Parallel Euclidean distance matrix computation on big datasets

Mélodie Angeletti, J.-M. Bonny, Jonas Koko

Parallel Euclidean distance matrix computation on big datasets *

Mélodie Angeletti[1,2], Jean-Marie Bonny[2], and Jonas Koko[1]

[1]LIMOS, Université Clermont Auvergne, CNRS UMR 6158, F-63000 Clermont-Ferrand, France (melodie.angeletti@uca.fr, jonas.koko@uca.fr)
[2]INRA AgroResonance - UR370 QuaPA, Centre Auvergne-Rhône-Alpes, Saint Genes Champanelle, France (Jean-Marie.Bonny@inra.fr)

Abstract

We propose, in this paper, three parallel algorithms to accelerate the Euclidean matrix computation on parallel computers. The first algorithm, designed for shared memory computers and GPU, uses a linear index to fill the block lower triangular part of the distance matrix. The linear index/subscripts conversion is obtained with triangular number and avoid loops over blocks of columns and rows. The second algorithm (designed for distributed memory computer) in addition to linear index uses circular shift on a 1D periodic topology. The distance matrix is computed iteratively and we show that the number of iterations required is about half the number of processors involved. Numerical experiments are carried-out to demonstrate the performances of the proposed algorithms.

Keywords: Euclidean distance matrix, parallelization, mutlicores, many-core, GPU

1 Introduction

The distance matrix refers to a two-dimensional array containing the pairwise distance (e.g., Euclidean, Manhattan, cosine) of a set of element and is a good measurement to tell the differences beween data points. Thus the distance matrix is widely used in various research areas using datasets: data clustering ([2]), pattern recognition [6], image analysis, many-body simulation [3], systems biology [10]. These applications involve the distance matrix computation between n entities (instances) with each of them described by a m-dimensional vector (attributes).

In many applications, the distance matrix computation is a step of a more complex algorithm (e.g. Ward algorithm). But the run-times are often dominated by the distance matrix computation. For instance, in [1], the distance matrix computation takes at least 4/5 of total computation time (in sequential implementation) fro clustreing of MRIf data.

*This work was supported by a research allocation SANTE 2014 of the Conseil Régional d'Auvergne

Standard implementations of the Euclidean distance matrix is to split the original matrix in tiles (or chunks/blocks) of columns (or lines). Two nested loops are then applied to fill the distance matrix. Since the distance matrix is symmetric with zero diagonal, only the lower of upper triangular part ic computed. We propose two algorithms for the distance matrix computation on huge datasets: one for share memory/GPU computers and another for the distributed memory computers. We use the triangular numbers properties to collapse the two nested loops over the blocks, in contrary to [7] who use external pattern (Map-Reduce) to avoid nested loops. Our strategy leads to efficient implementation for shared memory computers and GPU. The distributed memory algorithm is based on circular shift using a 1D periodic topology. The distance matrix is computed iteratively, each processor filling one block per iteration. We show that the number of iterations required is almost the number of tiles. The proposed paper is an extension to [1].

The paper is organised as follows. In Section 2 we described the standard implementation of the Euclidean distance matrix with two nested loops on blocks. In Section 3 we present our algorithm for the Euclidean distance matrix computation on shared memory and GPU computers, followed by the distributed memory algorithm in Section 4. The numerical experiments are carried-out in Section 5.

2 Euclidean distance matrix

Let us consider an m by n dataset matrix $\boldsymbol{X}$ where n is the number of data points and m is the size of the feature space. The i-th column of $\boldsymbol{X}$ is dentoed by $\boldsymbol{x}_i$, i.e. $\boldsymbol{X} = [\boldsymbol{x}_1\ \boldsymbol{x}_2\ \cdots\ \boldsymbol{x}_n]$. Computing the square Euclidean distance matrix for $\boldsymbol{X}$ consists in computing the symetric n by n matrix $\boldsymbol{D} = (d_{ij})$ with entries

$$d_{ij} := d(\boldsymbol{x}_i, \boldsymbol{x}_j) = \| \boldsymbol{x}_i - \boldsymbol{x}_j \|^2, \quad 1 \leq j < i \leq n$$

The complexity for computing each pair d_{ij} is $O(m)$. Since $\boldsymbol{D}$ has zeros diagonal entries ($d_{ii} = 0$), the storage requirement is $n(n-1)/2$. The standard algorithm for computing $\boldsymbol{D}$ consists of calculating each d_{ij} inside of two nested loop, using BLAS like routines or language intrinsic routines (i.e. `dot_product` in FORTRAN). En practice, this standard algorithm has a low performance even with small size problems.

Unlike other distance matrices, Euclidean distance matrix can be computed using a matrix operations based method to achieve the maximum performance. If we set

$$\boldsymbol{e} = (1\,1\cdots 1)^\top \text{ and } \boldsymbol{s} = (\| \boldsymbol{x}_1 \|^2 \cdots \| \boldsymbol{x}_n \|^2)^\top$$

D can be computed by using the vector-based formula

$$d_{ij} = \| \boldsymbol{x}_i \|^2 + \| \boldsymbol{x}_j \|^2 - 2\boldsymbol{x}_i^\top \boldsymbol{x}_j, \quad 1 < j < i < n \tag{2.1}$$

or the matrix-based formula

$$D = \boldsymbol{s}\boldsymbol{e}^\top + \boldsymbol{e}\boldsymbol{s}^\top - 2\boldsymbol{X}^\top \boldsymbol{X}. \tag{2.2}$$

Then, as for the dense matrix-matrix multiplication, a blocking strategy (see, e.g., [4, 5] is required to extract out the best performance of the modern multi-core/many-core systems.

