Computing network centrality measures on fMRI data using fully weighted adjacency matrices

Stefan Bränberg
Abstract

A lot of interesting research is currently being done in the field of neuroscience, a recent subject being the effort to analyze the human brain connectome and its functional connectivity. One way this is done is by applying graph-theory based network analysis, such as centrality, on data from fMRI measurements. This involves creating a graph representation from a correlation matrix containing the correlations over time between all measured voxels. Since the input data can be very big, this results in computations that are too memory and time consuming for an ordinary computer. Researchers have used different techniques to work around this problem, examples include thresholding correlations when creating the adjacency matrix and using a smaller input data with lower resolution.

This thesis proposes three ways to compute two different centrality measures, degree centrality and eigenvector centrality, on fully weighted adjacency matrices that are built from complete correlation matrices computed from high resolution input data. The first is reducing the problem by doing the calculations in optimal order and avoiding the construction of the large correlation matrix. The second solution is to distribute and do the computations in parallel on a large computer cluster using MPI. The third solution is to calculate as large sets as possible on an ordinary laptop using shared-memory parallelism with OpenMP. Algorithms are presented for the different solutions, and the effectiveness of the implementations of them is tested.
Acknowledgements

I wish to thank Anders Wåhlin for presenting me with a very interesting thesis subject, and for having an enthusiasm for that subject that has kept me motivated all the way.

I also want to thank my supervisor Jerry Eriksson for his support and his guidance through the process that is writing a thesis.

I want to thank my mother for lending me the books on how to write a thesis. I did read some of it, and I hope it rubbed off on me.

To the rest of my family - thank you for always supporting me in all my endeavours.

Last but not least I want to thank the love of my life, Britta, for all the love and patience that she has for and with me.
Contents

1 Introduction 1
  1.1 Centrality 1
    1.1.1 Creating a graph representation 1
    1.1.2 Degree centrality 2
    1.1.3 Eigenvector centrality 2
  1.2 Problem 3
  1.3 Goal 3
  1.4 Related work 4

2 Reducing the problem size 5
  2.1 Eigenvector centrality 5
  2.2 Degree centrality 6

3 Parallelizing the problem 7
  3.1 Parallel computation using MPI 7
    3.1.1 Abisko 7
    3.1.2 Correlation matrix 7
    3.1.3 Degree centrality 10
    3.1.4 Eigenvector centrality 10
  3.2 Parallel computation using OpenMP 13

4 Tests 15
  4.1 Test equipment 15
  4.2 Reduced problem size 15
  4.3 Parallelize 17
    4.3.1 MPI 17
    4.3.2 OpenMP 19
    4.3.3 MPI vs OpenMP 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Conclusions</td>
<td>21</td>
</tr>
<tr>
<td>6 Discussion and future work</td>
<td>23</td>
</tr>
<tr>
<td>References</td>
<td>25</td>
</tr>
</tbody>
</table>
1 Introduction

The studies of the human brain is an area where a lot of interesting research is taking place. Non-invasive methods such as functional magnetic resonance imaging (fMRI) has recently been used to analyse the human brain connectome [1], that is the “network of anatomical connections linking the neuronal elements of the human brain” [2]. fMRI uses measurements of magnetic differences between oxygenated and de-oxygenated blood (blood oxygenation level–dependent contrast, or BOLD) to detect metabolic activity. This is then interpreted as neuronal activity in the brain. Theses measures are made on a number of points in the brain with equal distance from each other with a certain frequency over a period of time. The data from the measurements is then stored as a matrix of volumetric pixels, or voxels, over time. One dimension of the matrix is the number of voxels, and the other is the number of measurements.

One way to study the connectome of the brain is by analysing it in terms of its functional connectivity. Functional connectivity is defined as statistic dependencies among remote neurophysiological events, or in this case the temporal correlation between the measured BOLD levels of the different voxels.

1.1 Centrality

To explore the functional connectivity of the human brain, researchers have begun using graph theory-based network analysis, and one class of graph theory-based network measures that have gained interest is centrality. Centrality assesses the functional importance of the nodes in a graph and allows for capturing the complexity of the whole functional connectome, between each voxel and all others, unlike seed-based approaches that measure the relationship between regions of voxels and all other voxels. The definition of graph centrality and the different measures of centrality has been taken from the article “Network Centrality in the Human Functional Connectome” by Zuo et al [3].

It defines four measures of centrality; degree centrality, eigenvector centrality, subgraph centrality and page-rank centrality. The two first ones will be considered in this thesis.

