FP7 Project ASAP

Adaptable Scalable Analytics Platform

ASAP D4.3

Execution Engine v.2

WP 4 – Dependency-aware query execution engine

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	05 Feb 2017	P. Pratikakis, P. Katsogridakis	Initial Version
0.2	09 Feb 2017	P. Pratikakis	First Revision
0.3	15 Feb 2017	P. Pratikakis	Second Revision
1.0	28 Feb 2017	P. Pratikakis	Final Version, review by H. Vandierendonck

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Executive Summary

This deliverable describes work done in WP4 during the third year of the ASAP project and presents evaluation results. WP4 has successfully produced an adaptation of the Spark analytics execution engine that allows programmers to express previously impossible queries, by allowing full recursion in the User Defined Functions. Moreover, we have produced an alternative scheduler for Spark that outperforms the standard scheduler by up to a factor of 2.5$\times$. The deliverable describes the overall design and implementation of the Spark extensions, the evaluation methodology, and evaluation results for a wide set of benchmarks that include generic representative benchmarks, microbenchmarks designed to stress certain aspects of the system, as well as benchmarks taken from the ASAP industrial applications.

Contents

1 Introduction

1.1 Task Description

This deliverable describes work performed in tasks T4.2 Engine dependence analysis, and T4.3 Engine scheduler, by FORTH and QUB. The tasks aim to produce a parallel and scalable scheduler for analytics computations, that allows "worker" nodes to also perform scheduling and reduce scheduling overheads. The scheduler developed maintains all task dependencies and does not relax fault-tolerance and elasticity of the previous Spark scheduler. During Y3 of the ASAP project, the scheduler produced during the second year was optimized and performance was improved from being several times slower than the default scheduler (cost caused by supporting recusive computations) to a speedup of 7x in some cases (corner cases of jobs with many small tasks where the scheduler becomes a bottleneck).

