Spark Scalability Analysis in a Scientific Workflow

Renan Souza[1, 2], Vítor Silva[1], Pedro Miranda[1], Alexandre A. B. Lima[1], Patrick Valduriez[3], Marta Mattoso[1]

[1]COPPE – Federal University of Rio de Janeiro
[2]IBM Research – Brazil
[3]Inria and LIRMM, Montpellier
{renanfs,silva,pmiranda,assis,marta}@cos.ufrj.br,
patrick.valduriez@inria.fr

Abstract. *Spark is being successfully used for big data parallel processing in many business domains (social media, finance, retail). Spark's scalability, usability, and large user community have motivated developers from scientific domains (bioinformatics, oil and gas, astronomy) to try it. However, scientific applications' profile, e.g., black-box programs and intense file writes, differs from traditional business workflows, which may affect its scalability. We present a scalability analysis of Spark in a real case-study in Oil and Gas domain. We explore workloads on a 936-cores HPC cluster processing 330 GB of scientific data. We show that it scales very well when running long-lasting scientific tasks, but its performance is lower for short-duration tasks.*

1. Introduction: Spark and Scientific Workflows

Spark is one of the most popular data parallel processing systems for business big data workflows. One common characteristic in such workflows is that they require large computing clusters to process vast amounts of data. This is also true for many workflows in scientific domains, such as bioinformatics, oil and gas, and astronomy [Atkinson et al. 2017]. Besides its scalability in business workflows [Armbrust et al. 2015, Shi et al. 2015], Spark has many other interesting features. For example, a dataflow-oriented execution model, an API using well-known programming languages (*e.g.*, Python, Scala, Java) that facilitates expressing linked data transformations (*e.g.*, map, reduce, filter, join) of typed data elements, an efficient fault-tolerance support, and a large user community. Such features have motivated scientific workflow users to migrate from traditional parallel scientific workflow management systems to Spark [Gittens et al. 2016, Oliveira et al. 2015, Zhang et al. 2017].

However, despite the shared characteristic of big data processing, scientific and business workflows have many differences. Scientific workflows are commonly designed as multiple linked Many-Task Computing (MTC) applications [Raicu et al. 2008]. That is, tasks in a scientific application consume input data, perform scientific computations, and produce output data that feeds a subsequent task of a different application, thus forming a dataflow between linked tasks. Each task may perform computations with varying complexity levels that last from milliseconds to a few minutes each. In addition, each task normally consumes and produces small files (tens of MB). Very often, scientific workflows expect that those files are stored at runtime on an HPC shared file system (*e.g.*, GPFS, Lustre) for data communication between tasks. In total, an entire workflow execution may take days of continuous run and process terabytes of data. In contrast, common big data workflows read large input files in their beginning, process them, and write aggregated output data on disk to be analyzed, queried, and used for plotting charts and interactive dashboards. Spark is highly benefited in this case, as it can load from disk only once, process the data in its Resilient Distributed Datasets in the cluster memory, and write results to disk in the end. Besides, differently than typical scientific workflows, business workflows are good fit for shared-nothing clusters, often seen in Spark deployments, as disk I/O is reduced.

In addition, traditional scientific workflow systems deal with scientific applications as black-boxes. That is, tasks are aware of input and output data, but the internal computation is opaque. When developing a scientific workflow, users do not want to rewrite very complex (already tested, optimized, and stable) scientific programs using a different programming language or, oftentimes, they only have the executable binaries and not the source code. Spark API allows developers to completely write their code using native data transformations. It enables Spark's engine to be aware of the internals of each task and perform runtime optimizations for the benefit of in-memory data processing and data locality. This is not achievable when running typical scientific workflows developed as chains of black-boxes. Thus, these scientific workflows' characteristics may harm Spark's performance. Although several works have analyzed Spark's performance in business workflows, very few have investigated Spark in scientific workflows [Gittens et al. 2016, Oliveira et al. 2015, Zhang et al. 2017]. The ones that do either adapt their code to enable Spark's runtime optimizations, or limit those multiple disk I/O commonly seen in scientific applications, or use a shared-nothing cluster, rather than a traditional HPC shared file system. Therefore, we are not aware of works that have investigated Spark's scalability in scientific workflows by exploring multiple typical scientific workloads using the traditional profile of scientific applications, such as parameter sweep workflows, one of the most representative class of workflow applications [F. da Silva et al. 2017].