1.1.1 Creating a graph representation

When computing the centrality of the voxels in a fMRI data set, a graph representation of the data must first be constructed. First a correlation matrix, \( C = (c_{ij}) \in \mathbb{R}^{n \times n} \), is created where the value of each position \( c_{ij} \) is the Pearson correlation between voxel \( i \) and voxel \( j \). If the measured data is normalized to have zero mean and a standard deviation of 1, and it is stored as a matrix \( D = (d_{ij}) \in \mathbb{R}^{n \times m} \) (where \( n \) is the number of voxels and \( m \) is the number...
of measurements), then $C$ is defined as:

$$C = DD^T \frac{1}{m-1}$$

In order to do graph based analysis of the network, an adjacency matrix is then created from the correlation matrix. In a graph adjacency matrix, the values of the matrix represents the weights of the edges between the nodes and are defined as being non-negative [4]. A number of different approaches have been proposed to create an adjacency matrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ with only positive values from a correlation matrix. An often used procedure is to choose a threshold value $c_0$ and set the value for $a_{ij}$ to 0 if $c_{ij} < c_0$ and $c_{ij}$ (or 1 if a binary graph is to be created) if $c_{ij} \geq c_0$ [3] [5].

This approach does not take into account the negative correlations between the voxels, and is removes information from the correlation matrix when forming the adjacency matrix. Zuo et al. [3] mentions that more consistent results might be obtained by using a fully weighted (not thresholded) adjacency graph. To do that Lohmann et al. [4] suggest, instead of thresholding, a rescaling of the correlations by 1 so that

$$a_{ij} = c_{ij} + 1$$

Since this assumes that strong negative correlation is a sign of strong dissimilarity, and not something that points to a negative coupling, they also propose the use of absolute correlations as an alternative when constructing the adjacency matrix.

Other approaches that have been proposed is using the square value of the correlations, sometimes with thresholding, to form the adjacency matrix [6].

The approaches considered in this thesis will two ones; creating an adjacency matrix by using absolute correlation values, and by rescaling the correlation values. This is because it was a clear wish from my external supervisor that I used fully weighted adjacency matrices.

### 1.1.2 Degree centrality

Degree centrality (DC) is the sum of the weight of all edges connecting to a node.

$$DC(i) = \sum_{j=1}^{n} a_{ij}$$

If the graph is a weighted graph, as opposed to a binary one, this measure is sometimes called node strength. Both kinds (binary and weighted) will in this work be referred to as degree centrality. Only the weighted version will be calculated.

### 1.1.3 Eigenvector centrality

The eigenvector centrality of a node $i$ is the $ith$ entry of the eigenvector $\mu_1$ that corresponds to the largest eigenvalue, $\lambda_1$, of the adjacency matrix.

$$EC(i) = \mu_1(i)$$
This measure favours nodes that are connected to other nodes that are central within the network.

A fast and easily implemented way to closely approximate the largest eigenvalue, and corresponding eigenvector, is the power method. The power method is an iterative method where a random vector, \( x \in \mathbb{R}^n \), is first chosen. At every iteration the vector is multiplied with the matrix and normalized:

\[
x_{k+1} = \frac{Ax_k}{||Ax_k||}
\]

where

\[
\lambda_k = ||Ax_k||
\]

converges to the largest eigenvalue \( \lambda_1 \).

The iteration is normally stopped when \( |\lambda_k - \lambda_{k-1}| < t \) for some set error margin \( t \) that is the value of difference that is allowed between the approximated eigenvalue and the exact one.

1.2 Problem

The data considered in this thesis is measurements made on patients at the University Hospital of Umeå. It is available in two different resolutions, 2 mm and 4 mm sized voxels measured 170 times, resulting in matrix sizes of \( \sim 160000 \times 170 \) and \( \sim 20000 \times 170 \). Computing a full correlation matrix for the larger one of these data sets results in a matrix of size \( 160000 \times 160000 \) (\( \sim 190 \) GB), and the time complexity for this computation, when the basic matrix multiplication algorithm is used, is \( O(mn^2) \)

where \( n \) is the number of voxels and \( m \) is the number of measurements.

This results in a computation that is both too memory and time consuming for an ordinary computer. Researchers have so far most often used both thresholding of the correlation matrix, and a only a smaller input ([3] uses a voxel size of 4 mm), to keep both the size of the results and the time computing them down to manageable levels.

1.3 Goal

The goal of this thesis is to implement and evaluate possible solutions to the problem of computing different measures of centrality on large datasets from fMRI scans, without thresholding the correlation matrix. Three different paths are suggested. The first is reducing the problem by doing the calculations in optimal order and thereby avoiding the construction of the large correlation matrix. The second solution is to distribute and do the computations in parallel on a large computer cluster, thereby avoiding the need for the whole correlation matrix to be saved in one place. The third solution is to calculate as large
sets as possible on an ordinary laptop using shared-memory parallelism.

