Execution of Recursive Queries in Apache Spark

Katsogridakis Pavlos

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors: Prof. *Angelos Bilas*, *Dr. Polyvios Pratikakis*

This work was partially supported by **Institute of Computer Science, Foundation of Research and Technology Hellas**

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Execution of Recursive Queries in Apache Spark

Thesis submitted by
Katsogridakis Pavlos
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Katsogridakis Pavlos

Committee approvals: _____
Angelos Bilas
Assistant Professor, Thesis Supervisor

_____
Polyvios Pratikakis
Associate Professor, Committee Member

_____
Panagiota Fatourou
Professor, Committee Member

Departmental approval: _____
Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, 01 2017

Abstract

MapReduce environments offer great scalability by restricting the programming model to only map and reduce operators. This abstraction simplifies many difficult problems occurring in generic distributed computations like fault tolerance and synchronization, hiding them from the programmer. There are, however, algorithms that cannot be easily or efficiently expressed in MapReduce, such as recursive functions. In this work we extend the Apache Spark runtime so that it can support recursive queries. Those queries produce a very large number of tasks, making scheduling a difficult and time consuming problem. To tackle this problem we also introduce a new parallel and more lightweight scheduling mechanism, ideal for scheduling a very large set of tiny tasks. We implemented the aforementioned scheduler and found that it simplifies the code for recursive computation and can perform up to 2.5$\times$ faster than the default Spark scheduler for certain kinds of benchmarks.

Περίληψη

Τα περιβάλλοντα MapReduce επιτρέπουν την επεξεργασία τεράστιου όγκου δεδομένων με το να περιορίζουν το προγραμματιστικό μοντέλο σε τελεστές map και reduce. Αυτό το επίπεδο αφαίρεσης απλοποιεί πολλά δύσκολα προβλήματα που προκύπτουν στα κατανεμημένα συστήματα, όπως το συγχρονισμό και την ανοχή σε σφάλματα, και τα κρύβουν απο τον προγραμματιστή. Παρ΄ όλα αυτά, υπάρχουν αλγόριθμοι οι οποίοι δεν μπορούν να εκφραστούν εύκολα σε MapReduce, όπως οι αναδρομικοί αλγόριθμοι.

Στην εργασία αυτή επεκτείναμε το Apache Spark(ένα σύστημα χρόνου εκτέλεσης MapReduce), ώστε να υποστηρίζει αναδρομικούς αλγορίθμους. Οι αναδρομικοί αλγόριθμοι MapReduce δημιουργούν μεγάλο αριθμό εργασιών, οι οποίες δυσκολεύουν το πρόβλημα της χρονοδρομολόγησης. Γι αυτό εισάγουμε ένα νέο παράλληλο και πιό ελαφρύ αλγόριθμο χρονοδρομολόγησης. Ο αλγόριθμος αυτός είναι κατάλληλος για χρονοδρομολόγηση ενός μεγάλου αριθμού απο εργασίες οι οποίες παίρνουν πολύ λίγο χρόνο. Υλοποιήσαμε τον παραπάνω αλγόριθμο και βρήκαμε ότι απλοποιεί την έκφραση αναδρομικών ερωτημάτων, και παράλληλα μπορεί να πετύχει μέχρι 2,5 φορές καλύτερο χρόνο απο τον ήδη υπάρχων αλγόριθμο του Spark σε κάποια είδη εργασιών.

Acknowledgements

This work was supported in part by the European Commission in the context of FP7 ASAP project (619706).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern analytics queries consist of complex computations operated on massive amounts of data. Those queries are impossible to execute on a single node, due to limitations in the cpu frequency and the memory capacity. Thus, the data have to be distributed across a cluster of nodes and processed in parallel. Conventional execution engines are not aware of cluster parallelism, and message passing runtimes like MPI offer precise control and great performance benefits, but the API they provide is very primitive to express complex applications. By restricting the programming model to only map and reduce, or equivalent operators, MapReduce [1] clusters scale out because they do not need to track task dependencies, have simpler communication patterns, and are tolerant to executor and even master node failures.

1.1 Motivation

However, this simplified programming model cannot easily express some applications, including applications with nested parallelism or hierarchical decomposition of the data. When faced with such algorithms, programmers often develop iterative versions that translate recursion into worklist algorithms. This may be inefficient as it introduces unnecessary barriers from one iteration to the next, and can be unintuitive and complicated to code.

An example of such an application with nested parallelism that cannot be easily expressed using flat map-reduce operators is the Barnes-Hut algorithm. The Barnes-Hut simulation [2] is an approximation algorithm for particle simulation. In its simple two-dimensional version, the simulation first recursively splits the space into four quads and computes the center of mass for each, resulting in a tree structure that represents the whole space. In its second phase, it uses the tree of all the centers of mass to compute the forces applied to each body in the space. That reduces the N-Body problem complexity from $O(n^2)$ to $O(nlogn)$, by grouping all objects in distant quads into one force.

Figure 1.1 shows a simplified version of the recurvise query that implements the second phase of the algorithm. Function `calcForces` traverses the tree computed during the first phase, to calculate all the forces applied to a single `particle`. If the particle

```
1  def calcForces(particle, tree) = {
2    if( isFar(particle, tree, THETA) )
3       Array( force(particle, tree) )
4    else
5       tree.map( child => {
6        calcForces(particle, child)
7       }).flatten
8  }
```

Figure 1.1: N-Body recursive query

is far enough from all particles in the tree, then the total force can be computed using the center of mass of the whole space represented by the tree (lines 2–3). If the particle is near the space represented by the tree, then the function recurses to compute all forces applied to the input particle by each sub-tree (lines 5–7). The above computation cannot be executed using the classic MapReduce abstraction, because MapReduce allows only flat map-reduce operations on the dataset. Assuming the `tree` argument is a distributed dataset, the map function would need to recursively apply a map-reduction to directly code the above algorithm.

In this work we extend the Apache Spark MapReduce engine [3] to directly support such nested and recursive computations. Spark is an implementation of the MapReduce model that outperforms Hadoop [4] by packing multiple operations into single tasks, and by utilizing the RAM memory for caching intermediate data. We target Apache Spark because it is a widely used, efficient, state-of-the-art platform for data analytics, and currently the fastest-growing such open-source platform [5, 6].

1.2 Spark programming model

Spark expresses and executes in-memory fault-tolerant computations on large clusters using the RDD abstraction. RDD stands for Resilient Distributed Dataset and RDD instances are immutable partitioned collections that can be either stored in an external storage system, such as a file in HDFS, or derived by applying operators to other RDDs. RDDs support two types of operations: (i) transformations, which create a new dataset from an existing one, and (ii) actions which return a value to the driver program after running a computation on the dataset. Examples of RDD transformations are *map* and *filter* operations, whereas *reduce* and *count* operations are typical actions. All transformations in Spark are lazy, which means that the result is not computed right away. Instead, Spark keeps track of all the transformations applied to the base dataset and they are only materialized when an action requires a result to be returned to the driver program.

Figure 1.2 shows how one can express the word count algorithm in Spark using the RDD abstraction. Variable `f` is the initial RDD representing data to-be-read from a file (line 1). Three consecutive operations are applied to `f`, namely (i) `flatMap` splits all lines into words, producing an intermediate RDD, on which `map` initializes each word to a count of 1, producing a second intermediate RDD, on which `reduceByKey` sums all

```
1 val f  = spark.textFile("hdfs://file")
2 val wc = f.flatMap(line=> line.split(" "))
3             .map(word => (word, 1))
4             .reduceByKey(_+_).collect()
```

Figure 1.2: Word count in Spark

ones for the same word to count how many times that word was found. These computations are coalesced by Spark and occur lazily when `collect` is finally called to read the end result. The `flatMap`, `map` and `reduceByKey` operations are transformations whereas `collect` is an action. Spark schedules all these computations on-demand, using the graph of RDDs created by the program. Figure 1.3 shows the graph of RDDs created by the program in Figure 1.2.

