










TECHNICAL NOTE

GigaSOM.jl: High-performance clustering and visualization of huge cytometry datasets

Miroslav Kratochvíl ^{1,2,*†}, Oliver Hunewald ^{3,*†}, Laurent Heirendt ⁴, Vasco Verissimo ⁴, Jiří Vondrášek ¹, Venkata P. Satagopam ^{4,5}, Reinhard Schneider ^{4,5}, Christophe Trefois ^{4,5} and Markus Ollert ^{3,6}

¹Institute of Organic Chemistry and Biochemistry, Flemingovo náměstí 542/2, 160 00 Prague, Czech Republic;

²Charles University, Department of Software Engineering, Malostranské náměstí 25, 118 00 Prague, Czech Republic;

³Luxembourg Institute of Health, Department of Infection and Immunity, 29 rue Henri Koch, L-4354 Esch-sur-Alzette, Luxembourg;

⁴University of Luxembourg, Luxembourg Centre for Systems Biomedicine, 6 avenue du Swing, Campus Belval, L-4367 Belvaux, Luxembourg;

⁵ELIXIR Luxembourg, University of Luxembourg, 6, avenue du Swing, Campus Belval, L-4367 Belvaux, Luxembourg and

⁶Odense Research Center for Anaphylaxis, Department of Dermatology and Allergy Center, Odense University Hospital, University of Southern Denmark, Klørvænget 15, DK-5000 Odense C, Denmark

*Correspondence address. Markus Ollert, 29, rue Henri Koch L-4354, Esch-sur-Alzette, Luxembourg. Tel.: +352 26970-829. E-mail:

markus.ollert@lih.lu  <http://orcid.org/0000-0001-7356-4075>

†Contributed equally.

Abstract

Background: The amount of data generated in large clinical and phenotyping studies that use single-cell cytometry is constantly growing. Recent technological advances allow the easy generation of data with hundreds of millions of single-cell data points with >40 parameters, originating from thousands of individual samples. The analysis of that amount of high-dimensional data becomes demanding in both hardware and software of high-performance computational resources. Current software tools often do not scale to the datasets of such size; users are thus forced to downsample the data to bearable sizes, in turn losing accuracy and ability to detect many underlying complex phenomena.

Results: We present GigaSOM.jl, a fast and scalable implementation of clustering and dimensionality reduction for flow and mass cytometry data. The implementation of GigaSOM.jl in the high-level and high-performance programming language Julia makes it accessible to the scientific community and allows for efficient handling and processing of datasets with billions of data points using distributed computing infrastructures. We describe the design of GigaSOM.jl, measure its performance and horizontal scaling capability, and showcase the functionality on a large dataset from a recent study.

Conclusions: GigaSOM.jl facilitates the use of commonly available high-performance computing resources to process the largest available datasets within minutes, while producing results of the same quality as the current state-of-art software. Measurements indicate that the performance scales to much larger datasets. The example use on the data from a massive mouse phenotyping effort confirms the applicability of GigaSOM.jl to huge-scale studies.

Received: 3 August 2020; Revised: 28 September 2020; Accepted: 20 October 2020

© The Author(s) 2020. Published by Oxford University Press GigaScience. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.

Keywords: high-performance computing; single-cell cytometry; self-organizing maps; clustering; dimensionality reduction; Julia

Background

Advances in single-cell technologies, such as mass cytometry, single-cell RNA sequencing, and spectral flow cytometry [1–3], provide deep and comprehensive insights into the complex mechanism of cellular systems, such as immune cells in blood, tumor cells and their microenvironments, and various microbiomes, including single-celled marine life ecosystems. Mass cytometry and spectral cytometry have enabled cells to be stained with >40 different markers to discover cellular differences under multiple conditions. The samples collected in recent studies often contain millions of measured cells (events), resulting in large and high-dimensional datasets. Traditional analysis methods, based on manual observation and selection of the clusters in 2D scatter-plots, is becoming increasingly difficult to apply on data of such complexity: for high-dimensional data, this procedure is extremely laborious, and the results often carry researcher or analysis bias [4].

Various dimensionality reduction, clustering, classification, and data mining methods have been used to aid with the semi-automated or fully automated processing, including neural networks [5], various rule-based and tree-based classifiers in combination with clustering and visualization [6, 7], or locality-sensitive and density-based statistical approaches [8]. However, computational performance of the algorithms, necessary for scaling to larger datasets, is often neglected, and the available analysis software often relies on various simplifications (such as downsampling, which impairs the quality and precision of the result) required to process large datasets in reasonable time without disproportionate hardware requirements.

To improve the performance, van Gassen et al. [9] introduced FlowSOM clustering, based on an algorithm that combines the self-organizing maps (SOMs) by Kohonen [10] and metaclustering [11], which allows efficient and accurate clustering of millions of cells [12]. FlowSOM is currently available as an R package that has become an essential part of many workflows, analysis pipelines, and software suites, including FlowJo and Cytobank® [13]. Despite the advance, the amount of data generated in large research-oriented and clinical studies frequently grows to hundreds of millions of cells, processing of which requires not only the efficiency of the algorithm but also a practical scalable implementation.