Things to consider will be:

1. Which calculations can be done without constructing the full correlation matrix?
2. Are the parallel solutions scalable with the number of cores/threads used?
3. How do the two parallel solutions compare when run on the same platform?

All solutions will be tested with existing measured data and evaluated in terms of efficiency.

1.4 Related work

There are a number of program libraries that can be used for computations on functional networks and some are even especially made for computing functional connectivity in fMRI data, for example CONN [7]. Another example is NetworKit [8] [9], a network analysis toolkit.

None of the toolkits or libraries that were found was specifically made for calculating centrality on fully weighted adjacency matrices from fMRI input data. CONN states in its functions that the input should be a thresholded correlation matrix, and it lacks the option for computing eigenvector centrality. NetworKit has a wide range of centrality measures that can be computed, but uses a shared memory parallel programming and is therefore limited in scalability, and it works on already created graphs and does not have any tool for creating a fully weighted graph representation from fMRI data.

Other tools, such as the Brain Connectivity Toolbox [10] uses Matlabs own functions for computing centrality, for example eigs to compute eigenvector centrality, and are therefore equivalent to the Matlab methods that are used for comparison later in the thesis.
2 Reducing the problem size

One way of reducing the computational complexity of the calculation of the different centrality measures is to perform the required calculations in an optimal order that avoids forming the large correlation matrix. This cannot be done if the absolute or squared value is to be used, since the matrix multiplication need to be performed in order for us to know the correlation values.

But if the correlation values are to be re-scaled by one, there is a way to calculate the different centrality measures without actually forming the correlation matrix.

2.1 Eigenvector centrality

If we have an adjacency matrix $A \in \mathbb{R}^{n \times n}$, and a vector $x \in \mathbb{R}^n$ with random entries each iteration of the power method for calculating eigenvector centrality is

$$x_{k+1} = \frac{Ax_k}{||Ax_k||}$$

and since $A = C + J$, where $C \in \mathbb{R}^{n \times n}$ and $J$ is a $n \times n$ matrix of ones we can rewrite each iteration as

$$x_{k+1} = \frac{(C + J)x_k}{||(C + J)x_k||}$$

and since $C = DD^T \frac{1}{m-1}$, where $D \in \mathbb{R}^{n \times m}$ is the input data matrix, we can write

$$(C + J)x_k = ((DD^T \frac{1}{m-1}) + J)x_k$$

which, if we introduce a vector $w \in \mathbb{R}^m$ can be written as

$$w = (D^T \frac{1}{m-1})x_k$$

$$x_{k+1} = (Dw) + Jx_k$$

and since every entry in $Jx_k$ is the sum of all elements in $x_k$ we can write, where $y \in \mathbb{R}^n$ and $s \in \mathbb{R}$,

$$y = Dw$$

$$s = \sum_{i=1}^{n} x_k(i)$$
\[ x_{k+1}(i) = y(i) + s \]

This reduces the complexity from \( O(mn^2) \) for computing the correlation matrix and \( O(n^2) \) for each iteration done using the power method, to only \( O(mn) \) for each iteration.

### 2.2 Degree centrality

If we introduce a \( n \) sized vector of ones \( e \), the degree centrality, \( DC \), calculation can be expressed as

\[ DC = eA \]

as in the previous example we can substitute \( A \) for \( (DD^T \frac{1}{m-1}) + J \) and get

\[ DC = e((DD^T \frac{1}{m-1}) + J) \]

this can be divided into

\[ w = eD \]

\[ DC = (w(D^T \frac{1}{m-1})) + (eJ) \]

and since every entry in \( (eJ) \) is \( n \) it can be expressed as a new vector \( q \in \mathbb{R}^n \) where all entries are \( n \), thereby avoiding the need for the \( n \times n \) size \( J \).

\[ DC = w(D^T \frac{1}{m-1}) + q \]

This reduces the complexity from \( O(mn^2) \) for computing and summing the rows, or columns, of the correlation matrix to \( O(mn) \).
3 Parallelizing the problem

Computing the correlation matrix is basically a matrix-matrix multiplication of $D$ and $D^T$, and the power method for eigenvector centrality is in essence a matrix-vector multiplication done iteratively. These are both problems known to be *embarrassingly parallel* [11], meaning that they can be distributed across cores on a multicore system with nearly optimal parallel speedup. Because of this, these computations are perfectly suited for parallelization.

