Taming multi-node accelerated analytics: an experience in porting MATLAB to scale with Python

Paolo Viviani, Giacomo Vitali, Davide Lengani, Alberto Scionti, Chiara Vercellino and Olivier Terzo

Abstract High Performance Data Analytics (HPDA) at scale is a multifaceted problem that involves distributed computing resources, their location with respect to the data placement, fast networks, optimized I/O, hardware accelerators and a fairly complex software stack that draws both from the legacy of HPC and the cloud world. This complexity does not cope well with the needs of domain experts who desire to focus on their algorithms without having to understand all the features of the underlying infrastructure. Among these domain experts, engineers often rely on MATLAB to quickly model their complex numerical computations with a simple, textbook-like syntax and an effective Integrated Development Environment (IDE). On the other end, MATLAB was not designed with large-scale, out-of-core computations in mind, despite the introduction of some parallel computing tools (e.g., distributed arrays and `spmd` loops). In an ideal world, a domain expert should only focus on its application logic, while runtime/parallel computing experts should provide tools that act behind the scenes to efficiently distribute the computations on available resources and to provide optimal performance. Conversely, in real life it often happens that the domain expert prototypes its code with MATLAB on a small scale, then HPDA/HPC experts will leverage proper tools to deploy it at scale; this

Paolo Viviani
LINKS Foundation, Torino e-mail: paolo.viviani@linksfoundation.com

Giacomo Vitali
LINKS Foundation, Torino e-mail: giacomo.vitali@linksfoundation.com

Davide Lengani
DIME - University of Genova, CINI HPC-KTT laboratory e-mail: davide.lengani@edu.unige.it

Alberto Scionti
LINKS Foundation, Torino e-mail: alberto.scionti@linksfoundation.com

Chiara Vercellino
LINKS Foundation, Torino e-mail: chiara.vercellino@linksfoundation.com

Olivier Terzo
LINKS Foundation, Torino e-mail: olivier.terzo@linksfoundation.com

causes a significant effort overhead, as development needs to be performed twice, possibly with multiple iterations. The rise of Python as the language of choice for a huge number of data scientists, along with its open ecosystem, led to the development of many tools that tried to achieve a convergence between the goals of domain experts and the need for performance and scalability. Sometimes building upon existing frameworks (e.g. PySpark) or starting from scratch in pure Python (e.g. Dask), these tools bear promise to allow engineers to write their code with a reasonably textbook-like syntax, while retaining the capability to scale among multiple nodes (and accelerators) with minimal intervention to the application code. This paper discusses the process of porting an engineering application written in MATLAB (parallelized with existing toolboxes) to Python using Dask. An indication of the scalability results is also given, including a 20x speedup with respect to the MATLAB code using the same setup.

1 Introduction

As the availability of high-fidelity engineering simulations becomes more common thanks to the low access barrier to High-Performance Computing (HPC) resources, the need to extract engineering knowledge and value from the significant amount of data they produce becomes critical. On the other hand, the sheer amount of data itself introduces significant technical challenges to perform the kinds of analyses, due to both the need to handle datasets that possibly do not fit the memory of the given workstation, and the need to provide results in a short time to inform product development decisions.

These challenges typically fall within the High Performance Data Analytics (HPDA) topic, that involves distributed computing resources, their allocation with respect to the data placement, fast networks, optimized I/O, hardware accelerators and a fairly stratified software stack that draws both from the legacy of HPC and the cloud world. This complexity does not cope well with the needs of domain experts who desire to focus on their algorithms without having to understand all the features of the underlying infrastructure; in fact, engineers typically like to quickly model their numerical computations using MATLAB, leveraging the simple, textbook-like syntax and the powerful Integrated Development Environment (IDE). On the other end, MATLAB was not designed with large-scale, out-of-core computations in mind, and despite the introduction of powerful parallel computing tools (e.g., distributed arrays), it is necessary to heavily mix the application logic with parallel computing concepts (i.e., when using the `spmd` construct).

Ideally, it would be desirable instead to decouple the concerns between domain experts and runtime/parallel computing experts, with the former focused on algorithms and application logic, while the latter providing suitable tools to efficiently distribute the computations on available resources and to provide optimal performance in a transparent way. Conversely, what often happens is that the domain ex-

pert models its application at a small scale, then the code is ported at large scale by an HPC expert, doubling the development effort.