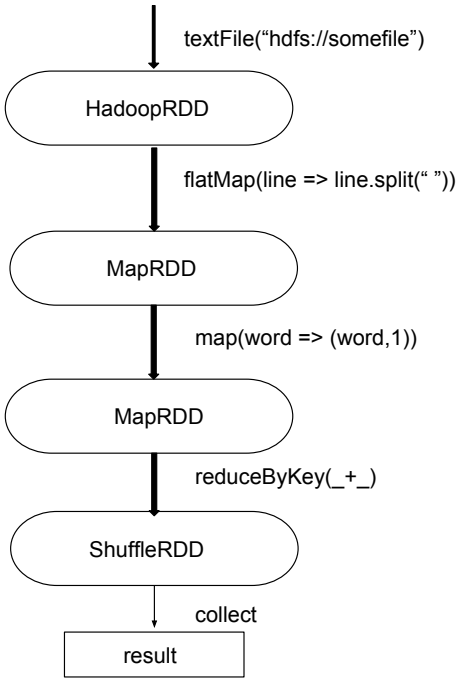


Figure 1.3: The dependency graph for the word count example

Every RDD operator uses a User Defined Function (UDF) that manipulates the data. By default, this UDF is not itself allowed to operate on RDDs in Spark, as RDD objects and their dependency graph are allocated in the master node containing the Spark *scheduler*, where the main program is executed, whereas UDFs are executed by the worker nodes containing the Spark *executors*. This restriction does not affect a large set of programs that do not use recursive computations. Moreover, even recursive computations can almost always be transformed to use a worklist and iteratively fixpoint, to bypass this restriction. That is, however, often ineffective in time and space, for example when not all recursive computations need to go to the same recursive depth, or when the created tasks

are few and not load-balanced. Finally, refactoring a simple recursive computation into a worklist algorithm often introduces complexity and, with it, the possibility of errors. The Barnes-Hut algorithm is an example of such a recursive application that cannot directly be expressed using the "vanilla" RDD abstraction, because it needs nested RDD operators to express the recursive function shown in Figure 1.1.

This work extends the Spark programming model and scheduler to support nested RDD operations, to facilitate expressing recursive and hierarchical computations. We implemented this extension by modifying the RDD scheduling mechanism in Spark and measured its performance. We found that recursive RDD operations can greatly simplify the code for algorithms of a recursive nature, although careless use of job nesting can result in many very small jobs that can greatly increase the overhead cost of scheduling.

The default Spark scheduling mechanism is quite efficient at scheduling coarse-medium grained tasks that may be heavy-tailed, or executed at heterogenous architectures. However, a non-negligible proportion of jobs in almost every analytics application consist of tiny tasks that need to be scheduled as soon as possible. The current Spark scheduler does not optimally schedule such fine-grain tasks as it introduces comparatively large latency from the time one task finishes to the time another task is scheduled to execute on that executor node. To address this issue, we designed and implemented an extension to the Spark scheduler that supports parallel, lightweight scheduling better suited for jobs with fine-grain tasks.

1.3 Contributions

Overall, this work makes the following contributions:

- We added support for nested RDD queries in the Spark scheduler and compare it against built-in operators implemented without nesting. To demonstrate the usability of the programming model extension we implement an N-Body particle simulation using the nested RDD mechanism.
- We added an extra RDD module that enhances the expression of applications with hierarhical parallelism. We implemented a hierarhical K-means application and measured it in a Spark cluster.
- We modified the default Spark task-scheduling mechanism so that it can support many parallel light schedulers. We measured its performance against the default Spark scheduler, and found a speedup of up to 2.5$\times$ for computations using fine-grain tasks.

Chapter 2

Methodology

2.1 Spark Support for Nested Operations

Consider the example code shown in Figure 2.1 that creates two RDDS from two HDFS files (lines 1–2) and performs a map operation on RDD `file1` (lines 3–7). The "mapper" function of the map operation performs a filter operation on RDD `file2` for every word in RDD `file1` to select all words of larger length. The calls to `collect()` are there to force the computation to take place and collect the results into an array, as otherwise RDDs would behave essentially like lazy futures. This is a simple example of one-level-nested operators.

By default, Spark does not support such nested RDD operations. This is mainly because the RDD metadata required to schedule new computations are stored only at the master node, with executor nodes simply running tasks assigned to them. In short, each RDD object created by the program tracks at least the following data:

- a set of partitions and their locations (e.g., in the memory store of executor nodes or in HDFS blocks)
- a dependency list of parent RDDs of which this RDD is a product
- a user-defined function that computes each partition in the dataset from its parents
- (optionally) a partitioner that defines how the elements in a key-value pair RDD are partitioned by key

```
1 val file1 = sc.textFile("hdfs://file1")
2 val file2 = sc.textFile("hdfs://file2")
3 file1.map(word1 =>
4   file2.filter(word2 =>
5     (word1.length > word2.length))
6       .collect())
7   .collect()
```

Figure 2.1: Example of nested RDD operations

- (optionally) a list of preferred locations to compute each partition on (e.g., block locations for an HDFS file)
- a unique ID
- a reference to the Spark context object that handles the Spark cluster

All of the scheduling and execution in Spark is done based on these data, allowing each RDD to implement its own way of computing itself. For most RDDs, a computation "task" is the application of one or more user-defined functions on a partition of data. To run a Spark job, the master node in the cluster schedules such tasks on the executor nodes.

Handling nested RDD operators inside the user-defined functions of a map operation as shown in Figure 2.1 (lines 4–6) requires the executor nodes that run the tasks of the outer RDD map operator (lines 3–7) to behave as the master node and schedule the "nested" filter job created in the mapper function (line 4). Adding such functionality to the executor nodes would greatly increase their complexity, as the RDD data would need to be replicated on executors, which would require maintaining RDD metadata consistent among all distributed copies of an RDD. Apart from being inefficient, this would undo the simplicity and efficiency of the MapReduce model. Thus, our design forwards the nested operators back to the master, to avoid a distributed scheduler setup.

In the example in Figure 2.1, the outer collect method will force the runtime to schedule the outer RDD computation. Since no shuffle operations are involved, the dependency DAG will consist of only one stage that contains one transformation of the HadoopRDD `file1` to a MappedRDD returned by the `map` method. The Spark scheduler will try to submit this stage and since there are not waiting parent stages it will proceed with creating and submitting the missing tasks. Then the scheduler will create tasks that execute the mapper function (lines 4–6) for each word in the `file1` RDD. Specifically, the scheduler will serialize a closure of the mapper function and distribute it to the executors; then it will create a task for every partition of the `file1` RDD and send each task to an idle executor to run the mapper function on that partition and thus create the corresponding partition of the result RDD.

In the executor nodes, when the mapper function shown in Figure 2.1 (lines 4–6) runs, it will try to invoke a filter operation on the `file2` RDD. We extended the executor functionality to capture this event and send a *CreateRDD* message to the scheduler node. The message contains an identifier of the RDD object referenced, the (reflective) name of the invoked operation, and a serialized version of the user-defined function that is applied.

We also extended the Spark scheduler to receive such messages from the executors. Upon receiving such a forwarded RDD operation message, the scheduler looks up the RDD with the specified id and, using jvm reflection, invokes the specified operation. In the execution of the example code in Figure 2.1 it will invoke the `map` method of the `file2` RDD, creating the desired RDD that describes the result. Note that, as RDDs are essentially futures in Spark, no computation will yet take place at this point. The scheduler then will simply send back to the executor an identifier of the created RDD. The executor, upon receiving that message, will create a proxy of that RDD object based on the identifier received, and use it to continue the computation of the mapper function

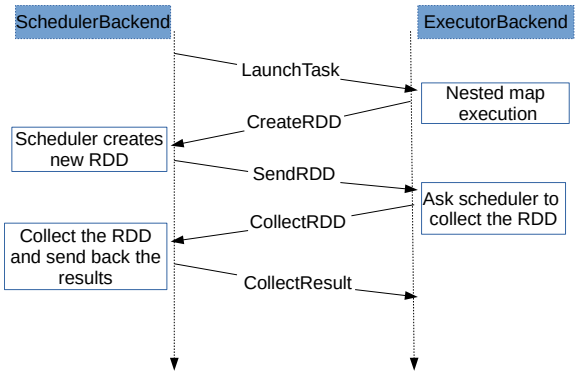


Figure 2.2: Executor asks the master to schedule a nested RDD operation

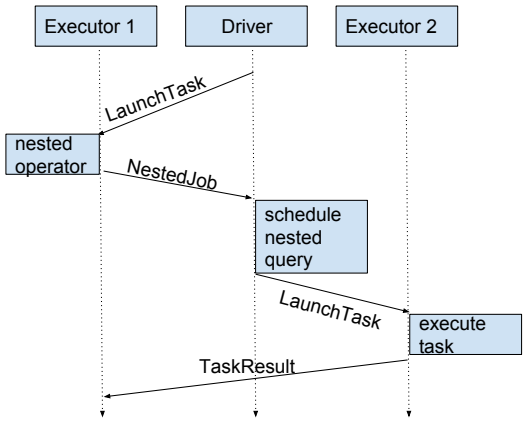


Figure 2.3: Forwarding the nested task result to the executor that issued the query

for the task of the outer map operation on `file1`. When that mapper function calls `collect()` (line 6), the executor will send a *CollectRDD* message to ask the scheduler to collect the new RDD, using its identifier, and send back the result. Figure 2.2 shows the sequence of messages that will be sent for this example.