3.1 Parallel computation using MPI

*MPI*, or *message passing interface*, is a library specification primarily used for distributed memory parallel programming [12]. It starts a number of processes and uses message-passing to send data between them, allowing a program to work on a distributed memory system where all processors do not have access to the same memory. MPI will be used for the implementation of the distributed-memory parallel computation of the correlation matrix and the different centrality measures.

3.1.1 Abisko

*Abisko* is a high performance computer cluster that resides at Umeå University and is run by High Performance Computing Center North (HPC2N), a national center for scientific and parallel computing [13]. Abisko consists of 332 computer nodes that each have 4 processors with 12 cores each. 10 of these nodes are considered ’fat’ nodes, and have a RAM of 512 GB, and the rest are considered ’thin’, and have 128 GB RAM each [14].

Since Abisko is a distributed-memory system it is an ideal candidate for performing MPI-based calculations, and will be used for testing the MPI-based algorithms.

3.1.2 Correlation matrix

It was earlier concluded that the full correlation matrix will be ~ 190 GB in size, and we can clearly see that it will not fit on one node unless we are using the ’fat’ ones. One solution to this problem is do distribute the matrix product across several nodes and do the centrality computations on the distributed matrix.

A way of doing this is to use *MPI* to *broadcast* the input matrix $D$ to a number of processes and let each of them compute a part, $local_C$, of the correlation matrix (as shown in Algorithm 1). The matrix product will be divided evenly in a *block* distribution where every process computes $\frac{\text{rows}}{\text{processes}}$ number of rows. If the number of rows is not evenly dividable with the number of processes, the last one will get the extra rows. The number of rows for each process is stored in an array, $size\_array$. 
Algorithm 1: Simple full correlation

1. process 0: broadcast D to all processes;
2. all processes:
3. for $i = 1$ to $size_{array}[my\_rank]$ do
4.   for $j = 1$ to rows do
5.     $mult\_sum = 0$;
6.     for $k = 1$ to cols do
7.       $mult\_sum = mult\_sum + D[(my\_rank \times size\_array[1]) + i][k] \times D[j][k]$;
8.     end
9.     $local\_C[i][j] = mult\_sum$
10. end
11. end
12. return $local\_C$

This computes the full correlation matrix, but since the result is a symmetric matrix ($c_{ij} = c_{ji}$ and $c_{ii} = 1$) we only need to compute the lower half of it to have all the information that we need, an operation that in theory should take roughly half the time taken to compute the full correlation matrix. This does however introduce a distribution problem. If the correlation matrix is distributed evenly row-wise between the processes, the last one will do much more work than the first one. This is due to the fact that the number of elements in a triangle with height $h$ is $T_h = \frac{h(h+1)}{2}$. This leads to each processor having $T_{my\_rank} - T_{my\_rank-1}$ elements to compute, a number that grows exponentially in size.

To counteract this, we can use the fact that $\frac{h(h+1)}{2} \approx \frac{h^2}{2}$ for large $h$ and distribute rows according to Algorithm 2. Each local matrix then has an integer $chunk\_size$, that is the number of rows assigned, an integer $displ$, that is the number of rows before the first row of the local matrix in the full correlation matrix, and an array $index\_array$, that stores how many elements are in the matrix before the start of each row. A temporary row displacement array is used for the calculations, since the different processes does not know what the displacement of the others are.

The matrix is stored as an array in row-major order, so each element $c_{ij}$ is accessed using the index array $C[index\_array[i] + j]$.

The index array is calculated as in Algorithm 3.
Algorithm 2: Triangular distribution of rows

```
1 all processes:
2 total = 0;
3 elements = rows(rows + 1)/2;
4 for rank = 1 to number_of_processes do
5     if rank < number_of_processes then
6         displ_array[rank] = total;
7         temp_size = \sqrt{\frac{2 \cdot rank \cdot elements}{num_processes} - total} ; // rounded to integer
8     if rank==my_rank then
9         chunk_size = temp_size;
10     end
11     total = total + temp_size;
12     else
13         if rank==my_rank then
14             chunk_size = rows - total;
15         end
16     end
17 end
```

Algorithm 3: Calculating index array

```
1 all processes:
2 index_array[1] = 0;
3 for i = 2 to chunk_size do
4     index_array[i] = displ + (i - 1) + index_array[i - 1];
5 end
```

The correlation matrix is then calculated as in Algorithm 4. Since this calculation is done completely in parallel it should theoretically achieve near optimal speedup, i.e. the computation time should be halved when twice the amount of processor cores are used.

