Scaling Dense Linear Algebra on Multicore and Beyond: a Survey

Paolo Viviani
Computer Science Department
University of Torino, Italy
Noesis Solutions NV
Leuven, Belgium
Email: pviviani@unito.it

Maurizio Drocco
Marco Aldinucci
Computer Science Department
University of Torino, Italy
Email: {drocco, aldinuc}@di.unito.it

***Abstract*—The present trend in big-data analytics is to exploit algorithms with linear or even sub-linear time complexity, in this sense it is usually worth to investigate if the available techniques can be approximated to reach an affordable complexity. However, there are still problems in data science and engineering that involve algorithms with higher time complexity, like matrix inversion or Singular Value Decomposition (SVD). This work presents the results of a survey that reviews a number of tools meant to perform dense linear algebra at "Big Data" scale: namely, the proposed approach aims first to define a feasibility boundary for the problem size of shared-memory matrix factorizations, then to understand whether it is convenient to employ specific tools meant to scale out such dense linear algebra tasks on distributed platforms. The survey will eventually discuss the presented tools from the point of view of domain experts (data scientist, engineers), hence focusing on the trade-off between usability and performance.**

I. Introduction

With the steadily growing amount of data involved in scientific and engineering applications, the research towards linear or even sub-linear (with respect to the problem size)algorithms is more urgent than ever. Nevertheless, several techniques, that are widespread in the mentioned domains, cannot depart from complex tasks completely: operations such as dense matrix inversions and factorizations are still fundamental kernels nowadays. Since this kind of *dense linear algebra* problems involve time and memory complexities up to $O(n^3)$ and $O(n^2)$, respectively, solving them for growing problem sizes poses severe challenges over common off-the-shelf workstations.

Over the last decades, the HPC community has proposed several approaches for tackling these challenges, for both shared-memory platforms (i.e., multi/many-core), distributed-memory environments, and their heterogeneous combinations [1]. In this context, the so-called MPI+x model, that consists in combining MPI with some shared-memory programming model, represents the de facto standard for chasing high performance over heterogeneous platforms.

While sophisticated, multi-node set-ups were usually available only to the HPC community, due to the spread of pay-per-use, XaaS public clouds, it is now common for data scientists to have access to a large amount of computational resources. This fed a substantial research effort for developing higher-level frameworks, allowing the HPC-agnostic domain experts (e.g., data scientists) to exploit the available resources. A paradigmatic example of this phenomenon is the rise of Apache Spark, that is becoming a de facto standard for data analytics over distributed platforms.

From the conventional HPC standpoint, this approach is often argued to lead sub-optimal performance. Instead of adopting such a prejudicial perspective, we try to understand when this higher-level approach is as effective as its wide adoption would suggest, both in terms of performance and usability. As part of the study, we also challenge the need to rely on distributed resources at all, by defining a consistent feasibility region for dense matrix factorizations on multi-core platforms.

In this paper, we try to put some order in the complicated landscape of linear algebra libraries, considering them from both the performance and usability viewpoints, but assuming a domain expert (rather than HPC expert) perspective. To this aim, we consider a variety of different tools, ranging from consolidated HPC libraries like BLAS/LAPACK and their vendor-optimized implementations, up to modern data analytics framework like the aforementioned Spark. Particular focus is put into the possible integration of the presented tools into a Python workflow.

A representative kernel, the Singular Value Decomposition (SVD), has been extensively benchmarked to define as accurately as possible a problem size feasibility boundary for shared-memory computations on single nodes. Moreover, the possibility to push such boundary by going distributed through different tools is probed. Furthermore, the performance trade-off of adopting higher-level tools is quantified.

This paper is structured as follows: in Sect. II, we present the SVD use case, defining the theoretical background, its applications, and the specific performance challenges it raises. In Sect. III, we introduce the tools under review, discussing some technical features and the reasons for their inclusion. In Sect. IV, we present the benchmarking methodology and we discuss the observed outcomes and, finally, in Sect. V, we outline the lessons learned from these results.

II. Background

This work will focus on evaluating the performance exhibited by different libraries when executing instances of a SVD computation, that is therefore regarded as benchmark. The importance that this technique plays in many applications of data science and the computational challenge that it represents are the main motivations to this choice, which is also driven by industrial interest in the potential performance improvement in the execution of this task.

A. SVD in Data Science and Engineering

Given a generic complex matrix A, the SVD [2] yields the following factorization:

$$A = U \Sigma V^* \tag{1}$$

where the following properties hold:

- Columns of U are orthonormal eigenvectors of AA^*;
- Columns of V are orthonormal eigenvectors of A^*A;
- Σ is diagonal with non-negative elements which are the square roots of the eigenvalues of both A^*A and AA^*.

