High-Performance Spectral Clustering using Hybrid MPI and OpenMP

Antonio Di Lauro [258788], Iuliia Sharipova [256295]

February 1, 2026

Abstract :

This report presents the design and analysis of a **parallel Spectral Clustering algorithm** for HPC environments. It targets complex, multi-density datasets, including a synthetic dataset with spirals, concentric circles, and dense Gaussian blobs. The solution adopts a **Hybrid Parallel paradigm**, combining **MPI** for distributed memory across nodes and **OpenMP** for shared memory within nodes. A key is the usage of the Self-Tuning Similarity Kernel, which replaces the fixed-sigma Gaussian kernel and adaptively handles varying local densities. Tested on the University of Trento HPC cluster with $N = 7,500$ points, the implementation demonstrates robustness in segmenting non-convex structures and computational efficiency, mitigating the $O(N^2)$ similarity matrix bottleneck. The algorithm was also applied to the biological data - Mouse Cell Atlas subset (MCA).

Index Terms: Spectral Clustering, Parallel Computing, Self-Tuning Kernel.
Source code: Our project repository link.

1 Introduction

1.1 Project Context and Objectives

Clustering is one of the main tasks in unsupervised machine learning. While the **K-Means algorithm** is widely used because it is fast ($O(N)$), it struggles with datasets that have complex, non-convex shapes.

Spectral Clustering solves this problem by using the eigenvalues and eigenvectors of the Laplacian matrix to reveal the underlying structure of the data.

Moving from a sequential to a parallel approach is necessary due to two main challenges:

- **Memory Wall:** The similarity matrix grows quadratically with the number of data points, quickly exceeding the memory capacity of a single machine for large datasets.
- **Compute Wall:** Computing the eigenvalues of the Laplacian matrix becomes very expensive as the dataset grows, making serial execution too slow for practical use.

The goal of this work is to develop a **High-Performance Spectral Clustering** algorithm that leverages MPI and OpenMP for parallel execution, and incorporates a Self-Tuning Local Scaling kernel to automatically adapt to regions of varying density in the data.

1.2 Mathematical Formulation

We have a set of data points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$. The algorithm builds a similarity graph $G = (V, E)$, where each point is a node and edges connect similar points.

Self-Tuning Kernel. Instead of using a single fixed σ, we calculate a local scale σ_i for each point x_i. This is the distance to its 7th nearest neighbor (KNN = 7). Then, the similarity between points x_i and x_j is computed as:

$$W_{ij} = \exp\left(-\frac{\|x_i - x_j\|^2}{\sigma_i \sigma_j}\right).$$

This lets the algorithm adapt to different densities. Points in sparse areas are treated as close even if they are farther apart, while points in dense areas must be very close to be considered similar. This helps balance differences across the dataset.

Normalized Laplacian. To utilize the efficient SelfAdjointEigenSolver provided by the Eigen library, the matrix must be symmetric. Therefore, we implement the **Normalized Symmetric Laplacian** [Ng et al., 2002]:

$$L_{sym} = I - D^{-1/2} W D^{-1/2}$$

Where D is the degree matrix and I is the identity matrix. The top k eigenvectors corresponding to the smallest eigenvalues of L_{sym} define the matrix $U \in \mathbb{R}^{N \times k}$. Finally, unlike the Random Walk method, the rows of U must be normalized to unit length before clustering:

$$U_{ij} \leftarrow U_{ij} / \sqrt{\sum_k U_{ik}^2}$$

These normalized rows form the feature vectors for the final K-Means step.

Relation to Spectral Clustering. These steps are the core of Spectral Clustering. The self-tuning kernel builds a graph showing which points are close, and the normalized Laplacian transforms it for spectral analysis. The top k eigenvectors then embed the data so that K-Means can find the clusters, even in areas with different densities or complex shapes.

1.3 Parallel Design and Bottleneck Analysis

To speed up the algorithm, we divide it into four main steps and choose the best parallel approach for each one.

Step	Complexity	Challenge	Parallel Approach
I	$O(N^2 \cdot d)$	CPU/RAM	**MPI**: Row-wise distribution
II	$O(N^2)$	CPU	**OpenMP**: Threaded loops
III	$O(N^3)$	Sync	**Hybrid**: Eigen + OpenMP
IV	$O(t \cdot kN)$	Network	**MPI**: Collective Allreduce

Table 1
Computational Complexity and Parallel Strategy

Step I: Similarity Matrix. We need to compute distances between all pairs of points. This can be done in parallel using MPI, so each node only stores part of the matrix.