Algorithm 4: Lower triangle correlation

```
1 process 0: broadcast D to all processes;
2 all processes:
3 for i = 1 to chunk_size do
4     for j = 1 to displ + i do
5         mult_sum = 0;
6         for k = 1 to cols do
7             mult_sum = mult_sum + D[displ + i][k] \times D[j][k];
8         end
9         mult_sum = mult_sum/(cols - 1);
10         local_C[index_array[i] + j] = mult_sum
11     end
12 end
13 return local_C;
```
3.1.3 Degree centrality

For a fully computed correlation matrix, this is a trivial computation. One simply has to calculate the sum of each row, and since the whole rows are stored on each process this can be done with two nested for loops. The MPI function MPI_Gatherv is then used to collect the different local sums into one resultant vector.

For the lower triangle version, Algorithm 5 can be used.

Algorithm 5: Lower triangle degree centrality

```
1 all processes:
2 for i = 1 to chunk_size do
3     sum_row = 0;
4     for j = 1 to displ + i do
5         sum_row = sum_row + local_C[index_array[i] + j];
6         if j < displ + i then
7             local_degree[j] = local_degree[j] + local_C[index_array[i] + j];
8         end
9     end
10    local_degree[displ + i] = sum_row
11 end
12 MPI_Gather from local_degree on each process to result_degree on process 0;
13 return result_degree;
```

This is also a completely parallel operation and should achieve near optimal speedup.

3.1.4 Eigenvector centrality

The full matrix approach to calculating eigenvector centrality can be seen in Algorithm 6. First a vector with random entries, x_vector, is created and broadcasted to all processes. The vector is multiplied with each local part of the matrix, followed by a summation of the squares of the local vectors values. The local sums are summed together and sent to all processes with MPI_Allreduce, and \( \lambda_k \) is calculated. The local vector is normalized and gathered to a result vector on process 0. If \( |\lambda_k - \lambda_{k-1}| \) is larger than a set error margin the result vector is broadcast and the process repeated.
Algorithm 6: Full correlation matrix eigenvector centrality

1  process 0: create random x\_vector;
2  all processes:
3     do
4         broadcast x\_vector to all;
5         for i = 1 to size\_array[my\_rank] do
6             local\_vector[i] = 0;
7             for j = 1 to rows do
8                 local\_vector[i] = local\_vector[i] + local\_C[i][j] \times x\_vector[j];
9         end
10     end
11  local\_sum = 0;
12  global\_sum = 0;
13  for k = 1 to size\_array[my\_rank] do
14      local\_sum = local\_sum + local\_vector[k] \times local\_vector[k];
15  end
16  allreduce local\_sum to global\_sum;
17  old\_lambda = lambda;
18  lambda = \sqrt{global\_sum};
19  diff = |old\_lambda - lambda|;
20  for l = 1 to size\_array[my\_rank] do
21      local\_vector[l] = local\_vector[l]/lambda;
22  end
23  gather local\_vector to x\_vector on process 0;
24  while diff \textgreater error\_margin;
25  result\_degree = x\_vector;
26  return result\_degree;

When only the lower part of the matrix is stored, we need two iterations, as in Algorithm 7, first one that multiplies the vector with matrix row-wise and then one that does the same column-wise. This is because the columns have the same values as the rows beyond the diagonal in the full matrix, due to it being symmetrical.
Algorithm 7: Lower triangle correlation matrix eigenvector centrality

1. process 0: create random \texttt{x_vector};
2. all processes:
   3. do
      4. process 0: broadcast \texttt{x_vector} to all;
      5. for \(i = 1\) to \texttt{chunk\_size} do
         6. \texttt{local\_vector}[i] = 0;
         7. for \(j = 1\) to \texttt{displ} + \(i\) do
            8. \texttt{local\_vector}[\texttt{displ} + \(i\)] = \texttt{local\_vector}[\texttt{displ} + \(i\)] + \texttt{local\_C}[\texttt{index\_array}[i] + j] \cdot \texttt{x\_vector}[j];
         9. end
      10. end
      11. for \(i = 1\) to \texttt{displ} + \texttt{chunk\_size} do
         12. \texttt{local\_vector}[i] = 0;
         13. if \(i \leq \texttt{displ}\) then
            14. for \(j = 1\) to \texttt{chunk\_size} do
               15. \texttt{local\_vector}[i] = \texttt{local\_vector}[i] + \texttt{local\_C}[\texttt{index\_array}[j] + i] \cdot \texttt{x\_vector}[\texttt{displ} + j];
            16. end
         17. else
            18. for \(j = i - \texttt{displ}\) to \texttt{chunk\_size} do
               19. \texttt{local\_vector}[i] = \texttt{local\_vector}[i] + \texttt{local\_C}[\texttt{index\_array}[j] + i] \cdot \texttt{x\_vector}[\texttt{displ} + j];
            20. end
         21. end
      22. end
      23. reduce \texttt{local\_vector} to \texttt{x\_vector} on process 0;
      24. scatter \texttt{x\_vector} to \texttt{sum\_vector} on all processes;
      25. \texttt{local\_sum} = 0;
      26. \texttt{global\_sum} = 0;
      27. for \(k = 1\) to \texttt{scatter\_size} do
         28. \texttt{local\_sum} = \texttt{local\_sum} + \texttt{sum\_vector}[k] \cdot \texttt{sum\_vector}[k];
      29. end
      30. allreduce \texttt{local\_sum} to \texttt{global\_sum};
      31. \texttt{old\_lambda} = \texttt{lambda};
      32. \texttt{lambda} = \sqrt{\texttt{global\_sum}};
      33. \texttt{diff} = |\texttt{old\_lambda} - \texttt{lambda}|;
      34. for \(l = 1\) to \texttt{scatter\_size} do
         35. \texttt{sum\_vector}[l] = \texttt{sum\_vector}[l] / \texttt{lambda};
      36. end
      37. gather \texttt{sum\_vector} to \texttt{x\_vector} on process 0;
      38. while \texttt{diff} > \texttt{error\_margin};
      39. \texttt{result\_degree} = \texttt{x\_vector};
      40. return \texttt{result\_degree};

