



MASTER IN HIGH PERFORMANCE COMPUTING

# A Parallel Clustering Algorithm for Image Segmentation

*Supervisors:*

Dr. Stefano PANZERI, IIT

Dr. Luca HELTAI, SISSA

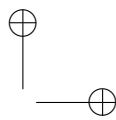
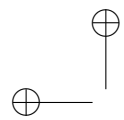
Dr. Alberto SARTORI, SISSA

*Candidate:*

Timoteo COLNAGHI

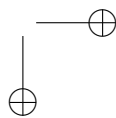
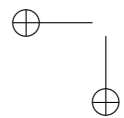
4<sup>th</sup> EDITION  
2017–2018





# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Implementation and benchmarks</b>	<b>3</b>
1.1 Algorithm implementation . . . . .	4
1.2 Benchmarks . . . . .	5
<b>Conclusions</b>	<b>13</b>
<b>Bibliography</b>	<b>13</b>







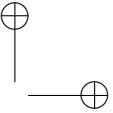
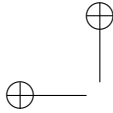
# Introduction

The brain encodes information into complex spatiotemporal signals evoked in its building cells. To unveil the details of this process, one has to devise robust methods to both register and alter brain cell signals without compromising either the cellular functions or the original brain network topology. Optogenetic techniques are by far the most promising toward this goal. They combine genetic intervention with two-photon stimulation and two-photon microscopy, mostly complemented by minimally to non-invasive procedures. In particular, target brain cell populations can be genetically modified so that they express a specific species of fluorescent molecules. These are classified into sensors and actuators, depending on their behavior. The intensity of light emission in sensors is ruled by the concentration of the chemical species they are sensitive to (*e. g.* genetically encoded calcium indicators with respect to  $\text{Ca}^{2+}$ ). Conversely, actuators act as neuromodulators if stimulated with controlled light, as they are able to modify the cellular permeability (*e. g.* Opsins) [1]. Recent studies have demonstrated that it is possible to activate actuators with a light beam which negligibly interferes with the imaging one [2, 3], thus allowing an efficient, combined employment of the two procedures. These new all-optical approaches enable the formulation of a novel class of groundbreaking experiments which would eventually test causal hypotheses about the neural code, as it is possible to test the consequences of writing or erasing pieces of information in brain cell activities. [4, 2, 3].

For the systematic storage and analysis of the many collected images recording the brain cell activity, experimenters have to face many non-trivial computational challenges which cannot be tackled by employing conventional methods. Neural imaging has eventually entered the big data era, as it has already happened for many other fields of study. Disentangling the relevant information contained in the collected data from noise and background can be unaffordable without employing high-performance computing algorithms.

As the final project for this Master's thesis, I have developed a parallel 3D

clustering algorithm to perform the spatiotemporal segmentation of the active brain cell regions in a series of black and white images recording a signal of interest. Its Python implementation—also boosted by a library written in C—has proven to be reasonably fast and to scale well according to both the strong and the weak scaling paradigms. I realized the present work while working as a postdoctoral researcher in the Center for Neuroscience and Cognitive Systems of the Italian Institute of Technology (IIT), Rovereto (Italy) lead by Dr. Stefano Panzeri. The raw data has been collected at the Optical Approaches to Brain Function Lab lead by Dr. Tommaso Fellin at the IIT headquarters, Genoa (Italy). In compliance to a non-disclosure agreement between these two labs and the Scientific Board of the SISSA-ICTP Master in High-Performance Computing, no details about either the raw data or their processing methods will be unveiled in this thesis, with the exception of a series of computational benchmarks of the developed clustering algorithm and a general overview about its implementation, collected in Chapter 1. This policy has been adopted in order to preserve the confidential status of the ongoing project this work is part of.



# Chapter 1

## Implementation and benchmarks

The modern optical microscopy allows the fast-pace imaging of brain cells' activity with single-cell resolution. A single experiment typically produces thousands of images, which translates into tens of gigabytes of raw data. If from one side their acquisition from the microscope, storage, access and visualization can be still accomplished resorting to customary software, several customary image processing routines cannot be employed anymore, due to their inefficient management of the available computational resources when dealing with big data.

