

In figure 8, we plotted the computation time for computing M eigenvectors (with CSR matrix representation) using the sequential version, MPI-only using 8 nodes and MPI + CUDA using 4 nodes. We observe that the MPI + CUDA with only 4 nodes outperforms even MPI-only with 8 nodes and that the computation time increases almost linearly with increasing number of eigenvectors to be computed in all three cases (due to the clever partial re-orthogonalization strategy). Note that the MPI + CUDA hybrid version scales the best with increasing problem size, due to its ability to effectively utilize the extra available hardware parallelism.

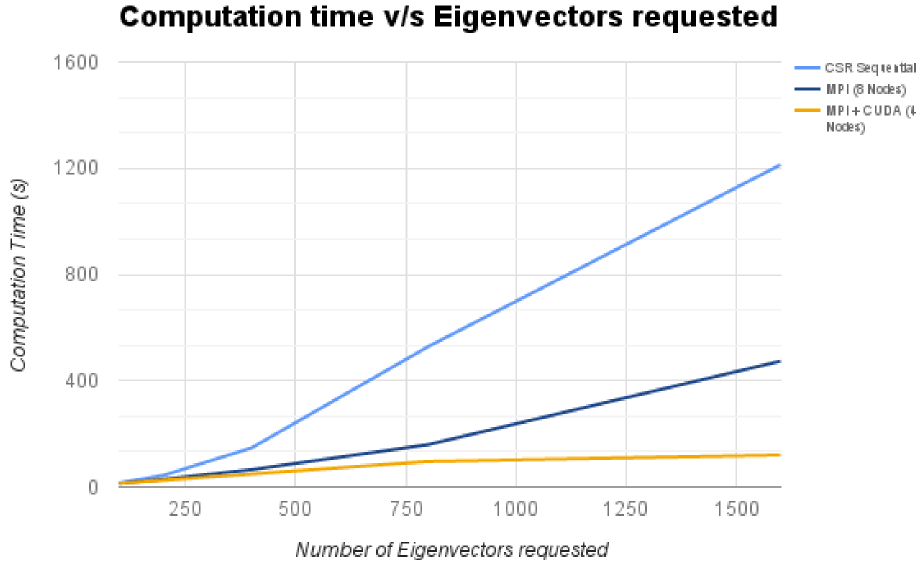

Figure 8: Computation time vs eigenvectors requested for different approaches

In figure 9, we show the results of **strong scaling** tests for our MPI+CUDA hybrid algorithm, on the YouTube Community graph dataset with 1 Million+ vertices. Due to the dynamic programming algorithm that determines the need to re-orthogonalize with **no synchronization overhead**, the graph exhibits a near-linear speedup structure. Note however that there are all-reduce communication patterns whenever the need to re-orthogonalize *is* determined, and that limits the algorithm from achieving perfect linear speedup.

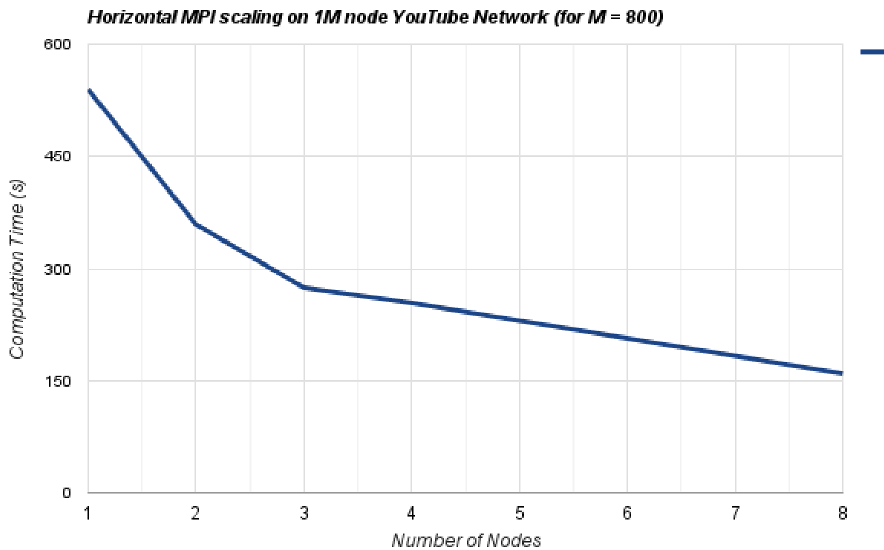

Figure 9: Computation time vs number of nodes for 1M+ element YouTube community graph

As a sample of the capability of the algorithm, we have visualized the output of clustering the Autonomous Networks dataset using Gephi in the following figure.. The red and cyan portions show the two different user clusters in the graph.