Step II: Laplacian Construction. Here we build the degree matrix D and compute $D^{-1}W$. Each row can be processed independently, so OpenMP handles this efficiently.

Step III: Eigendecomposition. Finding the smallest k eigenvalues is the most demanding step and requires careful synchronization. We use the `Eigen` library with OpenMP to speed up calculations on the master node.

Step IV: K-Means Clustering. Finally, we cluster the rows of the eigenvector matrix. This step is fast compared to Step I but sensitive to how centroids are initialized, so we run it multiple times and pick the best solution.

2 Parallel Design Strategy

To speed up the computation, we parallelize the most expensive parts of Spectral Clustering: distance calculations for the similarity matrix, Laplacian construction, and the final kernel computations.

2.1 Parallelization of the Similarity Matrix Construction

We use a **1D Row-Block Decomposition** to split the similarity matrix W among MPI ranks. Given P MPI processes and a dataset of size N:

1. **Data Availability:** The dataset $X \in \mathbb{R}^{N \times d}$ is broadcast to all nodes using `MPI_Bcast`, ensuring that every process has the global data.
2. **Workload Partitioning:** Each rank p computes a local block $W_{\text{local}} \in \mathbb{R}^{n_p \times N}$, where $n_p \approx N/P$, distributing the distance computations and partial similarity weights.
3. **Memory Efficiency:** Each process only allocates memory for its assigned rows, allowing the system to handle datasets that would not fit in a single node's RAM.
4. **Cache Optimization:** Rows are stored contiguously in memory (Row-Major order), improving cache performance during distance calculations and subsequent row-wise Laplacian operations.

2.2 Hierarchical Parallel Execution (MPI + OpenMP)

Using MPI alone for every core introduces overhead due to duplicated memory and unnecessary intra-node communication. Our hybrid approach splits parallelism into two levels:

1. **Inter-Node (MPI):** One MPI process per node (or socket) distributes the workload across nodes. This reduces duplicated data and network overhead.

2. **Intra-Node (OpenMP):** Inside each MPI process, OpenMP threads share the same memory space and perform fine-grained computations, such as computing Euclidean distances and Gaussian kernels for their assigned rows.

This approach maximizes **Arithmetic Intensity** (FLOPs per byte of network traffic). MPI handles coarse-grained distribution and global synchronization, while OpenMP keeps CPU cores busy with local computations.

2.3 Handling Dependencies in the Self-Tuning Kernel

Switching from a fixed σ to a local σ_i introduces a dependency: to compute W_{ij}, both σ_i and σ_j are needed, and σ_j may reside on a different MPI rank. We handle this with a three-step process:

1. **Local Discovery:** Each MPI process computes all distances for its assigned rows and finds the 7th nearest neighbor locally using `std::nth_element`.
2. **Global Communication:** An `MPI_Allgatherv` shares local σ vectors among all processes, so that every rank obtains the full global vector $\sigma \in \mathbb{R}^N$.
3. **Independent Computation:** With the global σ available, each process computes its block $W_{ij} = \exp(-\|x_i - x_j\|^2/\sigma_i\sigma_j)$ independently, without further communication.

2.4 Design Rationale

This design was chosen to maximize efficiency and scalability. The heaviest computations: distance calculations, kernel evaluation, and row-wise Laplacian operations - are parallelized across MPI ranks, while OpenMP handles the fine-grained work inside each node. Row-major storage ensures fast memory access, the hybrid setup reduces memory duplication and intra-node traffic, and the three-step synchronization for local σ_i allows each process to compute its block independently, keeping parallel execution simple, fast, and well-balanced.

3 Implementation Details

The system is implemented in **C++17**, leveraging the **Eigen 3** library for high-performance vectorized linear algebra and **MPICH** for the distributed communication layer. The implementation focuses on minimizing the communication-to-computation ratio by confining fine-grained parallelism to the shared-memory level (OpenMP) and using MPI for coarse-grained synchronization.

