Accelerated Iterative Image Reconstruction for Cone-Beam Computed Tomography through Big Data Frameworks

Estefania Serrano[a], Javier Garcia-Blas[a,b,*], Jesus Carretero[a,b], Manuel Desco[b,c,d,e], Monica Abella[b,c,d]

[a] *Computer Architecture and Technology Group, University Carlos III of Madrid, Madrid, Spain*
[b] *Instituto de Investigación Sanitaria Gregorio Marañon, Madrid, Spain*
[c] *Dept. Bioingeniera e Ingeniera Aeroespacial, Universidad Carlos III de Madrid, Madrid, Spain*
[d] *Centro Nacional de Investigaciones Cardiovasculares Carlos III, 28911 Madrid, Spain*
[e] *Centro de Investigacion Biomedica en Red de Salud Mental (CIBERSAM), Madrid, Spain*

Abstract

One of the latest trends in Computed Tomography (CT) is the reduction of the radiation dose delivered to patients through the decrease of the amount of acquired data. This reduction results in artifacts in the final images if conventional reconstruction methods are used, making it advisable to employ iterative algorithms to enhance image quality. Most approaches are built around two main operators, *backprojection* and *projection*, which are computationally expensive.

In this work, we present an implementation of those operators for iterative reconstruction methods exploiting the Big Data paradigm. We define an architecture based on Apache Spark that supports both Graphical Processing Units (GPU) and CPU-based architectures. The aforementioned are parallelized using a partitioning scheme based on the division of the volume

*Corresponding author: fjblas@inf.uc3m.es
Phone: +34916249143
Postal Address: Departamento de Informatica, Universidad Carlos III de Madrid, Av. de la Universidad 30, 28911, Leganes, Madrid, Spain

and irregular data structures in order to reduce the cost of communication and computation of the final images. Our solution accelerates the execution of the two most computational expensive components with Apache Spark, improving the programming experience of new iterative reconstruction algorithms and the maintainability of the source code increasing the level of abstraction for non-experienced high performance programmers. Through an experimental evaluation, we show that we can obtain results up to 10× faster for *projection* and 21× faster for *backprojection* when using a GPU-based cluster compared to a traditional multi-core version. Although a linear speed up was not reached, the proposed approach can be a good alternative for porting previous medical image reconstruction applications already implemented in C/C++ or even with CUDA or OpenCL programming models. Our solution enables the automatic detection of the GPU devices and execution on CPU and GPU tasks at the same time under the same system, using all the available resources.

Keywords

Apache Spark GPU Medical image processing Computed tomography Iterative reconstruction algorithms

1. Introduction

X-Ray Computed Tomography (CT) is a medical imaging modality that obtains high-contrast projection images (radiographs) of anatomical structures based on the intrinsic differences in attenuation properties to X-rays of bone, air and soft tissue. A three dimensional volume is obtained by the combination of several radiographs of the patient taken from different angles in a process called *reconstruction*.

The growing concern on the radiation dose delivered to the patients has pushed the design of new acquisition protocols based on the reduction of the number of acquired radiographs. However, this data reduction results in a substantial decrease of the quality of the reconstructed images if conventional reconstruction algorithms are used [1]. Iterative algorithms allow to incorporate *a priori* information to compensate the lack of data at the expense of a higher computational cost.

The development of iterative algorithms poses two main difficulties. First, new clinical applications for CT, such as image-guided surgery and intensive care unit, require near real-time processing. This requirement poses the necessity of exploiting the performance of the employed computational systems. Second, algorithms deal with large data volumes and are computationally expensive, thus leading to the need of better hardware and software optimizations. And third, the evolution of the various hardware setups increases the effort required for maintaining and adapting the implementations to current and future programming models.

Most iterative reconstruction implementations are based on the repeated execution of two main components, namely *backprojection* and *projection*, which represent the largest computational cost. To deal with the need of higher computational resources when the size of the images increases, distributed versions of these components have been proposed [2, 3]. Nevertheless, programming, designing, and porting these components to distributed environments require specific expertise to properly exploit the underlying hardware and potential parallelism. The native programming languages employed in distributed environments are not as accessible for domain experts.

The importance of accessibility is perceived by the growing number of languages, programming models, and frameworks proposed in the last years. Programming models for Big Data, like Map Reduce, have recently been embraced in many scientific fields. The easiness of managing data in massively distributed environments, like clusters or cloud systems [4], has transformed many applications of the biomedical field, such as DNA analysis, protein discovery, and image processing, to the Big Data paradigm. The simplification of the programming functions and the abstraction of both data and task management make it accessible to a wider range of developers, not necessarily skilled in low level programming.

In previous works [5, 6], we presented a preliminary study of a Python-based implementation of the *backprojector* component. This paper extends those works by presenting a novel approach for adapting iterative image reconstruction algorithms for low-dose CT to work on a Big Data framework using accessible programming languages without performance degradation. We focus on a specific programming language, Python, and a specific framework, Apache Spark.

Our approach, although similar to those presented in [7, 8], aims to be a more general solution for different iterative algorithms with support for legacy C/C++ code, new Python code, and CUDA kernels. With respect to the support of GPUs, this work does not require modifications in the base framework in contrast with [9, 10], making this approach accessible to a large variety of clusters or clouds environments.

We include a complete description of our heterogeneous solution, the implementation of the *projector* component, the integration of CPU-based and GPU-based architectures, and a wide evaluation for both architectures.

The rest of the paper is organized as follows. Section 2 describes the problem. Section 3 presents the CPU and accelerated architectures presented in this paper. Section 4 shows the evaluation experiments executed and the results obtained. Section 5 shows works related to this paper. Finally, Section 6 presents the discussion and major conclusions of the paper.

2. Problem description

The aim of this section is to briefly introduce the necessary concepts to facilitate the understanding of this paper and its complexity.

The objective of CT reconstruction algorithms is to obtain tomographic 3D images, also referred here as reconstructed volumes, from radiographs, or projections, taken from different angles of the sample. An example of different projections is shown in Figure 1 as well as two of the reconstructed slices that compose the three-dimensional volumes.

The reconstruction process can be defined by different algorithms. Depending on the quality of the input data and the required quality, the computational complexity of these algorithms can significantly increase. This is the case of the iterative reconstruction algorithms addressed in this paper.

Most iterative reconstruction algorithms are based on the repetition of the *backprojection* and *projection* operators over images that are gradually enhanced on every iteration until the desired image quality is achieved. Those components represent the most expensive computational steps of the algorithm and are normally optimized by using parallel, distributed, and heterogeneous architectures. Those components are described below.

2.1. Projection

Projection emulates the process of data acquisition in an X-ray scanner given an initial 3D volume and the geometry of the system (i.e. source-to-detector distance and angular positions) based on the computation of diverse trajectories of the X-rays from the source to the detector panel traversing the 3D volume.

The *projection* component presented in this work relies on the ray-driven interpolation approach, in which the line integral (the final pixel value) is

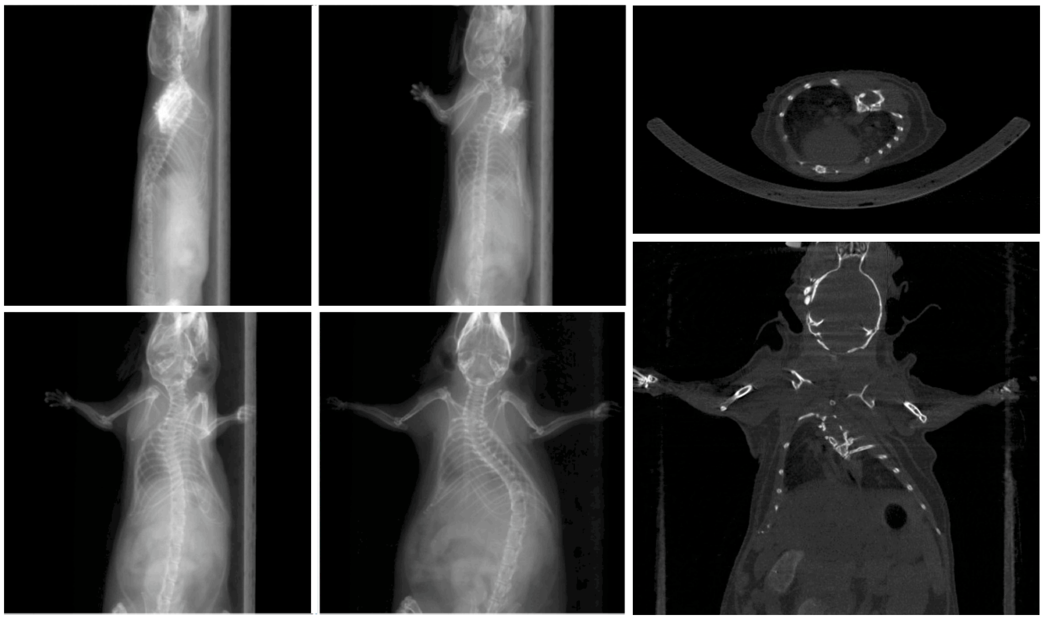


Figure 1: Left: Projection data of a rodent study for angles 1, 25, 45 and 90 degrees. Right: Axial and coronal views of the reconstructed image.

based on the computation of the sum of $Nstep$ values along the X-ray to update the contribution to the detector pixel, as shown in Equations 1, 2 and 3.

$$proj_{\theta}(x,y) = step \times \sum_{vol_i=-\frac{rad}{step}}^{\frac{rad}{step}} \frac{1}{cos\alpha} \times vol(\frac{1}{Mag} \times \\ \times cos\theta + vsin\theta, -\frac{1}{Mag} \times sin\theta + vcos\theta, \frac{1}{Mag} y) \quad (1)$$

where rad is the maximum radius of the field of view (FOV) (in mm), $f(u, v, z)$ is the voxel value in the sample at coordinates (u, v, z), $p\theta(x, y)$ is the projection data for position (x, y) in the detector at angle θ, α is the angle of the ray with respect to the central ray of the beam, and Mag is the magnification due to the cone angle, given by Equations 2 and 3.

$$x = arctg\frac{\sqrt{x^2 + y^2}}{DSO + DDO} \tag{2}$$

$$Mag = \frac{DSO + v}{DSO + DDO} \tag{3}$$

where DSO and DDO are the distance from the center of the FOV to the source and the detector, respectively. Sampling is performed along the v-axis given by *step* (in mm), which is set by default to the minimum dimension of the pixel, covering $2 \times rad$.

