Hybrid Parallelization of 3D Spectral Clustering

Guglielmo Boi

Abstract—Spectral clustering is effective for identifying non-spherical clusters but suffers from high computational and memory costs that limit scalability. This work presents a hybrid parallel implementation of spectral clustering for three-dimensional data, combining MPI and OpenMP. The implementation is evaluated on synthetic datasets with up to 131,072 points. Results show that the method maintains good clustering accuracy while achieving moderate speedup, with scalability primarily limited by eigenvalue decomposition time complexity and memory bandwidth saturation.

1. INTRODUCTION

Clustering is the problem of partitioning a population into classes. The goal is to construct a partition in which elements of a cluster are "similar" and elements of different clusters are "dissimilar" [1].

A partition is a set of mutually exclusive and exhaustive subsets of the population. As searching through all possible partitions is impractical, clustering methods use various heuristic search strategies to find an approximate optimal solution.

Spectral clustering is one of the various methods. Spectral clustering algorithms have shown to be more effective in finding clusters than some traditional algorithms such as k-means. [2]. Indeed, results obtained by spectral clustering often outperform the traditional approaches [3]. Moreover, it is convenient because it can be solved efficiently by standard linear algebra methods and it is more accurate for non-spherical clusters.

Like most clustering algorithms, spectral clustering can be used for data points in any dimension. The proposed parallel algorithm implements spectral clustering for 3D data points. In principle, the same algorithm could be used for datasets of any dimension, but results have shown that for larger dimensions the accuracy is not acceptable.

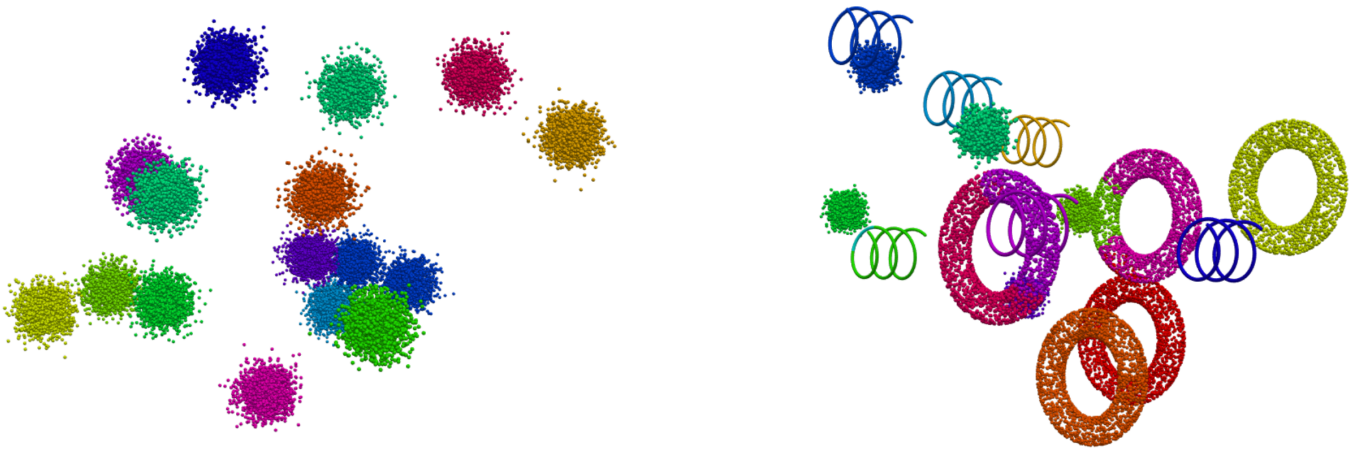

Figure 1. Plot of the proposed algorithm output with gaussian and mixed datasets: 32 768 points, 16 clusters.

The datasets used in the testing phase of the algorithm contain not only the three coordinates (x, y, z) of the points, but also a pre-assigned cluster *label*. Clearly, the input labels are not used by the algorithm itself, but rather to measure its accuracy. Nevertheless, the number of clusters k is determined by the number of pre-assigned labels. Therefore, the proposed algorithm does not implement any method to determine k independently.

The datasets were randomly generated in two different ways:

- **gaussian** datasets contain Gaussian distributed clusters.

- **mixed** datasets contain a mix of Gaussian, torus and helix shaped clusters.

For both of the test classes three different input sizes were used: 32 768, 65 536 and 131 072 points, as shown in Table 1.