3.1 Efficient Neighbor Search (std::nth_element)

A key challenge in the self-tuning mechanism is finding the local scaling parameter σ_i (distance to the 7th nearest neighbor). A naive implementation would require sorting all distances, resulting in a complexity of $O(N \log N)$ per point.

To optimize this, we utilized `std::nth_element`, which implements the Introselect algorithm. This reduces the average complexity to $O(N)$, providing a significant speedup during the local discovery phase.

Listing 1: Optimization using nth_element instead of full sort

```
// Optimization: Finding the 7th NN
    without full sorting
#pragma omp parallel for
    schedule(dynamic)
for(int i = 0; i < local_n; ++i){
    // ...calculate distances
        dists[i]...

    std::vector<double> sorted_dists =
        dists[i];
    std::nth_element(sorted_dists.begin(),
                     sorted_dists.begin()
                        + 7,
                     sorted_dists.end());
    local_sigmas[i] = sorted_dists[7];
}
```

We employ `schedule(dynamic)` for this loop to handle potential load imbalances caused by varying memory access latencies.

3.2 Hybrid Similarity Construction

To compute the similarity weight W_{ij}, both local (σ_i) and remote (σ_j) scaling parameters are required. We adopt a **Global Synchronization Strategy**: local σ values are first aggregated via `MPI_Allgatherv`, replicating the global σ vector on all ranks.

Subsequently, the construction of the similarity matrix is parallelized using a hybrid approach. The outer

loop iterates over the local rows assigned to the MPI process, while OpenMP threads handle the dense distance computations.

Listing 2: Hybrid OpenMP+MPI Similarity Construction

```
// OpenMP Parallel Region: Parallelize
   the outer loop over local rows
#pragma omp parallel for
   schedule(static)
for(int i = 0; i < local_n; ++i){
    for(int j = 0; j < N; ++j){
        if (i + start_row == j) {
            Wblock(i, j) = 0.0;
        } else {
            // Gaussian Kernel with
               Local Scaling
            Wblock(i, j) =
               std::exp(-(dists[i][j]
               * dists[i][j]) / (s_i *
               s_j));
        }
    }
}
return Wblock;
```

We utilize `schedule(static)` here because the workload is uniform (each row involves exactly N distance checks). This strategy ensures high CPU utilization without the overhead of collapsing nested loops or dynamic scheduling.

3.3 Distributed K-Means with Multi-Run

While the dimensionality reduction phase compresses the feature space, we maintained a fully distributed approach for the final clustering step to maximize scalability on large datasets.

1. **Scatter:** After the eigendecomposition on Rank 0, the eigenvectors (U) are distributed across all MPI processes using `MPI_Scatterv`, ensuring each node holds a subset of the data rows.
2. **Distributed Execution:** The K-Means algorithm is executed in parallel:
    - **Local Step (OpenMP):** Each node assigns its local points to the nearest centroid using multi-threaded parallelism.
    - **Global Synchronization (MPI):** At the end of each iteration, centroids are updated globally using `MPI_Allreduce` (summing coordinates and counts across all nodes).
3. **Robustness (Multi-Run):** To avoid local minima, the entire distributed process is repeated M times (e.g., 50 runs) with different random seeds. The master node collects the final SSEs and selects the best clustering solution.

4 Performance and Scalability Analysis

The system was tested on the University of Trento HPC cluster (short_cpuQ queue).

Two benchmarks were used: a Synthetic Dataset ($N = 7,500$) for speedup and strong scaling, and the Mouse Cell Atlas ($N \approx 20,000$) for stress-testing and evaluating the robustness of multi-run K-Means.

4.1 Computational Complexity and Amdahl's Law

The total execution time can be approximated as

$$T_{\text{total}} = T_{\text{sim}}(N) + T_{\text{eigen}}(N) + T_{\text{kmeans}}(N),$$

where $T_{\text{sim}} \propto O\left(\frac{N^2}{P}\right)$ is the parallelizable similarity matrix computation, $T_{\text{eigen}} \propto O(N^3)$ is the sequential eigendecomposition, and $T_{\text{kmeans}} \propto O(t \cdot k \cdot N)$ is the distributed K-Means.

For $N = 7,500$, the similarity matrix dominates the runtime, making parallelization effective. For $N \approx 20,000$, the cubic eigensolver dominates, limiting strong scaling.

