2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Medical Imaging Processing on a Big Data platform using Python: Experiences with Heterogeneous and Homogeneous Architectures

Estefania Serrano, Javier Garcia Blas, Jesus Carretero
Computer Architecture and Technology Group
University Carlos III of Madrid
Leganes, Spain
{*esserran, fjblas, jcarrete*}*@inf.uc3m.es*

Monica Abella, Manuel Desco
Instituto de Investigacin Sanitaria Gregorio Marañon (IiSGM)
Madrid, Spain
{*mabella, desco*}*@hggm.es*

***Abstract*—The apparition of new paradigms, programming models, and languages that offer better programmability and better performance turns the implementation of current scientific applications into a less time-consuming task than years ago. One significant example of this trend is the MapReduce programming model and its implementation using Apache Spark. Nowadays, this programming model is mainly used for data analysis and machine learning applications, although it has been expanded to its usage in the HPC community. On the side of programming languages, Python has positioned itself as an alternative to other scientific programming languages, such as Matlab or Julia. In this work we explore the capabilities of Python and Apache Spark as partners in the implementation of the backprojection operator of a CT reconstruction application. We present two interesting approaches with two different types of architectures: a heterogeneous architecture including NVidia GPUs and a full performance CPU mode with the compatibility with C/C++ native source code. We experimentally demonstrate that current CPU-based implementations scale with the number of computational units.**

***Keywords*-CUDA; Big Data; Apache Spark; CT; Backprojection; Python**

I. Introduction

The implementation and design of scientific applications have been for years in the hands of specialists. Performance needs, difficulties for optimization, and the complexity of the HPC (High Performance Computing) architectures in which the applications run created a division between domain experts and "computer" experts. However, in recent years new programming models have appeared and they have attracted experts from both worlds, mainly due to their possibilities of obtaining high performance, programmability, and maintainability for the applications.

One of this programming models is MapReduce, employed in many scientific applications in the biomedical field in which data analysis is fundamental, including but not limited to DNA analysis, protein discovery or even image processing. The simplification of the programming functions and the abstraction of the management of the tasks and data make it suitable to a wider range of developers, not necessarily skilled in low level programming.

Furthermore, the popularization of programming languages such as Python and Matlab, widely used by domain experts, have raised the attention of HPC experts because of the possibility of their execution and deployment over HPC based environments [1], [2]. Examples are the inclusion in Python of optimization techniques previously limited to traditional languages like C/C++ with highly specialized mathematical libraries or the appearance of many profiling tools [3]. This kind of programming languages has also been widely used in Computed Tomography (CT) algorithms [4], [5], [6], facilitating the implementation and testing of new reconstruction and simulation algorithms.

The main contribution of this paper is the presentation of a comparative evaluation of two MapReduce-based implementations of a *backprojection* algorithm over two different hardware architectures. This paper aims to discuss about the feasibility of executing compute-intensive image processing algorithms using affordable programming languages for the scientific community, without performance degradation. This work focuses on Python, a widely utilized language in many MapReduce frameworks. This language exploits many features for optimizing and reusing previous C/C++ codes and CT *backprojection* algorithms. In a previous work [7], we presented a framework for heterogeneous architectures based on Apache Spark and PyCUDA. In this work, we will compare this approach with a hybrid Python/C CPU-only design, affordable for many types of architectures (removing the need of a GPU) and for optimizing the performance of the subjacent algorithms. We will evaluate side by side both approaches to provide a first insight into heterogeneous vs homogeneous Big Data platforms and its programmability.

The paper is organized as follows. In Section II, we describe the CT algorithms employed and the advantages of Python and Big Data programming models. Section III presents the design of our solution for homogeneous and heterogeneous platforms. The evaluation results of both approaches are discussed in Section IV. Finally, we discuss the obtained results and propose future works related to the presented solution.

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.56

II. Background

A. CT Reconstruction Algorithms

The objective of CT reconstruction algorithms is to obtain tomographic 3D images, also referred here as volume (Figure 1 (left)), from radiographies taken from different angles of the patient (Figure 1 (right)).

