Computational Improvements of Microaggregation Algorithms for the Anonymization of Large-Scale Datasets

A Degree Thesis
Submitted to the Faculty of the
Escola Tècnica d’Enginyeria de Telecomunicació de Barcelona
at the Universitat Politècnica de Catalunya
by
Alejandro García Álvarez

In partial fulfilment
of the requirements for the degree in
Telematics ENGINEERING

Advisors: David Rebollo-Monedero and Ahmad M. Mezher

Barcelona, January 2017
Abstract

The technical contents of this work fall within the field of statistical disclosure control (SDC), which concerns the postprocessing of the demographic portion of the statistical results of surveys containing sensitive personal information, in order to effectively safeguard the anonymity of the participating respondents. The concrete purpose of this study is to improve the efficiency of a widely used algorithm for \( k \)-anonymous microaggregation, known as maximum distance to average vector (MDAV), in order to vastly accelerate its execution without affecting its excellent functional performance with respect to competing methods. The improvements tested in this project are quite diverse, encompassing algebraic modifications, divide-and-conquer algorithmic improvements, and last but not least, parallel computation both on CPU and graphics processing units (GPUs).

We focused more on parallelization because it is the most scalable and powerful tool we could use, comparing CPU and massive GPU parallelization. Nonetheless, the algebraic and algorithmic improvement in the code are conceived bearing in mind the optimization of the parallelization of the algorithm. We also propose a computational change involving a drastically reduced execution time, occasionally at the price of a slightly higher distortion.

**Keywords:** \( k \)-Anonymity, microaggregation, statistical disclosure control, optimization, parallelization.
ACKNOWLEDGEMENTS

I wish to express my sincere thanks to the Information Security Group (ISG) of the Department of Telematics Engineering at the Technical University of Catalonia, especially Dr. David Rebollo-Monedero, for giving me the opportunity to work with him and trust in me during the last stage of my degree.

I place on record, my sincere thanks to Dr. Ahmad Mohamad Mezher, also part of the ISG group, for helping to accomplish my goal and give me his unconditional support.

Finally, thanks to my family and friends, who have always been there to give me the support I needed and for all the work done to bring me up and allow me to be where I am.
Revision history and approval record

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>06/01/2017</td>
<td>Document creation</td>
</tr>
<tr>
<td>1</td>
<td>08/01/2017</td>
<td>Document revision</td>
</tr>
</tbody>
</table>

Document distribution list

<table>
<thead>
<tr>
<th>Name</th>
<th>e-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alejandro García Álvarez</td>
<td><a href="mailto:alejandro.garcia.alvarez@alu-etsetb.upc.edu">alejandro.garcia.alvarez@alu-etsetb.upc.edu</a></td>
</tr>
<tr>
<td>David Rebollo-Monedero</td>
<td><a href="mailto:david.rebollo.monedero@upc.edu">david.rebollo.monedero@upc.edu</a></td>
</tr>
<tr>
<td>Ahmad Mohamad Mezher</td>
<td><a href="mailto:ahmad.mohamad@upc.edu">ahmad.mohamad@upc.edu</a></td>
</tr>
</tbody>
</table>

Written by: Date 06/01/2017
Reviewed and approved by: Date 08/01/2017

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Name</th>
<th>Date</th>
<th>Name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alejandro García Álvarez</td>
<td>08/01/2017</td>
<td>David Rebollo-Monedero</td>
<td>08/01/2017</td>
<td>Ahmad Mohamad Mezher</td>
<td>08/01/2017</td>
</tr>
<tr>
<td>Project Author</td>
<td></td>
<td>Project Supervisor 1</td>
<td></td>
<td>Project Supervisor 2</td>
<td></td>
</tr>
</tbody>
</table>

Position: Project Author
Abstract

Acknowledgements

Revision history and approval record

List of figures

1. Introduction
   1.1. Motivation and Contributions
   1.2. Work Packages and Gantt diagram
      1.2.1. Work Packages
      1.2.2. Gantt Diagram
      1.2.3. Deviation from the initial plan and incidences

2. Background and state of the art
   2.1. Fundamentals of Statistical disclosure control (SDC)
   2.2. State of the art on k-anonymous microaggregation
   2.3. Fundamentals of parallel computing

3. Project development
   3.1. Experimental setup
   3.2. Improvements developed on MATLAB
      3.2.1. Single distance calculation exploit
      3.2.2. Distance algebraic improvement
      3.2.3. Centroid subtraction improvement
   3.3. Improvements developed on C++
      3.3.1. Sort algorithm improvement
      3.3.2. Records quantized removal improvement
      3.3.3. Algebraic libraries for distance calculation
      3.3.4. Intrinsic parallelization
      3.3.5. Extrinsic parallelization
   3.4. Improvements developed on CUDA

4. Experimental results
   4.1. Results for MATLAB
   4.2. Results for C++
   4.3. Intrinsic parallelization results
   4.4. Extrinsic parallelization results
   4.5. GPU massive parallelization results
   4.6. Overall results

5. Budget

6. Conclusions and future development

References
LIST OF FIGURES

Fig. 1. Contributions made in this project as computational improvements to the MDAV algorithm. .............................................................. 7
Fig. 2. Example of $k$ -anonymous microaggregation of published data with $k = 3$. .......................................................... 11
Fig. 3. Example of microaggregation, with $k = 5$. Each tuple of 5 points is assigned a representative (centroid). ......................................................... 12
Fig. 4. Pseudo-code of the maximum distance to average vector (MDAV). .......................................................... 13
Fig. 5. A graphical representation of Amdahl’s law. .......................................................... 15
Fig. 6. MATLAB improvements. ........................................................................ 17
Fig. 7. C++ improvements. ........................................................................ 19
Fig. 8. First removal of records approach. .......................................................... 20
Fig. 9. Improved removal of records approach. .......................................................... 21
Fig. 10. Example of CUDA processing flow. .......................................................... 24
Fig. 11. MDAV Elapsed time for $k = 10$ and $M = 10$. .......................................................... 25
Fig. 12. Quick-sort and Quick-select. .......................................................... 25
Fig. 13. MDAV without and with BLAS. .......................................................... 26
Fig. 14. Time percentages MDAV. .......................................................... 27
Fig. 15. MDAV no parallelized and parallelized. .......................................................... 27
Fig. 16. MDAV with 4 external areas no parallelized and parallelized. .......................................................... 28
Fig. 17. MDAV with 4 external areas no parallelized and parallelized. .......................................................... 29
Fig. 18. MDAV Intrinsic and MDAV extrinsic. .......................................................... 29
Fig. 19. Speed-up and distortion relation. .......................................................... 30
Fig. 20. MDAV GPU parallelization and MDAV BLAS no parallelized. .......................................................... 30
Fig. 21. Improvements percentages. .......................................................... 31
1. Introduction

BIG data is better data. Nowadays, the amount of accurate information provided by any new technology is geometrically increasing, leading to new and unseen opportunities among the intricacies of complex systems. Peter Norvig, director of research at Google, in his paper “The unreasonable effectiveness of data” [4], cleverly acknowledges that huge amounts of information may drastically improve the effectiveness of a machine-learning algorithm to the point of turning a hopeless computer model into an expert system exceeding human performance. Indeed, the opportunity of analyzing a massive wealth of data with modern systems may offer extraordinary potential.