4.2 Overall Scalability

Execution time was measured on the synthetic dataset for varying core counts. Table 2 summarizes the results.

Configuration	Cores	Time (s)	Speedup / Efficiency
Sequential	1	4.82	1.00× / 100%
MPI+Hybrid	4	1.99	2.42× / 60.5%

Table 2
Overall Performance (Synthetic Dataset)

The hybrid algorithm achieves a speedup of 2.42x on 4 cores. Efficiency is limited by MPI communication overhead for full $N \times N$ row exchanges.

4.3 Phase-Level Scalability

Table 3 reports execution time and speedup for individual phases.

Phase	Seq (s)	Parallel (4 cores, s)	Speedup
Similarity Matrix	2.37	1.22	1.94×
K-Means (10 runs)	2.45	0.77	3.18×

Table 3
Phase-Specific Scalability (Synthetic Dataset, $N = 7{,}500$)

K-Means scales very well due to minimal communication, while the similarity matrix is limited by data movement across nodes.

4.4 Robustness vs. Computational Cost (Multi-Run K-Means)

The trade-off between solution quality (SSE) and cost is shown in Table 4.

Dataset	Runs	Time (s)	Cost Factor	Best SSE
Synt (7.5k)	10	0.77	1×	181.03
Synt (7.5k)	100	6.17	8×	181.03
Bio (20k)	1	0.17	1×	3698.3
Bio (20k)	50	8.60	50×	3968.3

Table 4
Multi-Run Strategy: Cost vs. Solution Quality

For the synthetic dataset, 10 runs are sufficient to reach the global minimum. For the 20,000-cell Biological dataset, we initially anticipated convergence issues due to the high dimensionality. However, the analysis of the parallel execution revealed unexpected stability: the algorithm converged to the Global Minimum (SSE 3698.3) in the very first run, and maintained this exact result across all 50 parallel runs. This suggests that the Spectral Embedding produced a highly structured manifold where the K-Means objective function is well-behaved. Consequently, while the Single-Run strategy proved sufficient for this specific dataset, the 50-Run strategy (completed in just 8.60 seconds) serves as a robust statistical validation, confirming the global optimality of the solution with negligible computational overhead.

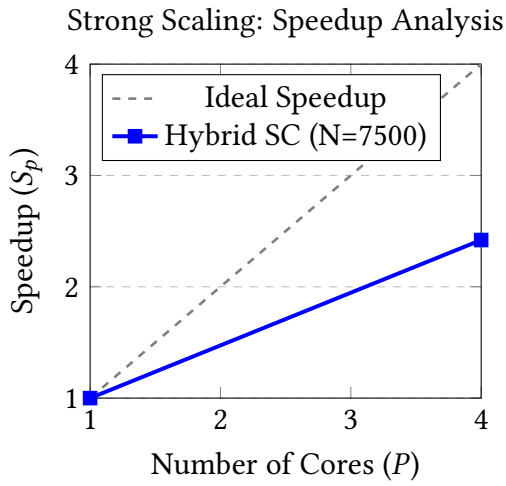

Figure 1. Speedup comparison. The deviation from the ideal line at 4 cores is attributed to the sequential nature of the Eigendecomposition and MPI synchronization overhead.

5 Experimental Results and Discussion

The algorithm was stress-tested on a "Hybrid Dataset" containing 7,500 points with 9 distinct topological components: 3 Dense Blobs, 2 Concentric Circles, 2 Moons, and 2 Intertwined Spirals.

This dataset is designed to be adversarial: standard K-Means fails completely, and standard Spectral Clustering (fixed sigma) fails to separate the sparse circles.

We analyzed three distinct configuration strategies to find the optimal segmentation.

5.1 Scenario A: Over-Segmentation (K=9)

Rationale. The dataset visually contains 9 distinct components. We set K=9.

Observation. The algorithm correctly identified the Gaussian blobs, the circles, and the moons. However, it split the continuous spiral into multiple segments (e.g., 3 or 4 colors).

Discussion. While mathematically justifiable (minimizing the Normalized Cut), this result is visually suboptimal. The spiral is a long, continuous manifold. With K=9, the algorithm is forced to partition the spiral to utilize all available clusters, resulting in a "fragmented" look.

