KTH Royal Institute of Technology

Department of Mathematics

Project Report

A Parallel Implementation of the Spectral Clustering Algorithm

Niclas Joshua Popp

Course: SF2568 Parallel Computations for Large-Scale Systems

Teacher: Niclas Jansson

Submission Date: 15. May 2022

Abstract

Clustering is a core technique in unsupervised learning and has numerous applications in science such as the graph partitioning problem. Spectral clustering is one of the most popular clustering methods. It can be implemented efficiently using algorithms from numerical linear algebra which have the potential for significant speedup when being parallelized. In this project a parallel implementation of the spectral clustering algorithm is developed and two applications are presented to showcase its functionality. The first example considers the partitioning of the Stockholm Tunnelbana network while the second application demonstrates the applicability of the presented implementation on an adjacency graph of a real-world sparse matrix coming from PDE models.

Contents

1 Introduction

Clustering is generally refered to as the process of grouping large sets of data. As one of the fundamental techniques in unsupervised learning it is applied to various different scientific problems such as graph partitioning. *Spectral Clustering* is one of the most popular algorithms for this purpose. It can be implemented efficiently using algorithms from numerical linear algebra which have the potential for significant speedup when being parallelized. The intended goal of this project is to develop a parallel implementation of the spectral clustering algorithm and demonstrate its applicability through two real-world problems.

The input for spectral clustering is an undirected graph together with a desired number of clusters k. The algorithm comprises of two main subproblems. In the first step the k leading eigenvectors of the normalized or unnormalized graph Laplacian are computed. For this project the correct Laplacian is assumed to be given. The calculation of the eigenvectors can be done by several different schemes. We use the the QR method with Gram-Schmidt orthogonalization for this purpose since it simultaneously outputs the eigenvalues and eigenvectors. The eigenpairs have to be sorted according to ascending absolute value of the eigenvalues. This can be carried out using a parallel sorting scheme such odd-even transposition sort which was used for this project. The second step involves k-means clustering on the rows of the matrix that stores the first k eigenvectors from the previous step as columns. The resulting clusters will then correspond to the final output.

The theoretical part of the project features both the individual analysis of the separate subproblems as well as the assessment of the complete algorithm. Similarly, the experimental speedup investigation was carried out independently for all parts and for the entire algorithm. These results are then linked to the theoretical performance predictions. The implementation is realized in C using MPI and the benchmarking was carried out on Dardel using a random graph with $2^{10} = 1024$ nodes and $2^{13} = 8192$ edges.
Apart from the theoretical and experimental analysis, we present two real-world applications of spectral clustering. The first example features the Stockholm Tunnelbana network which was partitioned into $k = 2, 3, 4$ clusters. The results can be interpreted geographically and thus intuitively show how the method works on a real network. As second application we partition an adjacency graph of a sparse matrix that occurs in thermoelasticity models using partial differential equations (PDE) [8]. This example is intended to test the applicability of the current implementation to larger problems.

The report for this project is divided into 5 parts. For better readability all necessary mathematical foundations are introduced in chapter 2. The parallelized version of the spectral clustering algorithm is analyzed theoretically and benchmarked experimentally in chapter 3. In chapter 4 the two applications are presented. We conclude with a discussion of the current implementation and possible further optimizations.

2 Mathematical Foundations

2.1 Graph Theory

In order to develop the right setup for spectral clustering, a few graph theoretic notions have to be introduced. A graph is given by a tuple $\{V, E\}$, where V describes its set of vertices and E describes the set of edges. In the following we assume an undirected and unweighted graph with $n = |V|$.

Definition 2.1. The *weight matrix* W of a graph is given by

$$w_{i,j} = \begin{cases} 1 \text{ if vertices i and j are connected} \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

The corresponding *degree matrix* D is a diagonal matrix with

$$d_{i,i} = \sum_{j=1}^{n} w_{i,j} \tag{2}$$

These two matrices are used to construct the *unnormalized graph Laplacian* L_{unnorm}

$$L_{unnorm} = D - W \tag{3}$$

and the *normalized graph Laplacian* L_{norm}

$$L_{norm} = D^{1/2} L D^{1/2} \tag{4}$$

The eigenvalue and eigenspace structure of these matrices are the first step in the justification of the spectral clustering algorithm.

Theorem 2.2. *The multiplicity k of the eigenvalue 0 of L_{norm} and L_{unnorm} equals the number of connected components $A_1, ..., A_k$ of the graph. The eigenspace of 0 is spanned by $1_{A_1}, ..., 1_{A_k}$ for L_{unnorm} and $D^{1/2}1_{A_1}, ..., D^{1/2}1_{A_k}$ for L_{norm}. $1_{A_i} \in \{0,1\}^n$ denote the indicator vectors of the different components*

$$1_{A_i}[j] = \begin{cases} 1 \textit{ if vertix } i \textit{ and is part of } A_j \\ 0 \textit{ otherwise} \end{cases} \tag{5}$$

In this case, a subgraph $A \subseteq G$ is called a *connected component* if any vertices in it can be joined by a sequence of edges between vertices in A and there are no connections to $G \setminus A$.