Table 1. Summary of the testing datasets.

Dataset class	Number of points (n)	Number of clusters (k)
gaussian	32 768	16
gaussian	65 536	32
gaussian	131 072	64
mixed	32 768	16
mixed	65 536	32
mixed	131 072	64

The number of clusters is fixed and depends on the number of points.

2. ALGORITHM

Given a dataset of n points arranged in a matrix $X \in \mathbb{R}^{n\times 3}$, where each row represents a 3-dimensional sample, the goal of the proposed algorithm is to partition the data into k clusters using the spectrum of a graph Laplacian derived from pairwise similarities.

2.1. Gaussian Similarity Matrix

We first construct a fully connected weighted graph where each node represents a data point and edge weights encode similarity.

The similarity between two points x_i and x_j is defined using the Gaussian kernel:

$$S_{ij} = \begin{cases} \exp\left(-\dfrac{\|x_i - x_j\|^2}{2\sigma^2}\right) & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases} \tag{1}$$

where $\sigma > 0$ controls the width of the neighborhood. This produces the symmetric Gaussian similarity matrix $S \in \mathbb{R}^{n\times n}$.

2.2. Diagonal Degree Matrix

From the similarity matrix, we compute the degree of each node:

$$d_i = \sum_{j=1}^{n} S_{ij} \tag{2}$$

These values form the diagonal degree matrix D:

$$D = \mathrm{diag}(d_1, d_2, \dots, d_n) \tag{3}$$

In the implementation, we directly build the matrix $D^{-\frac{1}{2}}$:

$$D^{-\frac{1}{2}} = \mathrm{diag}\left(\frac{1}{\sqrt{d_1}}, \dots, \frac{1}{\sqrt{d_n}}\right) \tag{4}$$

with safeguards for very small degrees.

2.3. Normalized Graph Laplacian

As proposed by Von Luxburg[3], we use the symmetric normalized graph Laplacian defined as:

$$L = I - D^{-\frac{1}{2}} S D^{-\frac{1}{2}} \tag{5}$$

where I is the $n \times n$ identity matrix.

2.4. Eigenvectors Embedding Matrix

We compute the first k eigenvectors corresponding to the smallest eigenvalues of the normalized graph Laplacian:

$$Lu_\ell = \lambda_\ell u_\ell, \quad \ell = 1, \dots, k \tag{6}$$

and form the embedding matrix by placing the k eigenvectors in columns, as proposed by Ng et al. [4]:

$$U = [u_1 \; u_2 \; \dots \; u_k] \in \mathbb{R}^{n \times k}. \tag{7}$$

Each row of U is then normalized to unit length.

2.5. K-Means Clustering

Finally, the rows of the normalized embedding matrix of size $n \times k$ are clustered using the standard k-means algorithm. Each row corresponds to a data point in the spectral space, and cluster assignments in this space define the final clustering in the original domain.

2.6. Accuracy Estimation

To evaluate the clustering quality of the proposed algorithm, we use the Adjusted Rand Index (ARI), which measures the similarity between two clustering instances by counting pairs of points that are assigned consistently in both partitions, while correcting for chance. The results are reported in Table 2.

The ARI ranges from -1 to 1, where:

- 1 indicates perfect agreement
- 0 corresponds to random labeling
- negative values indicate worse-than-random agreement

Table 2. Clustering accuracy measured using Adjusted Rand Index (ARI).

Dataset	Number of points (n)	ARI score
gaussian	32 768	0.873
gaussian	65 536	0.874
gaussian	131 072	0.774
mixed	32 768	0.868
mixed	65 536	0.726
mixed	131 072	0.619

Higher ARI values indicate stronger agreement between predicted labels and ground-truth labels.

For the gaussian datasets, the ARI remains above 0.87 for the small and medium sized inputs and decreases for the largest dataset. This suggests that while the spectral embedding remains effective as the problem scales, the increased similarity matrix size may reduce accuracy.

For the mixed datasets, the ARI values are lower, due to the more complex and less separable structure of the data. The score for the largest dataset is particularly lower, but still at a reasonable level.

Overall, the scores confirm that the algorithm maintains good clustering quality.

3. PARALLEL DESIGN

Spectral clustering suffers from a scalability problem in both computational time and memory use when the size of the dataset is large [2].