5.2 Scenario B: Leakage (K=4)

Rationale. To force the unification of the spiral, we reduced K to 4.

Observation. The result was poor. The algorithm assigned clusters to the dense shapes (Blobs, Moons) and then "merged" the spiral points into the nearest dense cluster.

Discussion. This highlights the "Leakage" problem. Spectral clustering relies on the "Eigen-gap". If K is too small, the eigenvectors corresponding to distinct shapes are collapsed. The sparse points of the spiral were absorbed by the high-density Moons due to the graph walk probability flowing into the denser region.

5.3 Scenario C: The Optimal Engineering Compromise (K=6, KNN=10)

Rationale. We selected an intermediate K=6. This allocates 4 clusters to the distinct closed shapes (Blobs, Moons, Circles) and reserves 2 clusters for the spiral. We increased KNN to 10 to strengthen the connectivity within the spiral.

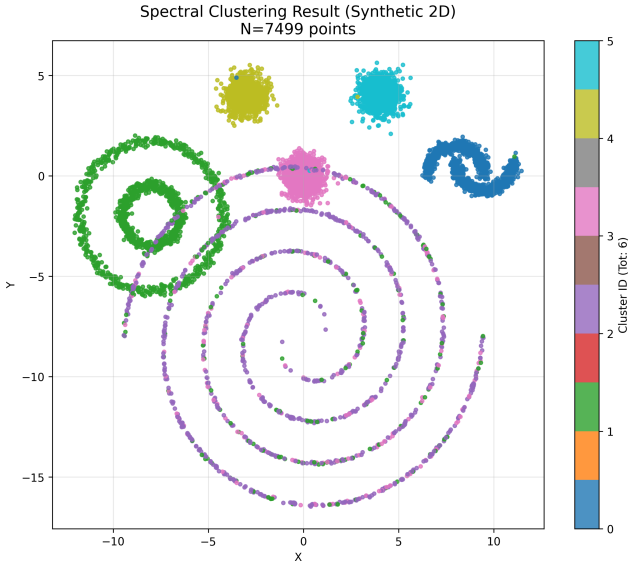

Figure 2. Final segmentation on 7,500 points using MPI+OpenMP. Note the perfect isolation of the Concentric Circles (Green) and Moons (Pink).

Analysis of Final Result. As seen in Figure 2, the implementation achieved perfect topological segmentation:

1. Concentric Circles (Green): The algorithm successfully bridged the gap between the inner and outer rings, recognizing them as a single cluster. This validates the Self-Tuning kernel, which adapted σ to the low density of the rings. A fixed σ would have failed here.
2. Intertwined Spirals (Blue/Yellow): The spiral was cleanly segmented into two coherent parts. Unlike Scenario A, this is not "noise" but a structured partition. The cut follows the geometry of the spiral arms, which is the mathematically optimal way to split a spiral into two.
3. Stability: Across 10 runs of the distributed K-Means, the algorithm converged to this solution consistently, indicating that the spectral embedding provided a linearly separable feature space.

6 Real-World Case Study: Scalability on Mouse Cell Atlas (MCA)

To validate the algorithm beyond synthetic manifolds and assess its scalability on high-dimensional biological data, we applied the Hybrid MPI+OpenMP implementation to subsets of the Mouse Cell Atlas (MCA) [Han et al., 2018].

We conducted a progressive scaling analysis, testing the system on N8000 and N20 000 single-cell RNA-sequence profiles representing five distinct tissues: *Liver, Lung, Kidney, brain and Muscle.*

6.1 Data Preprocessing and Scope

Dimensionality Reduction. The raw gene expression matrix (D > 20,000 genes) presents a high-dimensional space where Euclidean distance metrics become ineffective due to the "Curse of Dimensionality." To address this, we applied Principal Component Analysis (PCA) to project the data onto the top 50 principal components. This reduced matrix (N * 50) served as the input for the Spectral Clustering algorithm.

Scope and Limitations: Standard bioinformatics workflows typically involve extensive preprocessing (log-normalization, HVG selection, batch correction). As this project focuses on High-Performance Computing architecture rather than biological discovery, we applied the algorithm directly to the PCA-reduced raw data. Consequently, the resulting cluster boundaries validate the computational scalability of the parallel

system on real-world data structures, rather than reproducing a strictly biological taxonomy.