Blocking (or tiling/chuncking) is a well-known optimization technique for improving the effectiveness of memory hierarchies of the modern processors.

A blocking/tiling method consists of splitting $\boldsymbol{X}$ into $M \times N$ blocks $\boldsymbol{X}_{IJ}$ of size $m_B \times n_B$, with

$$\begin{aligned} m_B &= \lfloor m/M \rfloor, \quad n_B = \lfloor b/N \rfloor \\ \boldsymbol{X}_{IJ} &= (x_{ij}), \quad i = (I-1)m_B + 1, \ldots, \bar{m}_B, \quad j = (J-1)n_B + 1, \ldots, \bar{n} - B, \end{aligned} \tag{2.3}$$

where $\bar{m}_B = \min(Im_b, m)$ and $\bar{n}_B = \min(Jn_B, n)$. Consequently, sub-vector $\boldsymbol{s}_J$ of columns norms of $\boldsymbol{X}_{:J}$ becomes

$$\boldsymbol{s}_J = \sum_{L=1}^{M} \left(\boldsymbol{X}_{LJ}^\top \cdot \boldsymbol{X}_{LJ}^\top \right) \boldsymbol{e}_J,$$

where $(\cdot)$ stands for element-wise (or Hadamrd) multiplication. The decomposition of $\boldsymbol{X}$ into blocks (2.3) induces a decomposition of $\boldsymbol{D}$ into a block (lower) triangular $\boldsymbol{D} = (\boldsymbol{D}_{IJ})$ such that

$$\boldsymbol{D}_{IJ} = \boldsymbol{s}_I \boldsymbol{e}_J^\top + \boldsymbol{e}_J \boldsymbol{s}_I^\top - 2 \sum_{L=1}^{M} \boldsymbol{X}_{LI}^\top \boldsymbol{X}_{LJ} \tag{2.4}$$

where $\boldsymbol{s}_J$ and $\boldsymbol{e}_J$ are subvectors of $\boldsymbol{s}$ and $\boldsymbol{e}$ corresponding to tile $X_{:J}$. If $I = J$, only the lower triangular part of $\boldsymbol{D}_{IJ}$ is stored. For each tile I we compute the Euclidean distance between its columns. Then, we compute the Euclidean distance between each pairs formed of a column of tile I and a column of another tile J. Loop tiling is efficient even in sequential implementation because it improves the temporal data locality, and hence the cache utilization. As the distance matrix is symmetric with zero diagonal entries, we keep only the lower triangular part and store it in a vector $\boldsymbol{d}$ of size $n(n-1)/2$. The tile-based algorithm for the Euclidean distance matrix calculation is presented in Algorithm 1. Algorithm 1 has a severe drawback: it cannot be parallelized efficiently since loops in I (line 4) and J (line 7) cannot be merged. As a result, a poor scalability parallel algorithm with speed-up that can collapses from a number of processors [1].

3 Efficient Euclidean distance matrix algorithm

An efficient parallelization of Algorithm 1 requires the transformation of the double loop in (I, J) into a single loop in K from 1 to $N(N+1)/2$, the number of blocks D_{IJ}, for $J <= I$. Fortunately, there is a connection between the conversion of the linear index K into subscripts (I, J), and the triangular numbers.

3.1 Triangular numbers and index/subscripts conversion

The ith triangular number t_i is the number obtained by adding all positive integers less than or equal to a given positive integer i, i.e.

$$t_i = 1 + 2 + \cdots + i = i(i+1)/2.$$

Algorithm 1 Tile based Euclidean distance matrix calculation algorithm

Require: $\boldsymbol{X}$ of size $m \times n$, n_T and $N = \lfloor n/n_T \rfloor$
Require: $\boldsymbol{D}$ (block) distance matrix

1: **for** $J = 1$ to N **do**
2: $\quad \boldsymbol{s}_J = \sum_{L=1}^{M} \left(\boldsymbol{X}_{LJ}^\top \cdot \boldsymbol{X}_{LJ}^\top \right) \boldsymbol{e}_J$
3: **end for**
4: **for** $I = 1$ to N **do**
5: $\quad$ **for** $J = 1$ to I **do**
6: $\quad\quad \boldsymbol{D}_{IJ} = \boldsymbol{s}_I \boldsymbol{e}_J^\top + \boldsymbol{e}_J \boldsymbol{s}_I^\top - 2 \sum_{L=1}^{M} \boldsymbol{X}_{LI}^\top \boldsymbol{X}_{LJ}$
7: $\quad$ **end for**
8: **end for**

The triangular number are therefore 1, 3, 6, 10, 15, ... Furthermore, if t_i is the ith triangular number, then

$$i = \frac{1}{2} \left(\sqrt{8t_i + 1} - 1 \right). \tag{3.1}$$

Now consider, e.g., the following 4×4 lower triangular matrix in which the entries are replaced by the corresponding linear index k in the storage vector (row-wise numbering)

$$\begin{bmatrix} \mathbf{1} & & & \\ 2 & \mathbf{3} & & \\ 4 & 5 & \mathbf{6} & \\ 7 & 8 & 9 & \mathbf{10} \end{bmatrix}$$

The diagonal entries (i.e. 1, 3, 6, 10) correspond to the first four triangular numbers. We easily verify that the entry $(1, 1)$ is stored in $k = 1 = t_1$, the entry $(2, 2)$ in $k = 3 = t_2$, the entry $(3, 3)$ in $k = 6 = t_3$ and $(4, 4)$ in $k = 10 = t_4$. The linear index k of the diagonal (i, i) is given by $k = t_i = i(i + 1)/2$. Knowing k it is easy, using (3.1), to get i. If k is not a triangular number, then i given by (3.1) is no longer an integer. But rounding it down we obtain the highest value of i for which $t_i \leq k$ and $i + 1$ is the row in which the entry k is located in the original matrix. To get the column, we just substract t_i from k, i.e. $j = k - i(i + 1)/2$. Algorithm 2 describes the conversion of a linear index k to lower triangular subscripts (i, j).