Here, we present GigaSOM.jl, an implementation of SOM-based clustering and dimensionality reduction functionality using the Julia programming language [14]. Compared with FlowSOM, GigaSOM.jl provides 2 major improvements: first, it uses computational and memory resources efficiently, enabling it to process datasets of size >10⁸ cells on commonly available hardware. Second, the implementation provides horizontal scaling support and can thus utilize large high-performance computing clusters (HPC) to gain improvements in speed and tangible dataset size, allowing datasets with >10¹⁰ cells to be processed in the distributed environment. Additionally, the implementation in Julia is sufficiently high-level to allow easy extensibility and cooperation with other tools in the Julia ecosystem. Several technical limitations imposed by the R-wrapped implementation in the C programming language of FlowSOM are also overcome.

Methods

The Kohonen SOM algorithm [10] is a kind of simplified neural network with a single layer equipped with a topology. The task of the SOM training is to assign values to the neurons so that the training dataset is covered by neighborhoods of the neurons, and, at the same time, that the topology of the neurons is preserved in the trained network. A 2D grid is one of the most commonly used topologies because it simplifies interpretation of the results as neuron values positioned in the 2D space, and related visualization purposes (e.g., EmbedSOM [15]). At the same time, the trained network can serve as a simple clustering of the input dataset, classifying each data point to its nearest neuron.

Batch SOM training

The original SOM training algorithm was introduced by Kohonen [16]. The map is organized as a collection of randomly initialized vectors called “codebook,” with weights $W(1)$. The training proceeds in iterations (indexed by time t), where in each iteration a randomly selected data point in the dataset is used to produce an updated codebook as

$$W_i(t+1) = W_i(t) + \alpha(t)h(t) \odot [x - W_i(t)],$$

where α is the learning rate parameter, i is the neuron nearest to the randomly selected data point x , and h is the vector of topological distances of the codebook vectors to the best matching unit. The learning has been shown to converge after a predictable number of iterations if α and topological neighborhood size in h are gradually decreased [10].

A more scalable variant of the algorithm can be obtained by running the single updates in batches where the values of x are taken from the whole dataset at once, which can be expressed in matrix form

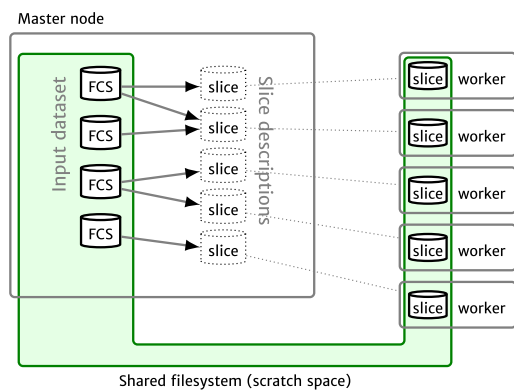
$$W(t+1) = \hat{H}(t) \cdot \mathcal{N}(X, W(t)) \cdot X,$$

where $\mathcal{N}(X, W(t))$ is a binary matrix that contains 1 at position i, j if and only if $W_i(t)$ is the closest codebook vector to X_j , and $\hat{H}(t)$ is a distance matrix of the codebook in 2D map topology with rows scaled to sum 1. Notably, the algorithm converges in the same cases as the online version [17] and may be viewed as a generalized version of k -means clustering, which is obtained by setting $H(t) = I$.

Implementations of the batch training may rely on several assumptions that are not available with the online training:

- computation of \mathcal{N} can use a pre-built spatial indexing structure on $W(t)$, which is constant for the whole batch;
- all computations involving X can be sliced and parallelized (moreover, because the accesses to X are not randomized, the implementation is more cache-efficient and more suitable for SIMD- and GPU-based acceleration);
- multiplication by $\hat{H}(t)$ can be associatively postponed to work only with the small codebook matrix, saving >50% required computation volume when compared with online training with large neighborhoods.

Data distribution



Computation

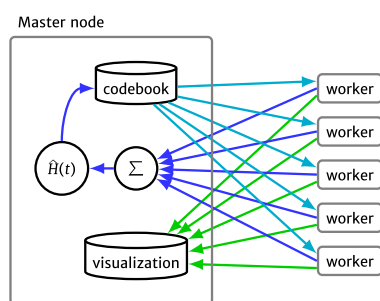


Figure 1 Architecture of GigaSOM.jl. *Top*: Data distribution process divides the available FCS files into balanced slices; individual workers retrieve their respective slice data using a shared storage. *Bottom*: The SOM learning and visualization processes require only a minimal amount of data to be transferred between the master and worker nodes, consisting of the relatively small codebook in the case of SOM learning (blue arrows) and pre-rasterized graphics in the case of visualization (green arrows).

Distributed implementation of GigaSOM.jl

The GigaSOM.jl package is a flexible, horizontally scalable, HPC-aware version of the batch SOM training written in the Julia programming language. The language choice has allowed a reasonably high-level description of the problem suitable for easy customization, while still supporting the efficient low-level operations necessary for fast data processing. GigaSOM.jl contains a library of functions for loading the data from Flow Cytometry Standard (FCS) files, distributing the data across a network to remote computation nodes present in the cluster, running the parallelized computation, and exporting and visualizing the results. The overall design of the main implemented operations is outlined in Fig. 1. Example Julia code that executes the distributed operations is provided in Supplementary Listing S1.