As an example, let us estimate the memory resources required to segment the regions of interest (ROIs) in a series of images by resorting to a SAHN (sequential, agglomerative, hierarchic, non-overlapping) clustering method [5]. This is typically the first step towards localizing the individual cell's functional units and extracting a signal of interests from them. Let therefore  $\mathcal{E}$  be an experiment in which a series of  $N$  subsequent, black and white frames featuring  $n_x \times n_y$  pixels each have been collected. If both  $n_x, n_y$  are  $O(10^{2.5})$  and  $N$  is  $O(10^3)$  and we indicate the set of all pixels in the image series with  $P$ , it holds that  $|P| = O(10^8)$ . In order to store the full series of images in memory, approximately  $O(10^{9.2})$  bytes are required if the `uint16` encoding is used to encode the pixel luminosity  $l: P \rightarrow \{0, \dots, 2^{16} - 1\}$ .

To our knowledge, the most efficient Python library at date implementing the seven most widely used SAHN algorithms is D. Müller's `fastcluster` [6, 7]. Each of these algorithms clusters together groups of pixels using a *dissimilarity index* as a discriminant, that is a reflexive, symmetric mapping  $d: \tilde{P} \times \tilde{P} \rightarrow [0, +\infty)$  meeting the constraints  $d(p, p) = 0$  and  $d(p, q) = d(q, p) \forall p, q \in \tilde{P}$ , where  $\tilde{P}$  denotes the set of all pixels in the  $N$  images such that  $l(p) > 0$ . Defining the policy to adopt in order to compute  $d$  is a crucial step in implementing these algorithms, as the programmer has to face a time-memory trade-off. A memory-saving approach [7] would avoid storing in memory the full *dissimilarity matrix*  $D \doteq (d_{p,q})$ , which

is instead computed on demand for any chosen pair of pixels  $(p, q) \in \tilde{P}$ . To this purpose, an extra array of  $k \cdot |\tilde{P}| \text{ float64}$  elements is needed at runtime by the library to hold the coordinates of the elements in  $\tilde{P}$  with respect to some  $k$ -dimensional coordinate systems. Although this policy helps prevent from memory overflows, the several consecutive calls to the routine evaluating  $d$  surely decrease the overall time performance.

This drawback can be overcome if one opts for a time-saving approach [7]. In this case, all the  $O(|\tilde{P}|^2)$  elements in the upper triangle of the dissimilarity matrix  $D \doteq (d_{p,q})$  are evaluated before the actual clustering procedure begins and must be allocated in memory. Being quadratic in space, such a memory request typically exceeds the available memory resources when dealing with big data. If *e. g.*  $|\tilde{P}| \approx |P|/100$ , it turns out that  $O(10^{13})$  bytes are needed to fully store  $D$  in memory and this is a serious no-go for embracing this strategy *as is*.

Following up to this analysis, a savvy programmer facing a clustering problem on big data would opt immediately for the memory saving option if she had no experience in high-performance computing. Nevertheless, she would also agree that a memory-controlled version of the time-saving policy could be a more efficient solution, if feasible. In the next Section, we will discuss how this goal can be successfully accomplished.

## 1.1 Algorithm implementation

To pave the way towards an efficient version of the time-saving approach, we devised a parallel algorithm developed in a distributed memory framework, where the processes communicate via the Message Passing Interface (MPI) protocol. Due to the non-disclosure agreement we have mentioned in the Introduction, in the following we will only outline the general implementation of the clustering algorithm and present some relevant benchmarks run on a collection of mock data.

Our clustering algorithm has been developed in Python 3.5.2 [8], it is compliant with the advanced programming interface (API) design adopted in `scikit-learn` [9] and leverages the `mpi4py` package [10] for binding Python to an MPI library. Its implementation can be divided in two stages:

- A. In Stage A we concurrently exploit the `fastcluster` library to find a pool of clusters in some disjoint, non-overlapping subsets of  $\tilde{P}$ , in such a way that memory overflow is avoided. We will refer to these clusters as *partial clusters*, since we have only performed the clustering in some disjoint regions of  $P$ .
- B. Stage B is instead devoted to assessing whether there are partial clusters to be merged together and to get eventually the pool of *global clusters* resulting from the merging operation. These global clusters are the final output of our algorithm.