1.2 Execution Engines Limitations

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 [13] 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. 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 [7] 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 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 is far enough from all particles in the

```
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: N-Body recursive query

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

Figure 2: Word count in Spark

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 WP4 we extended the Apache Spark MapReduce engine [35] to directly support such nested and recursive computations. Spark is an implementation of the MapReduce model that outperforms Hadoop [6] 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 [12, 28].

1.3 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 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 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 3 shows the graph of RDDs created by the program in Figure 2.

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*

and *driver*, 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.

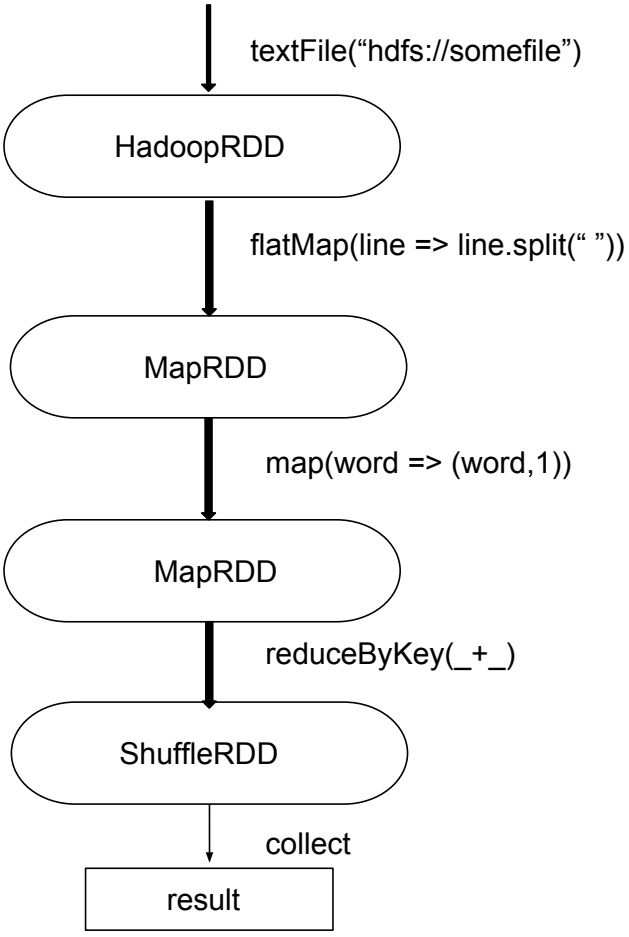

Figure 3: The dependency graph for the word count example

This deliverable presents work performed in WP4 that 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 driver 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.

1.4 Third year of the ASAP Project

In short, the main achievements of the work done during the third year of project ASAP are:

- We continued and finished work that adds support for nested RDD queries in the Spark scheduler. We performed an extensive evaluation of the resulting Spark engine on 3 different cluster environments and compare our extensions against built-in operators implemented without nesting. We adapt operators from the WIND application to use

our Spark and show measurable benefits to performance and scalability. To demonstrate the generic usability of the programming model extension beyond ASAP applications, we implement an N-Body particle simulation using the nested RDD mechanism.

- We optimized the RDD module —developed during Y2— that enhances the expression of applications with hierarhical parallelism. We use a hierarhical K-means operator we implemented to measured its performance in a Spark cluster and find considerable performance improvement compared to existing open-source hierarchical k-means implementations.
- 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 2.1$\times$ for computations using fine-grain tasks.

Overall, during the third year of the project we were able to overcome the limitations in performance reported in deliverable D4.2 and achieve better or equivalent performance to the existing Spark engine, while offering superior expressibility to its query language.

2 Extending the Spark engine to support nested operations

2.1 Use case

By default, Spark does not support nested RDD operations, because some of the RDD metadata are known only by the master. Assume that we want to execute the code in Figure 4, that for each word `word1` in `file1`, computes the set of words in `file2` that have `word1` as prefix. In this computation, the outer map function called by the file1 RDD, contains a filter operation on the file2 RDD. This is an example of one level nesting operator. This section discusses the modifications in the Spark runtime to enable nested RDD operations.

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

Figure 4: example of nested RDD operations

2.2 RDD internals

Internally, each RDD is characterized by five main properties:

- a set of partitions (e.g HDFS blocks)
- a dependency list on parent RDDs (the RDDs lineage graph)
- 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
- (optionally) a list of preferred locations to compute each partition on (e.g. block locations for an HDFS file)

All of the scheduling and execution in Spark is done based on these methods, allowing each RDD to implement its own way of computing itself.

Additionally, each RDD is accompanied with:

- a unique ID
- a reference to the spark context that instruments the execution of the calculation on the cluster

2.3 Design to support nested RDD operations

Handling nested RDD operators inside the executors means that the executor will also have to schedule the job created by that RDD. This wound be harsh to design and implement(but not impossible), and would result in very complex architectures and network communication, contradicting the simplicity of the MapReduce model. Thus we chose to forward the nested

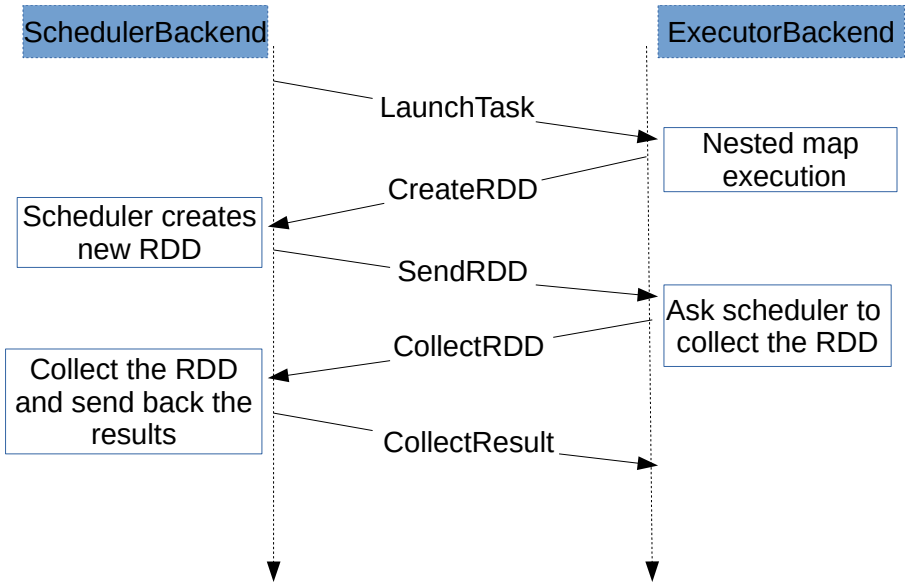

Figure 5: Executor asks the master to perform an RDD operation

operators back to the master, in order to avoid a distributed scheduler setup. To accomplish the forwarding of nested tasks, we added a few extra Driver-Executor messages, that allow the executor to ask the driver to create new RDDs and run nested jobs. For example, Figure 4 shows code that contains a nested RDD operation. The outer collect method forces the runtime to schedule the outer RDD computation. Since no shuffle operations are involved, the DAG graph will consist of only one stage. This stage will contain one transformation of the HadoopRDD (corresponding to the file1) to a MappedRDD as defined in the *map*. The 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 driver will create tasks which literally will force the nested code to be executed for each word of the file1. Each task will be serialized and sent to an idle executor. As soon as the executor will receive the task, it will try to apply the computation on its partitions of the RDD. When the executor tries to invoke the nested map operation, it figures out that it is on executor mode, thus cannot create RDDs, so it sends a CreateRDD message to scheduler with (rddid,"map",function) as arguments. Spark is implemented in Scala, a functional language that supports anonymous functions, thus it is straightforward to send functions as arguments from the executor to the driver. Then the scheduler, looks up the RDD with the specified id, and using JVM reflection, invokes the "map" method, creating the desired RDD. Then the scheduler sends back to the executor the id of the created RDD (SendRDD msg). Now the worker creates promise of the RDD based on the id received. When the nested collect is called the executor sends the CollectRDD message, asking the scheduler to collect the file2 RDD, and send back the result. Figure 5 shows the sequence of messages that have to be sent.

2.4 Nested Task Result forwarding

In the naïve driver-executor protocol described above the master after receiving the CollectRDD message, schedules the nested job. After the nested job is done it receives the result from the executors the job was scheduled and sends the result to the executor that issued the nested operation. This implies an unnecessary transfer of data to the driver, because the data are needed by the executor that issued the nested job. Thus we modified the executor code to send the task result directly to the executor that issued the nested job, and

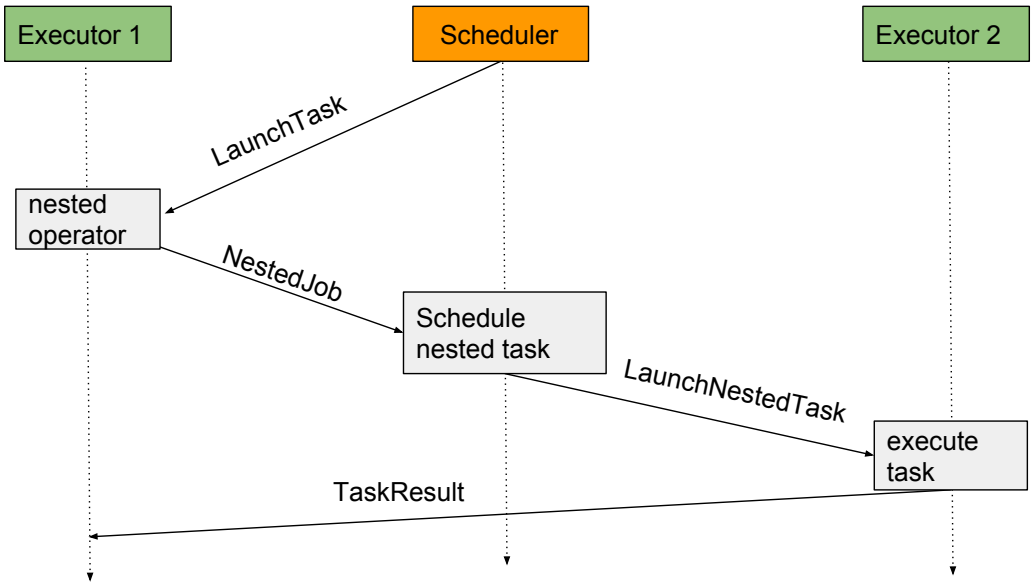

Figure 6: Forwarding the nested task result to the executor that issued the query

also send an ack to the driver that the nested task finished, in order to free the executor resources. Figure 6 shows the message flow between the driver and two executors executing a nested query.

2.5 Nested operators packaging

The RDD mechanism is lazy, which means that the transformations are not computed directly. Instead they form a computational graph. Every operator applied to an RDD triggers an addition of an edge to the corresponding graph. The graph nodes(operators) are grouped into stages, that are later packed into tasks sent to the workers. Following the protocol described in Section 2.3 the executor sends a CreateRDD message to the driver. The driver applies the message, adds a new node to the RDD graph, and sends back the new RDD. This means that if a nested RDD query contains lots of operators, then many CreateRDD-SendRDD messages will be exchanged to the driver from each executor. Those messages increase the latency and network communication. This extra communication is unnecessary, since those messages do not prompt any job creation. We solved this problem by grouping all the nested operators into a single message. The RDD laziness permits us to pack the rdd operator arguments in a per executor global data structure. When the nested collect is triggered, the executor sends all the RDD transformation list to the driver; then the driver, using JVM reflection, reconstructs the RDD transformations and schedules the nested RDD.

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.

3.1 Introduction

K-means [18] is the most common algorithm used in cluster analysis, which aims to partition the elememts into k clusters, such that each element 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 assigned 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 [30]. 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.

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.

3.2 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

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

Figure 7: API for creating hierarchical RDDs

```
class Cluster(id: Int, iter:Int)
  extends Splittable[Vector]{

  def id() = id
  var data:RDD[Vector]

  var m = KMeans.train(data, k=2)

  def contains(point: Vector) = {
    m.predict(point)==id
  }

  def split(level:Int) = {
    m.clusterCenters.map{
      case(c,idx) =>
        new Cluster(idx, iter)
    }.toArray
  }

  def splitPar(level:Int) = {
    split(level).par
  }
}
```

Figure 8: Splittable subclass for the Bisecting K-means benchmark

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 7. The Splittable object represents 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 8. We use this trait to code the bisecting K-means algorithm, discussed below. The constructor takes as arguments the identity of the cluster, and the number of iterations (k-means specific). 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

```
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 9: Bisecting K-means implementation with hierarchical RDDs

new Cluster instance with the id of the cluster.