Forward nested task results In the naïve master-executor protocol described above, the master will schedule the nested job after receiving the CollectRDD message. When the nested job is done it will receive the result from all the executors where the job was scheduled, combine all partitions into a collected array, and send the result to the executor that issued the nested operation. This would mean that there is an unnecessary transfer of data to the master node, and from there to the executor that issued the nested job. To avoid the transfer overhead that would also make the master node a centralized bottleneck for all nested computations, we modified the executor code to send the nested task result directly to the executor that issued the nested job and also send an ack to the master node that the

```
1 val array = (0 until N)
2 // make an rdd with P partitions
3 val rdd = sc.parallelize(array, P)
4 val sum = rdd.reduce(_ + _)
```

Figure 2.4: Example Spark benchmark with many tiny tasks

nested task finished, in order to free the executor resources. This way, the only transfer of data to the master is for `collect()` operations called at the master node in the top-level of the program. Figure 2.3 shows the messages exchanged between the master and two executors executing the nested query.

Coalesce multiple nested operators Note that, following the protocol described so far, the executor sends a CreateRDD message to the scheduler for every RDD operation that the mapper function of the `file1` map operation performs on `file2`. However, only the `collect()` operation requires the scheduler to actually schedule a nested computation. This is because the Spark RDD abstraction is lazy and the operations are not computed immediately. Instead, they form a dependency graph in the scheduler. Every operator applied to an RDD simply triggers an addition of a node and one or more edges to the graph. The graph nodes are grouped into stages, that are later packed into tasks sent to the workers. This allows Spark to coalesce and pipeline operations, fit intermediate data in memory, and avoid disk IO.

In standard Spark, there is no need to delay this process as the program executes all RDD operations on the master node that contains the Spark scheduler, hence there is no overhead in performing the dependence graph operations whenever an RDD operator is called. The nested-Spark extensions described above, however, introduce the latency of an unnecessary round-trip message exchange between the executor and the scheduler. In the execution of the example of Figure 2.1, the scheduler will receive a message for the nested `filter` operation, apply the message adding a new node to the RDD graph, and send back the new RDD. This means that if a nested RDD query contains lots of operators, then many CreateRDD-SendRDD messages will be exchanged between each executor and the master node, without these resulting in actual scheduling of a computation. These messages simply increase the latency and network communication. We solved this problem by grouping all the nested operators into a single message. The fact that RDDs are both lazy and immutable permits us to pack all the RDD operator arguments in a per-executor global data structure. Then, at the end of a task or when a nested `collect()` is triggered, the executor will send all the RDD transformations to the scheduler to be handled as described above, that is, to create all the RDDs described and schedule any required computations.

2.2 Scheduling

Consider the code shown in Figure 2.4, a Spark application that creates an RDD of N integers and P partitions and computes the sum of the RDD elements. When P is large enough, the job will comprise a huge set on tiny tasks. Although computations like this example rarely constitute the whole of a Spark program, they are often found within larger computations as, for instance, a stage in an analytics pipeline, or "inner" jobs in a Spark-nested program as described in the previous section. The default Spark scheduling algorithm underperforms for jobs like that, because:

1. The scheduling path is sequential, which means that if a job consists of many tiny tasks, scheduling itself will take a lot of time in the critical path of the computation, while the processing time will be negligible.

2. After a worker has finished a task, it has to send a request message to the scheduler, so that the scheduler sends a new task to the worker. That increases the total time by at least one RTT for every task and every worker, since the scheduler receives and handles these messages sequentially.

To solve these issues, which may be further exacerbated in cases where a large number of executors cannot be properly managed by a single, centralized Spark scheduler, we designed and implemented a parallel and distributed version of the Spark scheduler.

We aim to decrease the time between when a worker finishes a task and sends a message to the scheduler and when the scheduler answers with the next task to run. To accomplish this, we modified the Spark scheduler to send multiple tasks to each executor and amortize the idle time between tasks over many requests. Specifically, we inserted a local task queue per executor, and modified the centralized scheduler to keep track of these coalesced task sets. Every time a worker core finishes a task, it first tries scheduling one of the tasks in the local task queue, and only generates network traffic and a request to the centralized scheduler if the local queue is empty.

In addition to task-set coalescing, we parallelized the central Spark scheduler to schedule task-sets in parallel. Specifically, instead of using a single scheduler-master, we deploy a set of schedulers organized hierarchically as a set of *SecondLevelSceduler* actors under the standard Spark master node. The standard Spark scheduler then creates a few large task-sets per job and sends them to the second level schedulers; each second level scheduler is then responsible for sending smaller task-sets or individual tasks to the executors. This reduces the congestion at the Spark scheduler occuring either because tasks are too small or because there is a large number of pending executor messages. We do not assign specific executor groups to the second level schedulers, and instead allow all second level schedulers to send work to all available executors. This works well in practice when the available work is much more than the executors, which is almost always the case in Spark analytics applications. A simplified version of the scheduling algorithm is shown in Figure 2.5

To schedule and track tasks to executors, each second level scheduler keeps a copy of all the executor metadata that the default Spark master normally maintains. This creates a

```
1  def Executor:executeTask(task) {
2   if(availableCores>0) {
3     --availableCores
4     threadPool.execute(task)
5   } else {
6     taskQueue.enqueue(task)
7   }
8  }
9
10 def Scheduler:ProxyLaunchTasks(taskset) {
11  taskset.foreach( task =>
12    rid = selectExecutor(task)
13    executors(rid).send(LaunchTask(task))
14  )
15 }
16
17 def Scheduler:LaunchTasks(taskset) {
18  val splits = taskSet.split(nschedulers)
19  par foreach (proxy,subtasks) in splits =>
20    proxy.send(ProxyLaunchTasks(subtasks))
21 }
```

Figure 2.5: Scheduler pseudocode

consistency issue, as not all of these copies may be updated at the same time. We solve this by maintaining all the "heartbeat" messages Spark uses for tracking executor availability at the Spark master, and we only forward information about executors from the master to the second level schedulers. This means that at any given time the latest metadata about the state of one given executor's availability are at the master, and the metadata about all tasks in that executor's queue are distributed among all second level schedulers that may have sent tasks to that executor. To handle the case of executor state changes, the Spark master sends a message to all second level schedulers when the heartbeat process discovers that an executor has changed state. For example, when an executor is started, it sends a message to the master to inform that the executor is registered, as in the standard Spark scheduler. Then, the master node broadcasts to all second level schedulers the state of the newly registered worker. Eventually, all the schedulers will have the same view of the cluster state.

A similar problem of distributing copies of metadata occurs in tracking task completions. Specifically, the standard Spark scheduler uses *StatusUpdate* messages that contain information about whether a task has started, is executing, has finished, or has failed. In our distributed scheduler, these messages are sent from the workers to the proxy schedulers. Currently, the proxy schedulers eventually forward all *StatusUpdate* messages to the central Spark master. We have not yet managed to recreate any cases where this creates a bottleneck; in that case we expect it would be straightforward to reduce the strain on the Spark scheduler by handling task completions and failures in the second level schedulers without any forwarding of that information.

The standard Spark scheduler balances loads among executors by sending tasks only to the executors that have free cores. In avoiding the update messages by coalescing sets of tasks per executor and in allowing all second level schedulers to send tasks to all executors,

```
1  //sc is the Spark Context
2  val rdd = sc.parallelize(ran_array,npar)
3              .cache()
4  rdd.count() //some warmup
5  val result = rdd.filter( _%33 == 0)
6                  .collect()
```

Figure 2.6: Scala benchmark that checks if a number is multiple of 33

```
1  val rdd = sc.parallelize(array,npar)
2              .cache()
3  rdd.count() //some warmup
4  val result = rdd.collect()
```

Figure 2.7: Scala code that gathers all the dataset to the master node

we have removed the load balancing guarantees of the standard Spark scheduler. However, we found that by tranferring some of the scheduler functionality to the executors suffices in practice to give load-balanced executions. Specifically, we use a best-effort approach for balancing task loads, where each executor locally schedules tasks from a queue to cores as they become available. The per executor local queue we inserted is visible by all executor threads. This means that in a case where an executor is loaded with some heavy and some light tasks, the threads executing the light tasks that will finish earlier, will dequeue and execute more tasks. Thus, when a job consists of some heavy tasks, even if they are scheduled on the same executor it is highly improbable that they will be executed by the same core.