Unfortunately, spectral clustering can encounter a quadratic resource complexity when computing the pairwise similarity of data instances [5]. Furthermore, the algorithm requires a significant amount of time to compute the first k eigenvectors of a Laplacian matrix.

Therefore, a parallel solution based on distributed or shared memory is required for computing clusters in a reasonable amount of time. The proposed design is hybrid: it uses distributed memory to run multiple Message Passing Interface (MPI) processes on different nodes, with each process divided into OpenMP threads.

3.1. Memory Bottlenecks

A serious bottleneck for spectral clustering is the amount of memory required to store the similarity matrix and the normalized graph Laplacian, whose number of elements are equal to the square of the number of data points [2]. As demonstrated by Chen et al. [2], the most effective solution to these memory and computational challenges is to eliminate some of the similarity matrix's elements, or sparsify the matrix.

Although large-scale spectral clustering usually uses sparse similarity approaches, this work concentrates on moderate dataset sizes where a fully connected Gaussian similarity matrix can be computed. Therefore, a dense representation is adopted for simplicity.

For the largest considered datasets with $n = 131\,072$ points, the size of these matrices goes up to $n \cdot n \cdot 8\,B \simeq 137GB$.

3.2. Execution Time Bottlenecks

From profiling the algorithm, two stages clearly dominate the execution time:

- The similarity matrix computation
- The eigenvalue decomposition of the normalized graph Laplacian

The similarity stage can be trivially parallelized and requires read-only access to the input data, making it ideal for MPI parallelism.

Conversely, the eigenvalue decomposition step involves heavy linear algebra operations with data dependencies, making MPI considerably complex to integrate. External libraries that leverage OpenMP multithreading are used to accelerate this step.

The final k-means algorithm can be parallelized using a combination of both approaches.

3.3. Gaussian Similarity Matrix

The Gaussian similarity matrix construction is computationally intensive. The input matrix is divided into chunks of consecutive rows and each chunk is assigned to one of the MPI processes, in order to distribute data evenly among processes.

Then, each process computes the similarity scores of each point (row) of its assigned chunk, by applying the Gaussian kernel to it and every other point. Each scores row corresponds to one row of the similarity matrix, as shown in Figure 2.

Finally, the similarity score rows are gathered together, obtaining the similarity matrix, which can be assembled on the root process.

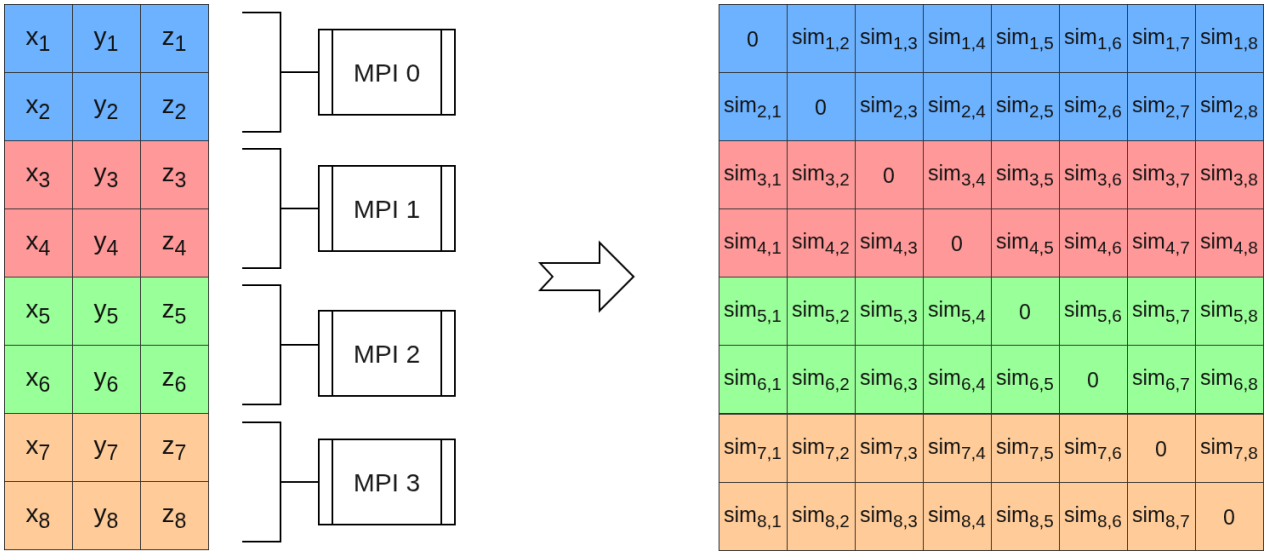

Figure 2. Diagram representing the parallel construction of the Gaussian similarity matrix (on the right), starting from the input matrix (on the left).