In Algorithm 2, the entries are numbering by rows in the orginal matrix. In Algorithm 3, the entries in the original matrix are numerbing column-wise as follows

$$\begin{bmatrix} \mathbf{1} & & & \\ 2 & \mathbf{5} & & \\ 3 & 6 & \mathbf{8} & \\ 4 & 7 & 9 & \mathbf{10} \end{bmatrix}$$

Algorithm 2 Converts the linear index k to lower triangular subscripts (i, j), row-wise numbering

Require: k the storage index, n the size of the triangular matrix
$p = (\sqrt{1 + 8 * k} - 1)/2$
$i_0 = \lfloor p \rfloor$
if $i_0 = p$ **then**
$i = i_0$, $j = i$
else
$i = i_0 + 1$, $j = k - i_0 * (i_0 + 1)/2$
end if

Algorithm 3 Converts the linear index k to lower triangular subscripts (i, j), column-wise numbering

Require: k the storage index, n the size of the triangular matrix
$k' = n(n + 1)/2 - k$
$p = \lfloor (\sqrt{1 + 8 * k'} - 1)/2 \rfloor$
$i = k - n(n - 1)/2 + p(p + 1)/2, \;\; j = n - p$

If the original matrix is strictly lower triangular, we do not need to store the diagonal entries as in Algorithm 3-2. For a column-wise storage, it suffices to replace, in Algorithm 3, n by $n - 1$ and i by $i - 1$. We get Algotithm 4.

Algorithm 4 Converts the linear index k to lower strictly triangular subscripts (i, j), column-wise numbering

Require: k the storage index, n the size of the triangular matrix
$k' = n(n - 1)/2 - k$
$p = \lfloor (\sqrt{1 + 8 * k'} - 1)/2 \rfloor$
$i = k - n(n - 1)/2 + p(p + 1)/2, \;\; j = n - 1 - p$

3.2 Efficient Euclidean distance matrix algorithm

The efficient distance matrix procedure is described in Algorithm 5. Loop over blocks D_{IJ} (Line 4-7) is now entirely linear and can therefore be parallelized efficiently. Note that, in practice, the size of end blocks ($\{\boldsymbol{X}_{LN}\}_L$ or $\{\boldsymbol{X}_{MJ}\}_J$) can be different from the size of the others blocks.

4 Distributed algorithm

Algorithm 5 is not suitable for a distributed system: a collection of processors that do not share memory nor clock. For a distributed memory system, the only way one processor

Algorithm 5 Efficient Euclidean distance matrix calculation algorithm

Require: $\boldsymbol{X}$ of size $m \times n$, $\boldsymbol{d}$ of size $n(n-1)/2$.
Require: m_B, n_B, $M = \lfloor m/m_B \rfloor$ and $N = \lfloor n/n_B \rfloor$

1: **for** $J = 1$ to N **do**
2: $\quad \boldsymbol{s}_J = \sum_{L=1}^{M} \left(\boldsymbol{X}_{LJ}^\top \cdot \boldsymbol{X}_{LJ}^\top \right) \boldsymbol{e}_J$
3: **end for**
4: **for** $K = 1$ to $N(N+1)/2$ **do**
5: $\quad$ Compute block subscripts (I, J) from K, e.g. using Algorithm 2
6: $\quad \boldsymbol{D}_{IJ} = \boldsymbol{s}_I \boldsymbol{e}_J^\top + \boldsymbol{e}_J \boldsymbol{s}_I^\top - 2 \sum_{L=1}^{M} \boldsymbol{X}_{LI}^\top \boldsymbol{X}_{LJ}$
7: **end for**

can exchange information with another is through passing information explicitly through the network. This type of architecture has the significant advantage that it can scale up to large numbers of processors (up to hundreds of thousands proessors).

At first glance, the distance matrix computation is not suitable for distributed memory calculations since we have to fill a block triangular matrix. Indeed, a direct adaptation of Algorithm 5 leads to a unpracticable topology with variable number of processors involved. There is a need to design a new algorithm for distributed memory computers. The distributed algorithm proposed in this Section is based on cyclic shifts. A circular shift is a permutation which shifts all elements of a set by a fixed offset, with the elements shifted off the end inserted back at the beginning (right shift), and the elements shifted off at the beginning inserted back at the end (left shift). Formally, a circular shift is a permutation σ of the n entries in the set $\{1, 2, \ldots, n\}$ such that, for all entries i

$$\sigma(i) = i + 1 \text{ (modulo } n) \quad \text{or} \quad \sigma(i) = i - 1 \text{ (modulo } n). \tag{4.1}$$

Applying (4.1) k times is equivalent to a shift of k places, denoted by σ_k and defined by

$$\sigma_k(i) = i + k \text{ (modulo } n) \quad \text{or} \quad \sigma_k(i) = i - k \text{ (modulo } n). \tag{4.2}$$