Figure 9 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.

4 Scheduling

4.1 Motivating example

Consider the code shown in Figure 10, 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 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 10: Many tiny tasks micro benchmark

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 driver 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 *ProxySceduler* 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 proxy schedulers; each proxy 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 proxy schedulers, and instead allow all proxy 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. Figure 11 shows the new parallel scheduler architecture.

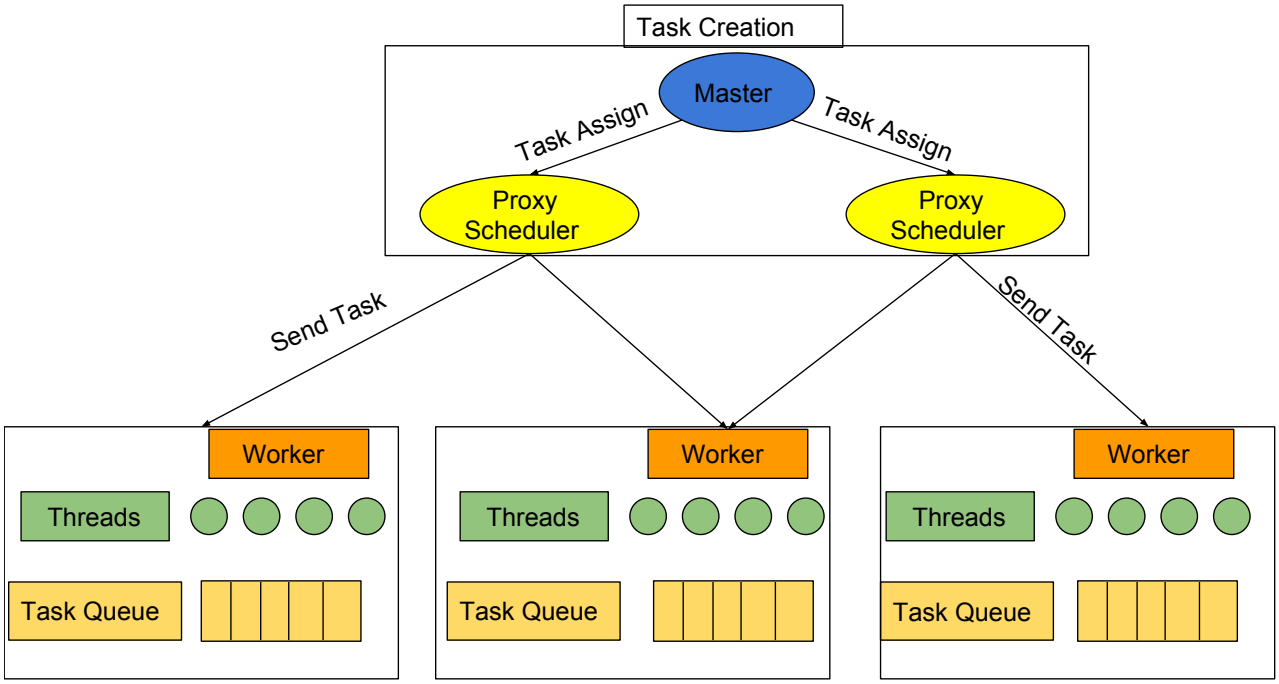

Figure 11: Scheduling Architecture

4.1.1 Scheduling Policy

The scheduling of a single task by each scheduler is simple; if the task has no locality preferences the scheduler picks a random worker and sends the task, alternatively the task is sent to one of the prefered locations. Also, multiple tasks can be sent at once under load, so that the worker queues are full.

4.1.2 Distributed Scheduling Algorithm

The main function of submitting a stage is done by the master. The stage is a part of the DAG that is ready to execute, and contains a list of RDD functions that are executed for each partition, and some metadata. First the task is serialized so that it can be transfered trough the network to the executors. Then the task binary is broadcasted to all workers. The stage is split into individual tasks that, that share the same binary code, each one referring to a different partition, indluded in the TaskSet class. The above scheduling process is necessarily done sequentially. Then in the default Spark scheduling mechanism, the taskset is added to the queue of active tasks, and the scheduler decides which task to schedule next based on fairness and locality factors. Our approach was to make this procedure more parallel and lightweight. First we split the tasksset into chunks of tasks, each one sent to a random proxy scheduler. Each scheduler that receives a tasks chunk, picks an executor following a scheduling policy, and sends the task. As a scheduling policy, currently we check if the task has any preferred workers (if it is cached). If it has, we send the task to that specific worker. Else we select an executor at random.

Each worker maintains a global variable that denotes how many cores are available, hence how many tasks are executed at that moment. When the executor receives the task, it checks the number of free cores. If the available cores is positive, then the task is executed immediately. Else, it is pushed to the task queue for later. Similarly, when the worker finishes

```
def Executor:executeTask(task){
  if(availableCores>0){
    --availableCores
    threadPool.execute(task)
  } else {
    taskQueue.enqueue(task)
  }
}

def Scheduler:ProxyLaunchTasks(taskset){
  taskset.foreach( task =>
    rid = selectExecutor(task)
    executors(rid).send(LaunchTask(task))
  )
}

def Scheduler:LaunchTasks(taskset){
  val splits = taskSet.split(nschedulers)
  par foreach (proxy,subtasks) in splits =>
    proxy.send(ProxyLaunchTasks(subtasks))
  )
}
```

the execution of a task, if the available cores is positive, he tries to execute another task from the queue. We chose the above algorithm for scheduling because it is the most lightweight, and won't add much overhead to the scheduling path.

4.1.3 Scheduler state

To schedule and track tasks to executors, each proxy scheduler keeps a copy of all the executor metadata that the default Spark master normally maintains. This creates a 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 proxy 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 proxy 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 proxy 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 broadcasts to all proxy schedulers the state of the newly registered worker. Eventually, all the schedulers will have the same view of the cluster state.

4.1.4 StatusUpdates

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.

```
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()
7
```

Figure 12: 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 13: Scala code that gathers all the dataset to the driver

Currently, the proxy schedulers eventually forward all *StatusUpdate* messages to the central Spark scheduler. 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 proxy schedulers without any forwarding of that information.

4.1.5 Load Balancing

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 proxy schedulers to send tasks to all executors, we have removed the load balancing guarantees of the standard Spark scheduler. However, we found that by tranferring some of the master 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 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 5), but were unable to create such a scenario in practice.

Figures 12,13,14, 15,16 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 process becomes the bottleneck. However many of those computations are used in analytics applications.

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

Figure 14: Scala benchmark that adds all the elements jin a dataset

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

Figure 15: Each tasks sleeps some time according to long tail distribution

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

Figure 16: Word count benchmark

5 Evaluation

5.1 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 [15].

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 node
- 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 proxy 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 aformentioned 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 17 presents average running time of a simple filter operation on 5 million elements. The first column shows the number of partitions of the input RDD, which is equal to the number of tasks. The second column shows the average running time of the query using our distributed scheduler, in milliseconds. The third column shows the average running time using the default Spark scheduler. 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$\times$ and up to 1.86$\times$. 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

tasks	parallel	default	speedup
32	126.00	143.00	1.13
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

Figure 17: Filter on 5M elements

task. Note that as task granularity becomes smaller, both schedulers perform worse. We did not manage to explain the performance “knee” that the default Spark scheduler consistently reproduced for 512 tasks, as it was not correlated with idle time in the worker cores nor network traffic measured. We conjecture, however, that it may be an artifact of waiting to move fine-grain partitions between executors.