This stage benefits from parallelization because the input is perfectly divisible into chunks and communication occurs only after local chunks are computed. It is important to note that the tested datasets are always of a size divisible by the number of processes, so that all chunks have the same size and no rows are left out.

3.4. Diagonal Degree Matrix and Normalized Graph Laplacian

The degrees vector is then calculated on the root process with multiple threads and distributed among the processes, so that they can compute the diagonal matrix values by calculating the inverse of the square root of the degrees. This is done by dividing the degrees vector into chunks in order to distribute the workload, just as it was done in the similarity step.

After gathering the diagonal matrix, which is stored only as a vector containing the non-zero diagonal values, the normalized graph Laplacian is evaluated on the root process.

3.5. Eigenvalue Decomposition and Normalization

The first k eigenvectors of the Laplacian matrix are found via decomposition to construct the embedding matrix. The eigenvalue computation is not parallelized with MPI for its complexity. Instead of a distributed eigensolver, the algorithm computes only the first k eigenvectors using iterative methods.

The time complexity of this step is approximately $O(n^2k)$, which is sufficient for the problem sizes considered. However, since the computation runs on a single process, it becomes the primary scalability bottleneck.

After eigenvectors are computed, the embedding matrix is redistributed across MPI processes, so that each process can independently normalize a subset of its rows.

3.6. K-Means Clustering

The final k-means stage is parallelized using both MPI and OpenMP. Data points are partitioned across MPI processes. Meanwhile, the labels assignment step and the computation of new centroids are accelerated with OpenMP.

This stage scales well because it involves independent distance evaluations from centroids when assigning labels and synchronization is required only when merging centroid updates.

3.7. Summary

- **MPI-parallelized:** similarity matrix computation, diagonal values computation, embedding matrix normalization, k-means points partitioning.
- **OpenMP-parallelized:** degrees vector computation, normalized graph Laplacian computation, eigenvalue decomposition, k-means distances computation and centroids updates.
- **Not parallelized:** similarity inner loops, k-means inner loops and clusters split.

4. PARALLEL IMPLEMENTATION

This section describes how significant design decisions are reflected in the algorithm codebase through MPI communication and OpenMP directives. The source code is developed with C++. Eigen [6] and Spectra [7] are the external libraries used for algebraic operations and eigen decomposition, respectively.

Eigen natively supports OpenMP multithreading by setting the number of threads after MPI initialization: `Eigen::setNbThreads(omp_get_max_threads())`. Spectra is based on Eigen, but adds more advanced and efficient algorithms to it.

4.1. MPI Gaussian Similarity Matrix

Each MPI process computes a subset of rows of the similarity matrix using `evaluate_gaussian_similarity_values`. The partitioning is based on the process rank.

```
int count = n / world_size;
int l = count * world_rank;
int r = count * (world_rank + 1);

std::vector<double> local_similarity_values = evaluate_gaussian_similarity_values(X, l, r, sigma);
```

Code 1. MPI input partitioning for similarity computation.

`n` is the number of input points (rows), while `count` represents the number of points handled by each process; `l` is the global index of the first point handled by the process and `r` is the index of the point after the last point handled by the process.

After Gaussian similarity computation, local similarity chunks are gathered row-by-row on the root process to reconstruct the full matrix.

```
for (int c = 0; c < count; ++c) {
    MPI_Gather(local_similarity_values.data() + c * n, n, MPI_DOUBLE,
               global_similarity_values.data(), n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
    for (int r = 0; r < world_size; ++r) {
        std::memcpy(similarity_matrix.row(r * count + c).data(),
                    global_similarity_values.data() + r * n, n * sizeof(double));
    }
}
```

Code 2. Global Gaussian similarity matrix gathering.

The gathering is done row-by-row, rather than all at once, because the largest input sizes caused the indexing of the global similarity matrix to overflow the MPI integer type. However, this solution requires to gather `local_similarity_values` inside a temporary buffer `global_similarity_values` before copying them into the correct position in the similarity matrix with `std::memcpy`.