2.2 Spectral Clustering Pipeline

If a graph completely dissolves into k different connected components, the graph clustering problem can be solved by applying theorem 2.2. Let $v_1, ..., v_k$ be the first k eigenvectors of L_{unnorm}. Define

$$U = [v_1, ..., v_k] \in \mathbb{R}^{n \times k} \tag{6}$$

as the matrix that has these vectors as columns. According to theorem 2.2, there will be exactly k distinct *row* vectors of U, each corresponding to a component of G. We can

use any distance based clustering scheme on the rows of U to identify them. The most common choice is the k-means algorithm. The following theorem provides justification for applying the same procedure on a graph that does not split completely into distinct components but can be clustered in a meaningful way.

Theorem 2.3. *(**Bauer-Fike Theorem**) Assume $A \in \mathbb{R}^{n\times n}$ is diagonalizable with eigenvalues λ_i. Let δA be a perturbation of A. Then for every eigenvalue μ of $A + \delta A$ there exists an eigenvalue λ_j of A such that*

$$|\mu - \lambda_j| \leq \kappa(S)\|\delta A\| \tag{7}$$

where S is the matrix with the eigenvectors of A as columns and $\kappa(S) = \|S\|\|S^{-1}\|$ denotes its condition number.

The Bauer-Fike theorem states that for a graphs that approximately is made up by k clusters $C_1, ..., C_k$, we can still expect the first k eigenvectors to roughly approximate span$\{1_{C_1}, ..., 1_{C_k}\}$. Thus, it is still justified to apply k-means on the rows of U in order to determine the clusters. A similar reasoning applies for L_{norm} although the resulting clusters have slightly different properties. A more in depth evaluation of the theory of spectral clustering can be found in [6]. The final algorithm is given as algorithm 1 and the computational pipeline is visualized in figure 1. In the next two sections we focus on the foundations that are necessary to calculate the first k eigenvectors of a Laplacian through the QR method with Gram-Schmidt orthogonalization and explain k-means in more detail.

Algorithm 1: Spectral Clustering Algorithm

Input: Normalized or Unnormalized Laplacian $L \in \mathbb{R}^{n\times n}$, number of clusters k
Output: k clusters represented by $C_1, ..., C_k$
Calculate the first k eigenvectors $u_1, ..., u_k$ of L and set $U = [u_1, ..., u_k]$
Apply k-means with k clusters on the rows of U
Output the resulting clusters $C_1, ..., C_k$

2.3 Eigenvector Calculation

2.3.1 QR Iteration

The QR algorithm [5] is a numerical scheme to approximate the Schur factorization of a matrix.

Definition 2.4. Given a matrix $A \in \mathbb{R}^{n\times n}$, a *Schur factorization* is of the form

$$A = PVP^* \tag{8}$$

where $P^*P = I$ and U is upper traingular.

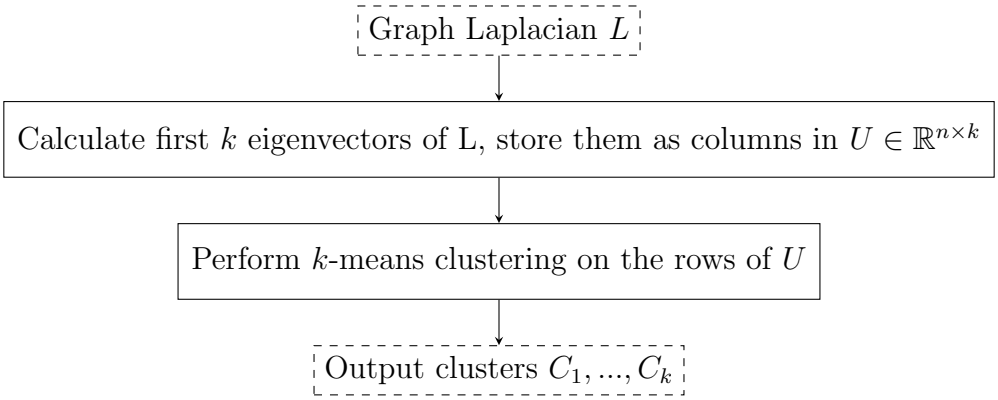

Figure 1: Pipeline for Spectral clustering

The QR method and the resulting Schur factorization are particularly useful for spectral clustering since it not only returns the eigenvectors of a matrix but also the corresponding eigenvalues which are simply the diagonal entries of V. The eigenvalues can be sorted by ascending absolute value and the eigenvectors corresponding to the k first eigenvalues can be extracted from P. These vectors are then used to construct U in algorithm 1. Other numerical methods usually only determine the eigenvalues or eigenvectors but not both simultaneously which is required for spectral clustering. Algorithm 2 states the QR method in the way it was used for this project. There are multiple ways to measure the convergence of the QR method. The important fact for spectral clustering is that for Laplacian matrices which are symmetric the error decays to a scale that is utilizable for the overall algorithm within only a few steps. A more detailed elaboration can be found in [2] and [4]. The parts marked in orange will be parallelized.

Algorithm 2: QR Algorithm

Input: $A \in \mathbb{R}^{n \times n}$
Output: Schur Factorization of $A = PVP'$
Set $Q_0 = I$ and $R_0 = a$
for *i=1,2,3,...* **do**
  Calculate QR factorization of A: $A = QR$
  Update Q and R:
  $Q_i = RQ$ and $R_i = R_{i-1}R$ (only diagonal entries are needed)
Set $V = Q_\infty$ and $P = R_\infty$

2.3.2 Gram-Schmidt Algorithm

In order to carry out the QR algorithm described in the previous section we need to determine a QR factorization of a matrix in each step. Arguably the simplest method for this purpose is the so-called Gram-Schmidt algorithm. The algorithm orthogonalizes the columns of a matrix A against each other and stores the resulting vectors in a matrix Q. The triangular matrix R contains the coefficients that are necessary to recover the original matrix A. There are multiple versions of the QR method and algorithm 3 states

the one which was implemented for this project. The parts highlighted in orange can be parallelized. One problem with the Gram-Schmidt method is that the q vectors can lose orthogonality die to numerical error. Thus, we re-orthogonalize the vectors in every step by carrying out Gram-Schmidt twice. This approach is called the *Double Gram-Schmidt method*.