This factorization is a significant result of linear algebra, but the theoretical background is out of the scope of this paper. Conversely, a brief recap over the applications of SVD is provided in the following, along with a discussion about the computational complexity involved.

The SVD has a large number of application in data science and engineering [3], hence we regard SVD as a meaningful example of dense linear algebra kernel. Here are reported two examples that, as discussed in Sect. II-B, are particularly representative from the computational viewpoint, given the different kinds of matrices involved:

a) Principal Component Analysis (PCA): it is a major approach to reduce the dimensionality of a complex dataset. The SVD can be used to perform PCA on a dataset [4]. In this case, the input matrices typically present a number of rows much larger than the number of columns. Moreover, only the matrices Σ and V are relevant as PCA outcomes.

b) Moore-Penrose Pseudoinverse (MPP): it represents a generalized inverse for rank-deficient or rectangular matrices [5]. It is widely used to solve least squares approximations in the form of rank-deficient linear systems. A relevant application is the calculation of the weights of a Radial-Basis Function interpolation: in this case the matrix A from equation (1) is square and slightly rank-deficient [6].

Further examples of applications based on SVD include low-rank matrix approximation, image compression, graph analysis, and recommendation systems.

B. Computational Challenges

The time complexity for SVD over a matrix with M rows and N columns such that $M > N$, is $O\left(M^2N + N^3\right)$, with constants defined by the specific algorithm and matrix shape [5]. Equations (2) and (3) represent the relative matrix sizes introduced in Sect. II-A for PCA and MPP, respectively.

$$\boxed{A} = \boxed{U} \; \boxed{\Sigma} \; \boxed{V^*} \tag{2}$$

$$\boxed{A} = \boxed{U} \; \boxed{\Sigma} \; \boxed{V^*} \tag{3}$$

In the following, Equation 2 will be referred to as the TALL-AND-SKINNY case, where $M \gg N$ and the complexity is dominated by the $O\left(M^2\right)$ term. Conversely, Equation 3 will be referred to as the SQUARE case, where $M = N$ and the complexity is $O\left(N^3\right)$.

In the context of SVD computation, the memory footprint significantly affects the capability of the calculation to scale on distributed architectures. By definition, matrix U in Equation (2) and (3) is equal in size to A, hence M by N, with $M \gg N$, while Σ and V are both N by N. This implies that, if matrix A does not fit the memory, then neither the results can be stored in memory: this in principle prevents the problem to scale at all to distributed architecture, at least in terms of problem size. However, as anticipated, there are applications which do not require U to be stored (e.g., PCA), opening up the possibility to scale for the TALL-AND-SKINNY SVD when the matrix A is distributed in advance. It is worth mentioning that this discussion is less relevant for the SQUARE problem, since the $O\left(N^2\right)$ space allocation is less than a bottleneck when compared to the $O(N^3)$ time complexity. Moreover, all the matrices involved are of the same size, hence the memory footprint cannot be reduced by dropping part of the result.

In order to give a more clear picture of the context of dense linear algebra, it is worth noting that the matrix multiplication presents roughly the same complexity, but in this case the available parallelism is larger and the computation pattern is by far more predictable. Along with the rise of techniques heavily based on matrix multiplications (e.g., Deep Neural Networks), this led to the development of dedicated hardware (e.g., GPUs, Google TPU [7], and Intel Nervana Neural Network Processor[1], allowing a significant speedup for matrix multiplications. The relevance of the SVD lies into the performance challenge that it poses, given also its wide range of applications described in Sect. II-A: there is no "SVD machine" that can enable large performance leaps, so it is mostly up to software developers to provide the best performing implementation.

As a final remark, although all the memory-related issues could be avoided by relying on out-of-core processing [8], this approach falls out of the topic of this work, since no tool providing out-of-core SVD can be considered mature enough to be a choice for domain experts.

[1]https://www.intelnervana.com/

III. Tools

The perspective driving the selection of tools considered for this work is that of a domain expert who is interested in performing data analytics and engineering tasks, having little to no knowledge about low-level programming or performance aspects.

Shared-memory tools are introduced in Sect. III-A, focusing on Python-based interfaces, according to the domain-expert perspective. Within the distributed-memory domain, the MPI+x approach (see Sect. I) is considered first, in Sect. III-B. Then, the emergent higher-level approach is considered in Sect. III-C, by presenting a number of tools based on Apache Spark.