2.2. Backprojection

Backprojection builds a 3D volume from different projections or radiographies obtained from the scanner or from the *projection* component. The *backprojection* algorithm implemented in this work follows the voxel-driven approach. Voxel-driven interpolation implies integrating all the intersecting rays, going from the source to the center of each voxel. The mathematical formulation of our implementation of the component can be seen in Equations 4 and 5.

$$\begin{multline} vol(u, v, z) = \Delta\theta \times \sum_{\theta=ini}^{ini+nproj} proj_\theta(Mag \times [ucos\theta- \\ - vsin\theta], Mag \times z) \tag{4} \end{multline}$$

where ini is the initial projection angle, $nproj$ is the total number of projections, $vol(u, v, z)$ is the value in the backprojected volume at coordinates (u, v, z), $proj_\theta(x, y)$ the projection data for position (x, y) in the detector at angle θ, $\Delta\theta$ the step angle in radians, and Mag the magnification due to the cone shape of the beam given that:

$$Mag = \frac{DSO + usin\theta + vcos\theta}{DSO + DDO} \tag{5}$$

where DSO (Distance Source Object) and DDO (Distance Detector Object) are the distance from the center of the FOV to the source and the detector, respectively.

3. Heterogeneous architecture for iterative CT

As described in Figure 2, our solution is fundamentally based on the offload of the computation of the most computationally expensive components, i.e., *projection* and *backprojection*, to a distributed environment, being compatible with heterogeneous resources containing CPU and GPU-based nodes.

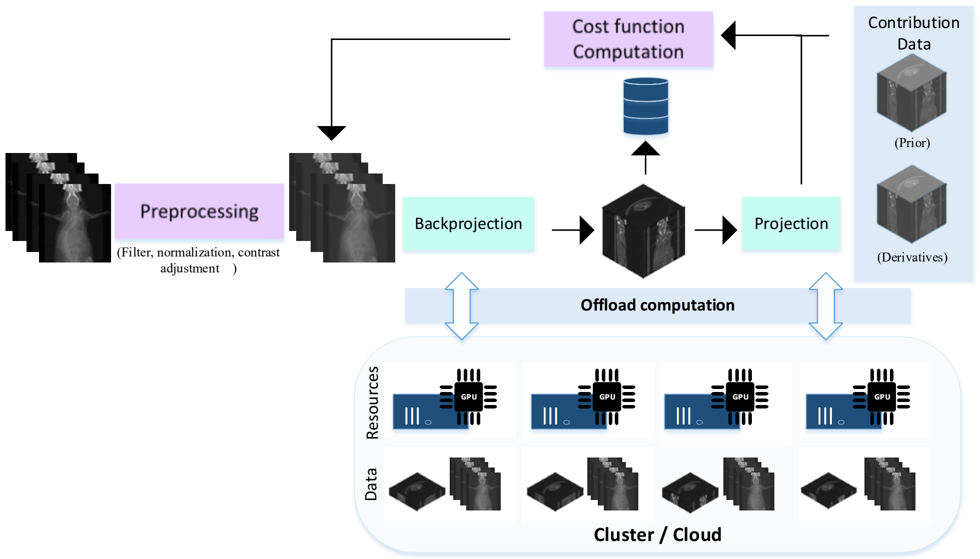


Figure 2: Diagram of a generic iterative reconstruction algorithm with accelerated analytical components: backprojection and projection. Iterative reconstruction algorithms consist of multiple phases in which different contributions are computed. Prior information of the reconstructed object can be included in the form of surface models or previously reconstructed volumes.

Our proposed architecture for Apache Spark, shown in Figure 3, includes the presence of a master process (*Driver*) that manages the execution of the different steps of the application and several workers in charge of executing the programmed tasks that compose the application, as in most Big Data frameworks. Additionally we enable the support of GPUs inside the *Worker nodes*.

The *Driver* component manages the application context and is in charge of communicating with the resource manager for the acquisition of computational resources from the Spark cluster. *Worker nodes* are in charge of executing the programmed tasks in different executor processes. An additional Intra-Scheduler is added to each worker node in order to manage the accelerator devices and assign the execution of these tasks to a specific GPU.

Our main components (*Backprojection* and *Projection*) are executed inside Worker nodes, either using the CPU or the GPUs, if present.

An explanation of the execution flow of an application in our architecture (with/out GPUs) can be found in Figure 4. Data (projections or volume) are distributed over all the nodes, assigning to each node one or more partitions of the data, on which each executor will work independently. The work is performed during the *map* stage of the application, in which data are transformed. *Projection* and *Backprojection* are considered the transformations of the volume and projections, respectively. Final results can be obtained by *reducing* the data obtained from the transformation of each partition (action).

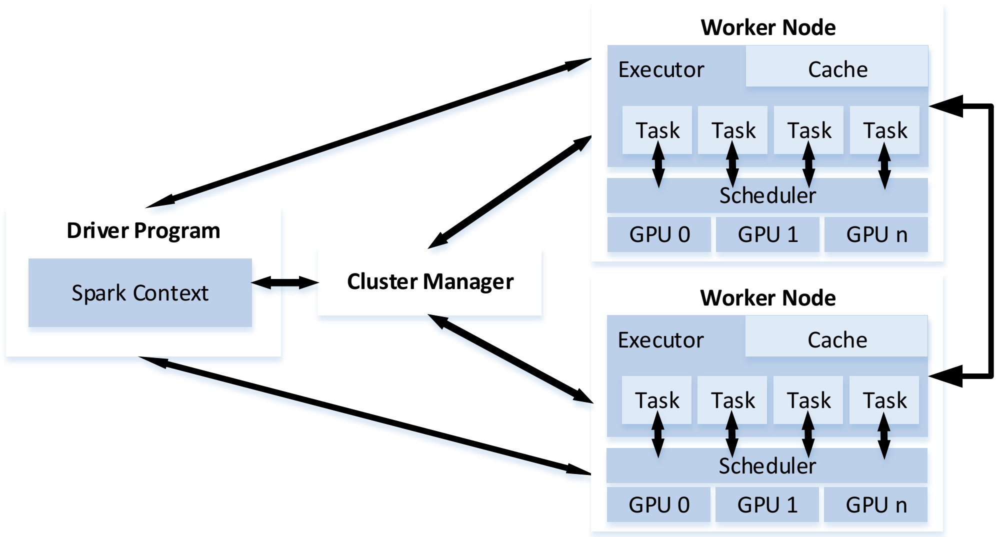


Figure 3: Extended architecture of Apache Spark including support for multi-GPU compute nodes. Tasks acquire/release an available GPU orchestrated by a intra-node scheduler.

The suitability of this architecture for both components derives from the easy parallelization of the ray computations, which are independent, and the automatic division of work and data management provided by the framework.

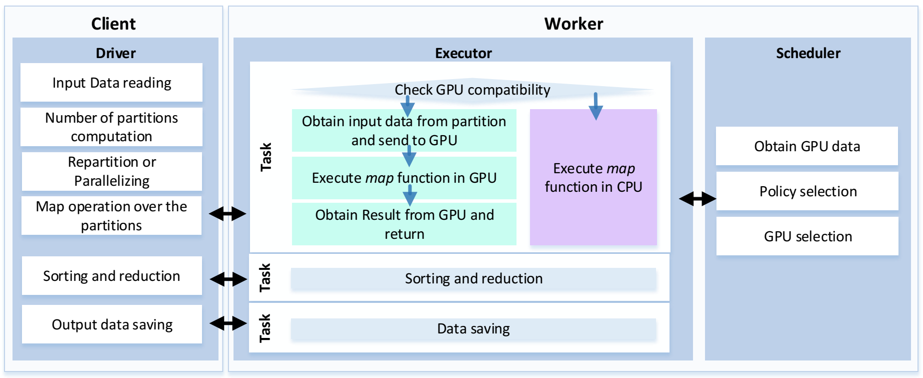

Client
Driver
Input Data reading
Number of partitions computation
Repartition or Parallelizing
Map operation over the partitions
Sorting and reduction
Output data saving
Worker
Executor
Task
Check GPU compatibility
Obtain input data from partition and send to GPU
Execute map function in GPU
Obtain Result from GPU and return
Execute map function in CPU
Task
Sorting and reduction
Task
Data saving
Scheduler
Obtain GPU data
Policy selection
GPU selection

Figure 4: Execution of an application in our accelerated architecture.