In this work, we use a real case study in Oil and Gas domain to build a scientific workflow in Spark and understand its scalability. We do not modify the original scientific application codes, but we model its parameter sweep as a dataflow using Spark. We use native Spark operators for external calls to the black-box applications, which write multiple files on a shared disk during execution. Source code of the dataflow implementation is on [GitHub]. We deploy Spark on a large HPC cluster, with 936 cores, where all nodes share a fast file system. We perform scalability analysis by varying multiple workloads, typical in scientific workflows, to understand the system performance. We show that Spark scales very well for workflows with long-lasting tasks, but its performance is lower for short-duration tasks. In the remainder of this paper, we explain the scientific workflow utilized in Section 2, the scalability analysis in Section 3, and the conclusion in Section 4.

2. Risers Fatigue Analysis Workflow in Spark

We model and execute a synthetic implementation of Riser Fatigue Analysis (RFA) scientific workflow based on a real case study in Oil and Gas domain. Specific details about this workflow can be found in [Souza et al. 2016]. Figure 1 shows RFA composed of 7 linked programs with a dataflow in between. The programs generate intermediary raw data files. Each of those files range from 8 to 14 MB, and there may be thousands of them in a real execution, generating terabytes of data.

To implement RFA in Spark, instead of re-writing those complex programs using its APIs, we follow the black-box approach to run external programs in a Spark dataflow. We use the natively available *Process* library in Spark to run external process executions. By doing so, it only manages the dataflow between each program call, while the actual task computation and file writes to the shared file system are outsourced in the scientific applications. Figure 2 shows a higher-level implementation of RFA in Spark, where we omit intermediary data manipulation for the dataflow, which is done inside Spark. The actual implementation is publicly available on [GitHub].

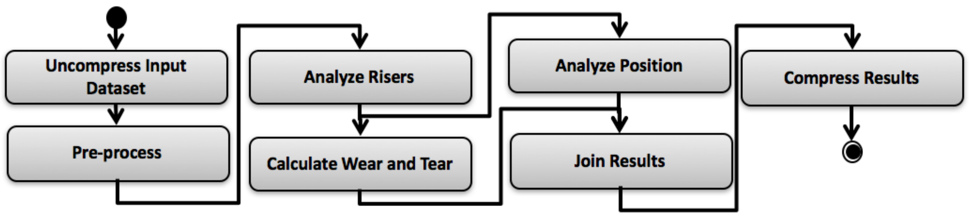

Figure 1. Risers Fatigue Analysis workflow.

```
val inputFiles = sc.textFile("/shared-disk/input-directory")
val uncompress = inputFiles.flatMap( x => Process("./Uncompress", x))
val preProcessing = uncompress.map(x =>Process("./Preprocess", x))
val analyzeRisers = preProcessing.map(x => Process ("./AnalyzeRisers", x))
val calcWearTear = analyzeRisers.filter(x => Process("./CalcWearTear", x))
val analyzePos = analyzeRisers.filter(x => Process("./AnalyzePos", x))
val jresults = calcWearTear.join(analyzePos)
val compressRes = jresults.reduceByKey((x,y) => Process("./Compress", x);  x)
```

Figure 2. Risers Fatigue Analysis workflow implemented in Spark.