Data distribution procedure

The distributed computation process in GigaSOM is structured such that each computation node (“worker”) keeps its own, persistent slice of the whole dataset, and the partial results from the nodes are aggregated by the master node. To establish this structure, GigaSOM implements a separate procedure that aggregates the input FCS files and creates a balanced set of slices equally distributed among the workers.

The distribution procedure is implemented as illustrated in Fig. 1 (top): First, the master node reads the headers and sizes

of individual FCS files, verifying their structure and determining the total number of stored data points. This is used to create minimal descriptions of dataset slices of equal size (each description consists only of 4 numbers of the first and last file and the first and last data point index), which are transferred to individual workers. Each worker interprets its assigned slice description and extracts the part of the data from the relevant FCS files saved on a shared storage. The resulting slices may be easily exported to the storage and quickly imported again by individual workers, thus saving time if multiple analyses run on the same data (e.g., in case of several clustering and embedding runs with different parameters).

Importantly, a shared filesystem is usually one of the most efficient ways to perform data transfers in HPC environments, which makes the dataset loading process relatively fast. If a shared filesystem is not available, GigaSOM.jl also includes optional support for direct data distribution using the Distributed.jl package.

Batch SOM implementation

After the nodes are equipped with the data slices, the batch SOM training proceeds as illustrated in Fig. 1 (bottom):

1. The master node initializes the SOM codebook (usually by random sampling from available data).
2. The codebook is broadcast to all worker nodes. Because the size of the usual codebook is at most several tens of kilobytes, data transfer speed does not represent a performance bottleneck in this case.
3. The workers calculate a partial codebook update on their data and send the results back to the master node.
4. Finally, the master node gathers the individual updates, multiplies the collected result by $\hat{H}(t)$, and continues with another iteration from step 2 if necessary.

The time required to perform 1 iteration of the SOM training is mainly derived from the speed of the codebook transfer between nodes, and the amount of computation done by individual nodes. The current GigaSOM.jl implementation transfers all codebook versions directly between the master node and the workers, giving time complexity $\mathcal{O}(b) + \mathcal{O}(n/c)$ for b computation nodes equipped with c CPUs working on a dataset of size n . This complexity can be improved to $\mathcal{O}(\log_b b) + \mathcal{O}(n/c)$ by using efficient algorithms for parallel data broadcast and reduction, but we have not found a realistic dataset of size sufficient to gain any benefit from such optimization.

Implementation methodology

The GigaSOM.jl implementation of the batch SOM algorithm follows a similar structure as reported by other authors [18–20]. All distributed computations are expressed as a series of MapReduce-style operations [21], which are implemented as high-order functions. This has allowed us to clearly separate the low-level code required to support the parallel processing from the actual implementation of algorithms, and thus improve the code maintainability and vastly simplify further custom, user-specifiable data manipulation in the distributed environment. This abstraction additionally enables future transition to more complex data-handling routines or different parallelization systems. GigaSOM.jl can be transparently modified to support distributed parallel broadcast and reduction that might be required for handling huge SOMs on a very large number of workers (Collange et al. [22] provide a comprehensive discussion on that topic), or even run on a different distributed framework, such as the industry-standard MPI [23].

Our choice of the Julia programming environment was mainly motivated by making this abstraction as efficient as possible—the relatively high-level Julia code is compiled into efficient low-level bytecode, which enables high algorithm execution performance without modifying the code to work with any specialized performance-supporting primitives. This benefit is rarely available in popular high-level programming environments: e.g., many approaches for distributed computation exist in R (R Project for Statistical Computing, [RRID:SCR.001905](#)) [24], such as GridR [25], DistributedR, ddR, and sparklyr (for Apache Spark) (Apache Spark, [RRID:SCR.016557](#)) [26], but most of the projects unfortunately did not achieve general adoption or have been abandoned. Python libraries provide good support for optimized execution of specialized operations; parallel and distributed computing is supported, e.g., by the Dask project [27], with similar mode of use as the distributed processing tools in Julia. Despite that, producing efficient Python code requires careful consideration and utilization of the low-level array processing primitives (such as NumPy) (NumPy, [RRID:SCR.00863](#)) [28], often by representing the algorithms using only the available optimized matrix operations, which are elusive for non-mathematicians.

Spatial indexing

Because the most computationally expensive step of the SOM training is the search for nearest codebook vectors for each dataset item (i.e., construction of the matrix \mathcal{N}), we have evaluated the use of spatial indexing structures for accelerating this operation. GigaSOM.jl implementation can use the structures available in the package NearestNeighbors.jl, which include kd-trees and ball trees (also called vantage-point trees). [29, 30]

Although the efficiency of spatial indexing is vastly reduced with increasing dataset dimensionality, the measurements in section Results show that it can provide significant speedup with very large SOMs, even on data with >20 dimensions.

