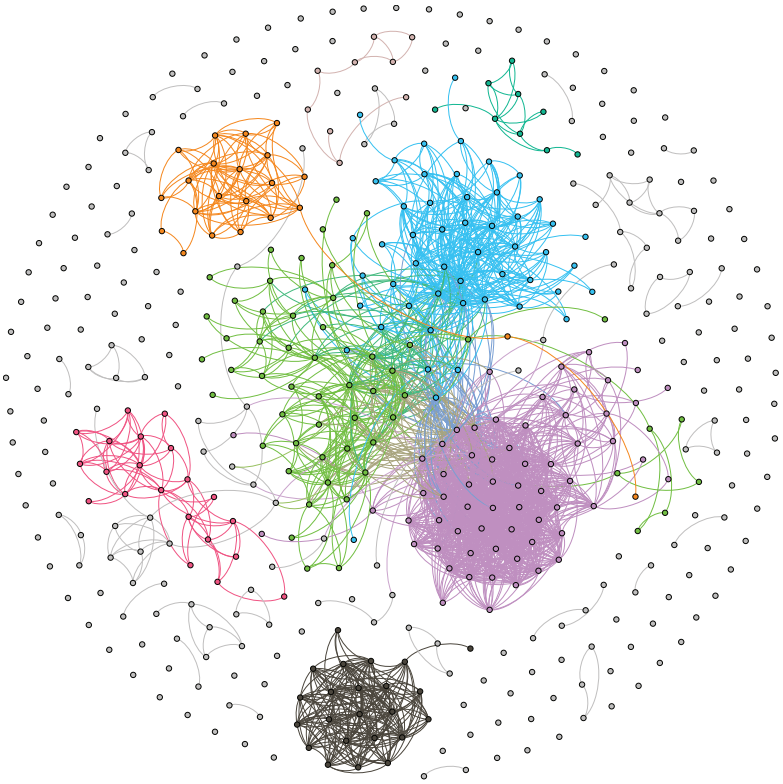
Community detection in parallel

Parallelizing Newman's Leading Eigenvector algorithm

CAB401 – High Performance and Parallel Computing

Author
David Stewart (n7540043)

Friday, 22 October (Semester 2, 2020)



Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Compilation Instructions & Running

Development took place in a `WSL2` environment for Windows 10.

1.1 Development Tools

- `GCC v 7.5.0`;
- OpenMP;
- MATLAB;
- Python 3 (with `numpy`);
- Godbolt compiler explorer (for exploring vectorisation methods);
- valgrind (bug hunting);
- cachegrind (investigating optimisations);

2 The sequential application

2.1 A description of the algorithm, and what we seek to find

The application to be parallelised is a Python 3 implementation of Newman's Leading Eigenvector Algorithm for modularity maximisation ('Newman's Algorithm') (Newman, 2006). The application is one I wrote to implement Newman's modularity algorithm for analysis on some synthetic data. It is hosted publicly at https://github.com/davidjmstewart/ParallelNewmanModularity. A thorough explanation of the algorithm is outside of the scope of this paper, however supplementary materials have been submitted alongside this report for the interested reader. Nonetheless, a high level overview of the algorithm is laid out below.

The task of maximising modularity is a key component in community detection within networks. Networks are described by Newman as "sets of nodes or vertices joined in pairs by lines or edges" (Newman, 2006, p. 103). Figure 1 shows two typical representations of a graph/network: In a directed network, an edge (link) is typically drawn with an arrow head to indicate the direction of connection. e.g. in Figure 1, Node 1 is connected to 2, but 2 is not connected to 1. In an undirected network, the existence of an edge indicates that both endpoints are connected to one another, i.e. Node 1 is connected to Node 2, and Node 2 is connected to Node 1.

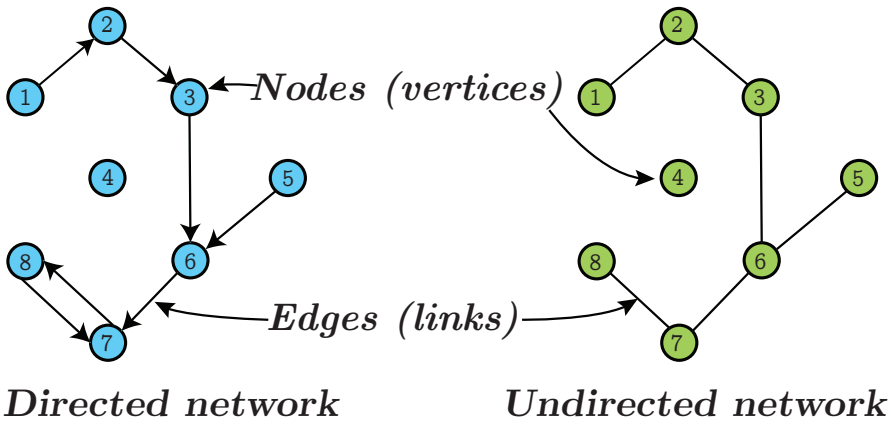


Figure 1: Two typical network types: Direct (left) and undirected (right)

These networks can be conveniently expressed in matrix notation, using the convention of an adjacency matrix (referred to in the submitted code and Newman's Paper as $\boldsymbol{A}$). Adjacency matrices are square and typically contain only the values 1 and 0, used to indicate a truth value regarding the connection of 2 nodes. $\boldsymbol{A}_{ij} = 1$ iff there is a connection between Node i and Node j. For a directed network, this means an arrow can be drawn pointing *from* Node i *to* Node j. For an undirected network, this is simply a line drawn between the 2 nodes. The adjacency matrices for the network in Figure 1 would thus be:

$$\underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}}_{\text{Directed network}} \quad \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}}_{\text{Undirected network}}$$

The adjacency matrix of a given network forms the basis of the algorithm.

The aim of Newman's Algorithm is to find a communities of nodes in the network that are unlikely to exist due to random chance. How well these nodes can be clustered is described by the modularity score Q.

2.2 Algorithm steps

The bulk of the algorithm is laid out in a recursive manner in Algorithm 2 below. Algorithm 1 shows how the input B for Algorithm2 is computed. This computation is discussed as part of Appendix 7.1 as it is not a computational bottleneck in the application.

Algorithm 1 Steps of Newman's Algorithm

Input: $\boldsymbol{A}$: An $n \times n$ (square), symmetric adjacency matrix describing the network
$D \leftarrow \langle \boldsymbol{A}, \boldsymbol{A}^T \rangle$ ▷ Create a vector that holds the degree of each node
$m \leftarrow \sum_{i=0} D_i$ ▷ Total number of edges in the network
$\boldsymbol{K} \leftarrow D^T \times D$ ▷ $\boldsymbol{K}_{ij}$ = number of ways A node with the same degree as Node i could be connected to a node with the same degree as Node j
$\boldsymbol{B} \leftarrow \boldsymbol{A} - \boldsymbol{K}\frac{1}{2m}$ ▷ Compute modularity matrix: $\boldsymbol{B}_{ij}$ is the difference between the actual number of connections between i and j and how many you would expect based on random chance
End Function

Algorithm 2 Newman's leading eigenvector algorithm for modularity maximisation

input : B: An $n \times n$ subgraph modularity matrix (equation 6 in Newman's paper)
: L: An incrementable label (implementation uses integers starting at 0)
: V: Indices of nodes. Nodes have their global index value e.g. a node of value 42 is the 42 node in the network
: L_v A vector of n labels corresponding to the community of each node
: Q: The modularity score of the network (Newman divides this by $4m$)
: L: An incrementable label (implementation uses integers starting at 0)

Function `assignCommunity`(B, L, V)**:**

$B_{ij}^{(g)} \leftarrow B_{ij} - \delta_{ij} \sum_{k \in g} B_{ik}$ ▷ Compute subgraph modularity matrix (equation 6)
$E_{val} \leftarrow$ Largest eigenvalue of $B^{(g)}$
$E_{vec} \leftarrow$ Eigenvector corresponding to E_{val}
$s_i \leftarrow 1$ if $E_{vec}(i) \geq 0$, else -1
$\Delta Q \leftarrow s \cdot B^g \cdot s$
if $\Delta Q \leq 0$ **then**
  $L_v \leftarrow$ Label vertices in V with label L
  Increment L
  $\Delta Q \leftarrow 0$
else
  $V_1 \leftarrow$ Vertices in V corresponding to $s_i == 1$
  $V_2 \leftarrow$ Vertices in V corresponding to $s_i == -1$
  $[L_{v1}, L, \Delta Q_1] \leftarrow$ `assignCommunity`$(s_i == 1, L, V_1)$
  $[L_{v2}, L, \Delta Q_2] \leftarrow$ `assignCommunity`$(s_i == -1, L, V_2)$
end
$\Delta Q \leftarrow \Delta Q + \Delta Q_1 + \Delta Q_2$
$L_v \leftarrow$ concatenate/join L_{v1} and L_{v2}
return L_v, L, ΔQ

End Function

2.3 Sequential implementation

The python implementation of the `assignCommunity` function can be found in Appendix 7.2.