3. Spark Scalability Analysis

We perform scalability analysis of Spark running RFA workflow. Scalability refers to the ability of a system to accommodate to a growth of the computing resources (*e.g.*, physical nodes) or the workload [Özsu and Valduriez 2011]. With respect to workload variation, in MTC workflows, there are at least two factors that compose a workload: number of tasks and task duration [Raicu et al. 2008]. Except for the last experiment, which has a real workload with varying task durations, we generate multiple synthetic workloads to represent different typical scenarios in scientific workflows to perform the scalability analysis. The task durations of the synthetic workloads follow a normal distribution with standard deviation 20% of the value of the task duration on average.

For software, we use native Spark version 1.6.1 with default settings, deploying on a standalone cluster using the cluster's shared file system. For hardware, we use a cluster with 39 nodes, with 24 cores each (936 cores in total), in Grid5000 (http://www.grid5000.fr). Each node has two AMD Opteron 1.7 GHz 12-core processors, 48GB RAM, connected via Gigabit Ethernet, and access a 10TB shared network storage. We repeat the experiments until the standard deviation of workflow execution times is less than 1, and the results are the average within the 1% margin.

Experiment 1: scalability varying workload and resources. We measure the execution time on 10 workers (240 cores) running about 6k tasks lasting 1 minute on average each. Then, we double the number of workers and tasks (480 cores - running 12k tasks). After, we execute using near four times the number of workers and tasks (936 cores - running 23.4k tasks). These numbers are representative for real RFA workloads [Souza et al. 2016] and other typical scientific workflows. Ideally, when the workload increases proportionally to the computing resources added to the cluster, the execution time is expected to remain constant. As Figure 3-I shows, Spark is scalable for running a typical execution of RFA, as its curve remains close to a horizontal line. When the scale increases, the system scalability starts to decrease. However, running 936 tasks concurrently, Spark runs only 17% away from the ideal line, which is considered small in such large scale. These results lead us to further investigate the system performance by exploring a wider variety of scientific workloads fixing the computing resources in its maximum, as in all following experiments.

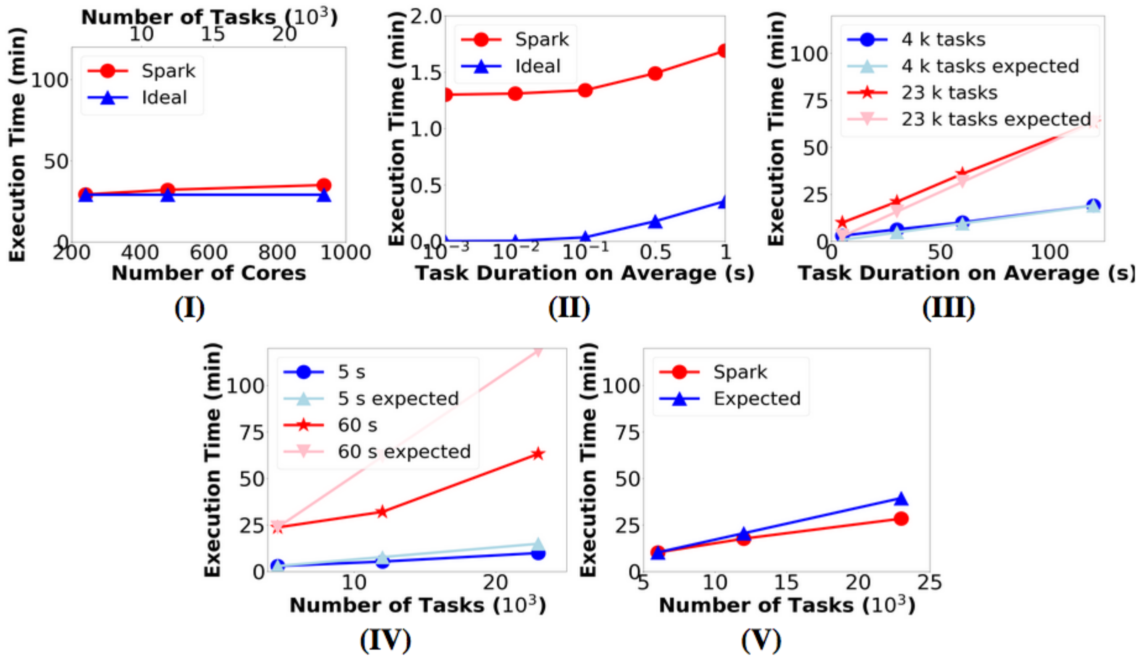

Figure 3. Experiments' results.