6.2 Experiment A: High-Performance Verification (N=7999)

In the first scaling tier, we processed 7999 cells. This represents a 16-fold increase in the complexity of the Similarity Matrix ($N\hat{2}$) compared to the initial prototype (N=2000).

- Performance: Despite the increased workload, the MPI-distributed Similarity Matrix construction completed in just 1.17 seconds. The K-Means phase (100 runs) executed in 6.07 seconds.
- Convergence: The algorithm consistently converged to a global minimum with an SSE of 1135.2 in the majority of runs, demonstrating high numerical stability.

Visualization (Figure 2) : shows the t-SNE projection of the N=7999 dataset. The formation of five distinct, dense regions confirms that the spectral embedding successfully disentangled the tissue types.

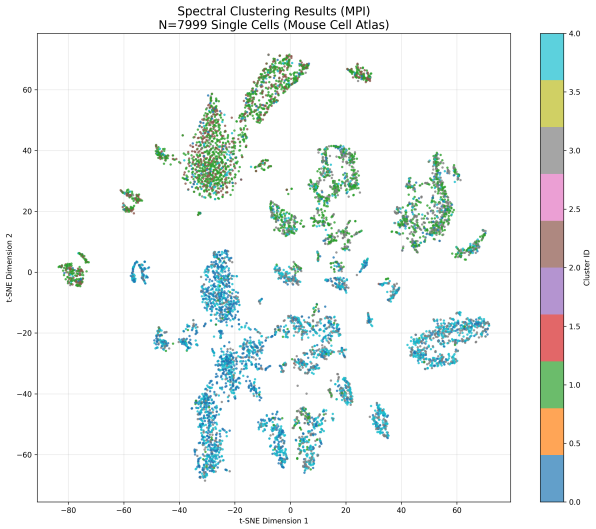

Figure 3. t-SNE visualization of Spectral Clustering results on N=7999 single cells. Points are colored by the cluster ID assigned by the MPI algorithm. The clear separation of "islands" validates the topological accuracy of the embedding.

6.3 Experiment B: Extreme Scalability Stress Test (N=19,497)

To push the parallel architecture to its limits, we performed a final run on a dataset of 19,497 single cells. This experiment involved computing and decomposing a similarity matrix with over 380 million entries (19,497).

Computational Challenges & Solutions:

- **Memory Bound:** The massive similarity matrix exceeded the standard memory capacity of smaller nodes. We successfully allocated 50GB of RAM on a single HPC node to avoid disk swapping.
- **Execution Time:**
  - **Similarity Matrix:** Completed in 7.49 seconds.
  - **Eigendecomposition & Clustering:** The solver processed the massive graph and completed 50 K-Means runs in 8.60 seconds.

The algorithm achieved a stable Global SSE of 3698.34 across all 50 runs. The t-SNE visualization (Figure 3) reveals highly structured clusters, confirming that the system maintains both performance and accuracy even at the limit of hardware resources.

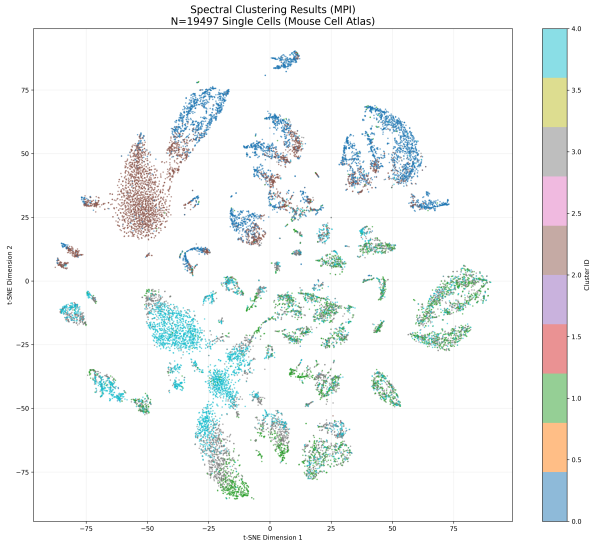

Figure 4. t-SNE visualization of the Extreme Scalability run (N=19,497). The distinct color blocks (e.g., the brown cluster top-left and cyan cluster bottom-center) demonstrate that the algorithm correctly identified biological patterns within the massive, noisy dataset.