Visualization support

To simplify visualization of the results, GigaSOM.jl includes a parallel reimplement of the EmbedSOM algorithm in Julia [15], which quickly provides interpretable visualizations of the cell distribution within the datasets. EmbedSOM computes an embedding of the cells to 2D space, similarly as the popular t-SNE or UMAP algorithms [31, 32]. Unlike the usual dimensionality reduction algorithms, it uses the constructed SOM as a guiding manifold for positioning the individual points into the low-dimensional space, and achieves linear time complexity in the size of the dataset. The parallel implementation of EmbedSOM is built upon the same distributed data framework as the batch SOMs—because EmbedSOM is designed to be trivially parallelizable, it can be run directly on the individual data slices and gain the same speedup from parallel processing.

To aid the plotting of the EmbedSOM output, we have additionally implemented a custom scatterplot rasterizer in package GigaScatter.jl, which includes functions for quick plotting of large amounts of low- α points. To enable plotting of exceedingly large datasets, the rasterization can be executed in a distributed manner within the MapReduce framework, as shown in Supplementary Listing S1.

Results

The main result achieved by GigaSOM is the ability to quickly cluster and visualize datasets of previously unreachable size. In particular, we show that construction of a SOM from 10^9 cells

with 40 parameters can be performed in minutes, even on relatively small compute clusters with less than hundreds of CPU cores. The SOM can be used to quickly dissect and analyze the samples, as with FlowSOM [9]. This performance achievement vastly simplifies the interactive work with large datasets because the scientists can, for instance, try more combinations of hyperparameters and quickly get the feedback to improve the analysis and clustering of the data.

In this section, we first compare the output of GigaSOM.jl to that of FlowSOM, showing that the change in the SOM training algorithm has minimal impact on the quality of results. Furthermore, we provide benchmark results that confirm that GigaSOM.jl scales horizontally, and details of the speedup achievable by employing spatial indexing data structures for acceleration of the nearest-neighbor queries. Finally, we demonstrate the results that can be achieved by processing a gigascale dataset from a recent study by the International Mouse Phenotyping Consortium (IMPC) [33].

The presented performance benchmarks were executed on a Slurm-managed HPC cluster equipped with Intel®Xeon®E5-2650 CPUs, each node with 2 physical CPUs (total 24 cores) and 128 GB of RAM. All benchmarks were executed several times; the times were measured as “real” (wall-clock) time using the standard Julia timer facility. Measurements of the first runs were discarded to prevent the influence of caching and Julia just-in-time compilation; remaining results were reduced to medians.

Validation of clustering quality

To compare the GigaSOM.jl output with that from FlowSOM (FlowSOM, [RRID:SCR.016899](#)) [9], we used a methodology similar to the one used by Weber and Robinson [12]. The datasets were first processed by the clustering algorithms to generate clusters, which were then assigned to ground truth populations so that the coverage of individual populations by clusters was reasonably high. The mean F1 score was then computed between the aggregated clusters and ground truth. Unlike Weber and Robinson [12], who use a complex method of cluster assignment optimization to find the assignment that produces the best possible mean F1 score, we used a simpler (and arguably more realistic) greedy algorithm that assigns each generated cluster to a population with the greatest part covered by that cluster.

The benchmark did not consider FlowSOM metaclustering [9] because the comparison primarily aimed to detect the differences caused by the modifications in SOM training.

For the comparison, we reused the datasets Levine13 and Levine32 from the clustering benchmark [12]. In a typical outcome, most populations were matched by GigaSOM.jl just as well as by FlowSOM, as displayed in Fig. 2 (detailed view is available in Supplementary Fig. S1). Both methods consistently achieved mean F1 scores in the range of 0.65–0.70 on the Levine13 dataset and 0.81–0.84 on the Levine32 dataset for a wide range of reasonable parameter settings. In the tests, neither algorithm showed a significantly better resulting mean F1 score.

Scalable performance on large computer clusters

The benchmark of implementation scalability was performed as follows: a randomly generated dataset was distributed among the available computation nodes (workers) so that all CPUs are assigned an equal amount of data. For the benchmark, node counts as powers of 2 up to 256 have been chosen while the numbers of dataset parameters were chosen from multiples of

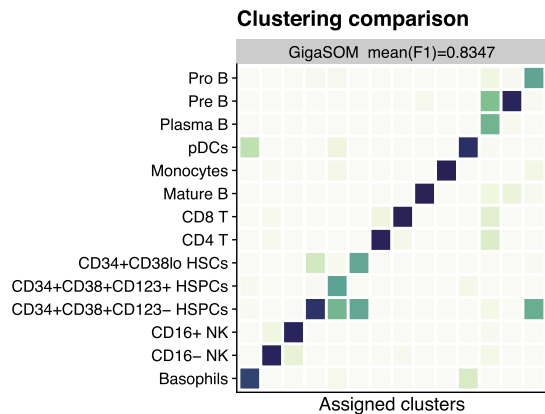


Figure 2 Comparison of GigaSOM.jl results with manual gating of the Levine32 dataset. The confusion matrix is normalized in rows, showing the ratio of cells in each aggregate of GigaSOM-originating clusters that matches the cell types from manual analysis. Darker color represents better match. The mean F1 score is comparable to FlowSOM. A more comprehensive comparison is available in Supplementary Fig. S1.

