A Methodology for Spark Parameter Tuning

Anastasios Gounaris[a], Jordi Torres[b]

[a] Department of Informatics, Aristotle University of Thessaloniki, Greece
gounaria@csd.auth.gr
[b] Department of Computer Architecture, Technical University of Catalonia, Spain
torres@ac.upc.edu

Abstract

Spark has been established as an attractive platform for big data analysis, since it manages to hide most of the complexities related to parallelism, fault tolerance and cluster setting from developers. However, this comes at the expense of having over 150 configurable parameters, the impact of which cannot be exhaustively examined due to the exponential amount of their combinations. The default values allow developers to quickly deploy their applications but leave the question as to whether performance can be improved open. In this work, we investigate the impact of the most important tunable Spark parameters with regards to shuffling, compression and serialization on the application performance through extensive experimentation using the Spark-enabled Marenostrum III (MN3) computing infrastructure of the Barcelona Supercomputing Center. The overarching aim is to guide developers on how to proceed to changes to the default values. We build upon our previous work, where we mapped our experience to a trial-and-error iterative improvement methodology for tuning parameters in arbitrary applications based on evidence from a very small number of experimental runs. The main contribution of this work is that we propose an alternative systematic methodology for parameter tuning, which can be easily applied onto any computing infrastructure and is shown to yield comparable if not better results than the initial one when applied to MN3; observed speedups in our validating test case studies start from 20%. In addition, the new methodology can rely on runs using samples instead of runs on the complete datasets, which render it significantly more practical.

Keywords: Spark configuration, parameter tuning, shuffling

1. Introduction

Spark [1, 2] has emerged as one of the most widely used frameworks for massively parallel data analytics. In summary, it improves upon Hadoop MapReduce in terms of flexibility in the programming model and performance [3], especially for iterative applications. It can accommodate both batch and streaming applications, while providing interfaces to other established big data technologies, especially regarding storage, such as HDFS and NoSQL databases. Finally, it includes components for SQL-like processing, graph processing, machine learning and data mining. However, its key feature is that it manages to hide the complexities related to parallelism, fault-tolerance and cluster setting from end users and application developers. This feature renders Spark practical for use in real-life data science and big data processing applications.

To support all these, Spark execution engine has been evolved to an efficient albeit complex system with more than 150 configurable parameters. The default values are usually sufficient for a Spark program to run, e.g., not to run out of memory without having the option to spill data on the disk and thus crash. But this gives rise to the following research question: *"Can the default configuration be improved and, if yes, how better configurations can be set efficiently?"*

The aim of this work is firstly, to provide evidence that the answer to the first part of the above question is affirmative, and then, to answer the second part in an efficient manner. Clearly, it is practically impossible to check all the different combinations of parameter values for all tunable parameters. Therefore, tuning arbitrary Spark applications by inexpensively navigating through the vast search space of all possible configurations in a principled manner is a challenging task. Very few research endeavors focus on issues related to understanding the performance of Spark applications and the role of tunable parameters [4, 5, 6]. For the latter, Spark's official configuration guides[1] and tuning[2] guides and tutorial book [7] provide a valuable asset in understanding the role of every single parameter.

Understanding the role of a parameter does not necessarily mean that the impact of each parameter on the performance of arbitrary applications is understood as well. Moreover, such an understanding does not imply that tuning is straightforward. An added complexity stems from the fact that most parameters are correlated and the impact of parameters may vary from application to application and it will also vary from cluster to cluster. In this work, we experiment with the MareNostrum III (MN3) petascale supercomputer at the Barcelona Supercomputing Center. After configuring the cluster in an application-independent way according to the results in [6], we examine the impact of configurable parameters with regards to shuffling, compression and serialization on a range of applications with a view to deriving a simple yet systematic tuning methodology

[1] `http://spark.apache.org/docs/latest/configuration.html`
[2] `http://spark.apache.org/docs/latest/tuning.html`

that can be applied to each application separately.

We build upon our previous work in [8], where: (i) We identified the most important parameters with regards to shuffling, compression and serialization in terms of their potential impact on performance and we tested them on MN3. The number of these parameters is 12. (ii) We summarized our experience in a tuning methodology to be applied on an individual application basis. The methodology treats applications as black boxes, follows an efficient trial-and-error approach that involves a low number of experimental runs for just 10 different configurations at most, and leverages the correlation between different parameters.

This article is a heavily extended version of the results in [8] (also repeating all experiments from scratch). The new contributions are summarized as follows:

- We provide new experimental evidence that the default Spark configuration leaves room for performance improvements when tuning the parameters under investigation. We also evaluate how these parameters are correlated.
- We propose a new tuning methodology that can be applied to any computing infrastructure; the new methodology serves as an alternative to the one in [8], which mostly reflected our experience with MN3, did not profile parameter correlations explicitly and is questionable whether it generalizes efficiently.
- We validate and compare the performance impacts of the new methodology and the one in [8]. Our new proposal can yield comparable if not better results than the initial one when applied to MN3. The observed speedups in our validating test case studies start from 20% and reach up to more than 4 times, when compared against the default MN3 configuration.
- With a view to rendering the proposal more practical, we provide evidence that the new methodology can rely on runs using samples instead of runs on the complete datasets.

The remainder of this work is structured as follows. The next section provides an overview of Spark and of the known results to date with regards to Spark tuning. In Section 3, we explain the chosen parameters and we present the methodology in [8]. Our new methodology along with its instantiation on MN3 is presented in Section 4. Section 5 deals with the evaluation of the methodologies. We conclude in Section 6.

2. Overview of Existing Results for Spark Configuration

Apache Spark is an open source massively parallel computing framework. It provides an interface enabling users to develop and deploy applications to run in parallel on clusters of machines, which typically adopt the shared-nothing parallel architecture [9]. For cluster management, the list of supported options includes deploying on an Amazon EC2 cloud cluster instantiated on the fly, employing a third-part manager, such as YARN or MESOS, or launching a standalone cluster. Overall, there are over 150 tunable parameters that define execution details.

2.1. Spark basics

Spark operates on data collections abstracted as Resilient Distributed Datasets (RDDs), which are partitioned across several nodes. A Spark application consists of two types of operations, namely *transformations* and *actions*. The former apply a function on each RDD element and result in a new RDD. Actions trigger the execution of such functions and produce meaningful results. For each action in the application a *job* is performed. For each job, typically several RDD transformations need to be computed. Spark's scheduler creates a physical execution plan for the job based on the directed acyclic graph (DAG) of transformations. The physical plan is divided into *stages*. A *stage* is a sequence of transformations that can be pipelined. The sequence of computations defined by a *stage* instantiated over a single data partition is called a *task*. A *task* is the actual unit of execution of the physical plan. The task scheduler assigns tasks to parallel *workers* via the cluster manager. On each worker node, each applications launches its own *executors*, which are responsible for task execution.