As the first step of Stage A, each process loads a slice of the input data  $P$ . In particular,  $\tilde{P}$  is split into a finite number of subsets  $\{\tilde{P}_i\}_{i=0}^{S-1}$ , where  $S$  denotes the number of spawned MPI processors and each subset  $P_i$  meets the following constraints:

$$\begin{cases} \tilde{P} = \cup_i P_i, \\ P_i \cap P_j = \emptyset \quad \forall i \neq j, \\ |P_i| \approx |P_j| \quad \forall i \neq j. \end{cases} \quad (1.1)$$

Within each processor, each of the subsets  $\{P_i\}$  is again split in a finite number of subsets  $\{Q_{i,j}\}_{j=0}^{M_i-1}$ , such that

$$\begin{cases} P_i = \cup_j Q_{i,j}, \\ Q_{i,j} \cap Q_{i,k} = \emptyset \quad \forall j \neq k, \\ |Q_{i,j}| \approx |Q_{i,k}| \quad \forall j \neq k \end{cases} \quad (1.2)$$

for all  $i \in \{0, \dots, S-1\}$ . The values  $M_i$  are determined at runtime by the program, taking into account the amount of available memory in the machine. Once a subset  $Q_{i,j}$  has been created, the  $i$ -th MPI processor calls a C-shared library we have developed to initialize the upper triangle of the dissimilarity matrix  $D$  for the pixels  $q \in Q_{i,j}$ .  $D$  is then fed to the `linkage` method of the `fastcluster` module, which yields the hierarchical clustering dendrogram for the events in  $Q_{i,j}$ .

The global clusters are then obtained in the second step by merging the partial clusters in the subsets  $Q_{i,j}$ , resorting to a dedicated Python routine.

## 1.2 Benchmarks

In this section we present the weak and strong scalability benchmarks for our algorithm on a collection of synthetic data, according to the following definitions:

**Definition 1.2.1.** *A parallel program scales in a strong sense if its speedup<sup>1</sup>  $S$  is equal to the number of working processors  $W$  for a fixed total problem size.*

**Definition 1.2.2.** *A parallel program scales in a weak sense if its speedup<sup>1</sup>  $S$  is equal to one for a fixed problem size per working processor.*

The input data for each benchmark is a 3-axis, `uint16 ndarray` of shape (256, 256, 750), which will be called  $P$  in the rest of this section, in accordance with the notation introduced in Section 1.1. All benchmarks have been run on a machine equipped with a couple of Intel<sup>®</sup> Xeon<sup>®</sup> Processors (Model E5-2643 v3 @ 3.40GHz),

<sup>1</sup>The speedup of a parallel application, as a function of the number of working processors  $W$  employed in parallel to run it, is defined as  $S(P) \doteq T_1/T(W)$ , where  $T_1$  is the overall execution time of the program executed by one single processor and  $T(W)$  is the overall execution time of the same program executed with  $W$  processors in parallel.

featuring 6 physical cores each and a total available Random Access Memory (RAM) of about 130 GB. Regarding the employed software, the Python distribution we used to run the benchmarks was CPython 3.5.2 [8], complemented with numpy 1.15.4 [11] linked to the openBLAS 3.0 shared library [12, 13]. The MPI standard implementation bound to mpi4py was instead MPICH 3.2 [14]. The processors in our machine are also provided with an AVX2 extension, which also boosts the execution of the developed C-library routines to compute the dissimilarity matrix, as well as any numpycall to its linked Basic Linear Algebra Subprograms (BLAS) library. Indeed, both the library we have developed and openBLAS 3.0 have been compiled with GNU gcc 5.4.0 and contain instructions which results to be vectorialized, since the flag `-O3` has been specified at compiling time in both cases.