One of the latest trends in CT is the reduction of the radiation dose that is applied to the patient during the acquisition of the images (aka projections). The reduction of the dose can be obtained in different ways, although it normally implies decreasing the power utilized by the scanner or the acquisition of less images. Both methods significantly reduce the quality of the final reconstructed 3D image. This reduction is translated to a more difficult diagnosis of the patients. Thus, iterative reconstruction algorithms are needed to enhance the quality of the resulting image. Medical applications cope with computationally expensive algorithms that are employed for obtaining better quality images with a reduced radiation dose.

The basis of an iterative algorithm is the repetition of the *backprojection* and *projection* operators over images that are enhanced on every iteration until the desired image quality is achieved. These operators represent the most computational expensive parts of the algorithm and are normally optimized by using parallel and heterogeneous architectures.

The *backprojection* algorithm transforms the projections into a 3D volume, integrating all the intersecting rays over each of the voxels (discrete divisions of the volume). The mathematical formulation of this operator can be seen in Equation 1, where $f(u, v, z)$ is the pixel value in the reconstructed image at coordinates, (u, v, z), $p_\theta(s, z)$ the projection data for angle θ and position (s, z) in the detector, and W_1 and W_2 are the weighting factors introduced to compensate for the different ray lengths. SO is the distance from the source to the detector, z is the axial coordinate, common for both detector and reconstructed volume reference frames,s is the radial coordinate in the detector, and u, v are the Cartesian coordinates in the reconstructed volume. The *projection* algorithm is the inverse of *backprojection*, obtaining the value of each of the pixels of each projection by simulating the rays that traverse the volume.

A basic schema of the geometry for *backprojection* and *projection* algorithms is shown in Figure 2. The computation of the trajectories of each ray is completely independent in both operators, making them suitable for implementation in parallel architectures like GPUs.

Apart from these operators, the iterative CT reconstructor contains other operations for the constant refinement of the images. However, these operators do not represent a significant amount of time in the overall execution time. For this reason, we focused on these main operators and in the scope of this work, we focus on the study the performance of the *backprojection* algorithm.

B. Big Data Programming Models

Big Data programming models have become very popular along the last years. Apache Hadoop MapReduce has been highly successful in providing a relatively easy and structured approach to process large-scale, data-intensive applications on commodity clusters [8]. However, for applications needing high-performance iterative computations, Apache Spark [9], an in-memory processing framework, has shown to be more effective for fast data analysis and iterative algorithms and workflows [10]. The main features of Spark are the in-memory computation and the exploitation of data locality. The MapReduce programming model on Spark also eases programmability in medical image reconstruction, as shown in [11]. The authors show the implementation and evaluation of four reconstruction algorithms over two large datasets on an IBM BG/Q supercomputer. The feasibility of running science applications in the Spark framework at large-scale has been also studied in [12].

However, there are still many open challenges, such as difficulties in data capture, data storage, data processing, and data visualization [13]. Some recent studies have shown that current generation Big Data frameworks (i.e, Hadoop) cannot efficiently leverage advanced features on modern clusters with high-performance networks [14].

C. Python

Python has shown to be a very efficient programming language for many embarrassingly parallel applications, a model that is present in many scientific computing applications [15], [16]. Python brings mainly three advantages to the development of scientific applications: multi-platform applications; easier programmability, as it is close to mathematical programming languages and provides powerful mathematical libraries like Numpy or Scipy; and backwards compatibility, as most of the libraries written in Python have been initially programmed in C/C++.

However, Python has some disadvantages, as it is usually slower than native languages (like C/C++) and it is difficult to provide native parallelism to the applications. On the one hand, these disadvantages can be tackled with the many packages that exist such as Numpy or Scipy, which provide better performance to mathematical operations. On the other hand, PyMPI and PyMP facilitate the addition of parallelism to Python apps.