A. Shared-Memory Tools

Given the large spectrum of applications for dense linear algebra, it is not surprising that the landscape of libraries aimed to such computations has grown to a quite complicated shape, spanning most of the existing programming languages and hardware platform. However, most of these tools either wrap or reimplement two reference libraries: Netlib's BLAS library [9], and Netlib's LAPACK library [10], respectively oriented to *basic* (i.e. matrix-vector multiplications) and *advanced* (i.e. eigenvalues, SVD) linear algebra operations.

These two libraries provided an API which still holds as a *de-facto* standard for dense linear algebra, even if, due to its complexity, it is almost never used directly nowadays, in favour of some higher-level wrappers, such as Armadillo [11] and NumPy [12], that is introduced below.

Moreover, BLAS and LAPACK libraries evolved in different directions over time. In addition to multithreaded (PLASMA [13]) and distributed-memory (ScaLAPACK [14]) evolutions, some of them are vendor-optimized implementations have been developed. In general, the API of these evolutions is more or less compatible with the reference one.

a) Intel MKL: the Math Kernel Library[2] is usually regarded as the best-in-class implementation of BLAS and LAPACK It comes out of the box with the popular Python distribution dedicated to data scientist: Anaconda[3]. As its API is totally BLAS/LAPACK compatible and hence quite complex, most of the times it is used by means of high-level wrappers. Given the domain expert orientation of this work, this approach has been favoured, relying on the following higher-level libraries.

b) NumPy: it is a package for scientific computing with Python. It provides Python users with N-dimensional arrays managed by a C back-end and it relies on BLAS and LAPACK distributions to perform dense linear algebra. Being a lightweight wrapper on C code, we can safely assume that, at least for the large matrices on which this paper is focused, the performance overhead is negligible with respect to direct calls to the underlying library.

[2]https://software.intel.com/en-us/mkl

[3]https://anaconda.org/anaconda/python

c) Intel DAAL: Intel Data Analytics Acceleration Library[4] is a high-level library that provides C++, Java/Scala, and Python bindings all with an API which is friendly for domain experts. Its peculiarity is the capability to execute algorithms in three different ways: shared memory, distributed, and streaming (referred by the documentation as batch, distributed and online); this work focused on the former two implementations. Also DAAL relies internally on MKL, hence the same state-of-the-art performance are expected for plain MKL.

Note the above-mentioned PLASMA multithreaded replacement for LAPACK has been deliberately excluded from the presentation, since it has been reported to perform very closely to MKL [13]; moreover, it has questionable capability to be integrated into higher-level libraries, that is a substantial requisite under the perspective considered for this work.

Finally, libraries aimed to perform SVD on GPUs exist (e.g. Magma [13], CuSolver[5] and higher-level wrappers as ArrayFire[6]) and their performance has been already assessed [15]. Although some good performance was observed, the capability to handle large problems is limited by the GPU memory, which is usually smaller than CPU memory; for this reason, this survey targets CPU-only libraries.

B. Conventional Distributed Tools

a) DPLASMA: Distributed PLASMA [16] is MPI-based and can be seen either as a multithreaded evolution of ScaLAPACK or a distributed evolution of PLASMA. It is included in this review since it represents the state of the art for what concern the conventional MPI+x HPC approach, so it is considered a reference for the performance of distributed dense linear algebra, regardless of the usability, which is hindered by an extremely complex API and the single program, multiple data (SPMD) programming model.

b) Intel DAAL: the distributed implementation of Intel DAAL is agnostic with respect to the programming model. In fact it provides building blocks that allow the user to rely on whatever technique to transfer data and execute the building blocs across different nodes. One of the options advertised in the documentation is the SPMD model based on MPI. The main advantage with respect to DPLASMA is the much simpler API, while the added complexity of SPMD remains, even if it is feasible to perform calculations only relying on Python code.

C. Emerging Technologies

a) Spark + MLlib: Apache Spark [17] is a framework developed to manipulate large datasets in memory on distributed architectures. Its main advantage is that provides the user with a fully sequential programming model, managing the data distribution in a transparent way, while relying on a Dataflow model to perform computations. Java, Scala, and

[4]https://software.intel.com/articles/opendaal

[5]http://docs.nvidia.com/cuda/cusolver/index.html

[6]https://github.com/arrayfire/arrayfire

Python bindings are provided. MLlib [18] is Spark's open-source distributed machine learning library, including linear algebra and the SVD, relying on Spark's own distributed data structure (i.e., the Resilient Distributed Dataset, RDD). At the lowest level, MLlib also relies on BLAS/LAPACK libraries to perform dense matrix computations, it is however not straightforward to make it use the vendor optimized implementation instead of a Java-based replacement. These details will be discussed in depth in the next section. Spark's outstanding popularity among the data science community makes it an obvious choice for inclusion in this survey.

b) Spark + DAAL: Intel DAAL provides also the capability to run its distributed algorithms on top of Apache Spark, in particular its Scala API acts as a *drop-in* replacement for MLlib. In order to benefit from the advertised DAAL performance boost, there is, in principle, no need to change the code developed for MLlib. The inclusion of this configuration in the survey aims to probe the possibility to improve the performance of a popular, easy to use, tool by leveraging best-in-class numerical libraries, without any impact on consolidated workflows.