### Weak scaling assessment

Twelve experiments  $\mathcal{W}_1, \dots, \mathcal{W}_{12}$  have been run to assess the weak scalability of our algorithm, where the subscripts specify the number of MPI processes spawned to run the benchmark. In each of them  $P$  has been initialized such that each MPI process would feature the same number of events in the loaded partition of  $P$ . In particular, we ensured that the process with rank  $k$  in experiment  $\mathcal{W}_i$  holds in its memory the following slice of  $P$ :

$$P_{i,k}[x] = \begin{cases} x+1 & \text{if } x = \lfloor X_k/2 \rfloor, \\ 0 & \text{otherwise} \end{cases} \quad (1.3)$$

where  $X_k$  is the size of the 0-th axis in the partition  $P_{i,k}$  and  $x$  denotes the index value along this axis.

In Figure 1.1 we plot the speedups  $S_i \doteq T_1/T_i$  ( $i \in \{1, \dots, 12\}$ ), where  $T_i$  indicates the elapsed time for the experiment  $\mathcal{W}_i$  to complete. The plot reveals that our algorithm scales well in a weak sense.

### Strong scaling assessment

To assess the strong scalability of our algorithm we have run one hundred eight experiments  $\mathcal{S}_{i,j}$ , where  $i \in \{1, \dots, 12\}$  indicates the number of spawned MPI processes and  $j \in \{2, \dots, 10\}$  labels how  $P$  has been initialized. In particular,  $P$  has been initialized as follows for experiments  $\mathcal{S}_{i,j}$ :

$$P[x] = \begin{cases} x+1 & \text{if } \text{mod}(x, j) = 0, \\ 0 & \text{otherwise} \end{cases} \quad (1.4)$$

where  $x \in \{0, \dots, 255\}$  denotes the global index value along the 0-th axis of  $P$ .

Figure 1.2 reports the speedups  $S_{i,j} \doteq T_{1,j}/T_{i,j}$  for the three pools of experiments  $(i, j) \in \{1, \dots, 12\} \times \{2\}$  (navy plot),  $(i, j) \in \{1, \dots, 12\} \times \{5\}$  (green plot) and  $(i, j) \in \{1, \dots, 12\} \times \{10\}$  (purple plot). Here  $T_{i,j}$  indicates the elapsed time for

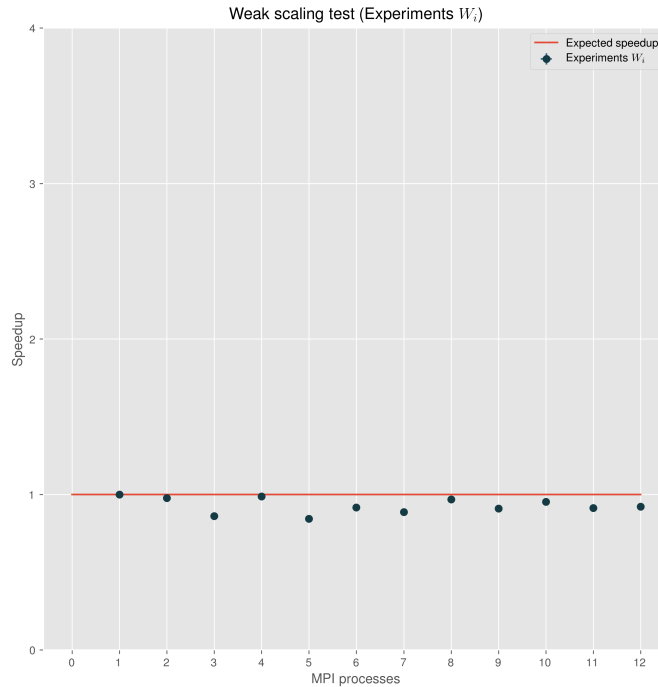


Figure 1.1: Weak scalability test for the developed clustering algorithm.

the experiment  $\mathcal{S}_{i,j}$  to complete. If we denote the overall workload of experiments  $\mathcal{S}_{i,10}$  with  $W_{10}$ , the overall workloads for experiments  $\mathcal{S}_{i,5}$  and  $\mathcal{S}_{i,2}$  are  $2 \cdot W_{10}$  and  $5 \cdot W_{10}$ , respectively.