Algorithm 3: Gram Schmidt Method

Input: $A \in \mathbb{R}^{n \times n}$
Output: QR Factorization of $A = QR$
Set $Q_0 = []$, $R_0 = []$ and $A_0 = A$
for *i=1,2,3,...,n* **do**
  Set $q_j = \frac{A_{i-1}(:,i)}{\|A_{i-1}(:,i)\|}$
  $Q_i = [Q_{i-1}, q_i]$
  $r_i = q_i^T A_{i-1}$
  $R_i = \begin{bmatrix} Q_{i-1} \\ r_i \end{bmatrix}$
  $A_i = A_{i-1} - q_i r_i^T$

2.4 K-means

The idea of the k-means algorithm is to partition a dataset $\mathcal{D}$ into k clusters $C_1, ..., C_k$ based on *centroids* $c_1, ..., c_k$. Each point belongs to the cluster that is defined by the closest centroid. The goal is to minimize

$$J(\mathcal{D}, \{c_i\}) = \sum_{j=1}^{k} \sum_{x \in C_j} d(x, c_j) \tag{9}$$

where $d(\cdot, \cdot)$ is a distance. For spectral clustering it satisfies to use the euclidean distance. Unfortunately, finding such optimal clustering in NP hard and thus an approximation is needed. Therefore, we start with k initial guesses for the centroids. In each step the datapoints are assigned to the cluster corresponding to the closest centroid. The centroids are then updated to be the mean of all points that belong to the same cluster. These two steps are repeated until the centroids stop moving or their increment is below a certain threshold. The computational procedure is given as algorithm 4 and the parts that will be parallelized are highlighted in orange. A more detailed elaboration can be found in [3].

Algorithm 4: k-Means Clustering

Input: Starting centers $c_1, ..., c_k \in \mathbb{R}^p$, Datapoints $x_1, ..., x_n \mathbb{R}^p$

Output: k clusters defined by centers $c_1, ..., c_k$

Set $Q_0 = []$, $R_0 = []$ and $A_0 = A$

for *l=1,2,3,...* **do**

Determine $V_j = \{x_i | \operatorname{argmin}_{m=1,...,k} d(x_i, c_m) = c_j\}$ for $j = 1, ..., k$ using a distance d

Update $c_j = \frac{1}{|V_j|} \sum_{x_i \in V_j} x_i$ for $j = 1, ..., k$

3 Parallelization of Spectral Clustering

3.1 Setup and Theoretical Analysis

3.1.1 Eigenvector Calculation

As described in the previous section, the eigenvector calculation through the QR method involves carrying out the (Double) Gram-Schmidt algorithm and then a matrix-matrix multiplication in every step. The following section describes how this can be carried out in parallel.

Gram Schmidt Algorithm The steps of the Gram-Schmidt algorithm that can be parallelized are highlighted in algorithm 3. First, we analyze the calculation of $\|A_{i-1}(:,i)\|$ which can be computed through a parallel scalar product using

$$\|A_{i-1}(:,i)\|^2 = A_{i-1}(:,i)^T A_{i-1}(:,i) \tag{10}$$

The parallel execution of scalar products has been introduced in lecture 6 [9]. Assuming there are P processes, we scatter the vector $\|A_{i-1}(:,i)\|$ and each process calculates the partial scalar product. The final result is determined through a global summation. For the following analysis we assume that the scatter and gather routines are implemented using recursive doubling and expect the worst-case performance. The parallel time is given by

$$\begin{aligned} t_{norm\ of\ q} &= t_{scatter} + t_{local\ computation} + t_{global\ summation} \\ &= \log(P) \times \Big(t_{startup} + n t_{data}\Big) \\ &\quad + 2\Big(\frac{n}{P} - 1\Big) t_a \\ &\quad + \log(P) \times (t_{startup} + t_{data} + t_a) \end{aligned} \tag{11}$$

for a vector with n entries. At the end only the root process holds the result and can calculate q

$$q = \frac{A_{i-1}(:,i)}{\|A_{i-1}(:,i)\|} \tag{12}$$

by normalizing it

$$t_{calculate\ q} = t_{norm\ of\ q} + (n+1)t_a \tag{13}$$

For simplicity we assume one algebraic operation to determine the square root in order to receive the norm. The entries of r

$$r = q^T A_{i-1} \tag{14}$$

are calculated using scalar products. In the communication part for r_i, the vector q_i has to be broadcasted, A_{i-1} has to be scattered and r_i has to be gathered at the end. We assume that P divides n such that each process receives n/P entire columns of A_{i-1} and

calculates n/P scalar products. The overall execution time is

$$\begin{aligned} t_{calculate\ r} &= t_{broadcast\ q\ and\ A_{i-1}} + t_{local\ computation} + t_{gather\ r} \\ &= \log(P) \times \Big(t_{startup} + (n+n^2)t_{data}\Big) \\ &\quad + 2\frac{n}{P}(n-1)t_a \\ &\quad + \log(P) \times \Big(t_{startup} + nt_{data}\Big) \end{aligned} \tag{15}$$

If we do not assume distributed memory, the last step to update A_i cannot be parallelized but adds a serial computational cost of

$$t_{update\ A_i} = 2n^2 t_a \tag{16}$$

The overall time for parallel Gram Schmidt is

$$t_{Gram\ Schmidt} = t_{calculate\ q} + t_{calculate\ r} + t_{update\ A_i} \tag{17}$$

The remaining steps only involve memory allocation and are neglected for the theoretical runtime analysis. The time for Double Gram-Schmidt would simply be twice the time of the standard version.