3.1. Data partitioning

For an efficient parallelization of the computation, suitable partitioning schemes for each component must be found. How the data is partitioned influences the execution of the application at several stages, from the total time spent in each task to the amount of communication between different nodes.

3.1.1. Backprojection

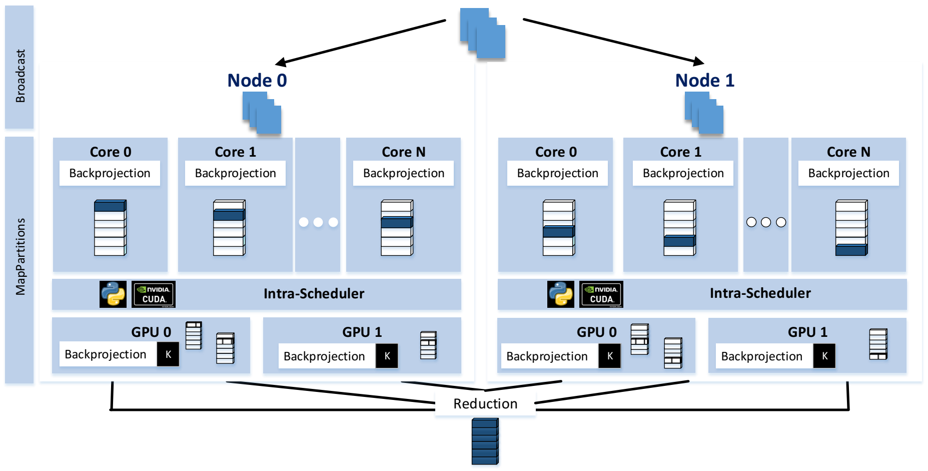


Figure 5: Execution and partitioning of the Backprojection in our proposed schema.

Inside the *backprojection* component, the *map* stage corresponds to the computation of the geometry of each independent ray. Since there is only one ray per voxel, we have $dimu \times dimv \times dimz$ *map* operations corresponding to the dimensions of the volume in the three different axes. The transformation function is defined as:

$$vol(u, v, z) \leftarrow \sum_{\theta=ini}^{ini+nproj} proj_{\theta}(x, y) \tag{6}$$

where each point in *proj* is obtained using Equation 4.

The volume is divided into subvolumes (partitions) along the z axis. As we can see in Figure 5, the *backprojection* task is executed per partition. This division is represented inside the framework as an ordered array with the position of each partition inside the complete volume. Thus, partitioning is performed at the output (volume), not at the input (projections). This strategy implies that input data must be transferred to all worker nodes through a *broadcast* operation inside Apache Spark.

3.1.2. Projection

In this case, the partitioning of the computation and data is also based on dividing the volume into smaller subvolumes, which are now the input data of the *map* operations. Therefore, in the *projection* task there is no need of a previous broadcast of data since each executor already possesses his partition, thanks to the functionality provided by Apache Spark RDDs (Resilient Distributed Dataset). The number of *map* operations is equal to $dimx \times dimy \times nproj$ corresponding to the number of projections and their dimension. The transformation function is described as:

$$proj_{\theta}(x, y) \leftarrow \sum_{v=0}^{depth} vol(u, v, z) \tag{7}$$

where the points in *vol* that contribute to each point in *proj* are obtained following Equation 1.

As shown in Figure 6, every core computes all projections from the corresponding subvolume. To reduce the computational load for each subvolume, the projection task only computes the specific rows of the projection data corresponding to each subvolume. Each row is identified by a key computed based on the projection angle (θ in Equation 7) and its spatial position inside the complete projection. Due to the cone geometry, multiple subvolumes can contribute to the same row of the projection thus having tuples containing the same key.

Then, when every projection task has been completed, a *reduction* of all the projection rows is executed over the tuple (key and its corresponding projection row).

The reduce stage does not require additional functions due to the native implementation of element-wise additions in Numpy arrays. Although this approach requires to have a temporal copy of all projections in each executor and it may seem memory expensive, it is important to highlight that low-dose CT techniques normally imply the use of a lower number of projections. Therefore, this replication does not create a high overhead in terms of memory usage.

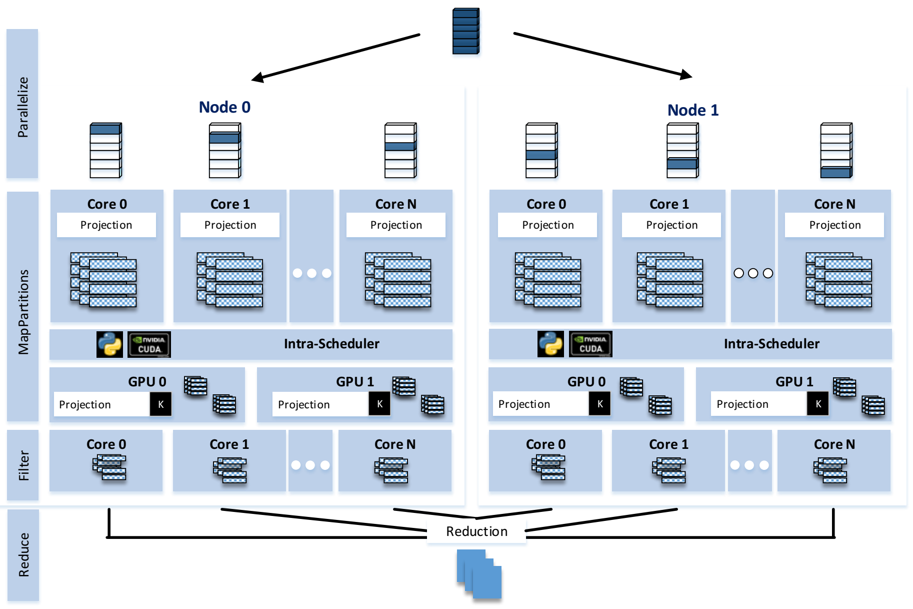


Figure 6: Execution and partitioning of the Projection in our proposed schema.

3.2. *Accelerated Architecture for GPU-based Big Data Computing*

With our approach, Apache Spark with GPUs can be used without any modification. The main addition to the architecture is the inclusion of a novel Intra-Scheduler that will be executed in each *Worker* node. This Intra-Scheduler orchestrates the different executors, offloading computation into the GPUs. Using different policies (Round-robin, Random, Least Processes) [6], the Intra-Scheduler decides which tasks are executed in the GPUs, taking into account the number of devices that are installed inside the node and the memory needs of the task to be executed. Additionally, the Intra-Scheduler collects information of the load of each available GPU. If no GPU is available (due to the limited memory), it stalls the execution of the task or sends it to the CPU if possible, until there is space available. Device management is done inside the Apache Spark *task*. The implementation of the function executed inside the task (*projection* and *backprojection* components) can be made using CUDA native kernels or using PyCUDA methods. Therefore, the execution of the mapping in the heterogeneous architecture consists of five phases:

1. Device acquisition: executor communicates with the Intra-Scheduler to obtain a device in which the function should be executed. Based on the acquired device, the context for that specific GPU is created.
2. Transfer of the input data onto the device memory: using the PyCUDA API, it is possible to transfer the memory containing the input data to the acquired device.
3. Execution of the CUDA/PyCUDA kernel: the kernel is loaded, compiled on-demand, and executed with the required parameters passed to the function.
4. Transfer of the output data to the host memory: to be able to obtain the final data, each executor should return its output stored in the device memory to host memory for Apache Spark to be able to manage it. The output data generated in the device is transferred to the host memory before finishing the task.
5. Device release: executor requests the Intra-Scheduler to release the device and the context for that device is destroyed.

The communication with the Intra-Scheduler, provided through RPCs, is implemented with the RPyC package [11].

3.3. CPU-Based Execution Model Architecture

We have implemented two alternatives to take advantage of the available parallelism in CPU-only systems:

- To exploit the parallelism already provided by the platform.

- To incorporate additional parallelism by using native programming models, widely employed in clusters and supercomputers, like OpenMP.

For both alternatives, we use a native programming language, C, due to its higher performance over interpreted programming languages (i.e., Java, Scala or Python). Since Apache Spark lacks of support for native programming languages, we take advantage of the close relation between Python and C, which allows the invocation of specialized functions implemented in this language using the provided Python headers.

The architecture of the framework (Figure 3) as well as the execution flow is the same regardless of the aforementioned alternatives. However, there are differences regarding the execution setup:

- Apache Spark-based parallelism: this alternative is the most widely used in the community. We evaluate this alternative as it takes advantage of the possibility of using more cores per executor in the node and provides straight-forward parallelization when running the reconstruction algorithm. In general, to obtain full parallelism, one executor per core would be ideal. However, since executors are launched independently with a decoupled memory space, there would be a need of larger resources on Worker nodes. When configuring the execution, we balance the number of executors per node and the cores per executor, complying with the trade-off between memory usage and resources exploitation.