10 up to 50. The size of the dataset slice for a single node varied between 100,000, 200,000, and 300,000 cells to verify the impact of data density in cluster. The dataset was then processed by the SOM training algorithm for SOM sizes 10×10 , 20×20 , and 40×40 . The resulting SOMs were used for classifying the dataset into clusters (each input data point was assigned to a cluster defined by the nearest neighbor). An embedded view of the data was produced with the Julia implementation of EmbedSOM. All algorithms were also tested in variants where the naive search for nearest neighbors (or k -neighborhoods in the case of EmbedSOM) was replaced by utilization of a spatial-indexing data structure, in particular by the kd-trees and ball-trees.

The scalability results are summarized in Fig. 3: all 3 implemented algorithms scale almost linearly with the dataset size, the size of the SOM, and the dimension of the dataset. They reach an almost linear speedup with added compute capacity. In the case of SOM training, the required communication among the nodes caused only a negligible overhead; the majority of the computation pauses was caused by the random variance in execution time of computation steps on the nodes. The parallelized classification and embedding algorithms were not impaired by any communication overhead. Detailed benchmark results that show precise energy requirements of the training per processed data point, useful for deployment in large environments, are available in Supplementary Fig. S2.

Influence of the spatial indexing on the speed of various operations was collected as relative speedups (or slowdowns) when compared to a naive search. The results are displayed in Fig. 4. We have observed that both kd-trees and ball-trees were able to accelerate some operations by a factor $>2\times$, but the use of spatial indexing was hindered by many trade-offs that often caused decreased performance.

Most importantly, the cost of building the index often surpassed the total cost of neighborhood lookups by the naive method, which is most easily observable on the measurements of ball-tree performance with smaller SOM sizes. Both trees struggled to provide sufficient speedup in the presence of higher-dimensionality overhead (>30) and had only negligible

impact on the execution time of EmbedSOM computation, which was dominated by other operations.

On the other hand, it was easily possible to gain speedups $\sim 1.5\times$ for SOM training in most tests with lower dimension and large SOM, reaching $2.7\times$ for a 20-dimensional dataset (typical for current flow cytometry) processed with large 40×40 SOM. From the results, it seems appropriate to use the spatial indexing when the cost of other operations outweighs the cost of building the index, and the dimensionality overhead does not impede the efficiency of indexed lookup—in particular when training large SOMs of dimensionality $<\sim 30$, and when data occupancy per node is sufficiently high. Detailed measurements for all SOM sizes and dataset dimensions are available in Supplementary Fig. S3.

HPC analysis of previously unreachable dataset sizes

To showcase the GigaSOM.jl functionality on a realistic dataset, we have used a large dataset from the IMPC phenotyping effort [33] that contains measurements of mouse spleens by a standardized T-cell targeting panel, with individual cohorts containing genetically modified animals (typically a single-gene knockout) and controls; total 2,905 samples contain 1,167,129,317 individual cells. (The dataset is available from FlowRepository under the accession ID FR-FCM-ZYX9.)

The dataset was intentionally prepared by a very simple process—cell expressions were compensated, fluorescent marker expressions were transformed by the common asinh transformation with co-factor 500, and all dataset columns were scaled to $\mu = 0$ and $\sigma = 1$. The resulting data were used to train a 32×32 SOM, which was in turn used to produce the embedding of the dataset (with EmbedSOM parameter $k = 16$), which was rasterized. The final result can be observed in Fig. 5. The detailed workflow is shown in Supplementary Listing S1.

Notably, on a relatively small 256-core computer cluster (total 11 server nodes within a larger cluster managed by Slurm), the whole operation, consisting of Julia initialization, data loading (82.6 GB of FCS files), SOM training for 30 epochs, embedding, and export of embedded data (17.4 GB), took slightly <25 minutes and consumed at most 3 GB of RAM per core. From that, each epoch of the parallelized SOM training took ~ 25 seconds, and the computation of EmbedSOM visualization took 3 minutes. Distributed plotting of the result was performed using the GigaScatter.jl package; the parallel rasterization and combination of partial rasters took slightly >4 minutes.

Conclusions

In this article, we presented the functionality of GigaSOM.jl, a new, highly scalable toolkit for analyzing cytometry data with algorithms derived from SOMs. The results conclusively show that GigaSOM.jl will support the growing demand for processing of huge datasets, and bolster the utilization of the HPC hardware resources that are becoming widely available for laboratories and universities.

The ability to process a gigascale dataset to a comprehensible embedding and precise, easily scrutinizable statistics in mere minutes may play a crucial role in both design and analysis methods of future cytometry experiments. We believe that the accessible and flexible nature of the GigaSOM.jl implementation in the Julia programming language will also drive a transformation of other tools in the ecosystem towards the support of big data processing paradigms.