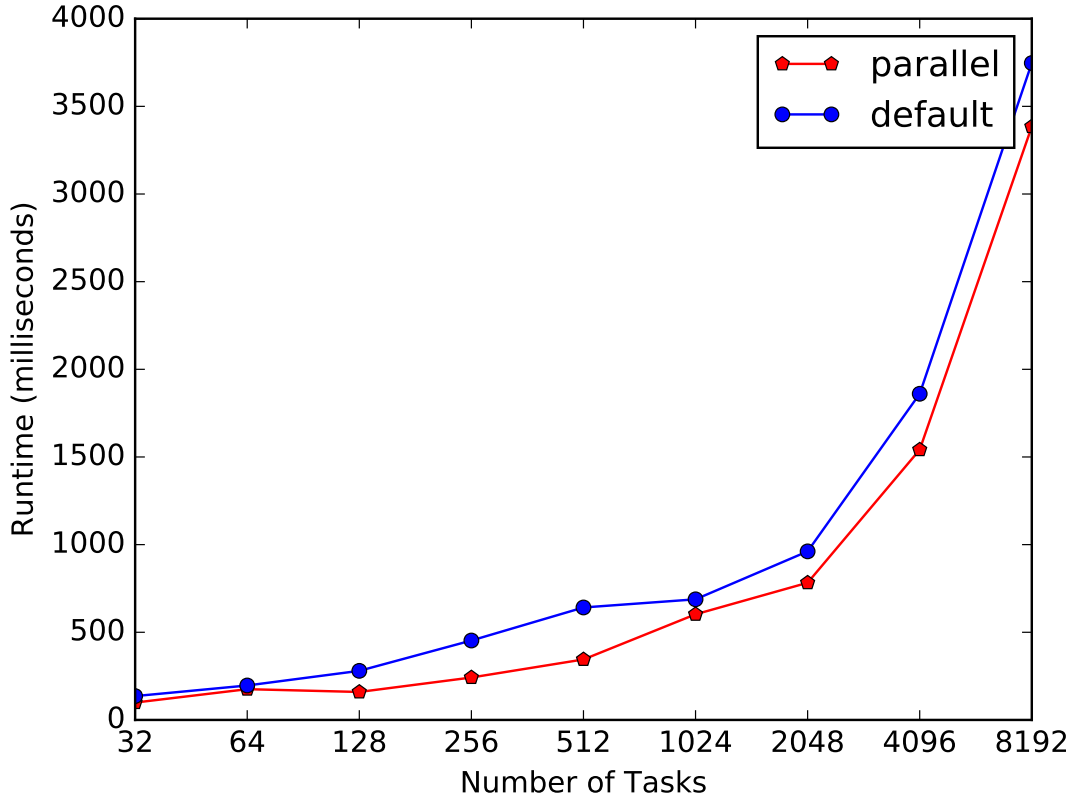

Figure 18: Reduce 5M elements

Similarly, Figure 18 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 1.12× to 1.87× better performance.

Figure 19 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 20 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 21 presents the comparison on a standard word count benchmark. Again, the distributed scheduler outperforms the default Spark scheduler by up to 2.15×.

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 1 shows the speedup of the distributed scheduling versus the vanilla Spark scheduler. Note that the