- Native-based parallelism: In this case, to parallelize the algorithm, we rely on OpenMP for our *backprojection* and *projection* components, in which we parallelize each ray with a different CPU core. Memory footprint also increases, as in the previous alternative. Nevertheless, in this case the number of executors needed is lower, resulting in a reduction of the memory requirements. This approach is based on a two level parallel strategy managed by two different techniques: coarse-grained parallelism at a distributed level using Apache Spark (external parallelism) and fine-grained parallelism inside the node using OpenMP (internal parallelism).

In both cases, an invocation of a C module is introduced inside each task. This C module contains the proper *map* function in which each element is transformed, returning the corresponding projection or volume. Data are not copied between Python and C since the reference is valid for both languages, saving memory and execution time. Python tasks are therefore only responsible of setting the parameters of the C module. Inside the C module, Python parameters are transformed into C variables and the algorithm is executed with or without OpenMP depending on what alternative is chosen. After the algorithm is executed, the resulting data is returned to Python with the correct format, in this case, a Numpy array containing the corresponding subvolume or projections.

4. Evaluation

We evaluated our solution using two environment setups:

- CPU-based cluster: 8 nodes with Intel(R) Xeon(R) CPU E5-2603 v4 (12 cores), 126 GB of DDR4 RAM, and 256 GB of SSD for temporary storage. The Apache Spark driver was launched in a separated node with two Intel(R) Xeon(R) CPU E5-2630 v3 processors (12 cores) and 252 GB of RAM.

- GPU-based cluster: 2 nodes with an Intel(R) Xeon Phi(TM) CPU 7210, 148 GB of RAM, 256 GB of SSD for temporary storage. Each node has 2 NVidia GTX 1070 installed, each with 8GB of GDDR5 memory. The Apache Spark driver was launched in a separated node with two Intel(R) Xeon(R) CPU E5-2630 v3 processors (12 cores) and 252 GB of RAM.

Both systems are supervised through Cloudera 5.13 over Ubuntu 16.04. We used Apache Spark version 2.2 in *stand-alone* mode. For the distributed evaluation, Apache Yarn 2.6, with the *FairScheduler*, was used as resource manager. The input files were stored in a local SSD and the result files were stored in a HDFS running on top of SSDs disks and a 10 Gbps Ethernet network. The Python version was 2.7, complemented with PyCUDA 1.3 and CUDA 9.0 for the heterogeneous architecture and a C/C++ module compiled with GCC 5.1 and OpenMP for the homogeneous architecture. Each result was obtained as the average of three consecutive executions.

For both *backprojection* and *projection* components we have worked with volume data of 1024^3 voxels and 360 projections of size 1024^2. This number of projections is higher than the one normally acquired in low-dose CT, therefore representing a worst case in performance. Input data is read from a SSD in the Driver program, requiring first a partitioning of the data.

To better evaluate all possibilities we used two configurations:

- *Configuration A*: single core per executor. Parallelization is done internally either by using OpenMP in the CPU based approach or with the GPU cores.

- *Configuration B*: 5 cores per executor. Parallelization in the CPU-based approach is done at external level without further usage of other parallel programming models. In the case of GPU-based execution, additional parallelization is introduced with the usage of the multiple GPU cores.

The executors can be assigned to the same node, provided that it has enough memory capacity.

4.1. CPU-based Architecture Evaluation

We executed both *backprojection* and *projection* components, with configurations A and B, in the CPU-based cluster.

In Figure 7, we plot the evaluation results of the *backprojection* component employing *configuration B*, which can execute 5 tasks at the same time per executor. We explore the results given a varying number of executors and partitions. As we increase the number of executors, we note the benefits of having a number of tasks larger than the number of executor threads. This is due to the better management of the imbalances between nodes when smaller tasks are scheduled. This effect can also be appreciated when employing OpenMP , as shown in Figure 8.

In the case of the *projection* component, the results obtained for *configuration B* depict a significant increase in the execution time when the number of partitions is incremented (as shown in Figure 9). In this case, a higher number of partitions does not decrease the computational size of the task, due to the discrepancies between the number of slices in a volume and the FOV projected. Because of this reason, the increment of parallelism using Apache Spark can degrade the performance of this component. Nevertheless, it is important to note that this growth is sublinear, as the execution time does not increase at the same rate as the number of partitions.

The *projection* performs better in *configuration A* than in *configuration B*, in contrast with what happens with the *backprojection*, as we can observe

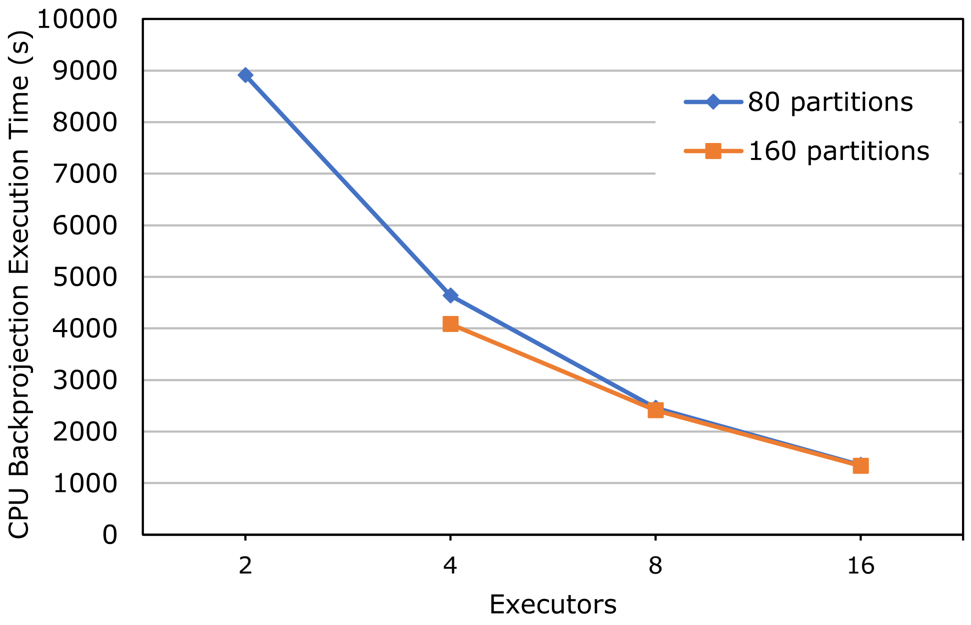


Figure 7: Overall execution time of the *backprojection* component on a CPU-based cluster varying the number of executors and partitions without using OpenMP (Configuration B).

when comparing Figure 10 and Figure 9. However, in *configuration A* the increment in the number of partitions does not positively affect the overall performance of the application, similarly to what we have seen in the case of the *backprojection* component. Considering the overall execution time, the number of partitions required when using OpenMP is lower than the one required in *configuration B*.

4.2. GPU-based Architecture Evaluation

We have also evaluated the performance of the Apache Spark/GPU-based architecture based on our proposed Intra-Scheduler, varying the number of executors and partitions. Given that the maximum number of GPUs present in the system is four, we evaluated up to four parallel executors over different number of cores. This decision of increasing the number of cores per executor allows us to exploit the available concurrency, given that modern NVidia GPUs enables the execution of multiple concurrent kernels if enough resources are available (mainly, computing units and internal main memory).

Figures 11 and 12 show the evaluation of the *backprojection* component in the GPU-based architecture. For both configurations we observe that beyond

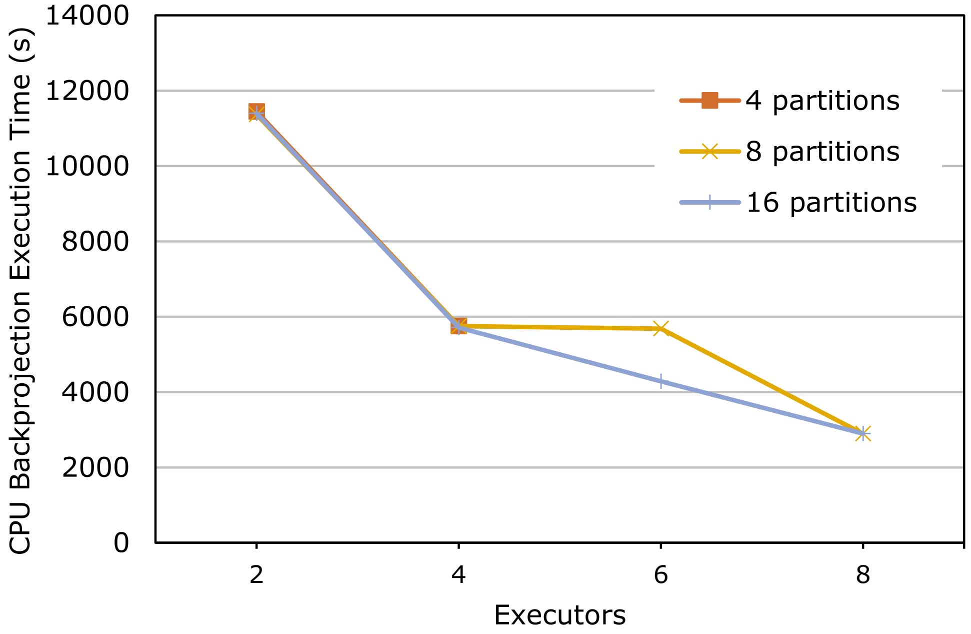


Figure 8: Execution time of the *backprojection* component on a CPU-based cluster varying the number of executors and partitions using OpenMP (Configuration A).