III. Homogeneous and Heterogeneous Design

The design of our application is based in the correspondence between the main phases of a MapReduce program and the *backprojection* algorithm. Having explained the *backprojection* operator in Subsection II-A, we can set its equivalence with a Map phase. In a first naive approach, we could map each of the input pixels of the projections to the correspondent output voxels. However, as we know, this relation is not constant: one pixel can affect several voxels

$$f(u,v,z) = \frac{1}{2}\int_{\theta=0}^{2\pi} W_2(v)\left[\int_{w=-\infty}^{\infty}\left(\int_{s=-\infty}^{\infty}[p_\theta(s,z)\cdot W_1(z,u)]\cdot e^{-j2\pi sw}ds\right)\cdot|w|\cdot e^{j2\pi ws}\cdot dw\right]\cdot d\theta \quad (1)$$

$$W_1 = \frac{SO}{\sqrt{SO^2+z^2+u^2}} \quad (2)$$

$$W_2 = \frac{SO}{(SO-v)^2} \quad (3)$$

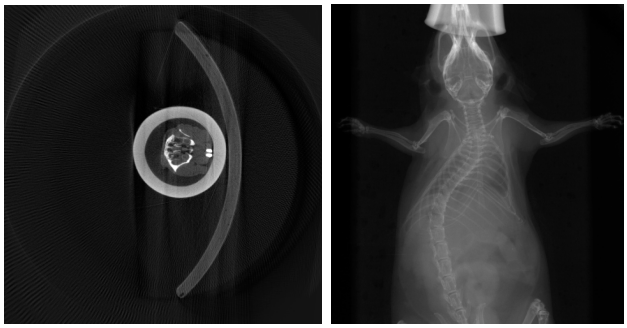

Figure 1. Reconstructed image of a rat (left). Input radiography for the reconstruction algorithm (right).

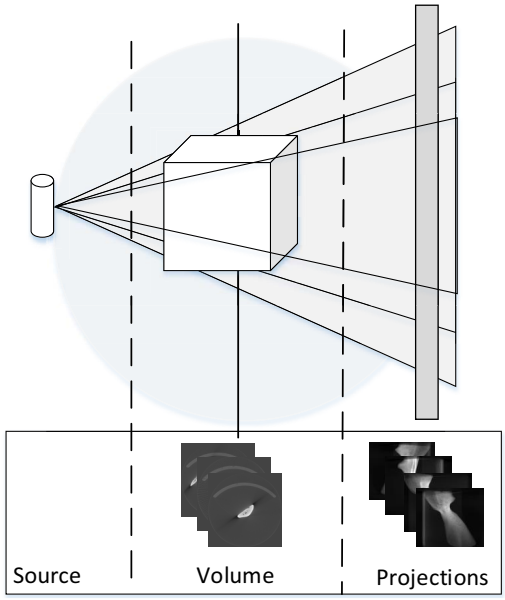

Figure 2. Geometric schema for both *backprojection* and *projection* algorithms. Lines from source to detector represent the X-rays intersecting the volume.

or none of them, causing irregularities that would impede a good partitioning of the problem.

Figure 3 shows the final partitioning model chosen for our design. Each of the mappers is in charge of generating one part of the volume. Meanwhile, the input data must be located in all mappers to obtain a correct result. Finally, the reduce phase consists of grouping/merging the different parts of the volume in the complete final result.

The mapping phase is the most interesting and complex phase and is where the main differences between both approaches reside. The result of each mapper is a key-value structure that identifies the number of slice (from 0 to the dimension in z) that has been generated and its correspondent values.

A. Heterogoneous Design

In the case of using a heterogeneous architecture, we decided to implement an additional intra-scheduler that is in charge of deciding in what GPU the mapping task will be executed. This decision is made depending on different policies and on the status of the selected accelerators. This intra-scheduler collects information of the load of each of the available GPUs and if no accelerator is available (due to the limited memory) it stalls the execution of the task until there is space available. The execution of the mapping in the heterogeneous architecture consists of five phases:

1) Communication with the intra-scheduler: the executor communicates with the scheduler to obtain the device in which the function should be executed. Based on the device obtained, the context for that specific accelerator is created.
2) Copy of the initial projection onto the device memory: using the PyCUDA API it is possible to transfer the arrays containing the initial projections to the device.
3) Execution of the CUDA kernel: the kernel is loaded and compiled and executed with the geometric configuration of the projections (number of angles, image resolution, distance between source and detector...etc).
4) Copy of the output volume to the host memory: to be able to obtain the final volume, each mapper should return its output stored in host memory. The volume generated in the device is transferred to the main memory before finishing.
5) Communication with the intra-scheduler: the executor communicates with the scheduler to announce the liberation of the device. The context for that card is destroyed.