In this context, it is possible to look at the rise of Python in the data science field as a virtuous example: its low access barrier and open ecosystem led to the development of many tools that aimed to achieve the much-needed convergence between the goals of domain experts and the need for performance and scalability. These tools build upon existing frameworks (e.g., PySpark [17]), or implement task parallelism facilities from scratch in pure Python (e.g., Dask [3, 11]), and provide a syntax that is reasonably close to the habits of domain experts, while trying to hide most of the complexity related to parallelism.

The goal of this work is to present a successful experience in leveraging Python scientific computing ecosystem to port a MATLAB application to an HPC infrastructure, reporting initial results and collecting useful lessons learned, in order to provide guidance for domain experts seeking the best performance for their numerical applications.

This paper is structured as follows: Sec. 2 presents the target application and its mathematical components, their theoretical background and computational complexities, and the current scenario of software tools that are suitable to handle this problem in the context of the broader ecosystem, discussing some technical features and the reasons for their use. Sec. 3 introduces the current approach to MATLAB HPDA and introduces the proposed implementation, discussing the advantages over the baseline. In Sec. 4, are presented the outcomes and quantitative results and, finally, in Sec. 5, are outlined the main lessons learned.

2 Background

The motivation for this work is related to a high-fidelity Large Eddy CFD Simulation (LES) [13] for aerospace applications that is performed iteratively in the context of a design space exploration: each run of this simulation produces a significant amount of data (several terabytes, depending on how frequently the status of the flow field is sampled) that should be processed in order to extract relevant knowledge and to drive the following iterations of the design exploration. This paper will discuss specifically how the analytics applied to the flow field data can be ported to a proper HPC infrastructure, as they are a good representative for a wide class of analytics problems. It should be noted that, while sec. 4 will report some scalability results achieved for this task on real data, this represents only a subset of the whole analysis of the CFD data. Details of the operations involved in each step are given below.

2.1 Analytics pipeline

Figure 1 represents the specific instance of the analytics problem at hand. It is possible to identify three main steps of the process: data ingestion, data reshuffling and an intensive floating-point computation (the matrix factorization). The execution of the three steps is also influenced (and sometimes dominated) by I/O operations on the underlying file system. Below are described in detail the three main steps of the analytics pipeline.

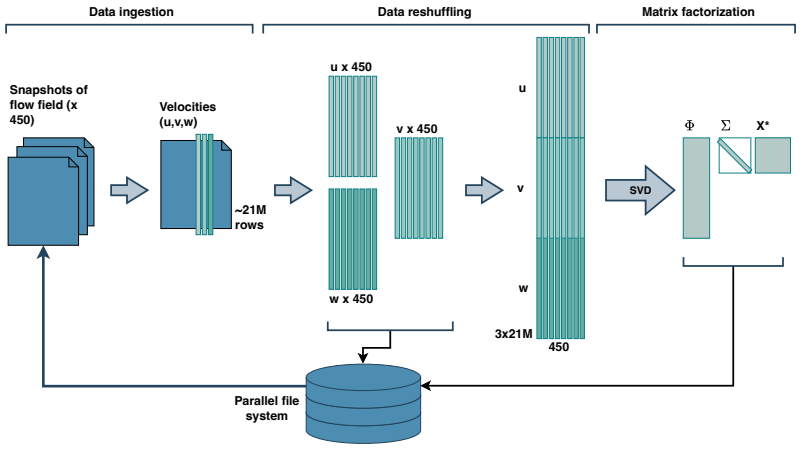

Fig. 1 Abstract representation of the data analytics problem considered in this work. $[u, v, w]$ represent the three components of the velocity of the flow field. Velocities are sampled periodically by the CFD simulation and dumped into CSV files (snapshots).

1. Data ingestion: this step involves reading a large number (450 in the specific instance of the problem, but potentially multiple thousands for production setups) of CSV files. The rows in the CSV files correspond to the mesh sites coming from a spatial discretisation of the instance's domain ($\sim 21M$ rows, totalling 3GB for each CSV file). From each file, the three components of the flow velocity are extracted and stored separately.
2. Data reshuffling: The velocity components extracted from the CSV files are stacked horizontally into 3 *tall-and-skinny* matrices containing each all the snapshots of one component of the velocity. The three matrices are then stored to disk, then they are stacked vertically to create a single tall-and-skinny matrix.
3. Matrix factorization: a Singular Value Decomposition (SVD) is computed on this matrix, in order to obtain relevant information about the simulation data. The singular values and the singular vectors are stored on disk.