The computation and gathering of the diagonal values is done in a similar way, but in this case the temporary buffer is not necessary because only the diagonal points are stored into a vector and the full diagonal matrix is not constructed.

4.2. Eigen + OpenMP Normalized Graph Laplacian

The normalized graph Laplacian is then built and symmetrized. `asDiagonal()` call is necessary to operate with `degrees_vector` as if it was a full matrix.

```
Matrix L = degrees_vector.asDiagonal() * similarity_matrix * degrees_vector.asDiagonal();
L = Matrix::Identity(n, n) - L;
L = 0.5 * (L + L.transpose());  // make sure L is symmetric
```

Code 3. Normalized graph Laplacian construction.

These operations rely on Eigen's internal OpenMP parallelization, scaling with the number of threads per MPI process.

4.3. Eigenvalue Decomposition with Spectra

The function `compute_first_k_eigenvectors` of the proposed algorithm calls `Spectra::SymEigsSolver` which has a $O(n^2k)$ time complexity by implementing an iterative method for eigenvectors computation.

Table 3. Time complexity of tested algorithms for finding the first k eigenvectors.

Algorithm	Matrix memorization	Time complexity
Eigen eigensolver	Dense	$O(n^3)$
Spectra eigensolver	Dense	$O(n^2k)$

Eigen eigensolver actually computes full eigenvalue decomposition and not only the first k eigenvectors.

In previous versions of the code `Eigen::SelfAdjointEigenSolver` was employed, but it caused an unbearable bottleneck due to its time complexity which is cubic with respect to n.

Since Spectra uses Eigen structures and operations under the hood, its eigensolver is expected to leverage multithreading. Nevertheless, this stage is by far the most time-consuming.

4.4. MPI Embedding Matrix Normalization

The row vectors of the embedding matrix are scattered and normalized locally. They are later gathered on all processes with a call to `MPI_Allgather`.

```
MPI_Scatter(global_eigenvectors.data(), count * k, MPI_DOUBLE,
            local_eigenvectors.data(), count * k, MPI_DOUBLE, 0, MPI_COMM_WORLD);

for (int i = 0; i < count; ++i) {
    local_eigenvectors.row(i).normalize();
}

MPI_Allgather(local_eigenvectors.data(), count * k, MPI_DOUBLE,
              global_eigenvectors.data(), count * k, MPI_DOUBLE, MPI_COMM_WORLD);
```

Code 4. Parallel normalization of embedding matrix.

4.5. Hybrid K-Means Implementation

At the beginning of each iteration the latest centroids are broadcasted to all MPI processes. Each process assigns the labels to a subset of points by calling `evaluate_labels`.

```
MPI_Bcast(global_centroids.data(), k * m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
local_labels = evaluate_labels(U, global_centroids, l, r);
```

Code 5. Centroids broadcast and labels assignment partitioning in k-means.

The labels assignment loop is parallelized with OpenMP using static scheduling to pre-divide iterations into chunks.

```
#pragma omp parallel for schedule(static)
for (int i = l; i < r; ++i) {
    [...]

    labels[i - l] = label;
}
```

Code 6. Labels assignment step in k-means accelerated with OpenMP.

Finally, the labels are gathered inside the root process which calculates the new centroids and eventually split clusters to avoid empty clusters.

OpenMP parallelization is also used to accelerate the computation of new centroids.

4.6. Memory Optimization

Large temporary containers are explicitly deallocated after use to reduce memory usage and contrast the memory bottleneck caused by storing the similarity and Laplacian matrices:

```
local_similarity_values = {};
degrees.resize(0);
```

5. PERFORMANCE ANALYSIS

Testing is fundamental to assess the performance, efficiency and scalability of the proposed parallel algorithm.

The algorithm was tested on the HPC cluster at the University of Trento, which uses PBS and comprises 126 machines. There are 6 092 processors in total, with a maximum of 96 processors per machine.

In principle, we wanted to distribute the processes among multiple nodes, but due to the limited available resources of the system it was possible to use only one node at the time. For every test 1 TB of memory was used. We used the `pack:excl` PBS strategy to maximize the available memory bandwidth. The system imposed a six hours time limit for job executions.