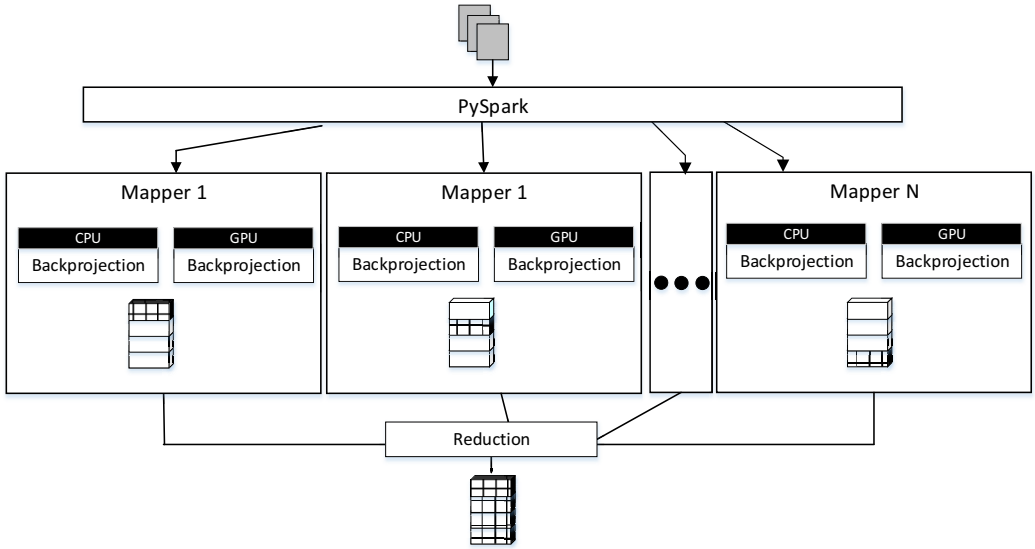

Figure 3. Execution and partitioning of the Backprojection in our proposed schema.

The communication with the intra-scheduler is done with the RPyC package [17].

B. Homogeneous Design

Regarding CPU-only architectures, there are two options: to execute the *backprojection* operator taking the whole computational power of the node through the OpenMP pragma directives, or to execute only in the core the assigned mapping task. The first approach can be useful and more efficient when distributing over a high number of nodes, obtaining a design with two level parallelism managed by two different programming models. The second approach allows a better scheduling of the tasks by the framework, as well as no further memory consumption, since the parallelism is already given by the framework. The execution of the *backprojection* operator following the homogeneous design implies the execution with Python of the preparation of the parameters and the data to be used in the operator. Nevertheless, no communication with the intra-scheduler is needed (Spark/YARN has already scheduled the task taking into account CPU and memory resources) and no memory transfer to any device is made inside the task. The *backprojection* algorithm is called as a normal Python function from the C module previously compiled. Inside the C module the Python parameters are transformed to C variables and the algorithm is executed with or without OpenMP depending on what is chosen. Finally, after the algorithm is executed the volume is returned to Python with the correct format.

IV. Experimental Results

In this section, we show the experiments carried out to evaluate the implemented solution using Python and Spark in both heterogeneous and homogeneous architectures.

A. Experimental Setup

We evaluated our solution using two environment setups:

- A single node with two Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz processors, 356 GB DDR4 of RAM, and three NVidia GPUs (one NVidia Tesla K40c and two NVidia GTX Titan Black), for evaluating the heterogeneous and homogeneous designs in one node.
- From 3 to 5 compute nodes of similar characteristics for evaluating a distributed environment.

The system is supervised through Cloudera 5.7 and runs with Ubuntu 14.04. The version of Apache Spark employed is 1.6 in *stand-alone* mode. For the distributed evaluation, YARN 2.6, with the *FairScheduler*, was used as resource manager. The input files are stored in a local SSD and the result files are saved into a HDFS directory, also in SSDs and with a 10 Gbps Ethernet network. The Python version was 2.7, complemented with PyCUDA 1.3 for the heterogeneous architecture and a C/C++ module compiled with GCC 4.8.5 and OpenMP for the homogeneous architecture. We executed each configuration three times and obtained the average of those executions.