The following section presents quantitative and qualitative results for the SVD benchmark, when executed by the mentioned tools.

IV. Benchmarks

The performance results are at first presented for individual tools varying the problem size, then a comparison is drawn and the relevant remarks are reported. The tests have been performed on a cluster [19] of up to 4 nodes, each equipped with $2\times$ Xeon E5-2680 v3 @ 2.50GHz with 24 cores and 128GB RAM. Interconnecting network is 10GbE. Synthetic datasets of double-precision real values are used to perform the benchmarks and their generation time is either not considered in the timing of the calculation or it is measured to be of negligible impact on the overall timing. Standard deviation of timing results is negligible.

Analysis of numerical accuracy of the results is not relevant to this work, since all the presented tools are based on widely adopted algorithms and implementations that are expected to provide consistent results.

A. Shared Memory

To be consistent with the high-level, domain expert approach, MKL (2018.0.0) is never used directly. Conversely, it is used by means of Python wrappers, namely NumPy (1.13.1) and DAAL (2018.0.0).

Both the TALL-AND-SKINNY (i.e. $M \gg N$) and the SQUARE (i.e., $M = N$) cases have been considered for the benchmark, while the range of problem sizes explored has been dictated by the capability to fit the matrices in memory for the former case, and by the exploding computing complexity for the latter.

Figure 1 presents the timing for TALL-AND-SKINNY matrices, while Fig. 2 reports results for SQUARE matrices. The largest matrix that can be handled with the considered tools is

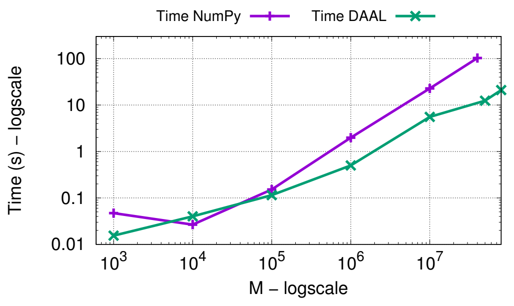

Fig. 1. Single node, shared memory. Computation of TALL-AND-SKINNY SVD. Matrix size is $M \times 100$. Lower is better.

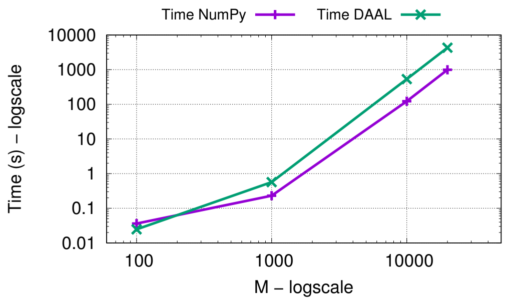

Fig. 2. Single node, shared memory. Computation of SQUARE SVD. Matrix size is $M \times M$. Lower is better.

$80M\times100$, achieved with DAAL. Given that both NumPy and DAAL rely on MKL to perform the SVD, it is quite surprising to observe that the performance is different. Moreover, the NumPy variant for the TALL-AND-SKINNY case crashes for smaller matrix sizes with respect to the DAAL variant.

This asymmetry can be traced back to the LAPACK function that is called internally from the two libraries: for NumPy is the divide-and-conquer `dgesdd`, while for DAAL it is the standard `dgesvd`. It is known that, while faster in some cases, the divide-and-conquer implementation is much more heavy on the memory, which can explain the presented results.

B. MPI-based

As mentioned in Sect. III-B, two libraries from the MPI-based family have been considered: DAAL over MPI and DPLASMA (from PaRSEC version 2.0.0rc2).

The former has been accessed using its Python interface that is, however, less straightforward with respect to the shared memory version. In fact the choice to make the library independent from the underlying message-passing back-end (i.e., the MPI+x approach), leaves the burden of setting up the data transfer between computation stages on the programmer's shoulders.