Data may be *repartitioned* across RDDs. This may be done as a result of a specific transformation, such as *groupByKey* and *sortByKey*, which result in a new RDD, where data are partitioned across machines differently. This data re-distribution is commonly referred to as data *shuffling*. Data shuffling is an expensive operation. First it incurs communication cost. Second, it incurs CPU cost, because it involves data serialization. Third, it may incur I/O cost, because it may store temporary data on disk if they cannot fit in main memory; temporary files are kept as long as they are needed for fault tolerance purposes. As such, the cost is multi-dimensional, while memory is stressed as well.

Table 1 provides a categorization of Spark parameters. In this work, we target parameters belonging to the *Shuffle Behavior* and *Compression and Serialization* aspects, which greatly contribute to a Spark application's running time, as supported by our experimental results, the official documentation, and the evidence provided in other works, such as [4, 5, 3]. Note that there are several other parameters belonging to categories such as *Application Properties, Execution Behavior* and *Networking* that may affect the performance, but these parameters are typically set at the cluster level, i.e., they are common to all applications running on the same cluster of machines, e.g., as shown in [6].

Next, we summarize the known results to date with regards to Spark configuration. These results come from three types of sources: (i) academic works that aimed at Spark configuration and profiling; (ii) official Spark documentation and guides that build on top of this documentation; and (iii) academic works that include evaluation of Spark applications on real platforms and, as a by-product, provide information about the configuration that yielded the highest performance. We also briefly discuss results on Spark profiling, because they directly relate to Spark configuration. The most relevant work to ours is the

Category	Description
Application Properties	The parameters in this group concern basic application properties, such as the application name, the CPU and the memory resources that will be allocated to the coordinating process (called driver), the memory for each executor process that runs the actual computations, and so on.
Runtime Environment	The parameters belonging to this group refer to environment settings, such as classpaths, java options and logging.
Shuffle Behavior	These parameters have to do with the shuffling mechanism of Spark, and they involve buffer settings, sizes, shuffling methods, memory allocated to shuffling, and so on.
Spark UI	The UI parameters are mostly related to UI event-logging.
Compression and Serialization	The parameters of this group target compression and serialization issues. They mostly have to do with whether compression will take place or not, what compression codec will be used, the codec's parameters, the serializer to be used and its buffer options.
Execution Behavior	The parameters of this group refer to a wide range of execution details including the number of execution cores and data parallelism.
Networking	This group contains parameters that provide options for network issues. Most parameters refer to timeout options, heartbeat pauses, ports and network retries.
Scheduling	Most parameters in the group cover scheduling options. Some noteworthy parameters have to do with scheduling mode, and whether to use the speculation optimization and the maximum number of CPU cores that will be used.
Dynamic Allocation	By using dynamic allocation, Spark can increase or decrease the number of executors based on its workload. This is only available in Yarn (which we do not use in this work).
Security	These parameters deal with authentication issues.
Encryption	These properties refer to encryption algorithms, passwords and keys that may be employed.
Spark Streaming and SparkR	These parameters are specific to the Spark Streaming and SparkR higher-level components.

Table 1: Summary of parameter categories

study of Spark performance on the MN3 in [6], which is complementary to this work and presented separately.

2.2. Optimization of Spark on MN3.

The work in [6] sheds lights onto the impact of configurations related to parallelism. In MN3, cluster management is performed according to the standalone mode, i.e., YARN and MESOS are not used. The main results, which are reused in our work, are summarized as follows. First, the number of cores allocated to each Spark executor has a big impact on performance and should be configured in an application-independent manner. In other words, all applications sharing the same cluster can share the same corresponding configuration. Second, the level of parallelism, i.e., the number of partitions per participating core, plays a significant role. In cpu-intensive applications, such as k-means, allocating a single data partition per participating core yields the highest performance. In applications with a higher proportion of data-shuffling, such as sort-by-key, increasing the number of data partitions to two per core is recommended. A range of additional aspects, e.g., using Ethernet instead of Infiniband, have been investigated, but their significance was shown to be small. Finally, the type of the underlying file system affects the performance, however, this is a property of the infrastructure rather than a tunable parameter. In a more recent work [10], dynamic parallelism issues are examined; in this work, we consider fixed degrees of parallelism throughout application execution, we reuse the earlier results, and we focus on the additional issues of shuffling, compression and serialization.

2.3. Guides from Spark documentation.

Spark official documentation presents a summary of tuning guidelines that can be summarized as follows. (i) The type of the serializer is an important configuration parameter. The default option uses Java's framework, but if `kryo` library is applicable, it may reduce running times significantly. (ii) The memory allocated to main computations and shuffling and the memory allocated to caching. It is stated that *"although there are two relevant configurations, the typical user should not need to adjust them as the default values are applicable to most workloads"*. Memory-related configurations are also related to the performance overhead of garbage collection (GC). (iii) The level of parallelism, i.e., the number of tasks in which each RDD is split needs to be set in a way that the cluster resources are fully utilized. (iv) Data locality, i.e., enforcing the processing to take place where data resides, is important for distributed applications. In general, Spark scheduling respects data locality, but if a waiting time threshold is exceeded, Spark tasks are scheduled remotely. These thresholds are configurable. Apart from the tuning guidelines above, certain other tips are provided by the Spark documentation, such as preferring arrays to hashmaps, using broadcasted variables, and mitigating the impact of GC through caching objects in serialized form. Also, similar guidelines are discussed more briefly in [7].

In our work, we explicitly consider the impact of serialization and memory allocation. Tuning the parallelism degree is out of our scope, but we follow the guidelines in [6], so that resources are always occupied and data partitioning is set in a way that is efficient for the MN3 supercomputer. Also, we do not deal with GC and locality-related thresholds; the latter have not seemed to play a big role in our experiments.

Based on these guidelines, Alpine Data has published online a so-called *cheat-sheet*[3], which is a tuning guide for system administrators. The guide is tailored to the YARN cluster manager. Compared to our tuning model, it is more complex, and contains checks from logs and tips to modify the application code and setting of configuration parameters per application. The configuring parameters refer to data partitioning and the potential use of the `kryo` serializer. By contrast, we regard each application as being a black box requiring significantly fewer comparisons, we resort to [6] for data partitioning issues, and we consider 11 more tuning parameters apart from `kryo`.