Let N be the number of processors available. We then split $\boldsymbol{X}$ into N tiles $\boldsymbol{X}_{:J}$ $(J = 1, \ldots, N)$ as described in Section 2. We assume that we can compute efficiently submatrix $D_{IJ} = d(\boldsymbol{X}_{:I}, \boldsymbol{X}_{:J})$, the distance matrix between any column of $\boldsymbol{X}_{:I}$ and $\boldsymbol{X}_{:J}$, by adptating Algorithm 5. In our distributed algorithm we start by assigning to processor I, the computation of the submatrix $D_{I,I}$. Then, we perform a circular (right) shift on the first subscript so that, in the second step, processor I computes submatrix $D_{I,\sigma(I)} = D_{I,I+1}$. In practice, the block subscripts are aranged in order to fill only the lower triangular part of the distance matrix. The circular shift procedure continues, Table 1, until all blocks D_{IJ} $(J \leq I)$ are computed. The following theorem shows that we

can fill the block triangular distance matrix using a successive circular shift procedures as a communication scheme between processors.

Process	1	2	$\cdots$	I	$\cdots$	N
1st Step	$D_{1,1}$	$D_{2,2}$	$\cdots$	$D_{I,I}$	$\cdots$	$D_{N,N}$
2nd Step	$D_{1,2}$	$D_{2,3}$	$\cdots$	$D_{I,I+1}$	$\cdots$	$D_{N,1}$
3rd Step	$D_{1,3}$	$D_{2,4}$	$\cdots$	$D_{I,I+2}$	$\cdots$	$D_{N,2}$
...						
kth Step	$D_{1,\sigma_k(1)}$	$D_{2,\sigma_k(2)}$	$\cdots$	$D_{I,\sigma_k(I)}$	$\cdots$	$D_{N,\sigma_k(N)}$

Table 1: The successive circular shift procedure on blocks

Theorem 4.1. *The cyclic shift procedure described in Table 1 stopps after* $(N+1)/2$ *steps.*

Proof. The lower/upper triangular part of D contains $N(N+1)/2$ blocks D_{IJ} and the cyclic shift procedure described in Table 1 generates N blocks each step. It is obvious that $(N+1)/2$ steps are needed to fill D. □

The main consequences of Theorem 4.1 are:

- If N is even, then at the last step only $N/2$ blocks are to be computed by N processes, as shown below with $N = 4$

$$\begin{array}{cccc} D_{1,1} & D_{2,2} & D_{3,3} & D_{4,4} \\ D_{1,2} & D_{2,3} & D_{3,4} & D_{4,1} \\ D_{1,3} & D_{2,4} & \mathbf{D_{3,1}} & \mathbf{D_{4,2}} \end{array}$$

- If N is odd, then in all steps, N blocks are to be computed by N processes, as shown below with $N = 5$

$$\begin{array}{ccccc} D_{1,1} & D_{2,2} & D_{3,3} & D_{4,4} & D_{5,5} \\ D_{1,2} & D_{2,3} & D_{3,4} & D_{4,5} & D_{5,1} \\ D_{1,3} & D_{2,4} & D_{3,5} & D_{4,1} & D_{5,2} \end{array}$$

In practice, cyclic shifts are performed efficiently by send/receive procedures in a 1-D cartesian periodic topology. A distributed algorithm for the Euclidean distance matrix calculation is described in Algorithm 6.

5 Numerical experiments

In this section we propose the performance results of the proposed algorithms for the computation of the Euclidean distance matrix. The numerical experiments are carried out on the following platforms:

COMP1 DELL PowerEdge R930 (shared memory) based on Intel Xeon E7-8890 with 2.50GHz CPU. The computer consists of 4 sockets of 18 cores each (i.e. 72 cores). Each socket has a private 45MB L3-cache, shared by all cores and each core has a private 256KB L2-cache and 32KB L1-cache. The total RAM is 3TB.

Algorithm 6 Distributed algorithm for the Euclidean distance matrix calculation

Require: $\boldsymbol{X}$ of size $m \times n$

Require: N the number of processors, organized as $1D$ cartesian periodic topology.

1: Split matrix $\boldsymbol{X}$ into N tiles $\boldsymbol{X}_{:I}$, $I = 1, \ldots, N$
2: Assign tile $\boldsymbol{X}_{:I}$ to processor I and compute the diagonal block D_{II}, $I = 1, \ldots, N$
3: Save a local copy of $\boldsymbol{X}_{:I}$ on each processor I
4: **for** $K = 2$ to $(N+1)/2$ **do**
5: Perform a circular shift scheme to exchange blocks between neighboring processors
6: Compute block $D_{I,\sigma_{K-1}(I)}$, for each processor $I = 1, \ldots, N$
7: **end for**

COMP2 DELL PRECISION RACK7910 based on 10-core Intel Xeon E5-2640 with 2.40GHz CPU and NVIDIA Quadro P5000 with 2560 CUDA cores. The total RAM is 62GB for the CPU and 16GB for the device. Note that in Quadro P5000, there are many fewer units devoted to double precision arithmetics. As a result the theoretical performance in double precision (277 GFLOPS) is more than three times lower than the single precision performance (8.9 TFLOPS).

We chose FORTRAN (2003-2008) language for its large number of intrinsic array functions. We study the performances of the following algorithms:

ALG1 FORTRAN parallel implementation of Algorithm 5 using OpenMP. The parallelisation is straightforward: the loops in J and K are parallelized using dynamic scheduling.