The DAAL implementation of SVD itself has some drawback, since its individual stages are not able to cope with chunks of matrices that have more columns than rows, which is always the case when dealing with SQUARE matrices. For this reason, the benchmark for the SQUARE case is omitted. For the same reason, the case with $M = 1000, N = 100$ need the number of processes to be ≤ 10 in order to keep the shape

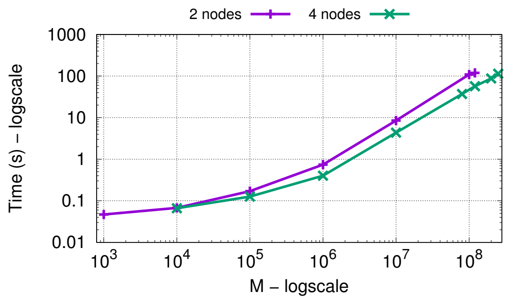

Fig. 3. DAAL over MPI. Computation of TALL-AND-SKINNY SVD. Matrix size is $M \times 100$. Number of MPI processes is variable, best case is shown. Lower is better.

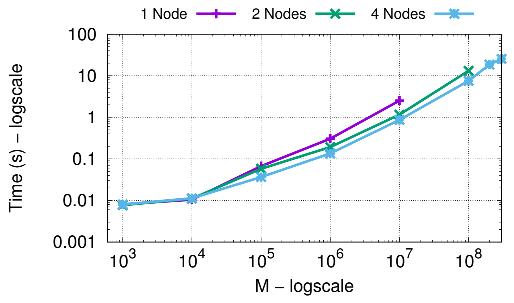

Fig. 4. DPLASMA over MPI. Computation of TALL-AND-SKINNY SVD. Matrix size is $M \times 100$. Lower is better.

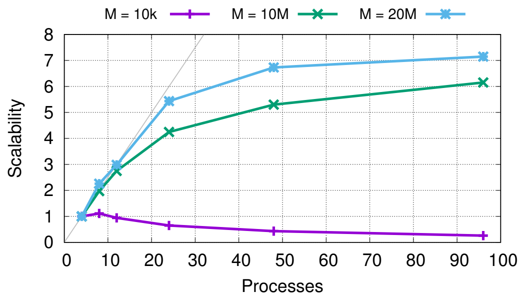

Fig. 5. DAAL over MPI, scalability relative to the 4 processes-time. 4 nodes. Computation of TALL-AND-SKINNY SVD. Matrix size is $M \times 100$. Higher is better, grey line represents ideal trend.

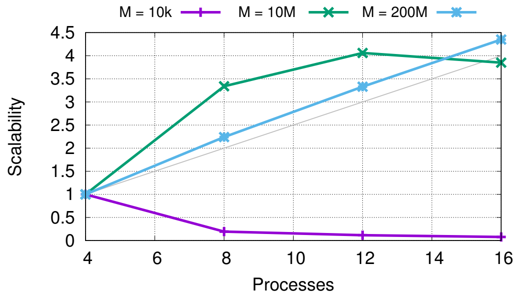

Fig. 6. DPLASMA, scalability relative to the 4 processes time. 4 nodes. Computation of TALL-AND-SKINNY SVD. Matrix size is $M \times 100$. Higher is better, grey line represents ideal trend.

of the chunks at least square.

Figure 3 shows the performance featured by DAAL over MPI when executing SVD in the TALL-AND-SKINNY case. A few notes can be made on the results reported: first, the largest matrix computed for 2 and 4 nodes is larger than what achieved for the shared-memory case, as expected from the considerations made in Sect. II-B. However, the maximum size is not larger by a factor of 2 or 4 with respect to the shared-memory deployment, hence some memory overhead is involved in this implementation.

As a reference about the state-of-the-art performance for dense distributed linear algebra, the DPLASMA library is considered. DPLASMA provides C and Fortran interfaces, but cannot be immediately integrated into a Python workflow. Fig. 4 reports the results for different test configurations: single-node multiprocessing, 2, and 4 nodes. DPLASMA scales almost linearly with the problem size and, surprisingly, it outperforms even the shared memory version. Note that, also in this case, the SQUARE case is omitted, since it has shown no scalability on multiple nodes and it performs worse than the shared-memory implementation. Further discussion of these results is reported in Sect. IV-D.

A point that emerges from both the MPI-based benchmarks, is that the optimal number of MPI processes does not match the number of physical nodes. Figure 5 and 6 show a few samples of the scalability data for a fixed problem size and 4 nodes. As somehow expected, the scalability depend on the granularity of the problem: using multiple processes for a small problem adds significant overhead.

C. Spark-based

Also in the case of libraries running upon Spark (2.2.0), this work followed a Python-based approach. All the results are presented using the maximum available parallelism as detected by Spark initial configuration.