Note however that this solution is best effort. In most cases given enough executor cpus the load will be equally balanced. In an exetremely bad scenario where too many straggler tasks are scheduled into the same executor while the other executor takes all the lightweight tasks, the runtime will be highly affected. We tried to stress our best-effort solution by constructing benchmarks with highly-imbalanced tasks (Section 3, but were unable to create such a scenario in practice.

2.2.1 Microbenchmarks description

Figures 2.6,2.7,2.8, 2.9,2.10 show the main code for the benchmarks we chose to compare the vanilla Spark scheduler with our alternative scheduling. We chose those benchmarks because they consist of non computationally demanding tasks, so that the scheduling pro-

```
1  val rdd = sc.parallelize(array,npar)
2              .cache()
3  rdd.count() //some warmup
4  val result = rdd.reduce(_+_)
```

Figure 2.8: Scala benchmark that adds all the elements of a dataset

```
1  val rdd = sc.parallelize(array,npar)
2               .cache()
3  rdd.count() //some warmup
4  //foreach task wait 50ms with 10% prob,
5  //and 100ns with 90% probability
6  val result = rdd.mapPartitions( p =>
7       {delay(samplep()); 1}
8  ).collect()
```

Figure 2.9: Longtail Benchmark: each tasks sleeps some time according to long tail distribution

```
1  val rdd = sc.parallelize(array,npar)
2              .cache()
3  rdd.count() //some warmup
4  val result = rdd.flatMap(_.split(" "))
5                  .map( e =>(e,1) )
6                  .reduceByKey(_+_)
7                  .collect
```

Figure 2.10: Word count benchmark

cess becomes the bottleneck. However many of those computations are used in analytics applications.

2.3 Queries with hierarchical data decomposition

The existing RDD representation and operators work well with a wide set of problems and algorithms, operating on "flat" data collections, such as map-reduce programs. There are, however, algorithms and computations that require structuring the data set in different ways. For example, the data decomposition in a divide-and-conquer algorithm or the hierarchical back-tracking of a dynamic programming algorithm require the programmer to create complex structures of RDDs that capture the "non-flat" structure of the data.

2.3.1 Introduction

K-means [7] is the most common algorithm used in cluster analysis, which aims to partition the elememts into k clusters, such that each elemenent belongs to the nearest cluster. K-means is an approximation algorithm, that iterates through the data till the error is minimized. In each iteration, first all elements are assinged to the closest centroid using Euclidean or some other distance metric, and then for each cluster the new centroid is calculated.

Hierarchical K-means is an interesting variation of K-means clustering, that builds a dendrogram on the clustered data, often used in bioinformatics, document clustering and machine learning. The most common algorithms for hierarchical clustering are bisecting K-means, and agglomerative clustering [8]. We focus on the bisecting variation, that splits

the elements in a top-down way. The basic steps are:

1. Select a cluster to split
2. Split the selected cluster into 2 sub-clusters using default K-means
3. Repeat step 2 for some iterations and select the best (minimizing error) clusters
4. Repeat the above steps until the requested number of clusters or granularity has been reached

In short, the algorithm uses the output of classification to split data and recursively classify and split these sets, down to a threshold size.

Describing hierarchical clustering in massive datasets is challenging, as one necessarily describes the computation as iterative analytics queries. We propose a high level abstraction to express hierarchical structures. Specifically, we present a higher level way of expressing hierarchical algorithms (while still using the MapReduce abstraction), that can assist the execution engine to more efficiently schedule such computations.

2.3.2 Design

For example, consider the divide-and-conquer algorithm for computing hierarchical K-Means clustering presented above. Note that the data is initially "flat", but the algorithm discovers and maintains structure during the computation.

Expressing such a hierarchical algorithm with the existing RDD operators can be quite challenging for the users. The splitting of the data in many levels results in a tree of RDDs, that are quite difficult to handle and maintain. Also nodes in the same level of the tree represent disjoint tasks, that can be issued in parallel.

2.3.3 Hierarchical RDDs

We present a new RDD abstraction that helps the programmer create a tree collection of RDDs and issue independent jobs in parallel. To make the Spark RDDs more expressive for hierarchical structures, we created an RDD extension, called hierRDD, and use hierRDD to encode bisecting k-means in a much more forward and intuitive way, while also improving execution performance.

In order to create hierarchical RDDs the user should first provide an object that implements the Splittable interface described in Figure 2.11. The Splittable object repserents a hierarchical structure that can be splitted into smaller sub regions. Thus the user should implement the function *contains* that specifies whether an element is contained into the Splittable object, and the *Split* function that returns an array of the subregions the object is splitted. The SplitPar method is identical to the Split method, except that it returns a Parrallel Array, so that the Spark scheduler can issue the jobs concurrently.

An example of implementing the Splittable trait is the Cluster class shown in figure 2.12, that is later used to code the bisecting K-means algorithm. The constructor takes as arguments the identity of the cluster, and the number of iterations (k-means specific).

```
1 trait Splittable[A] {
2   def id: Int
3   def contains(a:A): Boolean
4   def splitPar(l:Int): ParArray[_<:Splittable[A]]
5   def split(l:Int): Array[_<:Splittable[A]]
6 }
```

Figure 2.11: API for creating hierarchical RDDs

```
1  class Cluster(id: Int, iter:Int)
2    extends Splittable[Vector]{
3
4    def id() = id
5    var data:RDD[Vector]
6
7    var m = KMeans.train(data, k=2)
8
9    def contains(point: Vector) = {
10     m.predict(point)==id
11   }
12
13   def split(level:Int) = {
14     m.clusterCenters.map{
15       case(c,idx) =>
16         new Cluster(idx, iter)
17     }.toArray
18   }
19
20   def splitPar(level:Int) = {
21     split(level).par
22   }
23 }
```

Figure 2.12: Splittable subclass for the Bisecting K-means benchmark

To split the initial data we use the KMeansModel from the Spark MLlib library. The m variable represents the clustering model, used to define to which subcluster each point belongs. The split method iterates through the cluster center and for each one it creates a new Cluster instance with the id of the cluster.

Figure 2.13 describes the main loop in the bisecting K-means application. Line 1 creates the initial cluster that contains all the data elements(that implements the Splittable interface), and then in line 2 we create a hierarchical RDD from the data. The while loop in lines 5–8 continiously splits the cluster into smaller subclusters until we reach the desired number. The splitPar operator returns a ParArray(scala.collection) so the splitting is issued in parallel, resulting in reduced total time compared to the sequential one.

```
1 val initcluster = Cluster(id=0, data)
2 val hierrdd = data.hier(initcluster)
3
4 var split = hierrdd
5 for(i <- 1 until maxdepth){
6   split = split.flatMap(
7     subrdd => subrdd.splitPar()
8   )
9 }
```

Figure 2.13: Bisecting K-means implementation with hierarchical RDDs

Chapter 3

Evaluation

3.1 Parallel Scheduler Evaluation

We evaluated the performance of our scheduler using a set of micro-benchmarks. The datasets contain integers or words, split into a defined number of partitions, and intentionally cached so that the tasks do not take extra time loading the data. We first invoke a count operation in all benchmarks, without counting it in the total run time, so that we ensure that the dataset is stored in memory. We ran each benchmark is 15 times and measure the last 10 runs, so that the runtime is not affected by the JVM class loading, JIT compiling or other optimization techniques [9].

We used the following benchmarks:

- The filter benchmark generates a dataset of random numbers and returns those that are products of a defined number
- The sum benchmark adds the values of all the elements using the reduce operator
- The collect benchmark simply brings all the elements to the master
- The longtail benchmark simulates a taskset whose runtime follows a long tail distribution
- The word count benchmark counts the references of each word

We chose those benchmarks because they consist of non computationally demanding tasks, so that the scheduling overhead becomes a bottleneck. However, many of those computations are used in analytics applications. In fact, we have encountered very small tasks in map or filter operations that operate on fine-grain partitions in actual analytics applications; in most cases the programmer did not try to create larger tasks, as the overhead of repartitioning is comparable to the scheduling overhead of fine-grain tasks.

We implemented our scheduler in Apache Spark 1.6.0. We ran all experiments on a cluster of 5 nodes, where each node has 4 Intel i5-3470 cores, 16GB memory, and is running Debian Linux and OpenJDK7. We compare our scheduling algorithm with the default Spark scheduling. To have a valid comparison, we tried to use equal resources

for scheduling and for task execution; the runs with default Spark use one node as a Spark master and 4 nodes as executors, while the runs with our distributed scheduler deploy all second level schedulers together with the Spark master on one node, and use 4 nodes as executors. This way, both schedulers have exactly the same resources devoted to scheduling and to task execution.