ALG2 FORTRAN parallel implementation of Algorithm 6 using OpenMPI library. As mentioned in Section 4, we use 1-D cartesian periodic topology and `MPI_SENDRECV_REPLACE` (send/receive messages using a single buffer). Note that if N (the number of processors) is small, the code generates large arrays whose size can quickly exceeds MPI count `INT_MAX` (i.e. 2^{31}). Then we have to use MPI derived types to support large counts for Algorithm 6 with small N. Since using Algorithm 6 with small a number of processors is not relevant, we present the performances of Algorithm 6 with $N \geq 4$.

ALG3 CUDA implementation of Algorithm 5 on COMP2. CUDA (Compute Unified Device Architecture) [9] is an extension of C programming language for using with NVIDIA Graphics Processing Units (GPUs) and thus benefit from many-core architecture. The GPU kernel consists in computing $\boldsymbol{s}_I$ and $\boldsymbol{s}_J$ on on CPU, copying $\boldsymbol{X}_I$, $\boldsymbol{X}_j$, $\boldsymbol{s}_I$, $\boldsymbol{s}_J$ on GPU, and computing the distance sub-matrix (2.4) using CUBLAS [8]. As the call to CUBLAS is asynchronous, while the GPU computes, the CPU copies the result of the previous GPU calculation. We also overlap the copy and the computation using streams.

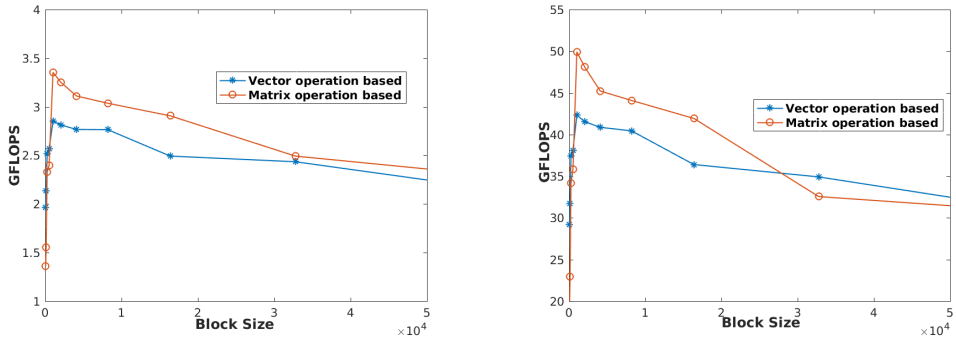


Figure 1: Performances of ALG1 on COMP1 (single precision) with varying block size: unicore (left), 18-core (right)

5.1 Benchmarking

The main difficulty in the parallel algorithms described in the previous sections is the choice of block sizes: the performance of the algorithms are strongly related to the partitionning of the matrices. Then a simple procedure is to perform a systematic benchmarking [11]. To this end, we consider a matrix of size $10^3 \times 10^4$. We begin the computations with blocks of size 4×16 since we deal with $m \times n$ rectangular matrices with $n >> m$. We then multiply, successively, each dimension by 2 to get blocks of size 8×16, 8×32, 16×32, etc. The performances are evaluated in terms of GFLOPS using one core or one socket (18-core for COMP1 and 24-core for COMP2), single precision or double precsision arithmetics. Note that unicore "optimal" block size will be used in the distributed Algorithm 6 while socket "optimal" block size will be used in Algorithm 5.

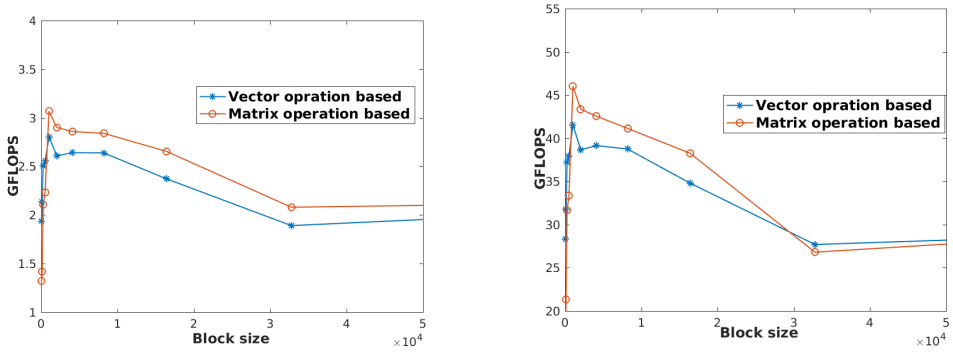


Figure 2: Performances of ALG1 on COMP1 (double precision) with varying block size: unicore (left), 18-core (right)

Figures 1-2 show the performances of ALG1 on COMP1. We observe that the performances peak at block size 16×64 for any mode (unicore or socket, single or double precision).

To benchmark ALG3 (on COMP2) we use a matrix of size $10^3 \times 3 \cdot 10^4$ and we begin with block of size 512×1024 to avoid excessive computational time due to small blocks. Figure 3 shows the performances of the CUDA implementation of Algorithm 5. For single or double precision arithmetics, the performances peak at block size 1000×2048. Note that using a test matrix with more rows and columns, the performances peak at block size 8192×16384.