Profiling results of the algorithm run on a 1000 x 1000 adjacency matrix reveal that the majority of time is, unsurprisingly, spent evaluating the eignevectors of the modularity matrices. Figure 2 contains the sequential profiling results.

```
         2085764 function calls (2082703 primitive calls) in 15.767 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.015    0.015   15.767   15.767 communities.py:1(<module>)
    271/1    0.044    0.000   13.806   13.806 communities.py:8(communities)
      271   13.713    0.051   13.727    0.051 linalg.py:1182(eig)
        1    0.000    0.000    1.756    1.756 npyio.py:803(loadtxt)
        2    1.161    0.581    1.710    0.855 npyio.py:1041(read_data)
  1000000    0.421    0.000    0.477    0.000 npyio.py:771(floatconv)
        1    0.004    0.004    0.188    0.188 __init__.py:106(<module>)
        5    0.007    0.001    0.150    0.030 __init__.py:1(<module>)
```

Figure 2: `cProfile` profiler results

The parallel implementation will seek to parallelise this eigenvector computation, as well as other

computations that are preformed in implementing the algorithm.

2.4 Where is the parallelism?

The Python implementation relies heavily on the `numpy` library for its easy-to-use linear algebra functions (such as the `eig` function for finding eigenpairs). This is also reflected in pseudocode of Algorithm 3. Thus it is not obvious at-a-glance where the exploitable parallelism lies.

The majority of operations in the python code are doing *something* with either vectors or matrices, thus many of the individual lines of code have some level of parallelism that is worth investigating. As we will see in the next section, many of these functions are embarrassingly parallel, but that does not necessarily imply excellent thread-level scalability. There are numerous single lines of code in the Python implementation that are investigated for their worthiness for parallelising, including matrix-vector multiplciation, dot-product calculation and eigenvector computation. See the subdirectories of `benchmarking/` to see other operations that were investigated and parallelised.

3 Parallel Implementation

The sequential application is ported to C and then parallelised. This section will discuss the key computations that are parallelised in the C implementation. Note that there are other sections that have been parallelised in the code, such as computing the subgraph modularity matrix, that are relatively small components of the overall running time and are thus omitted from discussion here. See Appendix 7.1 for a dependency analysis and discussion of the parallelism of the subgraph modularity matrix. The full codebase in C, including the `CDUtils.c` library file is 693 lines of code, a substantial increase from the 104 lines of Python code.

3.1 What parallelism can we exploit?

3.2 Matrix-vector multiplication

Matrix-vector multiplication is a key component of the power iteration algorithm used to calculate eigenvectors and eigenvalues. Every iteration of this algorithm involves calculating a matrix-vector product, thus poor parallelisability of this component would seriously hamper the parallel implementation.

For a given matrix $\boldsymbol{A}$ and vector $\vec{B}$, their product can be calculated as:

$$A\vec{B} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}b_2 + \cdots + a_{1n}b_n \\ a_{21}x_1 + a_{22}b_2 + \cdots + a_{2n}b_n \\ \vdots \\ a_{n1}x_1 + a_{n2}bx_2 + \cdots + a_{nn}b_n \end{bmatrix}$$

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_k$$

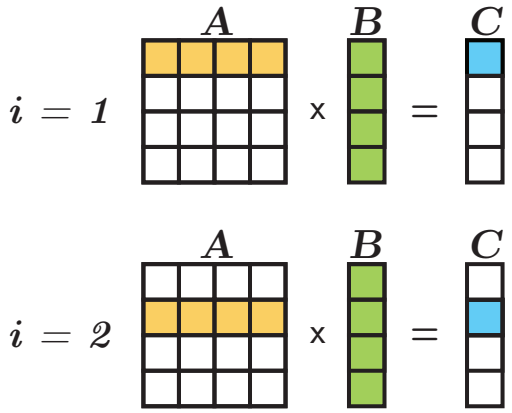


Figure 3: Matrix-vector multiplication

Figure 5 depics the data structures and how the elements interact. Importantly, we must take note that although there is no dependence between memory locations, there is a flow dependence as the operations on elements in $\boldsymbol{A}$ and $\vec{B}$ reduce into $\vec{C}$. This does not mean, however, that we cannot exploit parallelism here: given that we operate on rows of data in $\boldsymbol{A}$, we can parallelise the outer `i` loop to achieve thread level parallelism. The speedup achieved by this approach is contained in Figure 5. Two speedup curves are presented: one that uses the sequential `numpy` calculation of the computation, and another that uses a sequential C implementation as the baseline. We do not see any type of consistent speedup until matrices of size greater than 1024 x 1024. Even then, it appears to flatten out after 5 threads, indicating that for the tested sizes, the problem is potentially still memory bound.

Listing 1: Thread level parallelism of Matrix-Vector Multiplication

```
1  void matVectMultiply(double *A, double *V, double *results, unsigned long matrixSize, int
       numThreads) {
2      omp_set_num_threads(numThreads);
3      unsigned long i, j;
4
5      #pragma omp parallel for private(j)
6      for (i = 0; i < matrixSize; i++)
7      {
8          double tmp = 0;
9          double *AHead = &A[i * matrixSize];
10         for (j = 0; j < matrixSize; j++)
11         {
12             tmp += AHead[j] * V[j];
13         }
14         results[i] = tmp;
15     }
16 }
```