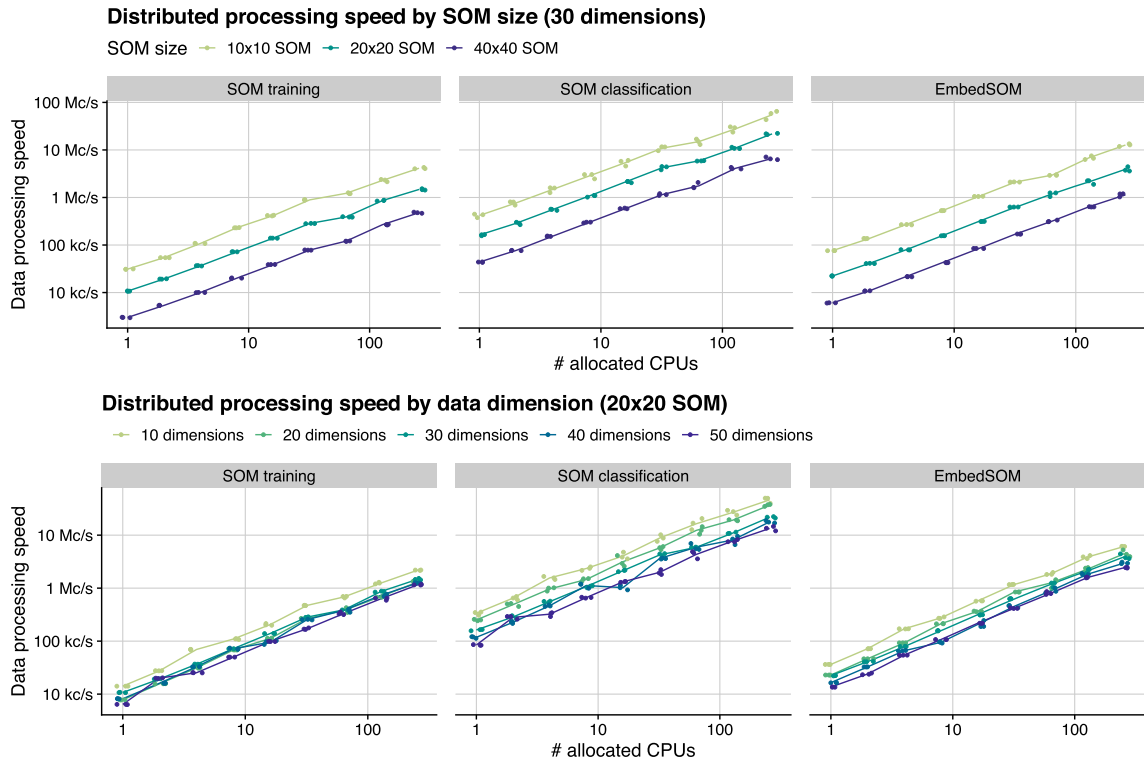


Figure 3 Performance dependency of distributed algorithms in GigaSOM on data dimensionality, SOM size, and number of available workers. Data processing performance is displayed as normalized to median speed in cells per second (c/s).

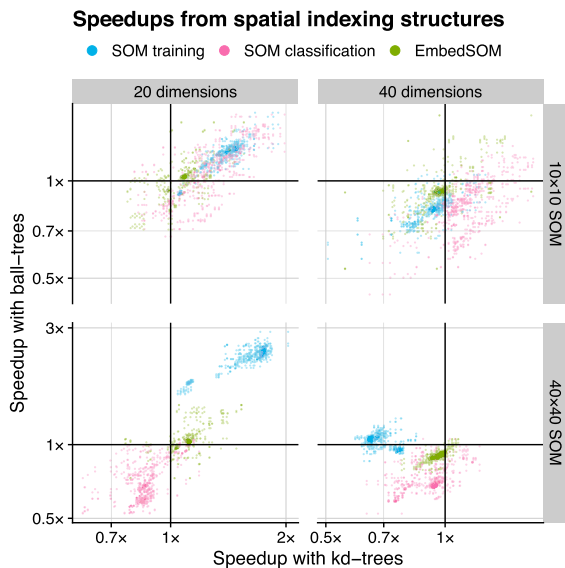


Figure 4 Effect of data-indexing structures on GigaSOM performance. The plotted points show relative speedup of the algorithms utilizing kd-trees (horizontal axis) and ball-trees (vertical axis) compared with brute-force neighbor search. Baseline (1x speedup) is highlighted by thick grid lines—a point plotted in the upper right quadrant represents a benchmark measurement that showed speedup for both kd-trees and ball-trees, upper left quadrant contains benchmark results where ball-trees provided speedup and kd-trees slowed the computation down, etc.