2.4. Additional sources for Spark configuration.

In [5], the impact of the input data volume on Spark's applications is investigated. The key parameters identified were related to memory and compression, although their exact impact is not analyzed. By contrast, we examine a superset of these parameters. An interesting result of this study is that GC time does not scale linearly with the data size, which leads to performance degradation for large datasets. In addition, it is shown that if the input data increases, the idle cpu time may increase as well. The work in [11] is similar to [6] in that it discusses the deployment of Spark on a high-performance computing (HPC) system. In its evaluation, it identifies four key Spark parameters along with the application-dependent level of parallelism. All these parameters are included in our discussion. Finally, the work in [12] focuses on Spark's shuffling optimization using two approaches. The first one is through columnar compression, which however does not yield any significant performance improvement. The second approach employs file consolidation during shuffling. We consider file consolidation as well.

2.5. Profiling Spark applications and other related work.

A thorough investigation of bottlenecks in Spark is presented in [4]. The findings are interesting, since it is claimed that many applications are CPU-bound and memory plays a key role. In our experiments, we perform memory fine-tuning, since we investigate parameters that affect the memory usage.

In [13], the authors present their tuning methodology, termed as Active Harmony. Given a range of parameters, Active Harmony can tune and improve performance. Due to the fact that, during parameter tuning, the search space can become extremely large and time consuming, the parameters that are considered most important are prioritized. Splitting parameters based on the effect they have on performance is a common strategy, an approach that we also follow in our work. Each parameter can be considered as an extra dimension in the search space that needs to be tuned, so eliminating the ones that do not have or may not have any significant impact is important. To identify such parameters, the authors in [13] test the sensitivity of the application by changing each parameter's values. In general, this approach is effective when there is minimal correlation between parameters. However, as shown in our case, tuning parameters are correlated in general.

The work in [14] deals with the issue of parameter optimization in workflows. The main motivation behind this work is the fact that, while for small workflows a trial-and-error optimization method can be applied, when more complex workflows are considered, the combined effect of multiple parameters cannot be easily identified. The authors employ genetic algorithms to navigate through the space of possible parameter configurations. However, such a solution may involve far too many experimental runs, whereas we advocate a limited number of configuration runs independently from the application size.

Finally the issue of parameter optimization is also addressed in [15]. Motivated by two applications, namely pixel intensity quantification and neuroblastoma classification, the parameters that impact the performance of spatial data analysis applications are presented and classified. This classification includes two categories, quality-preserving parameters and quality-trading parameters. The authors also introduce an integrated framework for performance optimization in multiple dimensions of the parameter space and conduct and experimental evaluation. The results of tuning parameters of both categories yield improvements for both the applications. In our work, the parameters do not affect the quality of the result, as we only focus on performance improvement.

3. The Parameters of Interest and an Initial Tuning Methodology

Based on evidence from (i) the documentation and the earlier works presented in Section 2 and (ii) our previous work in [8], we narrow our focus on 12 parameters related to shuffling, compression and serialization. The configuration of these parameters need to be investigated according to each application instance separately; i.e., even the same program operating on different datasets may need different configuration.

1. `spark.serializer:` In Spark's documentation, it is stated that *KryoSerializer* is thought to perform much better than the default Java serializer, when speed is the main goal. However, the Java serializer is still the default choice, so this parameter needs to be considered. Checking whether to enabling the *kryo* serializer is also the recommendation of Alpine Data's *cheat-sheet*.

2. `spark.shuffle.manager:` The available implementations are three: *sort, hash* , and *tungsten-sort*. In the most recent versions of Spark, *tungsten-sort* is the default option, as it is reported to yield the highest performance in general provided that certain requirements are met. It improves upon simple *sort Hash*

[3]available from `http://techsuppdiva.github.io/spark1.6.html`

creates too many open files for certain inputs and aggravates memory limitations. However, enabling the `spark.shuffle.consolidateFiles` parameter along with the *hash* manager may mitigate this problem. Overall, there is no clear winner among the shuffle manager options. In our methodology, we choose between *tungsten-sort* (default) and *hash*.

3. `spark.shuffle.memoryFraction:` If, during shuffling, spills are often, then this value should be increased from its default to allow for more memory-resident temporary files during shuffling. On the contrary, when it is decreased, more memory can be used to cache RDDs. Since this parameter is directly linked to the amount of memory that is going to be utilized, it may have a high performance impact. However, any increase is at the expense of the next parameter, i.e., these two parameters need to be considered in combination. The default value is 20%, and we also investigate values of 10% and 30% (50% less and more, respectively).

4. `spark.storage.memoryFraction:` The default value is 60%. When we decrease `spark.shuffle.memoryFraction` to 10%, `spark.storage.memoryFraction` is increased to 70%. Similarly, for `spark.shuffle.memoryFraction` set to 30%, `spark.storage.memoryFraction` is set to 50%.

5. `spark.reducer.maxSizeInFlight:` If this value is increased, reducers would request bigger output chunks during shuffling. This would increase the overall performance but may aggravate the memory requirements. So, in clusters that there is adequate memory and where the application is not very memory-demanding, increasing this parameter could yield better results. On the other hand, if a cluster does not have adequate memory available, reducing the parameter should yield performance improvements. The default value is 48MB, and we investigate values of 50% more and less, i.e., 72MB and 24 MB, respectively.

6. `spark.shuffle.file.buffer:` The role of this parameter bears similarities to the `spark.shuffle.maxSizeInFlight` parameter, i.e., if a cluster has adequate memory, then this value could be increased in order to get higher performance. If not, there might be performance degradation, since too much memory would allocated to buffers. The default value is 32KB, and we also investigate configurations of 48KB and 16 KB.

7. `spark.shuffle.compress:` In general, compressing data before they are transferred over the network is a good idea, provided that the time it takes to compress the data and transfer them is less than the time it takes to transfer them uncompressed. But if transfer times are faster than the CPU processing times, the main bottleneck of the application is shifted to the CPU and the process is not stalled by the amount of data that are transferred over the network but from the time it takes for the system to compress the data. Clearly, the amount of data transmitted during shuffling is application-dependent, and thus this parameter must not be configured to a single value for all applications. The default value is to compress data during shuffling and we investigate the option to disable shuffle compression.