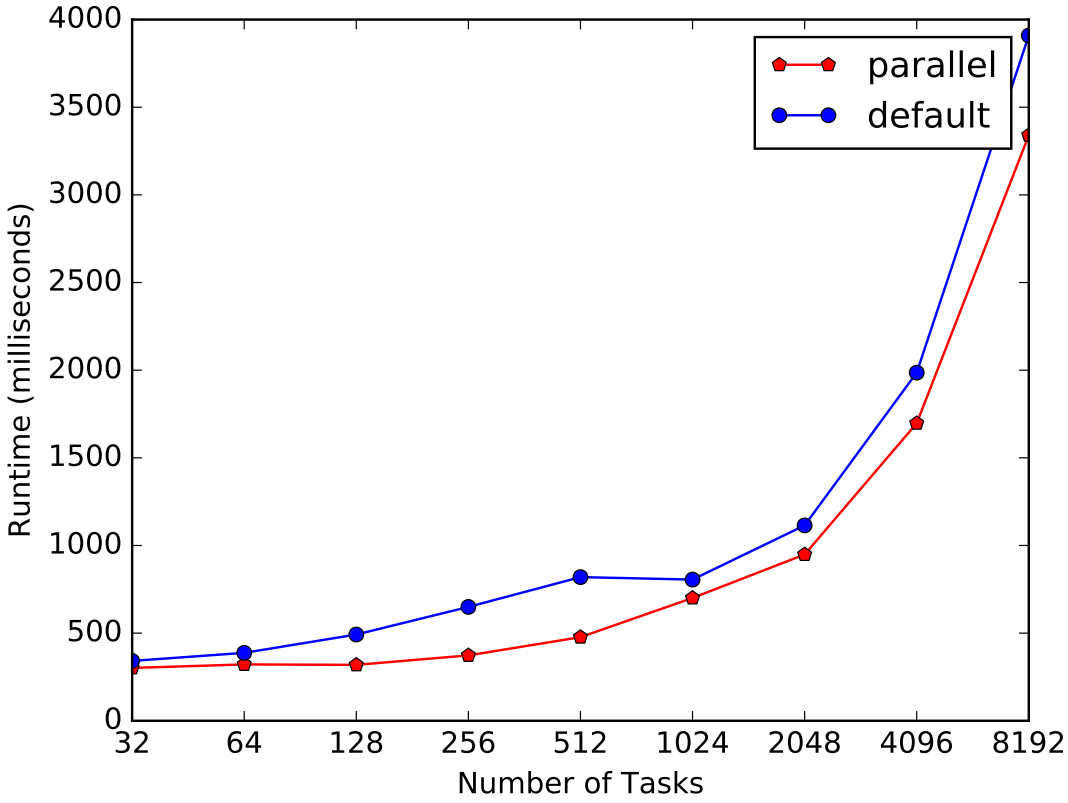

Figure 19: Collect all data at the master

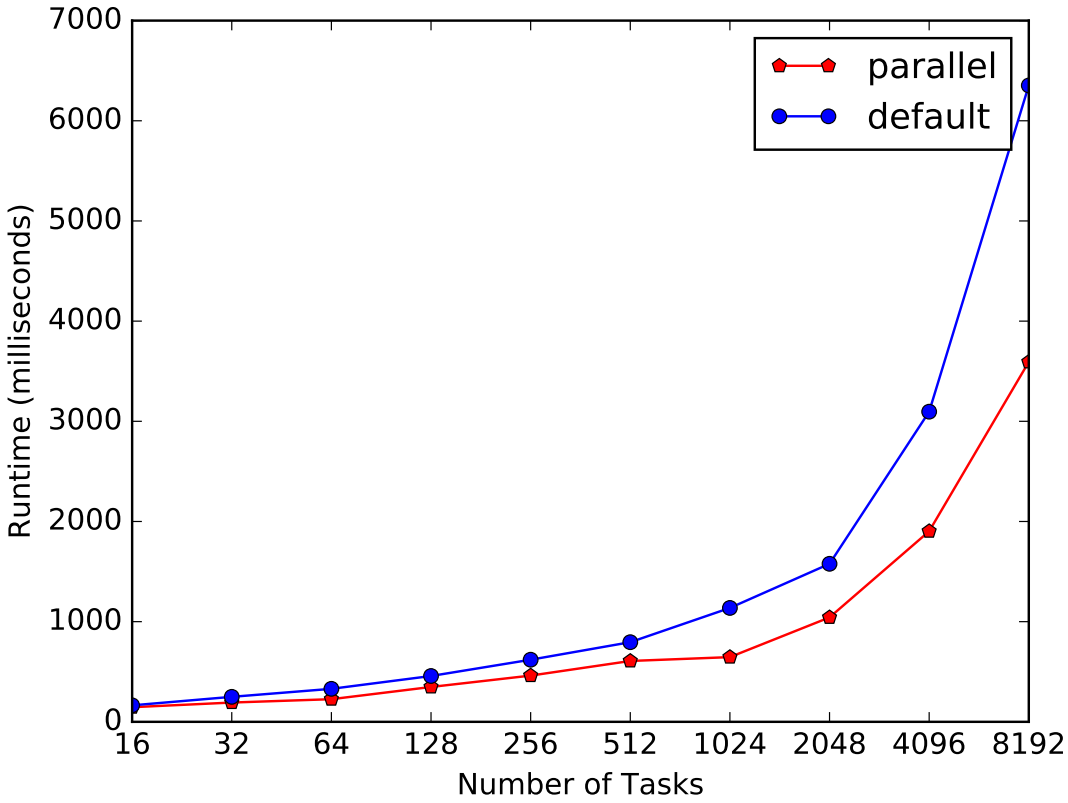

Figure 20: Long-tail distribution of task runtimes

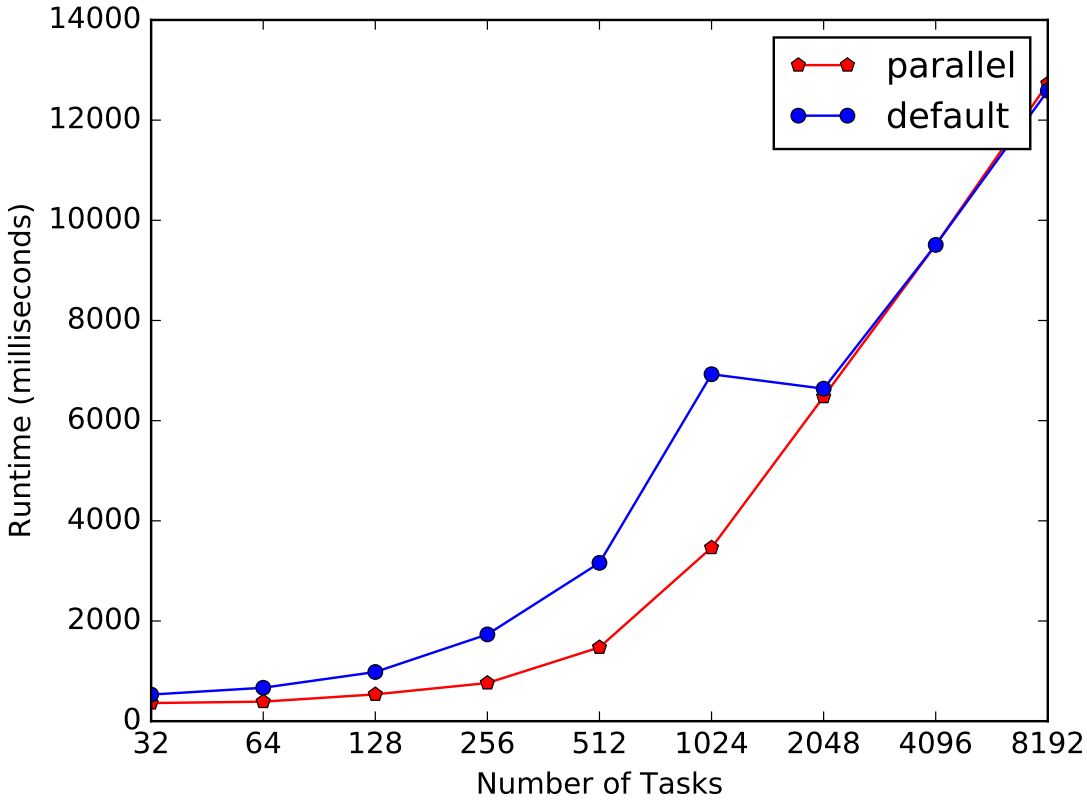

Figure 21: Word count

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

```
val file1 = sc.textFile("hdfs://file1")
val file2 = sc.textFile("hdfs://file2")
val cartesian = file1.map( e1 =>
  file2.map( e2 =>
    (e1,e2)
  ).collect().flatten).collect()
```