QR Algorithm After the determining a QR factorization, the QR method consist of one matrix-matrix multiplication in each step. The matrix Q is broadcasted while the matrix R is distributed such that every process holds n/P entire rows. Again, we assume that P divides n. Figure 2 visualizes this data distribution.

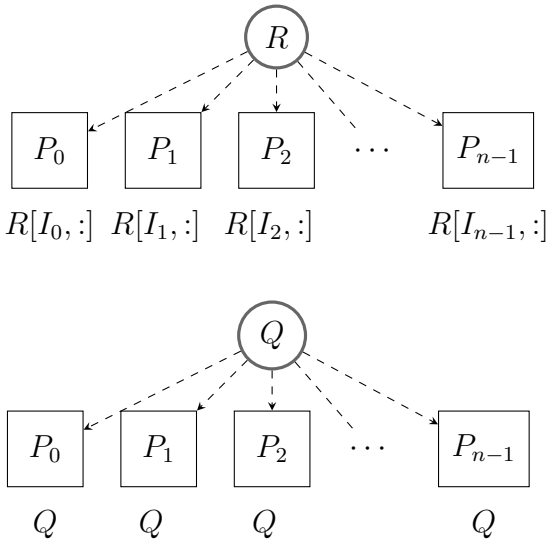

Figure 2: The schematic visualizes the data distribution for one matrix-matrix multiplication during the QR method. Q is broadcasted while the rows of R are distributed according to $I_j = \{k \times \frac{n}{P}, k \times \frac{n}{P} + 1, ..., k \times \frac{n}{P} + (\frac{n}{P} - 1)\}$

Each process then calculates n/P rows of $R \times Q$ which corresponds to n^2/P scalar products. the result matrix is gathered at the end.

$$\begin{aligned} t_{R\times Q} &= t_{broadcast\ Q} + t_{scatter\ R} + t_{local\ computation} + t_{gather\ R\times Q} \\ &= \log(P) \times \Big(t_{startup} + n^2 t_{data}\Big) \\ &\quad + \log(P) \times \Big(t_{startup} + n^2 t_{data}\Big) \\ &\quad + \frac{n^2}{P} 2(n-1) t_a \\ &\quad + \log(P) \times \Big(t_{startup} + n^2 t_{data}\Big) \end{aligned} \tag{18}$$

The procedure for R could be fully parallelized in theory since it involves a matrix product. However, we are only interested in the diagonal entries which reduces the problem to a pointwise product of two vectors. As we do not use distributed memory, every diagonal entry of R would require only one algebraic operation in contrast to multiple communication steps. Thus, this part is kept in serial since t_{data} is larger than t_a.

$$t_{update\ R} = n t_a \tag{19}$$

Overall the parallel execution time for the QR algorithm with Gram Schmidt orthogonalization and M_{QR} iterations is

$$t_{QR} = M_{QR} \times (t_{Gram\ Schmidt} + t_{R\times Q} + t_{update\ R}) \tag{20}$$

3.1.2 Eigenvalue Sorting

As explained in the foundations section, the eigenvalues of the Laplacian L have to be sorted by ascending absolute value in order to select the correct first k eigenvectors. Therefore, the list of n eigenvalues is divided into P parts of size n/P. The local sorting part at the beginning is carried out using quicksort and has complexity

$$t_{local\ sort} = \mathcal{O}\Big(\frac{n}{p}\log(\frac{n}{P})\Big) \tag{21}$$

Subsequently, there are P steps, either odd or even, during which two neighboring processes send/receive their lists and perform local merges to keep the larger or lower part. The time complexity for this part is

$$t_{exchange\ and\ sort} = P \times 2\Big(t_{startup} + \frac{n}{P} t_{data}\Big) + P \times (2\frac{n}{P} - 1) \tag{22}$$

and the overall complexity

$$t_{eigenvalue\ sorting} = t_{local\ sort} + t_{exchange\ and\ sort} \tag{23}$$

Note that we are not explicitly interested in the first k eigenvalues themselves but only their location on the diagonal of R such that we can extract the corresponding eigenvectors form Q using their indices. Thus, during the sorting process we pass an index array together with the eigenvalues and perform the same exchanges to it. We select the first k elements from this list after the sorting process to receive the correct positions.