Experiment 2: varying task duration from very short to short-term. Although in MTC workflows tasks are commonly long-term (*i.e.*, many seconds or few minutes on average each) [Raicu et al. 2008], we evaluate Spark's scalability on 20 k tasks, varying task duration on average from very short (taking few milliseconds) to short (up to 1 second). These workloads are present in some scientific applications that perform very simple processing. By running short tasks, we are stressing the task scheduling system. Results are in Figure 3-II. We calculate the ideal time as *(Number of Tasks * Tasks Duration on Average)/Number of Cores*. Results show that Spark runs considerably below the ideal time. Spark has a higher initialization time (it is scheduling to all 936 cores) compared to the workload being computed (the entire computation takes at most 2 minutes only). When this happens, Spark's performance is compromised, as the best result (80% away from ideal) happened for 1-second tasks. It indicates that when the duration increases, Spark's performance gets slightly closer to the ideal line.

Experiment 3: fixed number of tasks, varying task duration. Experiment 2 indicates that when task duration increases, Spark's performance tends to increase. In this experiment, we further analyze this. We vary average task duration, from short (5 s) to long (120 s), fixing two different numbers of tasks: small (4.6 k) and large (23.4 k, about 5 times 4.6 k). Based on Experiment 2 results, we plot two expected lines by setting the base result (where Spark achieves best performance) as the longest task durations evaluated, *i.e.*, 120 s on average, and then we expect Spark's performance to degrade when task duration decreases. As Figure 3-III shows, this is exactly what happens. To understand the results, we analyze the differences between the expected lines and the actual lines. In this experiment, when these differences are small, Spark maintains its high scalability for each workload analyzed. Ideally, this difference should remain small. For both fixed number of tasks, these differences vary similarly. However, in both cases we see that the shorter the task duration, the greater the difference. For 60-seconds tasks, the differences are about 10%, whereas for 5-seconds tasks, the difference is 73%. The reasons for this are that there are many more frequent concurrent writes to disk during execution and the task scheduling system is overloaded.

Experiment 4: Fixed task durations, varying number of tasks. In this experiment, we fix two task durations (5 s and 60 s) and increase the number of tasks from small (4.6 k), to mid (12 k), and large (23.4 k). We expect that execution time increases proportionally to the number of tasks. Results are in Figure 3-IV. We also analyze the difference between actual and expected results. In this experiment, the larger these differences, the faster Spark runs than the expected time. That is, Spark is more scalable when these differences are larger. The differences between expected and actual time does not vary significantly when number of tasks increases. Thus, Spark's performance is less sensitive to the number of tasks. However, for 23.4 k tasks, the difference between expected and actual time for 60 s task duration is 46%, whereas for 5 s task duration, the difference is 33%. Therefore, Spark has better scalability for longer tasks and number of tasks variation does not significantly affects its performance.

Experiment 5: Mixed task durations, varying number of tasks. Up to this point, we analyzed Spark's scalability under different conditions by executing synthetic workloads. Now, we investigate a more realistic scenario. We analyze a real workload for the RFA workflow, which mixes short, mid, and long-term tasks. In this workload, on average, *Uncompress* tasks take 1 s each; *Pre-processing* tasks take 10 s, *Analyze Risers* tasks take 60 s, *Calculate Wear and Tear* tasks take 30 s, *Analyze Position* tasks take 15 s, *Join Results* takes 1 s, and *Compress Results* tasks take 15 s. We vary number of tasks as 6 k, 12 k, and 23.4 k. We expect that the execution time increases proportionally to the number of tasks growth. Figure 3-V shows that Spark scales well on all executions as it runs faster than the expected. The difference between actual and expected computations increases as the number of tasks increases. An execution for the 23.4 k tasks workload takes about 28 min to complete, whereas the expected time would be 39.4 min (28% difference). Thus, Spark scales well for a real scientific workload.