Figure 22: Cartesian product written in nested RDD query

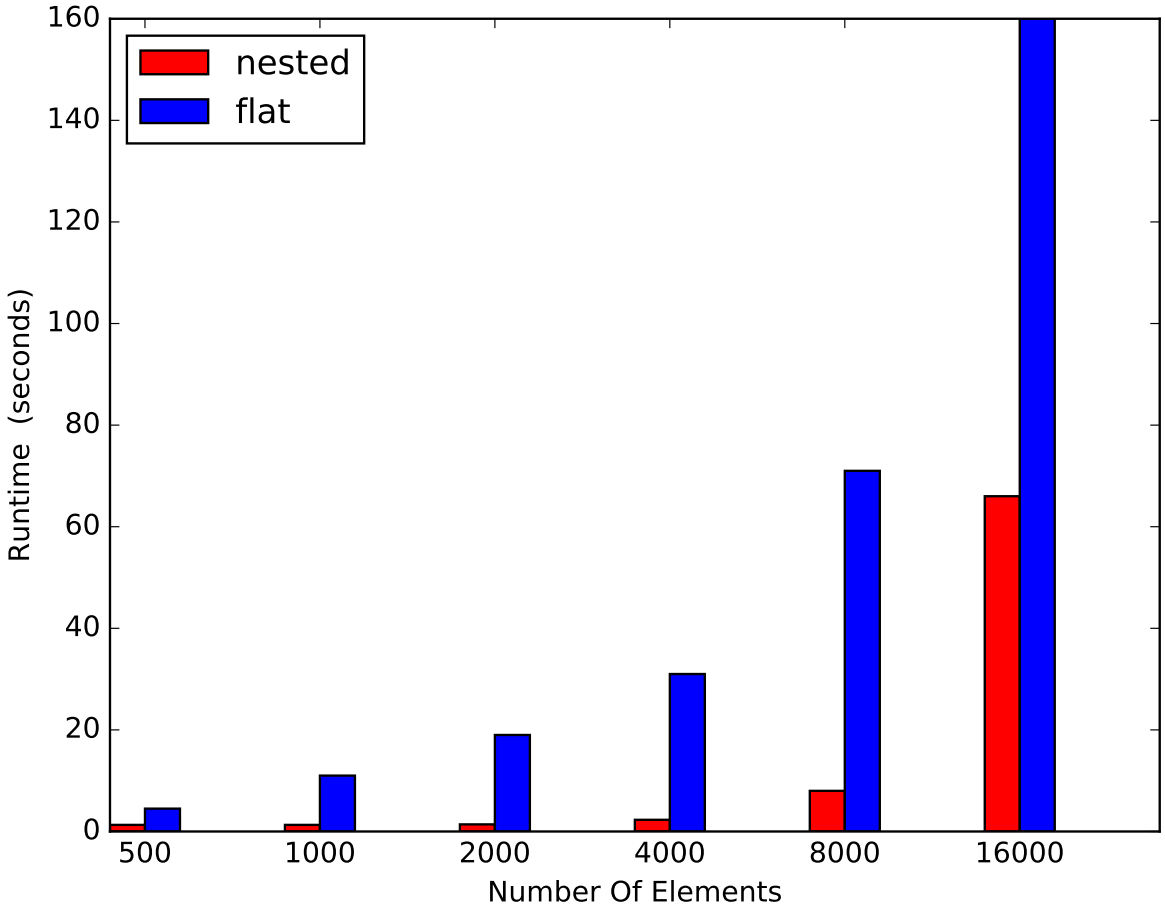

Figure 23: Comparison between flat and nested operators in cartesian product

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.

5.2 Nesting Evaluation

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 in Figure 22. For the flat, non-nested version we used the cartesian RDD operator.

Figure 23 compares the total running times between the two schedulers. We found that writing a cartesian product as a two-level nested RDD operation parallelizes it into smaller

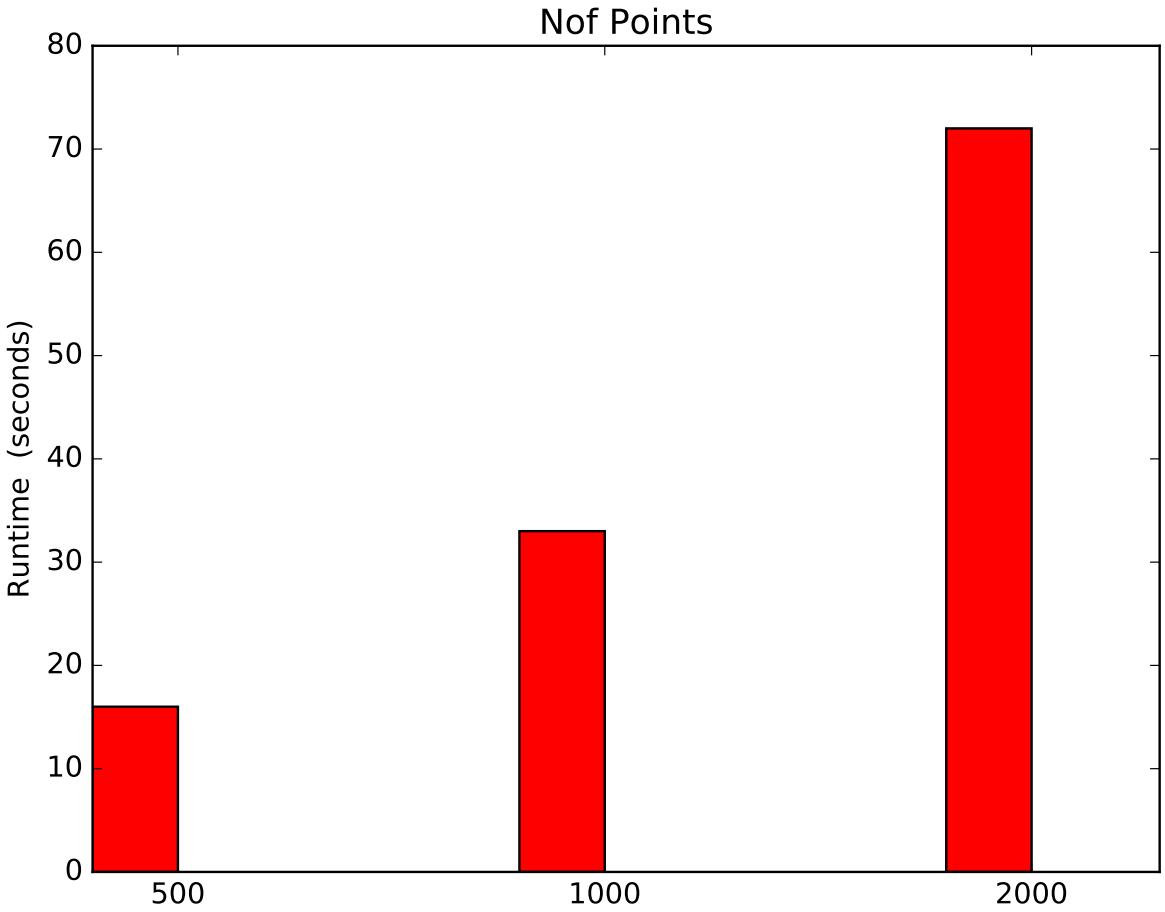

Figure 24: Barnes-Hut measurements

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 24 presents the results.

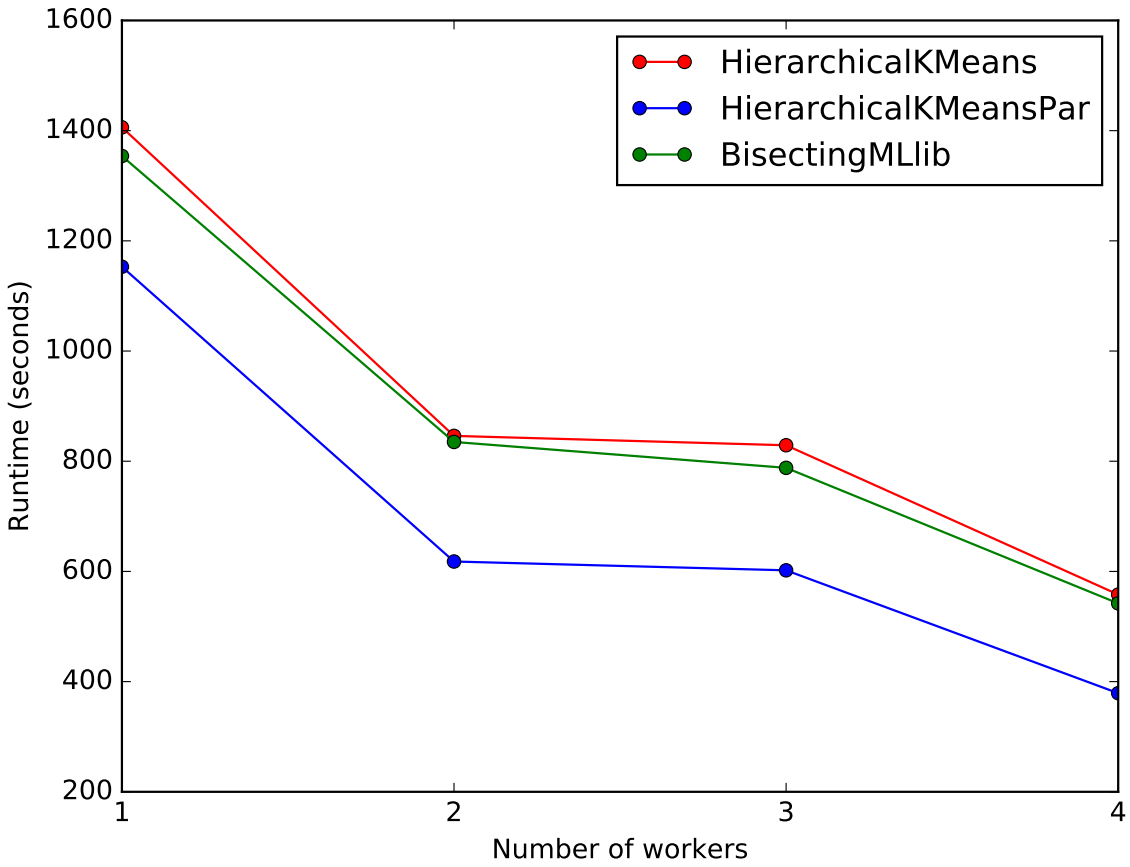

Figure 25: HierRDD: Strong scaling graph for 1 million data points

6 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 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 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 28 shows the time elapsed for both applications for various data points. For 750 points, our implementation is 2.4× faster, and that increases to 3.7× for 6000 points.

