Task-based randomized singular value decomposition and multidimensional scaling

Emmanuel Agullo, Olivier Coulaud, Alexandre Denis, Mathieu Faverge, Alain Franc, Jean-Marc Frigerio, Nathalie Furmento, Adrien Guilbaud, Emmanuel Jeannot, Romain Peressoni, et al.

Task-based randomized singular value decomposition and multidimensional scaling

Emmanuel Agullo, Olivier Coulaud, Alexandre Denis, Mathieu Faverge, Alain Franc, Jean-Marc Frigerio, Nathalie Furmento, Adrien Guilbaud, Emmanuel Jeannot, Romain Peressoni, Florent Pruvost, Samuel Thibault

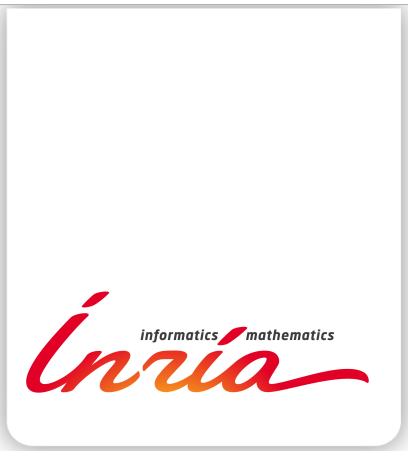
RESEARCH REPORT

N° 9482

September 2022

Project-Team Concace, Hiepacs, Pleiade, Storm, Tadaam

ISSN 0249-6399 ISRN INRIA/RR--9482--FR+ENG



Task-based randomized singular value decomposition and multidimensional scaling

Emmanuel Agullo*, Olivier Coulaud*, Alexandre Denis*, Mathieu Faverge*, Alain Franc†, Jean-Marc Frigerio†, Nathalie Furmento*, Adrien Guilbaud*, Emmanuel Jeannot*, Romain Peressoni*, Florent Pruvost*, Samuel Thibault*

Project-Team Concace, Hiepacs, Pleiade, Storm, Tadaam

Research Report n° 9482 — September 2022 — 37 pages

Abstract: The multidimensional scaling (MDS) is an important and robust algorithm for representing individual cases of a dataset out of their respective dissimilarities. However, heuristics, possibly trading-off with robustness, are often preferred in practice due to the potentially prohibitive memory and computational costs of the MDS. The recent introduction of random projection techniques within the MDS allowed it to be become competitive on larger test cases. The goal of this manuscript is to propose a high-performance distributed-memory MDS based on random projection for processing data sets of even larger size (up to one million items). We propose a task-based design of the whole algorithm and we implement it within an efficient software stack including state-of-the-art numerical solvers, runtime systems and communication layers. The outcome is the ability to efficiently apply robust MDS to large data sets on modern supercomputers. We assess the resulting algorithm and software stack to the point cloud visualization for analyzing distances between sequences in metabarcoding.

Key-words: task-based programming, randomized singular value decomposition (RSVD), multidimensional scaling (MDS), random projection, distributed memory, heterogeneous machine, runtime system

* Inria, Talence, France
† Inria, Talence, France (Firstname.Name@inria.fr) and Inrae BioGeCo, Cestas, France

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Décomposition en valeurs singulières randomisée et positionnement multidimensionel à base de tâches

Résumé : Le positionnement multidimensionnel (MDS) est un algorithme important et robuste pour représenter les cas individuels d'un ensemble de données en fonction de leurs dissimilarités respectives. Cependant, des approches heuristiques, qui peuvent induire un compromis en termes de robustesse, sont souvent préférées en pratique en raison de de la consommation mémoire et des coûts potentiellement prohibitifs du MDS. L'introduction récente de techniques de projection aléatoire dans le MDS lui a permis de devenir compétitif sur des cas test plus importants. L'objectif de ce manuscrit est de proposer un MDS haute performance basé sur la projection aléatoire pour le traitement d'ensembles de données de taille encore plus grande (jusqu'à un million d'éléments). Nous proposons une conception de l'algorithme et nous l'implémentons dans une pile logicielle efficace, comprenant des solveurs numériques de pointe ainsi des systèmes d'exécution et des couches de communication optimisés. L'aboutissement de ce travail résultat est la capacité d'appliquer efficacement le MDS robuste à de grands ensembles de données sur des super-ordinateurs modernes. Nous évaluons l'algorithme et la pile logicielle résultants à la visualisation de nuages de points pour l'analyse des distances entre séquences de metabarcoding.

Mots-clés : programmation à base de tâches, décomposition en valeur singulière randomisée, positionnement multidimensionel, projection aléatoire, mémoire distribuée, machine hétérogène, moteur d'exécution

Contents

1 Introduction

Points in large dimension spaces are often expected to live on an unknown small dimensional manifold, and the question is how to reveal it. Many methods have been proposed to do so such as Sammon mapping, curvilinear component analysis, stochastic neighbour embedding and t-SNE, isomap, Laplacian eigenmaps, just to present the diversity of available methods (here, nonlinear, with a global survey [1]). Those methods reach limits in time for, say, 10,000 items or more. An old classical method is the multidimensional scaling (MDS) [2, 3, 4], reviewed in [5]. From a numerical point of view, the MDS or SVD-MDS essentially resorts to processing the singular value decomposition (SVD) [6, 7] of an input matrix built from representing dissimilarities between pairs of items. When dealing with large data sets, an SVD may be out of reach due to memory or time to solution constraints, even when performed on a distributed-memory machine. For these reasons, large scale MDS is often processed with heuristic approaches, which may yield good results in practice, though no guarantee can be provided on their quality. A major step forward has been the design of randomized SVD (RSVD) [8, 9] algorithms in the 2000', a probabilistic, fast approach, ensuring the quality of the solution via random projections. Its usage within the MDS (RSVD-MDS) [10, 11, 12, 13], has allowed to process large data sets while preserving the numerical robustness [13] of the standard SVD-MDS.

The objective of this article is to design a high-performance distributed-memory RSVD-MDS for processing data sets of even larger size (up to one million items). We propose a task-based design of the whole RSVD-MDS algorithm and we implement it within an efficient software stack including state-of-the-art numerical solvers, runtime systems and communication layers. The outcome is the ability to efficiently apply robust MDS to large data sets on modern supercomputers. We assess the resulting algorithm and software stack to the point cloud visualization for analyzing distances between sequences in metabarcoding [14, 15, 16].

The rest of the article is organized as follows. In section 2, we review the numerical (MDS, SVD and RSVD) and computational (the task-based programming model as well as the runtime and communication layers) building blocks we rely on and present related work. We then present our contributions in section 3 consisting of the design of a fully task-based MDS and the fine tuning of the whole software stack to execute this algorithm. In section 4, we assess it in the context of an application to metabarcoding before presenting our conclusions and perspectives in section 5.

2 Background

2.1 Numerical components

2.1.1 MDS

Before presenting the method itself, we propose a short and partial early historical digression on the original motivations for the term *scaling* in MDS, as it may otherwise be ambiguous in a high performance context. Representing items onto a *linear continuum* (a one-dimensional space in modern terminology or simply a *scale*) may be loosely related to measure theory and thus to some extent for instance dated back to Ancient Greece when Archimedes tried to calculate the area of a circle. This idea has been pushed very far in the early twentieth century, in particular by the psychology community who developed advanced methods to design scales onto which positioning judgements (see *e.g.* [17, 18]) or other non trivial concepts. However, if such a traditional *scaling* method (design of scale and positioning of items onto it) was very elaborated in the one dimensional case (representing the information onto a linear continuum only, *i.e.* along one axis), it remained to design a robust process in the multidimensional case (representing the information along multiple well chosen axes). The popularization [19] in the main psychometric journal of the truncated SVD (TSVD) [20], a fundamental mathematical tool, opened the door for the robust extension of such scaling procedures to the multidimensional case. Built on top of the TSVD, the *muldimensional scaling* (MDS) method [2, 3] may be viewed at retrieving the most relevant possible multidimensional scale for a prescribed dimension. As it was immediately noted [19], when the matrix is (square and) symmetric (as it is the case in our context), the SVD and the eigenvalue value decomposition (EVD) coincide up to the sign of the eigenvalues. MDS may thus be equally viewed as based on TSVD or truncated EVD (TEVD). We employ the SVD/TSVD term and rely on it in this manuscript, following [10, 11, 12].

MDS has been continuously enriched since these very early developments and its modern treatment is only loosely related to this early vision. There exists many excellent textbooks presenting it such as [5] or [21, chap. 13]. A classical and rigorous reference with many results, their demonstration and history is [22]. We refer to this literature for a thorough presentation of the MDS and now only propose a short (but, we hope, progressive) introduction to the method, following [5, chap. 2]. Let $M = (E, d)$ be a discrete metric space of m points endowed with a distance d (in what follows, d can be a dissimilarity too). Let

$k_{\text{MDS}} \in \mathbb{N}$ be an integer. MDS [2, 3, 4] aims at building a map x:

$$i \in E \xrightarrow{x} x_i \in \mathbb{R}^{k_{\text{MDS}}} \tag{1}$$

such that the norms of the distances $\|x_i - x_j\|_2$ between points x_i and x_j in $\mathbb{R}^{k_{\text{MDS}}}$ are as close as possible to $d(i,j)$. The implicit assumption is that the distances $d(i,j)$ are known (they are the *input* data of the problem) and come from an unknown point cloud $X = (x_i)_i$ (the *output* data to be computed) in a Euclidean space. The objective is to recover X and approximate it as accurately as possible for a prescribed dimension k_{MDS}. In the following, we will denote by $D = (d_{ij})_{i,j}$ the $m \times m$ input pairwise distance matrix. Assuming the Euclidean space is endowed with a given inner product $\langle ., . \rangle$, we furthermore denote by $G = (g_{ij})_{i,j}$ the $m \times m$ matrix of inner products $g_{ij} = \langle x_i, x_j \rangle$, often referred to as the Gram matrix. The solution comes from the observation that the Gram matrix G can be built from the distance matrix D. Indeed, in a Euclidean geometry, G and D are related through the law of cosines by:

$$d_{ij}^2 = g_{ii} + g_{jj} - 2g_{ij}. \tag{2}$$

To obtain the g_{ij} coefficients of G from the d_{ij} coefficients of D, it remains to express the diagonal coefficients $(g_{ii})_i$ (and thus $(g_{jj})_j$) in terms of coefficients of D. To do so, we fist remark that two isometric point clouds cannot be discriminated through the distance between their points, inducing that X can be recovered up to an isometry only. In particular, without loss of generality, we may thus assume that the point cloud X is centered, which imposes:

$$\sum_j g_{ij} = \langle x_i, \sum_j x_j \rangle = 0, \tag{3}$$

i.e., the sum of each row of the Gram matrix is zero. Noting $d_{i+}^2 = \sum_j d_{ij}^2$, $d_{+j}^2 = \sum_i d_{ij}^2$ and $d_{++}^2 = \sum_{ij} d_{ij}^2$, we may then sum (2) over j. We obtain that $\sum_j d_{ij}^2 = \sum_j g_{ii} + \sum_j g_{jj} - 2\sum_j g_{ij}$, By definition of d_{i+}^2, by (3), and reminding that $\sum_j g_{jj}$ is the trace of matrix G (*i.e.* $\text{Tr}(G) = \sum_j g_{jj}$), it translates into $d_{i+}^2 = ng_{ii} + \text{Tr}(G)$, yielding $g_{ii} = \frac{1}{n}(d_{i+}^2 - \text{Tr}(G))$. To obtain an explicit expression for g_{ii}, it remains to obtain an expression of the trace of G, which can be done by summing (2) over both i and j, yielding $d_{++}^2 = 2n\,\text{Tr}(G)$. Finally, the Gram matrix G may therefore be written from the distance matrix D as follows:

$$g_{ij} = -\frac{1}{2}\left(d_{ij}^2 - \frac{1}{n}d_{i+}^2 - \frac{1}{n}d_{+j}^2 + \frac{1}{n^2}d_{++}^2\right). \tag{4}$$