4. Conclusion

In this paper, we analyzed Spark's scalability when running a typical HPC scientific workflow that processes large amounts of data, does multiple file writes in a shared file system, and is implemented as black-box scientific programs linked through dataflows. As mentioned in [Zhang et al. 2017], reusing existing codes avoids rewriting, which is error prone. We use a real case-study workflow in Oil and Gas domain on a large HPC cluster with 936-cores and a fast shared file system. We showed that Spark scales well, even for a real workload. We found that the number of tasks does not significantly affect its performance, but the task duration does. For multiple short tasks representing simple scientific tasks with little computation, Spark does not scale well, whereas for long-lasting scientific tasks, it does. Since long-lasting tasks are more common in scientific workflows, using Spark to manage the dataflow between tasks with black-box scientific application external calls does not harm its scalability. Altogether, in addition to its large user community and easy APIs, its scalability encourages scientific workflow developers to migrate their many-task parameter sweep workflows to Spark, without requiring re-implementation of the complex linked scientific programs. As future work, we plan to carry out more and deeper performance analyses to investigate load balancing, memory utilization, and disk throughput during execution in different scientific workloads.

Acknowledgements

This work was partially funded by CNPq, FAPERJ and Inria (MUSIC and SciDISC projects), EU H2020 Programme and MCTI/RNP-Brazil (HPC4E grant no. 689772), and performed (for P. Valduriez) in the context of the Computational Biology Institute (www.ibc-montpellier.fr). The experiments were carried out using the Inria Grid'5000 testbed (https://www.grid5000.fr).

References

Armbrust, M., Zaharia, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., et al., (2015), "Scaling spark in the real world: performance and usability", *PVLDB*, v. 8, n. 12, p. 1840–1843.

Atkinson, M., Gesing, S., Montagnat, J., Taylor, I., (2017), "Scientific workflows: past, present and future", *FGCS*, v. 75, p. 216–227.

F. da Silva, R., Filgueira, R., Pietri, I., Jiang, M., Sakellariou, R., Deelman, E., (2017), "A characterization of workflow management systems for extreme-scale applications", *FGCS*, v. 75, p. 228–238.

GitHub. RFA Spark Repository. Available on: github.com/hpcdb/RFA-Spark.

Gittens, A., Devarakonda, A., Racah, E., Ringenburg, M., Gerhardt, L., Kottalam, J., Liu, J., Maschhoff, K., Canon, S., et al., (2016), "Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+MPI using three case studies". In: *IEEE Int. Conf. on Big Data*, p. 204–213

Oliveira, D., Boeres, C., Neto, A., Porto, F., (2015), "Avaliação da localidade de dados intermediários na execução paralela de workflows bigdata". In: *SBBD*, p. 29–40

Özsu, M. T., Valduriez, P., (2011), *Principles of distributed database systems*. 3 ed. New York, Springer.

Raicu, I., Foster, I. T., Zhao, Y., (2008), "Many-task computing for grids and supercomputers". In: *MTAGS*, p. 1–11

Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., Özcan, F., (2015), "Clash of the titans: MapReduce vs. Spark for large scale data analytics", *PVLDB*, v. 8, n. 13, p. 2110–2121.

Souza, R., Silva, V., Coutinho, A. L. G. A., Valduriez, P., Mattoso, M., (2016), "Online input data reduction in scientific workflows". In: *WORKS*, p. 44–53

Zhang, Z., Barbary, K., Nothaft, F. A., Sparks, E. R., Zahn, O., Franklin, M. J., Patterson, D. A., Perlmutter, S., (2017), "Kira: processing astronomy imagery using big data technology", *IEEE Trans. Big Data*, v. PP, n. 99, p. 1–14.