Nevertheless, it is always a daunting challenge to protect the privacy of the users involved, as the sensitive information included in this technologies poses privacy risks that cannot remain unnoticed. Personal information, including that inferred from observed behavior without the user’s notice, poses glaring privacy risks, especially when combined across several information services, and when enriched with metadata indicating size, location, time, frequency, and other contextual information.

Fortunately, advanced privacy-enhancing technologies may complement the more traditional approaches of encryption and granular access-control policies envisioned in the last years. When the intended recipients of sensitive information are themselves untrusted, access control alone is notoriously insufficient. Modern privacy methods can be applied with data-perturbative strategies in order to obtain a desired degree of privacy, at the expense of a controlled loss in data utility. This is precisely the case of statistical disclosure control (SDC), which concerns the post processing of the demographic portion of the statistical results of surveys containing sensitive personal information, in order to effectively safeguard the anonymity of the participating respondents. As the data collected is meant to be publicly released or at least distributed within an institution, traditional encryption alone cannot prevent attacks aimed to infer the identity of the individuals in the database released, on the basis of their demographic attributes.

Some of these privacy protection mechanisms come from mathematical formalisms originally conceived for information-theoretic and operational data compression, as well as convex optimization, in an attempt to systematically measure and get the optimal trade-off between data privacy and data utility loss [12, 11, 13].

1.1. Motivation and Contributions

This study comes from the necessity of privacy-enhancing algorithms to preserve the anonymity of information in big data. Nowadays, these algorithms are not yet computationally fast to be used on a great scale with large data-sets. Koopmans in 1975 in his book “Concepts of optimally and their uses” Nobel Memorial Lecture [9] defines optimization as “the best use of scarce resources”, which is the goal of this project.

On the other hand, the power of optimizing and parallel computing is a tool which can be applied to our project, and indeed, this will reduce the total execution time. In addition, it is worth mentioning that if users know that their data will be anonymized, the consequent increase in users willing to provide additional data may very well represent a far greater gain in utility.

This technical report offers a summarized account of the project devoted to the improvement and optimization of the $k$-anonymous microaggregation algorithm maximum distance to average vector (MDAV) explained in §2.2. This project is the continuation of a previous work done by the Information Security Group which stated the bases for the following improvements.
Fig. 1. Contributions made in this project as computational improvements to the MDAV algorithm.

This project consists of three parts, as seen in Fig. 1:

1. MATLAB
2. C++
3. CUDA C++

The MATLAB improvements are the following:

- **Single distance calculation optimization**: We have detected that we may reuse certain computations in a loop (with a high percentage of the total execution program time) as it was written in the pseudocode §3.2.1.
- **Centroid subtraction improvement**: Calculation of the centroid by subtracting the used records instead of recalculating the whole centroid again §3.2.3.
- **Distance algebraic improvement**: Algebraic improvement in distance calculation §3.2.2.

The C++ improvements are the followings:

- **Sort algorithm modifications**: We have used quick-select instead of quick-sort. For more details check §3.3.1.
- **Records quantized removal improvement**: Explained in §3.3.2.
- **Basic linear algebra subprograms (BLAS) use for distance computation**: Use of BLAS described in §3.3.3, when calculating distances to vectors.
- **Intrinsic parallelization**: Internal parallelization of the algorithm, described in §3.3.4.
- **Extrinsic parallelization**: External parallelization of the algorithm, described in §3.3.5.
The compute unified device architecture (CUDA) C++ experiments consists of using the same algorithm, including all the improvements above mentioned implemented in graphics processing unit (GPUs).

### 1.2. Work Packages and Gantt diagram

#### 1.2.1. Work Packages

<table>
<thead>
<tr>
<th>Task: Research</th>
<th>WP ref: 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Software Installation</td>
<td>Sheet 1 of 5</td>
</tr>
<tr>
<td>Short description: Installation of MATLAB R2015b, Eclipse for C and C++ and CUDA programming.</td>
<td>Planned start date: 15/09/2016&lt;br&gt;Planned end date: 17/09/2016</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>WP ref: 1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Review current documentation</td>
<td>Sheet 2 of 5</td>
</tr>
<tr>
<td>Short description: In order to achieve a required knowledge of k-anonymous microaggregation and MDAV is necessary to review the current documentation about the topic.</td>
<td>Planned start date: 18/09/2016&lt;br&gt;Planned end date: 30/12/2016</td>
</tr>
<tr>
<td>Internal task T1: Search of documentation about of k-anonymous microaggregation and MDAV.</td>
<td></td>
</tr>
<tr>
<td>Internal task T2: Reading and understanding of found documentation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>WP ref: 1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Develop C++ MDAV</td>
<td>Sheet 3 of 5</td>
</tr>
<tr>
<td>Short description: Develop MDAV algorithm in C++ with the proper algebraic and programming improvements.</td>
<td>Planned start date: 01/10/2016&lt;br&gt;Planned end date: 10/10/2016</td>
</tr>
<tr>
<td>Internal task T1: First MDAV development in C++</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>WP ref: 1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Improve C++ MDAV with CPU parallelism</td>
<td>Sheet 4 of 5</td>
</tr>
<tr>
<td>Short description: Improve the MDAV code with CPU parallelism in order to know how is CPU improving the efficiency and compare with the normal one.</td>
<td>Planned start date: 11/10/2016&lt;br&gt;Planned end date: 31/10/2016</td>
</tr>
</tbody>
</table>
**Internal task T₁: Try CPU parallelism in MDAV**

<table>
<thead>
<tr>
<th>Task</th>
<th>WP ref: 1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Develop CUDA MDAV</td>
<td>Sheet 5 of 5</td>
</tr>
<tr>
<td>Short description: Develop a CUDA MDAV without any parallelism.</td>
<td>Planned start date: 01/11/2016  Planned end date: 15/1/2016</td>
</tr>
</tbody>
</table>

**Internal task T₁: Develop with CUDA**

<table>
<thead>
<tr>
<th>Task</th>
<th>WP ref: 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Improve CUDA MDAV with GPU parallelism</td>
<td>Sheet 1 of 4</td>
</tr>
<tr>
<td>Short description: Improve the CUDA MDAV code with GPU parallelism in order to see the efficiency improvement.</td>
<td>Planned start date: 16/11/2016  Planned end date: 03/12/2016</td>
</tr>
</tbody>
</table>

**Internal task T₁: Improve the CUDA MDAV code with GPU parallelism**

<table>
<thead>
<tr>
<th>Task</th>
<th>WP ref: 1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Compare and graph the different implementations</td>
<td>Sheet 2 of 4</td>
</tr>
<tr>
<td>Short description: In order to see the improvements in time efficiency between the implementations and parallelisms with CPU and GPU we need to compare them.</td>
<td>Planned start date: 04/12/2016  Planned end date: 20/12/2016</td>
</tr>
</tbody>
</table>

**Internal task T₁: Compare and graph the different implementations**

<table>
<thead>
<tr>
<th>Task</th>
<th>WP ref: 1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: Think of more things to improve the code or implementation</td>
<td>Sheet 3 of 4</td>
</tr>
<tr>
<td>Short description: In order to see if there is anything more that we can do to improve the efficiency think of more improvements.</td>
<td>Planned start date: 21/12/2016  Planned end date: 31/12/2016</td>
</tr>
</tbody>
</table>

**Internal task T₁: Think of more improvements to do**
1.2.2. Gantt Diagram

1.2.3. Deviation from the initial plan and incidences

There has not been much deviation along the project; almost every part of the initial plan has been completed. Nonetheless, as every project, there are always some minor changes to the initial plan.