This computation of G out of D through (4) is the first step of MDS (line 1 in Algorithm 1, referred to as `GRAM` in the rest of the paper). It remains to deduce $X \in \mathbb{R}^{m \times k_{\text{MDS}}}$ from G. First, we remind that, by its above definition, $G = XX^{\text{T}}$. Second, as G is symmetric, it admits a unitary diagonalization so that its EVD may writes $G = U\Lambda U^T$ where Λ is diagonal and U is unitary. In the case G is semi-definite positive, Λ has non-negative diagonal values and we may thus write $X = U\Lambda^{1/2}$. Otherwise, the most common heuristic consists in keeping only the k^+ non-negative eigenvalues Λ^+ and associated eigenvectors

U^+ and considering $X \simeq U^+\Lambda^{+1/2}$. This is in practice required to relax the hypothesis that D is a Euclidean distance matrix (and furthermore support the case where it is only a dissimilarity matrix, *i.e.* the triangular inequality does not need to hold). We follow this path except that, as mentioned above, we proceed through an SVD of G. To do so, one may notice that the EVD of G may also write $G = U|\Lambda|\text{sign}(\Lambda)V^T$, where $|\Lambda|$ is the diagonal matrix whose diagonal values are the absolute values of the eigenvalues and $\text{sign}(\Lambda)$ is the diagonal matrix whose diagonal values are the signs of the eigenvalues. If we note $\Sigma = |\Lambda|$ and $V = \text{sign}(\Lambda)U$, we observe that we also have $G = U\Sigma V^T$, which is an SVD of G (see below in section 2.1.2). Therefore, we compute the SVD $U\Sigma V^{\mathrm{T}} = G$ of G (line 2). We name this step `RSVD` as we proceed it with an RSVD (see below) and we will come back later on the details of the `CHECK` step (line 3). Once the SVD is obtained, following the above common filtering heuristic singular vectors such that $v = -u$ are ignored and clipped off from U (line 4, `Compute_X (1/2)`). Hence, we have $G \simeq U^+\Sigma^+U^{+\mathrm{T}}$, with $U^+ \in \mathbb{R}^{m\times k^+}$ and $\Sigma^+ \in \mathbb{R}^{k^+\times k^+}$ if k^+ eigenvalues of G are non negative. X can thus be computed as $X \simeq U^+\Sigma^{+1/2}$ (line 5, `Compute_X (2/2)`). Finally, the solution of MDS is provided by returning the information associated with the part $X_{k_{\text{MDS}}}$ of the map X associated with the k_{MDS} (assuming $k_{\text{MDS}} \leq k^+$, which is the case in practice as a reasonably small k_{MDS} is most often targeted) largest singular values (line 6).

Algorithm 1: MDS: $(X, \Sigma) = \text{MDS}(D, k_{\text{MDS}})$

Input: a distance matrix $D \in \mathbb{R}^{m\times m}$; a dimension $k_{\text{MDS}} \leq m$ // `READ_D`

Output: a set of points $X_{k_{\text{MDS}}}$, $\Sigma_{k_{\text{MDS}}}$ the largest singular values associated with positive eigenvalues // `WRITE_X`

1 Compute the Gram matrix of D: $G = \text{GRAM}(D)$ // `GRAM`

2 Compute the SVD of G: $U\Sigma V^{\mathrm{T}} = G$ // `RSVD`

3 Check $\tau = \frac{\|\Sigma\|_F}{\|G\|_F} \overset{?}{>} \tau_{\min} = 1.0 - \epsilon$ // `CHECK`

4 Compute U^+ and Σ^+ by discarding in U all columns u such that $v = -u$ and corresponding terms in Σ // `Compute_X (1/2)`

5 Compute $X = U^+\Sigma^{+1/2}$ // `Compute_X (2/2)`

6 **return** $X_{k_{\text{MDS}}}, \Sigma_{k_{\text{MDS}}}$, associated with the k_{MDS} largest singular values

2.1.2 SVD

The SVD [6, 7] of a real rectangular matrix $A \in \mathbb{R}^{m\times n}$ consists of the factorization of the form $A = U\Sigma V^T$ where $\Sigma \in \mathbb{R}^{m\times n}$ is a diagonal rectangular matrix whose diagonal entries $\sigma_i = \Sigma_{i,i}$ are non negative, ranged in decreasing order, and referred to as the singular values and $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal matrices representing the so-called left and right singular vectors, respectively. In practice, the SVD is stored in a compact form. Assuming A is of rank $r \leq \min(m, n)$, $\Sigma \in \mathbb{R}^{r\times r}$ is stored as a diagonal square matrix and $U \in \mathbb{R}^{m\times r}$ and $V \in \mathbb{R}^{n\times r}$ are stored as rectangular matrices with orthogonal columns. A major breakthrough in the numerical computation of the SVD has been proposed by Golub et al. [23, 24, 25]. We refer the reader to [26] for

details and to [27, 28] for an early history. The SVD plays a central role in data analysis because it can been shown that keeping the information related to the first k singular values, referred to as a truncated SVD (TSVD) [20] at rank k, consists of the best approximation in commonly employed norms such as 2-norm among all possible rank k approximations of A.

In our context, it has been shown [10, 11, 12] that we can rely on a much faster algorithm than a deterministic SVD, the SVD with random projection [8, 9]. In this algorithm, a deterministic SVD is still required but with much lower dimensions, such that it is not critical to have a fast distributed-memory deterministic SVD to ensure the efficiency of the overall algorithm. We refer the reader to [29] for a detailed review of HPC (deterministic) SVD developments and we instead focus here on the SVD with random projection, or, for short, randomized SVD (RSVD). We will employ it in the SVD step of our MDS and, for this reason, we will refer to the step associated with line 2 in Algorithm 1 as `RSVD` in the rest of the paper. Because the RSVD results in an approximated SVD, we assess its quality. This is why the MDS (Algorithm 1) is enriched with a `CHECK` step at line 3. This check consists in measuring the part τ of the information captured by Σ among G through their relative Frobenius norms ($\tau = \frac{\|\Sigma\|_F}{\|G\|_F}$) and verifying that it is large enough ($\tau > \tau_{\min} = 1.0 - \epsilon$, with $\epsilon = 10^{-3}$, being satisfying for the target test case, see section 4.1).

2.1.3 Randomized SVD

Let $A \in \mathbb{R}^{m \times n}$ be a matrix with $m \geq n$. The number of floating point operations (flop) of the SVD is $O(mn^2)$ and it becomes unaffordable for large values of m and n. Alternatively, randomized algorithms are very efficient algorithms with bounds on errors, to compute the first singular values and vectors in a reasonable amount of time and memory [8]. One of them is the recently developed RSVD [30, 31]. The main idea is to approximate the column space of the matrix A by only a small number of vectors through a linear combination of the columns (lines 1-2 in Algorithm 2, in the following referred to as `RAND` and `GE1`, respectively) and orthogonalize (line 3, `QR1` and `Q1`) them to obtain a basis (Q) of an approximation of the range of A. Then, the matrix A is projected onto this space (line 4, `GE2`) and factorized via a (deterministic) SVD (line 5). After this step, the right singular vectors V are obtained whereas it remains to explicitly form the left singular vectors U (line 6, `GE3`).

Algorithm 2: Random projection SVD: $(U, \Sigma, V) \simeq \text{RSVD}(A, k)$

Input: A a $m \times n$ matrix, k a prescribed rank
Output: an approximate factorization $A \simeq U\Sigma V^T$

1 Draw a $n \times k$ Gaussian random matrix Ω // `RAND`
2 Form the $m \times k$ matrix $Y = A\Omega$ // `GE1`
3 Compute the QR decomposition of Y: $QR = Y$ // `QR1, Q1`
4 Form the $n \times k$ matrix $C = A^T Q$ // `GE2`
5 Compute the SVD of C: $U_C \Sigma V_C^T = C$ // `QR-SVD`
6 Form the matrix $U = QV_C$ and note $V = U_C$ // `GE3`
7 **return** U, Σ, V

The procedure for approximating the range of A consists in considering Ω a random $n \times k$ matrix and performing the matrix-matrix product $Y = A\Omega$ (where Y is then a $m \times k$ matrix). There are several ways to choose Ω [8]. We here restrict ourselves to the Gaussian distribution, *i.e.* Ω is a random Gaussian matrix whose coefficients satisfy the standard normal distribution $\mathcal{N}(0, 1)$. Usually, for a good precision at rank k, it is advised to select Ω as $n \times k'$ with $k' = k + s$, where s is called the oversampling. Because taking s as low as $s = 5$ or $s = 10$ is often (and in particular in this study) sufficient [8], we do not distinguish k and k' in the rest of the article. We also assume $k \ll \min(m, n)$, which is also a reasonable practical hypothesis in general and in the particular metabarcoding case study we consider here (where, in addition $m = n$, as the RSVD is called on an input square matrix A consisting of the Gram matrix G of the MDS). A $m \times k$ matrix with such dimensions $k \ll m$ is often referred to as *tall and skinny* and this shape can be computationally exploited following Algorithm 3 to reduce the complexity of the SVD with respect to the original developments from [23, 24, 25]. The idea is to first compute the QR factorization (line 1 in Algorithm 3, referred to as `QR2` and `Q2` in the following) of the matrix whose SVD is sought. Doing so, it only remains to process a $k \times k$ matrix (R) with the standard (such as [25] or one of its recent derivatives [26]) SVD algorithm (line 2 in Algorithm 3, $k \times k$ `SVD`) before reconstructing the left singular vectors U (line 3, `GE4`). Such a QR-SVD approach was first proposed by Lawson and Hanson [32] and further analyzed by Chan [33]. The original motivation was the reduction of the computational complexity. In a distributed-memory context, it is also an opportunity for ensuring a separation of concerns: ensuring only a fast $m \times k$ distributed-memory QR factorization while relying on a sequential (or shared-memory) $k \times k$ standard SVD. We will employ such a QR-SVD algorithm to process the deterministic SVD step of the RSVD algorithm, this is why lines 5 in Algorithm 2 will be referred to as `QR-SVD` in the following.

Algorithm 3: QR-SVD algorithm: $(U, \Sigma, V) = \text{QR-SVD(A)}$

Input: A a $n \times k$ matrix
Output: a factorization $A \simeq U\Sigma V^T$

1 Compute the QR decomposition of A: $QR = A$ // `QR2, Q2`
2 Compute the SVD of the triangular matrix R: $U_R\Sigma V^T = R$ // $k \times k$ `SVD`
3 Form the matrix $U = QU_R$ // `GE4`
4 **return** U, Σ, V

The overall RSVD-MDS we propose to rely on thus consists of the MDS algorithm (Algorithm 1) for which the SVD (step 2) is processed with an RSVD (Algorithm 2) for which the input $m \times n$ matrix A is the $m \times m$ Gram G matrix of the MDS, in particular involving that the RSVD is processed on a square matrix ($m = n$). The internal deterministic SVD step within the RSVD (step 5 in Algorithm 2) is itself processed with a QR-SVD (Algorithm 3). We also highlight that, in addition to the prescribed rank k_{MDS} of the baseline MDS algorithm (Algorithm 1), the RSVD-MDS algorithm is also parameterized with the dimension k of the randomization and must satisfy $k_{MDS} \leq k^+ \leq k$, which we do positively check (see 4.3). The dimensions m and k drive the computational load of the RSVD and RSVD-MDS algorithms. On the other hand, the

k_{MDS} parameter is only used in the ultimate step of the MDS, essentially depends on what the MDS is used for (for instance, if the goal is to obtain a point cloud visualization, it corresponds to $k_{MDS} = 2$), and does not significantly impact the computational load. As a consequence, the performance study will be parameterized with m and k only.