8. `spark.io.compression.codec:` Three options are available, namely *snappy, lz4,* and *lzf.* Although there are many tests conducted by various authors for the generic case, the best performing codec is application-dependent. The first option is the default one, and we also investigate whether employing any of the two remaining can lead to performance improvements.

9. `spark.shuffle.consolidateFiles:` This parameter provides the option of consolidating intermediate files created during a shuffle, so that fewer files are created and performance is increased. It is stated however that, depending on the filesystem, it may cause performance degradation.

10. `spark.rdd.compress:` By default, RDDs are not compressed. The trade-offs with regards to this parameter are similar to those for shuffle compress. However, in this case, the trade-off lies between CPU time and memory.

11. `spark.shuffle.io.preferDirectBufs:` In environments where off-heap memory is not tightly limited, this parameter may play a role in performance. By default it is enabled, but it may be worth disabling it to force all shuffle-related memory allocations to be on-heap.

12. `spark.shuffle.spill.compress:` As for the previous compression options, a trade-off is involved. If transferring the uncompressed data in an I/O operation is faster than compressing and transferring compressed data, then this option should be set to false. Provided that there is a high amount of spills, this parameter may have an impact on performance. The default option is spill compression to be enabled.

3.1. An Initial Trial-and-Error Methodology

Based on (i) the results of thousands of hours of experimentation on the MN3 platform, where, in principle, each parameter was tested in isolation using three representative benchmarking applications, and (ii) the expert knowledge as summarized in Section 2, an easily applicable tuning methodology has been presented in [8]. This methodology is presented in Fig. 1 in the form of a block diagram. In the figure, each node represents a *test run* with one or two different configurations. Test runs that are higher in the figure are expected to have a bigger impact on performance and, as a result, a higher priority. As such, runs start from the top and, if an individual configuration improves the performance, the configuration is kept and passed on to its children replacing the default value for all the test runs in the same path. If an individual configuration does not improve the

Id	Parameter name	Value
1	`spark.serializer`	KryoSerializer
2	`spark.shuffle.manager`	hash
3	`shuffle.memoryFraction, storage.memoryFraction`	0.3, 0.5
4	`shuffle.memoryFraction, storage.memoryFraction`	0.1, 0.7
5	`spark.reducer.maxSizeInFlight`	72mb
6	`spark.reducer.maxSizeInFlight`	24mb
7	`spark.shuffle.file.buffer`	48k
8	`spark.shuffle.file.buffer`	16k
9	`spark.shuffle.compress`	false
10	`spark.io.compress.codec`	lzf
11	`spark.io.compress.codec`	lz4
12	`spark.shuffle.consolidateFiles`	true
13	`spark.rdd.compress`	true
14	`spark.shuffle.io.preferDirectBufs`	false
15	`spark.shuffle.spill.compress`	false

Table 2: Parameters values tested

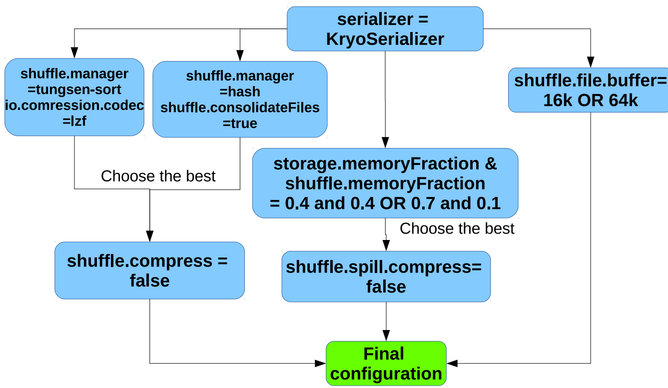

Figure 1: The spark parameter tuning methodology proposed in [8]

performance, then the configuration is not added and the default is kept. In other words, each parameter configuration is propagated downstream up to the final configuration as long as it yields performance improvements.

Overall, as shown in the figure, at most ten configurations need to be evaluated referring to nine of the parameters in Section 3; the remainder three parameters are discarded, because their impact was shown to be small during benchmarking. Note that, even if each parameter took only two values, exhaustively checking all combinations would result in $2^9 = 512$ runs. Finally, the methodology can be employed in a less restrictive manner, where a configuration is chosen not only if it improves the performance, but, in addition, if the improvement exceeds a threshold, e.g., 5% or 10%.

4. A More Systematic Tuning Methodology

The methodology in Figure 1 has been effective when tested on MN3 as shown in [8], but suffers from the following two main limitations:

- It is iterative in the sense that configurations at the lower parts can be tested only after configurations at the upper parts of the figure have been completed. As such, deriving the final configuration is a long-lasting process.
- It is built based on experimental evidence regarding the performance impact of the parameters, when the latter are tested in isolation or combined only with the *kryo* serializer. The combinations of the parameters in the diagram of Figure 1 basically reflect the experience of the authors in [8] with Spark applications on MN3 and thus may not generalize efficiently for arbitrary platforms.

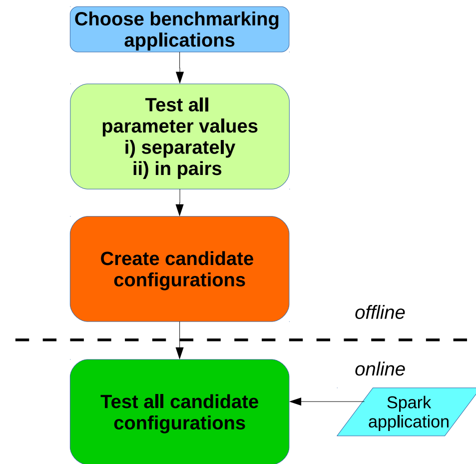

Figure 2: The new spark parameter tuning methodology

We address the above concerns through an alternative methodology, which firstly, directly profiles the impact of pairs of parameters on benchmarking applications and applies a graph algorithm to create complex candidate configurations, and then, follows a different approach to deriving the final configuration for each application instance: namely, a set of candidate parameter settings is checked in parallel in a single shot and the best performing one is chosen (see Figure 2).