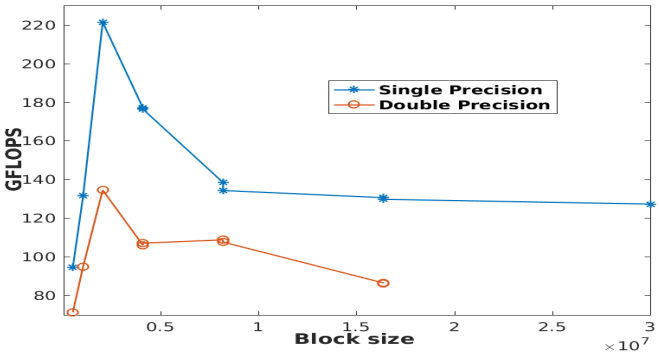


Figure 3: Performances of Algorithm 5 using COMP3 with varying block size

	$n = 32,000$		$n = 64,000$		$n = 128,000$		$n = 256,000$	
NPROC	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2
1	323.327	—	1270.853	—	5095.186	—	20335.726	—
2	164.552	—	659.487	—	2661.569	—	10820.598	—
4	82.317	101.622	333.825	408.717	1335.422	1646.935	5370.388	6589.662
8	42.230	44.551	169.643	178.967	698.557	715.106	2814.481	2875.162
16	22.107	23.585	89.599	94.839	368.040	377.659	1470.060	1513.006
32	11.850	13.248	46.603	52.718	188.248	209.852	751.216	841.881
64	6.565	8.223	24.908	32.221	95.577	128.490	381.067	517.898
ALG3	5.835		20.182		77.617		OoM	

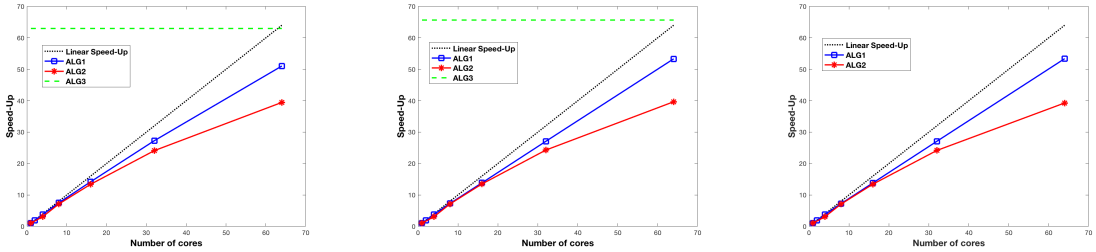
Table 2: Single precision execution times (Sec.) for ALG1-ALG3, $m = 1,000$ and various n.

5.2 Speed-up

We now present performances results of ALG1-ALG3 for different problems sizes and precisions. The data used were generated according (in problem size and data magnitude) to fMRI data ([1]): m is the number of scans and n is the number of voxels. Functional Magnetic Resonance Imaging (fMRI) is a noninvasive technique for studying brain activity. During the course of an fMRI experiment, brain images (scans) are acquired between 1001000 times, with each image consisting of roughly $20,000 - 20,0000$ volume elements (voxels). The experience can be repeated several times for the same subject (subject analysis), as well as for multiple subjects (group analysis) to facilitate the population inference. As shown by Thirion *et al.*, for a great number of clusters, the best clustering

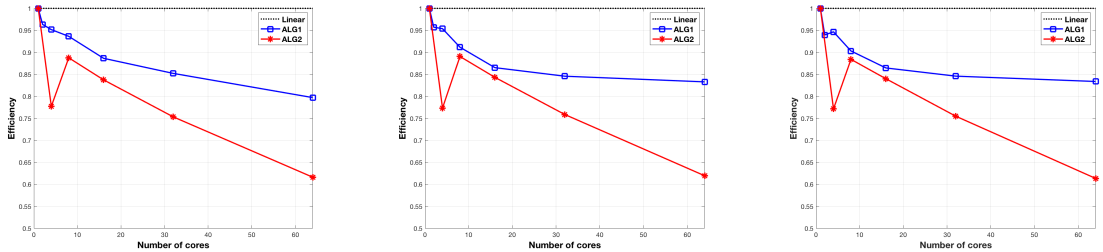
algorithm is the agglomerative hierarchical clustering with Wards criterion because of the goodness of the fitting and its reproducibility. Computing the Euclidean distance matrix for fMRI dataset is the first step for applying the Ward clustering algorithm.

We report in Table 2 single precision execution times (in Sec.) for ALG1-ALG3 on datasets with $m = 1000$ (subject analysis) and varying number of columns n. Figure 4-5 show the corresponding speed-ups and parallel efficiency with respect to a sequential implementation running on a single core of COMP1. We notice that ALG1 outperforms ALG2 in all cases, with an almost linear speed-up and a parallel efficiency of at least 80%. Note that the execution time of ALG2 includes communication times, e.g., the times devoted to messages passing and the time to collect the distance submatrices from all processors. ALG3 outperforms ALG1 and ALG2 with a speed-up of about 65. For the largest problem ($n = 256,000$) ALG3 fails because of the (host) CPU memory since the matrices ($\boldsymbol{X}$ and $\boldsymbol{D}$) must be loaded in CPU memory.



$n = 64,000$ $n = 128,000$ $n = 256,000$

Figure 4: Single precision speed-up for ALG1-ALG3, $m = 1000$ and various n.



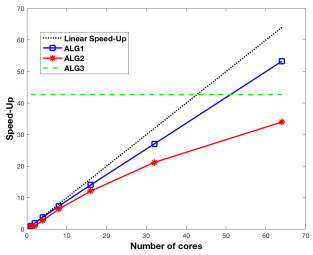
$n = 64,000$ $n = 128,000$ $n = 256,000$

Figure 5: Single precision efficiency for ALG1-ALG2, $m = 1000$ and various n.