It is to be noted that the randomization technique we employ breaks the symmetry when forming the approximate matrix QQ^TA. However, in our context, the QQ^TA approximation is an excellent approximation of A so that the symmetry is almost preserved and does not significantly impact the numerical result. We will check (and confirm) it in 4.3. Alternatively, a symmetry-preserving randomization technique [8] might be employed but that was not necessary in the present study and will not be further discussed in the paper.

2.2 Related work on distributed-memory RSVD and MDS

The term MDS has been ambivalent in the literature. In the present manuscript, it corresponds to "classical MDS", as pioneered by [4, 3]. In 1964, J. B. Kruskal issued a paper [34] with the same name, MDS, but another approach, even if related. The aim is to minimize on X a so-called stress function of the form $\Phi(X) = \sum_{i<j} \left(\|x_i - x_j\| - d_{ij}\right)^2$, where $(d_{ij})_{i,j}$ are dissimilarities, $(x_i)_i$ points in a Euclidean space, and X the set of points. It is not a linear algebra method, but a hard nonlinear optimisation problem. It is nowadays known as Least Square Scaling (LSS), to avoid confusion with classical MDS [5, 22, 21, 35].

To the best of our knowledge, no distributed-memory classical MDS had been proposed before this study. However, multiple propositions have been discussed for designing distributed-memory LSS [36, 37, 38, 39, 40]. They all rely on a full MPI [41] parallelization scheme. Zilinskas and Zilinskas designed the parallelization of a genetic LSS algorithm, assessed on up to 24 cores [36]. Pawlizeck and Dzwinel considered a heuristic based on particle dynamics simulation to find the minimum of the stress function. They validated their approach up to 144 cores [38]. The last three LSS parallelization schemes are based on a non trivial variant of the gradient descent algorithm. The method is named Scaling by MAjorizing a COmplicated Function (SMACOF) [42]. At each iteration of the SMACOF algorithm, the dominant part is a matrix-matrix product to update the points. The parallel design proposed by Bae was initially assessed up to 8 cores [37], and extended up to 256 cores [40, 39], obtaining an efficiency of 70% for dataset of size $100,000 \times 100,000$.

The RSVD itself, at the core of the present manuscript, has been parallelized for shared-memory [43] and GPU-accelerated [44, 45] single-node machines but, to the best of our knowledge, not for distributed-memory machines.

To the best of our knowledge, the present study is the first task-based RSVD (and MDS). It allowed us to perform a "classical MDS" at an unprecedented scale, in terms of both size (up to $1,000,000, \times 1,000,000$ distance matrices) and number of computational units (up to 2,400 CPU cores in the homogeneous case, and 640 CPU cores enhanced with 64 GPUs in the heterogeneous case).

2.3 Task-based programming

With the advent of supercomputers composed of a large number of multicore nodes enhanced with accelerators (such as GPUs) in the 2010's, the HPC community has faced a dilemma: shall numerical codes for scientific simulation and data analysis continue to be written with relatively low-level primitives for handling parallelism or be tackled with a higher level of abstraction? The former approach allows the programmer to directly specify all parallel directives to carefully exploit the whole potential of the hardware, but requires to develop and maintain complex codes combining multiple levels of parallel expressions such as message passing for inter-node communications, multithreading for exploiting multicore chips and vendor primitives such as `cuda` for exploiting GPUs. The latter approach, though more exploratory, has shown a few successful accomplishments during the last decade. In particular, the task-based programming paradigm has been shown to significantly reduce the difficulty of programming these complex machines, delegating the burden of handling data consistency or advanced scheduling strategies to third party software such as runtime systems.

One of the challenges of task-based programming for the developers of numerical libraries or applications is to deliver the same performance as competitors based on lower level parallel schemes, as part of the control has been delegated to another entity, the runtime system. That being said, because that entity is not aware of the numerical scheme, it may take care of successive numerical steps as if they were a single algorithm and pipeline these steps efficiently, avoiding unnecessary synchronization points.

In the last fifteen years, a new class of task-based runtime systems such as `starpu` [46], PaRSEC [47], SuperGlue [48], OmpSs [49], to name a few, has been proposed to better take advantage of multicore, manycore and heterogeneous accelerated architectures. From a programming point of view, many of those initiatives process a sequential series of tasks as the input algorithm. The tasks operate on data for which the programmer provides the data access mode (Read (R), Write (W) or Read-Write (RW)). Based on this information, dependencies between tasks may be inferred. The code may then be represented by a directed acyclic graph (DAG) where vertices are tasks and edges dependencies between them. This includes tasks for the computation parts of the application, but also the data input/output from/to the disk and from/to the network. This programming model is sometimes referred to as *Sequential Task Flow* (STF) [50, 51] and is supported by a large number of runtimes, including OpenMP since revision 4.0 (through the *task* construct and *depend* clause inspired from OmpSs), `starpu` with the default configuration and PaRSEC through its dynamic task discovery (DTD) mode. In addition to the ease of programming it provides (only a task-based sequential code is requested), the strength of this paradigm is that the runtime system has a complete and precise high-level view (the DAG of tasks) of the composition of the application. As a consequence, it can automatically delegate ready tasks to *workers* (typically threads associated with CPU cores or GPUs in the heterogeneous case) and perform many optimizations, including using advanced scheduling heuristics, ensuring data transfers to/from the disk and the network concurrently with computational tasks.

We have shown in [50] that the STF programming model can be efficiently

employed in a distributed-memory context to automatically manage the distribution of tasks over nodes of a cluster, by producing the corresponding data transfers over the network. The principle, further described in [50], is the following. The application instructs the runtime about the distribution of data to be used over the nodes of the cluster. For dense linear algebra this is typically a block-cyclic distribution for instance. This is named the *owner* node of a data. On each node, the application unrolls the whole task graph of the whole computation. On each node, the runtime system automatically determines which tasks of the graph should be performed by the node: if and only if it owns the data that is written to by the task. This is deterministic, and thus all nodes automatically agree on which of them will perform each task, without having to exchange any message. On each node, the runtime system automatically determines which network transfer should be performed: if the node executes the task, it has to receive the data it does not own from the nodes that own it; if the node does not execute the task, it has to send the task input data that it owns to the node that will execute the task. Additionally, the runtime system caches data to avoid multiple transfers of the same data. We will rely on this model to design our task-based RSVD-MDS.

Writing a task-based code in the STF model is twofold. First, the data structures must be designed and declared to the runtime system, we will refer data declared to the runtime to as *data handles*. Second, the sequence of tasks operating on those data handles must be written. Although the code is written sequentially, the challenge is to design data structures and algorithms with appropriate granularity (a too low granularity may cost overhead due to runtime management and a too coarse granularity may prevent parallelism) and concurrency opportunities.

2.4 Software

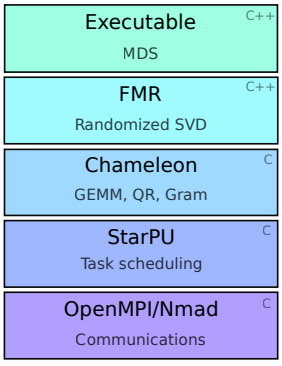


Figure 1: Software stack.

We propose to implement the above RSVD-MDS task-based algorithm within a software stack composed of five layers as depicted in Figure 1. We here present these five building blocks as they were at the beginning of the project (and will explain later in section 3 how they have been enriched and arranged as a software stack for the purpose of the study). The first top three layers are

mathematical software while the last two ones are runtime supports. At the top, the `mds` application loads input files from disk to main memory (distance matrix D in Algorithm 1) with which it computes the Gram matrix G. In the second layer, `fmr` implements the RSVD (Algorithm 2). The coupling of the `mds` application with `fmr` to implement an RSVD-MDS was pre-existent to the project [10, 11, 12]. Both the MDS application and `fmr` were relying on BLAS/LAPACK [52] dense linear algebra kernels and were hence limited to single-node machines. The goal of this study being to design a task-based RSVD-MDS for distributed-memory machines, we decided to rely on a state-of-the-art dense linear algebra task-based library. We chose the Chameleon solver [53, 54, 55, 50], our third layer, for that purpose. It is based on tile algorithms [56]. As many dense linear algebra algorithms, tile algorithms operate on matrices partitioned in square (or possibly rectangle) submatrices, often referred to as blocks in the literature. Contrary to most of their past literature, their main innovation is that these algorithms were thought and re-designed to ensure the highest possible level of pipelining of the computational tasks [56]. In this context, submatrices or blocks, are referred to as tiles. The `plasma` library [57, 58] was the first fully-featured dense linear algebra library following this tile design, with shared-memory multicore machines in mind. `chameleon` is an extension of `plasma` to heterogeneous [53, 54, 55], and distributed-memory [50] machines. The main idea is to benefit from the great concurrency opportunities of these algorithms while delegating the effective parallelization to a runtime system. As a consequence, `chameleon` may be viewed as a tile-based dense linear algebra library where *tiles* are declared to the runtime as *data handles* (see section 2.3). Though not fully-featured, `chameleon` provides a large subset of the BLAS/LAPACK standards. In particular, it provides multiple linear algebra routines required for designing our RSVD and RSVD-MDS on top of it, though it has been necessary to design or improve some of the routines, as detailed in section 3.

`chameleon` may be executed on multiple runtime systems (including `starpu`, ParSEC and OpenMP). We chose to focus on the usage of the `starpu` [46] back-end as we showed it can support the STF programming model in a scalable fashion [50]. The fourth layer is thus the `starpu` task-based runtime system. It manages the concurrency, schedules the tasks and handles the data consistency. In a distributed-memory setting, `starpu` delegates data transfers to a fifth layer, the communication engine. `starpu` supports two such engines being coupled either with MPI [41] or with `newmadeleine` [59, 60], a fully multithreaded high-performance communication library. Most experiments (see section 4.5 and section 4.6) will be carried with the MPI communication back-end using the OpenMPI implementation of the MPI standard whereas the scalability will be eventually further assessed using `newmadeleine` as a prospective (see section 4.7).

3 Contribution

The main contribution of this article is the design of a high-performance distributed-memory task-based RSVD (section 3.1) and RSVD-MDS (section 3.2) following the STF programming model [50] discussed in section 2.3. This design essentially consists in coupling the numerical steps of the `fmr` and `mds`

high-level codes (top two levels of the stack, see section 2.4 and Figure 1 in particular) with the task-based chameleon dense linear algebra library (third level of the stack). In addition, the complexity (section 3.3) of the proposed RSVD and RSVD-MDS, comforted by a preliminary performance analysis not reported here, motivated us to pay further attention to the matrix multiplication, the dominant numerical operation, leading us to refine the original chameleon baseline matrix multiplication to follow a SUMMA [61] communication pattern (section 3.4). While I/Os were a small part of the computation at the beginning of this project, the efficiency of our task-based RSVD and RSVD-MDS changed the game, the I/Os becoming a non negligible part of the overall execution time of the mds application. We have therefore proposed a task-based management of the I/Os (section 3.5) to further pipeline the overall workflow. All these algorithms have been implemented in a software stack (see Figure 1) crafted with care (section 3.6).

3.1 Task-based RSVD

We present the task-based design within fmr of the RSVD proposed in section 2.1, consisting in an RSVD (Algorithm 2) whose internal tall and skinny SVD (line 5 in Algorithm 2) is processed with a QR-SVD (Algorithm 3). The input matrix A of the RSVD (Algorithm 3) is assumed to be provided as a tile matrix (we remind that the concept of tile is presented in section 2.4). In the context of the RSVD-MDS, this input matrix will be the Gram matrix G and we will explain how to produce it in tile layout in section 3.2 and section 3.5. The mds and fmr codes, originally thought for dealing with BLAS/LAPACK matrices and routines on shared-memory architectures, have thus been fully re-designed to cope with tile matrices on distributed-memory machines. In addition, the chameleon library has been integrated to process linear algebra operations instead of BLAS/LAPACK. From a software engineering point of view, actually, the matrix format and operation handlers have been abstracted so that we can choose between the original centralized version using BLAS/LAPACK or the new distributed-memory tile layout using chameleon.