The MLlib syntax is very similar to NumPy and, combined with the sequential programming model exposed by Spark, it makes very easy to develop code effectively without having to deal with low-level implementation details. Figure 7 reports the results for MLlib on Spark: it is possible to note that, while there is never a performance advantage in moving the calculation on multiple nodes, there is good capability to scale the problem size. Results for the SQUARE case are again not reported as they are not relevant: there is a significant performance penalty with respect to the shared memory approach, no performance scalability on more nodes and, given what discussed in Sect. II-B, no possibility to scale on more nodes.

The DAAL over Spark configuration has been also tested. However, while the Scala API for DAAL provides a direct, drop-in replacement for MLlib (see Sect. III-C), due to compatibility issues between the involved frameworks we were not able to test such implementation.[7]. Conversely, we were forced to exploit the Python-based interface to implement the DAAL over Spark variant, that required an effort comparable with the one required to move from the shared memory to the MPI version. Moreover, the performance achieved are significantly

[7]This is probably due to immaturity of DAAL-over-Spark implementation, compared to the DAAL-over-MPI version.

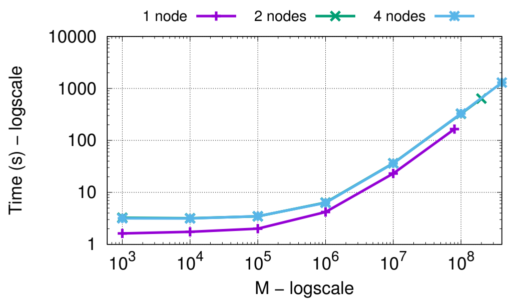

Fig. 7. MLlib over Spark. Computation of TALL-AND-SKINNY SVD. Matrix size is $M \times 100$. Lower is better. 2 nodes and 4 nodes lines are almost superimposed, 2 nodes line stops earlier.

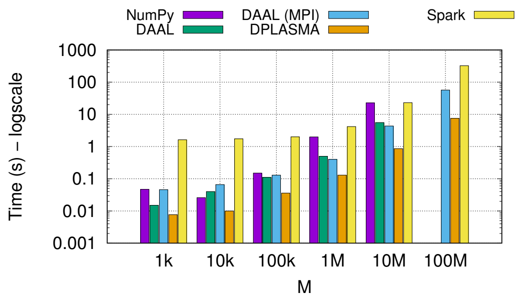

Fig. 8. Comparison of all the different configurations for computation of TALL-AND-SKINNY SVD. Matrix size is $M \times 100$. Lower is better.

worse than what can be achieved with naive MLlib, hence it does not make sense to report the figures here. This can be possibly due to implementation issues, which might be solved in a more mature release of the package.

D. Comparisons and Remarks

a) TALL-AND-SKINNY *case:* This problem turned out to be by far the more interesting to test, as the slightly lower computational complexity compared to the memory requirements, and the possibility to scale on distributed platforms with the problem size, allowed the authors to observe a rich collection of phenomena.

Figure 8 and Tab. I resume the data gathered during the survey, presenting both a comparison of the execution time of the best configuration tested for each tool as well as all the upper bound of the problem size that can be handled with each tool. Given these data, it is possible to make some general remarks, which are reported hereafter.

The first observation is that, for any applicable problem size, DPLASMA turned out to be the best performing library for the SQUARE case. While not surprising for distributed architectures, this result is not expected for a shared-memory single node. There are two points that may explain this behaviour, that should be further investigated in a future work:

1) It was observed that the CPU is not fully utilised during the whole calculation by the best performing shared-memory library (DAAL): even if a certain portion of the calculation keeps all the CPU cores busy, for the largest part there are at most 2 cores working at a given time. The DPLASMA implementation spawn a number of processes, where each of them multithreaded, performing in fact some resource overbooking: this does not bring any advantage for the most intesive part of the computation, but parallelises the other parts through multiple processes, running indeed faster overall. Moreover, the following point could also contribute to explain why this holds regardless of the problem size.
2) The shared-memory LAPACK implementation underlying DPLASMA, which is PLASMA, might be more efficient than previous benchmarks would suggest [13], when compared to MKL.

The aforementioned point 1 can also explain another interesting behaviour: resource overbooking provides significant advantage on distributed set-ups. As Fig. 5 and 6 highlight, for a constant number of physical nodes, and given a sufficient problem size, the performance improves with the growing number of processes. This resource overbooking can, thanks to an efficient implementation, exploit more parallelism than the equivalent single node version up to the point that the trend in Fig. 6 is superlinear for DPLASMA.

Apart from the purely performance point of view, the usability trade-off of these approaches should be discussed. Given a domain expert used to Python-based workflows, we can safely assume that the effort of performing as SVD with NumPy is negligible, to the point that such implementation might be already in place. From this perspective, the transition to single-node DAAL is almost painless as shown by the comparison of Listings 1 and 2.