Variable number of tasks We ran the aforementioned benchmarks with a fixed number of elements (5M), and a variable number of partitions (64 to 8192) to measure how the number and granularity of tasks affects the runtime difference between the two schedulers.

Table 3.1(a) presents average running time of a simple filter operation on 5M elements. The first column contains the number of partitions of the input RDD (number of tasks); the second column contains the average running time of the query using our distributed scheduler, in milliseconds; the third column contains the average running time using the default Spark scheduler; and the final column presents the speedup of our scheduler over the Spark default scheduler. Our scheduler consistently outperforms the default Spark scheduler by at least 1.11× and up to 2.22×. Much of that difference seems to be a constant factor, which we believe is due to the reduction of worker idle time while waiting for the next task. This is better observed in Figure 3.1(b) plotting the difference between the two schedulers. Note that as task granularity becomes smaller, both schedulers perform worse.

Similarly, Figure 3.2 compares the default Spark scheduler to our distributed scheduler on a reduction that sums 5M random integers. The horizontal axis is the number of partitions that the dataset is distributed into. Again, we note that reducing the number of messages and parallelization of scheduling gains a constant factor over the default Spark scheduler, resulting from 1.11× to 1.86× better performance.

Figure 3.3 compares the two schedulers on simply collecting all the elements of a partitioned RDD to the master node. We observe the same behavior even when the task execution time is zero in this case, again due to the reduction in scheduling overhead and message latencies.

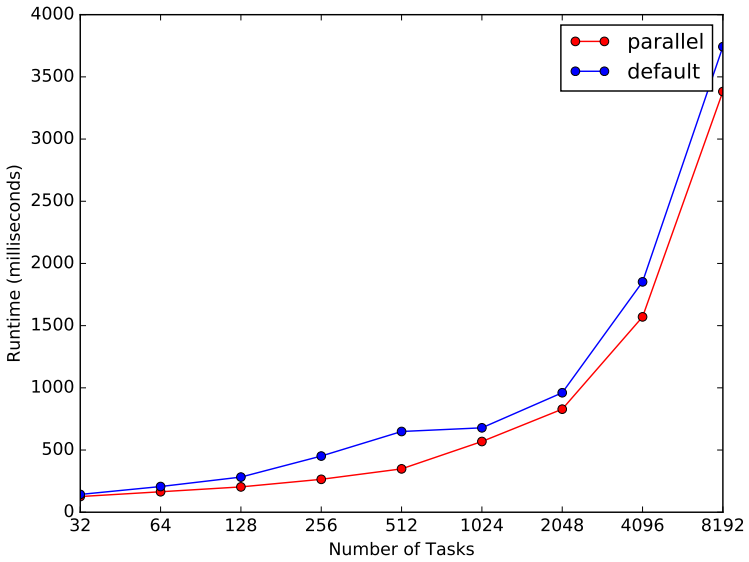
To evaluate how well our best effort load balancing heuristic performs compared to the load balancing guarantees provided by the default Spark scheduler, we ran a microbenchmark that simulates tasks with highly different running times, following a long-tailed distribution. Figure 3.4 presents a comparison of the two schedulers on the long-tail benchmark. Again, the distributed scheduler achieves a speedup between 1.13× and 1.77× over the default Spark scheduler. This result is consistent accross executions with negligible variance, hence we conjecture that for executors with more than 4 cores it is highly unlikely that straggler tasks will cause imbalance and large latency in the total job execution time.

Finally, Figure 3.5 presents the comparison on a standard word count benchmark. Again, the distributed scheduler outperforms the default Spark scheduler by up to 2.22×.

Variable number of elements To measure the effect of the task granularity on performance, we ran the reduce benchmark with a fixed number of partitions (512). Table 3.1

tasks	distributed	default	speedup
64	164.50	206.50	1.26
128	203.50	282.50	1.39
256	264.50	451.50	1.71
512	348.50	649.00	1.86
1024	568.50	678.50	1.19
2048	828.50	960.50	1.16
4096	1570.50	1852.00	1.18
8192	3382.00	3742.50	1.11

(a) Running time in milliseconds



(b) Difference in execution time

Figure 3.1: Filter microbenchmark on 5M elements

elements	distributed	default	speedup
250K	331.30	798.80	2.41
1M	328.70	826.70	2.52
4M	333.10	854.00	2.56
16M	333.90	951.50	2.85
64M	382.60	1117.70	2.92

Table 3.1: Comparing runtime(ms) of default and parallel scheduling in reduce benchmark, 512 partitions