The input data for the experiments in one node consisted of 360 projections with 1024x1024 pixels. The size of the output data was 1024x1024x1024 voxels. In the case of the distributed evaluation we employed 360 projections with 2048x2048 pixels. The size of this output data was 2048x2048x2048 voxels.

We selected the *backprojection* algorithm for this preliminary evaluation, which should scale better due to the partitioning scheme employed. The presented results correspond with the average of five consecutive executions.

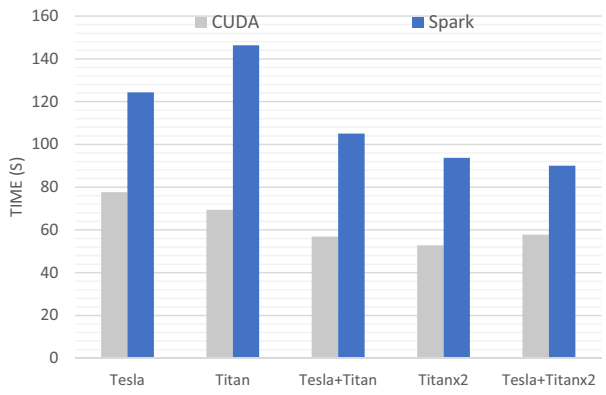

Figure 4. Execution of the starting implementation vs heterogeneous Spark with different configurations of GPUs.Each GPU is controller by one thread or executor (in the case of Apache Spark).

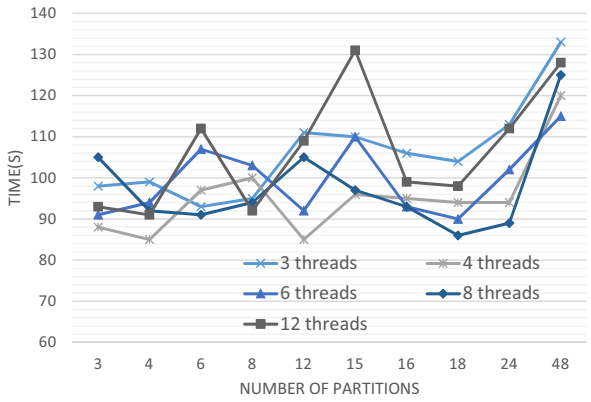

Figure 5. Execution of heterogeneous Spark with a different number of threads and partitions.

B. Heterogeneous Architecture Evaluation

The Spark results for different number of GPUs, presented in Figures 4 and 5, are not as satisfactory as expected. For the same hardware type (different model cards from NVidia) our heterogeneous distributed architecture works almost two times slower than the starting implementation of the algorithm, implemented on the original system using C and CUDA. This occurs despite of the fact that it obtains a better occupancy of the GPU and a better exploitation of the resources thanks to the intra-scheduler.

However, when analyzing the results with the bigger number of GPUs (both Titan and Tesla cards executing at the same time) the difference is reduced to only 50% slower executions with Spark. This means that under the overhead of the overall framework and a reduced number of accelerator devices the original native implementation is executed with full performance meanwhile the heterogeneous Spark version needs to handle the communication between the executors (even when they are in the same node) and additional tasks such as GC (Garbage Collection) or initialization of the framework and network interfaces.

But, when the number of devices increases the intra-scheduler provides a better planning of each of the tasks as well as reducing the overall overhead of the execution due to the automatic partitioning provided by the framework. In this case the simulator must divide and manage the devices manually thus reducing the advantages of this approach vs heterogeneous Spark. Additionally, there exist a difference in performance between plain C/CUDA code and Python/CUDA code due to the characteristics of each of the languages.

As shown in Figure 5, these results do not change when changing the number of partitions or threads, leading to a non scalability problem. In fact, we can observe that a higher number of threads does not lead to a significant improvement, even worsening the execution time in some cases.