Moving to the MPI-based tools, the landscape changes significantly: DPLASMA can not be integrated into a data science workflow without a deep understanding on numerical libraries that goes far beyond the domain expert's knowledge. DAAL can partially overcome such limitation, along with Python bindings for MPI, but the SPMD paradigm still represents a burden as it makes the code longer and less readable.

For what concerns MLlib and Spark, there is a very good trade-off between programmability and scalability with respect to the problem size. However, it is less than ideal when coming to pure performance.

The considerations above are made regardless of the effort to set-up the benchmarking environment, but in a real-life application, this mainly IT work must be taken into account when choosing the right tool to perform a calculation. In fact, while the effort to set up an MPI cluster is well known and might be mitigated by leveraging already existing infrastructures, the apparent simplicity of Spark programming hides all the complexity of setting up a well-performing cluster, that is also something not affordable for a researcher focused on the domain logic.

b) SQUARE *case:* In this case the data obtained is less rich, but not less significant: given that scaling beyond the capability of a single machine is not possible for in-memory computations, due to the definition of the problem itself (Sect. II-B), the focus is only on the performance that can be achieved. NumPy proved to be the fastest library, not

TABLE I
TIME FOR SOLVING TALL-AND-SKINNY SVD. BOLD FIGURES REPRESENT THE BEST TIME FOR THE GIVEN TOOL/SIZE SET-UP AND OPTIMAL NUMBER OF PROCESSES. TIME IN SECONDS. - NOT TESTED, $\times$ CRASHED FOR INSUFFICIENT MEMORY.

M	NumPy	DAAL	DAAL (MPI)		DPLASMA (MPI)			MLlib (Spark)		
($N = 100$)	1 node	1 node	2 nodes	4 nodes	1 node	2 nodes	4 nodes	1 node	2 nodes	4 nodes
1k	0.047	0.015	**0.046**	-	0.0080	**0.0077**	0.0080	**1.62**	3.25	3.16
10k	0.026	0.04	0.067	**0.066**	**0.010**	0.011	0.011	**1.74**	3.18	3.14
100k	0.15	0.11	0.17	**0.13**	0.067	0.057	**0.036**	**2.00**	3.46	3.48
1M	1.99	0.50	0.74	**0.40**	0.30	0.19	**0.13**	**4.18**	6.37	6.28
10M	22.7	5.56	8.50	**4.35**	2.52	1.15	**0.86**	**23.0**	36.4	36.1
40M	102.9	-	-	-	-	-	-	-	-	-
80M	$\times$	21.0	-	**36.9**	23.3	-	-	165.4	-	-
100M	$\times$	$\times$	108.7	**56.9**	$\times$	13.1	**7.5**	$\times$	**324.7**	328.8
120M	$\times$	$\times$	119.1	-	$\times$	-	-	-	-	-
200M	$\times$	$\times$	$\times$	**87.4**	$\times$	$\times$	**18.5**	$\times$	**638.6**	-
250M	$\times$	$\times$	$\times$	**113.8**	$\times$	$\times$	-	$\times$	$\times$	-
300M	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	**25.4**	$\times$	$\times$	-
400M	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	**1297.4**

```python
import numpy as np
data = #numpy 2D-array
U,s,V = np.linalg.svd(data, full_matrices=False)
```

Listing 1. SVD computation with NumPy. `full_matrices=False` is required for the $m > n$ case.

```python
from daal.algorithms import svd
from daal.data_management import
    HomogenNumericTable
npdata = #numpy 2D-array
data = HomogenNumericTable(npdata)
algorithm = svd.Batch()
algorithm.input.set(svd.data, data)
results = algorithm.compute()
```

Listing 2. SVD computation with DAAL.

matched by any other tool in any configuration, either single-node or distributed. The gap with DAAL presented in Fig. 2 is due to the LAPACK routine called, in particular dgesdd is much faster than dgesvd when all the singular vectors are required [20].

It is noteworthy that in this case no scalability is shown by neither DAAL or DPLASMA, while the latter's dgesvd is slower than MKL's dgesdd.

V. CONCLUSION AND FUTURE WORK

The presented results and the lessons learned by setting up the benchmarking environment lead eventually to a classification of the tools based on three coordinates: performance, usability and set-up complexity. As expected, there is no such thing as one library to rule them all, but of course this classification changes depending on the specific problem size and shape. For instance, in the SQUARE case (i.e., $M = N$), the indication is to use the simplest shared-memory available, possibly linking to some dgesdd *divide-and-conquer* implementation of the SVD algorithm. Within the scope of this work, the obvious choice is NumPy, but there are high-level libraries for other languages that provide the same functionality, such as Armadillo [11].