Figure 27 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 execu-

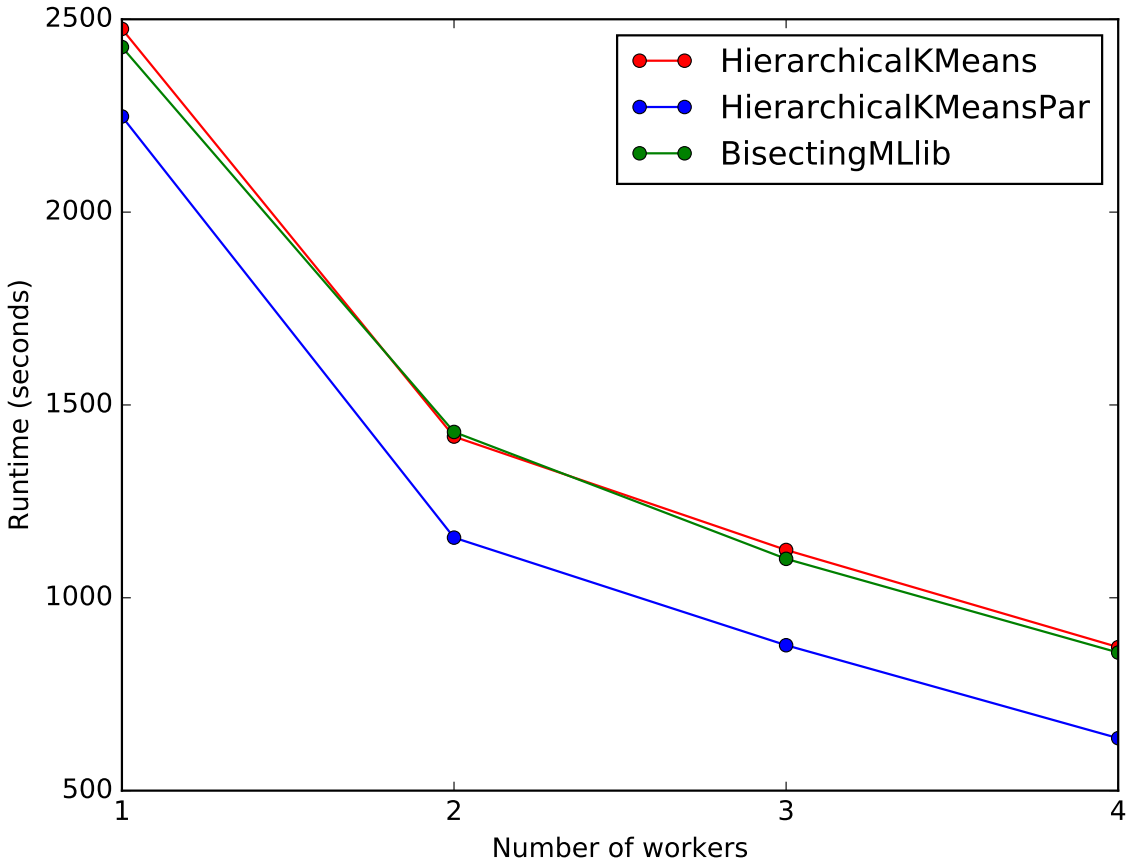

Figure 26: HierRDD: Strong scaling graph for 2 million data points

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

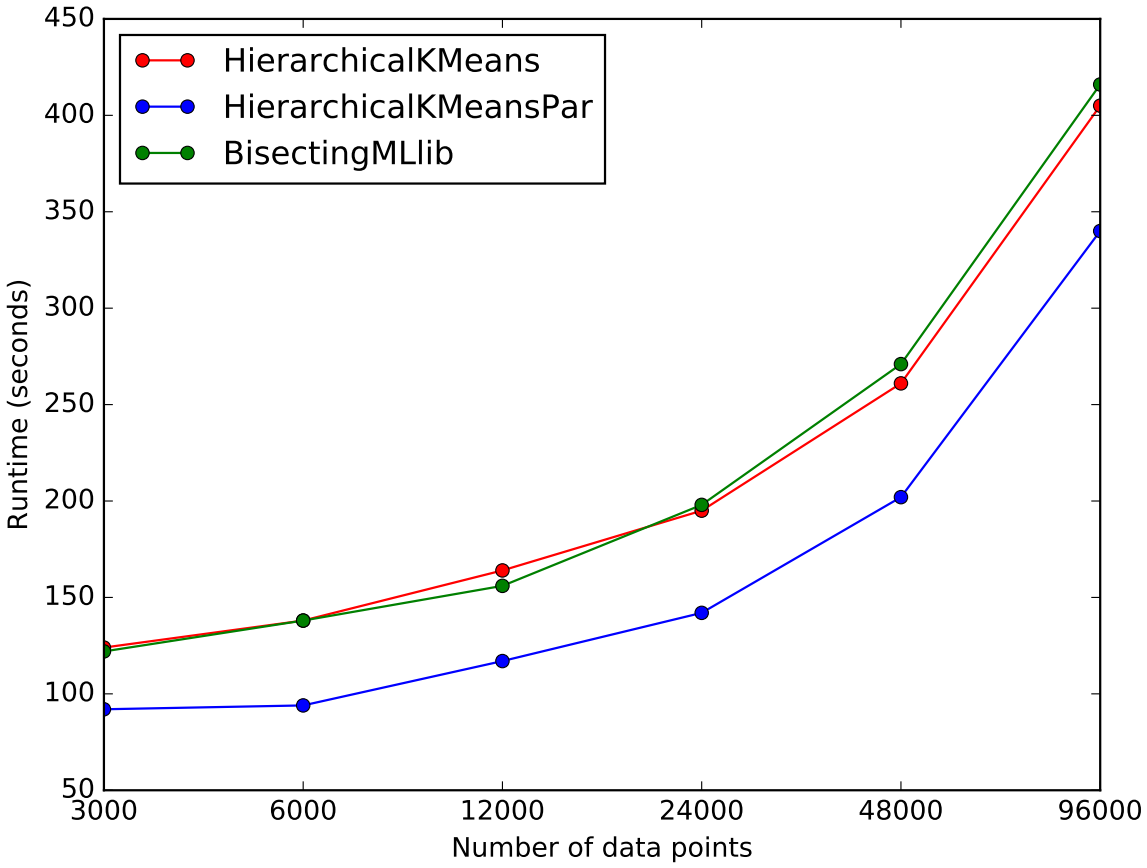

Figure 27: HierRDD: Weak scaling graph

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 4: Comparing the runtime(secs) between hierRDD and naïve solution