This kind of pipeline captures different interesting aspects concerning HPDA problems; it is indeed worthy to note that the data ingestion and reshuffling is not as

trivial as reading several CSVs and appending them as additional rows, depending on the specific implementation, this may involve transposing data structures from *colum-major* to *row-major* ordering and vice-versa. The SVD is also a good benchmark for distributed matrix computations beyond the typical complexity $O(n)$ algorithms that are mostly used in the context of large-scale, distributed analytics [16]. The particular instance of SVD considered here is the so-called *tall-and-skinny* case, where the input matrix with M rows and N columns has $M \gg N$. Its complexity, that for the general case of the SVD is $O\left(M^2N + N^3\right)$, here is dominated by the $O\left(M^2\right)$ term [5].

From the perspective of computational challenges it should be noted that keeping in memory the full matrix of $\sim 63Mx450$ double precision floating point values needs more than 220GB of memory, and this represents a small instance of the problem. This goes beyond what is feasible on a typical workstation, and also beyond most conventional HPC nodes, meaning that an out-of-core approach is necessary to handle this problem (either spilling data on disk, or going distributed). Also leveraging big memory HPC nodes can be feasible, but this approach is still constrained by the size of available memory, which can become an issue when considering larger instances of the same problem.

2.2 Available Tools

The rationale of this work is to show how domain experts can leverage Python's open ecosystem of data science tools, to accelerate their engineering analytics pipelines on HPC infrastructures without having to focus too much on parallel computing aspects. In order to achieve this goal, investigation has been carried out to identify the suitable tools that could satisfy the following requirements:

1. Simple, textbook-like syntax for linear algebra and mathematics in general.
2. Capability to scale out computations without modifying the application logic.
3. Optional: capability to exploit GPUs for parts of the computation.

NumPy [6] and Pandas [15, 8] emerged in the last decade as *de facto* standards for array and data manipulation in Python; their simple APIs, abundant documentation and large user base satisfies the first requirement and provide a starting point to convert MATLAB code to Python. On the other hand, while both rely on fast numerical libraries like BLAS [4] and LAPACK [1] to perform numerical computations, none of them is natively able to distribute the computation and the data among multiple nodes. Hence, further investigation has been necessary to identify the right tool to extend NumPy and Pandas capabilities beyond the shared-memory domain.

2.2.1 Distributed computing with Python

The catalogue of tools to distribute computations with Python is quite large and spans the whole spectrum from low-level, message-passing tools (e.g., mpi4py[2]), to a host of task-based parallelism frameworks (e.g., Dask[3], PySpark[17], Ray[10], MARS[7], PyCOMPS[14]), up to extremely domain-specific libraries to parallelise specific tasks like distributed neural network training (e.g., ad-hoc facilities provided by Tensorflow and Pytorch, as well as Horovod[12]).

This work is not meant to be a comprehensive review of distributed computing tools for Python, hence a formal architectural analysis of the available tool is out of the scope of this paper. On the other hand, it is useful to report the results of the scouting performed in order to port the analytics pipeline described in sec. 3.1. From this perspective, the critical assumption was to be able to simply write the application logic with NumPy/Pandas, then the second requirement dictates that out-of-core scalability should be achieved with minimal modifications with respect to that implementation, ruling out all the low-level approaches that would require the application developer to both know the details of the analytics algorithm and to be familiar with concepts of distributed computing. However, among the tools listed above, some exist that advertise exactly the kind of seamless *distributed NumPy/-Pandas* functionality needed for this use-case.

Dask, PySpark and MARS all provide some flavour of this concept built on top of graph task-based parallelism. Among them, the combination of PySpark and MLlib [9] has already been tested in a previous work with a similar rationale [16] with promising results, however the complexity of the cluster setup and the features proposed by the other candidates driven the choice away from this option. In fact, Dask and MARS advertise direct interoperability of data structures (NDarrays and DataFrames) with NumPy and Pandas, along with simple cluster set-up and native python implementation. Between the two, Dask has been preferred due to the maturity of the project and the large amount of documentation available. While, the MARS project is definitely worthy of consideration, especially given the interesting architectural choices (i.e., an agent-based model instead of the master-worker scheduler used by Dask) that may provide a performance edge, the goal of this work is not necessarily ultimate performance, but rather the best tradeoff of performance and overall usability of the tools.

3 Proposed Methodology

3.1 Baseline implementation

Prior to this work, the pipeline was implemented with a MATLAB code making heavy use of the `spmd` construct to distribute the computation among workers living on multiple computational nodes. Figure 2 reports the process as implemented with

MATLAB: the CSV files are read by different workers in an `spmd` loop, then data is manually reshuffled in order to distribute rows among workers instead of columns. It is worth noting that the formulation with the eigenvectors of $M^T M$ is equivalent to the one described in the previous paragraph, using SVD. This implementation uses the HDF5[1] to store the intermediate matrices and the eigenvectors.