100 partitions (4 executors), the overall execution time increases for both cases. Execution times do not differ between configurations when executing with similar conditions (around 200 seconds for one partition per thread and 4 Executors and also around 200 seconds for one partition per executor and 4 executors), which indicates that the limiting factor is the computing power of the GPU device.

Furthermore, in case of *projection*, Figures 13 and 14 show that there is a significant difference between both configurations in the GPU based approach. Executing the algorithm with 4 executors employing *configuration A* results in an execution time of around 1,000 seconds, meanwhile with 4 executors and one partition per core in *configuration B* the algorithm takes around 700 seconds. The number of tasks in the second case is higher, 20 tasks for 4 executors vs 4 tasks, in contrast with the previous observations for the *projection* component in the CPU-based cluster. However, as seen in Figure 9, the number of tasks from which we can perceive a significant increase in time is 40. Being 20 tasks, a lower number than this threshold and considering the benefits of executing smaller tasks for better balance

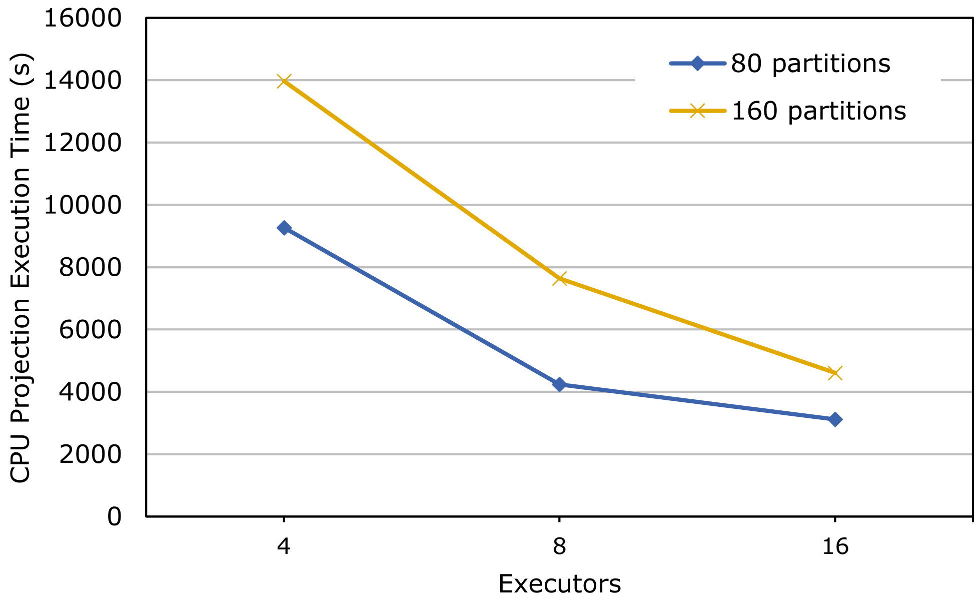


Figure 9: Execution time of the *projection* component on a distributed CPU-based cluster for a different number of executors and partitions using 5 cores per executor (Configuration B).

the load between the GPUs, it is reasonable that *Configuration B* performs better than *Configuration A* in this case. The drawback detected in the CPU-based approach is also reproduced when employing GPUs with a much higher number of tasks. The decrease of performance starts with a lower number of partitions than in the case of the *backprojection*, since in *projection* a greater number of partitions also increases the amount of work that is executed. For the same amount of work (same number of partitions), both configurations scale out almost perfectly when increasing the number of executors.

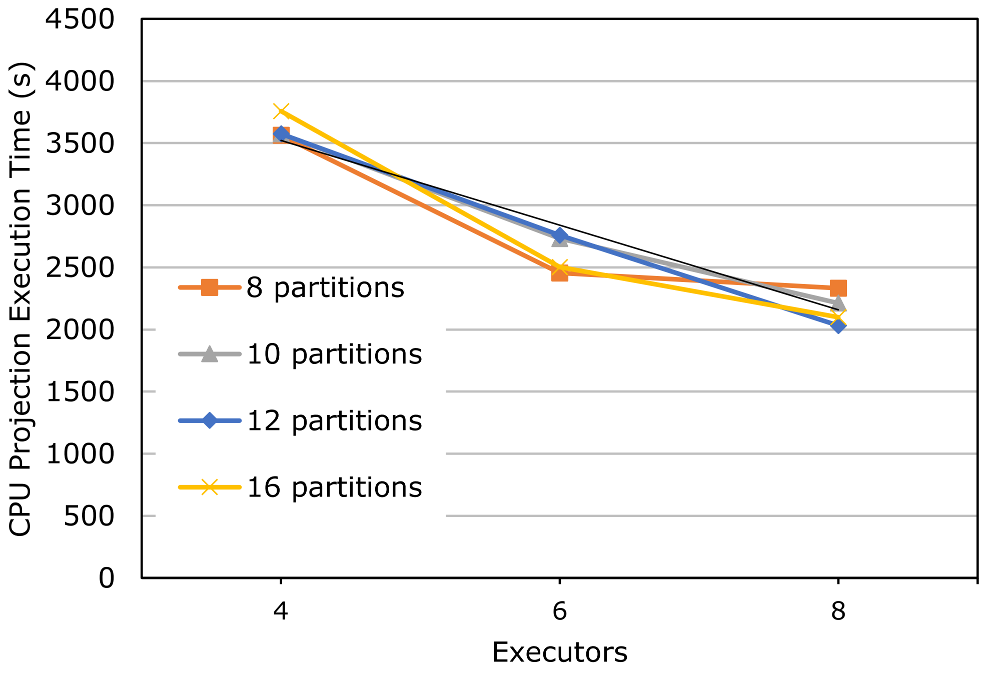


Figure 10: Execution time of the *projection* component on a distributed CPU-based cluster for a different number of executors and partitions using OpenMP in each executor (Configuration A)

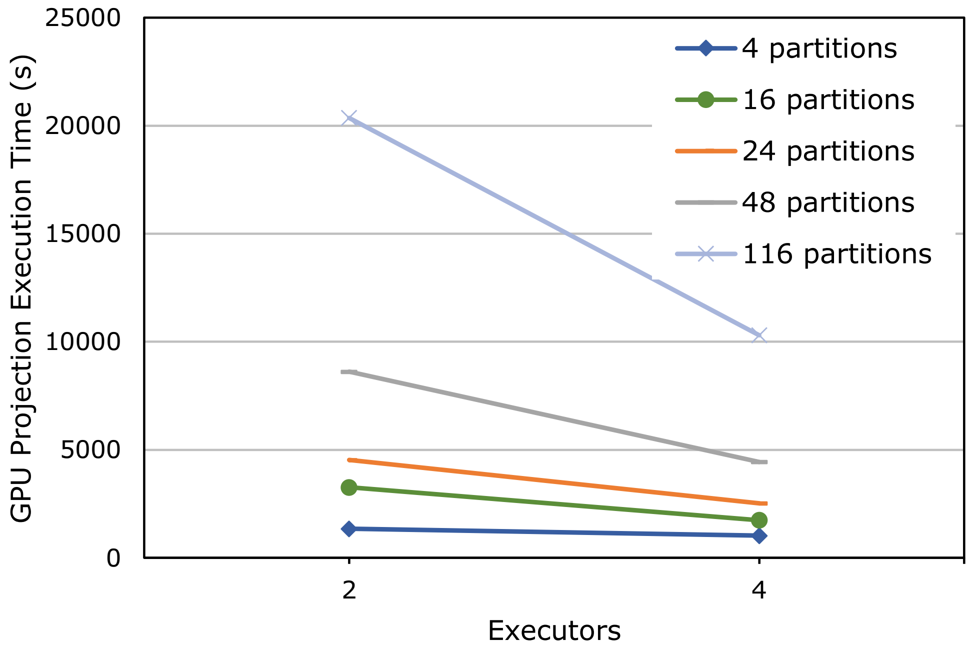


Figure 13: Execution time of the *projection* component on a distributed GPU-based cluster for a different number of executors and partitions (Configuration A).