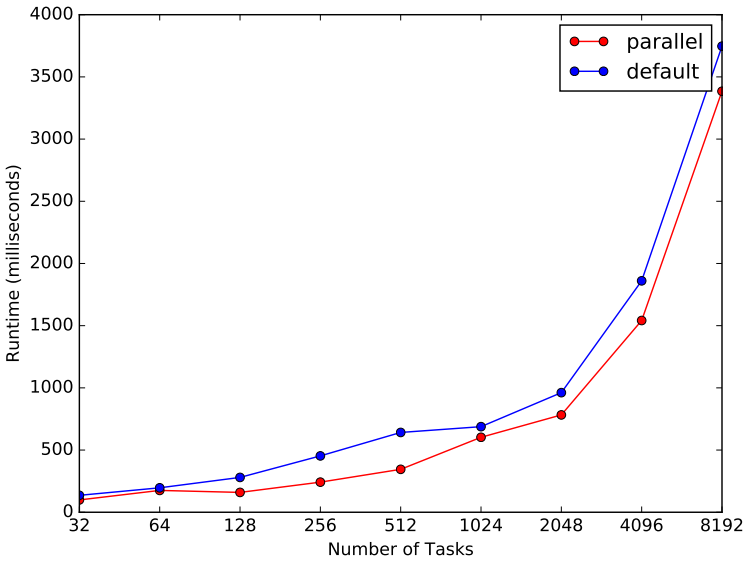


Figure 3.2: Reduce Benchmark results for 5M elements

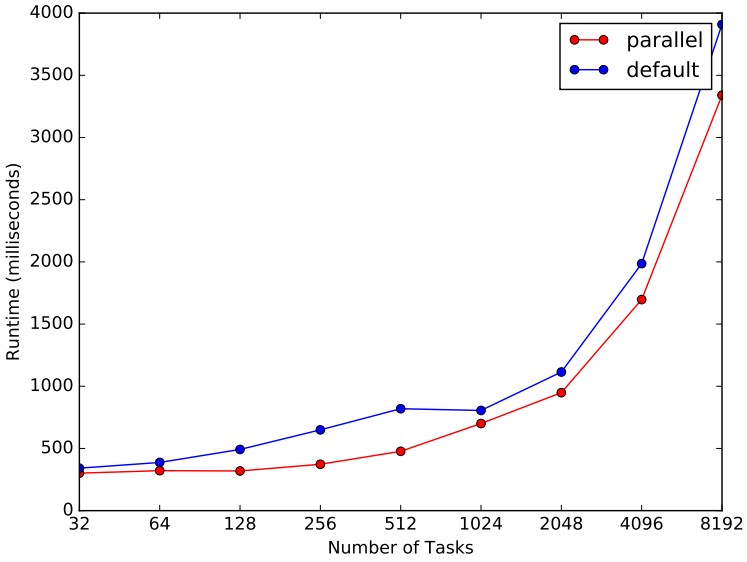


Figure 3.3: Collect all data at the master

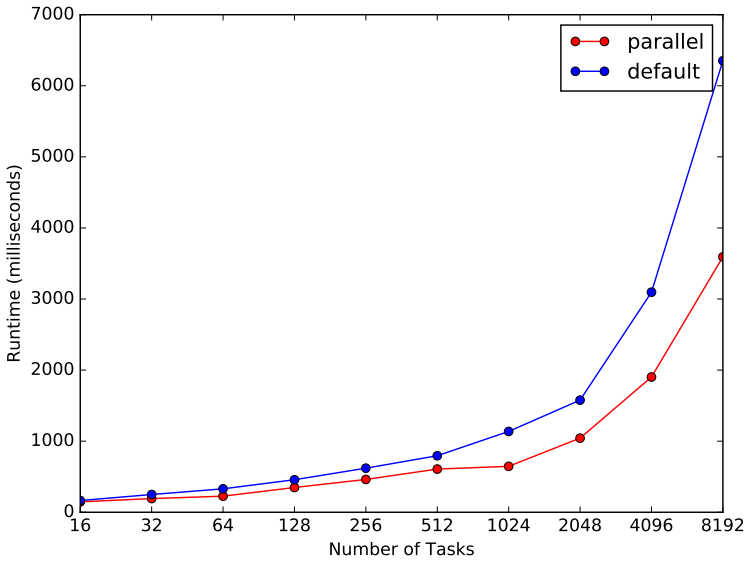


Figure 3.4: Long-tail distribution of task runtimes

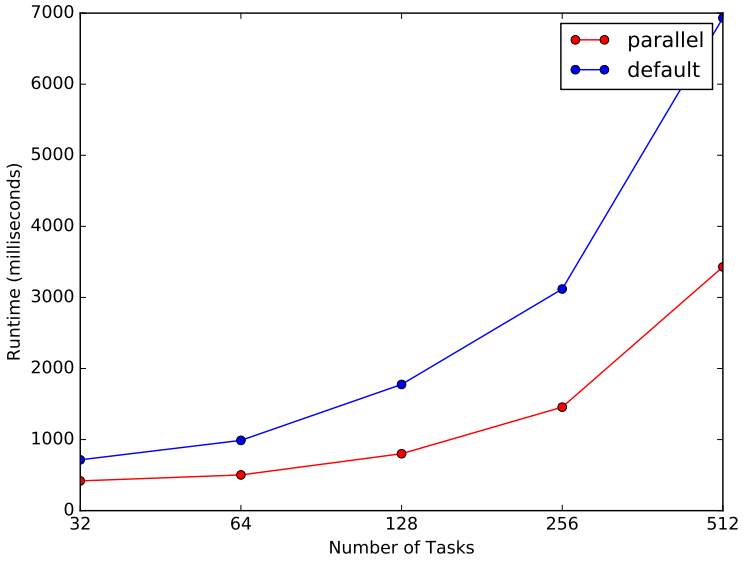


Figure 3.5: Word count benchmark results

```
1 val file1 = sc.textFile("hdfs://file1")
2 val file2 = sc.textFile("hdfs://file2")
3 val cartesian = file1.map( e1 =>
4   file2.map( e2 =>
5     (e1,e2)
6   ).collect().flatten).collect()
```

Figure 3.6: Cartesian product written in nested RDD query

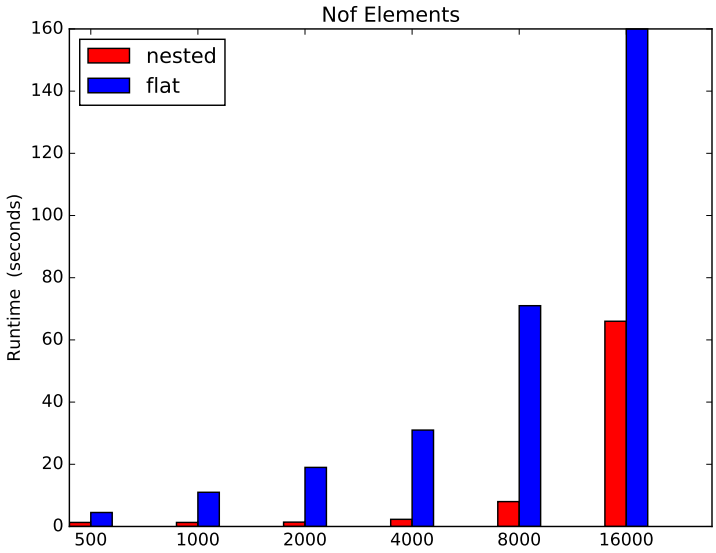


Figure 3.7: Comparison between flat and nested operators in cartesian product

shows the speedup of the distributed scheduling versus the vanilla Spark scheduler. Note that the distributed scheduler is almost insensitive to the number of data elements, whereas the default Spark scheduler slows down for more data. We believe this is due to the fact that the default Spark scheduler puts computation, scheduling overhead, and communication in the critical path, whereas by parallelizing the handling of scheduling messages the distributed scheduler overlaps them.

3.2 Nesting evaluation

To evaluate the nesting RDDs performance, we used our parallel scheduler as a scheduling mechanism. We used the cartesian product as a benchmark to compare the performance of nested queries versus flat queries. A simplified version of the code for cartesian product using nested operators is shown at figure 3.6. For the flat, non-nested version we used the cartesian RDD operator.

Figure 3.7 compares the total running times between the two schedulers. We found

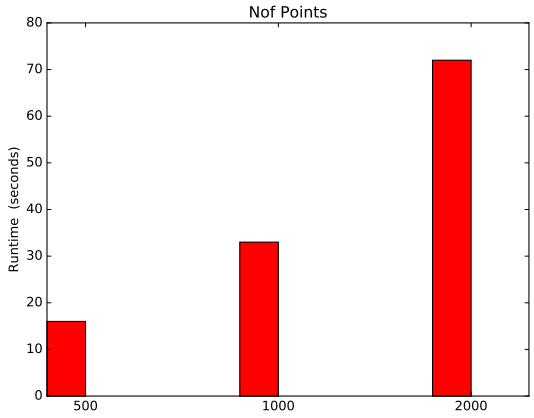


Figure 3.8: Barnes Hut measurements

that writing a cartesian product as a two-level nested RDD operation parallelizes it into smaller but parallel jobs and achieves a total speedup of up to 8×, mainly due to the parallel scheduling of the work.

To demonstrate the programming expressiveness of using nested RDD operations we implemented the Barnes-Hut n-body gravity simulation algorithm using nested operators, and evaluated it for various numbers of data points, up to 2000 bodies. Note that there is no comparison against the default Spark scheduler as Barnes-Hut is recursive and thus not directly portable to flat MapReduce, without completely restructuring the algorithm to use explicit iterations and simulate a stack. Figure 3.8 presents the results.

3.3 Hierarchical RDD Evaluation

We evaluated our design comparing the hierRDD, hierRDD with parallel splitting, and the default implementation using simple RDDs and MLlib. We run the experiments with 1, 2, 3, and 4 slaves to measure scalability.

The fitst data-set contains 1 million points, of 20 dimensions each. Table 3.3 shows the time in seconds for each K-means variation, for different number of slaves. The last column measures the speed up gained from hierRDDpar compared to hierRDD. For the maximum number of workers the speedup is 40%. The second data-set has 2 million data points. Table 3.2 shows the time scale and the speed up. For 1 workers hierRDDpar gains speedup 10% compared to hierRDD and 37% for 4 workers.

Parallel hierRDD gains speedup over sequential hierRDD because the cluster utilization is higher and the load imblance between different tasks is mitigated.

To expose the expressiveness of our hierRDD implementation we compared our hierarchical K-means variation with one using the default RDDs found in `https://gist.github.com/freeman-lab/5947e7c53b368fe90371`. Figure 3.12 shows the time elapsed for both applications for various data points. The figure shows that for 750