In Table I, we show the average execution times for different distributed configurations, being the best time (456 seconds) similar to the one obtained by the original CUDA code in one node for that size. In the distributed configuration we have a similar problem of scalability to the one mentioned in the one node configuration. Even if we obtain lower execution times with a higher number of nodes, the reduction does not correspond to the increase in the computational power.

C. Homogeneous Architecture Evaluation

As a preliminary evaluation we executed several configurations of the *backprojection* code to obtain a baseline time from which we can compare to. The results of this preliminary evaluation are shown in Table II. The Spark version was executed by only one thread with OpenMP on the C/C++ kernel. The difference between the Python version and the PySpark version can be considered to be the overhead introduced by Spark. This is the lowest overhead introduced by the framework, since the scheduling is minimum (only one executor is present) and data transfers do not exist (only one node is used).

Then we executed the PySpark version with a different number of threads in one compute node. The average execution time for each number of threads, as well as the ideal time for that execution, are shown in Figure 6. The ideal time was computed taking into account ideal parallelism, for a number of threads i : $Idealtime_i = \frac{sequentialtime}{i}$.

The time obtained is near ideal although as the number of threads increase the difference with the ideal time is higher probably due to the increment in the overhead of the execution of the framework which has to schedule and manage a higher number of executors.

Distributed environments also provide good scalability, as may be seen in Table III that presents the average execution times for different configurations. In this case, the number

Table I
EXECUTION OF THE HETEROGENEOUS SPARK APPLICATION OVER DIFFERENT NODES FOR A 2048X2048X2048 VOLUME OF RESULT.

Nodes	Executors	Threads per executor	Partitions	Time (sec)
5	5	2	120	**456.00**
3	3	4	60	462.37
3	3	5	40	458.09
3	3	2	60	591.76

of executors is fixed to be identical to the number of nodes, since each executor will be able to launch the rest of the threads with the C/C++ with OpenMP *backprojection*. This time the results are not as near to the ideal computed times, but the speedups obtained (shown in Figure 7) are in an acceptable range taking into account that there exist an extra overhead due to the communication and the scheduling of the framework.

D. Discussion

The heterogeneous and homogeneous approaches tested have yielded very different results. In the case of a heterogeneous architecture, we have seen that the scalability of the solution is highly compromised by two factors: the overhead imposed by the framework and the number of devices in the computer.

The second factor seems obvious since a greater number of GPUs increase potentially the computational performance of the computer, although as we have seen from the study of performance with different GPUs, the model of the accelerator largely affects the final execution time. However, regarding the first factor, we do not observe that type of behaviour in the homogeneous architecture. The reason behind this behaviour is the relation between the computational power and the memory capacity. Regarding the heterogeneous architecture, the ratio of time spent in the computation of the tasks and the overhead imposed by the framework is higher, since the execution of the *backprojection* operator in the accelerator is fast. Nonetheless, in the CPU this ratio decreases due to the time spent in the *backprojection* phase.

Moreover, the main problem found, specially in the case of the heterogeneous implementation, was the overhead imposed by the Apache Spark framework, which causes not only a problem of performance, but also a problem of memory exhaustion. The execution of the several layers of software which conform Spark produces the usage of additional memory to hold the RDDs (Resilient Distributed Datasets), Hadoop containers, the buffers for networking, serialization and the GC memory. This is a problem already noticed in some other works, especially when compared with plain Hadoop [18], [19]. To overcome these problems, it is necessary to perform a meticulous configuration of the framework and the execution parameters. This process can take, in some cases, more time than the time needed to implement the application itself.

Table II
AVERAGE EXECUTION TIME IN SECONDS FOR DIFFERENT VERSIONS OF THE *backprojection* CODE.

Environment	Time (seconds)
C/C++ (OpenMP)	260.79
Python (C/C++)	6,124.47
Python (C/C++ OpenMP)	439.48
PySpark (C/C++ OpenMP)	479.35

Table III
EXECUTION OF THE HOMOGENEOUS SPARK APPLICATION OVER DIFFERENT NODES FOR A 2048X2048X2048 VOLUME.