The TALL-AND-SKINNY case (i.e., $M \gg N$) presents a different situation: if the problem fits the memory, shared-memory DAAL is the best trade-off between performance and usability. PLASMA, being is faster by almost an order of magnitude, it is the best option if speed is paramount, but it is hardly useful to domain experts unless an integration with higher-level tools become readily available. The same argument holds when A does not fit the memory: DPLASMA is by far the fastest option and it is the second best when considering the capability to scale on more nodes with the problem size. However, Spark provides both good scalability (at least up to the limited number of nodes considered in this work) and a simpler SPMD-free programming model, making MLlib-Spark a valuable option among those presented for domain experts. Finally, distributed DAAL is slow when using Spark as a back-end, whereas it represents a good trade-off when coupled with MPI: the performance is better than Spark and the usability is better than DPLASMA, but it does not match the strengths of the two.

It is noteworthy that, as mentioned before, setting up a Spark cluster is not easier than dealing with MPI, even if this complexity is hidden further from the user and weights more on the shoulders of IT professionals and system administrators. In particular the offer of Spark itself, and Spark-ready infrastructures as XaaS is steadily growing, making such set-ups readily available to data scientists.

As an additional lesson learned, the different relative performance of MKL and DPLASMA for the two different SVD cases is an evidence of how complicated is the landscape of linear algebra libraries, further justifying the effort to put some order into this scenario. On the other hand, much work still needs to be done: matter for a future work will be the identification of a broader class of problems and the related linear algebra functions to be benchmarked, from the viewpoints of high-level workflows typically used by domain experts. Still, this work somehow addresses an upper bound of complexity for linear algebra kernels, hence other applications that present more parallelism can benefit even more from the high-performance libraries presented here.

Acknowledgment

This work has been developed thanks to the C3S supercomputing facility and it has partially supported by the OptiBike experiment in the H2020 I4MS Project 680481 Fortissimo 2.

References

[1] M. Aldinucci, S. Campa *et al.*, "Behavioural skeletons for component autonomic management on grids," in *Making Grids Work*. Springer, 2008, pp. 3–15.

[2] A. K. Cline and I. S. Dhillon, *Computation of the Singular Value Decomposition*. CRC Press, Jan. 2006.

[3] D. J. Hand, "Understanding Complex Datasets: Data Mining with Matrix Decompositions by David B. Skillicorn," *International Statistical Review*, vol. 76, no. 1, pp. 142–142, 2008.

[4] J. Shlens, "A tutorial on principal component analysis," *CoRR*, vol. abs/1404.1100, 2014.

[5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[6] W. H. Press, S. A. Teukolsky *et al.*, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. New York, NY, USA: Cambridge University Press, 2007.

[7] N. P. Jouppi, C. Young *et al.*, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017.

[8] K. Kabir, A. Haidar *et al.*, "A framework for out of memory SVD algorithms," in *Proc. of the 32nd International Supercomputing Conference (ISC)*, Frankfurt, Germany, 2017.

[9] J. Dongarra, J. Du Croz *et al.*, "An extended set of fortran basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, 1988.

[10] E. Anderson, Z. Bai *et al.*, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[11] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," *Journal of Open Source Software*, vol. 1, p. 26, 2016.

[12] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[13] E. Agullo, J. Demmel *et al.*, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, p. 012037, 2009.

[14] J. Choi, J. Demmel *et al.*, "Scalapack: A portable linear algebra library for distributed memory computers – design issues and performance," *Computer Physics Communications*, vol. 97, pp. 1–15, 08 1996.

[15] P. Viviani, M. Aldinucci *et al.*, "Multiple back-end support for the armadillo linear algebra interface," in *Proc. of the 32nd ACM Symposium on Applied Computing (SAC)*, Marrakesh, Morocco, Apr. 2017.

[16] G. Bosilca, A. Bouteiller *et al.*, "Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA," in *Proc. of the 25th IEEE Intl. Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, Anchorage, USA, May 2011.

[17] M. Zaharia, R. S. Xin *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[18] X. Meng, J. Bradley *et al.*, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, Jan. 2016.

[19] M. Aldinucci, S. Bagnasco *et al.*, "OCCAM: a flexible, multi-purpose and extendable HPC cluster," *Journal of Physics: Conference Series*, vol. 898, no. 8, p. 082039, 2017.

[20] M. Gu, J. Demmel, and I. Dhillon, "Efficient computation of the singular value decomposition with applications to least squares problems," Tech. Rep., 1994.