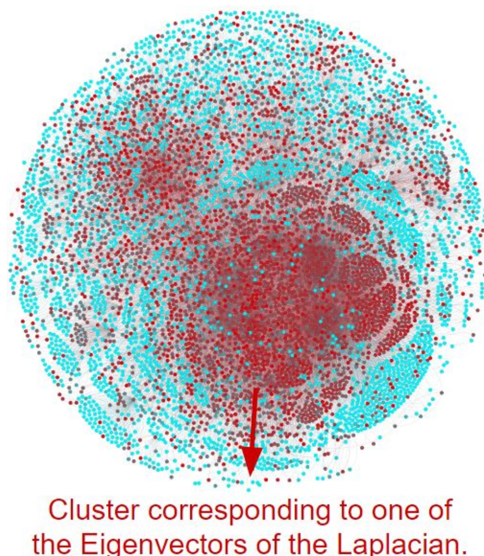

Figure 10: The result of normalized cut clustering visualized in Gephi

5 Surprises and Lessons Learned

- A strong **domain knowledge** of the problem area is required to correctly choose the algorithm which can efficiently exploit the available hardware parallelism. Our background study showed us that there are many different classes of eigendecomposition algorithms. Some of these (such as Implicitly Restarted Lanczos used in ARPACK) are commonly used as they are very efficient when implemented sequentially, but cannot be easily parallelized. Others, such as Lanczos with Partial Re-orthogonalization, are more conducive to parallelization, but are slower sequentially and have a higher memory overhead. We spent a significant amount of time trying to understand these trade-offs, and to choose the right algorithm for our library.

- We tried a variety of different techniques to reduce the **high communication overhead** invovled in various steps of the algorithm. There were all-reduce and all-gather communication patterns that were proving to be intractable, and data had to be moved between device and host memories often for CUDA processing. We had to re-write large parts of the algorithm so that all matrix and vector operations were done on the GPU, and all the necessary data was available to be transferred in a single operation. Most importantly, we learned that a seemingly **high synchronization overhead was acceptable, as long as the computation time was the bottleneck for large problem sizes**.

- Sparse **matrix representations** have different access locality costs and computation density properties for different application and operations. For us, Compressed Sparse Row (CSR) representation is the most efficient for matrix-vector operations, but comes at the cost of increased storage. Since our matrix was symmetric, we considered saving storage space by only storing the Upper or Lower triangular matrix, but this introduced a variety of problems in the hybrid parallel setting in terms of work division. cuSparse symmetric matrix-vector multiplications were also found to be slower, as it uses CUDA atomics to compute the transpose. We learnt that **it is important to understand the advantages**

and disadvantages of various representations with respect to the operations involved in an algorithm to make an informed decision about which one to use.

- The MPI + CUDA hybrid parallel version gave a greater speedup than we had originally expected. This was due to the double reduction in input size to each of the individual computation units. For our use case, a computation unit was a thread/warp within the GPU of a single node. The already partitioned matrix was further partitioned and passed along to the GPU units, which resulted in a double reduction in the problem size. This taught us that **we must estimate speedups correctly, so that we do not incorrectly write away an algorithm for having a seemingly large synchronization overhead.**

6 Conclusions

We have designed and implemented a high performance library for spectral graph clustering, that uses MPI and CUDA parallelism to efficiently handle large graphs without compromising on the quality of the results. Our HPSCG library provides a flexible and powerful interface for clustering large graphs, and is intuitive to use.

The library is future-proof and contains modular components that can easily be swapped out in favor of custom user modules. It is also extensible, as the code associated with each module is tightly contained, and adding new procedures would large involve local changes only.

Our evaluations suggest that the library performs better than popular available sequential libraries, while providing output that is comparable to them. The hybrid MPI + CUDA parallel Lanczos algorithm also scales well with an increase in computation units.

7 Future Work

- With the regular MPI implementation, only pointers to host memory buffers can be passed to MPI. As a result, when we needed to send GPU buffers to other nodes, we had to first stage it through host memory using cudaMemcpy. Using CUDA-aware MPI would enable us to directly send and receive GPU buffers without staging in host memory, reducing our communication overhead.
- We could provide wrappers to read the input graph from a larger number of commonly used network file systems (like HDFS). This would directly result in an increased adoption of the library among users.
- Currently, the intermediate Lanczos vectors are being stored in memory, which could be an issue if one wants to request a very large number of eigenvectors. We can dump the intermediate vectors to disk and use a cache (with a suitable replacement policy like LRU) to determine which vectors need to reside in memory.
- We could reduce external dependencies by implementing the QR decomposition algorithm (used to compute the eigenvectors of the reduced size tri-diagonal matrix) within the library, instead of using an LAPACK routine.
- We could use more sophisticated clustering algorithms, such as K-means and hierarchical clustering, to transform the output eigenvectors into a clustering of the graph.
- Documenting the scaling properties of our algorithm on datasets possessing different sparsity structure is essential for our library to be adopted by a wide community of users.

Distribution of Credit: Equal work was done by both members.

References

[1] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[2] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[3] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances In Neural Information Processing Systems*, pages 849–856. MIT Press, 2001.

[4] Jane Cullum and Ralph A. Willoughby. A survey of lanczos procedures for very large real 'symmetric' eigenvalue problems. *Journal of Computational and Applied Mathematics*, 12:37 – 60, 1985.

[5] R. Lehoucq, D. Sorensen, and C. Yang. *ARPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1998.

[6] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed ¡today¿].

[7] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[8] Parallel eigensolver for graph spectral analysis on gpu. `http://linhr.me/15618/`. Accessed: 2016-10-31.

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

[10] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[11] Horst D Simon. The lanczos algorithm with partial reorthogonalization. *Mathematics of Computation*, 42(165):115–142, 1984.

[12] Chapter 3: Sparse BLAS. `http://www.netlib.org/blas/blast-forum/chapter3.pdf`. Accessed: 2016-12-11.