The first deviation was the use of low level languages, as the prior approach was to focus mostly on MATLAB, unfortunately, MATLAB is a 4th generation language not conceived for developing production-ready software. So, in order to implement it we used Eclipse first, but then we realized that Visual Studio is a better developer environment and with a better compiler. Another noteworthy problem, is the fact that the implementation of such algorithm is not trivial, so it took more time than previously expected.

Overall, as all the tool used in this project are not mundane and easy to learn, the expected times have always been slightly increased.

2. Background and state of the art

We have remarked that, in general, the most extensively studied aspects of privacy for any information system deal with unauthorized access to sensitive data, by means of authentication, policies for data-access control and confidentiality, implemented as cryptographic protocols. However, the provision of confidentiality against unintended observers fails to address the practical dilemma when the intended recipient of the information is not fully trusted. Even more so, when the database collected is to be made accessible to external parties, or openly published for scientific correlating sensitive information with demographics.
2.1. Fundamentals of Statistical disclosure control (SDC)

Statistical disclosure control is a technique used in data-driven research to ensure no person or organization is identifiable from the statistical results of the analysis of surveys or administrative data, to protect the confidentiality and subjects of the research. This sensible information usually contains a set of attributes that may be classified into identifiers, quasi-identifiers and confidential attributes. Firstly, identifiers allow the unequivocal identification of the individuals. For example, full names, National ID or medical record numbers, which in order to prevent the identification of the individual and preserve the anonymity would be deleted before the publication of the microdata. Secondly, quasi-identifiers also known as key attributes, are those attributes that, in combination, may be linked with external, usually publicly available information to reidentify the respondents to whom the records in the microdata set refer. This is the case for age, address, gender, job, and physical features. Finally, the dataset contains confidential attributes with sensitive information on the respondent, such as salary, political affiliation, religion, and health condition.

Intuitively, the perturbation of numerical or categorical quasi-identifiers enables us to preserve privacy to a certain extent, at the cost of losing some of the data utility, in the sense of accuracy with respect to the unperturbed version. $k$-Anonymity is the requirement that each tuple of key-attribute values be identically shared by at least $k$ records in the dataset. This may be achieved through the microaggregation approach illustrated by the synthetic example depicted in Fig. 2. Rather than making the original table available, we publish a $k$-anonymous version containing aggregated records, in the sense that all quasi-identifying values within each group are replaced by a common representative tuple.

In Fig. 2, name is an identifier; gender, age and ZIP code are quasi-identifiers, and monthly income and geolocation are confidential attributes. As can be seen, identifiers are removed before publishing the table. Further, the published table contains groups of $k$ records with a common value for its quasi-identifiers. In fact, the published table is a $k$-anonymous version of the original one containing aggregated records. This prevents people and/or companies from the possibility of linking unambiguously the corresponding record by combining information in the published table with information from external sources.

Fig. 2. Example of $k$-anonymous microaggregation of published data with $k = 3$.

As a result, a record cannot be unambiguously linked to the corresponding record in any external sources assigning identifiers to quasi-identifiers. In principle, this prevents a privacy attacker from ascertaining the identity of an individual for a given record in the microaggregated database, which contains confidential information.
Ideally, microaggregation algorithms strive to introduce the smallest perturbation possible in the quasi-identifiers, in order to preserve the statistical quality of the published data. More technically speaking, these algorithms are designed to find a partition of the sequence of quasi-identifying tuples in $k$-anonymous cells, while reducing as much as possible the distortion incurred when replacing each original tuple by the representative value of the corresponding cell. For numerical key attributes representable as points in the Euclidean space, the mean-squared error (MSE) is the usual criterion to quantify said distortion. Data utility is measured inversely as the distortion resulting from the perturbation of quasi-identifiers.

As can be seen in Fig. 3, $k$-anonymous microaggregation is a similar process to vector quantization, except for the fact that it imposes the restriction that cells must be of size $k$ (in some cases there may be a few cells with a number similar to $k$, depending on if the total number of records is multiple of $k$).

Fig. 3. Example of microaggregation, with $k = 5$. Each tuple of 5 points is assigned a representative (centroid).

2.2. State of the art on k-anonymous microaggregation

Ideally, microaggregation algorithms strive to introduce the smallest perturbation possible in the quasi-identifiers, in order to preserve the statistical quality of the published data. More technically speaking, these algorithms are designed to find a partition of the sequence of quasi-identifying tuples in $k$-anonymous cells, while reducing as much as possible the distortion resulting from the modification of the tuple with the corresponding cell.

Microaggregation algorithms for numerical data use the mean squared error (MSE), defined as

$$\text{MSE} = \sum_{j=1}^{n} ||x_j - \hat{x}_j||^2$$

where $n$ is the number of $m$-dimensional records in the dataset, $x_j \in \mathbb{R}^m$ is the $j^{th}$ record, and $\hat{x}_j$ is the representative for this record, and this is used as the usual criterion to quantify said distortion. In the context of SDC, $n$ is the number of individuals in the released survey, and $m$ the number of numerical quasi-identifiers. Data utility is measured inversely as the distortion resulting from the perturbation of quasi-identifiers. The problem of this kind of algorithm lies in finding a $k$-partition with minimum MSE.
Even though many $k$-anonymity microaggregation algorithms have been proposed, this project focuses on the maximum distance to average vector (MDAV) due to its excellent functional performance in terms of low MSE for a given $k$-anonymity constraint, so we would like to conclude this section with the proper description of the algorithm’s behavior. It was proposed in [7] as a practical evolution of a multivariate fixed-size microaggregation method in conceived in [1]. Our specification of MDAV, formalized as Algorithm A, is functionally equivalent simplification of that given as Algorithm 5.1 in [2], and termed MDAV “generic”.

**Algorithm A. MDAV “generic”, functionally equivalent to Algorithm 5.1 in [2]**

<table>
<thead>
<tr>
<th>function MDAV</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong> $k$, $(x_n)<em>{n=1}^N$  (\triangleright) Anonymity parameter $k$, quasi-ID portion $(x_n)</em>{n=1}^N$ of dataset of $N$ records</td>
</tr>
<tr>
<td><strong>output</strong> $q$ (\triangleright) Assignment function from records to microcells $n \rightarrow q(n)$</td>
</tr>
</tbody>
</table>

1. **while** $2k$ points or more in the dataset remain to be assigned to microcells  
2. find the centroid (average) $C$ of those remaining points  
3. find the furthest point $P$ from the centroid $C$, and the furthest point $Q$ from $P$  
4. select and group the $k-1$ nearest points to $P$, along with $P$ itself, into a microcell, and do the same with the $k-1$ nearest points to $Q$  
5. remove the two microcells just formed from the dataset  
6. **if** there are $k$ to $2k-1$ points left **then**  
7. form a microcell with those and finish  
8. **else** (there are 1 to $k-1$ points)  
9. adjoin them to the last microcell \(\triangleright\) Typically also nearest microcell

Fig. 4. Pseudo-code of the maximum distance to average vector (MDAV).