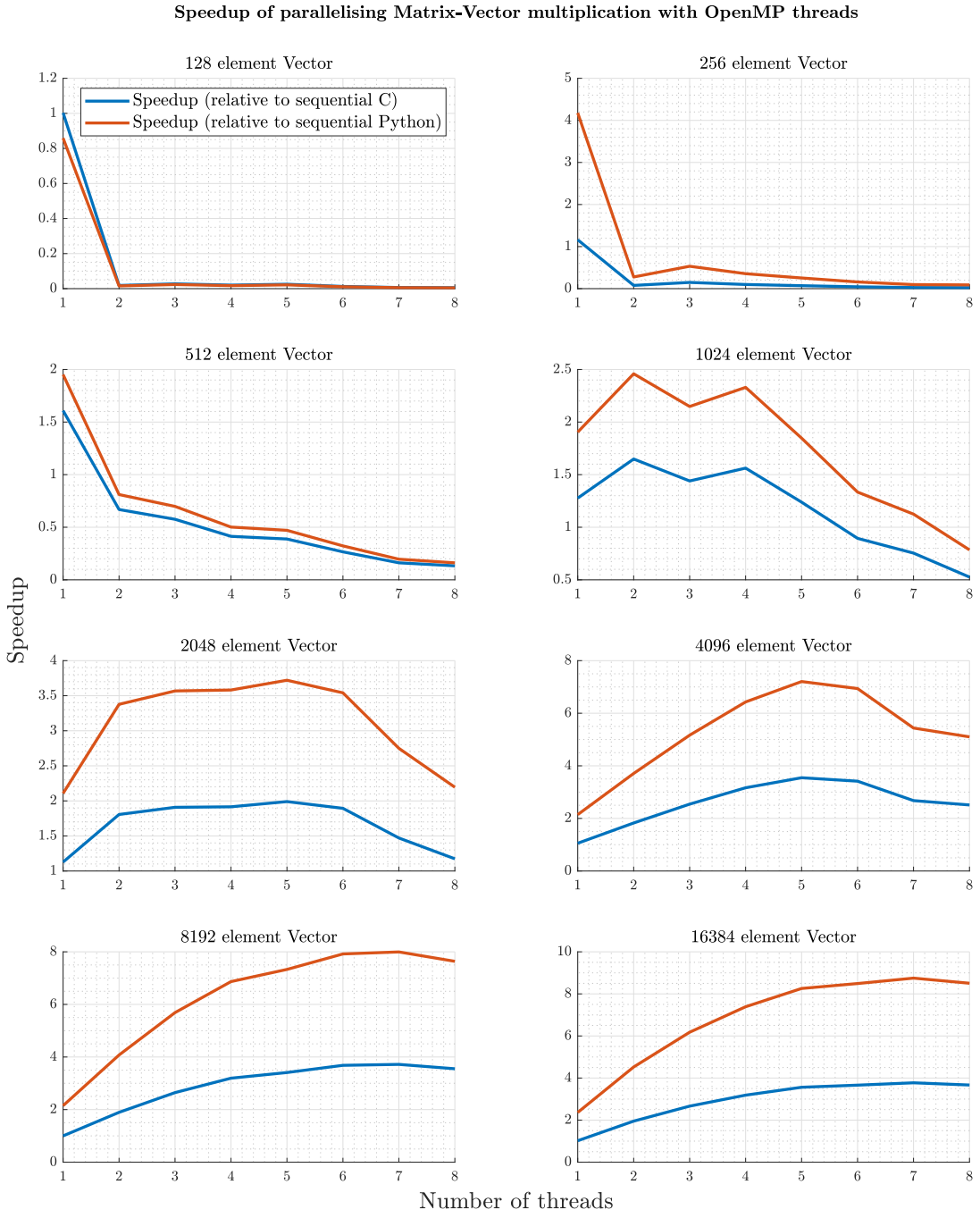


Figure 4: Matrix-vector multiplication speedup by parallelising outer `i` loop

We can leverage another form of parallelism here to increase speedup further: vectorisation. Many modern CPU's support hardware-level parallelism in the form of vector (or SIMD) instructions. These instructions make use of special vector registers, and can be used to perform the same operation (such as addition) to multiple pieces of data simultaneously (hence, **S**ingle **I**nstruction **M**ultiple **D**ata). Autovectorisation is a technique to achieve vectorisation of code without explicitly telling the compiler which sections should be vectorised. `gcc` is capable of vectorising in such a manner, but certain tweaks are necessary to give the compiler the necessary

assurances that the code is safe to vectorise. We first change the argument types of the matrices and vectors from `double *` to `double *restrict`: this tells the compiler that the blocks pointed to by the pointers do not overlap. When compiling, it was found via experimentation that the `-march` flag needed to be set appropriately: merely specifying `-O3` was not enough.

Development was performed in WSL 2 for a target i7-9700k processor which supports AVX instructions, thus `-march=native` is appropriate here. Note the inclusion of the `reduction` clause: it is *not* necessary for synchronisation, but it results in the use of packed double vector instructions rather than scalar double instructions (see Figure 6).

Loop unrolling and loop tiling was attempted in conjunction with autovectorisation, with no success in achieving any further speedup for the problem sizes investigated.

Listing 2: Thread and vector level parallelism of Matrix-Vector Multiplication

```
1  // compile with -O3 and specify -march suitably (e.g. -march=native if host machine supports
       vector instructions)
2  void matVectMultiply(double *restrict A, double *restrict V, double *restrict results, unsigned
       long matrixSize, int numThreads) {
3      omp_set_num_threads(numThreads);
4      unsigned long i, j;
5
6      #pragma omp parallel for private(j)
7      for (i = 0; i < matrixSize; i++)
8      {
9          double tmp = 0;
10         double *AHead = &A[i * matrixSize];
11         for (j = 0; j < matrixSize; j++)
12         {
13             tmp += AHead[j] * V[j];
14         }
15         results[i] = tmp;
16     }
17 }
```

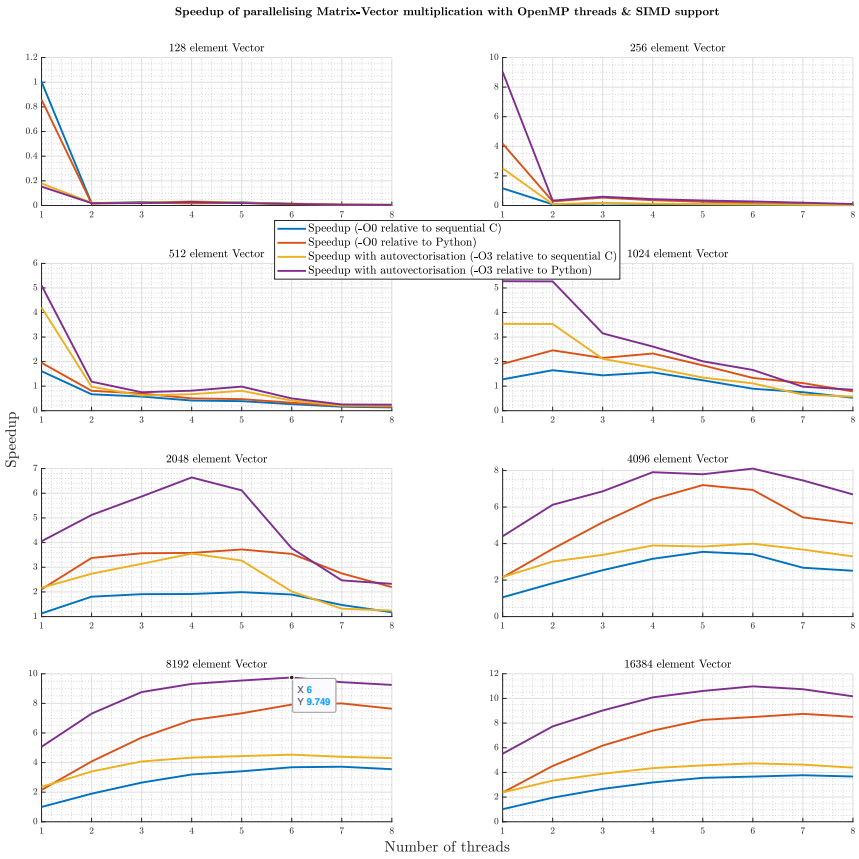


Figure 5: Matrix-vector multiplication speedup by parallelising outer `i` loop and vectorising the reduction operation

Without reduction clause, we only use scalar double instructions

```
#pragma omp simd
for (j = 0; j < matrixSize; j++)
{
    tmp += AHead[j] * V[j];

}
results[i] = tmp;
```

```
.L32:
        vmovsd  xmm2, QWORD PTR [rdi+rax]
        vfmadd231sd     xmm0, xmm2, QWORD PTR [rsi+rax]
        add     rax, 8
        cmp     rcx, rax
        jne     .L32
        vmovsd  QWORD PTR [rdx], xmm0
        add     rdx, 8
        add     rdi, rcx
        cmp     r8, rdx
        jne     .L33
```

Adding the reduction clause results in use of packed doubles,
rather than scalar double instructions (operates on more data at once)

```
#pragma omp simd reduction(+:tmp)
for (j = 0; j < matrixSize; j++)
{
    tmp += AHead[j] * V[j];

}
results[i] = tmp;
```

```
.L6:
        add     rdx, 1
        vmovupd ymm2, YMMWORD PTR [r14+rax]
        vfmadd231pd     ymm0, ymm2, YMMWORD PTR [rcx+rax]
        add     rax, 32
        cmp     rdx, rsi
        jb      .L6
        vmovapd YMMWORD PTR [rbp-80], ymm0
        mov     rax, r10
        cmp     rbx, r10
        je      .L7
```

Figure 6: Use of OpenMP `reduction` clause results in use of packed double instructions

3.3 Dot product

We will see below that the dot product of two vectors is a component of the power iteration algorithm, thus the possibility of parallelising this operation should be explored. A sequential implementation of the dot product is contained in 3. There is no dependence between memory locations in $\vec{A}$ or $\vec{B}$, however it must be noted that the `+=` operator is *not atomic*: it essentially decomposes into a read and a write operation. Thus the `result` variable creates a flow dependence between iterations. However, this does not mean the loop cannot be parallelised: OpenMP supports this pattern of programming with the `reduction` clause which can be used here to ensure access to the result variable is threadsafe and performant. Moreover, the loop is amenable to parallelisation as arithmetic addition is associative [1] (i.e. it does not matter how the numbers to be added are grouped). Figure 7 shows these dependencies and how the memory locations of the vectors interact to produce the result.

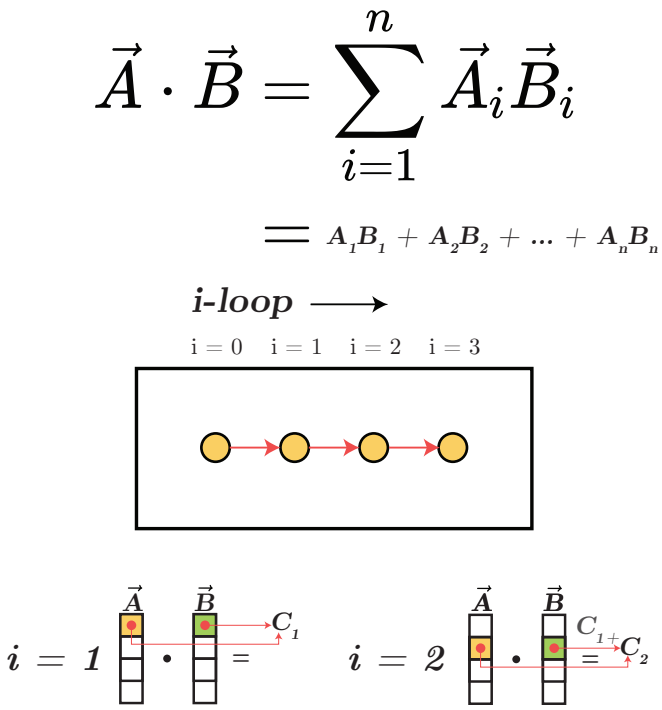


Figure 7: Dependencies in the dot product calculation

Code listing 4 clontains a parallelised version of the sequential code presented in 3. The speedup graph is contained in Figure 8. We see immediately that for the given vector sizes, thread level parallelism is a considerable performance detriment. Such performance losses are due to the overhead of thread management far exceeding the time saved by processing chunks for the `i` loop. The largest vector in the figure contains 16,384 double type numbers. Assuming the loop is statically scheduled such that each thread does an equal number of iterations, at 8 threads that is only $\frac{16{,}384}{8} = 2048 \frac{\text{elements}}{\text{thread}}$. Thus the application is better off, in terms of performance, not using thread-level parallelism of the dot product.