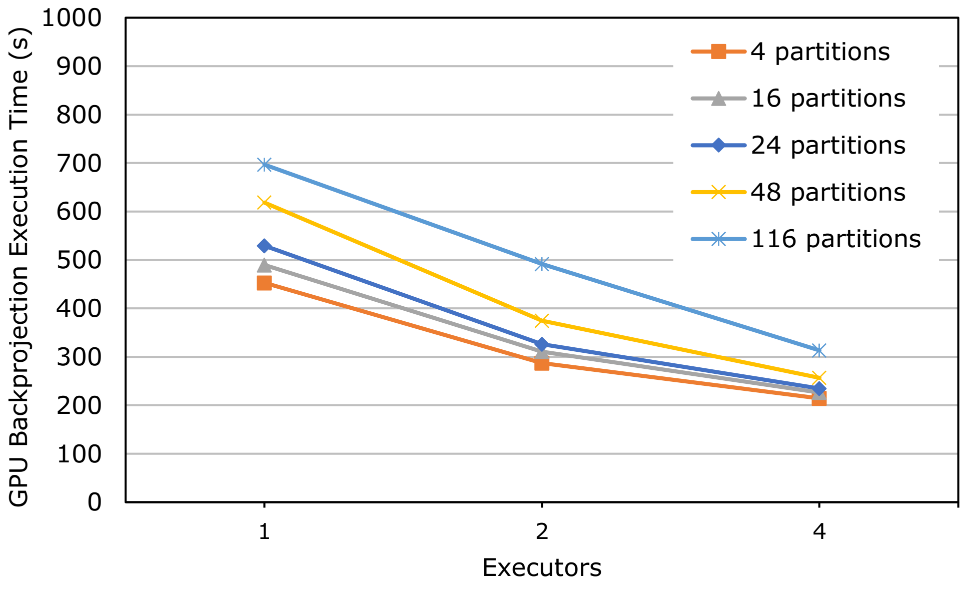


Figure 11: Execution time of the *backprojector* component on a GPU-based cluster for a different number of executors and partitions (Configuration A).

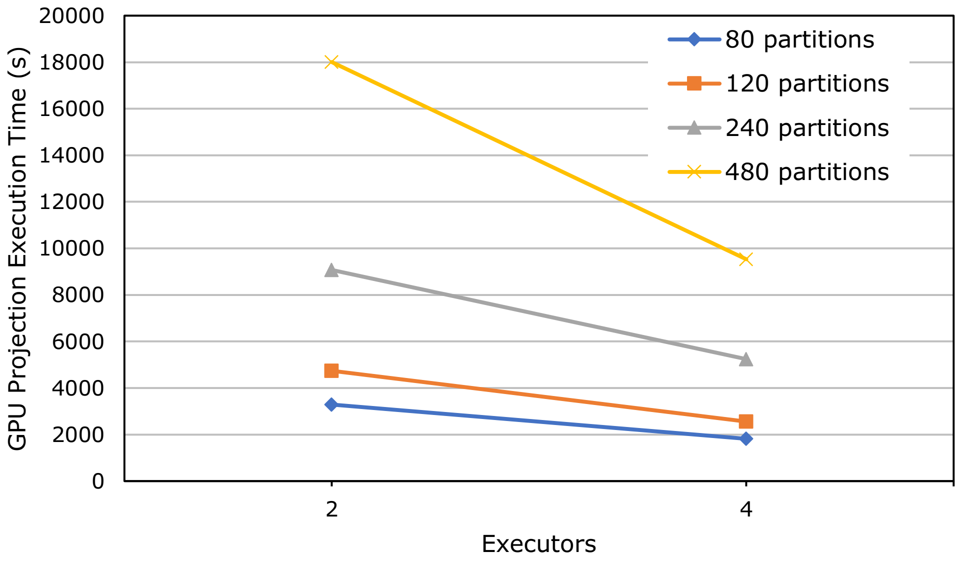


Figure 14: Execution time of the *projection* component on a GPU-based cluster for a different number of executors and partitions with 5 cores per executor (Configuration B).

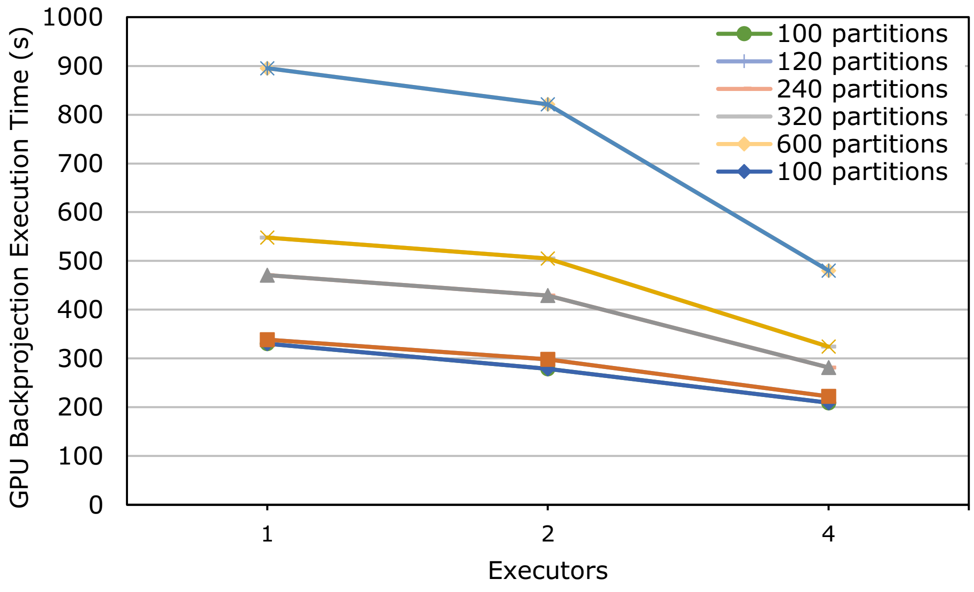
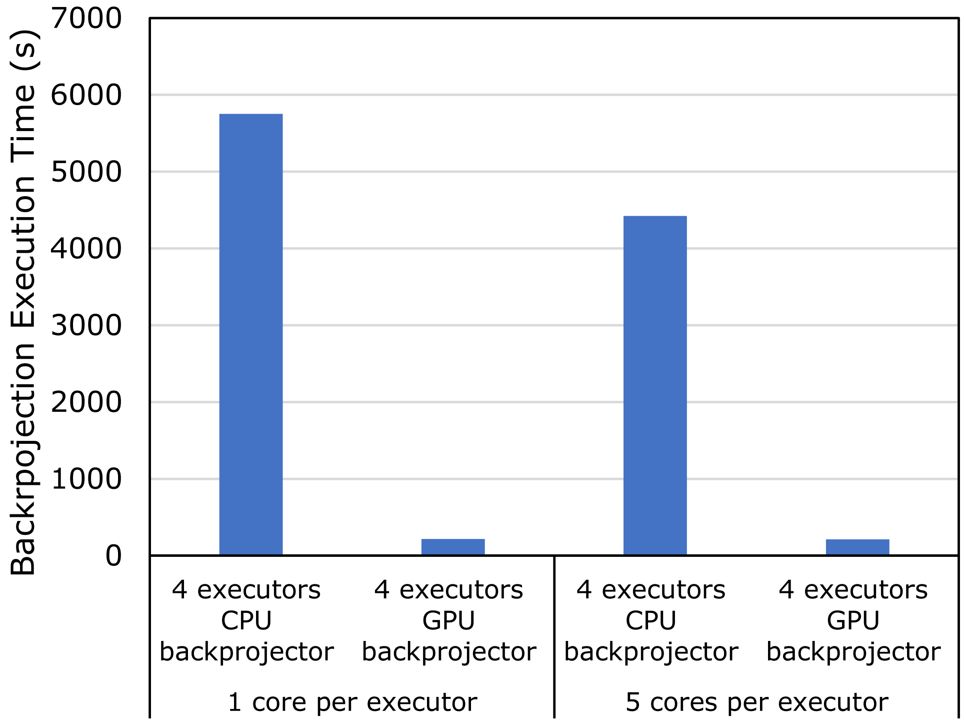
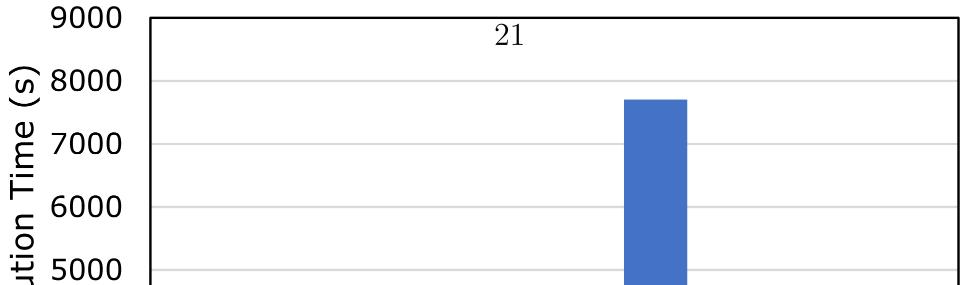


Figure 12: Execution time of the *backprojector* component on a GPU-based cluster for a different number of executors and partitions with 5 cores per executor (Configuration B).





4.3. Image quality

We have evaluated the effect of our partitioning scheme on the final results using simulations with the Digimouse phantom[1]. We performed a *backprojection* followed by a *projection* with 1, 8 24 and 48 partitions, with 360 projections of 512^2 pixels, resulting in a volume of 512^3 voxels. The Root Mean Square Error (RMSE) in the projections with respect to the 1 partition case is significantly low (under $1.0e-4$) for all partition sizes. The small errors, not noticeable with visual inspection, appear in the intersection between the chunks due to the reduction stage in projection, as shown in Figure 16 (the difference images are shown with a narrow window so the small errors are noticeable). No effect was observed for the *backprojection* step.