This calculation is also done almost entirely in parallel, and should in theory achieve near optimal speedup. There can however be a small overhead introduced by the message pass-
3.2 Parallel computation using OpenMP

OpenMP is an API for shared-memory parallel programming that uses preprocessor directives called pragmas to define its behaviour [15]. Its ease of use makes it a good candidate for shared-memory programming.

OpenMP has a pragma that divides the workload of a for loop between different threads in a thread pool. To accomplish this (in C) one simply adds

```
#pragma parallel for
```

before the for loop. This starts a number of threads and divides the loop so that each thread handles a different portion of it. For the algorithms in this thesis it consists of adding this pragma to the outermost loops of all calculations, and ignoring all MPI sends and receives. The updates of local degree in the degree centrality algorithm (Algorithm 5 rows 7 and 9) are also preceded with an atomic pragma that ensures that two threads can’t update an array entry at the same time.

Since all processors are using the same memory, the if-else inside the for loop of Algorithm 7 is not needed. The result is Algorithm 8. The OpenMP implementations should theoretically achieve the same amount of speedup as the MPI ones, except for the difference that the overhead introduced by OpenMP differs from the overhead introduced by MPI. In MPI it is mainly the message passing that adds extra time, but in OpenMP there is an extra cost of starting new threads and the possibility for sub-optimal scheduling of the for loops. The fact that threads can be waiting for each other when trying to write to the same variable in the degree centrality calculation can also be a factor.
Algorithm 8: Lower triangle correlation matrix eigenvector centrality with OpenMP

1. process 0: create random $x_{\text{vector}}$;
2. do
3. parallel for $i = 1$ to rows do
4.   $sum_{\text{vector}}[i] = 0$;
5.   for $j = 1$ to $i$ do
6.     $sum_{\text{vector}}[\text{displ} + i] = sum_{\text{vector}}[\text{displ} + i] + C[index_{\text{array}}[i] + j] \cdot x_{\text{vector}}[j]$;
7.   end
8. end
9. parallel for $i = 1$ to rows do
10.   $sum_{\text{vector}}[i] = 0$;
11.   for $j = i$ to rows do
12.     $sum_{\text{vector}}[i] = sum_{\text{vector}}[i] + C[index_{\text{array}}[j] + i] \cdot x_{\text{vector}}[j]$;
13. end
14. end
15. $x_{\text{sum}} = 0$;
16. parallel for $k = 1$ to rows do
17.   $x_{\text{sum}} = x_{\text{sum}} + sum_{\text{vector}}[k] \cdot sum_{\text{vector}}[k]$;
18. end
19. $old_{\text{lambda}} = \text{lambda}$;
20. $\text{lambda} = \sqrt{x_{\text{sum}}}$;
21. $diff = |old_{\text{lambda}} - \text{lambda}|$;
22. parallel for $l = 1$ to columns do
23.   $x_{\text{vector}}[l] = sum_{\text{vector}}[l]/\text{lambda}$;
24. end
25. while $diff > error_{\text{margin}}$;
26. result_degree = $x_{\text{vector}}$;
27. return result_degree;

The work load of the OpenMP loops can be scheduled in a couple of different ways [16]:

1. static - Statically divides the work in as equal chunks as possible to the different threads in a round-robin fashion.
2. dynamic - Dynamically allocates chunk sized blocks of loop iterations from an internal work queue. When a thread is done with its assigned iterations it picks the next block of loops from the queue.
3. guided - Also assigns loops from a work queue, but the chunk sizes start of large and decrease in size for each assignment.
Time measurements have been done with different sized input matrices to tests the efficiency of the algorithms. The data was provided in two different resolutions, 4 mm between each voxel (19955x170 matrix size) and 2 mm (159570x170 matrix size). Two more input matrices, here called 2 mm/2 and 2 mm/4, were constructed by taking the every second, or fourth, elements from the 2 mm matrix.