As with matrix vector multiplication, manual loop unrolling was trialled and benchmarked, with no appreciable benefit in speedup to the autovectorised version without unrolled loops.

Listing 3: Sequential dot product calculation in C

[1] Floating point addition performed by our computers, however, is not strictly associative, but this is not an issue for the present purposes

```
double dotProduct(double *A, double *B, unsigned long matrixSize)
{
    unsigned long i;

    double result = 0.0;
    for (i = 0; i < matrixSize; i++)
           result += A[i] * B[i];
    return result;
}
```

Listing 4: Thread level parallelisation of dot product calculation in C

```
double dotProduct(double *A, double *B, unsigned long matrixSize, int numThreads) {
    omp_set_num_threads(numThreads);
    unsigned long i;
    double result = 0.0;

    #pragma omp parallel reduction(+:result)
    for (i = 0; i < matrixSize; i++)
        result += A[i] * B[i];

    return result;
}
```

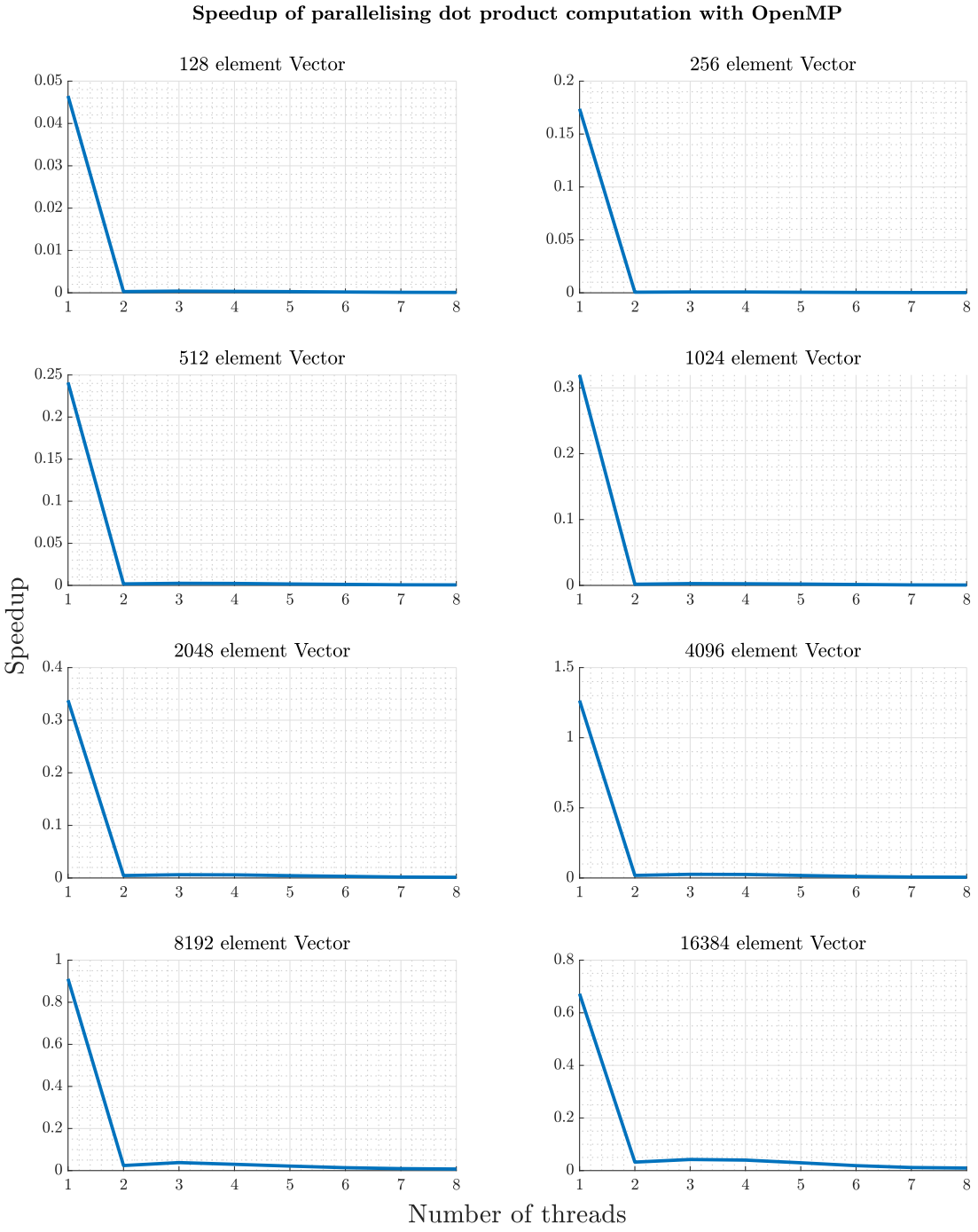


Figure 8: Dot product speedup with thread level parallelisation

As with matrix-vector multiplication, discussed under Heading 3.2, we can exploit hardware level parallelism here to achieve parallel speedup. The process of autovectorising with `gcc` is much the same as discussed under that heading. The code for achieving vectorised calculations is contained in Listing 5. The speedup compared to threading and threading with SIMD is contained in Figure 9. We see a significant performance improvement across all vector sizes when only using vector level parallelism of the dot product.

Listing 5: Autovectorisation of dot product calculations in C

```
double dotProduct(double *A, double *B, unsigned long matrixSize, int numThreads) {
    omp_set_num_threads(numThreads);
    unsigned long i;
    double result = 0.0;

    #pragma omp simd reduction(+:result)
    for (i = 0; i < matrixSize; i++)
        result += A[i] * B[i];

    return result;
}
```

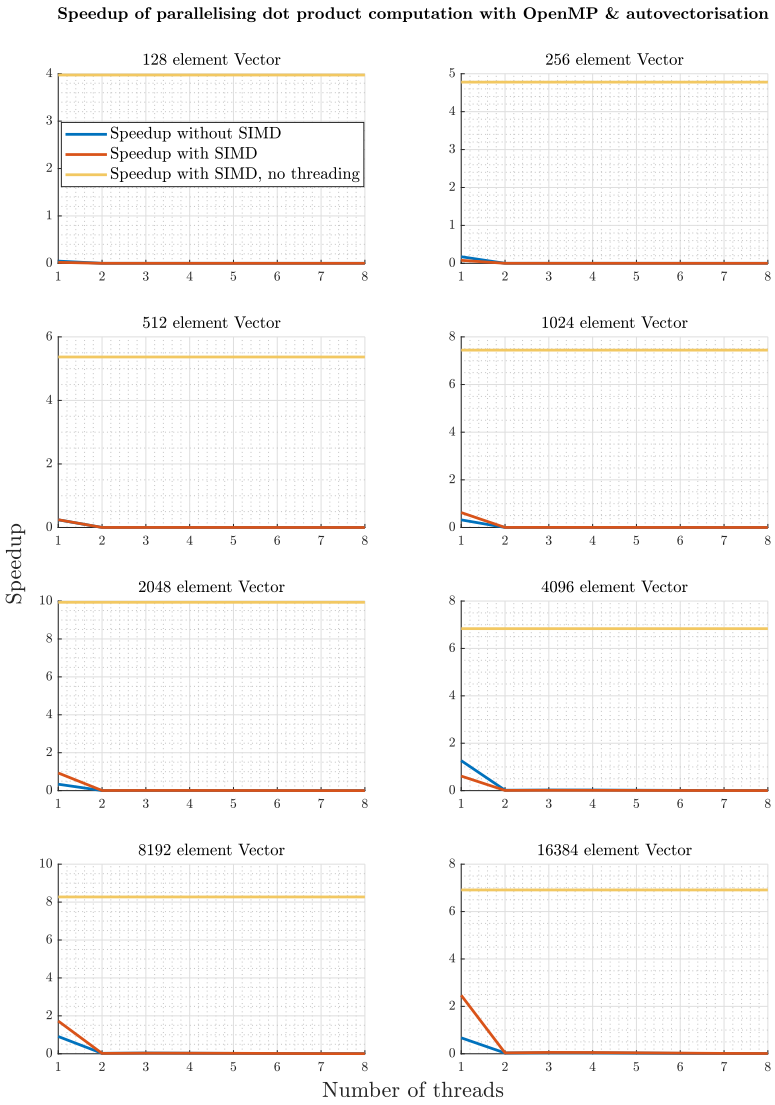


Figure 9: Dot product speedup comparison of parallelisation techniques

3.4 Eigenpair calculations

It is not surprising that Newman's Leading Eigenvector algorithms based on the repetition of finding eigenvectors (and eigenvalues) of matrices. The previous sections discussed the speedup of crucial components of the algorithm that will be used to find the eigenvectors and eigenvalues necessary to implement Newman's algorithm. This section will discuss one algorithm for finding eigenvectors, where it can be parallelised, and the speedup we obtain in doing so.