5.1. Testing Phase

The algorithm was tested with the six datasets mentioned above using up to 64 processors and with the following OpenMP configurations:

- 1 OpenMP thread: 1 processor is assigned to each MPI process.
- 2 OpenMP threads: 2 processors are assigned to each MPI process.
- 4 OpenMP threads: 4 processors are assigned to each MPI process.
- 8 OpenMP threads: 8 processors are assigned to each MPI process.

The number of MPI processes can be obtained dividing the number of processors by the number of OpenMP threads. Testing showed that execution time depends slightly on the dataset class. Therefore, only Gaussian tests are reported and discussed.

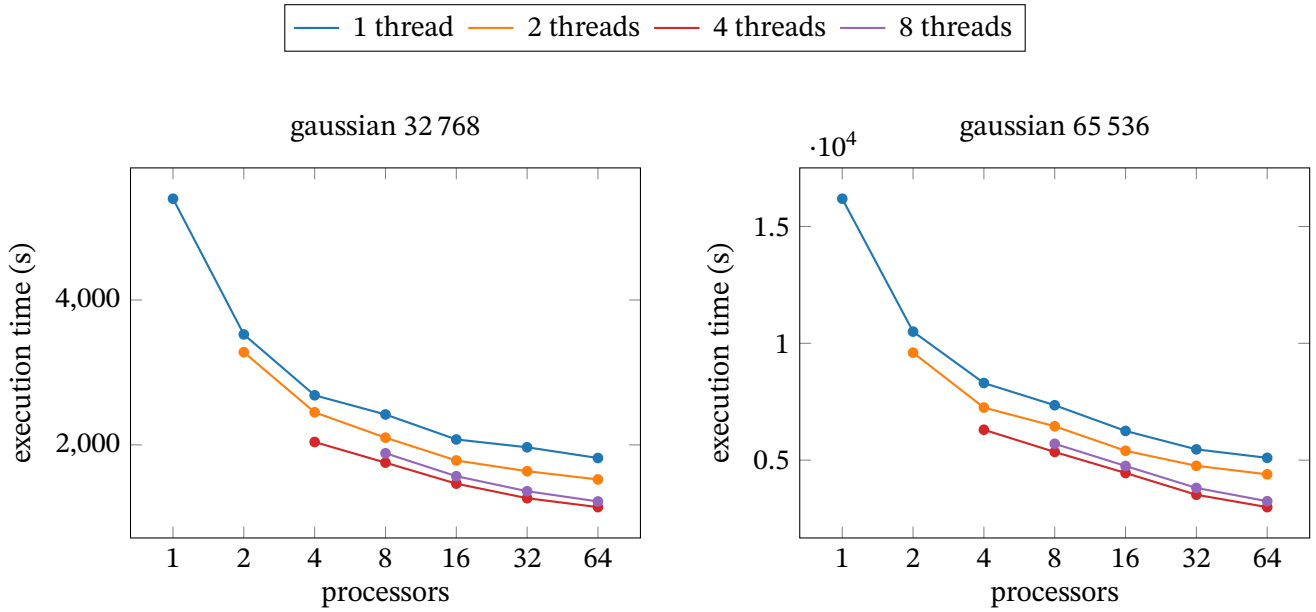

Figure 3. Execution time over number of processors, with the different OpenMP threads configurations.

The execution times of gaussian 32 768 and gaussian 65 536 tests are reported in Figure 3. The 131 072 test did not finish for all configurations and number of processors due to the six hours time constraint, in particular it was

not possible to calculate the speedup and efficiency for this dataset due to the baseline time missing. For this reason, its partial results are not reported here, but they are available in the repository.

It is easy to see that the 4 OpenMP threads configuration is the one performing the best, probably because it is the best trade-off between code that is parallelized with MPI, such as the similarity matrix construction, and code that is parallelized with OpenMP, such as the Laplacian calculation and eigen decomposition.

5.2. Speedup Evaluation

Speedup is defined as the ratio of the execution time of the serial algorithm to that of the parallel version.

$$S = \frac{T_{serial}}{T_{parallel}} \tag{8}$$

Since running the serial algorithm on the system required additional setup, the execution time of the parallel version with one processor, which approximates the serial algorithm, was used as the baseline for calculating the speedup.