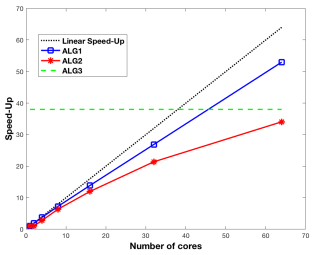
We report in Table 3 the performance results using double precision arithmetics with $m = 1000$ (subject analysis) and various values n. Figure 6-7 show the corresponding speed-ups and parallel efficiency with respect to a sequential implementation running on a single core of COMP1. Similar to the single preision results, ALG1 outperforms ALG2 with an almost linear speed-up. The speed-up, for ALG1, is between 50 and 60, and the parallel efficiency greater than 0.8 for all problem sizes. For ALG2, the parallel efficiency

n	$n = 32,000$		$n = 64,000$		$n = 128,000$		$n = 256,000$	
NPROC	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2
1	342.297	—	1373.270	—	5453.231	—	22137.345	—
2	177.214	—	711.573	—	2877.604	—	11639.543	—
4	89.828	120.721	359.484	491.837	1444.051	1957.464	5780.013	8056.589
8	46.150	53.499	187.482	214.528	751.333	857.863	3003.863	3494.472
16	24.179	28.464	97.755	113.572	392.937	455.158	1585.501	1802.658
32	12.730	16.252	50.724	64.770	202.828	254.906	811.418	1028.867
64	6.977	10.247	25.792	40.400	102.945	160.118	414.101	646.936
ALG3	9.259		32.147		143.528		OoM	

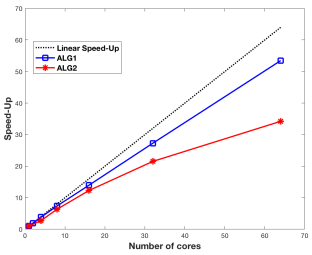
Table 3: Double precision execution times (Sec.) for ALG1-ALG3, $m = 1,000$ and various n.



$n = 64,000$

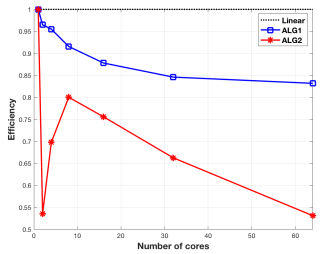


$n = 128,000$

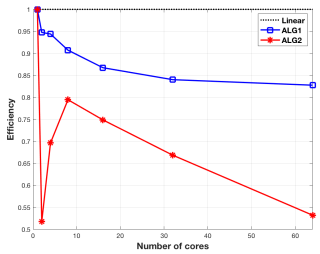


$n = 256,000$

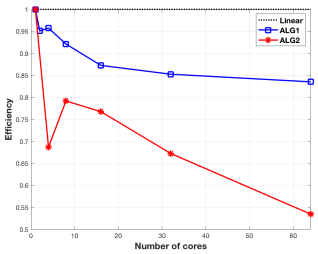
Figure 6: Double precision speed-up for ALG1-ALG3, $m = 1000$ and vaious n.



$n = 64,000$



$n = 128,000$



$n = 256,000$

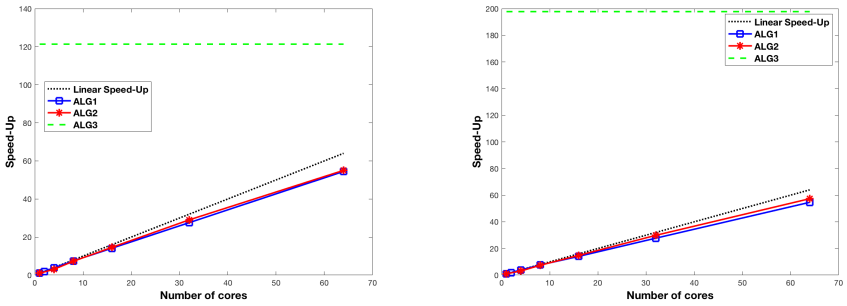
Figure 7: Double precision efficiency for ALG1-ALG2 with $m = 1000$ and various n.

is now about 0.5-0.55. An interesting observation, in the double precision case, is that the GPU implementation (ALG3) is outperformed by the OpenMP implementation (ALG2). For ALG3, the spped-up is now around 40, which is less than that for a single preicision (around 65). This is beacause of the small number of double precision units (FP64) in Quadro P5000 GPU.

	$m = 1,000$		$m = 5,000$		$m = 10,000$	
NPROC	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2
1	5095.186	—	24282.427	—	48455.671	—
2	2661.569	—	12497.112	—	24977.728	—
4	1335.422	1646.935	6202.373	7852.281	12362.524	15500.902
8	698.557	715.106	3257.418	3327.252	6372.089	6554.438
16	368.040	377.659	1717.981	1661.936	3445.819	3268.145
32	188.248	209.852	880.823	836.477	1748.490	1623.331
64	95.577	128.490	445.833	441.224	886.729	844.680
ALG3	77.617		200.097		244.990	

Table 4: Single precision execution times (Sec.) for ALG1-ALG3, $n = 128,000$ and various m.

We now study the behavior of ALG1-ALG3 on datasets from group analysis: $n = 128,000$ and $m = 1,000,\ 5,000,\ 10,000$ corresponding to one, 5 and 10 subjects. We report in Table 5 the single precision execution times of the algorithms. The speed-up achieved by the implementations are shown in Figure 9. ALG3 outperforms ALG1-ALG2 with a speed-up of 121 ($m = 5,000$) and 198 ($m = 10,000$). Another interesting fact is that the speed-up of ALG2 (MPI implementation) is almost equivalent to that of ALG1 for $m = 5,000$ and slightly better for $m = 10,000$ (57 versus 54).