3.4.1 Power iteration

The algorithm chosen to be implemented here for the purposes of finding eigenvectors and their corresponding eigenvalues is the power iteration algorithm (also commonly referred to as the power method). Finding eigenvectors is absolutely essential to Newman's algorithm: the signs (+ and -) of the eigenvectors that we compute tell us which community nodes belong to. This section will detail what it is we seek to compute and how we will do it, but will eschew any substantial mathematical rigour.

An eigenvector is a vector that, for some matrix $\boldsymbol{A}$, satisfies the following:

$$\boldsymbol{A}\vec{v} = \lambda\vec{v} \tag{1}$$

Where λ is a scalar value (the eigenvalue), and $\vec{v}$ is an eigenvector of $\boldsymbol{A}$. In plain English then, an eigenvector of some matrix $\boldsymbol{A}$ is a vector that, when multiplied by $\boldsymbol{A}$, results in a new vector that is simply the original vector, multiplied by a scalar value.

Power iteration is an iterative algorithm that computes this vector $\vec{v}$. The gist of the algorithm is contained in Figure 10: start with an initial "guess" vector x_0 and keep multiplying by the matrix of interest $\boldsymbol{A}$. Eventually you will converge on the dominant eigenvector of $\boldsymbol{A}$. What is not conveyed by this graphic is the need to renormalise the calculated vector at each step (which is as simple as dividing each element in the vector by the magnitude of the vector).

This algorithm offers a few advantages:

- It converges on the *dominant* eigenvector (i.e. the eigenvector with the largest eigenvalue in terms of magnitude). This is useful as it is closely related to what we seek to find: the eigenvector that corresponds to the most positive eigenvalue. It does not compute any other eigenvectors (we will see that this is a substantial source of speedup);
- It is heavily reliant on matrix-vector multiplication which benefits from the forms of parallelisation discussed under Heading 3.2;
- It is conceptually quite simple and is thus a useful starting point for discussing parallelisability of eigenproblems.

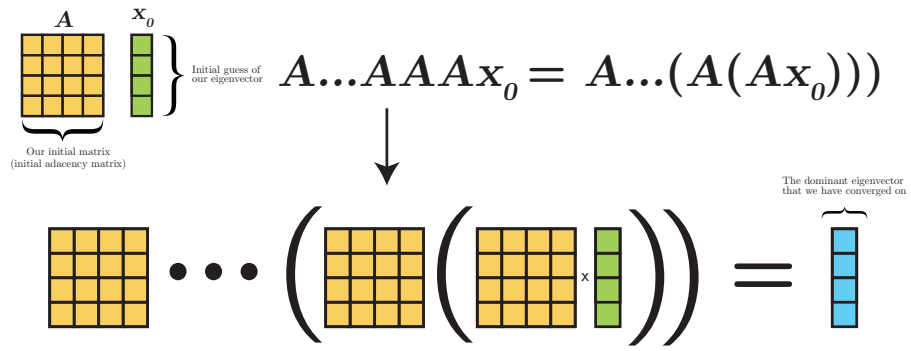


Figure 10: Matrix-vector multiplication speedup by parallelising outer `i` loop and vectorising the reduction operation

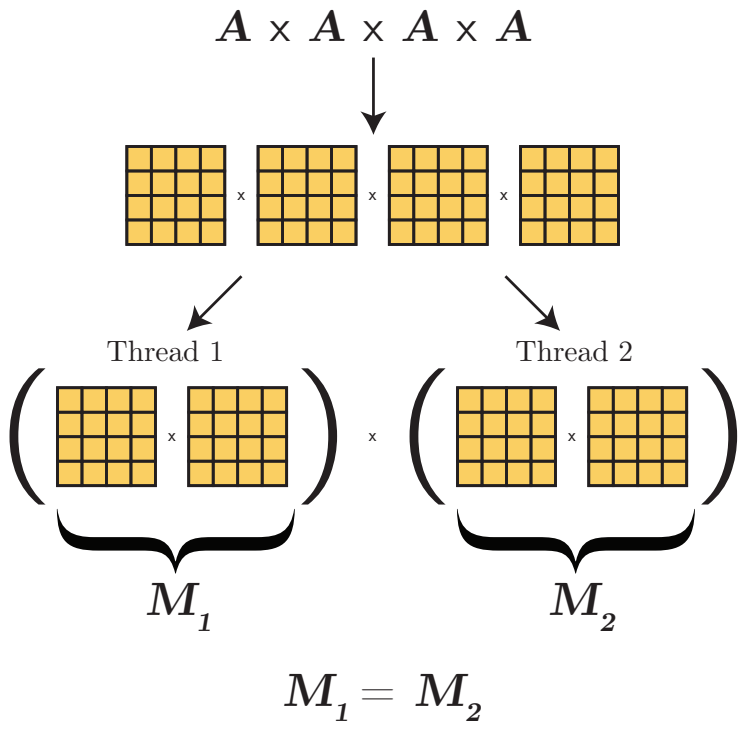


Figure 11: Matrix-vector multiplication speedup by parallelising outer `i` loop and vectorising the reduction operation

Algorithm 3 The power method algorithm for computing the dominant eigenvector of a matrix *matrA*

input : A: A matrix we wish to find the dominant eignenvector of

Function `powerMethod(`A`)`**:**

$\vec{x_0} \leftarrow$ Initial eigenvector guess ▷ We will create a random vector as our guess in code

while *not converged* **do**

$\vec{x}_{\text{tmp}} \leftarrow \boldsymbol{A}\vec{x}_0$

$\vec{x}_0 \leftarrow \|\vec{x}_{\text{tmp}}\|$

$\lambda \leftarrow \frac{\vec{x}_0 \cdot \boldsymbol{A}\vec{x}_0}{\vec{x}_0 \cdot \vec{x}_0}$ ▷ Compute the Rayleigh Quotient to estimate eigenvalue

if $|\lambda - \lambda_{previous}| < THRESHOLD$ **then**

converged = true

else

$\lambda_{\text{previous}} \leftarrow \lambda$

end

end

return x_0, λ

End Function

Almost every step involves some type of matrix or vector operation, which has potential for parallelising. In fact, the main operations of this algorithm are matrix-vector multiplication and dot product computation, both of which have been analysed (under Headings 3.2 and 3.3 respectively). There is another avenue of parallelisation that is graphically depicted in Figure 11: Matrix-multiplication is *not* commutative, but it is associative: the order of operations matter but the grouping does not. Thus it would be possible to parallelise the multiplication of many matrices by computing the products of matrices simultaneously (e.g. in Figure 11 we see 2 matrix multiplications being computed simultaneously).

With a bit of thought, we can see why this is emphatically *not* a good idea. Foremost, there is no benefit to computing multiple matrix-products in parallel in this manner where the matrices are identical. In Figure 11 $\boldsymbol{M}_1 = \boldsymbol{M}_2$: 2 threads were used to compute an identical result, whereas we could have used these computing resources to compute only $\boldsymbol{M}_1$ more quickly (for an appropriately large matrix $\boldsymbol{A}$), and simply copy the result into $\boldsymbol{M}_2$.

Benchmarking of the power method (contained in Listing 7.3) shows good speedup obtained by the algorithm implementation. Figure 12 shows the speedup obtained with and without vectorisation, relative to a sequential implementation of the power method in C, and relative to calculation the eigenvectors in Python with `numpy`. The power method implementation is up to 16 faster than using `eig` in `numpy` for a moderate size network (described by a 512 x 512 matrix). This calculation took, on average, 30 milliseconds in C with SIMD support and 4 threads, and approximately 500 milliseconds in Python.