MDAV is the best known and most widely used $k$-anonymous microaggregation algorithm, and has been proven to be a good perform in terms of time and one of the best regarding the homogeneity of the resulting groups. The computational cost of MDAV is quadratic with the number of records of the dataset. More concretely, it can be shown that the computational cost of MDAV can be expressed as $\frac{n^2}{k}(m + m_0)$ with $m_0 \geq 4$ and dependant on the computer on which the algorithm is executed. Nevertheless, when the number of records is too big, the microaggregation process becomes unfeasibly. Additionally, when the total number of dimensions $m$ increases, the computational cost also becomes bigger. That is why, as explained in the next section, we introduce some techniques and modifications in order to gain speed execution, which scales with the number of records and the number of dimensions.

Our study of the computational improvements of the $k$-anonymous microaggregation algorithm for large data-sets is not a novel study, and although we include some new improvements, there are other studies such as the PhD thesis which tests CPU and GPU parallelization in a different way we do [8]. However, the results obtained in our study are strikingly better.

2.3. Fundamentals of parallel computing

One of the reasons parallelization is a powerful tool is its high scalability, in order to execute many tasks simultaneously the more processing units at hand the more processes able to handle. That means that there is another way of improving execution speed as well as improving the frequency scaling, which was the dominant reason of improvement in computer performance from the mid-
1980s until 2004. *Frequency scaling* is the technique of increasing a processor’s frequency so as to enhance the performance of the system containing the processor in question.

The optimal implementation and execution of a program in parallel computing comes when all the parts of the algorithm are parallelizable. This is called *embarrassingly parallel*, which is a workload or problem where little or no effort is needed to separate the problem into a number of parallel tasks [5]. This is often the case where there is little or no dependency or need for communication those parallel tasks, or for results between them [3]. A common example of an embarrassingly parallel problem is 3D rendering handled by a *graphics processing unit*, where many vertices and pixels may be handled with no interdependency. The term is first found in the literature in a 1986 book on multiprocessors by MATLAB’s co-founder Cleve Moler [10], who claims to have invented the term.

Nonetheless, there are always problems when a technology is implemented. When an algorithm is not completely parallelizable the serial parts will slow down the execution, this is called *Amdahl’s Law* (or Amdahl’s argument [14]), optimally the speedup from parallelization would be linear, doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. Nevertheless, very few algorithms achieve optimal speedup. Most of them have a near-linear speedup for small numbers of processing elements, which flattens out into constant value for large numbers of processing elements.

The potential speedup of an algorithm on a parallel computing platform is given by

\[ S_{\text{effective}}(s) = \frac{1}{1 - p + \frac{p}{s}}, \]

where

- \( S_{\text{effective}} \) is the theoretical overall speedup of the execution of the entire task.
- \( s \) is the speedup of the execution of the parallelizable part of the task.
- \( p \) is the percentage of the execution time of the whole task concerning the parallelizable part of the task before parallelization.

Interestingly, Amdahl’s Law definition may be obtained from the harmonic mean which expresses the reciprocal of the arithmetic mean of the reciprocals, and is one of the Pythagorean means. Consider the definition of the harmonic mean

\[ H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}, \]

or its weighted version,

\[ H[x, y] = \frac{1}{\frac{1}{x} + \frac{y}{p}}, \]

Routine manipulation, with \( p_1 = (1 - p) \), \( p_2 = p \), \( x = 1 \), and \( y = s \), leads to

\[ H[1, s] = \frac{1}{1 - p + \frac{p}{s}}, \]

which finally gives Amdahl’s Law.
Fig. 5. A graphical representation of Amdahl’s law.

Since $S_{\text{effective}} < 1/(1 - p)$ it shows that a small part of the program which cannot be parallelized will limit the overall speedup available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (serial) parts. If the non-parallelizable part of a program accounts for the 10% of the runtime ($p = 0.9$), we can get no more than a 10 times speedup, regardless of how many processors are added as we can see in Fig. 5. This puts a limit on the usefulness of adding more parallel execution units.

Though the problem of a non-parallelizable part of the code can be one of the problems for a program not to run fast enough, there are other problems that should be mentioned. Parallel slowdown, for example, is a phenomenon where parallelization of an algorithm beyond a certain point causes the program to run slower. This is usually the result of a communication bottleneck. As more processors nodes are added, each processing node spends progressively more time doing communication than useful processing. At some point, the communications overhead created by adding another processing node surpasses the increased processing power that nodes provides, and parallel slowdown occurs. The typical occurrence of parallel slowdown is when the algorithm requires significant communication, particularly of intermediate results. So in order to not get this parallel slowdown, there should be a previous brief study on the number of processors were the parallelization would be optimum, this would enquire testing the algorithm with different number of working threads separations and make a relation between the time needed for every job and the communication time. This kind of problem depend on the device where the execution is being made, so the program running time in one hardware could improve in another and vice versa.

Another feature worth mentioning is the algorithm predisposition to being parallelized. For that reason we will have to introduce two properties, superadditive and subadditive. A superadditive function is defined by the inequality $f(x + y) \geq f(x) + f(y)$, which means that division of the problem in smaller parts and then combining them, is better than solving the problem directly. It can be shown that any convex function of a nonnegative real variable which vanishes at zero is superadditive. This includes the case of algorithms with quadratic running time, such as MDAV.

A powerful method to improve the running time of superadditive algorithms is the divide and conquer technique, which is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The
solutions to the sub-problem are then combined to give a solution to the original problem. Consider the division of a simple task on $n$ data samples with running time $t(n) = n^2$. The total running time, assuming no cost due to breaking down or recombining the parts, would be

$$2t\left(\frac{n}{2}\right) = 2\left(\frac{n}{2}\right)^2 = \frac{n^2}{2} = \frac{t(n)}{2},$$

that is, half of the original running time. This division may often be repeated recursively.

On the other hand, if the reverse is happening, we call them subadditive

$$f(x + y) \leq f(x) + f(y).$$

Examples include affine running times of the form

$$t(n) = an + b, \quad \text{with} \quad a, b \geq 0,$$

and sublinear functions such as $t(n) = n^a$ with $a \in (0,1)$. The real life example of this kind of problems are economies of scale, which in microeconomics are the cost advantages that enterprises obtain due to size, output, or scale of operation, with cost per unit of output generally decreasing with increasing scale as fixed costs are spread out over more units of output. Economies of scale apply to a variety of organizational and business situations and at various levels, such as a business, or manufacturing unit, plant or an entire enterprise. For example, a large manufacturing facility would be expected to have a lower cost per unit of output than a smaller facility, all other factors being equal, while company with many facilities should have a cost advantage over a competitor with fewer.

### 3. Project Development

The previous §2.3 shows the troubles on optimizing a program or algorithm. It requires a deep understanding and knowledge about it.