Algorithm 1 Outline of creation of candidate configurations

1: Add the best performing parameter for each benchmarking application to the set of candidate configurations CC
2: Add the best performing parameter pair for each benchmarking application to CC
3: Discretize relative differences from default
4: Detect beneficial pairs
5: Keep mutually beneficial pairs
6: Construct more complex configurations and add them to CC
7: **return** CC

Difference Interval	Value	Difference Interval	Value
$(-2\%, 0\%]$	0	$(0\%, 2\%]$	0
$(-5\%, -2\%]$	-1	$(2\%, 5\%]$	1
$(-10\%, -5\%]$	-2	$(5\%, 10\%]$	2
$(-15\%, -10\%]$	-3	$(10\%, 15\%]$	3
$(-20\%, -15\%]$	-4	$(15\%, 20\%]$	4
$(-30\%, -20\%]$	-5	$(20\%, 30\%]$	5
$(-\infty\%, -30\%]$	-6	$(30\%, \infty]$	6

Table 3: Mapping of relative differences from the default to discrete values

Algorithm 2 Constructing more complex configurations (Step 6 in Algorithm 1)

```
1: L ← sorted triangles
2: CCcomplex ← empty bit-array with 15 columns
3: while L ≠ ∅ do
4:     flag ← 1
5:     for tr(i, j, k) ∈ L do
6:         if there exists row in CCcomplex where at least 2 of the 3 places
           i, j, k are 1 then
7:             Set all 3 places i, j, k to 1 in this row
8:             L ← L − tr
9:         else if flag = 1 then
10:            Create a new row in CCcomplex with places
               i, j, k set to 1
11:            flag ← 0
12:            L ← L − tr
13:        end if
14:    end for
15: end while
16: Add CCcomplex configurations to CC
```

Overall, our new methodology consists of four phases. The first three are offline and are shared among all Spark applications to run on the same cluster, while the fourth derives the final parameter configuration for a specific application instance. These phases are detailed below:

(I) *Choice of benchmarking applications (offline):* a limited number of representative benchmarking applications, denoted as $Apps$, need to be selected and tested in a setting (i.e., a combination of input dataset and number of nodes participating in the execution), where the memory resources are stressed. Moreover, the applications need to cover both shuffling and computation intensive applications. This phase is similar to the one in the methodology in Section 3.1.

(II) *Testing of the parameter values (offline):* all parameter values need to be tested for all benchmarking applications. The different parameter settings are 15 as shown in Table 2. The parameters are tested in two modes: individually and in compatible pairs; non-compatible pairs refer to values of the same parameter, e.g., 10 and 11 in the table. Overall, for each benchmarking application, 117 runs are required (15 for single parameters + 101 for compatible pairs + 1 for the default values). In each run, the running time of the benchmarking application is recorded. The number of runs is high but not prohibitive given that these experiments are required only once per computing platform. Compared to the methodology in Section 3.1, parameter pairs are explicitly profiled, so that correlation of parameters can be analyzed more systematically.

(III) *Creation of candidate configurations (offline):* This is a 6-step process, outlined in Algorithm 1. The first 2 steps are trivial and select the best performing parameters from the profiles. The rationale of the next 4 steps is to regard the relationships between parameters as edges in a graph, in which the parameters are the vertices; this is done in order to construct more complex parameter configurations, where more than 2 parameters are modified at the same time. More specifically, the steps are as follows.

(1) For each benchmarking application, the best performing parameter is added to the candidate set.

(2) Similarly, the best performing parameter pair is added to the candidate set.

(3) The profiles of parameters for each application can be represented as a 15×15 triangular matrix, where the cell (i, j) corresponds to the pair of the ith and j^{th} parameter in Table 2. The results of the tests of each parameter individually are in the diagonal. Then, each result is normalized according to its relative difference from the default configuration. Negative differences denote speedups, while positive differences denote cases, where applying a parameter yields longer execution times. Finally, the differences are discretized, as shown in Table 3. The resulting matrices are denoted as D^m, $m = 1 \ldots |Apps|$

(4) In this step, a single binary 15×15 non-triangular matrix B is derived through processing of the matrices containing the discretized values for each application of the previous step. Moreover, each cell $B(i, i)$ is 1 if there exists at least one D^m, for which $D^m(i, i) \leq t$, where t is a cutoff (negative) threshold. $B(i, j), i \neq j$ is 1 if there exists at least one D^m, for which $D^m(i, j) \leq t$ and $D^m(i, j) - D^m(i, i) \leq -1$. Otherwise, the cell values are 0. In our cases, typical values of t are -2 or -3. The rationale is each cell of $B(i, j)$ to denote whether a given combination of

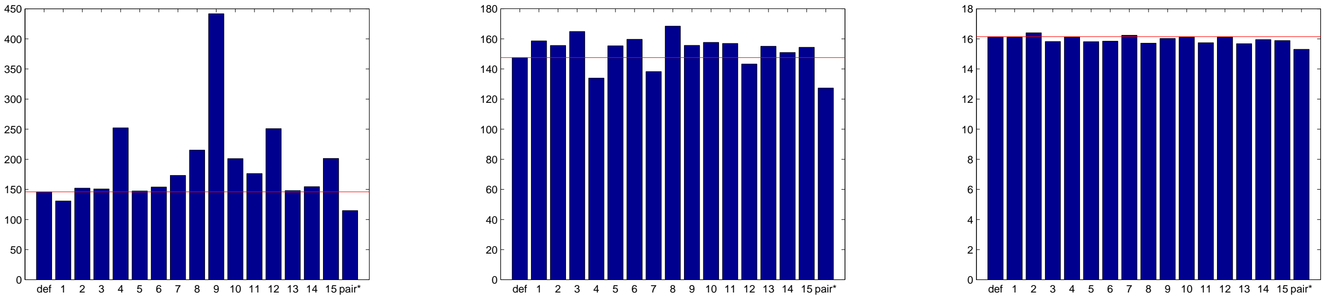

Figure 3: Results of profiling for sort (left), shuffle (middle) and k-means (right).

parameters leads to significant improvements, while at the same time, adding parameter j to a configuration containing parameter i leads to performance improvements by at least one discretized scale.

(5) This step keeps only combinations of parameters that both assist each other in decreasing running time by at least one discretized scale. A new binary matrix B' is derived, where $B'(i, j) = 1$ iff $B(i, j) = B(j, i) = 1$.

(6) The resulting B' matrix of the previous step is treated as an adjacency matrix of an undirected graph, where vertices are parameters and edges denote positive correlation between parameters. Then, we extract all triangles of such graph. Each triangle is denoted as a triple of parameter identifiers (i, j, k) where $1 \leq i < j < k \leq 15$. For each triangle, we assign a score equal to the $min_{1 \leq m \leq |App|}(D^m(i, j) + D^m(i, k) + D^m(j, k))$. This score reflects the highest performance improvement due to the pairs in the triangle in any benchmarking application. The triangles are then sorted by the score in ascending order, after pruning all triangles with non-negative score. Then a greedy algorithm groups triangles. The algorithm is presented in Algorithm 2. It parses the list with the triangles and adds triangles to a bit-array matrix, where each row corresponds to a distinct configurations with 3 or more parameters of Table 2. Its running time is $O(n^2)$, where n is the number of triangles.