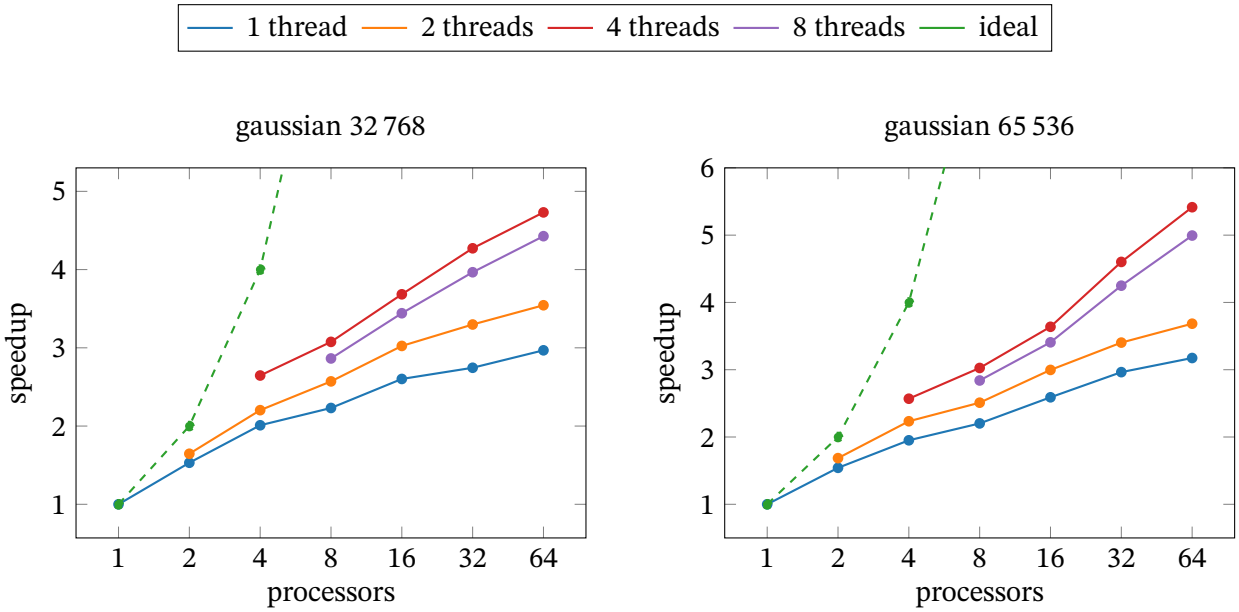

Figure 4. Speedup over number of processors, with the different OpenMP threads configurations.

As expected, the speedup steadily increases as the number of processors increases, with a slight flattening between 32 and 64 processors. Overall, the speedup is moderate with values going up to 4.731 for the gaussian 32 768 dataset and up to 5.415 for the gaussian 65 536 dataset, as shown in Figure 4.

The modest speedup results reflect the complexities discussed in previous sections. The speedup increase is linear rather than exponential, most probably due to the time bottleneck caused by eigen decomposition and the memory bandwidth saturating when transferring large chunks of the similarity matrix.

5.3. Efficiency and Scalability Evaluation

Efficiency is defined as speedup divided by the number of processors.

$$E = \frac{S}{P} \tag{9}$$

The efficiency results of the two analyzed datasets are reported in Figure 5. The results of the two datasets are comparable, with the largest input having slightly higher efficiency values.