The description of the problem as well as the theoretical analysis is held during this section. Furthermore, a detailed description of each improvement is written.

#### 3.1. Experimental setup

To realize this project some tools were used. Below, we list all programs that we have been using during the development of our project.

<table>
<thead>
<tr>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB R2015b</td>
</tr>
<tr>
<td>Eclipse NEON</td>
</tr>
<tr>
<td>Minimalist GNU for Windows (MinGW)</td>
</tr>
<tr>
<td>Visual Studio 2015</td>
</tr>
<tr>
<td>Open Multi-Processing (OpenMP)</td>
</tr>
<tr>
<td>Intel Parallel Studio 2017</td>
</tr>
<tr>
<td>CUDA 8.0 from NVIDIA</td>
</tr>
</tbody>
</table>

For the first implementation of the MDAV algorithm, we used MATLAB considering its power and compact programming language.

Moving on to the next development tool, Eclipse Neon was the first one used for the C++ implementation, which is an integrated development environment (IDE), however, in order to compile and run the code the use of a compiler was needed, so we decided to use Minimalist GNU for Windows (MinGW) which is a minimalist development environment for native Microsoft Windows.
applications. It is noteworthy to mention that MinGW and Eclipse NEON are open sourced, no license needed, nonetheless, Eclipse was first meant to be used for Java development. For that reason another IDE was decided to be used, explained in the paragraph below.

In order to test a better environment, we decided to give it a try and use professional software as Visual Studio 2015, which is meant to be used for C development and has the proper tools for its evaluation and a better compiler. Also, it is user friendly and has a better library management. Open Multi-processing (OpenMP) API and the Intel Parallel Studio 2017 library were used for the parallel computing in CPU, described in §3.3.4, and the CUDA 8.0 from NVIDIA was used in the GPU implementation, explained in §3.4.

The hardware used in this project is an Intel i7 4790 CPU with Hyper-Threading, and a NVIDIA GeForce GTX 960 GPU with 1024 CUDA Cores and 2GB of Standard Memory Config (VRAM).

3.2. Improvements developed on MATLAB
The early implementation of the MDAV algorithm was developed on MATLAB given its powerful tools and the compact programming language with optimized built-in methods, which makes it the best environment for the evaluation and fundamental testing of the algorithm.

The MDAV algorithm goal is to generate groups of records in clusters where its input must be a data-set with N number of records and M dimensions that would be the quasi-identifiers as explained in §2.1.

The improvements made in MATLAB are mainly algebraic modifications making our program faster and more optimized.

![Fig. 6. MATLAB improvements.](image)

3.2.1. Single distance calculation exploit
As we can see in Fig. 6, this is the first improvement implemented, however, it is needed to mention a part of the MDAV code to correctly explain the improvement. This is the part responsible of computing the distance between a point (in this case the centroid), and the remaining points, and after this, finding the furthest one named P. Then the next step is selecting the $k-1$ nearest points to P and adding them into a microcell. The same process is followed on to the next process, calculate points distance to P, finding furthest point Q, and selecting $k-1$ points nearest to Q.

The paragraph before describes the way it was first implemented, however, in order to select the $k-1$ points nearest to P it was necessary to calculate the distance of all the points to P, and after that, the same computation needed to be made to find the furthest point Q. So, in order to not calculating twice this distance, it was modified and changed to only compute it once and then reusing it, which ended up being the most efficient.
3.2.2. Distance algebraic improvement

The next improvement is an algebraic modification. This is related to the calculation of the distance, and it is a huge advance because it is totally parallelizable, which is one of the things we were looking for in the algorithm.

It is named distance calculation made by norm pre-computation; it follows a concept where there is a part of the equation previously computed which is then used to facilitate the execution of the problem. First of all it is needed to describe the Euclidian distance of a vector to another:

\[ D_s = \|x - x_0\|^2. \]

In any computation where distances need to be compared, say in finding the closest point \( x \) to \( x_0 \), or the one furthest from it, instead of computing \( \|x - x_0\|^2 \), we may save a few operations by employing the following algebraic trick. First, observe that

\[
\|x - x_0\|^2 = \|x\|^2 + \|x_0\|^2 - 2\langle x, x_0 \rangle,
\]

where \( \frac{1}{2}\|x\|^2 \) will be precomputed at the start of the program, once before any loops, and then reused afterwards. Aside from saving a few computations by computing \( \langle x, x_0 \rangle \) in lieu of \( \|x - x_0\|^2 \), it turns out that the implementation of the inner product for several samples \( x \) as a matrix multiplication may be done extremely efficiently with sophisticated algebraic libraries optimized to this end.

3.2.3. Centroid subtraction improvement

Another improvement worth mentioning is the computation of the centroid made by the subtraction of quantized points, this procedure is a mathematical improvement. The original idea was to recalculate the centroid every time needed, so the mathematical representation was

\[
X_0 = \frac{\sum_{i=0}^{N} x_i}{N}
\]

where

- \( X_0 \) is the centroid.
- \( N \) is the number of records or points.
- \( x_i \) is the sample which is being added to the centroid sum.

However, there is the possibility of optimizing this. The proposition is the following, first, computing the sum of all the points once, after that the only thing needed is to divide by the number of records not quantized. Finally, when the records have been yet aggregated to a microcell, they are subtracted from the previous sum of points, which will be used for the centroid calculation afterwards. The process would be as follows.

Firstly, this is the previous sum calculation where \( X_{\text{sum}} \) is the sum of all records made once at the beginning,

\[
X_{\text{sum}} = \sum_{i=0}^{N} x_i.
\]

Secondly, the first iteration to calculate the centroid will be as follow:

\[
X_0 = \frac{X_{\text{sum}}}{N}.
\]

Thirdly, at the end of the loop (Fig. 4), being \( j \) a iterator from 0 to the size of the cell \( k \), the quantized records are removed from the sum \( X_{\text{sum}} \) with the equation
\[ X_{\text{sum}} = X_{\text{sum}} - \sum_{j=0}^{k} x_j. \]

Lastly, when the loop gets to the calculation of the centroid once again, the centroid calculation becomes as follows:

\[ X_0 = \frac{X_{\text{sum}}}{N_{\text{left}}}, \]

where \( N_{\text{left}} \) is the number of points not quantized yet, and \( X_0 \) will be the centroid for the left records.

The reason why this is a better approach, is because if we calculate the centroid in the original way, the number of operations that must be done would be \( O(n) \) plus a division every time, however, with our new trick, we decrease the number of operations to \( O(k) \) plus a division, knowing that \( k \ll n \). As a consequence, the time will be much smaller than the previous one.

### 3.3. Improvements developed on C++

In this section, we illustrate the improvements made on C++, these improvements are algorithmic, algebraic and parallelization approaches. Also, the improvements made on the previous §3.2 using MATLAB are also included in this one.

#### 3.3.1. Sort algorithm improvement

The first improvement is a sort algorithmic improvement. The problem is that in order to get the \( k \) nearest elements from P and Q, after calculating the distance from every vector, then a sorting of the elements in the array was used, and finally it selected the \( k \) nearest. Nowadays, there are a large number of sorting algorithms, so we decided to go for the most efficient one, the quick-sort.