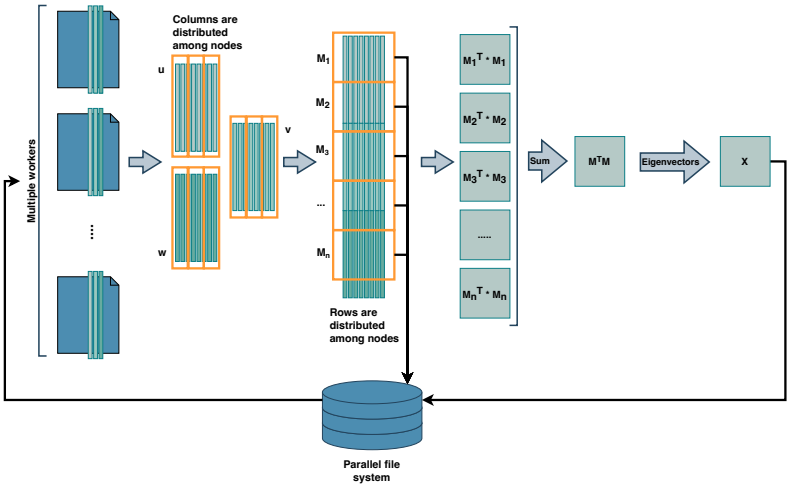

Fig. 2 MATLAB implementation of the analytics pipeline. Data is distributed among workers manually using MATLAB primitives.

3.2 Dask implementation

Since the original MATLAB code implementation of the application logic was heavily interleaved with code related to the parallel implementation (i.e., explicit segmentation of the data, handling of remainders when the data size is not multiple of the number of workers, etc.), the first step has been to re-write the abstract logic presented in figure 1 with NumPy and Pandas. This has been accomplished with minimal effort and represented the starting point for the Dask implementation, as well as a benchmark for the development effort to port sequential code. Before diving into the details of the proposed approach, it is worth to review the architecture of the Dask runtime.

[1] https://www.hdfgroup.org/solutions/hdf5/

3.2.1 Dask overview

Dask emerged as a tool to implement distributed task graphs with Python and, as such, maintained its task-based parallelism nature also when higher-level concepts like distributed NDarrays were included. Its conceptual architecture is depicted in figure 3: the user code runs as the *client*, which defines the task graph to be executed, and sends it to the *scheduler*, which in turn assigns individual tasks to the workers. Dask workers can also communicate with each other if the algorithm requires so, possibly leveraging high-performance MPI implementations and networking (e.g., Infiniband). The processes for the client, the schedule and each worker can be started

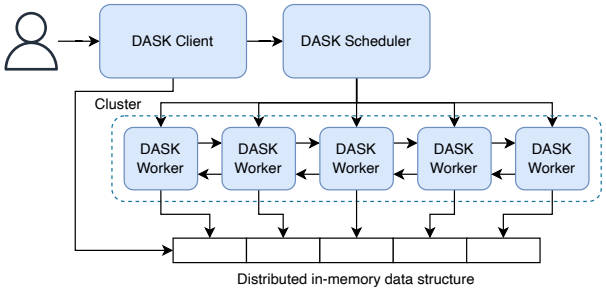

Fig. 3 Dask architecture.

individually via command line interface or by means of facilities provided by the Dask distributed package that can leverage cluster management tools (i.e., ssh, Kubernetes, SLURM, Docker, PBS and also plain `mpirun`). The rendezvous between all the components, if launched manually can be achieved by pointing to a JSON file created by the scheduler.

If the application logic can be described only using the native Dask data structures and their related primitives (e.g., `dask.array`, `dask.dataframe`), then the task graph is defined by Dask itself and it is completely hidden behind the scenes. Otherwise, the user can either access facilities for lazy evaluation of functions, built on the `dask.delayed` concept, or he can directly manipulate the task graph.

3.2.2 Dask cluster

This work involved running the HPDA pipeline on the Galileo100 HPC cluster provided by CINECA[2], composed by 528 computing nodes, each with 2 Intel Cascade-Lake 8260 CPUs with 24 cores each, and 384GB of memory. Some of the nodes were also equipped with 3TB Intel Optane storage configured as additional system memory. The cluster is managed by the SLURM job scheduler[3].