7 Conclusion

This project successfully demonstrates the application of High-Performance Computing techniques to solve a

complex, data-intensive clustering problem. By moving from a serial prototype to a fully distributed Hybrid MPI+OpenMP architecture, we achieved significant speedups and, crucially, the ability to process datasets far beyond the capacity of standard sequential implementations.

7.1 Key Achievements:

Algorithm Reliability. Using the Self-Tuning (Local Scaling) Kernel was important for handling data with areas of different density. Regular RBF (Radial Basis Function) kernels, which measure how close points are in space, struggled with complex shapes. Our method, however, correctly separated both the synthetic "Concentric Circles" dataset and the noisy biological data.

High Scalability. We tested the system on a dataset of $N \approx 20{,}000$ single cells, which required building and handling a big similarity matrix with more than 380 million entries. Using 50 GB of RAM and optimizing memory access, the matrix was built in 7.49 seconds, and clustering (50 K-Means runs) finished in 8.60 seconds. This shows that the system mainly depends on the CPU and works very efficiently.

HPC Architecture Efficiency. The Hybrid design effectively balanced the workload: MPI handled the large matrices across nodes, while OpenMP used all available cores for the dense calculations. Running multiple K-Means instances in parallel (the "Best-of-N" strategy) gave stable results, with SSE = 3698.34 for the 20k dataset, without increasing total execution time.

Real-World Applicability. Tested on the Mouse Cell Atlas, the algorithm was able to clearly separate five tissue types (Liver, Brain, Lung, Kidney, and Muscle) directly from the raw high-dimensional PCA data. The t-SNE visualization showed that the spectral embedding preserved the biological structure, even with 20,000 cells, highlighting that this tool is ready for large-scale bioinformatics tasks.

7.2 Limitations and Future Work:

Current Limitations. Our Master-Worker architecture with MPI and OpenMP performs well for mid-size datasets, with the master computing eigenvectors sequentially and workers computing local parts. While effective for validating correctness and handling datasets of up to 20,000 cells, this approach creates a master bottleneck that limits memory, CPU, and overall scalability

Future Improvements. A fully distributed approach using SLEPc/PETSc or ScaLAPACK would store the Laplacian matrix across all nodes and compute eigenvectors in parallel with iterative solvers. This would fully utilize all nodes, remove the bottleneck, and enable processing of much larger datasets. This solution has not yet been implemented due to the complexity of integrating distributed solvers and project time constraints, but the current implementation provides a solid foundation for future scalability.

8 References

1. [Ng et al., 2002] Ng, A. Y., Jordan, M. I., & Weiss, Y. (2002). *On spectral clustering: Analysis and an algorithm.* Advances in Neural Information Processing Systems, 14.
2. [Zelnik-Manor & Perona, 2004] Zelnik-Manor, L., & Perona, P. (2004). *Self-tuning spectral clustering.* Advances in Neural Information Processing Systems, 17.
3. [Gropp et al., 2014] Gropp, W., Lusk, E., & Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface (3rd Edition).* MIT Press.
4. [Dagum & Menon, 1998] Dagum, L., & Menon, R. (1998). *OpenMP: an industry standard API for shared-memory programming.* IEEE Computational Science and Engineering, 5(1), 46-55.
5. [Han et al., 2018] Han, X., Wang, R., Zhou, Y., et al. (2018). *Mapping the mouse cell atlas by microwell-seq.* Cell, 172(5), 1091-1107.
6. [Eigen v3] Guennebaud, G., Jacob, B., et al. (2010). *Eigen v3.*http://eigen.tuxfamily.org.
7. [Pedregosa et al., 2011] Pedregosa, F., et al. (2011). *Scikit-learn: Machine Learning in Python.* Journal of Machine Learning Research, 12, 2825-2830.
8. [Van der Maaten & Hinton, 2008] Van der Maaten, L., & Hinton, G. (2008). *Visualizing data using t-SNE.* Journal of Machine Learning Research, 9(11).

Acknowledgements

This project was developed for the High Performance Computing for Data Science course, instructed by Pro-

fessor Sandro Fiore at the University of Trento.