(IV) *Testing of candidate configurations (online):* for each application instance, i.e., combination of Spark application and dataset, all the candidate configurations are tested in parallel. The best performing one is kept for future reuse. Compared to the iterative methodology in Section 3.1, the configuration to be adopted is not built step-by-step through additional profiling experiments, but is selected from a pre-defined pool, the candidate set created in the previous phase. In order to decrease the configuration overhead, the test runs can be performed using samples instead of the complete input dataset.

The above methodology yields at most $|App|$ single-parameter configurations, and $|App|$ 2-parameter configurations; moreover, the number of more complex configurations is limited in practice. Overall, its main advantage over the iterative methodology in Section 3.1 is that, apart from addressing the limitations at the beginning of the description (i.e., iterative nature and ad-hoc-ness in leveraging parameter correlations), it can adapt to the profiling results different computing platforms.

4.1. Applying the methodology to MN3

Hereby, we describe the application of the first three phases of our proposed methodology to MN3. We employed three benchmark applications: (i) *sort-by-key*; (ii) *shuffling* and (iii) *k-means*. *K-means* and *sort-by-key* are also part of the HiBench benchmark[4] [16]. These applications were selected because they can be considered as representative of a variety of applications given that they cover both cpu- and shuffling-intensive cases. *Sort-by-key* is both computation- and shuffling-intensive. It was tested on 100GB of raw data, and more specifically 1 billion of 100-byte records, where the key is 10 bytes. *Shuffling* was tested on the same dataset but without performing any sorting; as such, it is only shuffling-intensive. Finally, *k-means* is CPU-intensive. Its input dataset is 100 million 200-dimensional double vectors. In all applications, 20 16-core dedicated machines were employed. MN3 hardware details are described in the Appendix. According to the results of [6], in the first two applications, the RDDs were partitioned in a way that there are 2 partitions per CPU core, while, for *k-means*, we created 1 partition per core. The memory of each executor was limited to 1.5GB per core. The version of Spark was 1.5.2, which, at the time of the writing of this manuscript, is the latest version compiled and installed on MN3. Each experiment was conducted five times and the median value is computed in order to create the D^m matrices.

The results of profiling of single parameter configurations and their improvement over the default are shown in Figure 3. The red reference line corresponds to the default Spark configuration, as described in `http://spark.apache.org/docs/latest/configuration.html`. The configuration ids are

[4] `https://github.com/intel-hadoop/HiBench`

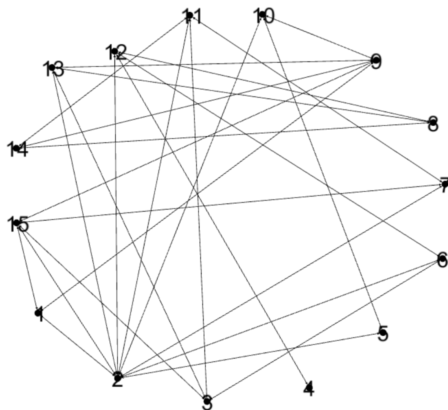

Figure 4: The graph representation of the B' matrix showing the correlations between parameters.

i	j	k
2	15	7
2	11	7
2	10	5
1	15	2
1	9	15
2	12	6

Table 4: The triangles in Figure 4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1_1	1_1	0	0	0	0	1_3	0	1_2	0	1_4	0	0	0	1_1
0	1_1	0	0	1_1	0	0	0	0	1_1	0	0	0	0	0
0	1_1	0	0	0	1_1	0	0	0	0	0	1_1	0	0	0

Table 5: The result bitarray *CCcomplex*

those in Table 2. The best performing single-parameter changes for the three benchmarking applications are employing the *kryo* serializer, setting `spark.shuffle.memoryFraction` to 0.1 (and `spark.storage.memoryFraction` to 0.7), and enabling the compression of RDDs, respectively. In each plot, the rightmost bar corresponds to the best performing pair of configurations. For *sort-by-key*, the best performing pair is the *kryo* serializer combined with the *hash* shuffle manager. For *shuffling*, the best pair is to disable `spark.shuffle.spill.compress` and set `spark.shuffle.file.buffer` to 48Kb. Finally, the *k-means* benchmarking application performs better on MN3 when using the *lz4* compression codec along with disabling `spark.shuffle.io.preferDirectBufs`.

The main observations that can be drawn from Figure 3 are threefold: (i) the default Spark configuration leaves room for performance improvements; (ii) combining parameter modifications yields higher performance; and (iii) the impact of parameters differs significantly between applications.

Figure 4 shows the graph representation of the B' matrix of the 5th step of Phase III, when the threshold t in the 4th step is set to -2. For completeness, the corresponding D^m matrices are presented in the Appendix. Table 4 enumerates the triangles in the graph in ascending order of their scores. Applying Algorithm 2, yields the *CCcomplex* matrix shown in Table 5. The subscript in the table denotes the order in which parameters are added to a configuration. Overall, the set of candidate configurations is of size 9, out of which one configuration contains 6 parameter modifications, two contain 3 parameter modifications and the other six configurations are directly derived from the initial profiling according to the first two steps in Phase III (corresponding to 1 or 2 modifications, repsectively).

5. Validating Experiments

In this section, we present our validating experiments on MN3 (phase IV). The purpose is (i) to provide strong insights into the capability of our proposals to drop running times using three real-world case studies; (ii) to show that our new methodology performs similarly if not better than the iterative one, despite the fact that the latter is tailored to the MN3 infrastructure; and (iii) provide evidence that our new methodology is capable of relying on tests using samples instead of complete runs and adapting to different default settings.

5.1. Case studies and setup

We consider three real-world validating case studies:

1. Building a collaborative filtering model for providing personalized recommendations: the dataset used is the *Audioscrobbler* one[5] and the application runs on 10 MN3 nodes with a single partition per participating CPU core.

2. Finding frequent itemsets: the dataset is also the *Audioscrobbler* one and 10 nodes are used, too. The dataset is partitioned in a way so that there exist two partitions per CPU core. The support threshold is set to 5%.

3. Latent Dirichlet allocation (LDA) topic modeling: the dataset is the *NYTimes* news articles[6]. As in the previous cases, 10 nodes are employed with a single partition per core, and the number of topics is 100.