The `RAND` (line 1 in Algorithm 2) operation for generating the random matrix Ω has been implemented through a call to the `p?plrnt`[1] (non blocking, parallel) chameleon random matrix generator, similar to the one used by HPL, which generates the matrix in an embarrassingly parallel manner. The `GE1` (line 2 in Algorithm 2) matrix multiplication has been implemented through a call to the `p?gemm` general matrix multiplication routine of chameleon. The QR factorization (line 3) actually consists of two successive calls: (`QR1`) a call to the chameleon `p?geqrf` general QR factorization. Similarly to LAPACK `?geqrf`, it produces R as a formed matrix but Q is implicitly returned through its Householder reflectors. Therefore, we furthermore (`Q1`) perform a call to the chameleon `p?orgqr` orthogonal generation routine for explicitly forming the matrix Q. Once Q has been formed, the product $C = A^T Q$ (`GE2`, line 4) can be processed through a standard general matrix multiplication operation, hence once again

[1] chameleon nomenclature follows the BLAS/LAPACK/ScaLAPACK conventions, `p` standing for *parallel*, `?` being among `d` (double), `s` (single), `z` (double complex) or `c` (single complex) precision, `pl` standing for `plasma` (from which chameleon is inherited), `rn` for *random number (generator)*, and `t` specifying the tile layout.

through a chameleon p?gemm call[2]. The QR-SVD step (line 5) has been implemented internally within fmr, we describe it below. It explicitly retrieves V_C as a formed matrix, therefore the ultimate step $U = QV_C$ (GE3, line 6) is also processed with a standard general matrix multiplication through a chameleon p?gemm call.

The QR-SVD step (Algorithm 3, called at line 5 in Algorithm 2) has been implemented internally within fmr as follows. For the same dimension motivations as above, the QR decomposition (line 1 in Algorithm 3) is performed through two calls: (QR2) a call to chameleon p?geqrf for performing the QR factorization followed by (Q2) a call to the chameleon p?orgqr for explicitly forming matrix Q. The $k \times k$ R we obtain being of small dimension, it is centralized and processed with (a deterministic) LAPACK ?gesvd SVD call ($k \times k$ SVD, line 2). Q and U_R being formed at this stage, their final product $U = QU_R$ (GE4, line 3) is once again simply performed by a standard general matrix multiplication through a chameleon p?gemm call.

3.2 Task-based RSVD-MDS

We now present the task-based design within the mds application of the overall RSVD-MDS (Algorithm 1) proposed in section 2.1. The input distance matrix D of the MDS is assumed to be pre-computed (see section 4.1) and stored on disk in hdf5 format. Following a tile algorithm design, our task-based RSVD aims at reading it from disk and arranging it into a distributed-memory tile matrix (READ_D in Algorithm 1). Multiple strategies for doing so have been investigated and are discussed in section 3.5. For now, we simply assume that D has been read and arranged into a distributed-memory tile layout and focus on the numerical steps. Similarly to most distributed-memory dense linear algebra algorithms, the matrix is distributed according to a block-cyclic pattern (see *e.g.* Figure 3, right). The code is able to handle both 1D and 2D block-cyclic distributions.

The first numerical step of the MDS is the computation of the Gram matrix G from the distance matrix (GRAM, line 1 in Algorithm 1). As we had no task-based routine at our disposal, we implemented a task-based GRAM algorithm. From a data access point of view, GRAM is a reduction-like algorithm and is mainly communication-bound. A per-column reduction (the per-row reduction is avoided by symmetry) is performed to compute all the d^2_{+i} (and d^2_{i+} by symmetry) while a complete reduction is performed to compute d^2_{++}. We followed the design of the norm computation available in the chameleon library and illustrated in Figure 2. In the context of the Gram matrix computation, note that after the distributed row reduction (step 3 in Figure 2), all the d^2_{i+} values are computed in small vectors of the same size as the tiles. The reduction is then pursued up to the distributed column reduction (step 6 in Figure 2) in order to obtain the d^2_{++} value. Finally, an embarrassingly parallel update of

[2]It is to be noted that it would also be possible after the QR factorization (QR1) to directly apply Q (stored as Householder reflectors) to A^T through a p?ormqr call. However, in the BLAS/LAPACK/ScaLAPACK standard, such a call is performed *in-place*, overwriting A. In our context, A being the dominant matrix, this would induce a major memory and performance penalty. An option would be to consider a non standard *out-of-place* p?ormqr routine, but this is out of the scope of the present work.

each tile of the matrix is submitted. Each task computes the final g_{ij} values with the respective vectors from the Step 3 and the final value from the Step 6. This task-based GRAM algorithm has been incorporated within `chameleon` as a `p?gram` routine and the `GRAM` step of the `mds` application now consists in a call to that routine.

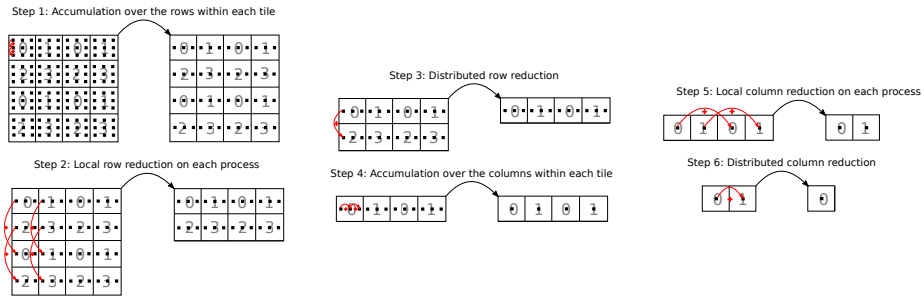


Figure 2: Task-based `GRAM` tile algorithm

The second numerical step (`RSVD`, line 2 in Algorithm 1) is the SVD, effectively computed with an RSVD in the considered RSVD-MDS algorithm. The `mds` application has been modified to call the `fmr` task-based RSVD routine presented earlier in section 3.1. Σ being diagonal, its Frobenius norm is immediate to compute and the third step (`CHECK`, line 3 in Algorithm 1) therefore essentially consists in computing the Frobenius norm of G. The `mds` application has been modified to call the `chameleon` `p?lange` norm computation routine to do so. The last two steps of the MDS consist in computing matrix X (`Compute_X`) through $X = U^+\Sigma^{+1/2}$ (line 5 in Algorithm 1) based on the filtered (line 4 in Algorithm 1) output (U^+, Σ^+) of the (R)SVD and scaling each column i of U^+ to compute $X \simeq U^+\Sigma^{+1/2}$. We have developed a task-based algorithm with a loop over tiles of U to copy the columns of interest into U^+. Then to compute the scaling we use an internal `chameleon` `map` facility to perform the multiplication of each column of U^+ by $\sigma^{+1/2}$. The `map` facility consists of a task-based loop over tiles offering the opportunity to apply an operator on each tile individually.

3.3 Complexity and key performance steps of the resulting RSVD and RSVD-MDS algorithms

The flop count of the RSVD (Algorithm 2) is dominated by `GE1` and `GE2` matrix multiplications and satisfies $\text{RSVD}\ (m,n,k) \sim_{m,n\to\infty} 4mnk$ (we refer appendix A for more details). In particular, we have $\text{RSVD}\ (m,n,k) = \mathcal{O}(mn)$ as $m,n \to \infty$. The RSVD step (called with $m = n$) dominating the RSVD-MDS (Algorithm 1), the flop count of the latter satisfies $\text{RSVD-MDS}\ (m,k) = \mathcal{O}(m^2)$ as $m \to \infty$. In a nutshell, the `GE1` and `GE2` general matrix multiplication steps are the dominant numerical operations of both the RSVD and the overall RSVD-MDS algorithms, when considering practical dimensions ($k \ll m = n$, see section 4.1). For these reasons, we have refined the original `chameleon` `p?gemm` algorithm as discussed below in section 3.4. Although less critical, it may be

noted that this outcome advanced routine will also benefit to other matrix multiplications steps (`GE3` and `GE4`).

Both QR factorization (`QR1` and `QR2`) and subsequent orthogonal generation (`Q1` and `Q2`) steps have a lower computational complexity than the `GE1` matrix dominant multiplication step. However, their tall and skinny shape had long made them challenging to process efficiently in parallel. A new numerical scheme [62, 63] has been proposed about fifteen years ago to reduce the number of communications when processing such matrices. It has then been demonstrated [64] that the original tile QR factorization [65] can cope with this scheme. `chameleon` is based on this advanced QR factorization scheme [54] and now includes the refinements of [66]. As a consequence, the available `p?geqrf` and `p?orgqr` routines already include state-of-the-art techniques for performing QR factorizations on tall and skinny matrices we target in the proposed RSVD algorithm and no improvement was needed for the purpose of the present study.

The other numerical steps do not represent as high challenges and we therefore do not discuss further their internal task-based design. It remains however to explain how to arrange the sequence of calls in a distributed framework, which is often a hard challenge to make without synchronization. In this work, as explained in section 2.3, we rely on the STF programming model. It allows for writing a sequential task-based algorithm and let the runtime system infer (and handle) the dependencies between them. As a consequence, thanks for this design, no synchronization is needed between each above discussed step. All the `p?xxxx` routines discussed above are non blocking, the runtime system being able to detect whether a particular task is ready to be triggered. As a consequence, the whole RSVD and RSVD-MDS algorithms are fully pipelined up to the actual numerical dependencies of the tasks. This opportunity provided by the STF programming model has already been discussed in the literature. However, the present study is, to the best of our knowledge, one of the pioneer to illustrate this potential for designing an application with as many steps and such a care in a distributed-memory framework.

3.4 Task-based SUMMA matrix multiplication

As discussed in section 3.3, in our context, the `GE1` and `GE2` general matrix matrix multiplication (GEMM) steps are the dominant numerical operation of both the RSVD and the RSVD-MDS algorithms. While a GEMM routine (`p?gemm`) was already available in `chameleon`, the associated communication pattern was very basic and preliminary experiments (not reported here) showed limited performance. We have therefore designed a task-based version of the SUMMA GEMM algorithm [61] for the purpose of the present study. More precisely, we implemented the pipeline version of SUMMA [61]. The main idea is to communicate both A and B matrices along a ring of communications to set up a pipeline of computation. This avoids a potential overload of the communication buses with a large number of broadcasts, and ensures that the local amount of temporary data is kept under a given limit controlled by the look-ahead parameter (number of rows/columns sent in advance). The task-based design is inspired from [67] which was designed over the PaRSEC runtime system [47] following a programming paradigm (sometimes referred to as parameterized task

graph (PTG)) where dependencies are expressed explicitly and communications can thus be tightly controlled. However, in the STF programming model we rely on in the present work, communications are induced and are not explicitly controlled by the programmer. Thus, to control the pipeline, we introduced temporary working matrices W_A and W_B – respectively associated with the A and B matrices to be multiplied – and whose size is proportional to the look-ahead parameter. The task flow is enriched with copy tasks. In the STF model, if the input and output data associated with such a copy are mapped on different processes, it automatically translates into a communication. In addition, we exploit the in-place reception capability of the `starpu` runtime system through MPI datatypes so that such a task copy is translated into a communication... without copy! This way we can indirectly (through the insertion of copy tasks) but precisely and without overhead (no actual copy is performed) control the communication pattern. We implemented the pipeline version of SUMMA doing so. For example, if we use a look-ahead of 1, the first column of A (row of B) is sent through a ring to all nodes of the same row (column) to W_A (W_B), and as long as all computations involving W_A are not finished, the second column (row) can not be received, thus creating a flow of communication and computation similar to the pipeline version of the SUMMA algorithm. This STF pipeline SUMMA algorithm has been implemented in `chameleon` and is employed for all GEMM calls (`GE1`, `GE2`, `GE3`, `GE4`). Note that we did not exploit the symmetry of A in the `GE1` and `GE2` steps, as, although it would save memory, the symmetric matrix multiplication (SYMM) imposes an additional challenge in terms of arithmetic intensity and we reserve it for future work.