Nodes	Executors	Partitions	Time (seconds)
5	5	5	6,840
4	4	4	8,640
3	3	4	11,880
2	2	4	22,680

V. RELATED WORK

There exist multiple implementations for reconstruction algorithms and hardware platforms. However, most of them target HPC environments that allow the optimization of the main computational operators, (*projector* and *backprojection*). Examples of this approach are [20], [21], and [22]. Some of them use only CPU resources, meanwhile the last one is also optimized for GPU architectures. Optimized Distributed implementations of reconstruction algorithms can be found employing either the MapReduce programming model (as we presented in this work) [23] or MPI [24], [25].

Python has become increasingly popular for image processing in all branches of science. An early work was presented in Matplotlib [26], a 2D graphics package used for image generation and user interfaces, including toolset for medical images. More recently, scikit-image [27], an open source image processing library, has been also applied to 3-D magnetic resonance or computed tomography imaging. Another example is Gadgetron [4], an open source framework for medical image reconstruction that implements a flexible system for creating streaming dynamically configurable data

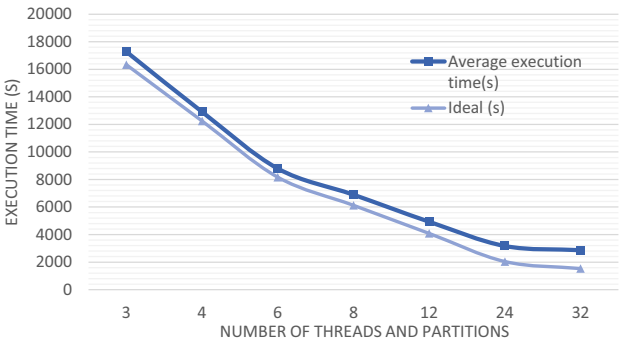

Figure 6. Execution of homogeneous Spark with a different number of threads. The number of partitions is identical to the number of threads.

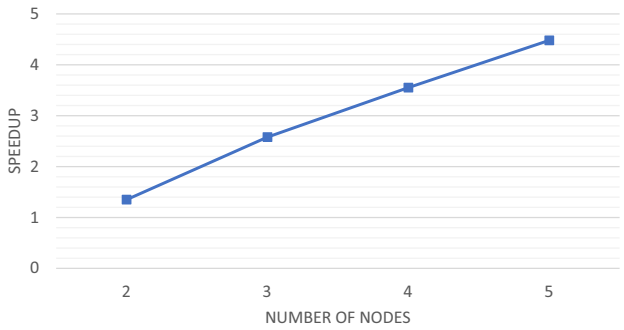

Figure 7. Speedup of the distributed execution of the homogeneous architecture for a different number of nodes.

processing pipelines from raw data to reconstructed images. Modules are typically implemented in C/C++, but new modules can be implemented in Python scripting language for rapid prototyping.

VI. Conclusion

In this work we have explored the capabilities of Python and Apache Spark as partners in the implementation of the *backprojection* operator of a CT reconstruction application. We show two interesting possibilities with two different types of architectures: heterogeneous architecture, including NVidia GPUs, and homogeneous architectures, with full performance CPU-only mode with the compatibility with C/C++ native code.

Although the performance does not reach a linear speed up, our approach is a good alternative for porting faster previous HPC applications already programmed in C/C++ or even with CUDA or OpenCL programming models, since Python is already compatible with this type of implementations. This work is an approach based on Apache Spark, but it is applicable to other Big Data frameworks that support Python, making it possible to update quickly to new needs of other applications.

Moreover, since the approaches presented here are based on the union of different components, they can be generalized to other type of architectures or devices. In the case of the heterogeneous approach, OpenCL could also be used for compatibility with other accelerators or even a better exploitation of the CPU. Regarding the solution for a homogeneous architecture, it can also be applied to the Intel Xeon Phi, with a recompilation of the module, which should provide much higher performance than conventional CPUs.

Acknowledgment

This work has been partially supported through grants TIN2016-79637-P "Towards unification of HPC and Big Data Paradigms" from the Spanish Ministry of Economy and Competitiveness, FPU14/03875 from the Spanish Ministry of Education, and by NECRA RTC-2014-3028-1 project. We also want to thank NVidia for providing the device Tesla K40 which with we have been able to perform the experiments.