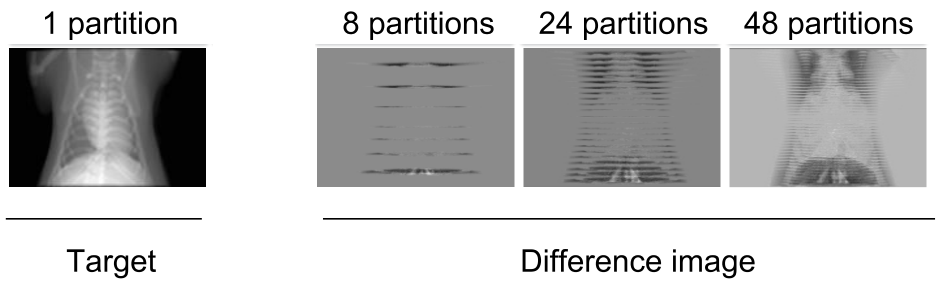


Figure 16: Projection obtained with one partition and different image (subtraction from the one partition case) corresponding to 8, 24 and 48 partitions setups.

5. Related work

Iterative reconstruction algorithms have normally taken advantage of most traditional parallel environments [12, 13] as well as of the use of GPUs [14, 15, 16, 17, 18, 19] at a non-distributed level due to their increasing popularity and reduced price. In many cases they have followed a similar approach focusing on the adaptation of the main two components (i.e., *projection* and *backprojection* in the case of iterative reconstruction) to the underlying architecture, even taking into account low level details such as exploiting the different caches of the device [20]. The use of GPUs has also been expanded

[1]http://neuroimage.usc.edu/neuro/Digimouse

to the distributed level in an HPC environment in works such as the one presented by Harley et al. in [21], where the authors presented a work on image analysis on a cooperative cluster of GPUs and multi-cores, with the coexistence of multiple execution modes.

However the use of these distributed environments is not available for everyone. Last years have shown a trend of enabling the use of traditional programming models for parallel execution, typically employed in High Performance Computing (HPC) environments to non HPC programmers, with more accessible paradigms. IN this context, we can find in the literature a large variety of solutions using HPC and Big Data paradigms for the medical image processing field, with good results in terms of execution time [22]. Optimized distributed implementations of image processing tools and reconstruction algorithms can also be found employing either the MapReduce programming model using Hadoop [23, 24, 25, 26] or MPI [3, 27], with the idea of exploiting massive parallelism. An example of fast tomographic image reconstruction via large-scale parallelization combining a MapReduce-like computing middleware with MPI can be found in [28]. However, most of those works using MapReduce do not obtain an acceptable scalability due to limitations in the MapReduce frameworks [29]. To address this scalability problem, some works have proposed the exploitation of GPUs in the Apache Hadoop framework [30, 31, 32].

Nowadays, Apache Spark [33] has become very popular for implementing applications using an extension of the MapReduce model, instead of Hadoop. For this reason, several works have focused on extending Apache Spark to support GPUs: HeteroSpark [10], a heterogeneous CPU/GPU Spark platform for machine learning algorithms, IBMSparkGPU [9], with the same approach as the one used in [34]. Recently, Fukutomi et al. [35] added to Apache Yarn, a resource manager mainly used in MapReduce frameworks, support for GPUs, although targeting Java applications. Even without native support for GPUs in these frameworks, Boubela et al. [8] managed to combine Big Data approaches for the pre-processing of large-scale fMRI data using Apache Spark with separate GPU servers for accelerating specific steps of the processing pipeline. Cao et al. [7] compared the performance of a distributed medical image application in Spark with respect to a GPU-based implementation, showing the benefits of the use of GPUs but without integrating them in their system. At the end, our solution closes the gap between big data distributed solutions and the use of HPC architectures for medical imaging.

For both multi-core CPUs and distributed solutions, Python has become a widely extended language for image processing in all branches of science. An early work presented Matplotlib [36], a 2D graphics package and programming interfaces that includes a toolkit for medical image processing. More recently, scikit-image [37], an open source image processing library, has been also applied to MRI or CT reconstruction. Another example is Gadgetron [38], an open source framework that implements a flexible system for creating dynamical streaming configurable data processing pipelines. A complete toolkit for image reconstruction in tomography is TomoPy [39], which includes a large number of reconstruction and projection algorithms that can be invoked from Python.

Our approach, although similar to those presented in [7, 8], aims to be a more general solution for different iterative algorithms with support for legacy C/C++ code, new Python code, and CUDA kernels. With respect to the support of GPUs, this work does not require modifications in the base framework in contrast with [9, 10], making this approach accessible to a large variety of clusters or clouds environments.

6. Discussion and Conclusions

In this work, we present a new approach based on Python and Apache Spark for the implementation of the *projection* and *backprojection* components of an iterative reconstruction method for cone-beam geometry. Our solution enables two alternatives for different architectures: a GPU-based architecture, supporting NVidia GPUs, and a CPU-based architecture, relying on CPU-only acceleration and the compatibility with C/C++ native code. The main contributions of this work are the following. First, we present a novel approach for accelerated iterative reconstruction algorithms based on the offload of the most computationally expensive components. Second, we introduce a GPU-based architecture for the Apache Spark framework. Third, we carry out a study of the partitioning problem in the *projection* and *backprojection* component in terms of performance. Finally, the proposed architecture is evaluated in a heterogeneous CPU/GPU-based cluster, combining both HPC programming models and Big Data frameworks.

Although both *projection* and *backprojection* components are similar in terms of execution and complexity, their inclusion in a Big Data framework exposes different behaviour. As seen in Section 4, the partitioning of 3D volumes (which is the most memory consuming data structure) is a suitable approach for *backprojection*, in which partitioning also implies increasing the computational parallelism in the application. However, this principle does not hold for the case of the *projection* component, in which computational cost is based on the total number of projections and their size, and a partitioning of the volume can even increase the cost due to the increment of the FOV.

Out of the four different evaluated configurations, the best solution combines an increased number of partitions in the *backprojection* component (*configuration B*), aiming at maximizing external/coarse-grained parallelism (even increasing the number of executors per node), and a lower number of partitions in the case of *projection*, exploiting the computational resources through a shared-memory mechanism like OpenMP (as shown in Figure 15) (*configuration A*). This shared-memory mechanism enables the parallelism at a fine grained level inside the partition and not per partition as it happens when using the *configuration B*. Moreover, with this configuration, OpenMP internal parallelism can be favoured by a larger amount of work organized larger partitions, a factor that it is penalized in *configuration B*.

Commonly, in GPU-based architectures parallelism is provided by using the underlying GPU (internal/fine grained) and the usage of multiple GPUs devices by different executors (external/coarse grain). To avoid the limitation on the external parallelism offered by multiple concurrent executors running in the same node, the execution of a large amount of tasks is orchestrated by our proposed internal scheduler. This approach is specially beneficial in the case of the *backprojection* component.

We have identified that the Apache Spark framework presents certain performance limitations, especially noticeable in the case of the GPU-based architectures. The overhead imposed by Apache Spark can cause, not only a problem of performance, but also a problem of memory exhaustion. The layered software architecture of Apache Spark is translated to a higher memory usage to hold both RDDs and network serialization structures. This is a problem already mentioned in other works [40, 29, 41, 42]. To overcome these problems, we have carried out a meticulous configuration of the framework execution parameters. This tuning process, in some cases, is more cumbersome than implementing the application itself.

Our approach is based on Apache Spark, but it is applicable to other Big Data frameworks that support Python, enabling quickly updates to new requirements. Moreover, since the alternatives presented here are based on the combination of different components, they can be generalised to other types of architectures or acceleration devices. In the case of the heterogeneous approach, OpenCL could also be used for compatibility with other accelerators or even for a better exploitation of the CPU.

Acknowledgments

This work was supported by the NIH under Grant R01-HL-098686 and Grant U01 EB018753, the Spanish Ministerio de Economia y Competitividad (projects TEC2013-47270-R, RTC-2014-3028 and TIN2016-79637-P), the Spanish Ministerio de Educacion (grant FPU14/03875), the Spanish Ministerio de Ciencia, Innovacion y Universidades (Instituto de Salud Carlos III, project DTS17/00122; Agencia Estatal de Investigacion, project DPI2016-79075-R - AEI/FEDER, UE), co-funded by European Regional Development Fund (ERDF), A way of making Europe. The CNIC is supported by the Ministerio de Ciencia, Innovacion y Universidades and the Pro CNIC Foundation, and is a Severo Ochoa Center of Excellence (SEV-2015-0505).

References

[1] J. F. P. J. Abascal, M. Abella, A. Sisniega, J. J. Vaquero, M. Desco, Investigation of Different Sparsity Transforms for the PICCS Algorithm in Small-Animal Respiratory Gated CT, PLOS ONE 10 (4) (2015) 1–18. doi:10.1371/journal.pone.0120140.
URL `https://doi.org/10.1371/journal.pone.0120140`

[2] E. Serrano, T. Vander Aa, R. Wuyts, J. G. Blas, J. Carretero, M. Abella, Exploring a Distributed Iterative Reconstructor Based on Split Bregman Using PETSc, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2016, pp. 191–200.