3.5 Task-based management of the I/Os

The MDS is performed on a square input distance matrix D (see Algorithm 1) of order m, possibly very large (beyond 10^6 in the target test case of the present study). As further described in section 4.1, the matrix D is assumed to be stored in one or several `hdf5` files, which must be first read. Conversely, the MDS eventually computes an output point cloud X of size (m, k^+) and we save the k_{MDS} first columns ($X_{k_{\text{MDS}}}$) into an `hdf5` file. I/Os are thus important in the MDS workflow and we propose to handle them with a task-based management in order to ensure their efficient pipeline. For a matter of conciseness, and because D is a far larger matrix than X, and thus more impacting the overall performance, we restrict our discussion to the reading of the input distance matrix D (`READ_X`) and report to appendix B for the design of the writing of the output point cloud (`WRITE_X`). D is thus assumed to be initially stored in multiple `hdf5` files (*e.g.* 6 files in the example of Figure 3, left), representing sub-blocks D_{ij} of the global matrix. The files store the upper part of the global matrix, D being symmetric. These files must be read and arranged into a tile matrix in memory (Figure 3, right). We propose three methods aiming at reading the matrix in parallel. Preliminary tests (not reported here) showed a better performance of the overall RSVD-MDS when arranging data into a tile (column-wise) 1D block-cyclic distribution (Figure 3, right), which is due to the tall and skinny pattern of the matrices involved in the numerical key steps of the RSVD-MDS. Therefore, without loss of generality, we illustrate our discussion

with the particular case of an output 1D block-cyclic distribution, for all three methods (though the code also works in the 2D case). All three methods rely on the `hdf5` `h5dread` routine for reading (partial) dataset from an `hdf5` file and on the `chameleon` `map` facility (see section 3.2) for setting up the tiles of the matrix in memory. The performance of these three methods will be assessed in section 4.4.

3.5.1 Method 1: by-tile

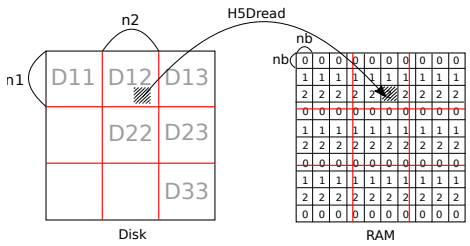


Figure 3: Reading the distance matrix from 6 files with the *by-tile* method using 3 MPI processes.

The first method consists in reading the data associated with the distance matrix from disk (Figure 3, left) *by-tile* and directly storing it in memory in the corresponding tile (Figure 3, right). The `h5dread` routine is thus called to read a subset of the data associated with a tile of size $nb \times nb$ (*e.g.* $nb = 320$). The algorithm is task-based, embarrassingly parallel, and allows one to load the matrix in memory in a distributed way and with all available workers.

Although appealing from a high-level point of view, this approach actually suffers from multiple drawbacks. First, although `hdf5` is thread-safe, it is not currently thread-efficient when multiple threads (and as mentioned in section 2.3, workers are managed with threads) tackle a single file – even if different parts of the file are tackled – due to the current internal `hdf5` design (through locks for accessing the global data structure). Second, *by-tile* read data accesses are not the most efficient ones. Third, concurrent accesses of multiple MPI processes on the same `hdf5` file may also lead to a reduced efficiency.

3.5.2 Method 2: by-tile + single-worker

A first refinement of the above `by-tile` method for alleviating the first drawback is to restrict the execution on one single worker per node. By doing so, we prevent workers to be busy waiting in `hdf5` internal locks and therefore let them available for other potential available tasks.

3.5.3 Method 3: hdf5-tailored

The third method in addition aims at tackling the above second and third drawbacks. The second drawback is due to the fact that the `hdf5` files we read are stored contiguously by full rows. The idea for alleviating it is to cope with

this disk storage arrangement when reading the data. We have subsequently developed a task-based algorithm for reading panels of rows in `hdf5` files. Panels are of size $nb \times n_l$, nb being the number of rows in a tile and n_l being the number of columns (*i.e.* size of rows) in the sub-part of the global matrix stored in the `hdf5` file $D_{.l}$. Because the splitting of the global matrix into several `hdf5` files does not match the splitting into tiles, we have to read small pieces of data into the neighbouring file (storing the next data in the line of the global matrix) to extend the panel so that it is constituted of complete tiles. This is done so as to simplify the transformation of panels into tiles in the subsequent step.

The third drawback being that the `hdf5` performance is reduced when multiple MPI processes access the same file, the algorithm reads large blocks of rows following a simple 1D block-non-cyclic distribution, *i.e.* splitting the set of `hdf5` files as blocks of rows allocated in a balanced way over MPI processes (see Figure 4). This way, there are far fewer calls to the `hdf5 h5dread` read routine, done on much larger data blocks, accessed on a per-row basis following the `hdf5` storage, and with a limited number of MPI processes handling each file. Splitting the matrix into panels allows one to exhibit pipelining since, once the first panel is read, the following tasks performing data copies from panel to tile can be scheduled directly without waiting for the whole matrix to be read.

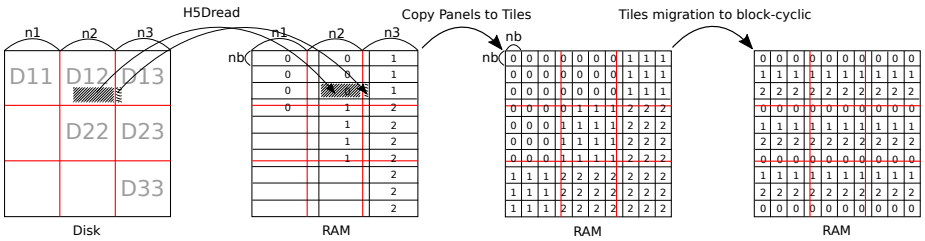


Figure 4: Reading the distance matrix from 6 files with the *hdf5-tailored* method using 3 MPI processes.

Now that we have optimized the way we read data from `hdf5` files, it remains to set up the tile matrix distributed in a 1D block-cyclic fashion. This is done by first building a tile matrix, following the same 1D block-no-cyclic distribution used for reading, using the `chameleon map` function to copy the panels into tiles (step "Copy Panels to Tiles" in Figure 4). The ultimate step consists in performing a tile shuffle by transferring the tiles to the appropriate MPI process in order to obtain the target block-cyclic distribution (step "Tiles migration to block-cyclic" in 4). The whole algorithm is task-based and without synchronization between the tasks that read panels, the ones that copy panels into tiles and those migrating tiles. As a consequence, it is not necessary to fully host the matrix twice in main memory, the buffer for reading a panel being a temporary workspace whose maximum size is under prescribed control.

3.6 A software stack crafted with care

Beyond the above discussed careful task-based numerical and I/O design, the software stack has been further tuned to deliver high performance and maintain memory usage under control. In section 2.3, we summarized how task-based runtime systems can cope with a very dynamic availability of tasks. This allows us to fully pipeline all the steps of the RSVD and of the RSVD-MDS, including the overlapping of the I/O with computation. The general-purpose heuristics of the `starpu` runtime [46] also already cope well with the RSVD and RSVD-MDS task graphs. We have however tuned two parameters.

First, the application does not need to expose the whole task graph to the `starpu` runtime, it only needs to expose enough tasks for enough parallelism to be available to exploit all the processing units of the target system. On the contrary actually, we had to limit the task graph exposure to the `starpu` runtime, to avoid overflowing the memory of the target system with temporary buffers. We thus used task submission throttling [68] to control how much of the task graph is exposed to the runtime. The `STARPU_LIMIT_MAX_SUBMITTED_TASKS` environment variable allows to specify the maximum number of tasks submitted to the runtime. Beyond this value, the task submission function blocks. The `STARPU_LIMIT_MIN_SUBMITTED_TASKS` environment variable allows to specify the minimum number of tasks submitted to the runtime. Below this value, the task submission function unblocks. The application thus alternates between submitting a series of tasks and blocking, waiting for some tasks to complete before submitting more tasks. With a large-enough `STARPU_LIMIT_MIN_SUBMITTED_TASKS` value, the system does not run out of parallelism, and with a small-enough `STARPU_LIMIT_MAX_SUBMITTED_TASKS` value, the system does not overflow its memory. In the MDS case, we configured the values according to the parallelism induced by the SUMMA algorithm.

Second, we tuned the usage of cores on the system. When using an MPI library for inter-node communications, `starpu` automatically reserves a whole CPU core for the MPI communications. It is known [69] that communication only makes progress when MPI functions are called, thus dedicating a whole thread to this purpose ensures that MPI communications complete asynchronously. In the `newmadeleine` case however, `starpu` does not dedicate a thread for communications but instead leaves a core empty so that `newmadeleine` may use it for its progression engine [70]. The mere submission of tasks by itself takes a significant amount of CPU time. We have thus reserved another CPU core to this end, with the `STARPU_RESERVE_NCPU` environment variable. Eventually, we made sure that the Intel MKL library does not introduce spurious core binding, so that the `starpu` runtime can properly control how CPU cores are used.

4 Experimental study

We now present an experimental study on the behaviour of the proposed task-based RSVD and RSVD-MDS. We first present the numerical set up that motivated this work in section 4.1 as well as the hardware and software set up in section 4.2. We then assess the capability of our algorithms and software

stack to process the target problem (*Lall* test case, $m = 1,043,192$, $k = 1,000$). We then present a performance study. Section 4.4 first assesses the I/O strategies proposed in section 3.5, motivating to rely on the most advanced scheme (*hdf5-tailored*). We then present an overall performance study of the RSVD and RSVD-MDS (with the *hdf5-tailored* scheme for the I/Os) in section 4.5. Section 4.6 and section 4.7 eventually show the versatility of the designed software stack to take advantage of heterogeneous machines and employ various communication back-ends (MPI and `newmadeleine`), respectively, illustrating the strength of the task-based programming model in abstracting the architecture and the execution model while ensuring performance portability.

4.1 Numerical set up

In the following, the MDS is performed on matrices storing the genetic distances of Diatoms collected in Geneva. The structure of the distance array between sequences in an environmental sample reflects its diversity. Here, we have tried to decipher the diversity of ten related environmental samples by associating to them a point cloud in a low dimensional Euclidean space with MDS, where each point is a sequence, and the distance between two points is as close as possible to the genetic distance between the sequences. The dataset used as input for the MDS is a $10^6 \times 10^6$ matrix of distances between sequences, provided by Inrae BioGeCo, split into 55 smaller blocks. It comes from 10 environmental samples, denoted by L_t for $t \in \{1, \ldots, 10\}$, which have been collected in Geneva lake by UMR Carrtel at Inrae Thonon, at about monthly intervals at times $t = 1, \ldots, 10$ between April 2012 and March 2013. Their objective was to investigate seasonal dynamics. All pairwise distance matrices $D_{tt'}$ for $1 \leq t < t' \leq 10$ have been computed (45 matrices, available in `hdf5` format). The sizes of diagonal blocks vary between 7.0×10^4 and 1.4×10^5 rows. Diagonal blocks are square, and $D_{tt}[i, j]$ in diagonal block t is the distance between read i and j in sample L_t. In off-diagonal blocks (t, t'), $D_{tt'}[i, j]$ is the distance between read i in L_t and read j in $L_{t'}$. The total number of reads (hence the order of the assembled matrix) exceeds one million. The molecular biology protocol for extracting DNA, amplifying marker rbcL, sequencing, is presented in [16, 71] and has been implemented by Inrae UMR Carrtel at Thonon and PGTB Platform at Bordeaux, providing over one million reads in total. Distances between the reads have been then computed with Smith-Waterman algorithm [72, 73] as part of e-biothon project [74], which provided a use of massive parallelization for biodiversity studies. Pairwise distances have been computed with C/MPI program `MPI-disseq` from BioGeCo. These pre-calculated resulting distances constitute the input distance matrix for the present study (input matrix D in Algorithm 1). We may consider either part of the data (leading to a matrix of reduced dimension) of the whole data set (Lall). Table 1 shows the different samples we have considered and the associated matrix orders (m). While the software stack works both in single (32 bits) or double (64 bits) precision, all the tests we report on here are conducted in single precision, as it is enough in practice to obtain results of wished quality (which is confirmed in section 4.3).