One drawback of the power method can be seen: it does not appear to scale well against the `numpy` algorithm as the problem size grows. It retains respectable parallelism speedup relative to its own sequential implementation, but there is diminishing return on investment compared to using the Python libraries as the problem size increase (the parallelism speedup is almost identical for the 1024 x 1024 matrix, after which the speedup is diminishing).

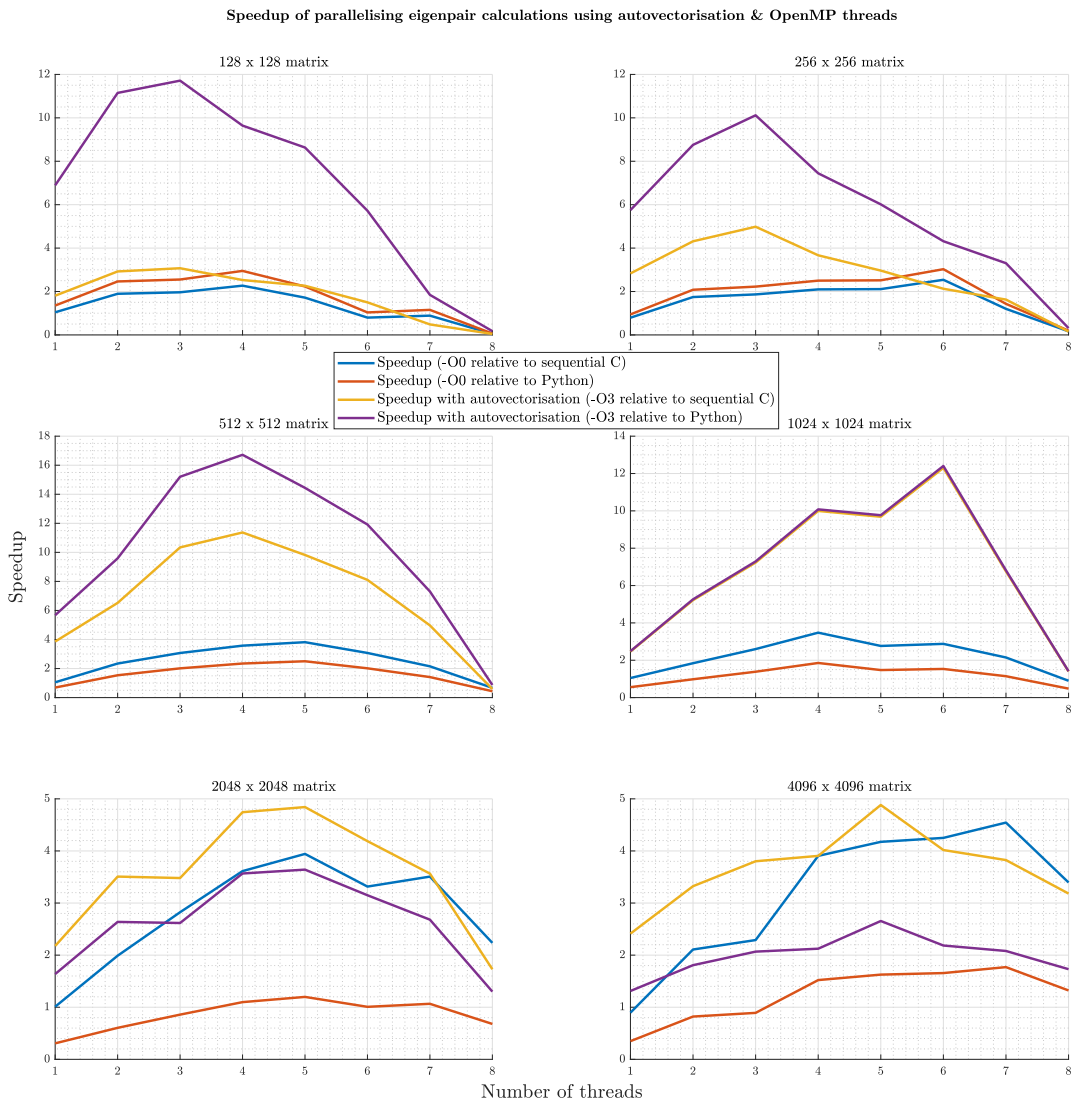


Figure 12: Power method speedup

3.5 Tasking the recursion

Each pass of the algorithm seeks to split the nodes into two groups (distinguished as `left` and `right` in the code). For each split of nodes that is deemed to increase the modularity, we repeat the process, trying to split these nodes into `left` and `right` groups. This divide and conquer pattern of recursion can be implemented in OpenMP with the `task` directive (introduced into OpenMP in v3.0).

The `task` construct is especially useful in situations where an unknown amount of work is needed to be performed concurrently. Tasks would allow the program to execute the `assignCommunity` function concurrently for the `left` community and the `right` community.

This theory was trialled initially with 2 concurrent matrix-vector multiplications but found to be either unreliable for small inputs, or no performance benefit for large inputs. Moreover, tasking the recursion was also tried on the application, with execution time *increasing* by 5-10x.

4 Discussion

4.1 Algorithm speedup

The previous sections have discussed the speedup of the essential computations in the algorithm. We will now look at the speedup realised by running the application as a whole. Figure 13 contains the speedup for various network sizes (note that this graph is in log scale). The blue curve is indicative of the speedup we get due to parallelising alone. We see sublinear speedups when comparing to the C implementation running on only 1 thread: For a 4096 element network we see a maximum speedup of $\sim$ 4.1x when computing with 6 threads. Threading speedup relative to the original Python implementation is $\sim$35x for 6 threads in this network size (with 1 thread it is initially $\sim$8.5x faster). The speedup we see when compiling with `-O3 -march=native` is as high as 134x in the 1024 element network. It is unlikely that this respectable speedup jump is due entirely to the vectorising of computations, although it is certainly a contributing factor.

For very small matrices (such as Zachary's Karate Network at 34 nodes) it was found that the running time of the Python application and single threaded C application were roughly equal.