[2] https://www.hpc.cineca.it/hardware/galileo100

[3] https://slurm.schedmd.com

```
#!/bin/bash
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=12
#SBATCH <Other slurm options>

dask-scheduler --scheduler-file /path/to/scheduler/file &
srun dask-mpi --interface ib0 --nthreads 12 --memory-limit <
     memory limit> --scheduler-file /path/to/scheduler/file --no
     -scheduler --worker-class distributed.Worker &
python client_logic.py /path/to/scheduler/file
```

Listing 1 Batch script to launch Dask cluster on a SLURM-managed HPC facility.

To run Dask distributed on HPC facilities it is possible either to specify the options of the available job scheduler from the client code using Dask-jobqueue[4]. Otherwise, it is possible to launch the jobs using SLURM job scripts; the latter approach has been followed in this case, as it allows the re-use of existing job script configurations and gives more control on the deployment of the components. However, the configuration described in listing 1 deserves some clarification. The scheduler process requires orders of magnitude less resources than the workers, hence if we allow Dask-distributed to launch the scheduler along with the workers, it will count against the number of jobs defined by SLURM and it will cause uneven load balancing; in this case, the scheduler is launched as a standalone process and the option `--no-scheduler` is specified to prevent Dask to automatically spawn a scheduler alongside the workers. Using `dask-mpi` to launch the workers ensures that they will use a high-performance MPI library to communicate with each other, moreover, it is possible to force them to use the Infiniband network interface (`--interface ib0`). The memory limit for each worker should be defined taking into account the `--ntasks-per-node=4` option value by dividing the physical node memory by it. The scheduler file is needed to facilitate the rendezvous between the workers, the client and the scheduler: it is important to include its path in the client logic as a parameter, to ensure it can connect to the scheduler.

3.2.3 Implementation approach

The process in figure 1 requires ingesting a certain number of CSV files, which Dask is equipped to do out of the box (i.e., using `dd.read_csv('/path/*.csv')`), however, its API is only meant for appending homogenous rows to a single dataframe. In this case, the task involves extracting *columns* and appending them in order. This required a dedicated data ingestion process, that has been implemented as a `@dask.delayed` function to leverage parallelism when reading files from a parallel filesystem. The delayed function extracts the relevant columns into a NumPy ar-

[4] https://jobqueue.dask.org/en/latest/

ray that is then processed by the main as a `dask.array.from_delayed` object where the three velocities are split and aggregated in three separate `dask.array`. These arrays are then stored to the filesystem as *Parquet* files[5], allowing for parallel writing of the serialized data structure. The three matrices are then stacked vertically, and `dask.array.linalg.svd` is applied to compute singular values/vectors.

All these operations differ only slightly from the plain NumPy/Pandas counterparts: in fact, only the delayed-based data ingestion differs from the original implementation as a concept, while the Pandas code is almost identical.

4 Results

All the tests have been performed on the HPC cluster presented in section 3.2.2. Table 1 reports the results achieved with the presented implementation of the HPDA process presented in section 2.1. It is worth mentioning that the timing of the MATLAB implementation has been provided by the end-user as an order of magnitude comparison, but it has been tested on the same cluster as the rest of the tests performed. The NumPy/Pandas results refer to the fully sequential re-writing discussed at the beginning of section 3 and the tests have been executed on the Optane-equipped, large-memory nodes of the same Galileo100 cluster. From the numbers in

Table 1 Time required to complete HPDA process. - not performed or not applicable; *order of magnitude, reported for comparison.

Nodes	MATLAB (s)	NumPy/Pandas (s)	Dask (s)
1	-	3h 09m	-
2	1h 20m*	-	23m 30s
4	-	-	08m 44s
8	-	-	06m 02s
16	-	-	03m 44s

table 1, it is clear how the advantage of the Dask implementation vs. the MATLAB is not limited to the ecosystem or to the ease of use, but it is also related to the significant improvement of performance. This improvement can be related to several factors, with efficient parallel I/O one of the most impactful (i.e., faster read of CSV files, extremely fast write speed of Parquet file format w.r.t. the HDF5, which must be written sequentially).

Finally, figure 4 presents the scalability results achieved by the Dask implementation: the results are quite close to the ideal trend up to 16 nodes. Moreover, it is interesting to see how 4 nodes provide superlinear speedup with respect to 2 nodes: it is the author's opinion that the cause is related to the larger total amount of mem-

[5] https://parquet.apache.org

ory of 4 nodes (with constant problem size) that allows some Dask operation to avoid spilling to disk.