Table 1: Samples considered in the study and associated distance matrix order m. For instance, *Leven* distance matrix of order 616,644 consists of the assembly of samples L2, L4, L6, L8 and L10 (hence 15 files, 5 for diagonal blocks and 10 for off-diagonal ones).

Name	m
L6	99,594
L2L3L6	270,983
L1L3L5L7L9 (Lodd)	426,548
L2L4L6L8L10 (Leven)	616,644
L1L2L3L4L5L6L7L8L9L10 (Lall)	1,043,192

4.2 Hardware and software set up

Three hardware configurations have been considered for the experiments. One homogeneous machine with a partition made of Intel Haswell nodes with 128 GB of RAM used for section 4.4 and section 4.5. For the experiments with `newmadeleine`, see section 4.7, the Haswell partition had been dismantled in between, hence we used another partition of the same machine, with Intel Broadwell nodes with 64 GB of RAM. For the tests with GPUs, see section 4.6, we have used an heterogeneous machine, with Intel Cascade Lake nodes associated with Nvidia Tesla V100 GPUs.

4.2.1 HSW24 and BDW28 homogeneous partitions

The first system considered is the Occigen homogeneous supercomputer, a Bull B720 machine on a partition composed of 2,106 Haswell E5-2690V3@2.6 GHz (24 cores per node) CPU nodes, equipped with 128 GB of RAM per node, an Infiniband FDR interconnect (56 Gb/s), and a Lustre file system with 5 PB of usable space and a maximum bandwidth that exceeds 105 GB/s. This partition will be referred to as **HSW24** and used for the numerical assessment in section 4.3, the study of the I/O strategies in section 4.4, and the overall RSVD-MDS evaluation in section 4.5. We also consider another partition of the same machine, composed of Broadwell E5-2690 V4@2.6GHz (28 cores per node) CPU nodes, equipped with 64 GB of RAM per node (and the same network and file system). This partition will be referred to as **BDW28** and employed for the prospective study of communication back-ends in section 4.7.

4.2.2 CAS40+V100x4 heterogeneous machine

The Jean Zay supercomputer is considered for assessing the software stack in the heterogeneous case. It is a HPE SGI 8600 machine composed of 261 nodes, each having two Intel Cascade Lake 6248 (20 cores at 2.5 GHz per processor) CPUs (*i.e.* 40 cores per node) with 192 GB of memory and enhanced with four Nvidia Tesla V100 SXM2 GPUs (32 GB). The network is an Omni-Path 100 Gb/s interconnect and the I/Os rely on an IBM Spectrum Scale parallel file system (ex-GPFS). This machine will be referred to as **CAS40+V100x4** and employed in section 4.6.

4.2.3 Software

We use the latest stable versions of the software stack with the `mds` and `fmr` master branches at commit b724117e and 5a3fa5c3 respectively, `chameleon` v1.1.0, `starpu` v1.3.8, `hdf5` v1.12.1, OpenMPI v4.1.1, `newmadeleine` branch master at commit 615a8334, UCX v1.11.0, Hwloc v2.4.1, Intel MKL v2019.5.281 on the homogeneous HSW24 and BDW28 partitions and v2019.4.243 on the heterogeneous CAS40+V100x4 machine. CUDA/cuBLAS v10.1.2 were used on CAS40+V100x4.

4.3 Numerical assessment

The first result is the capability of the proposed task-based design and software stack to process the target problem (*Lall* test case, $m = 1,043,192$, $k = 1,000$). The execution completed, being able to produce X (and write it to disk. We could also assess that the RSVD was successful through the CHECK step (line 3 in Algorithm 1) with $\epsilon = 10^{-3}$, obtaining $\tau = \frac{\|\Sigma\|_F}{\|G\|_F} = 0.99994 > \tau_{\text{MIN}} = 1.0 - 10^{-3}$. Additional checks were performed and were all successful (we do not further report on them but, for instance, on the same target test case with $k = 10,000$, we did successfully capture more information, obtaining $\tau = 0.99999$). As discussed in 2.1.1, another potential error to be assessed is the impact of the non-symmetric randomization process we have employed, which breaks the assumption that left (U) and right (V) singular vectors are equal ($U = V$) in the $G = U\Sigma V^T$ SVD decomposition, as it does not ensure anymore that the SVD and the EVD of the QQ^TA approximation coincide up to the sign of the eigenvalues. We therefore check how $U^+\Sigma^+U^{+\text{T}} = XX^{\text{T}}$ departs from $U^+\Sigma^+V^{+\text{T}}$. For that, we assessed $\frac{\|U^+\Sigma^+V^{+\text{T}} - XX^T\|_F}{m\|U^+\Sigma^+V^{+\text{T}}\|_F}$ and we observed that it was systematically lower than 2×10^{-7}. The third validation we ensured is the visualization of the computed point cloud X, in which case we have to restrict to a 2D representation ($k_{MDS} = 2$). Figure 5 shows the point cloud associated to the *Lall* test case after conserving the dimensions associated with the 2 dominant singular values ($k_{MDS} = 2$, after a processing of the RSVD-MDS at rank $k = 1,000$). We obtained $k^+ = 583$ for the *Lall* dataset and a further analysis of the outcome of the MDS for $k_{MDS} \in [2, k^+ = 583]$ is thus now possible but out of the scope of the present study and we reserve it for future work.

4.4 Performance analysis of the I/Os

We now assess the performance of the set up of the input distance matrix D in memory from the `hdf5` files (`READ_D` step of the MDS). Table 2 shows the performance of the three I/O strategies proposed in section 3.5 on the homogeneous machine HSW24 with tiles of size $nb = 320$. The *by-tile + single-worker* method ensures a performance gain between 15% and 50% compared to the baseline *by-tile* method where all workers perform I/O requests. It is is even the optimum method (358 s) in the single node case (assessed on L6). However none of these strategies scale in a distributed memory context. On the contrary,

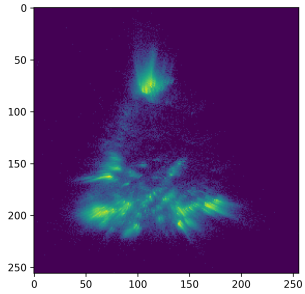


Figure 5: Lall test case ($m = 1,043,192$, $k = 1,000$) point cloud (first two axis).

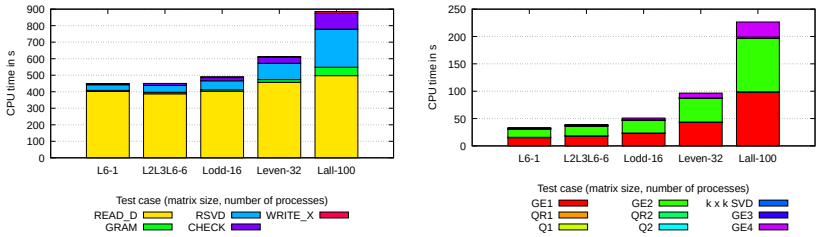


Figure 6: Execution time (s) of the overall RSVD-MDS (left) and focus on the RSVD (right) on the homogeneous machine HSW24, $nb = 320$, $k = 1,000$. Five test cases assessed, ranging from *L6* on 1 node to the *Lall* on 100 nodes.

the *hdf5-tailored* strategy, which accesses the data as they are stored on disk (by row) and (as much as possible) tackles different files with different processes, ensures a much better scaling. For instance, on the target test case (*Lall*) on 100 nodes, the *hdf5-tailored* improves by a factor 7.2 and 8.3 over the *by-tile* and *by-tile + single-worker* strategies, respectively.

Table 2: Time (in s) of the `READ_D` step (homogeneous machine HSW24, $nb = 320$).

Test case	order m	#nodes	by-tile	by-tile + single-worker	hdf5-tailored
L6	99,594	1	640	358	403
L2L3L6	270,983	6	1,297	973	399
Lodd	426,548	16	2,029	1,482	364
Leven	616,644	32	2,938	2,452	435
Lall	1,043,192	100	4,128	3,595	497

4.5 Overall performance analysis of the RSVD-MDS on an homogeneous machine

Figure 6 shows the overall performance of the RSVD-MDS (left), including I/O steps, and of the RSVD (right) in particular, for problems of increasing sizes (m)

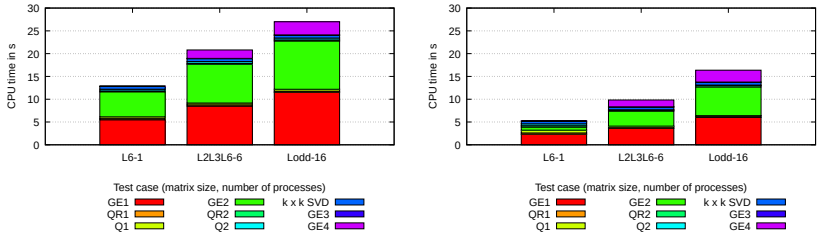


Figure 7: Execution time (s) of the RSVD on the heterogeneous machine CAS40+V100×4 with GPU accelerators turned off (left) or on (right), $nb = 1,000$, $k = 1,000$. First three test cases assessed, ranging from *L6* on 1 node to the *Lodd* on 16 nodes.

associated with the test cases from Table 1 (from *L1* to *Lall*), on an increasing number of processes (ranging from 1 to 100), employing an RSVD of fixed prescribed rank $k = 1,000$. We remind, as discussed in section 3.3, that the computational complexity of both the RSVD-MDS and RSVD is proportional to m^2, the number of coefficients of the input distance matrix D. These results may thus be viewed as a weak scaling study: when the matrix order m is 10 times larger, the number of resources (nodes) being roughly 100 times larger. The main observation is that the overall RSVD-MDS scales fairly well, the overall execution time increasing by factor 2 only on a 100 hundred times more complex problem. This is the case both for the I/O steps (following the *hdf5-tailored* strategy) and the numerical steps, dominated by the RSVD. Finally, we are able to solve a one million problem in less than 900 seconds.

Focusing on the RSVD (Figure 6, right), we furthermore observe, as expected from the discussion of section 3.3, that the dominant steps are effectively the `GE1` and `GE2` matrix multiplication steps (roughly 100 seconds). Their scalable design (section 3.4) ensures the scalability of the whole RSVD. In addition we may observe that, thanks to their advanced design (see section 3.1), the QR factorizations (`QR1` and `QR2`) and associated construction of Q (`Q1` and `Q2`) do manage to parallelize well enough so that their impact on the RSVD remains negligible at scale (roughly 1 second), in spite of the challenging pattern of the tall and skinny matrices they are applied on. Finally, we remind that the use of a QR-SVD algorithm for processing the internal deterministic SVD step of the RSVD was motivated by the wish of ensuring a separation of concern, relying on a fast $m \times k$ distributed-memory QR factorization and relying on a centralized small $k \times k$ standard SVD expecting that it is small enough not to prevent the algorithm to scale (see section 2.1.3). The results do confirm this wish: as just discussed, `QR2` and `Q2` scale well enough to remain a small proportion of the overall time, and, the use of a $k \times k$ `SVD` (less than 1 second) does not penalize the overall execution time.