4.1 Test equipment

All MPI tests, unless otherwise stated, were performed on Abisko, and the other tests were performed on a test laptop with a 2.3 GHz 4-core CPU and 8 GB RAM running Ubuntu 14.04.

4.2 Reduced problem size

The computation of the two measures of centrality was implemented as Matlab functions and tested on the test laptop. The functions will here be referred to as scaled_degree and scaled_eigen. Matlab's own functions tic and toc, were used to measure the time.

The tests that were run were:

1. Degree centrality time.
2. Eigenvector centrality time with different $\lambda$ error margins.

The runtimes for scaled_degree for the different inputs are shown in Figure 1. The only input where degree centrality could be computed with Matlab using corr without creating a correlation matrix larger in memory than the available RAM size was the 4 mm resolution matrix. The measured times vary greatly when using the Matlab functions, from $\sim$280 seconds when running the function the first time since starting Matlab to $\sim$10 seconds when the function has been run a few times. This is likely due to how Matlab handles variables in memory. Worth noting is that the first 4-5 runs always took more than 100 seconds.

The run-time of the calculation of eigenvector centrality with scaled_eigen on the input with 2 mm resolution was measured with different error margins and can be seen in Figure 2. The error margin is the minimal allowed difference between the calculated $\lambda$-values when
Resolution of input data

Figure 1: Times measured when computing degree centrality with different inputs using the power method for finding the largest eigenvalue and its corresponding eigenvector. $\varepsilon$ is the distance from 1.0 to the next larger double-precision number in Matlab.

Again only the smallest input could be used for calculating centrality with Matlab, this time using the functions `corr` and `eigs`. The times were varying ranging from $>100$ seconds to $\sim20$ seconds.

Figure 2: Times measured when calculating eigenvector centrality with different $\lambda$ error margins, with the Matlab solution.
When running the `scaled_eigen` function with $\varepsilon$ as input, the values of the resulting eigenvector, calculated from the input with 4 mm precision, differ by at most $4.58 \times 10^{-16}$ from the values calculated by using Matlab's `eigs`. This is a high accuracy and since there is no large computation cost, $\varepsilon$ could easily be used every time when running the function.

4.3 **Parallelize**

The parallel algorithms have been implemented in C as command-line programs using MPI and OpenMP, and different options, such as eigenvalue error margin and input file name, can be set using command-line arguments.

4.3.1 **MPI**

The tests performed on the MPI solution were:

1. Comparing the naive solution to the solution that only computes the lower triangle of the correlation matrix.
2. Eigenvector centrality calculated with different error margins.
3. Scalability of the different algorithms.

The first test that was run was a comparison of the time taken by the naive version, that computes the whole correlation matrix, versus the time taken by the algorithm that only computes the lower half of the correlation matrix. Figure 3 shows the times measured with the full 2 mm input using 48 cores. These tests were run on 'fat' nodes on Abisko, the rest were run on 'thin' nodes. The version that only computes the lower triangle of the correlation matrix was considerably faster, and that version was chosen for the rest of the tests.

![Figure 3: Times measured when computing correlation matrix on full 2 mm input matrix with 24 cores on abisko.](image-url)
Figure 4 shows the calculation times for eigenvector centrality measured on both the test laptop and Abisko. Here it can be seen that the runtime is growing linearly with the number of decimals.

![Figure 4: Times measured when calculating eigenvector centrality using MPI, with different λ error margins. The laptop results were measured with a smaller input (every fourth voxel of the 2 mm resolution data was used as input), and a smaller number of cores (4), than the one used when testing on Abisko (here the full 2 mm input was used and the computation was run on 48 cores)](image)

The scaling of the algorithms was measured with four different numbers of cores and can be seen in Figure 5.

![Figure 5: Times measured when computing correlation matrix and centrality on Abisko with different number of cores. The number of iterations for the eigenvector calculation was set to 20 for a more exact comparison. Input was the full 2 mm resolution matrix.](image)
The algorithms show strong scalability, since the calculation times roughly halves when the amount of used cores is doubled.