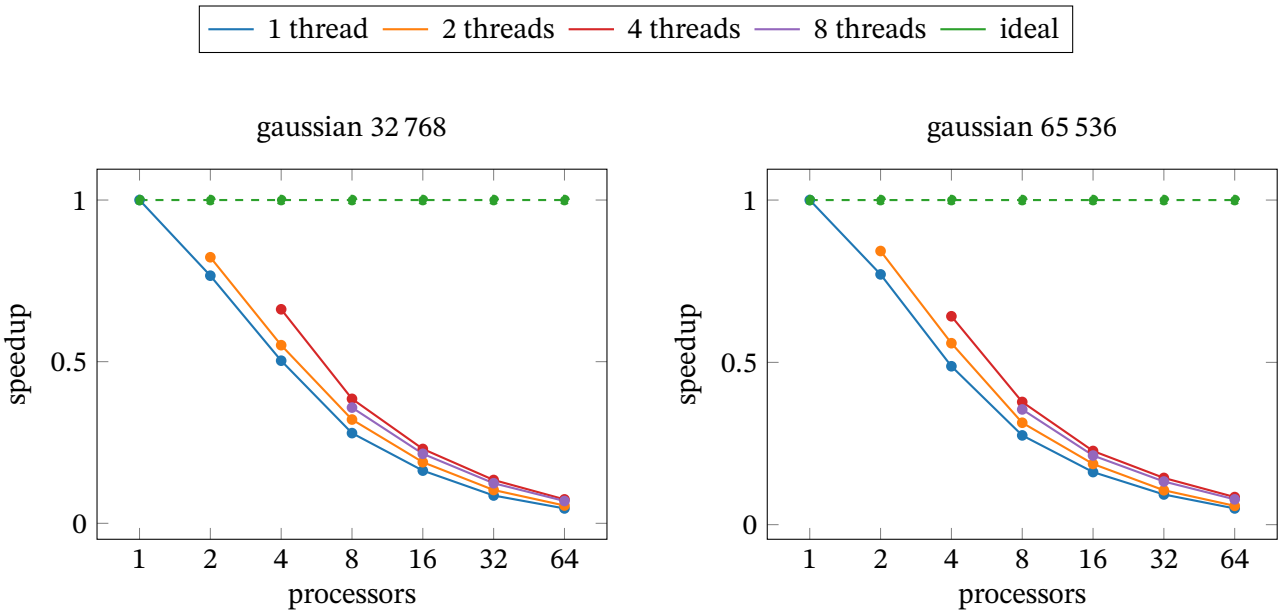

Figure 5. Efficiency over number of processors, with the different OpenMP threads configurations.

Scalability measures the rate at which efficiency changes as the number of processors increases. Parallel programs that can maintain a constant efficiency without increasing the problem size have a strong scalability. Parallel programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes have a weak scalability.

Although the efficiency increases slightly with larger inputs, the proposed algorithm does not demonstrate strong or weak scalability.

Table 4. Speedup and efficiency for the gaussian 65 536 dataset.

Processors	OpenMP threads							
	1		2		4		8	
	S	E	S	E	S	E	S	E
1	1.000	1.000	–	–	–	–	–	–
2	1.542	0.771	1.686	0.843	–	–	–	–
4	1.951	0.488	2.234	0.559	2.569	0.642	–	–
8	2.202	0.275	2.510	0.314	3.026	0.378	2.840	0.355
16	2.590	0.162	2.997	0.187	3.638	0.227	3.407	0.213
32	2.964	0.093	3.403	0.106	4.602	0.144	4.249	0.133
64	3.174	0.050	3.684	0.058	5.415	0.085	4.994	0.078

Speedup is measured relative to the parallel execution with 1 processor. Efficiency is computed as speedup divided by the number of processors.

CODEBASE REPOSITORY

https://github.com/guglielmo-boi/hybrid-spectral-clustering

The source code was developed with C++, MPI and OpenMP, using Eigen and Spectra external libraries. The build system is CMake. Some Python scripts were made to generate datasets, plot clustering data, and measure ARI accuracy. Input and output data is available in folder `data`, while complete logs of the testing phase with all configurations can be found in folder `logs`.

PERSONAL REFLECTIONS

Developing the algorithm was fairly complex. Implementing the serial version alone was quite challenging due to its mathematical complexity.

It was convenient to restrict the problem to 3D data points because visualizing data was helpful during development.

Although the testing phase required many steps to set up the automated system for launching tests and retrieving related logs, it was very valuable in the end.

While the algorithm's performance was underwhelming, its analysis led to critical comments about the work.

REFERENCES

[1] P. Michaud, "Clustering techniques", *Future Generation Computer Systems*, vol. 13, no. 2-3, pp. 135–147, 1997.

[2] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering in distributed systems", *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 3, pp. 568–586, 2010.

[3] U. Von Luxburg, "A tutorial on spectral clustering", *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[4] A. Ng, M. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm", *Advances in neural information processing systems*, vol. 14, 2001.

[5] C. Fowlkes, S. Belongie, F. Chung, and J. Malik, "Spectral grouping using the nystrom method", *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 2, pp. 214–225, 2004.

[6] G. Guennebaud and B. Jacob, *Eigen: A c++ template library for linear algebra*, Accessed: 2026-02-05, The Eigen Developers, 2025. [Online]. Available: https://libeigen.gitlab.io/.

[7] Y. Qiu, *Spectra: A c++ library for large scale eigenvalue problems*, Accessed: 2026-02-05, The Spectra Developers, 2025. [Online]. Available: https://spectralib.org/.