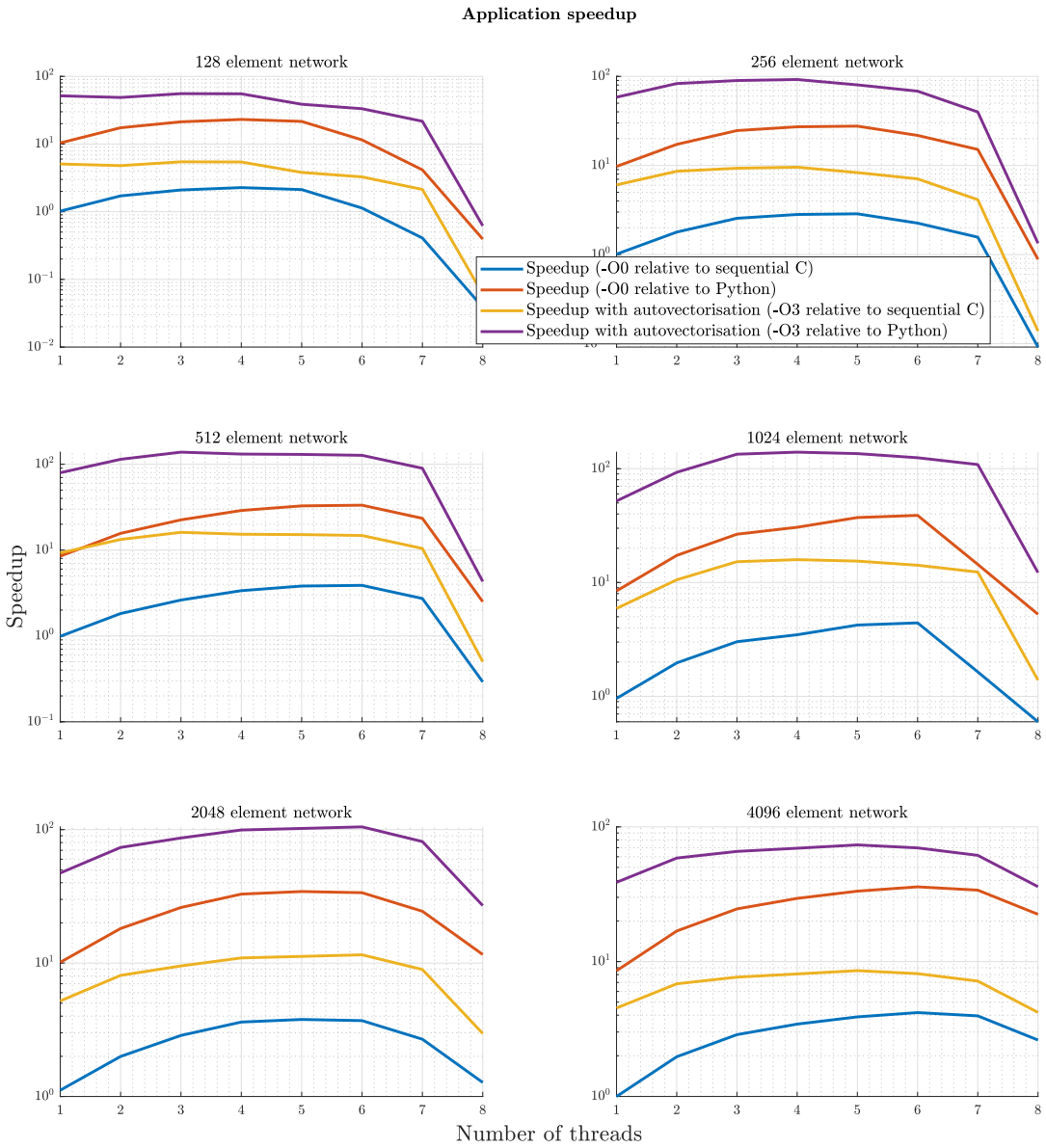


Figure 13: Application speedup

4.2 Benchmarking & correctness

Great care was taken to benchmark and verify the main components of the application for their suitability in parallelisation. All major computational tasks have their own corresponding benchmarking build configuration that can be run in VisualStudio Code. Some of these benchmarks are shown in Figure 14. Each benchmark can be easily reconfigured in the source code to run a sequential version and a parallel version of the computation a specified number of times, for a specified thread range (starting from 1) and for a specified list of problem sizes. This made it quick to thoroughly test the effect optimisations would have on speedup and problem-size scalability. See Listing 6 for an example of the configurable components from a benchmarking file.

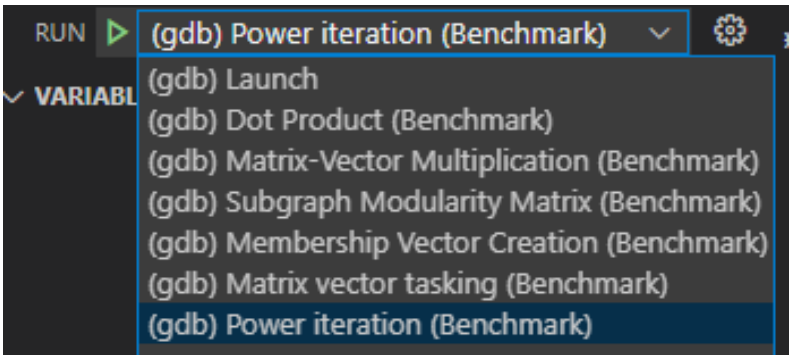


Figure 14: Benchmarking configurations

Listing 6: Configurable components contained in the benchmarking files

```
1 #define THREAD_RANGE 8 // Run for 1:THREAD_RANGE threads
2 #define NUM_AVERAGES 15 // Average number of timings for each matrix size, and each number of
      threads
3 #define NUM_MATRIX_SIZES 8
4 unsigned long matrixSizes[NUM_MATRIX_SIZES] = {128, 256, 512, 1024, 2048, 4096, 8192, 16384};
```

Every benchmark also contains some form of assertion loop that ensures that the sequential and parallel versions of the benchmarked computation compute the same result.

File writing functions can be found in `CDUtils.c`: these functions take as input either a matrix or a vector and write to disk. This is used for checking the calculations in a program such as MATLAB which will read these matrices and vectors and compare the results against its own similar function implementations. For example, `eigen_comparison.m` is a MATLAB file that will take from disk a matrix and the membership vector, computed in the C implementation, and check the signs of the membership vector against its own eigenpair calculations. Errors of less than 2% are commonly achieved from the parallel implementation.

The adjacency graph describing the Zachary's Karate Club network (an often used example when looking at community detection) saw results that were consistent with those reported in Newman's paper: all 34 nodes are placed in the correct communities. Compiling `main.c` and running will output the communities of this network to a textfile. The communities are identical to the communities found in the original Python implementation.

4.3 Difficulties, reflection & future work

Numerous setbacks were encountered in this application. I wished to implement a number of linear algebra functions from scratch to better engage with the theory of parallel computing. Such functions are available out-of-the-box in `numpy`, and took time to implement successfully. Despite the simplicity of the code needed to achieve vectorisation of tasks, it took a while to figure out the proper compiler flags and OpenMP directives needed to autovectorise the relevant sections of code. It is still not clear to me why the `reduction` clause is needed to ensure packed double instructions are generated by the compiler.

A day of development time was lost due to bug hunting a heap corruption issue that was caused by overflowing allocated space when copying the subgraph modularity matrices before recursing. `valgrind` was useful here but took some time for me to learn (as well as research causes of heap corruption).

There was significant experimentation, code refactorings and many benchmarks were run to achieve the final result. Having the benchmarks setup to be configurable in the way that they are reduced the complexity of this process.

4.4 Improvements and future work

The implementation could be refined further. For instance, cache optimisation could be explored for the matrix-vector product. Loop tiling was trialled for various network sizes but not found to be a significant speedup (in some cases, it was a source of slow down).

Moreover, there are techniques that can be used to speed up the rate of convergence of the power method algorithm that should be investigated. Research should be undertaken for methods of generating the initial "guess" vector at the start of the power method algorithm: variations in the output result due to the random-generation approach currently implemented. Whilst the accuracy is suitable, there is room for improvement.

The application could, alternatively, be rewritten to not use custom implementations of standard linear algebra algorithms, and instead make use of well established and tested BLAS and LAPACK routines that are better optimised for this sort of computation. These functions were deliberately avoided as I wished to challenge myself and engage with the subject material more closely. More serious implementations (especially for production level code) would be well advised to use these codebases.

5 Conclusion

This report has looked at parallelising an implementation of a popular community detection algorithm. The original application was written in Python, making use of `numpy`, and was ported to C to be parallelised using OpenMP and vectorising computations where possible. Effort was made to speedup the computational hotspot: calculating eigenvectors. The algorithm chosen to be implemented was the power method. The parallelism of this algorithm was explained, focusing specifically on matrix-vector multiplication and dot product computation. Alternative methods of parallelising that resulted in slow down were discussed (such as tasking).

Overall speedup of the application is modest when only considering the effects of threading (4x speedup typical). The speedup when considering compiler optimisations (such as autovectorisation) relative to the original application in Python, however, is respectable, seeing up to 134x speedup.

Possibilities of future work and refinement have been discussed, including reinvestigating cache optimisations and moving to more mature and well tested codebases such as LAPACK.

6 References

Newman, M. E. (2006). Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582.

7 Appendix

7.1 Creating the subgraph adjacency matrix

The matrix that we seek to find the leading eigenvector of is referred to in Newman's paper as the "modularity matrix": it is a square, symmetric matrix that contains as its entries the difference between the actual number of connections between 2 nodes, and the expected number of connections if you were to "rewire" the nodes, based on random chance. Figure 15 shows the matrix and vector data structures that are relevant here, and how their respective elements interact to form the modularity matrix $\boldsymbol{B}$. The *subgraph* modularity matrix $\boldsymbol{B}_g$ is the generalisation of this matrix $\boldsymbol{B}$ that contains only the entries of $\boldsymbol{B}$ that are relevant for the current community we are looking at. In the first pass of this algorithm, $\boldsymbol{B} = \boldsymbol{B}_g$ because all nodes are in the same group initially.

We first start by performing a reduction operation of summing the rows of $\boldsymbol{A}$ and storing the results in $\boldsymbol{D}$ (for $\boldsymbol{D}$egrees). As discussed above, there is a flow dependency here however this type of pattern is trivial to implement due to OpenMP's `reduction` clause.

From this degree vector we can construct a matrix $\boldsymbol{K}$ (there is no good mnemonic device here, it's just how it appears in the original paper): $\boldsymbol{K}$ is the result of multiplying $\boldsymbol{D}$ and its transpose, and dividing each term by $2m$. This result is a matrix such that $\boldsymbol{K}_{ij}$ is the probability that a node with the same degree as Node i would be connected a node with the same degree as Node j in a network with the same total number of connections as the one being investigated.

Finally, $\boldsymbol{B}$ can be constructed by performing the matrix arithmetic $\boldsymbol{B}_{ij} = \boldsymbol{A}_{ij} - \boldsymbol{K}_{ij}$. This operation is trivially parallelisable as there are no dependencies between iterations.