3.1.3 K-means

Before starting the iteration of k-means the rows of the U matrix are scattered among the processes

$$t_{scatter\ U} = \log(P) \times \left(t_{startup} + nkt_{data}\right) \tag{24}$$

In each step of the iteration the centroids are broadcasted and the processes determine the distance to the centroids for their chunk of U. These distances are then used to determine the closest centroid for each row vector and that information is gathered by the root process. For M_{km} iterations the time complexity is

$$t_{k-means\ iteration} = M_{km} \times (t_{broadcast\ centroids} + t_{calculate\ distance} + t_{gather\ labels} + t_{serial}) \tag{25}$$

The separate steps have the following complexities

$$t_{broadcast\ centroids} = \log(P) \times \left(t_{startup} + k^2 t_{data}\right) \tag{26}$$

$$t_{calculate\ distance} = k\frac{n}{P}2k \tag{27}$$

$$t_{gather\ labels} = \log(P) \times \left(t_{startup} + nt_{data}\right) \tag{28}$$

Furthermore, the root process determines the new centroids in serial by summing up all the points that belong to one cluster and then normalize that vector.

$$t_{serial} = kn + k \tag{29}$$

The overall complexity is

$$t_{k-means} = t_{scatter\ U} + t_{k-means\ iteration} \tag{30}$$

3.1.4 Overall speedup

The time for the overall algorithm is given by

$$t_{Spectral\ Clustering} = t_{QR} + t_{eigenvalue\ sorting} + t_{k-means} \tag{31}$$

and the speedup can be calculated by

$$S_P = \frac{T_1^*}{T_P} \tag{32}$$

where T_1^* is the execution time on one processor and T_P is the execution time on P processors [1]. Note that $t_{Spectral\ Clustering}$ is dependent on P and T_1^* is received by setting $P = 1$. The closed form formula for the overall algorithm is omitted for clarity.
From the theoretical timing analysis we can derive some performance predictions. We expect best speedup for QR algorithm since the dominating complexity is $\mathcal{O}(n^3)$ from matrix multiplication and the work can be split evenly between the processes. The serial part for updating the diagonal entries of R has much lower time complexity in comparison to the parallel part and thus should not be a significant bottleneck according to Amdahl's law. The speedup of odd-even sorting will be affected by the communication time especially when processors are located on different nodes. Thus, we do not except it to perform as good as the QR method. The distance calculation in k-means is embarrassingly parallel which has the potential for good parallel speedup. However, the serial part of calculating the new centroids will limit the speedup due to Amdahl's law.

3.2 Experimental Benchmarking

The algorithm was benchmarked on a random graph with $2^{10} = 1024$ nodes and $2^{13} = 8192$ edges which was sampled using the Julia graphs library [7]. We used 10 iterations of the QR methods and $k = 16$ clusters. For each $P \in \{1, 2, 4, 8, 16\}$ we measured the execution time of 10 runs and calculated the average. The results together with the speedup can be found in figure 3. The exact times are listed in supplementary table 1. We observe that the predictions from the theoretical analysis are indeed represented in the experimental benchmarking. The implementation of QR algorithm features the best speedup of all subproblems and reaches $S_P \approx 10$ for $P = 10$. As noted in the previous section we expect this since the algorithm is fully parallelized and features a high complexity such that it can benefit from distributing the calculations amongst the processes. In contrast, the speedup of the odd-even transposition sort implementation is limited by the amount of communication that is necessary in every step. Nevertheless, the algorithm is fully parallel apart from the local sort at the beginning and for 16 processes it is still ≈ 3.5 times faster than serial quicksort. The k-means implementation reaches a speedup of ≈ 2.4 for $P = 16$ which is the lowest among the subproblems. This reflects the prediction from the previous section according to which the serial computation part for updating the centroids will limit the speedup according to Amdahl's law. Additionally, the execution time increases slightly from $k = 4$ to $k = 8$. A potential reason could be the sensitivity to the initialization of the centroids which might lead to a different number of iterations before the algorithms stops. The execution time of the entire algorithm is mainly determined by the QR method since it requires the largest amount of computations due to the matrix multiplications. Consequently, the speedup is similar to the QR method as well and reaches ≈ 9.5 for $P = 16$.