In each of these plots, we notice that the speedup grows almost always if the number of MPI processes is increased. Moreover, the speedup enhancements for two problems of different sizes results to be greater for the one with the greater workload as the number of employed processes grows. This is an expected behavior for our algorithm—which is not embarrassingly parallel—, in qualitative accordance with Amdahl’s law [15, Section 2.6.2].

A more detailed analysis of these plots reveals some interesting behaviors that are worth to be discussed. Firstly, we notice that the speedup for a fixed workload deviates dramatically from the expected trend for a small group of experiments. This is the case *e. g.* of experiments  $\mathcal{S}_{11,2}$ ,  $\mathcal{S}_{7,9}$ ,  $\mathcal{S}_{11,9}$  and  $\mathcal{S}_{7,12}$ —where the speedup drops away—, and *e. g.* of experiments  $\mathcal{S}_{2,4}$ ,  $\mathcal{S}_{4,6}$ ,  $\mathcal{S}_{8,8}$  and  $\mathcal{S}_{2,10}$ —where the speedup rises and meets the theoretical expectation. In some other experiments—like *e. g.* in  $\mathcal{S}_{4,8}$ , and  $\mathcal{S}_{2,j}$  for  $j \in \{2, 3, 6, 8\}$ —the speedup is even slightly higher

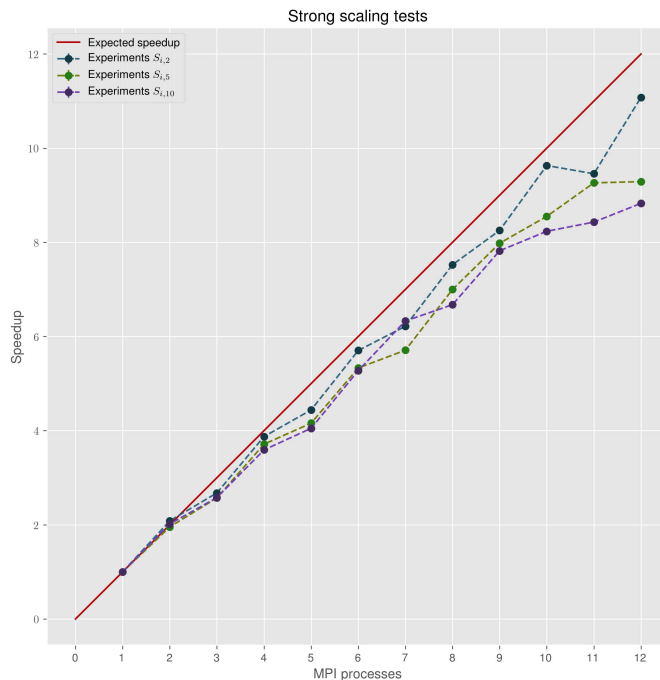


Figure 1.2: Strong scalability tests for the developed clustering algorithm.

than the theoretical expectation. The first effect witnesses a slight unbalance in both the computation and communication workload among the MPI processes if compared with the other experiments featuring the same initialization for  $P$ . In the other two cases, the overall workload is instead perfectly balanced among the processes. We would like to stress that this slight load unbalance is not happening as a consequence of a bad algorithm design. Instead, it always occurs whenever the overall computational workload cannot be evenly distributed among the processors, to preserve the completeness of some portions of data within each process. Indeed, let *e. g.* the input data  $X$  be a  $n$ -dimensional array and let  $n_0$  be the size of the first axis. If we need to distribute  $X$  into the local memories of  $K$  MPI processors by splitting  $X$  across its 0-th axis and  $\text{mod}(n_0, K) \neq 0$ , then  $\text{mod}(n_0, K)$  process(es) will store one more column with respect to the others<sup>2</sup>. In our case,  $P$  is a 3-dimensional array and we split it along its 0-th axis. As a result, some of the spawned MPI processes may store  $n_y \cdot N$  data points more than the others.

To conclude, we can state that our algorithm scales well in a strong sense when-

<sup>2</sup>See *e. g.* Ref. [16, chapter 8].

ever the workload can be perfectly balanced among the spawned MPI processors.