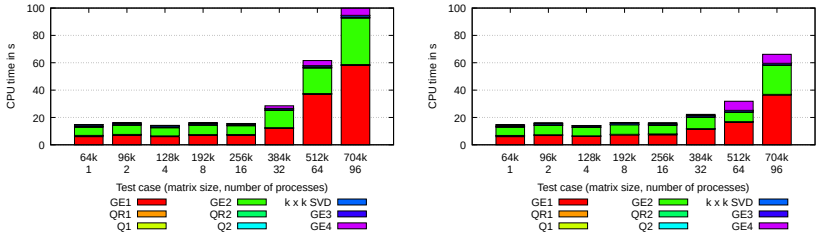


Figure 8: Execution time (s) of the RSVD with OpenMPI (left) and New-Madeleine (right), homogeneous partition BDW28, $nb = 320$, $k = 1,000$. Matrices of order 64,000 to 704,000 on 1 to 96 nodes, respectively (weak scaling).

4.6 Performance analysis of the RSVD on an heterogeneous machine

One of the strength of task-based programming is that by allowing for an abstraction of the architecture, the paradigm allows for a portable design. In particular, the software stack could be successfully executed on an heterogeneous machine without changing the code. We have performed the MDS for the first three test cases of Table 1 (from *L1* to *Lodd*) on the CAS40+V100x4 machine. We however report on the performance of the RSVD (Figure 7) only as I/Os, which dominate the rest of the RSVD-MDS execution time, are orthogonal to the potential use of GPU accelerators. The results show that we do successfully benefit from the GPU usage to speed-up the overall performance of the RSVD (16 seconds instead of 27 without GPUs on *Lodd*), in particular due to the improved performance of the dominant matrix multiplication steps `GE1` and `GE2` (6 seconds instead of 11 without GPUs on *Lodd*).

4.7 Study of the impact of the communication back-end on the performance of the RSVD

We now illustrate an other interesting feature of task-based programming in terms of versatility. Because the numerical algorithm is encoded at a high-level of abstraction as a DAG of tasks, the runtime has the freedom to employ various communication engines without requiring to change anything in the numerical code. As a result, our software stack can be executed either with an MPI back-end (with the possibility to rely on a library extremely well tested such as OpenMPI or any other MPI implementation), or, with a more original back-end, such as the research-oriented `newmadeleine` project discussed in section 2.3. As the communication layer is orthogonal to the I/O one, we again focus on the RSVD only. Figure 8 shows that the `newmadeleine` back-end delivers a more sustainable weak scaling (with about 35% of improvement on the GEMM and QR steps). Though a detailed analysis is out of the scope of this paper (we refer to appendix C), this result illustrates that by coupling the task-based runtime with a light-weight communication back-end, performance gains can be obtained, while not requiring to change anything in the numerical part (here, the top three layers) of the software stack.

5 Conclusion

The main contribution of this article is the design of a task-based RSVD and RSVD-MDS. The RSVD has been implemented within `fmr` on top of the task-based `chameleon` existing `p?plrnt` (`RAND`) (`plasma-inherited`) tile random generator, `p?geqrf` (`QR1`, `QR2`) QR factorization, `p?orgqr` orthogonal generation (`Q1`, `Q2`) and of the new `p?gemm` SUMMA general matrix multiplication (`GE1`, `GE2`, `GE3`, `GE4`). The usage of a QR-SVD for processing the internal deterministic SVD allowed us to simply rely on a centralized SVD kernel ($k \times k$ `SVD`) without a prohibitive performance overhead. The RSVD-MDS has been implemented within the `mds` application on top of this RSVD and of new task-based numerical (`GRAM`, `COMPUTE_X`) and I/O (`READ_D`, `WRITE_X`) algorithms (the `CHECK` step being essentially designed on top of the existing task-based norm computation). The result is a fully pipelined RSVD and RSVD-MDS for (homogeneous and heterogeneous) modern supercomputers. The RSVD-MDS method being a fast SVD-MDS, the outcome is the unique (to the best of our knowledge) capability of processing large distance matrices with a robust MDS. We were able to process with success a distance matrix D of order $m = 1,043,192$, using a random projection of rank $k = 1,000$, which are dimensions of great practical interest for the metabarcoding community, and, so far, out-of-reach (to the best of our knowledge) with a robust numerical MDS.

This work also pioneers – to the best of our knowledge – for illustrating the potential of the STF programming model for carefully and completely pipelining an application with as many steps in a distributed-memory framework. We hope that it will attract the attention of the high performance, scientific computing community for further considering task-based programming in general and the STF model in particular.

The dimensions of the RSVD and RSVD-MDS make them mostly driven by the performance of the matrix multiplication. We have thus designed a state-of-the-art distributed-memory SUMMA matrix multiplication to ensure their scalability. However, the proposed design has been by means of relatively low-level techniques, consisting in forcing communication patterns by expressing copies in the task flow. As a consequence, the elegance of the STF model is slightly reduced and the code looses part of its versatility. In particular, only a C-stationary variant has been implemented while a A-stationary variant should further improve the performance in the 2D case. We plan to rely on a more versatile matrix multiplication based on an extension of the STF programming model to express scalable communication patterns in a high-level expression [75] (which takes further advantage of `newmadeleine` capabilities). In addition, we plan to tackle the issue of exploiting symmetry in the matrix multiplication while maintaining an arithmetic intensity as high as in the non symmetric case.

The MDS is one method among others of linear dimension reduction [21, 76]. All of them rely on a best low-rank approximation of a given matrix, which can be achieved by SVD. The other operations are pre-processing (such as column-wise centering and scaling in Principal Component Analysis, which has roughly speaking the same complexity as computing the Gram matrix), and post-processing (such as the computation of coordinates, often a matrix-matrix product). The top layer of the software stack presented in Figure 1 could be

enriched beyond MDS by the family of linear dimension reduction methods for which it is relevant. This is an on-going project named `diodon`.

6 Acknowledgement

This work has been supported by the Région Nouvelle-Aquitaine, under grant 2018-1R50119 HPC scalable ecosystem. This work has been supported by the Inria Gordon ADT project. The computation of distances has been done at IDRIS on a Blue Gene Q, as part of DARI project *Biodiversiton* i2015037360 given to AF. This work was granted access to the HPC resources of Cines under the allocation 2021- A0100601567 attributed by GENCI). JMF and AF acknowledge the help of Sylvie Thérond at IDRIS for parallelizing home made code `disseq` for computing distances and developing `mpi-disseq`. Sequencing has been performed at the Genome Transcriptome Facility of Bordeaux (grants from the Conseil Régional d'Aquitaine n°20030304002FA and 20040305003FA, from the European Union FEDER n°2003227 and from Investissements d'Avenir ANR-10-EQPX-16-01) and with support of ONEMA project on Mayotte. Humid lab (DNA extraction, PCR) have been done at UMR Carrtel, under guidance of Agnès Bouchez, with support of ONEMA project Mayotte.

References

[1] John A Lee and Michel Verleysen. *Nonlinear dimensionality reduction*, volume 1. Springer, 2007.

[2] Marion Webster Richardson. Multidimensional psychophysics. *Psychological Bulletin*, 35:659–660, 1938.

[3] Gale Young and Aiston S Householder. Discussion of a set of points in terms of their mutual distances. *Psychometrika*, 3(1):19–22, 1938.

[4] W. S. Torgerson. Multidimensional Scaling: I. Theory and Method. *Psychometrika*, 17(4):401–419, 1952.

[5] T.F. Cox and M. A. A. Cox. *Multidimensional Scaling - Second edition*, volume 88 of *Monographs on Statistics and Applied Probability*. Chapman & al., 2001.

[6] E Beltrami. *Giornale di Matematiche ad Uso degli Studenti Delle Universita*, 1873.

[7] Camille Jordan. Mémoire sur les formes bilinéaires. *Journal de mathématiques pures et appliquées*, 19:35–54, 1874.

[8] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[9] Vahid Dehdari and Clayton V Deutsch. Applications of randomized methods for decomposing and simulating from large covariance matrices. In *Geostatistics Oslo 2012*, pages 15–26. Springer, 2012.

[10] Pierre Blanchard, Philippe Chaumeil, Jean-Marc Frigerio, Frédéric Rimet, Franck Salin, Sylvie Thérond, Olivier Coulaud, and Alain Franc. A geometric view of biodiversity: scaling to metagenomics. *arXiv preprint arXiv:1803.02272*, 2018.

[11] Pierre Blanchard, Olivier Coulaud, Eric Darve, and Alain Franc. Fmr: Fast randomized algorithms for covariance matrix computations. In *Platform for Advanced Scientific Computing (PASC)*, 2016.

[12] Pierre Blanchard. *Fast hierarchical algorithms for the low-rank approximation of matrices with applications to materials physics, geostatistics and data analysis*. Theses, Université de Bordeaux, February 2017.

[13] Emmanuel Paradis. Multidimensional scaling with very large datasets. *Journal of Computational and Graphical Statistics*, 27(4):935–939, 2018.

[14] H. M. Bik, D. L. Porazinska, S. Creer, J. G. Caporaso, R. Knight, and W. K. Thomas. Sequencing our way towards understanding global eukaryotic biodiversity. *Trends in Ecology and Evolution,*, 27:233–243, 2012.

[15] P. Taberlet, E. Coissac, F. Pompanon, C. Brochman, and E. Willerslev. Towards next-generation biodiversity assessment using DNA metabarcoding. *Molecular Ecology*, 21:2045–2050, 2012.

[16] L. Kermarrec, A. Franc, F. Rimet, P. Chaumeil, J.-F. Humbert, and A. Bouchez. Next-generation sequencing to inventory taxonomic diversity in eukaryotic communities: a test for freshwater diatoms. *Molecular Ecology Resources*, 13:607–619, 2013.

[17] Louis L Thurstone. Psychophysical analysis. *The American journal of psychology*, 38(3):368–389, 1927.

[18] Louis Leon Thurstone. Theory of attitude measurement. *Psychological review*, 36(3):222, 1929.

[19] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[20] Erhard Schmidt. Zur theorie der linearen und nichtlinearen integralgleichungen. iii. teil. *Mathematische Annalen*, 65(3):370–399, 1908.

[21] A. J. Izenman. *Modern Multivariate Statistical Techniques*. Springer, NY, 2008.

[22] K. V. Mardia, J.T. Kent, and J. M. Bibby. *Multivariate Analysis*. Probability and Mathematical Statistics. Academic Press, 1979.

[23] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[24] Gene Howard Golub. Least squares, singular values and matrix approximations. *Aplikace matematiky*, 13(1):44–51, 1968.

[25] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In *Linear algebra*, pages 134–151. Springer, 1971.

[26] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU press, 2013.

[27] Gilbert W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4):551–566, 1993.

[28] Roger A Horn and Charles R Johnson. *Matrix analysis*. Cambridge university press, 2012.

[29] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale. *SIAM review*, 60(4):808–865, 2018.

[30] Vladimir Rokhlin, Arthur Szlam, and Mark Tygert. A Randomized Algorithm for Principal Component Analysis. *SIAM Journal on Matrix Analysis and Applications*, (3):1100–1124, jan.

[31] Martinsson, Per Gunnar, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, (1):47–68.

[32] Charles L Lawson and Richard J Hanson. Solving least squares problems. *Prentice-Hall Series in Automatic Computation*, 1974.

[33] Tony F Chan. An improved algorithm for computing the singular value decomposition. *ACM Transactions on Mathematical Software*, 8(1):72–83, 1982.

[34] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[35] Alain Franc, Pierre Blanchard, and Olivier Coulaud. Nonlinear mapping and distance geometry. *Optimization Letters*, 14(2):453–467, 2020.