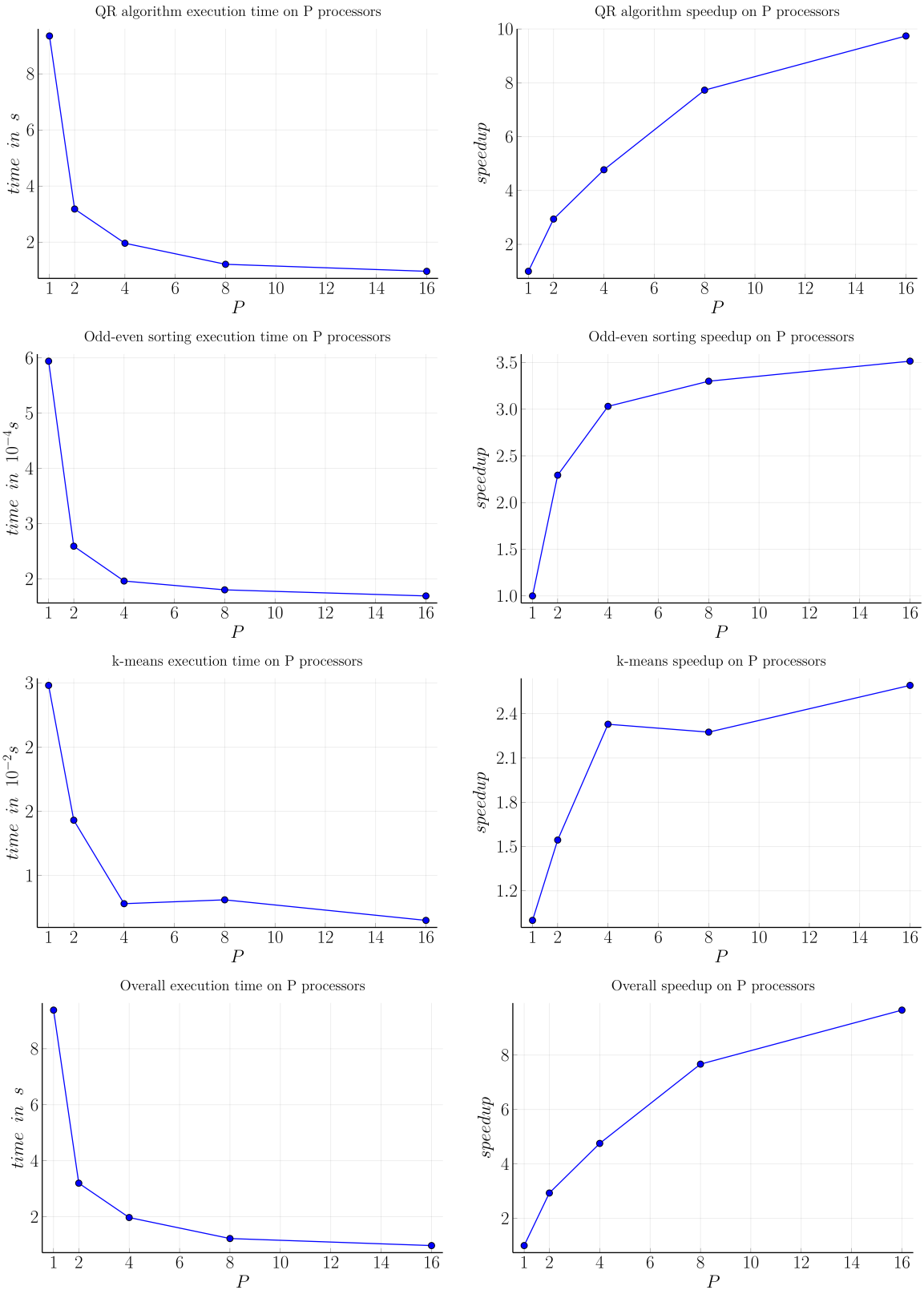

Figure 3: Experimental execution time and speedup for each subproblem as well as the overall algorithm

4 Applications

4.1 Partitioning the Stockholm Tunnelbana Network

For the first application we partition the Stockholm Tunnelbana network using unnormalized spectral clustering. The corresponding graph is created by assigning a node to each intersection of two lines or more. Two nodes are connected if there is a direct connection between them. The overall timing of such a small problem using $P = 4$ processes on Dardel was in the order of $10^{-2}s$. The original map with the assigned node labels is displayed in figure 4 and the results are visualized in figure 5.
For $k = 2$ the part south of Gullmarsplan is identified as a cluster. Additionally, the section north of Östermalmstorg is separated when increasing k to 3. For $k = 4$ this part is united with the main cluster again but stations south of Liljeholmen and north of Västra Skogen form their own clusters. In this example, the spectral clustering algorithm indeed identifies geographically meaningful clusters. Furthermore, clusters can increase or decrease in size for varying k. Such behavior distinguishes spectral clustering from bisection based partitioning methods.

Figure 4: The Stockholm Tunnelbana network together with the node labelling which was used to create the unnormalized Laplacian (Source: Storstockholms Lokaltrafik, 2022). The dotted lines are still under construction.