References

[1] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and 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.

[2] V. H. Naik and C. S. Kusur, "Analysis of performance enhancement on graphic processor based heterogeneous architecture: A cuda and matlab experiment," in *Parallel Computing Technologies (PARCOMPTECH), 2015 National Conference on*. IEEE, 2015, pp. 1–5.

[3] A. Supalov, A. Semin, C. Dahnken, and M. Klemm, *Optimizing HPC Applications with Intel Cluster Tools*. Apress, 2014.

[4] M. S. Hansen and T. S. Srensen, "Gadgetron: An open source framework for medical image reconstruction," *Magnetic Resonance in Medicine*, vol. 69, no. 6, pp. 1768–1776, 2013. [Online]. Available: http://dx.doi.org/10.1002/mrm.24389

[5] W. Birkfellner, *Applied medical image processing: a basic course*. CRC Press, 2015.

[6] J. L. Semmlow and B. Griffel, *Biosignal and medical image processing*. CRC press, 2014.

[7] E. Serrano, J. G. Blas, J. Carretero, and M. Abella, "Architecture for the execution of tasks in apache spark in heterogeneous environments," in *4th International Workshop on Parallelism in Bioinformatics*, 2016.

[8] S. Okur, C. Radoi, and Y. Lin, "Hadoop+ aparapi: Making heterogeneous mapreduce programming easier."

[9] "Apache Spark." [Online]. Available: http://spark.apache.org/docs/latest/index.html

[10] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter, "Scientific computing meets big data technology: An astronomy use case," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 918–927.

[11] T. Bicer, D. Gursoy, R. Kettimuthu, F. De Carlo, G. Agrawal, and I. T. Foster, "Rapid tomographic image reconstruction via large-scale parallelization," in *European Conference on Parallel Processing*. Springer, 2015, pp. 289–302.

[12] J. G. Shanahan and L. Dai, "Large scale distributed data science using apache spark," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 2323–2324.

[13] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[14] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, "A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures," in *2014 IEEE International Congress on Big Data*, Jun. 2014, pp. 645–652.

[15] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[16] F. Perez, B. E. Granger, and J. D. Hunter, "Python: an ecosystem for scientific computing," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 13–21, 2011.

[17] "RPyC - Transparent, Symmetric Distributed Computing RPyC." [Online]. Available: https://rpyc.readthedocs.io/en/latest/index.htm

[18] A. G. Shoro and T. R. Soomro, "Big data analysis: Apache spark perspective," *Global Journal of Computer Science and Technology*, vol. 15, no. 1, 2015.

[19] L. Gu and 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.

[20] S. Rit, M. van Herk, and J.-J. Sonke, "Fast distance-driven projection and truncation management for iterative cone-beam ct reconstruction."

[21] C. Jian-Lin, L. Lei, W. Lin-Yuan, C. Ai-Long, X. Xiao-Qi, Z. Han-Ming, L. Jian-Xin, and Y. Bin, "Fast parallel algorithm for three-dimensional distance-driven model in iterative computed tomography reconstruction," *Chinese Physics B*, vol. 24, no. 2, p. 028703, 2015.

[22] V.-G. Nguyen, J. Jeong, and S.-J. Lee, "Gpu-accelerated iterative 3d ct reconstruction using exact ray-tracing method for both projection and backprojection," in *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2013 IEEE*. IEEE, 2013, pp. 1–4.

[23] B. Meng, G. Pratx, and L. Xing, "Ultrafast and scalable cone-beam ct reconstruction using mapreduce in a cloud computing environment," *Medical physics*, vol. 38, no. 12, pp. 6603–6609, 2011.

[24] Palenstijn, W.J., Bdorf, J., Batenburg, K.J., King, M., Glick, S., Mueller, K., and NWO, "A distributed SIRT implementation for the ASTRA Toolbox." None, Jun. 2015.

[25] J. M. Rosen, J. Wu, T. F. Wenisch, and 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.

[26] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[27] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in python," *PeerJ*, vol. 2, p. e453, 2014.