[36] Antanas Žilinskas and Julius Žilinskas. Parallel genetic algorithm: Assessment of performance in multidimensional scaling. *Proceedings of GECCO 2007: Genetic and Evolutionary Computation Conference*, (January 2007):1492–1501, 2007.

[37] Seung-Hee Bae. Parallel multidimensional scaling performance on multicore systems. In *2008 IEEE Fourth International Conference on eScience*, pages 695–702. IEEE, 2008.

[38] Piotr Pawliczek and Witold Dzwinel. Parallel implementation of multidimensional scaling algorithm based on particle dynamics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6067 LNCS(PART 1):312–321, 2010.

[39] Jong Youl Choi, Seung-Hee Bae, Xiaohong Qiu, and Geoffrey Fox. High performance dimension reduction and visualization for large high-dimensional data analysis. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 331–340. IEEE, 2010.

[40] Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. High performance multi-dimensional scaling for large high-dimensional data visualization. *IEEE Transaction of Parallel and Distributed System*, 2012.

[41] Message Passing Interface Forum. MPI: A message-passing interface standard – version 4.0, June 2021.

[42] Jan de Leeuw. Applications of convex analysis to multidimensional scaling. 2000.

[43] U-Wai Lok, Pengfei Song, Joshua D Trzasko, Eric A Borisch, Ron Daigle, and Shigao Chen. Parallel implementation of randomized singular value decomposition and randomized spatial downsampling for real time ultrafast microvessel imaging on a multi-core cpus architecture. In *2018 IEEE International Ultrasonics Symposium (IUS)*, pages 1–4. IEEE, 2018.

[44] Hao Ji and Yaohang Li. Gpu accelerated randomized singular value decomposition and its application in image compression. *Proc. of MSVESCC*, pages 39–45, 2014.

[45] Yuechao Lu, Ichitaro Yamazaki, Fumihiko Ino, Yasuyuki Matsushita, Stanimire Tomov, and Jack Dongarra. Reducing the amount of out-of-core data access for gpu-accelerated randomized svd. *Concurrency and Computation: Practice and Experience*, 32(19):e5754, 2020.

[46] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[47] Chongxiao Cao, Thomas Herault, George Bosilca, and Jack Dongarra. Design for a soft error resilient dynamic task-based runtime. In *International Parallel and Distributed Processing Symposium*.

[48] Martin Tillenius. *Scientific Computing on Multicore Architectures*. PhD thesis, Uppsala Universiteit, 2014.

[49] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*.

[50] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[51] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Transactions on Mathematical Software*.

[52] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[53] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. A hybridization methodology for high-performance linear algebra software for gpus. In *GPU Computing Gems Jade Edition*, pages 473–484. Elsevier, 2012.

[54] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 932–943, 2011.

[55] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. Lu factorization for accelerator-based systems. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 217–224, 2011.

[56] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[57] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[58] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. Plasma users guide. Technical report, Technical report, ICL, UTK, 2009.

[59] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks. In *Workshop on Communication Architecture for Clusters (CAC 2007), workshop held in conjunction with IPDPS 2007*, Long Beach, California, United States, March 2007.

[60] Francois Trahay, Alexandre Denis, Olivier Aumage, and Raymond Namyst. Improving reactivity and communication overlap in mpi using a generic i/o manager. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 170–177. Springer, 2007.

[61] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274.

[62] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations: theory and practice. *arXiv preprint arXiv:0806.2159*, 2008.

[63] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[64] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Tile qr factorization with parallel panel processing for multicore architectures. In *2010 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2010.

[65] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[66] Jack Dongarra, Mathieu Faverge, Thomas Herault, Mathias Jacquelin, Julien Langou, and Yves Robert. Hierarchical QR factorization algorithms for multi-core cluster systems. *Parallel Computing*, 39(4-5):212–232, 2013.

[67] Dalal Sukkari, Hatem Ltaief, David Keyes, and Mathieu Faverge. Leveraging Task-Based Polar Decomposition Using PARSEC on Massively Parallel Systems. In *IEEE Cluster 2019*, Albuquerque, United States, September 2019.

[68] Marc Sergent, David Goudin, Samuel Thibault, and Olivier Aumage. Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. In *HIPS - 21st International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Chicago, United States, May 2016.

[69] Alexandre Denis and François Trahay. MPI Overlap: Benchmark and Analysis. In *International Conference on Parallel Processing*, 45th International Conference on Parallel Processing, Philadelphia, United States, August 2016.

[70] Alexandre Denis. pioman: a pthread-based Multithreaded Communication Engine. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, March 2015.

[71] L. Kermarrec, A. Franc, F. Rimet, P. Chaumeil, J.-M. Frigerio, J.-F. Humbert, and A. Bouchez. A next-generation sequencing approach to river biomonitoring using benthic diatoms. *Freshwater Science*, 33:349–363, 2014.

[72] D. Gusfield. *Algorithms on strings, trees, and sequences.* Cambridge University Press, Cambridge, UK, 1997.

[73] S. B. Needleman and C. D. Wunsch. A general method applicable to search for similarities in the amino-acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[74] Michel Daydé, Benjamin Depardon, Alain Franc, Jean-François Gibrat, Romaric Guilllier, Yasaman Karami, Frédéric Sutter, Bruck Taddese, Marie Chabbert, and Sylvie Thérond. E-biothon: An experimental platform for bioinformatics. In *2015 Computer Science and Information Technologies (CSIT)*, pages 1–4, 2015.

[75] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. Task-Based Parallel Programming for Scalable Algorithms: application to Matrix Multiplication. Research Report RR-9461, Inria Bordeaux - Sud-Ouest, February 2022.

[76] J. Wang. *Geometric structure if high-dimensional data and dimensionality reduction.* Springer & Higher Education Press, 2012.

A Slightly more details on the complexity of RSVD

We here present more details on the complexity of the main algorithmic steps of the RSVD, as a complement to section 3.3.

We note $f(m,k) \sim_{m\to\infty} g(m,k)$ when $\lim_{m\to\infty} \frac{f(m,k)}{g(m,k)} = 1$, and $f(m,n,k) \sim_{m,n\to\infty} g(m,n,k)$ when $\lim_{m,n\to\infty} \frac{f(m,n,k)}{g(m,n,k)} = 1$ and assume standard matrix multiplications and Householder QR factorizations. Here is the asymptotic flop complexity of the computational steps from the RSVD (Algorithm 2) for dimensions $m \geq n \gg k$: GE1 $(m,n,k) \sim_{m,n\to\infty} 2mnk$, QR1 $(m,k) \sim_{m\to\infty} 2mk^2$, Q1 $(m,k) \sim_{m\to\infty} 2mk^2$, GE2 $(m,n,k) \sim_{m,n\to\infty} 2mnk$, QR-SVD $(n,k) \sim_{n\to\infty} 4nk^2$, GE3 $(m,k) \sim_{m\to\infty} 2mk^2$. Note that, in details, the `QR-SVD` (Algorithm 3) consists of QR2 $(n,k) \sim_{n\to\infty} 2nk^2$, Q2 $(n,k) \sim_{n\to\infty} 2nk^2$, $k\times k$ SVD $(k) = \mathcal{O}(1)$ as $m,n \to \infty$, GE4 $(n,k) \sim_{n\to\infty} 2nk^2$. All in all, the flop count of the RSVD (Algorithm 2) is therefore dominated by `GE1` and `GE2` matrix multiplications and satisfies RSVD $(m,n,k) \sim_{m,n\to\infty} 4mnk$. In particular, we have RSVD $(m,n,k) = \mathcal{O}(mn)$ as $m,n \to \infty$. The RSVD step (called with $m = n$) dominating the RSVD-MDS (Algorithm 1), the flop count of the latter satisfies RSVD-MDS $(m,k) = \mathcal{O}(m^2)$ as $m \to \infty$.

B I/Os: task-based management of the write operations (`WRITE_X`)

The `mds` application eventually aims at writing the resulting point cloud $X_{k_{\text{MDS}}}$ on disk in an HDF5 file. The size of $X_{k_{\text{MDS}}}$ is much smaller than D because the number of columns k_{MDS} is at least one hundred times smaller than m in our context. Still, this object is distributed in order to avoid memory imbalance and such that we can study MDS with an RSVD parameterized by any value of the prescribed rank k (including for $k > 1,000$). We therefore also save $X_{k_{\text{MDS}}}$ in a distributed way. Note that, similarly to the read case, we avoid concurrent writing of all the tiles for a performance concern. We therefore convert the tile matrix into large blocks of rows distributed over MPI processes (1d decomposition over rows) and then perform a parallel HDF5 call to write the blocks (see Figure 9). The conversion of tiles to large blocks is handled thanks to a task-based algorithm relying on the `map` function. Figure 6 discussed in section 4.5 showed the resulting performance of this `WRITE_X` operation. For instance, for the largest matrix (Lall) of the collection, and with $k = 1,000$, we save to disk $k_{\text{MDS}} = k^+ = 583$ columns (we remind that $k_{MDS} \in [2, k^+ = 583]$). The corresponding elapsed for the whole process (conversion into large blocks and HDF5 writing) is then of 11 seconds.

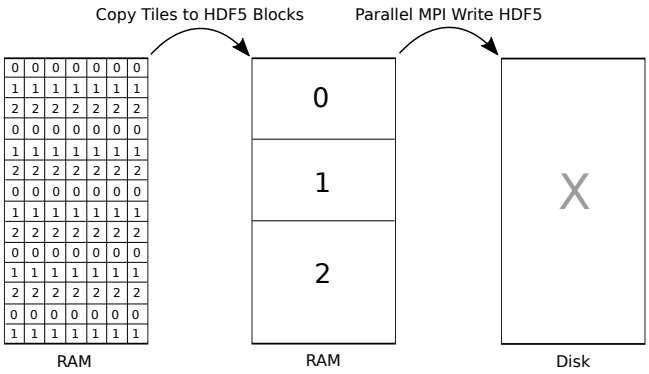


Figure 9: Copy tiles of the X matrix into large blocks with 3 MPI processes, then call parallel MPI HDF5 write.

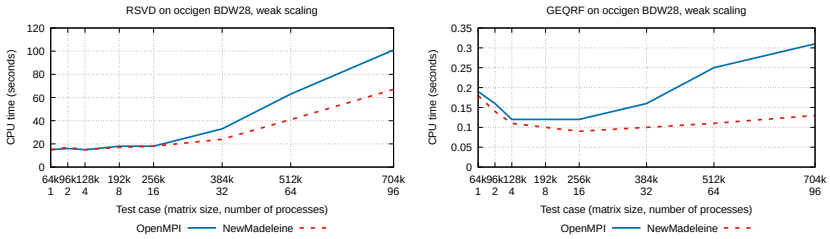
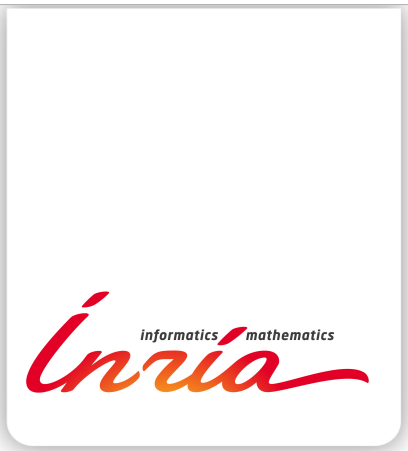


Figure 10: RSVD (left) and GEQRF (right) CPU times the homogeneous machine BDW28 with OpenMPI and nmad, weak scaling, $nb = 320$, $k = 1,000$.

C Slightly more details on the impact of the communication back-end on the performance of the RSVD

We provide the execution time of the RSVD and GEQRF operations in a weak scaling set up on the homogeneous machine BDW28 with both OpenMPI and `newmadeleine` back-ends in Figure 10.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399