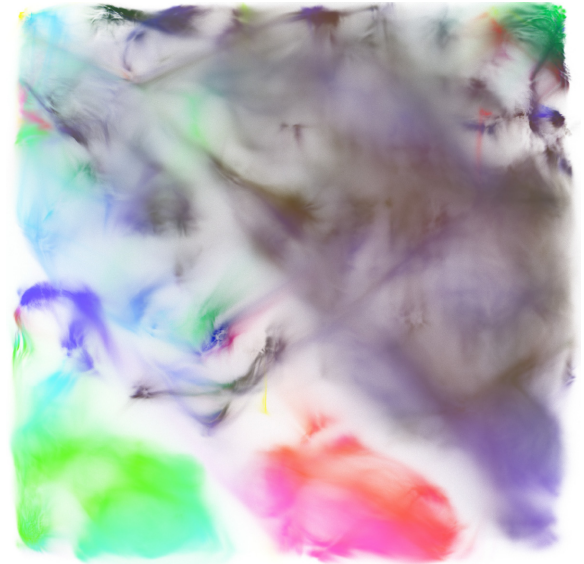


Figure 5 Raw IMPC Spleen T-cell dataset, processed by GigaSOM.jl and embedded by the Julia implementation of EmbedSOM. The figure shows an aggregate of 1,167,129,317 individual cells. Expression of 3 main markers is displayed in combination as mixed colors: CD8 in red, CD4 in green, and CD161 in blue. A more detailed, annotated version of the visualization is available in Supplementary Fig. S4.

The resulting software is publicly available as a Julia package. The interoperability with the Julia ecosystem allows GigaSOM.jl to benefit from many other available scientific computing packages, which simplifies its deployment not only in cytometry but also in other areas of research that use SOMs to extract information from large datasets.

Availability of Source Code and Requirements

All software is available under <https://doi.org/10.17881/lcsb.z5vy-fa75>.

- Package name: GigaSOM.jl
- Package home page: <https://git.io/GigaSOM.jl>
- Operating system(s): Portable to all Julia-supported platforms
- Programming language: Julia
- License: Apache License v2.0
- Julia package registry name: GigaSOM
- bio.tools ID: `biotools:GigaSOM.jl`
- RRID:SCR_019020

Availability of Supporting Data and Materials

All supporting data and materials are available in the GigaScience GigaDB database [34].

Abbreviations

CPU: central processing unit; FCS: Flow Cytometry Standard; GPU: graphics processing unit; HPC: high-performance computing; IMPC: International Mouse Phenotyping Consortium; MPI: Message Passing Interface; RAM: random access memory; SIMD: single instruction, multiple data; SOM: self-organizing map; t-SNE: t-distributed stochastic neighbor embedding; UMAP: Uniform Manifold Approximation and Projection.

Competing Interests

The authors declare that they have no competing interests.

Funding

M.K. and J.V. were supported by ELIXIR CZ LM2018131 (MEYS).

This work was supported by the Luxembourg National Research Fund (FNR) through the FNR AFR-RIKEN bilateral program (TregBar 2015/11228353) to M.O., and the FNR PRIDE Doctoral Training Unit program (PRIDE/11012546/NEXTIMMUNE) to V.V., R.S., and M.O.

Funding for open access publication was provided by the Institute of Organic Chemistry and Biochemistry of the CAS (RVO: 61388963).

The Responsible and Reproducible Research (R3) team of the Luxembourg Centre for Systems Biomedicine is acknowledged for supporting the project and promoting reproducible research.

The experiments presented in this article were carried out using the HPC facilities of the University of Luxembourg [35] (see <https://hpc.uni.lu>).

The project was supported by Staff Exchange programme of ELIXIR, the European life-sciences infrastructure.

Authors' Contributions

Conceptualization: O.H., L.H., C.T. Formal analysis, investigation, methodology: O.H., M.K., L.H. Software: O.H., M.K., L.H., V.V.

Funding acquisition, supervision: J.V., V.P.S., R.S., C.T., M.O. Validation: O.H., M.K. Visualization: M.K. Writing: O.H., M.K. All authors participated in reviewing, editing, and finalization of the manuscript.