The new methodology is compared against the previous iterative one from [8] and against employing the *Kryo* serializer, as recommended by the *cheat-sheet* at `http://techsuppdiva.github.io/spark1.6.html`. Also, two values of t in Step 4 of Phase III, namely $t = -2$ and $t = -3$, and a more conservative manner to assign scores in Step 6, namely to take the maximum of the quantity $D^m(i, j)+D^m(i, k)+D^m(j, k)$ instead of the minimum, are evaluated. For example, this more conservative version for $t = -2$ constructs a single 3-parameter complex configuration, which is the same as the 2nd row of Table 5.

All experiments were repeated 5 times and the median times are reported.

5.2. Methodology efficiency

The main results in the three validating case studies are summarized in Figure 5. Figure 7 presents the average speedup

[5]available from `http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html`

[6]available from `https://archive.ics.uci.edu/ml/datasets/Bag+of+Words`

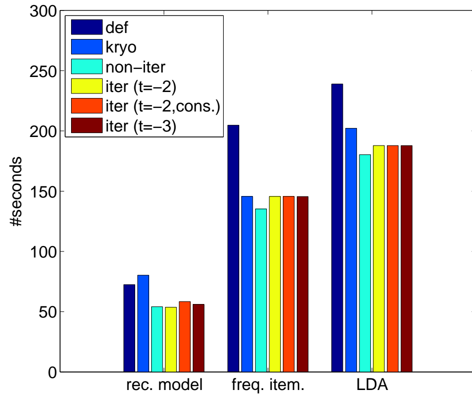

Figure 5: The performance of the different configurations.

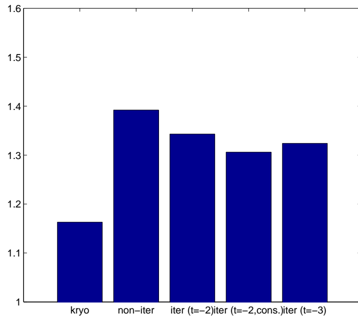

Figure 6: The average speedup achieved by each tested configuration.

achieved by each of the configuration policies. The key remarks are summarized as follows:

1. Both the default Spark configuration and simply employing the *kryo* serializer are outperformed by our configuration methodologies. The decrease in running time due to our non-iterative (resp. iterative) methodology is between 21.4% and 28.89% (resp. 24.53% and 33.93%).

2. The behavior of the new non-iterative methodology, where the average speedup is by a factor of 1.34, is very close to that of the iterative one, where the speedup is by a factor of 1.39, despite the fact that the latter is tailored to MN3.

3. The different flavors of our new proposed methodology exhibit similar behavior, which means that the methodology is robust with regards to changes in its own configurable parameters.

5.3. Sample-based configuration tests

As discussed in Section 4, applying the methodology to MN3 with $t = -2$ yielded 9 candidate configurations. Thus, running 9 times the application in order to find the best performing configuration incurs a non-negligible overhead. However, it is possible to use samples in a smaller cluster and produce dependable results. In general, performance estimates using samples and

	rec. model		freq. item.		LDA	
	full	sample	full	sample	full	sample
conf1	10.82	4.51	**-28.89**	**-2.87**	-15.38	-20.52
conf2	1.63	-1.22	-17.99	12.83	6.69	-3.99
conf3	2.66	-3.80	-1.13	1.00	-2.87	-7.81
conf4	-21.65	-18.11	-25.65	2.45	-14.13	**-24.95**
conf5	7.42	-0.90	3.43	7.15	2.89	0.35
conf6	3.46	1.63	-1.57	7.14	1.23	-1.25
conf7	-19.15	-18.43	-27.07	**-3.07**	-20.46	**-20.84**
conf8	-19.46	**-19.55**	-0.01	5.65	**-21.40**	-8.18
conf9	**-25.95**	**-18.91**	7.34	9.44	-8.55	-12.06

Table 6: Comparison of relative difference percentages from the default of runs on full input vs samples

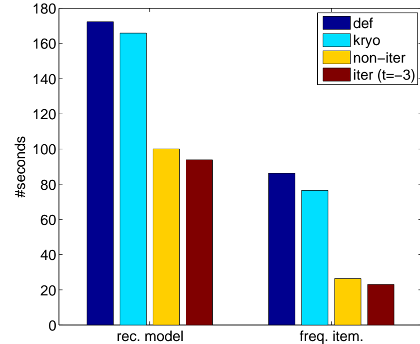

Figure 7: The performance of the different configurations when tested against the default MN3.

fewer nodes for Spark applications is a very challenging topic [10]. In our cases, we observed that when we ran the same validating applications on 20% of their normal input using 20% of the nodes (i.e., just 2 nodes), the actual best performing configuration was either within the first two best performing ones using samples or very close to them. The results are presented in Table 6. As such, our non-iterative methodology can employ samples in order to prune the candidate set to a very small number, e.g., 2 or 3.

5.4. Additional Experiments

Until now, all tests were against the default Spark configuration. However, the default installation of Spark on MN3 employs a modified default configuration, the most prominent features of which is that spill and all kinds of compression are disabled and buffer sizes are larger. We applied our methodology against this default setting, which yielded a different set of candidate configurations. We tested again the methodologies using the case studies with the *Audioscrobbler* dataset, and the results are shown in Figure 7. We can observe (i) that, through comparison with Figure 5, MN3 default yields higher execution time for the first case study and lower for the second; (ii) the decrease in running times due to our methodology is far more significant reaching speedup by a factor of more than 4 times; and (iii) the new non-iterative methodology outperforms the iterative one.

6. Conclusions

This work deals with configuring Spark applications in an efficient manner. We focus on 12 key application instance-specific configurable parameters with 15 different parameter values overall and we assess their impact using real runs on a petaflop supecomputer. Based on the results and the knowledge about the role of these parameters, initially we derive a trial-and-error methodology, which requires a very small number of experimental runs. We then go a step beyond, and we propose a systematic methodology for profiling and deriving candidate configurations that can adapt to any Spark platform. We evaluate the effectiveness of our proposals using three real-world case studies, and the results show that we can achieve significant speed-ups yielding at least 20% lower running times, simply by tuning the configuration parameters in an informed manner.

References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012, pp. 15–28.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), 2010, p. 95.

[3] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, F. Özcan, Clash of the titans: Mapreduce vs. spark for large scale data analytics, PVLDB 8 (2015) 2110–2121.

[4] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, Making sense of performance in data analytics frameworks, in: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2015, pp. 293–307.