[3] Palenstijn, W.J., Bedorf, J., Batenburg, K.J., King, M., Glick, S., Mueller, K., NWO, A distributed SIRT implementation for the ASTRA Toolbox, None, 2015.

[4] C.-T. Yang, W.-C. Shih, L.-T. Chen, C.-T. Kuo, F.-C. Jiang, F.-Y. Leu, Accessing medical image file with co-allocation HDFS in cloud, Future Generation Computer Systems 43 (2015) 61–73.

[5] E. Serrano, J. Garcia-Blas, J. Carretero, M. Abella, M. Desco, Medical Imaging Processing on a Big Data Platform Using Python: Experiences with Heterogeneous and Homogeneous Architectures, in: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 830–837. doi:10.1109/CCGRID.2017.56.

[6] E. Serrano, J. G. Blas, J. Carretero, M. Abella, Architecture for the Execution of Tasks in Apache Spark in Heterogeneous Environments, in: 4th International Workshop on Parallelism in Bioinformatics, 2016.

[7] L. Cao, P. Juan, Y. Zhang, Real-Time Deconvolution with GPU and Spark for Big Imaging Data Analysis, in: Algorithms and Architectures for Parallel Processing, Springer, 2015, pp. 240–250.

[8] R. N. Boubela, K. Kalcher, W. Huf, C. Našel, E. Moser, Big Data approaches for the analysis of large-scale fMRI data using Apache Spark and GPU processing: A demonstration on resting-state fMRI data from the Human Connectome Project, Frontiers in neuroscience 9.

[9] S. Moore, M. Kandasamy, J. Samuel, GPUEnabler (Aug. 2016).
URL `https://github.com/IBMSparkGPU/GPUEnabler`

[10] P. Li, Y. Luo, N. Zhang, Y. Cao, HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms, in: Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on, IEEE, 2015, pp. 347–348.

[11] RPyC - Transparent, Symmetric Distributed Computing - RPyC.
URL `https://rpyc.readthedocs.io/en/latest/index.htm`

[12] X. Wang, A. Sabne, S. Kisner, A. Raghunathan, C. Bouman, S. Midkiff, High Performance Model Based Image Reconstruction, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, ACM, New York, NY, USA, 2016, pp. 2:1–2:12. doi:10.1145/2851141.2851163.

[13] R. Sampson, M. McGaffin, T. Wenisch, J. Fessler, Investigating multi-threaded SIMD for helical CT reconstruction on a CPU, in: Proceedings of the 4th International Meeting on image formation in X-ray CT, 2016, pp. 275–278.

[14] A. Sabne, X. Wang, S. J. Kisner, C. A. Bouman, A. Raghunathan, S. P. Midkiff, Model-based Iterative CT Image Reconstruction on GPUs, in: Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '17, ACM, New York, NY, USA, 2017, pp. 207–220. doi:10.1145/3018743.3018765.

[15] T. Bai, H. Yan, X. Jia, S. Jiang, G. Wang, X. Mou, Z-Index Parameterization for Volumetric CT Image Reconstruction via 3-D Dictionary Learning, IEEE Transactions on Medical Imaging 36 (12) (2017) 2466–2478. doi:10.1109/TMI.2017.2759819.

[16] C. Jian-Lin, L. Lei, W. Lin-Yuan, C. Ai-Long, X. Xiao-Qi, Z. Han-Ming, L. Jian-Xin, Y. Bin, Fast parallel algorithm for three-dimensional distance-driven model in iterative computed tomography reconstruction, Chinese Physics B 24 (2) (2015) 028703.

[17] M. Schellmann, S. Gorlatch, D. Meilander, T. Kosters, K. Schafers, F. Wubbeling, M. Burger, Parallel medical image reconstruction: from graphics processing units (GPU) to Grids, The Journal of Supercomputing 57 (2) (2011) 151–160. doi:10.1007/s11227-010-0397-z.

[18] M. Abella, E. Serrano, J. Garcia-Blas, I. García, C. De Molina, J. Carretero, M. Desco, FUX-Sim: Implementation of a fast universal simulation/reconstruction framework for X-ray systems, PloS one 12 (7) (2017) e0180363.

[19] J. Garcia-Blas, M. Abella, F. Isaila, J. Carretero, M. Desco, Surfing the optimization space of a multiple-GPU parallel implementation of a X-ray tomography reconstruction algorithm, Journal of Systems and Software (0) (2014) –.

[20] Y. Lu, F. Ino, K. Hagihara, Cache-Aware GPU Optimization for Out-of-Core Cone Beam CT Reconstruction of High-Resolution Volumes, IEICE TRANSACTIONS on Information and Systems E99-D (12) (2016) 3060–3071.

[21] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, M. Ujaldon, Biomedical image analysis on a cooperative cluster of GPUs and multicores, in: ACM International Conference on Supercomputing 25th Anniversary Volume, ACM, 2014, pp. 413–423.

[22] C. A. Gulo, A. C. Sementille, J. M. R. Tavares, Techniques of medical image processing and analysis accelerated by high-performance computing: a systematic literature review, Journal of Real-Time Image Processing 1–18.

[23] B. Meng, G. Pratx, L. Xing, Ultrafast and scalable cone-beam CT reconstruction using MapReduce in a cloud computing environment, Medical physics 38 (12) (2011) 6603–6609.

[24] C. Sweeney, L. Liu, S. Arietta, J. Lawrence, HIPI: a Hadoop image processing interface for image-based mapreduce tasks, Chris. University of Virginia.

[25] Scaling Up Fast: Real-time Image Processing and Analytics using Spark (May 2014).

[26] X. Yang, T. Jejkal, H. Pasic, R. Stotzka, A. Streit, J. v. Wezel, T. d. S. Rolo, Data Intensive Computing of X-Ray Computed Tomography Reconstruction at the LSDF, in: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013, pp. 86–93. doi:10.1109/PDP.2013.21.

[27] J. M. Rosen, J. Wu, T. F. Wenisch, J. A. Fessler, Iterative helical CT reconstruction in the cloud for ten dollars in five minutes, in: Proc. Intl. Mtg. on Fully 3D Image Recon. in Rad. and Nuc. Med, 2013, pp. 241–4.

[28] T. Bicer, D. Gursoy, R. Kettimuthu, F. D. Carlo, G. Agrawal, I. T. Foster, Rapid Tomographic Image Reconstruction via Large-Scale Parallelization, in: Euro-Par 2015: Parallel Processing, Springer, Berlin, Heidelberg, 2015, pp. 289–302. doi:10.1007/978-3-662-48096-0˙23.

[29] L. Gu, H. Li, Memory or time: Performance evaluation for iterative operation on Hadoop and Spark, in: High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on, IEEE, 2013, pp. 721–727.

[30] A. Abbasi, F. Khunjush, R. Azimi, A preliminary study of incorporating GPUs in the Hadoop framework, in: 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS), 2012, IEEE, 2012, pp. 178–185.

[31] H. Zheng, J. Wu, Accelerate k-means algorithm by using GPU in the Hadoop framework, in: Web-Age Information Management, Springer, 2014, pp. 177–186.

[32] W. He, H. Cui, B. Lu, J. Zhao, S. Li, G. Ruan, J. Xue, X. Feng, W. Yang, Y. Yan, Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters, in: Proceedings of the 29th ACM on International Conference on Supercomputing, ACM, 2015, pp. 143–153.

[33] Apache Spark.
URL `http://spark.apache.org/docs/latest/index.html`

[34] S. Hong, W. Choi, W.-K. Jeong, GPU in-memory processing using Spark for iterative computation, in: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE Press, 2017, pp. 31–41.

[35] D. Fukutomi, Y. Iida, T. Azumi, S. Kato, N. Nishio, GPUhd: Augmenting YARN with GPU Resource Management, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, ACM, 2018, pp. 127–136.

[36] J. D. Hunter, Matplotlib: A 2D graphics environment, Computing In Science & Engineering 9 (3) (2007) 90–95.

[37] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, scikit-image: image processing in Python, PeerJ 2 (2014) e453.

[38] M. S. Hansen, T. S. Srensen, Gadgetron: An open source framework for medical image reconstruction, Magnetic Resonance in Medicine 69 (6) (2013) 1768–1776. doi:10.1002/mrm.24389.

[39] D. Gürsoy, F. De Carlo, X. Xiao, C. Jacobsen, Tomopy: a framework for the analysis of synchrotron tomographic data, Journal of synchrotron radiation 21 (5) (2014) 1188–1193.

[40] A. G. Shoro, T. R. Soomro, Big data analysis: Apache spark perspective, Global Journal of Computer Science and Technology 15 (1).

[41] S. Han, W. Choi, R. Muwafiq, Y. Nah, Impact of Memory Size on Big-data Processing based on Hadoop and Spark, in: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, ACM, 2017, pp. 275–280.

[42] S. Caino-Lores, A. Lapin, P. Kropf, J. Carretero, Lessons Learned from Applying Big Data Paradigms to a Large Scale Scientific Workflow, in: 11th Workshop on Workflows in Support of Large-Scale Science (WORKS 2016), 2016.