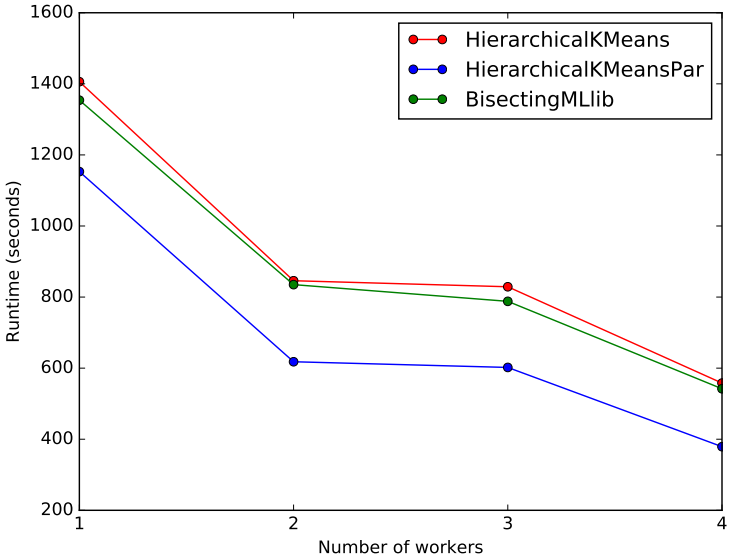


Figure 3.9: HierRDD: Strong scaling graph for 1 million data points

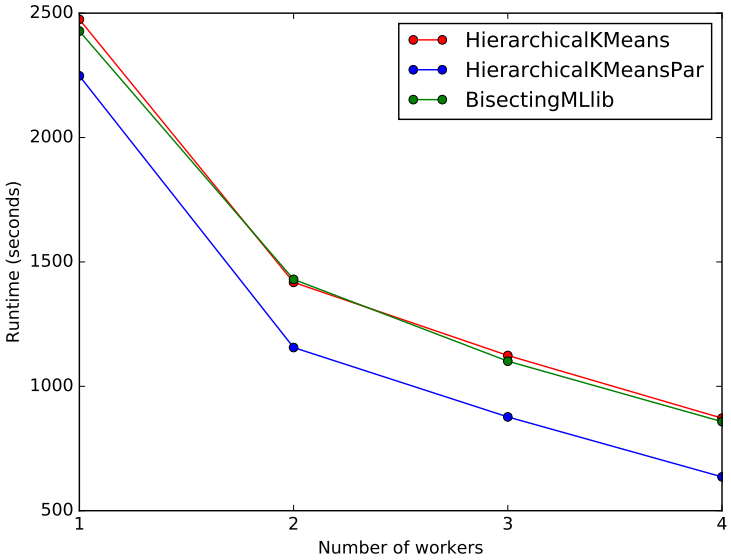


Figure 3.10: HierRDD: Strong scaling graph for 2 million data points

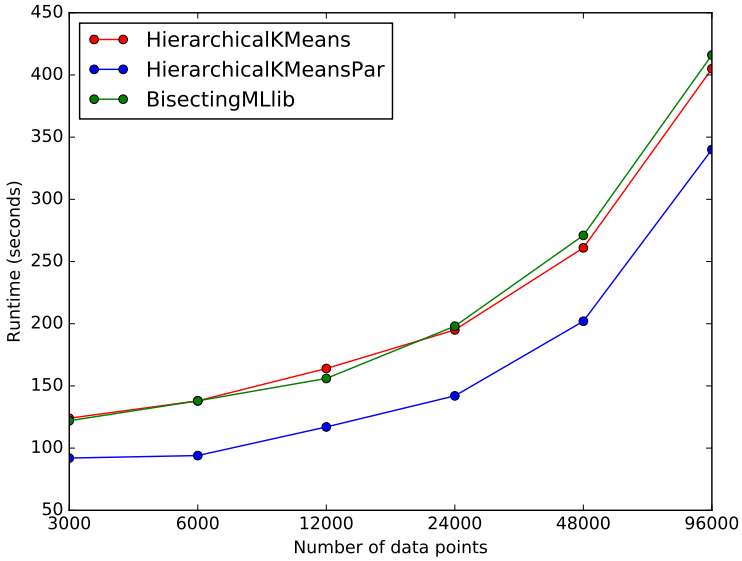


Figure 3.11: HierRDD: Weak scaling graph

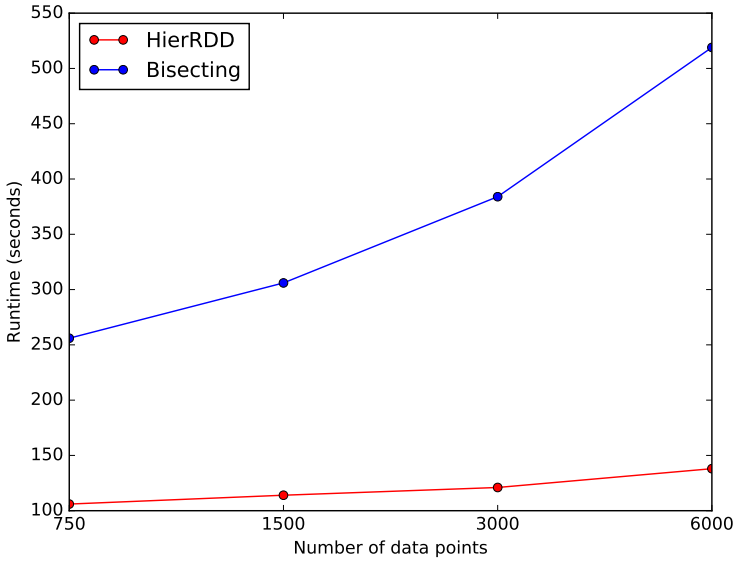


Figure 3.12: Naïve version versus hierarchical RDDs

workers	hierRDD	hierRDDpar	mllib	speedup
1	2475	2248	2542	1.10
2	1418	1156	1394	1.22
3	1124	877	1101	1.28
4	872	636	858	1.37

Table 3.2: Comparing the runtime(secs) between flat RDD and hierRDD for 2 million data points

workers	hierRDD	hierRDDpar	mllib	speedup
1	1406	1153	1354	1.21
2	846	618	835	1.36
3	829	602	788	1.37
4	558	379	542	1.42

Table 3.3: Comparing the runtime(secs) between flat RDD and hierRDD for 1 million data points

npoints	hierrdd	bisecting	speedup
750	106	256	2.42
1500	114	306	2.68
3000	121	384	3.17
6000	138	519	3.76

Table 3.4: Comparing the runtime(secs) between hierRDD and naive solution

points our implementation is 2.4× faster, and that increases to 3.7× for 6000 points.

Figure 3.11 is a time plot for various data points, from 3000 to 96000, which means the lower value the better, for depth 8 (the leaves have 256 nodes), utilizing 5 slaves (10 executors). Three versions of the bisecting k-means are compared, the default algorithm without the hierarchical RDDs, while the other two use the hierarchical RDD abstraction, one with sequential job issue and one with parallel job issue. The results show that the hierarchical RDDs implementation incurs zero overhead when done sequentially.

Chapter 4

Related Work

4.1 Related Work

4.2 Nested Parallelism

Cilk [10] was the first compiler-runtime that helps the programmer express recursive parallelism. They introduced a spawn operation that can be executed in parallel and implemented job scheduling using concurrent dequeues and work stealing. However Cilk does not support dependence analysis, which may result to unnecessary stall cycles. Modern runtimes such as (ParTEE), support dynamic dependence analysis, by using custom memory allocators to keep track of each task's memory footprint. Another approach to recursive-hierarchical parallelism is Sequoia. Sequoia [11] is a programming language designed to facilitate the development of memory hierarchy aware parallel programs. Sequoia provides a language abstraction helps express the hierarchical split and communication, in an abstract and portable way.

4.3 Distributed Parallelism

Distibuted applications can be expressed like shared memory applications using abstractions like UPC [12]. In this work we focused on task based distributed programming models. X10 [13] is an object-oriented language, that offers a bunch of primitives such as async, future, and foreach to enhance distributed programmability. Ciel [14] was one of the first distributed runtimes to support nested tasks, using a dynamic task graph that stores the relations between tasks and objects.

4.4 Frameworks for distributed data analytics

Among Spark, there also exist other engines for distributed data analytics. Stratosphere [15] treats data parallelism, in a very similar way with Spark, by providing the user with

distributed data collections, along with a set of operators to transform the data. In addition to those operators, Stratosphere supports data pipelines, oftenly used in machine learning applications. Naiad [16] introduces a new programming model for data parallel programms, called timely dataflow. Naiad supports structuted loops, stateful dataflow vertices, and also notifications.

4.5 Nested Queries

Spark has support for the execution of SQL queries, along with a compiler(Catalyst) that generates efficient bytecode. Those queries cannot contain recursive calls. Shkapsky [17] et al. implemented the datalog query language, that can create recursive queries in Spark. Their work is similar to ours, they enabled the execution of recursive Datalog queries, while we enabled nested RDD queries. However they are not identical, and maybe they could be combined.

4.6 Scheduling Related Work

Scheduling and resource management is a challenging problem that cannot be perfectly solved. However there exist many appximate solutions. The default scheduling mechanism Spark implements is called Delay scheduling. Delay scheduling [18], addresses the issue of locality and fairness gap in task scheduling. In HDFS clusters the tasks are usually scheduled where the data are stored. This way a task can wait for a long time before being launched, resulting to unfairness. To solve this issue delay scheduling sets a small time for each task to launch on local data, and if a timeout occurs, it is executed on the next free node.

In [19] Ousterhout et al. propose a new alternative scheduling algorithm, called Sparrow. Sparrow is a decentralized scheduling mechanism, that uses a power of two choices technique to improve the load balancing effiency. In Sparrow setup, there is a fixed number of workers and schedulers, and every scheduler can send tasks to every worker. Each worker has a task queue. The scheduling of a job is assigned to a random scheduler, which for each task, sends a task probe to 2 random workers. When the worker dequeues a task probe, it asks the task binary from the scheduler, and the corresponding scheduler sends the task to the worker that asks first. The Sparrow setup was very influential to our scheduler design; however our design treats load imbalance within the same machine instead of interacting with the scheduler.