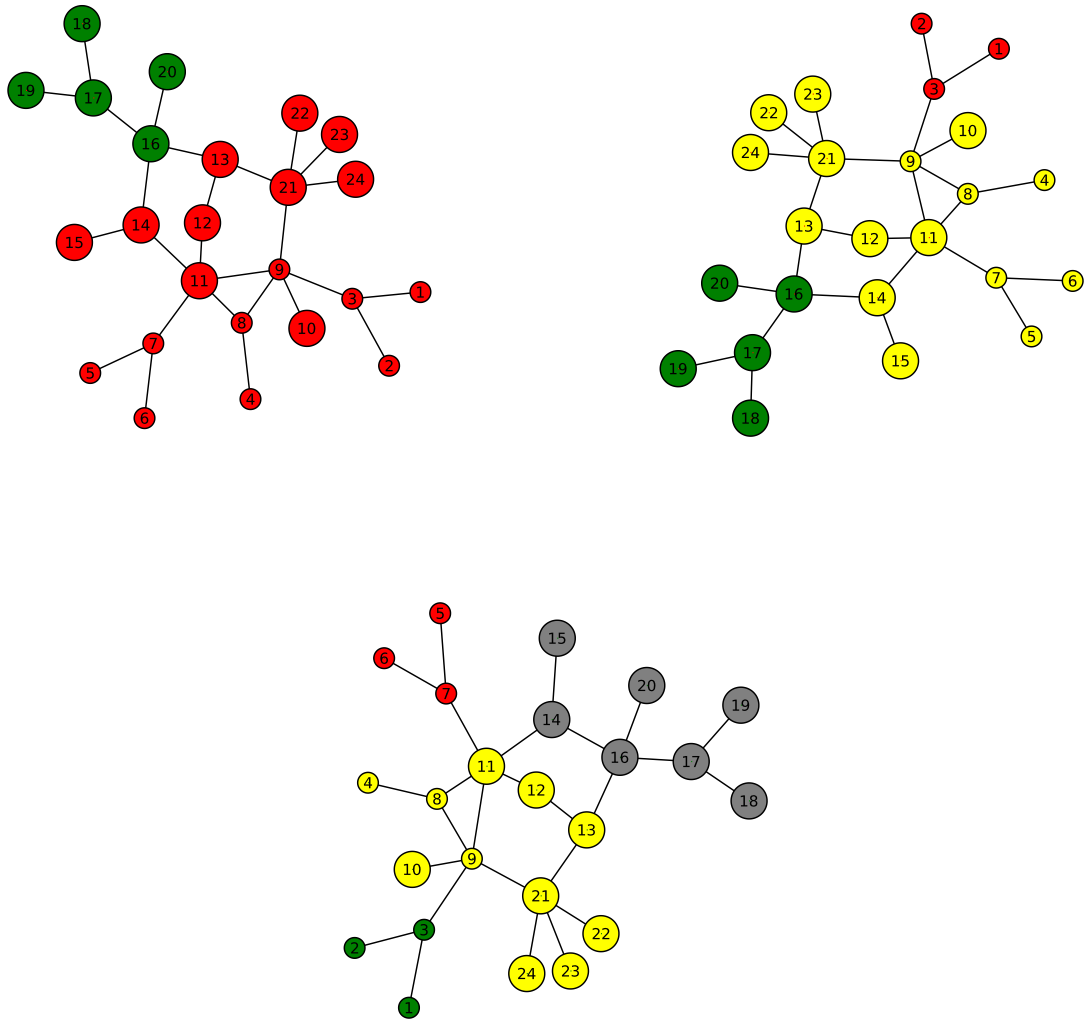

Figure 5: Visualization of spectral clustering with $k = 2, 3, 4$ on the Stockholm Tunnelbana network.

4.2 Partitioning a PDE Mesh

Application two considers the graph partitioning problem for a sparse adjacency graph. The corresponding matrix originates from partial differential equations that occur in thermoelasticity models which were set up by Cerveny et al. [8]. It is symmetric and has size 16129×12129. Figure 7 visualizes the sparsity patterns as well as the clustered adjacency graph for $k = 8, 32$. Further results for different k can be found in the github repository. The sparsity pattern appears to be less separated but the clusters in the adjacency graph are visually clearly distinct. This highlights that spectral clustering is a graph-based method.

The computations were performed on the shared partition of Dardel using $P = 32$ processes within an execution time in the order of 10 minutes. The calculations for a bigger

matrix of size 65025×65025 originating from the same PDE models [8] could not be completed within 30 minutes of the allocated computation time with $P = 32$ processes on the main partition. Thus, we conclude that the current implementation is applicable to smaller sized adjacency graphs with up to 10^4 nodes using 32 processes on Dardel. For matrices of size 10^5 and larger, computing time in the scale of hours together with more memory resources would be required.