At the beginning of the project, we have been using quicksort algorithm to sort an array that was an extremely important step to select the kth smallest element from P and Q. The quick-sort algorithm was developed by Tony Hoare and published in 1961 [6], it is inspired in the “divide and conquer” technique, explained in §2.3, which is an optimization technique used for superadditive algorithms, also defined in §2.3, such as quick-sort. Moreover, the time complexity of this improved sorting method shows that, on average the algorithm takes \( O(n \log n) \) comparisons to sort \( n \) items, while the normal sort takes \( O(n^2) \). The algorithm works as follows.

1. Choose a random element of the list which we will call **pivot**.
2. Partition, reorder the array so that all the elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it(equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partitioning operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

However, our algorithm does not need to have all the elements sorted, the requirement we have is to select the $k$ nearest (minor) elements to the point $P$ and $Q$. As a consequence, we decided to use a select algorithm, which only returned the minor elements we are searching for. We decided to use a variation of the quick-sort algorithm, named as *quick-select*. This algorithm is related to *quick-sort*. It was also developed by Tony Hoare, and is also known as Hoare’s selection algorithm [6]. *Quick-select* and its variants are the selection algorithms most often used in efficient real-world implementations.

*Quick-select* uses the same overall approach as *quick-sort*, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in *quick-sort*, *quick-select* only recurses into one side, the side with the elements it is searching for. Eventually, it will get to the point that the $k$ selected elements would be the resulting ones. Additionally, this algorithm reduces the average complexity to $O(n)$.

The execution of MDAV the select algorithm is used $N/k$ times, where $N$ is the number of records and $k$ is the cell size, making the usage of quick select algorithm an important advance. In addition, quick-select can be parallelized, although this has not been possible to be done in this project due to a lack of time.

### 3.3.2 Records quantized removal improvement

This section describes the code improvement performed on the first C++ implementation. We realized that in order to delete records from an array, MATLAB had a built-in method which has almost no cost, nonetheless, our implementation in C++ was not the most efficient one. In order to delete $k$ elements from an array, our first approach was to move each element to the left of the array.

As we can see in Fig. 8 for every record to remove from the array, all the records are also moved to the left. After realizing that this was not the optimal implementation of the remove records algorithm, we thought of a better one, which deleted the $k$ records with less iterations, the algorithm is shown in Fig. 9.
Indeed, the number of operations in this new removal of records implementation is around $k$ times faster compared to the first one. If $p$ is the first sample position to remove, and $n$ is the total number of records left, we will have $n - p$ operations to execute, however, in the initial one we had to run this $k * (n - p)$ operations.

### 3.3.3. Algebraic libraries for distance calculation

To improve our MDAV algorithm, first of all, we decided to check which code lines in MDAV consumes the highest percentage of time in comparison to the total time of the program execution. The result was that the distance calculation has the highest percentage in almost 60% of total time of execution.

As we have observed there are a large number of techniques for optimizing process, and in our project, the main goal is to improve the performance of the MDAV algorithm. For that reason, we analyzed which was the process of the code with more percentage of execution time. After analyzing it, we found out that it was the distance calculation, (see Fig. 4).

So to reduce the time execution of the distance calculation, which is a linear algebra operation, is by using a library called Basic Linear Algebra Subprograms (BLAS). BLAS are routines that provide standard building blocks for performing basic vector and matrix operations. The level 1 BLAS perform scalar, vector and vector-vector operations, the level 2 BLAS perform matrix-vector operations, and the level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software, LAPACK for example.

However, there are different libraries regarding this BLAS routines. Eventually, we tried to use the more efficient one, which was the Intel Math Kernel Library (MKL). The routine we make use of in our project, is the matrix-vector multiplication in the level 3 BLAS.

```c
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
        m, n, k, alpha, A, k, B, n, beta, C, n);
```

In our case, $A$ is the matrix and $B$ is a vector, where

- $m$= number of records
- $n$=1 dimension of the vector.
- $k$= number of dimensions.

Which are the elements defining the matrix-vector multiplication.

### 3.3.4. Intrinsic parallelization

As described in §2.3, parallelization is a powerful tool that can be performed in every computer with more than 1 processing units. Unfortunately, there are some limitations to this technique. One of the reasons why the parallelization section is separated in two parts, is the fact that the MDAV algorithm...
is not entirely parallelizable without having some modifications on its final results. For that reason, we decided to implement two different ways of parallelization: *intrinsic parallelization*, and *extrinsic parallelization* explained in §3.3.5.

The parallelization of the algorithm was made with Open Multi-Processing (OpenMP), which is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran, on most platforms, processor, architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. Additionally, it eases the implementation of parallel computing with a simple line of code.

The main goal, is to parallelize internally the algorithm. However, we must keep in mind that not all lines of the code could be parallelized as we will observe and discuss in the results section.

For this kind of parallelization we analyzed every part of the code, in order to detect the lines with more proportion of execution time. The lines of code with more execution time where:

- The distance calculation,
- The quantized records removal,
- The selecting algorithm
- The finding of the max element.

However, we realized that the execution time for the distance calculation is directly proportional to the number of dimensions, this means that the percentages would vary in proportion of the dimensions.

<table>
<thead>
<tr>
<th>Code section</th>
<th>Is parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance calculation</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection algorithm</td>
<td>Yes</td>
</tr>
<tr>
<td>Find max element</td>
<td>No</td>
</tr>
<tr>
<td>Used records removal</td>
<td>Yes</td>
</tr>
</tbody>
</table>

In order to parallelize the sections of the code we use the OpenMP as stated before, and the lines of code we need to use are the following ones.

1. `#pragma omp parallel private()`
2. `#pragma omp for schedule (static)`

These two lines are the ones in charge of starting the parallelization in a “for-loop”. Nevertheless, in order to parallelize independent process the lines are.

1. `#pragma omp parallel private()`
2. `#pragma omp sections`
3. `#pragma omp section`

The time complexity for every part of the code parallelized expected is \( t/c \), where \( t \) is the time spent on the process and \( c \) is the number of cores used for the parallelization, so, if we use 4 cores we will make this part 4 times faster. Nevertheless, we have to consider *Amdahl’s law* and *parallel slowdown* §2.3.
3.3.5. Extrinsic parallelization

Using the external parallelization we have obtained the best results in terms of time execution. The technique used is the “divide and conquer”, consisting on separating the algorithm in sub-problems of the same type. The steps to follow on this approach are, separating the records into groups using MDAV itself, and then applying the MDAV algorithm to each one of the groups previously obtained. This would highly improve the performance of the algorithm, however, if we parallelize the execution of every MDAV we will improve it even more.

The time complexity in a normal MDAV is $O(n^2/k)$, however, if we use the “divide and conquer” technique to the dataset, the time complexity would be as follows:

$$t_{ext} = \frac{n^2}{c} + c \left( \frac{n^2}{k} \right)^2 = cn + \frac{n^2}{ck},$$

where

- $c$ is the number of groups made at the first MDAV.
- $n$ is the number of records.
- $k$ is the size of every cell.
- $cn$ is the pre-partition time, and the other part is the time for the $c$ MDAV.