In [20] Malte Schwarzkopf et al. deployed a distributed scheduling mechanism, called Omega. In Omega each scheduler has full access to the cluster, thus the state is shared. Each scheduler is given a private, local, frequently-updated copy of the cluster state that it uses for making scheduling decisions. They compare Omega against a monolithic scheduler and a two-level scheduling based on the Mesos design.

4.7 Straggler Mitigation

In [21] Ousterhout et al. address the problem of straggler mitigation by increasing the task granularity, resulting in a lot of very small tasks. They found a 5.2x improvement in response time due to the use of small tasks. SkewTune [22] is a Hadoop extension tries to eliminte skew in map reduce jobs. The approach SkewTune follows is that it first identifies using some heuristics. To mitigate a straggler that occurs in either the map or the reduce phase SkewTune repartitions the remaining data, in order to increase parallelism. Yadwadkar et al. describe in [23] a way to reduce the straggler mitigation problem to multi-task learning. They use multi-task learning because similar nodes or similar workloads may behave differently duting execution. The classifiers they train can predict if a task will be a straggler, creating a separate model for each cluster node. In [24] they introduce KMN a framework that optimizes jobs that use a subset of the data. In order to speed up the execution they launch some extra tasks, using combinatorial heuristics, and select the output of the fastest ones.

4.8 Constraint Scheduling

Quincy [25] is a framework that schedules concurrent distributed jobs that share common resources. Quincy models the tasks and computing nodes as a DAG, with the weights representing competing demands for locality and fairness, and tries to solve the problem by finding the maximum flow in the graph.

Tetris [26] is an alternative way of task scheduling that also considers the task's network and disc utilization requirements, in contrast with most of the other schedulers that mainly consider CPU and memory constraints. Tetris treats the multi-resource scheduling as a multi-dimensional bin packing problem. To make scheduling more efficient, Tetris uses an heuristic that assigns a task to the machine that maximizes the <task,machine> product value. Another heuristic combined by Tetris is to assign resources to the job with the smallest remaining time.

In [27] Chowdhury et al. introduce Orchestra, a global management architecture that focuses on optimizing the transfer time for a set of communication patterns such as broadcasting and shuffling, both commonly used in MapReduce environments. Orchestra has an Inter-Transfer Controller, that implements the scheduling policies, and selects a mechanism for transfers based on the data size, the number of nodes, their locations and other factors. To optimize shuffle performance, they introduce the Weighted Shuffle Scheduling that allocates rates to each flow using weighted fair sharing.

GRASS [28] focuses on approximation algorithms, where there is either a deadline in time, either a error threshold. GRASS is a scheduling technique that combines two policies. The first is the Greedy Speculative scheduling, that selects the next task based on the approximation goal. The second is the Resource Aware Speculative, that schedules a task copy if it saves time. GRASS was implemented in Hadoop and Spark, and results show up to 47% speedup.

Zaharia et al. developed LATE [29], an alternative job scheduler for Hadoop, that

tackles the scheduling problem for heterogeneous clusters. The default Hadoop scheduler measures the progress of every task and if a task seems to take longer than expected, the scheduler launches a copy of the first task in another machine. In general, LATE aims at speculating the task that seems to have the greatest expected time to finish. LATE monitors each task and computes the progress score, from which it computes the task remaining time. It and also maintains a per node progress, that measures the task throughput of each node, that is used to indicate not to schedule a task to a node whose progress rate is below some threshold.

Mantri [30] was deployed and evaluated on a Bing's cluster, to mitigate stragglers caused by bad machines, data loss and crossrack traffic. The main parts of the scheduler are the resource-aware restart, network placement and recomputation avoidance. Resource-aware restart decides whether to restart or duplicate a task, based on its progress and expected remaining time. Network placement refers to the reduce phases, and finds the best rack to start the reduce phase based on each racks ingoing and outgoing traffic. Also Mantri replicates the outputs whose cost to recompute is more expensive is considered more expensive than to duplicate.

4.9 Distributed File systems and caching protocols

The Google File System [31] is a distributed file system, deployed in the Google data center. It is designed for scalability and fault-tolerance, while running on commodity hardware. The architecture consists of a master server that holds all the file metadata, and a multitude of chunk servers, that hold the data in fixed size chunks. In order to avoid extra synchronization, the only supported operators are read and append.

Pacman [32] is a caching service that coordinates access to the distributed caches. The goal is to minimize the job completion time. They implemented and measured two cache replacement policies, LIFE and LFU-F. LIFE favors the jobs that have the smallest wave-widths, while LFU-F gives priority to the most frequent used files.

4.10 Hierarchical Clustering

Hierarchical clustering has many advantages over traditional clustering, and applies to a wide variety of applications, but extracting parallelism can be challenging. The SHRINK [33] algorithm is a shared memory algorithm for single linkage hierarchical clustering that works by partitioning the initial data set into partitions, for each pair of partitions it builds a dendrogramm, and then starts combining those dendrogramms together.

A distributed version of hierarchical KMeans, DISC [34] splits the dataset to partitions, and for each one it produces a local MST(minimum spanning tree) using Prims algorithm. At the next step the local MSTs are merged using Kruskal's algorithm. Jin et al. [35] implemented the aforementioned algorithm in Spark. However their approach is different than ours because we implemented the bisecting KMeans algorithm described in [8].

Chapter 5

Conclusions and Future Work

We present an extension of the default Spark scheduler that supports nested RDD operations. This extension allows the programmer to express recursive computations intuitively, as we have demonstrated by using it to implement the Barnes-Hut n-body simulation in Spark. Since our extension of the RDD abstraction tends to create many small jobs, as are often found in parts of large analytics pipelines, we extended the default Spark scheduler with distributed scheduling. We evaluated our extensions and found that they outperform the standard Spark scheduler on a set of benchmarks.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[2] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[4] Apache Software Foundation, "Hadoop." [Online]. Available: https://hadoop.apache.org

[5] Databricks, "Spark survey results," 2015.

[6] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proc. VLDB Endow.*, vol. 8, no. 13, pp. 2110–2121, Sep. 2015. [Online]. Available: http://dx.doi.org/10.14778/2831360.2831365

[7] Wikipedia, "k-means clustering." [Online]. Available: https://en.wikipedia.org/wiki/K-means_clustering

[8] M. Steinbach, G. Karypis, and V. Kumar, "A comparison of document clustering techniques," in *In KDD Workshop on Text Mining*, 2000.

[9] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 57–76. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297033

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system,"

SIGPLAN Not., vol. 30, no. 8, pp. 207–216, Aug. 1995. [Online]. Available: http://doi.acm.org/10.1145/209937.209958

[11] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543

[12] G. Almási, P. Hargrove, I. Gabriel, and T. Y. Zheng, "Upc collectives library 2.0," in *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094852

[14] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A universal execution engine for distributed data-flow computing," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 113–126. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972470

[15] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014. [Online]. Available: http://dx.doi.org/10.1007/s00778-014-0357-y

[16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 439–455. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522738

[17] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1135–1149. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915229

[18] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*,

ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755940

[19] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 69–84. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522716

[20] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 351–364. [Online]. Available: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf

[21] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 14–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2490483.2490497

[22] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213840

[23] N. J. Yadwadkar, B. Hariharan, J. Gonzalez, and R. H. Katz, "Faster jobs in distributed data processing using multi-task learning," in *SDM*. SIAM, 2015, pp. 532–540.

[24] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 301–316. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/venkataraman

[25] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 261–276. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629601

[26] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 455–466. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626334

[27] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 98–109. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018448

[28] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 289–302. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/ananthanarayanan

[29] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744

[30] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924962

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450

[32] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 267–280. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/ananthanarayanan

[33] W. Hendrix, M. M. A. Patwary, A. Agrawal, W. keng Liao, and A. N. Choudhary, "Parallel hierarchical clustering on shared memory platforms." in *HiPC*. IEEE Computer Society, 2012, pp. 1–9. [Online]. Available: http://dblp.uni-trier.de/db/conf/hipc/hipc2012.html#HendrixPALC12

[34] C. Jin, M. M. A. Patwary, A. Agrawal, W.Hendrix, W. k. Liao, and A. Choudhary, "Disc: A distributed single-linkage hierarchical clustering algorithm using mapreduce," in *Proceedings of the 4th International SC Workshop on Data Intensive Computing in the Clouds (DataCloud)*, 2013.

[35] C. Jin, R. Liu, Z. Chen, W. Hendrix, A. Agrawal, and A. N. Choudhary, "A scalable hierarchical clustering algorithm using spark," in *First IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2015, Redwood City, CA, USA, March 30 - April 2, 2015*, 2015, pp. 418–426. [Online]. Available: http://dx.doi.org/10.1109/BigDataService.2015.67