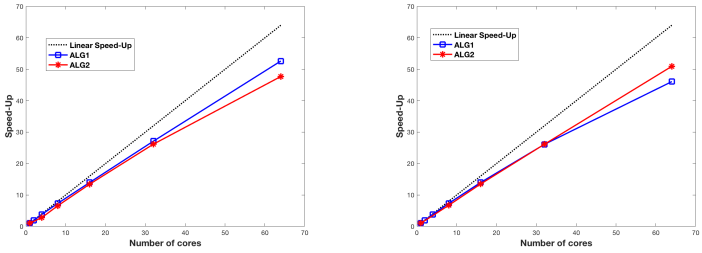
$m = 5,000$ $m = 10,000$

Figure 8: Single precision speed-up for ALG1-ALG3, $n = 128,000$ and various m.

We report in Table 5 the double precision preformance results for a dataset from group analysis. For ALG3 (GPU implementation) the execution fails for $m = 5,000$ and $m = 10,000$ because of the host CPU memory limit. The speed-up for ALG1-ALG2 are shown in Figure 9. For $m = 5,000$, ALG1 outperforms ALG2 but for the largest problem ($m = 10,000$), the speed-up obtained with ALG2 on 64 processors is about 50 compared to 46 for ALG1. We can think that on huge datasets, with a large number of processors, ALG2 will be better than ALG1.

	$m = 1,000$		$m = 5,000$		$m = 10,000$	
NPROC	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2
1	5453.231	—	26721.462	—	53218.082	—
2	2877.604	—	14072.942	—	28057.677	—
4	1444.051	1957.464	7040.014	9553.563	13931.026	—
8	751.333	857.863	3652.245	4071.456	7258.762	7925.922
16	392.937	455.158	1914.008	1991.188	3810.251	3947.947
32	202.828	254.906	985.877	1021.801	2039.642	2037.573
64	102.945	160.118	507.908	560.293	1155.392	1044.874
ALG3	143.528		OoM		OoM	

Table 5: Double precision execution times (Sec.) for ALG1-ALG3, $n = 128,000$ and various m.



$m = 5,000$ $m = 10,000$

Figure 9: Double precision speed-up for ALG1-ALG2 with $n = 128,000$ and various m.

6 Conclusion

We have studied parallel algorithms, on various architectures (share and distributed memory, GPU), for the Euclidean matrix computation on large datasets. Numerical experiments have shown that the three algorithms proposed are competitive based on architecture, problem size and single/double precision arithmetics.

Note that ALG3 allows to designed hybrid implementations. For instance, we can replace Step 6 of Algorithm 6 by ALG1 or ALG3 if blocks (I, J) are distributed over a set of nodes with multi-core CPUs of GPUs. Further work is underway to design hybrid implementations involving CPU and GPU.

The proposed algorithm can be easily adapted for other distance matrice computation, particularly l_p-distance defined by

$$d_{ij} = d(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left(\sum_{k=1}^{m} \mid x_{ki} - x_{kj} \mid^p \right)^{1/p}.$$

It suffices to replace matrix-based formulas, used for the Euclidean matrix, by a suitable formula.

References

[1] Angeletti M., Bonny J.-M., Duriff F. and Koko J. Parallel hierarchical agglomerative clustering for fMRI data. In *Proc. 12nd Conf. Parallel Processing and Applied Mathematics PPAM 2017*, volume 10777 of *LNCS*, pages 1–11. Springer, 2018.

[2] Berkhin P. . A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[3] Hendricksson B. and A:Plimpton S. Parallel many-body simulations without all-to-all communication. *Journal of Parallel and Distributed Computing*, 25:15–25, 1995.

[4] Krishnan M. and Nieplocha J. Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *Proc. 18th International Parallel and Distributed Processing Symposium IPDPS 2004*. IEEE, 2004.

[5] Lam M., Rothberg E. E. and Wolf M. E. The cache performance and optimizations of blocked algorithms. In *Proc. ASPLOS IV*, volume 19, pages 63–74. ACM SIGARCH Computer Architecture News, 1991.

[6] Li J. and Lu B.L. An adaptive image Euclidean distance. *Pattern Recognition*, 42:349–357, 2009.

[7] Li Q., Kecman V. and Salman R. A chunking method for Euclidean distance matrix calculation on large dataset. In *Proc. Int. Conf.Machine Learning and Applications*, pages 208–2013. IEEE, 2010.

[8] NVIDIA Corporation. *CUDA Tookit 4.1: CUBLAS Library*. NVIDIA Corporation, 2012.

[9] NVIDIA Corporation. *NVIDIA CUDA*. NVIDIA Corporation, 2012.

[10] Zola J., Aluru M. and Aluru S. Parallel information theory based construction of gene regulatory networks. In *Proc. 15th annual IEEE High-Performance Computing HiPC 2008*, volume 5374 of *LNCS*, pages 336–349, Berlin, 2008. Springer-Verlag.

[11] Zuckerman S., Pérache M. and Jalby W. Fine tuning matrix multiplications on multicore. In *Proc. 15th annual IEEE High-Performance Computing HiPC 2008*, volume 5374 of *LNCS*, pages 30–41, Berlin, 2008. Springer-Verlag.