In big data, normally the number of records $n$ is high, in this case we can say that, $\ll n^2/ck$, and as a consequence the time complexity would be approximately $O(n^2/k)$.

On the other hand, if we use the parallelization, the execution time would be:

$$t_{ext} = \frac{nc}{\alpha p} + \frac{1}{\beta p} c \frac{n^2}{k} = \frac{nc}{\alpha p} + \frac{n^2}{\beta pck},$$

where

- $p$ is the number of cores on the CPU.
- $\alpha$ is the efficiency of intrinsic parallelization.
- $\beta$ is the efficiency of extrinsic parallelization

3.4. Improvements developed on CUDA

This section describes the improvements developed on compute unified device architecture (CUDA), which is a parallel computing platform and API model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing, an approach termed General-Purpose computing on graphics processing units (GPGPU). The CUDA platform is a software layer that gives direct access to the GPU’s virtual instruction set and parallel computational elements, for the execution of compute kernels.

This platform is designed to work with programming languages such as C, C++ and Fortran. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL, which required advanced skills in graphic programming. In order to use CUDA we had to use a hardware from NVidia, in our case we used a GTX GeForce 960 GPU, otherwise, we would not have been able to test this tool.

As this devices are made to be used in the computer game industry, they are made to execute simple operations, such as the rendering, and for game physics calculation. However, CUDA has also been used to accelerate non-graphical applications in computational biology, cryptography and other fields by an order of magnitude or more. For that reason we tried to use it in our project for its
algebraic computation power, so we had to implement kernel functions on CUDA, in order to use them inside the code developed for the MDAV in C++.

The problems we encountered with this technology, is the fact that we need two types of memory, one for the CPU (host memory) and another for the GPU (device memory), when we want to use some kernel function, in GPU, we need to copy this memory from the host to the device and the other way, the main problem of this is that, this process is a performance hit due to system bus bandwidth and latency.

![CUDA processing flow](image)

Fig. 10. Example of CUDA processing flow.

The process described in Fig. 10, is the following:

- Copy data from main mem to GPU mem
- CPU instructs the process to GPU
- GPU execute parallel in each core
- Copy the result from GPU mem to main mem

As seen in Fig. 10, this process can create a bottleneck for the algorithm, so we tried to use the maximum number of kernel functions.

4. EXPERIMENTAL RESULTS

After having explained the different techniques used to achieve a reduction in the execution time, here we present some experimental results obtained after running them. Different datasets have been used, with different number of records (points) or varying the dimension (number of quasi-identifiers).

4.1. Results for MATLAB

In this section we compare the different improvements realized in the MDAV algorithm, comparing them to the first approach. As we can see, Fig. 11 shows the improvements performance with different number of records.
Fig. 11. MDAV Elapsed time for $k=10$ and $M=10$.

It is shown in the graph, that every improvement decreases the total execution time. Indeed, these improvements are additive. The single distance calculation improvement, is 15% more efficient than the first approach, the distance algebraic improvement is 16%, and the computation of the centroid by subtraction is 22.8% faster. However, as the distance algebraic improvement is modified by the single distance computation the percentage of those two combined is not the sum of them, is slightly smaller. Finally, the performance with all of the improvements together is 35.2% more efficient.

4.2. Results for C++

The results of the improvements done in C++ are shown in the following section. First of all, we illustrate the selecting algorithm improvements. As in Fig. 12 is shown, it is compared the performance of quick-sort against quick-select.

Fig. 12. Quick-sort and Quick-select.
We explained in §3.3.1, that quick-select is the most efficient algorithm in our project. As we can see in Fig. 12, quick-sort increases much more than the quick-select in proportion with the number of records. We used a selecting number of \( k = 50 \) records for the quick-select, a reasonable number for big data-sets, to obtain these results.

Moving on to the records quantized removal improvement, which is a big increase of efficiency. As it is explained in §3.3.2, this new implementation makes the removal of records algorithm \( k \) times faster. However as the MDAV algorithm has to use it 3 times in every iteration, this new approach is making the removal of records \( 3k \) times faster than the previous version.

The next improvement is the use of the algebraic routines called BLAS with the Intel MKL software. The graph below Fig. 13, is a representation of the MDAV without and with the use of the library. In addition, the number of records used is \( n = 50000 \) and the cell size is \( k = 10 \).

![MDAV without and with BLAS](image)

Fig. 13. MDAV without and with BLAS.

As it can be observed in Fig. 13, the elapsed time increases along with the number of dimensions for both methods, however, with BLAS the slope is drastically smaller than without it. Additionally, if the number of dimensions is small, there is no big difference between both of them, nonetheless, as our project goal is to optimize the algorithm for large datasets, and big dimensions, we can benefit from this feature.

4.3. Intrinsic parallelization results

Starting from this point, all the improvements illustrated in sections before, are included in the next experiments. In this section the intrinsic parallelization results are described.

First of all what we analyzed the code parts with more percentage of total execution time, in order to see where is worth using the parallelization.

The percentages represented in Fig. 14, are for a number of records, \( n = 100000, m = 10 \), and a cell size \( k = 10 \). As we can observe in the bar chart, the percentage for the distance calculation is almost 60% and for the sorting of elements is more than 20%, which are the ones with more proportion of total execution time. Finally, the ones with less percentage are, the finding of maximum record, which is more than 10%, and the quantized records removal, which is around 5%. Noticeably, the part which is spending the most time in the algorithm is the distance calculation, and although we have optimized this part a few times, is still the most time consuming part.
However, as Fig. 13 shows, we can see that the distance calculation execution time is increasing along with the number of dimensions $m$. So we will be able to obtain a greater speed-up with a bigger number of dimensions. The figure below uses number of records $n = 100000$, and $k = 10$.

Unfortunately, these results obtained and exposed in Fig. 15, are not the expected ones. Considering that when parallelizing an algebraic routine such as a matrix multiplication, we should obtain a speed-up of $c$ times faster, where $c$ is the number of cores used (in our case $c = 4$). However, distance calculation is a vector-matrix multiplication, which is not as superadditive as, for example, a matrix multiplication. In order to demonstrate that, we tested the parallelization on a matrix multiplication and the results were that it was almost 4 times faster.

In our case, we can see in the graph that we have around 20% of speed-up, which is a good result worth using. Moreover, these results are also because of a parallel slowdown, where the time used for parallelizing is higher than the communication between the threads, probably happening in the operation.

Apart from the slowdowns described before, it is worth mentioning the fact that when parallelizing parts of the algorithm, the total speed-up is affected. This is because of the Amdahl’s law explained in §2.3, for example with the percentages obtained in Fig. 14, with a 65% of the algorithm
to parallelize, the distance calculation and the removal of quantized records, both with a 35% speedup, the maximum parallelization would be of

\[ S_{\text{effective}} = \frac{1}{(1 - 0.65) + 0.65/s} = \frac{1}{0.35} = 2.85. \]

where \( s \) would be \( \infty \). However, with our speedup, \( s = 1.35 \), we obtain

\[ S_{\text{effective}} = \frac{1}{0.35 + 0.65/s} = \frac{1}{0.35 + 0.65/1.35} = 1.202, \]

so we are obtaining a 20% as commented before.