### Usage of computing resources

For all the experiments  $\mathcal{S}_{i,j}$  we have recorded the amount of computing resources used by our program by running the top command in background. As an example, Figures 1.3 and 1.4 (1.5 and 1.6) report the average CPU usage and the overall RAM usage for experiments  $\mathcal{S}_{12,10}$  ( $\mathcal{S}_{12,5}$ ), respectively.

The qualitative analysis of the memory exploitation plotted in Figures 1.4 and 1.6 also allows us to grasp a general insight on the implementation of our algorithm. As stated in Section 1.1, our algorithm starts by seeking for the partial clusters<sup>3</sup>, after that it assesses whether there are partial clusters to be merged together and it merges them eventually. To perform the partial clustering without memory overflows, each MPI processor had to call the `linkage` function in the `fastcluster` library seven times<sup>4</sup> in both of these benchmarks for each 2-dimensional array  $P[x] \subset P$  such that  $P[x] \neq 0$ . Moreover, the data size in input to the last of these calls is one order of magnitude less than the one in input to the previous ones<sup>4</sup>. In experiment  $\mathcal{S}_{12,10}$  ( $\mathcal{S}_{12,5}$ ), two (four) processors were storing in their memory a portion of  $P$  containing three (five) slices of  $P$  where  $P[x] \neq 0$ , whereas the remaining ten (eight) were storing only two (four) of them<sup>4</sup>. Therefore, the total number of calls to the `linkage` function were twenty-one (thirty-five) for the first group of processors and fourteen (twenty-four) for the second one. Let us now compare these numbers with the number of spikes in Figure 1.4 (1.6). The total number of either red or green dots in Figure 1.4 (1.5) equals thirty-five (twenty-four), which coincides with the number of calls to the `linkage` function made by the processors with a higher workload. The groups of red dots gather the first six consecutive, parallel calls to the `linkage` function to get the partial clusters in each of the slices of  $P$  where  $P[x] \neq 0$ . Instead, the green dots following the red ones witness the seventh call to `linkage`, whose input data size is one order of magnitude less than the one in input to the previous ones. In both of the figures we also notice that the height of the spikes marked in red gradually starts dropping from pools B. At the same time a smaller spike appears just after them, whose magnitude is equal to the loss of height of the preceding spike. This behavior is a consequence of the different speed with which each processor gets the partial clusters during the Stage A of the program. It is also worth noticing that the difference in height between the red spikes in the last pool of each figure and the spikes in the other ones reveals the load unbalance between the processors. Indeed, two (four) processors in experiment  $\mathcal{S}_{12,10}$  ( $\mathcal{S}_{12,5}$ ) have to retrieve the partial clusters in a larger event space, as stated above. The

<sup>3</sup>Partial clusters have been defined in Section 1.1 as the pool of clusters in some disjoint, non-overlapping subsets of  $\tilde{P}$ .

<sup>4</sup>The interested reader would excuse us if we do not prove this and some other results in the following. This would imply unveiling the details of the algorithm, thus breaking the non-disclosure agreement we have to honor.

memory usage during the Stage B of the program is instead negligible ( $\approx 1MB$ ) and cannot be highlighted by the plots in Figures 1.4 and 1.6.

As far as the mean CPU usage is concerned, we see from Figures 1.3 and 1.5 that our program makes an intensive usage of the CPU cores overall.

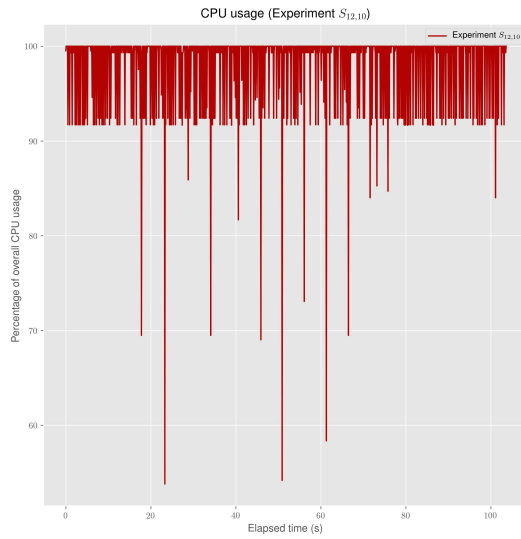


Figure 1.3: Average CPU usage during experiment  $S_{12,10}$ .