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2m}$$

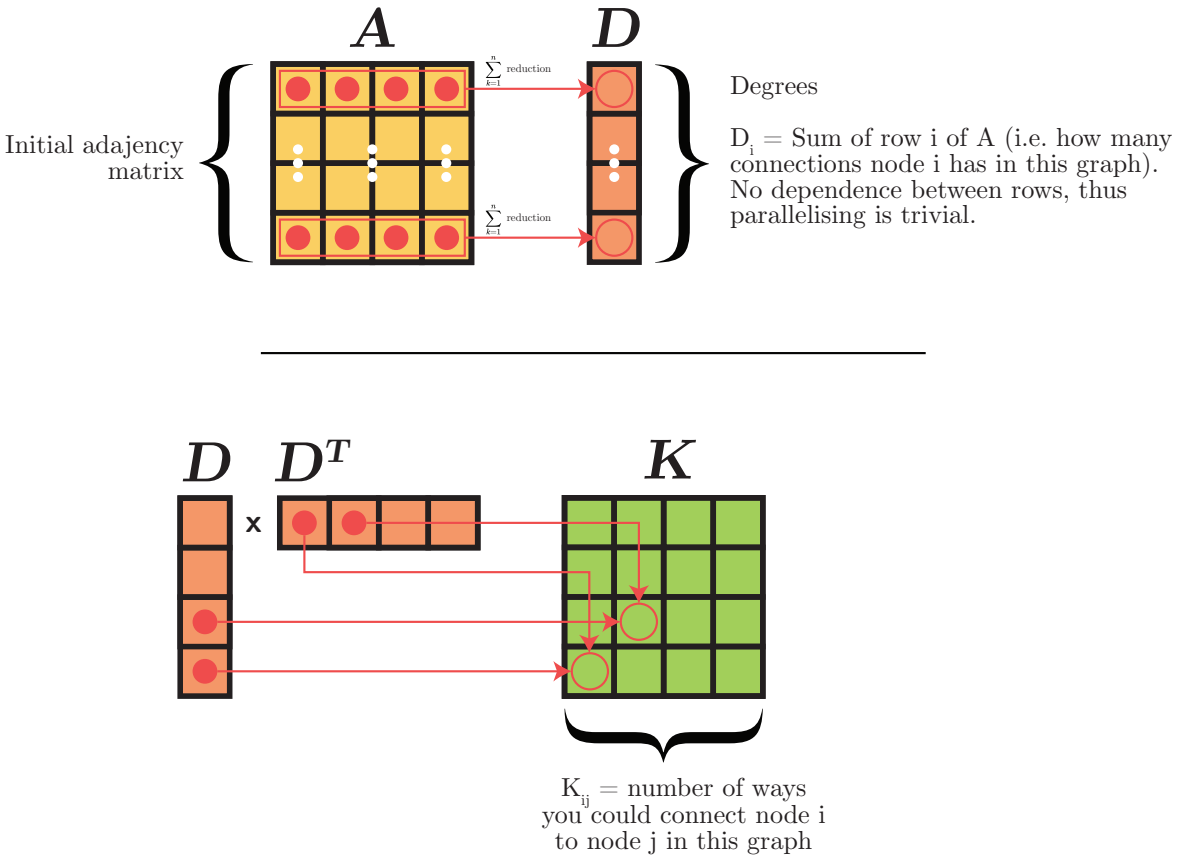


Figure 15: Matrix-vector multiplication speedup by parallelising outer `i` loop and vectorising the reduction operation

7.2 Python implementation of `assignCommunity`

Listing 7: Python 3 implementation of Newman's Algorithm

```
1  def assignCommunity(B, category, globals):
```

```
2
3       I = np.identity(B.shape[0])
4       B_transpose = B.transpose()
5
6       # kronecker_sum calculates the kroencer delta * sum of B rows (from equation 6)
7       kronecker_sum = np.multiply( I ,
8                           np.sum(B_transpose, axis = 1).reshape(B.shape[0],1) # sum up the
                                tranpose of B, and turn it into a column vector for the next step
9                         )
10
11      # Compute equation 6
12      Bg = np.subtract(B, kronecker_sum)
13
14      eigen_values, eigen_vectors = LA.eig(Bg)
15
16      # Find the most positive eigenvalue, and the corresponding eigenvector
17      leading_eigen_value = np.amax(eigen_values)
18      index_of_lead = np.where(eigen_values == leading_eigen_value)
19      leading_eigen_vector = eigen_vectors[:, [index_of_lead]] # extract the column vector
            representing the leading eigenvector
20
21      # membership vector (place network nodes in 1 group if the same eignevector index is geq to
            0, else put into a different group)
22      s = np.where(leading_eigen_vector >= 0, 1, -1)
23
24      if (leading_eigen_value < 0.1):
25          labels = np.full((1, B.shape[0]), category)
26          category = category + 1
27          return [labels, category]
28      else:
29          # node indices in Bg that correspond to the first group
30          left_indices = np.array([elem[0] for elem in np.argwhere(s == 1)])
31
32          # node indices in Bg that correspond to the second group
33          right_indices = np.array([elem[0] for elem in np.argwhere(s == -1)])
34
35          # Elements in B corresponding to nodes in our first and second groups respectively
36          left_B = B[np.ix_(left_indices,left_indices)]
37          right_B = B[np.ix_(right_indices,right_indices)]
38
39          # recurse on our group, try and split them up further
40          [left, category] = communities(left_B, category, globals[np.ix_(left_indices)])
41          [right, category] = communities(right_B, category, globals[np.ix_(right_indices)])
42
43          labelled_vertices = np.zeros(max(left.shape) + max(right.shape)) # allocate an array of
                the correct size to put our labelled nodes in
44          labelled_vertices[np.ix_(left_indices)] = left
45          labelled_vertices[np.ix_(right_indices)] = right
46          return [labelled_vertices, category]
```

7.3 C implementation of eigenvector calculation

Listing 8: C implementation of the power iteration algorithm

```
1   eigenPair powerIteration(double *B, unsigned long size, int numThreads, double tolerance, int
        iterationLimit)
2   {
```

```
3       double *eigenVectorTmp = (double *)malloc(size * sizeof(double));
4       double *eigenVector = (double *)malloc(size * sizeof(double));
5       eigenPair eigP;
6       genRandMembershipVector(eigenVectorTmp, size);
7       for (int i = 0; i < size; i++){
8           eigenVector[i] = eigenVectorTmp[i];
9       }
10
11      double eigenValue = 0;
12      double prevEigenValue = __INT_MAX__;
13      double norm = 0;
14
15
16      bool converged = false;
17      int numIterations = 0;
18      while (!converged && numIterations < iterationLimit)
19      {
20          int k;
21          matVectMultiply(B, eigenVectorTmp, eigenVector, size, numThreads);
22          norm = 0;
23          #pragma omp parallel for private(k) reduction(+:norm)
24          for (k = 0; k < size; k++)
25          {
26               norm += eigenVector[k] * eigenVector[k];
27          }
28          #pragma omp parallel for private(k)
29          for (k = 0; k < size; k++)
30          {
31              eigenVectorTmp[k] = eigenVector[k] / sqrt(norm);
32          }
33
34          eigenValue = rayleighQuotient(B, eigenVector, size, numThreads);
35          double diff = fabs(eigenValue - prevEigenValue);
36          if (diff < tolerance)
37          {
38              // printf("converged after %d iterations, matrix size: %ld\n", numIterations, size);
39              converged = true;
40          }
41
42          prevEigenValue = eigenValue;
43          numIterations++;
44
45          // numIterations++;
46      }
47
48      // N.B. This is not part of the typical power iteration algorithm
49      // this is specifically for our purposes of finding the eigenvector
50      // corresponding to the most positive eigenvalue. If the eigenvalue is negative we need
51      // to perform a spectral shift and repeat the process one more time
52
53
54      // double eigenValue = rayleighQuotient(B, eigenVector, size, numThreads);
55      if (eigenValue < 0) {
56          double *newB = (double *)malloc(size * size * sizeof(double));
57
58          memcpy(newB, B, size * size * sizeof(double));
59          for (int i = 0; i < size; i++){
60              newB[i * size + i] += fabs(eigenValue);
61          }
```

```
62          free(eigenVectorTmp);
63          free(eigenVector);
64
65          eigP = powerIteration(newB, size, numThreads, tolerance, iterationLimit);
66
67          free(newB);
68          return eigP;
69
70      } else {
71          eigP.eigenvalue = eigenValue;
72          eigP.eigenvector = eigenVector;
73          return eigP;
74      }
75      eigP.eigenvalue = eigenValue;
76      eigP.eigenvector = eigenVector;
77      free(eigenVectorTmp);
78      return eigP;
79  }
```