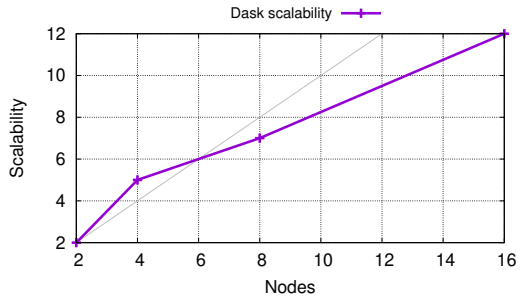

Fig. 4 Scalability of the Dask implementation. Grey line represents ideal trend.

5 Conclusion and future work

The approach and results presented so far provide a successful case-study in support of the idea of using Python tools to perform engineering data analytics on HPC infrastructures. In particular, Dask satisfied all the requirements mentioned in section 2.2, proving to be capable of delivering most of what its developers advertise: the considered NumPy/Pandas application could be ported to a distributed cluster with small (but still some) modifications to the code. As a bonus, the presented implementation outperformed significantly the original MATLAB code, while sporting good scalability up to the largest test set-up (16 nodes).

In future work, it is expected that two main lines of investigation will be built on top of this work: first, other steps of the CFD data analytics pipeline will be implemented with Dask in order to provide results that are more significant for the end users. Moreover, the feasibility and the efficiency of accelerating some parts of the pipeline on GPU will be investigated.

Acknowledgements This work has been supported by by the EuroHPC-02-2019 ACROSS project, grant agreement No. 955648 and by PRACE, which awarded access to the Fenix Infrastructure resources at CINECA, partially funded from the European Union's Horizon 2020 research and innovation programme through the ICEI project under the grant agreement No. 800858.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, third edn.

Society for Industrial and Applied Mathematics, Philadelphia, PA (1999)
2. Dalcín, L., Paz, R., Storti, M.: MPI for Python. Journal of Parallel and Distributed Computing **65**(9), 1108–1115 (2005). DOI 10.1016/j.jpdc.2005.03.010
3. Dask Development Team: Dask: Library for dynamic task scheduling (2016). URL https://dask.org
4. Dongarra, J., Du Croz, J., Hammarling, S., Hanson, R.J.: An extended set of fortran basic linear algebra subprograms. ACM Transactions on Mathematical Software **14**(1), 1–17 (1988). DOI 10.1145/42288.42291
5. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3 edn. Johns Hopkins University Press, Baltimore, MD, USA (1996)
6. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020). DOI 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2
7. MARS development team: MARS. URL https://docs.pymars.org/en/latest/
8. Wes McKinney: Data Structures for Statistical Computing in Python. In: Stéfan van der Walt, Jarrod Millman (eds.) Proceedings of the 9th Python in Science Conference, pp. 56 – 61 (2010). DOI 10.25080/Majora-92bf1922-00a
9. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M.J., Zadeh, R., Zaharia, M., Talwalkar, A.: Mllib: Machine learning in apache spark. J. Mach. Learn. Res. **17**(1), 1235–1241 (2016)
10. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., Stoica, I.: Ray: A distributed framework for emerging ai applications. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, p. 561–577. USENIX Association, USA (2018)
11. Rocklin, M.: Dask: Parallel computation with blocked algorithms and task scheduling. In: K. Huff, J. Bergstra (eds.) Proceedings of the 14th Python in Science Conference, pp. 130 – 136 (2015)
12. Sergeev, A., Balso, M.D.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 (2018)
13. Smagorinsky, J.: General circulation experiments with the primitive equations: I. the basic experiment. Monthly Weather Review **91**(3), 99–164 (1963). DOI 10.1175/1520-0493(1963)091<0099:GCEWTP>2.3.CO;2
14. Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R.M., Torres, J., Cortes, T., Labarta, J.: PyCOMPSs: Parallel computational workflows in Python. The International Journal of High Performance Computing Applications **31**(1), 66–82 (2017). DOI 10.1177/1094342015594678
15. The pandas development team: pandas-dev/pandas: Pandas (2020). DOI 10.5281/zenodo.3509134. URL https://doi.org/10.5281/zenodo.3509134
16. Viviani, P., Drocco, M., Aldinucci, M.: Scaling dense linear algebra on multicore and beyond: a survey. In: Proc. of 26th Euromicro Intl. Conference on Parallel Distributed and network-based Processing (PDP). IEEE, Cambridge, United Kingdom (2018)
17. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: A unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). DOI 10.1145/2934664