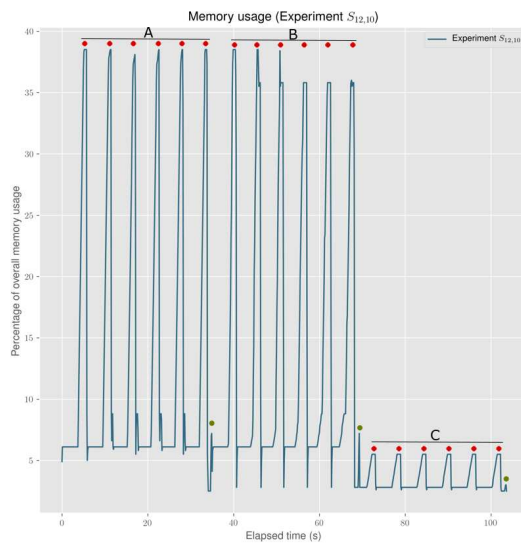


Figure 1.4: Overall RAM usage during experiment  $S_{12,10}$ .

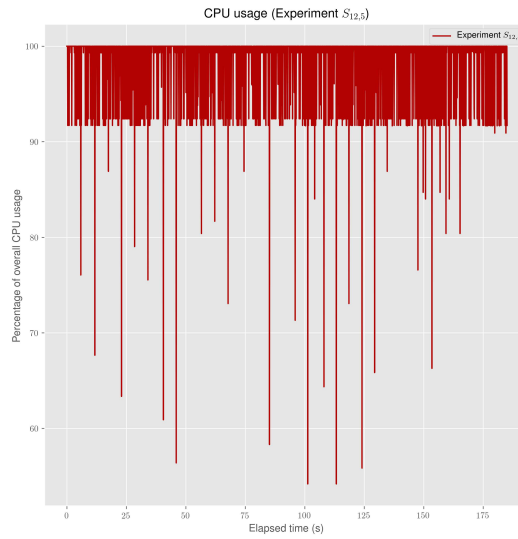


Figure 1.5: Average CPU usage during experiment  $S_{12,5}$ .

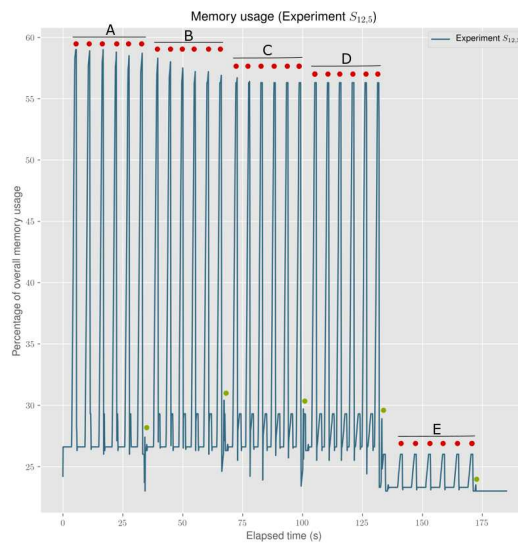


Figure 1.6: Overall RAM usage during experiment  $S_{12,5}$ .





# Conclusions

The development of optogenetics, combined with two-photon microscopy, has allowed to both monitor and alter the brain cell's activity with accurate precision. In particular, the activity of single brain cells can be recorded with a fast acquisition rate in a series of high-resolution images even during behavioral experiments *in vivo*. To perform efficiently the spatiotemporal segmentation of the active brain cells in the collected images, one needs to rely on high-performance software, as the size of the collected data can be huge ( $O(10)$  GB).