### 4.3.2 OpenMP

The tests performed on the OpenMP implementation were:

1. Calculation times with different scheduling schemes
2. Scalability of the different calculations

All test were done with a solution that only uses the lower triangle of the correlation matrix. This is due to the fact that memory is more of an issue on the test laptop than on Abisko. The largest of the inputs that can be run on the laptop is the one were every fourth voxel in the full 2 mm resolution input is used (39893 voxels). The size of the correlation matrix for this input is $\sim 6.5$GB and fits just about in the memory of the test laptop.

The calculation times obtained when using the different OpenMP scheduling can be seen in Figure 6. Since the eigenvector centrality algorithm contains several for loops, a version that utilises both dynamic and guided scheduling was introduced. The dynamic and guided scheduling perform consistently better than the static one, and the dynamic one performed best when computing eigenvector centrality. The fastest schedule for each computation was used in the other tests.

**Figure 6:** Times measured when computing the correlation matrix and centrality with different OpenMP scheduling on the test laptop. The number of iterations of the eigenvector centrality calculation was set to 15. Input was the matrix created by taking every fourth voxel of the 2 mm input matrix (39893 voxels).

Figure 7 shows the calculation times of the different algorithms using different number of threads. The correlation matrix and degree centrality calculations are strongly scalable, since the calculation times are roughly (even though slightly less than) halved when using double the amount of threads. The eigenvector centrality calculation is not as strongly scalable, and this can be due to either sub-optimal scheduling, or some by OpenMP introduced overhead, for example starting new threads each time the parallel for loops are run.
4.3.3 MPI vs OpenMP

Some tests were conducted to see which one of the parallel solutions that was faster. These tests were conducted on the test laptop with each implementation using 4 threads, or processes. Figure 8 shows the times for these tests. The OpenMP implementation performs slightly better when calculating the correlation matrix, but the MPI implementation is more efficient at both centrality calculations. This again points to some overhead being introduced in the OpenMP implementations that the MPI implementations does not have. The degree centrality calculation for OpenMP uses two atomic directives where a thread might have to wait to update a variable with a value, this might also slow down the computation.

Figure 7: Times measured when computing the correlation matrix and centrality on the test laptop with different number of threads. The number of iterations for the eigenvector centrality calculation was set to 15 for a more exact comparison. Input was the matrix created by taking every fourth voxel of the 2 mm input matrix (39893 voxels).

Figure 8: Times measured when computing the correlation matrix and centrality on the test laptop with OpenMP and MPI. The number of iterations for the eigenvector centrality calculation was set to 15. Input was the matrix created by taking every fourth voxel of the 2 mm resolution input matrix (39893 voxels), and the 4 mm resolution input matrix (19955 voxels).
5 Conclusions

All three proposed solutions are viable alternatives for use when calculating degree and eigenvector centrality. If the rescaled version is desired, or another solution that does not need the calculation of the correlation matrix, one can use the Matlab version, or a modified version of it, with good results. It is however limited in that no other ways of constructing the adjacency matrix can be used.

The MPI version scales well with the number of cores used, and is the version that might be needed for future work in the cases where different ways of constructing the adjacency matrix is to be explored. The data sizes of fMRI measurement are steadily growing, and this solution is viable for even larger data sets than the ones used in this thesis.

The OpenMP version showed slightly worse scalability than the MPI one, but is all the same a viable solution for use with a personal computer, since no extra libraries need to be installed in order for a program to run, as is the case with MPI.

The resolution of the fMRI measurements, and thereby the resulting centrality resolution, used plays a crucial part in the exploration of the brain by neuroscientists. Figure 9 shows the difference in resolution of scaled eigenvector centrality when using the different provided input resolutions. It is clear that the higher input resolution produces a much more detailed picture, and that as high resolution as possible is desired.
Figure 9: Cross sections showing eigenvector centrality of a human brain. Above is 4 mm input resolution and below is 2 mm. Yellow colour indicate high centrality.
Due to the time limit of a 15 hp thesis, only the easiest and most straightforward way of multiplying matrices was used. There are most probably a lot of tweaking of the implementations that can be done, especially in order to optimize cache memory access and other similar improvements. The focus of the parallel solutions in this work has been to lower the memory usage by only calculating the lower part of the correlation matrix, and that has been done.

Future work might be to explore more efficient solutions for matrix multiplication and eigenvector problems, for example ScaLAPACK (Scalable Linear Algebra PACKage) [17] or other linear algebra libraries.

The thesis does not take into account the thresholding of a correlation matrix into an adjacency matrix. If that would have been asked for, a solution that works with sparse matrices might have been a better one.

The scheduling tests performed with OpenMP only used the default chunk size, this is due to the fact that all other chunk sizes seemed to decrease performance, but further studies into that could be done.
References