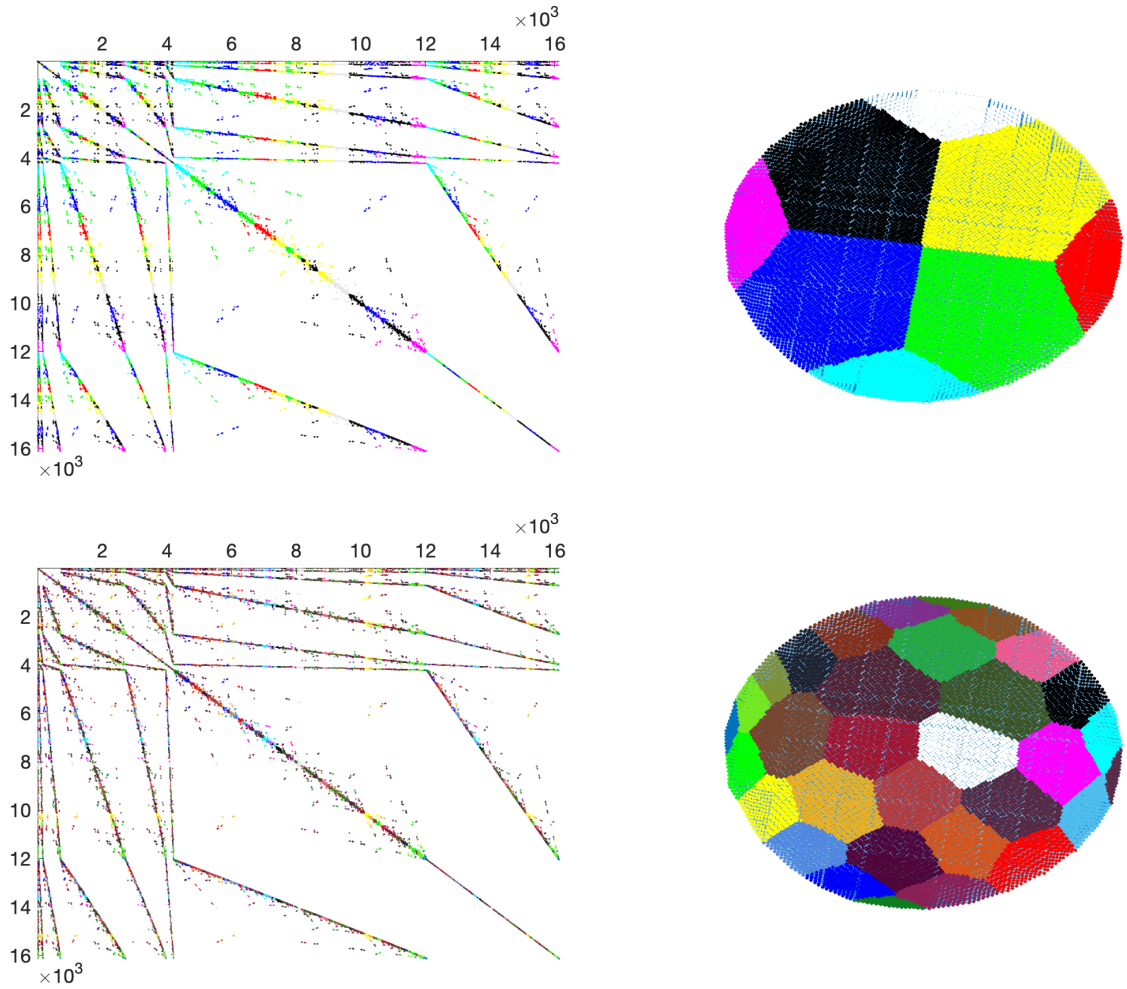

Figure 6: Partitioning of the adjacency graph of a 16129×12129 sparse matrix coming from a PDE model [8] using the parallel implementation of spectral clustering with $k = 8, 32$ clusters. The left images visualize the sparsity patterns with color coding for the different clusters and the right images show the partitioned adjacency graph.

5 Conclusion

In the course of this project the spectral clustering algorithm was successfully parallelized, implemented in C using MPI, benchmarked and applied to two real world problems. It was possible to parallelize almost the entire algorithm which is leads to good parallel speedup as could be demonstrated in the benchmarking section. The performance predictions drawn from the theoretical analysis in chapter 3 match with the experimental results. The complexity of the presented implementation is dominated by the QR method using Gram Schmidt orthogonalization. Apart from having the longest running time, the QR method also features the largest speedup amongst all subproblems. The speedup of the odd-even sorting and the k-means algorithm are limited either by the amount of communication or serial computation.

In order to further optimize the presented implementation, it would be necessary to decrease the runtime of the eigenvector calculation which is the current major bottleneck. A possibility would be to make use of more efficient ways to determine the QR factorization such as Householder transformations and the Hessenberg QR algorithm [2]. This could be further developed into the Arnoldi iteration. While such algorithms feature lower complexity, it might not be possible to fully parallelize them.
Another way to potentially improve the algorithm for large graphs with lower interconnectivity would be to store the Laplacian L through adjacency lists and use sparse matrix-matrix multiplication. Apart from decreasing its runtime, the applicability of the algorithm for very large matrices could be increased by using distributed memory.

In the end, the choice of the preferred implementation will be dependent on the size and sparsity of the graph to be partitioned, the available hardware and the possible allocated time. The advantages of the presented realization are the good parallel speedup, the use of reliable and well-known algorithms for the different subproblems as well as the possibility to apply it to sparse or dense graphs. From the two applications and the benchmarking results we conclude that the current implementation can be applied to smaller or medium sized graphs with up to 10^4 nodes within an execution time in the order of minutes using a smaller number of processors. To be applicable to larger problems it would either require a large number of processes or further optimization of the QR method.

6 Supplementary

6.1 Benchmarking

Processes	Eigenvector calculation	Eigenvalue Sorting	k-means
1	9.35e+00	5.94e-04	2.98e-02
2	3.19e+00	2.09e-04	1.94e-02
4	1.97e+00	1.96e-04	1.28e-02
8	1.21e+00	1.80e-04	1.31e-02
16	0.96e+00	1.69e-04	1.15e-02

Table 1: Execution time in seconds

6.2 Code Availability

The code is available at https://github.com/niclaspopp/SpectralClusteringMPI.

References

[1] Barry Wilkinson, Mark Allen (2005) Parallel Programming, 2nd Edition, Pearson Education

[2] Lloyd N Trefethen, David Bau (1997) Numerical linear algebra, 2nd Edition, Society of Applied and Industrial Mathematics

[3] Christopher M. Bishop (2006) Pattern Recognition and Machine Learning (Information Science and Statistics), 2nd Edition, Springer-Verlag

[4] Gene Howard Golub, Charles F. Van Loan (2013) Matrix Computations, 4th Edition, Johns Hopkins University Press

[5] John G. F. Francis (1961), The QR Transformation, The Computer Journal 4(3), pages 265–271

[6] Ulrike von Luxburg (2007) A Tutorial on Spectral Clustering (preprint), http://arxiv.org/abs/0711.0189

[7] James Fairbanks, Mathieu Besançon, Simon Schölly, Júlio Hoffiman, Nick Eubank, Stefan Karpinski (2021) JuliaGraphs/Graphs.jl: an optimized graphs package for the Julia programming language, https://github.com/JuliaGraphs/Graphs.jl/

[8] J. Cerveny, I. Dolezel, L. Dubcova, P. Karban and P. Solin (2009) Higher-order finite element modeling of electromagnetically driven flow of molten metal in pipe, 2009 International Conference on Electrical Machines and Systems

[9] Niclas Jansson (2021) Parallel Computations of Large-Scale Systems, Lecture notes

[10] Per Gunnar Martinsson (2016) Fast Algorithms for Big Data, Lecture notes

[11] Elias Jarlebring (2021) Numerical Algorithms for Data-Intensive Science, Lecture notes