In this thesis I have presented a parallel, memory-distributed 3D clustering algorithm which I have implemented in Python 3.5.2 and C as the final project for the SISSA-ICTP Master in High-Performance Computing. This algorithm can be employed to segment the active brain cell regions in a series of black and white images recording a signal of interest.

In Chapter 1, we have described the implementation procedure for our algorithm and we have brought evidences of its good scaling properties. In particular, our implementation has proven to scale well according to both the weak and to the strong scaling definitions. Crucial to our achievement have been a savvy usage of the interfaces contained in the Python modules `mpi4py` and `fastcluster`, along with the development of a C-library to compute the dissimilarity matrix among the events.

The modular approach we adopted to writing our software makes it adaptable to be reused for the implementation of other clustering algorithms in a parallel, memory-distributed framework.





# Bibliography

- [1] A. Guru, R. J. Post, Y.-Y. Ho, and M. R. Warden, “Making sense of optogenetics”, *International Journal of Neuropsychopharmacology*, vol. 18, no. 11, pyv079, 2015.
- [2] S. Bovetti, C. Moretti, S. Zucca, M. Dal Maschio, P. Bonifazi, and T. Fellin, “Simultaneous high-speed imaging and optogenetic inhibition in the intact mouse brain”, *Scientific Reports*, vol. 7, no. 40041, pp. 1–17, 2017.
- [3] A. Forli, D. Vecchia, N. Binini, F. Succol, S. Bovetti, C. Moretti, F. Nespoli, M. Mahn, C. Baker, M. M. Bolton, O. Yizhar, and T. Fellin, “Two-photon bidirectional control and imaging of neuronal excitability with high spatial resolution *in vivo*”, *Cell Reports*, vol. 22, pp. 3087–3098, 2018.
- [4] S. Panzeri, C. D. Harvey, E. Piasini, P. E. Latham, and T. Fellin, “Cracking the neural code for sensory perception by combining statistics, intervention, and behavior”, *Neuron*, vol. 93, pp. 491–507, 2017.
- [5] P. H. A. Sneath and R. R. Sokal, *Numerical Taxonomy*. W. H. Freeman, 1973.
- [6] D. Müllner, “Modern hierarchical, agglomerative clustering algorithms”, *Computing Research Repository*, vol. abs/1109.2378, pp. 1–29, 2011. arXiv: 1109.2378. [Online]. Available: <http://arxiv.org/abs/1109.2378>.
- [7] D. Müllner, “**fastcluster**: Fast hierarchical, agglomerative clustering routines for R and Python”, *Journal of Statistical Software*, vol. 53, pp. 1–18, 2013.
- [8] G. van Rossum *et. al.* (2018). The python programming language, [Online]. Available: <http://www.python.org/>.

- [9] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Müller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, “API design for machine learning software: Experiences from the scikit-learn project”, *Computing Research Repository*, vol. abs/1309.0238, pp. 1–15, 2013. arXiv: 1309.0238. [Online]. Available: <http://arxiv.org/abs/1309.0238>.
- [10] L. Dalcin. (2018). Mpi for python (mpi4py), [Online]. Available: <https://mpi4py.readthedocs.io/en/stable/>.
- [11] T. Oliphant *et al.* (2018), [Online]. Available: <http://www.numpy.org/>.
- [12] K. Goto, Z. Xianyi, W. Quian, and W. Saar. (2018). Openblas - github repository, commit 78d877b5, [Online]. Available: <https://github.com/xianyi/OpenBLAS/commit/78d877b54bc2d5731627ae09b3a94d1ea5d18ff5>.
- [13] K. Goto, Z. Xianyi, W. Quian, and W. Saar. (2018). Openblas, [Online]. Available: <https://www.openblas.net/>.
- [14] The MPICH authors. (2018), [Online]. Available: <https://www.mpich.org/>.
- [15] P. S. Pacheco, *An introduction to parallel programming*. Morgan Kaufmann, 2011.
- [16] N. MacDonald, E. Minty, J. Malard, T. Harding, S. Brown, and M. Antonioletti, *Writing message passing parallel programs with mpi*. [Online]. Available: <https://www.archer.ac.uk/training/course-material/2018/07/mpi-epcc/notes/MPP-notes.pdf>.