[5] A. J. Awan, M. Brorsson, V. Vlassov, E. Ayguade, How data volume affects spark based data analytics on a scale-up server, arXiv:1507.08340 (2015).

[6] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, M. Valero, Spark deployment and performance evaluation on the marenostrum supercomputer, in: IEEE International Conference on Big Data (Big Data), 2015, pp. 299–306.

[7] H. Karau, A. Konwinski, P. Wendell, M. Zaharia, Learning Spark, Lightning-Fast Data Analysis, O'Reilly Media, 2015.

[8] P. Petridis, A. Gounaris, J. Torres, Spark parameter tuning via trial-and-error, in: Advances in Big Data - Proceedings of the 2nd INNS Conference on Big Data, 2016, pp. 226–237.

[9] D. J. DeWitt, J. Gray, Parallel database systems: The future of high performance database systems., Commun. ACM 35 (1992) 85–98.

[10] A. Gounaris, G. Kougka, R. Tous, C. Tripiana, J. Torres, Dynamic configuration of partitioning in spark applications, IEEE Transactions on Parallel and Distributed Systems (to appear) (2017).

[11] Y. Wang, R. Goldstone, W. Yu, T. Wang, Characterization and optimization of memory-resident mapreduce on hpc systems, in: 28th International Parallel and Distributed Processing Symposium (IPDPS), 2014, pp. 799 – 808.

[12] A. Davidson, A. Or, Optimizing shuffle performance in spark, University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep (2013).

[13] I.-H. Chung, J. K. Hollingsworth, et al., Using information from prior runs to improve automated tuning systems, in: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 2004, p. 30.

[14] S. Holl, O. Zimmermann, M. Palmblad, Y. Mohammed, M. Hofmann-Apitius, A new optimization phase for scientific workflow management systems, Future Generation Computer Systems 36 (2014) 352–362.

-3	-5	1	5	-2	-2	-2	-2	-3	-2	-2	-1	-3	-2	-3
0	1	0	6	-2	-2	-3	3	6	-3	-3	-2	-3	-1	-2
0	0	1	0	2	2	2	0	6	5	0	2	1	2	2
0	0	0	6	6	6	6	6	6	6	6	6	5	6	6
0	0	0	0	0	0	2	3	6	1	3	5	6	3	3
0	0	0	0	0	2	1	0	6	0	0	6	0	1	0
0	0	0	0	0	0	4	0	6	0	6	3	5	2	0
0	0	0	0	0	0	0	6	6	3	6	5	0	4	1
0	0	0	0	0	0	0	0	6	6	6	6	6	6	6
0	0	0	0	0	0	0	0	0	6	0	3	0	3	5
0	0	0	0	0	0	0	0	0	0	5	1	1	2	0
0	0	0	0	0	0	0	0	0	0	0	6	0	3	4
0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
0	0	0	0	0	0	0	0	0	0	0	0	0	2	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	6

$D^{sort-by-key}$

2	0	0	-1	-1	0	-1	2	-2	-1	3	2	-1	-1	-2
0	2	-1	-2	-3	2	-1	-1	3	-2	-1	-3	-2	-1	4
0	0	3	0	-1	-3	-1	1	1	2	-2	-1	-3	0	-2
0	0	0	-2	-2	3	-2	0	-1	2	-2	-3	-2	-2	-2
0	0	0	0	2	0	3	2	3	-2	3	1	6	2	3
0	0	0	0	0	2	1	2	0	4	0	-2	0	6	1
0	0	0	0	0	0	-2	0	5	2	-3	-2	-2	-2	-3
0	0	0	0	0	0	0	3	-1	1	-1	-2	-2	-2	-1
0	0	0	0	0	0	0	0	2	-2	0	1	-2	-2	-2
0	0	0	0	0	0	0	0	0	2	0	0	-1	0	3
0	0	0	0	0	0	0	0	0	0	2	0	4	4	-1
0	0	0	0	0	0	0	0	0	0	0	-1	5	2	0
0	0	0	0	0	0	0	0	0	0	0	0	2	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

$D^{shuffling}$

0	1	0	1	1	0	0	0	0	0	0	0	0	1	0
0	0	0	-1	0	0	0	-1	0	0	0	1	0	-1	0
0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	0	0	-1	0	0	0	0	-1	0	0	0	0	0
0	0	0	0	0	0	-1	-1	0	0	0	-1	0	0	0
0	0	0	0	0	0	0	0	1	-1	0	0	2	-1	0
0	0	0	0	0	0	0	-1	0	-1	0	0	1	0	-1
0	0	0	0	0	0	0	0	0	0	-1	0	0	-1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	-1	0	1	-2	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$D^{k-means}$

Table .7: The D^m matrices when applying the proposed methodology to MN3

[15] V. S. Kumar, P. Sadayappan, G. Mehta, K. Vahi, E. Deelman, V. Ratnakar, J. Kim, Y. Gil, M. Hall, T. Kurc, et al., An integrated framework for performance-based optimization of scientific workflows, in: 18th ACM International Symposium on High Performance Distributed Computing (HPDC), 2009, pp. 177–186.

[16] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The hibench benchmark suite: Characterization of the mapreduce-based data analysis, in: IEEE 26th International Conference on Data Engineering Workshops (ICDEW), 2010, pp. 41–51.

Appendix A. MN3 details

MN3 is an IBM System X iDataplex based on Intel Sandy Bridge EP processors at 2.6 GHz (two 8-core Intel Xeon processors E5-2670 per machine), 2 GB/core (32 GB/node) and around 500 GB of local disk (IBM 500 GB 7.2K 6Gbps NL SATA 3.5). Currently the supercomputer consists of 48,896 Intel Sandy Bridge processors mentioned above in 3,056 JS21 nodes, and 84 Xeon Phi 5110P in 42 nodes (not used in this work), with more than 104.6 TB of main memory and 2 PB of GPFS (General Parallel File System) disk storage. More specifically, GPFS provides 1.9 PB for user data storage, 33.5 TB for metadata storage (inodes and internal filesystem data) and total aggregated performance of 15GB/s. With the last upgrade, MN3 has a peak performance of 1.1 Petaflops. At June 2013, MareNostrum was positioned at the 29th place in the TOP500 list of fastest supercomputers in the world, whereas according to the latest TOP500 list in November 2015, MareNostrum is

93rd. A full technical description of MN3 and how it supports Spark applications is in [6].

Appendix B. Benchmarking applications on MN3

Table .7 shown the D^m matrices when applying the proposed methodology to MN3, based on which Figure 4 is produced.