References

1. Bandura DR, Baranov VI, Ornatsky OI, et al. Mass cytometry: technique for real time single cell multitarget immunoassay based on inductively coupled plasma time-of-flight mass spectrometry. *Anal Chem* 2009;**81**(16):6813–22.
2. Jaitin DA, Kenigsberg E, Keren-Shaul H, et al. Massively parallel single-cell RNA-Seq for marker-free decomposition of tissues into cell types. *Science* 2014;**343**(6172):776–79.
3. Schmutz S, Valente M, Cumano A, et al. Spectral cytometry has unique properties allowing multicolor analysis of cell suspensions isolated from solid tissues. *PLoS One* 2016;**11**(8):e0159961.
4. Mair F, Hartmann FJ, Mrdjen D, et al. The end of gating? An introduction to automated analysis of high dimensional cytometry data. *Eur J Immunol* 2016;**46**(1):34–43.
5. Arvaniti E, Claassen M. Sensitive detection of rare disease-associated cell subsets via representation learning. *Nat Commun* 2017;**8**(1):1–10.
6. Bruggner RV, Bodenmiller B, Dill DL, et al. Automated identification of stratifying signatures in cellular subpopulations. *Proc Natl Acad Sci U S A* 2014;**111**(26):E2770–7.
7. Qiu P, Simonds EF, Bendall SC, et al. Extracting a Cellular Hierarchy from High-dimensional Cytometry Data with SPADE. *Nat Biotechnol* 2011;**29**(10):886–91.
8. Lun ATL, Richard AC, Marioni JC. Testing for differential abundance in mass cytometry data. *Nat Methods* 2017;**14**(7):707–9.
9. van Gassen S, Callebaut B, Helden MJV, et al. FlowSOM: Using self-organizing maps for visualization and interpretation of cytometry data. *Cytometry Part A* 2015;**87**(7):636–45.
10. Kohonen T. Essentials of the self-organizing map. *Neural Netw* 2013;**37**:52–65.
11. Caruana R, Elhawary M, Nguyen N, et al. Meta Clustering. In: Sixth International Conference on Data Mining (ICDM'06); 2006:107–18.
12. Weber LM, Robinson MD. Comparison of clustering methods for high-dimensional single-cell flow and mass cytometry data. *Cytometry Part A* 2016;**89**(12):1084–96. <https://onlinelibrary.wiley.com/doi/abs/10.1002/cyto.a.23030>.
13. Chen TJ, Kotecha N. Cytobank: Providing an analytics platform for community cytometry data analysis and collaboration, Fienberg HG, Nolan P. In: High-Dimensional Single Cell Analysis. Berlin, Heidelberg: Springer; 2014:127–57.
14. Bezanson J, Edelman A, Karpinski S, Shah VB, Julia: A fresh approach to numerical computing, *SIAM review* 2017;**59**(1):65–98, SIAM.
15. Kratochvíl M, Koladiya A, Vondrášek J. Generalized Embed-SOM on quadtree-structured self-organizing maps. *F1000Res* 2019;**8**:2120.
16. Kohonen T. Self-organized formation of topologically correct feature maps. *Biological Cybernetics* 1982;**43**(1):59–69. <http://link.springer.com/10.1007/BF00337288>.
17. Cheng Y. Convergence and Ordering of Kohonen's Batch Map. *Neural Comput* 1997;**9**(8):1667–76.
18. Sul SJ, Tovchigrechko A. Parallelizing BLAST and SOM Algorithms with MapReduce-MPI Library. In: 2011 IEEE International Symposium on Parallel and Distributed Process-

- ing Workshops and Phd Forum Anchorage, AK, USA: IEEE; 2011:481–9. <http://ieeexplore.ieee.org/document/6008868/>.
19. Liu Y, Sun J, Yao Q, et al. A Scalable Heterogeneous Parallel SOM Based on MPI/CUDA. In: Asian Conference on Machine Learning; 2018. p. 264–279. <http://proceedings.mlr.press/v95/liu18b.html>.
 20. Sarazin T, Azzag H, Lebbah M. SOM Clustering Using Spark-MapReduce. In: 2014 IEEE International Parallel and Distributed Processing Symposium Workshops Phoenix, AZ, USA: IEEE; 2014. p. 1727–1734. <http://ieeexplore.ieee.org/document/6969583/>.
 21. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun ACM* 2008;**51**(1):107–13.
 22. Collange S, Defour D, Graillat S, et al. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Comput* 2015;**49**:83–97.
 23. Gropp W, Lusk E, Doss N, et al. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput* 1996;**22**(6):789–828.
 24. Ihaka R, Gentleman R. R: A language for data analysis and graphics. *J Comput Graph Stat* 1996;**5**(3): 299–314.
 25. Wegener D, Sengstag T, Sfakianakis S, et al. GridR: An R-based tool for scientific data analysis in grid environments. *Future Generation Comput Syst* 2009;**25**(4): 481–8.
 26. Zaharia M, Xin RS, Wendell P, et al. Apache Spark: a unified engine for big data processing. *Commun ACM* 2016;**59**(11):56–65.
 27. Rocklin M. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. Austin, Texas; 2015:126–32. https://conference.scipy.org/proceedings/scipy2015/matthew_rocklin.html.
 28. Harris CR, Millman KJ, van der Walt SJ, et al. Array programming with NumPy. *Nature* 2020;**585**(7825):357–62.
 29. Bentley JL. Multidimensional binary search trees used for associative searching. *Commun ACM* 1975;**18**(9): 509–17.
 30. Omohundro SM. Five Balltree Construction Algorithms. *Int Comput Sci Inst* 1989; 22.
 31. Maaten Lvd, Hinton G. Visualizing Data using t-SNE. *J Mach Learn Res* 2008;**9**(Nov):2579–605.
 32. McInnes L, Healy J, Saul N, Grossberger L, UMAP: Uniform Manifold Approximation and Projection, *Journal of Open Source Software* 2018;**3**(29):861.
 33. Brown SDM, Moore MW. The International Mouse Phenotyping Consortium: past and future perspectives on mouse phenotyping. *Mammalian Genome* 2012;**23**(9-10):632–40. <http://link.springer.com/10.1007/s00335-012-9427-x>.
 34. Kratochvíl M, Hunewald O, Heirendt L, et al. Supporting data for “GigaSOM.jl: High-performance clustering and visualization of huge cytometry datasets”. *GigaScience Database* 2020. <http://dx.doi.org/10.5524/100810>.
 35. Varrette S, Bouvry P, Cartiaux H, et al. Management of an academic HPC cluster: The UL experience. In: 2014 International Conference on High Performance Computing and Simulation (HPCS) Bologna, Italy: IEEE; 2014. p. 959–967. <http://ieeexplore.ieee.org/document/6903792/>.