Another feature worth mentioning is the hyper-threading in the hardware used, the CPU used in this project has a modification in it, which is supposed to improve the performance of the multi-threading programming, this feature doubles the number of processing units, so in our case instead of having 4 we have 8 available. However, we have observed that, when using the 8 processing units the performance of the parallelization dropped, so in order to optimize the maximum, we use the normal 4.

4.4. Extrinsic parallelization results

The next improvement is the external parallelization using the “divide and conquer” technique, this is the best approach for the time optimization, because has the best results. There is a little difference from previous methods, which is the slight alteration of the final results and distortion.

We can see that the results obtained in Fig. 16, comparing an extrinsic parallelization, with an external separation, are not a great improvement, the parallelization is speeding up the algorithm a little bit.
Fig. 17. MDAV with 4 external areas no parallelized and parallelized.

In the figure above Fig. 17 we can see how the parallelization affects to the elapsed time, we can see that with a big number of dimensions Fig. 16 and a big number of records, the speed-up will increase.

Fig. 18. MDAV Intrinsic and MDAV extrinsic.

The figure above represents the elapsed time along the number of dimensions, for a number of records, $n = 100000$, and cell size $k = 10$, the graph shows how with only 4 areas for the extrinsic technique we can get a big speed-up. If we increase the number of areas we will get an even better result. However, the distortion will be affected with every new separation, the figure below describes this behavior.
The graph above Fig. 19, describes the relation between the time and distortion from the serial MDAV with a number of records $n = 100000$, number of dimensions $m = 10$, and a cell size $k = 10$. We can observe that the intrinsic parallelization does not affect the distortion, and has a speedup of around 25%. However, the extrinsic parallelization is a lot faster than the intrinsic, but with a higher distortion. It is worth mentioning the increase of speed-up with a bigger number of pre-partitions (areas).

4.5. GPU massive parallelization results

This sections shows the results obtained with the CUDA implementation for GPU massive parallelization, as we can see in the figure below, Fig. 20.
The graph illustrated in Fig. 20, represents the elapsed time along with the number of dimensions, $m$, with a number of records $n = 100000$ and a cell size of $k = 10$. It shows that, GPU parallelization is a better approach than CPU for big dimensions. It is worth mentioning that, for a $m = 200$ the speed-up is of 6.5 times faster, so it is a highly valuable tool.

4.6. Overall results

Finally, the picture Fig. 21, illustrates the overall percentages of improvement. The percentages equation used is

$$P = \frac{T_{\text{improvement}}}{T_{\text{old}}} \times 100,$$

which calculates the speedup. C++ improvements include the MATLAB improvements, that is the reason they are in separated tables. The second column of the table in the figure, illustrates the speedup on the part of the algorithm itself, and then the third column, shows the speedup of improvement in the whole algorithm.

<table>
<thead>
<tr>
<th>MATLAB Improvement</th>
<th>Speedup</th>
<th>MDAV Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single distance calculation optimization</td>
<td>128%</td>
<td>117%</td>
</tr>
<tr>
<td>Distance algebraic improvement</td>
<td>130%</td>
<td>119%</td>
</tr>
<tr>
<td>Centroid subtraction improvement</td>
<td>1000%</td>
<td>129%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++ Improvements</th>
<th>Speedup</th>
<th>MDAV Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort algorithm modification</td>
<td>≈400%</td>
<td>≈133%</td>
</tr>
<tr>
<td>Samples quantized removal improvement</td>
<td>≈774%</td>
<td>≈154%</td>
</tr>
<tr>
<td>BLAS use for distance computation</td>
<td>≈400%</td>
<td>≈333%</td>
</tr>
<tr>
<td>Intrinsic parallelization</td>
<td>--</td>
<td>125%</td>
</tr>
<tr>
<td>Extrinsic parallelization 4 pre-partitions</td>
<td>--</td>
<td>1098%</td>
</tr>
<tr>
<td>CUDA C++ parallelization</td>
<td>--</td>
<td>650%</td>
</tr>
</tbody>
</table>

Fig. 21. Improvements percentages.
5. BUDGET

In this project, no hardware has been needed. Only new algorithms have been developed, and thus, the total economic cost of the project should be measured in terms of the hours spent by each one of the project members. Below is the breakdown of hours spent by each one, as well as the total cost of the project.

<table>
<thead>
<tr>
<th>Project member</th>
<th>Number of hours</th>
<th>Hourly cost (€/hour)</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alejandro García Álvarez</td>
<td>470</td>
<td>10.00*</td>
<td>4700€</td>
</tr>
<tr>
<td>David Rebollo-Monedero</td>
<td>80</td>
<td>34.56*</td>
<td>2,756.8€</td>
</tr>
<tr>
<td>Ahmad Mohamad Mezher</td>
<td>80</td>
<td>30.00*</td>
<td>2,400€</td>
</tr>
</tbody>
</table>

In conclusion, the total cost of this project has been 9,856.8 €.

*This value is an estimation.
6. Conclusions and Future Development

Nowadays, the use of statistical techniques in big data is increasing considerably, and the main problem is the sensible information managed. The privacy and anonymity of people and companies involved in these large-scale datasets is crucial, and thus, the need for efficient methods to protect them is of great importance.

A widely used technique for this kind of protection is $k$-anonymous microaggregation. An algorithm with excellent performance in terms of low distortion for a given $k$-anonymity constraint is the maximum distance to average vector (MDAV) algorithm. MDAV is certainly a heuristic which attains $k$-anonymity with a great preservation of the statistical quality of the information, and with small datasets the microaggregation process is reasonable in terms of time. However, when dealing with larger datasets, the execution time is prohibitive.

In this study, algebraic modifications, divide-and-conquer algorithmic improvements, among other techniques, and last but not least, parallel computation both on CPU and graphics processing units (GPUs), have been introduced in the MDAV algorithm. Indeed, all the improvements held in the project substantially improve the computational performance of the algorithm, leading to a most valuable reduction in running time for its practical application to real, large-scale datasets.

Apart from the extrinsic parallelization, all the other improvements carry with a time gain, which happens to be a free gain, because they do not imply any consequence in the resulting output. Moreover, some of the improvements, such as the intrinsic parallelization and the GPU implementation are highly scalable and can vastly increase its performance with hardware modifications. However, the extrinsic parallelization, which happens to be the best approach, comes with a small distortion loss because of its computational procedure. Fortunately, the distortion loss is reasonable when compared to the time gained, and the greater the dataset, the more noticeable the performance improvement.

In order to implement an MDAV algorithm, the first approach in the performance hierarchy would be the extrinsic parallelization, when distortion loss is secondary, secondly massive GPU parallelization, when in possession of the hardware needed and the distortion loss is important, and finally, intrinsic parallelization with CPU, if statistical quality is needed and no hardware for the previous mentioned available.

In addition, there are some improvements left to implement on this project, such as the parallelization of the selecting algorithm, which would increase the performance in the intrinsic parallelization. Also the introduction of BLAS in the GPU implementation, which observing the results obtained in CPU can be very astonishing. Finally, the optimization of the extrinsic parallelization, where finding the optimal number of pre-partitions could increase drastically the time gain. This will be part of the future work to be done with this project.
REFERENCES