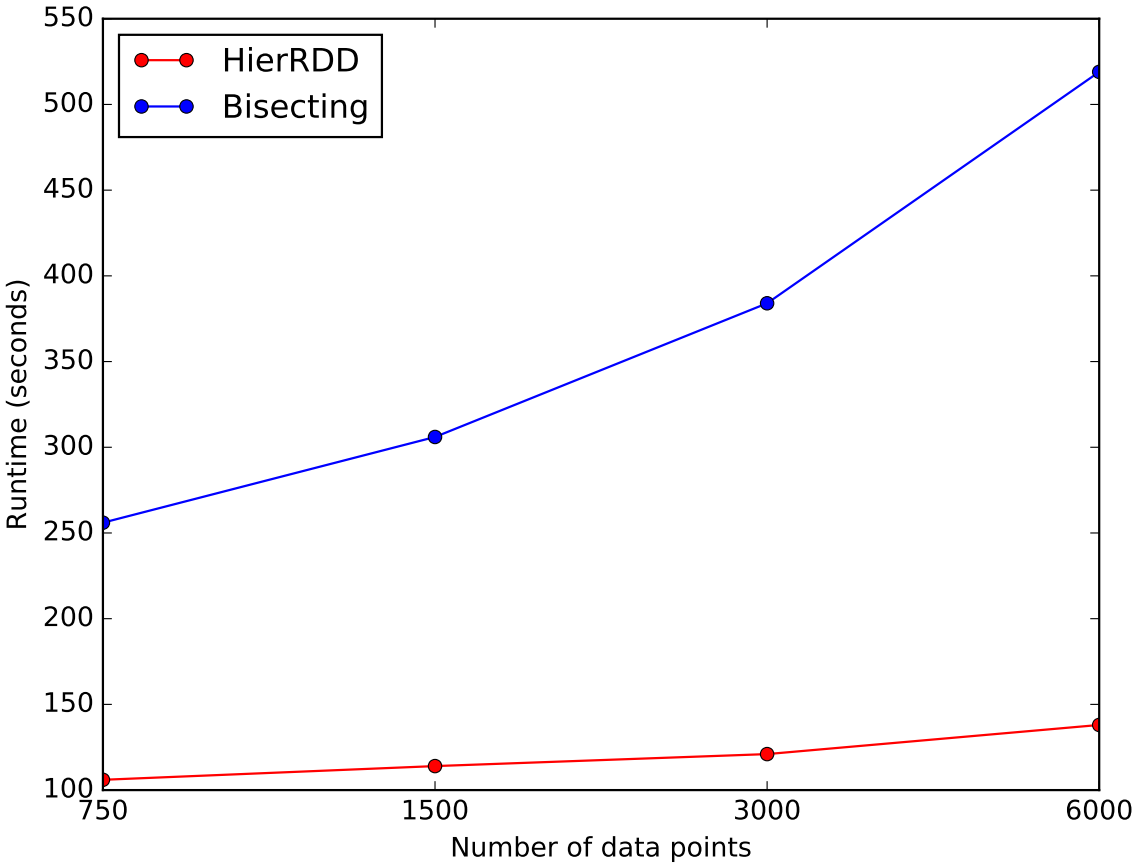

Figure 28: Naïve version versus hierarchical RDDs

Dataset	Size	Nested	Flat
1 hour	46.45K	33s	39s
1 day	450MB	4m	4.4m
1 month	12.6G	57m	OOM
1 month	17.3G	1.5h	OOM
2 months	30G	3.8h	OOM

Figure 29: Flat vs Nested query

tors). 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.

6.1 Evaluation on ASAP applications

We evaluated the effect of the Spark-Nesting scheduler on parts of the ASAP applications using the following process. Initially, we held two teleconferences with the WIND engineers for code review, in which identified interesting operators in the WIND application that were using iterative computation. We then identified points in the operator code that could be refactored to take advantage of Spark-Nesting, and performed refactoring to modify iterative queries to be recursive instead. Table 29 presents the effect on the Sociometer operator of the WIND application. We used synthetic, anonymized datasets that match all distributions of actual datasets, as provided by WIND, because this experiment was performed in the installation of Spark-Nesting on the FORTH cluster, and actual customer data cannot be allowed to leave the WIND data center. All reported running times are the average of 3 runs, on 5 PCs with 256GBs of main memory and 40 CPU cores. The first column shows the size of the data set, in terms of the time of user traffic simulated in the corresponding dataset. To analyze and classify users on data that amounts to 1 total hour of phonecall traffic, both Spark and Spark-Nested perform similarly, with a small speedup for the second, due to the better scheduling of small tasks produced by the parallel scheduler described above. A notable difference appears for larger data sets, namely the two 1-month's worth of data and their aggregate, where the default Spark implementation does not fit in memory and the computation terminates with an Out-of-Memory error. Being able to perform nested queries allows large chunks of computation that would not fit in memory, to be split into smaller jobs, allowing Spark-Nesting to run these operators in the available memory, without need for repartitioning into smaller data sets.

7 Related Work

7.1 Recursive Parallelism

Several task-parallel programming models offer high-level abstractions for expressing recursive and generic parallel computations. Most, like Cilk [8] and OpenMP [11], target shared memory architectures and require manual synchronization. Others, like Sequoia [14], support distributed computations but also require manual synchronization and focus on HPC systems rather than analytics or other big-data computations where fault-tolerance is key. Extensions of such runtimes with automatic dependence analysis and synchronization [26, 23] abstract much of the synchronization burden, although these systems still require considerable effort of parallelization compared to Map-Reduce models.

7.2 Distributed Parallelism

Distibuted applications can be expressed like shared memory applications using abstractions like UPC [2]. In this document we focused on task based distributed programming models. X10 [9] is an object-oriented language, that offers a bunch of primitives such as async, future, and foreach to enhance distributed programmability. Ciel [22] was one of the first distributed runtimes to support nested tasks, using a dynamic task graph that stores the relations between tasks and objects. Naiad [21] is a distributed data analytics engine, that executes a cyclic dataflow programs. Also Naiad introduces a new programming model, called timely dataflow. Stratosphere [1], now called Flink, is a distributed data analytics engine, whose programming model is similar to Spark. Flink targets micro-batching streaming applications, achieving very low latencies. However, Flink, like Spark, does not support recursive operators.

7.3 Nested Queries

Spark has support for the execution of SQL queries, along with a compiler (Catalyst) that generates efficient bytecode. Such queries, however, cannot contain recursive calls, and nested SQL statements amount to simply sequenced computations. Shkapsky [29] et al., implement the datalog query language that can express queries on recursive relations in Spark. Their work relates to ours, as datalog relations can be recursively defined, and may require a worklist or fixpoint computation. Recursively computed relations, however, may not be able to express fully recursive computations such as, for instance, the N-Body Simulation where the map and reduce functions are themselves recursive. Myria [32] is a big-data management system that executes iterative datalog queries incrementally and asynchronously. REX [20] introduces a new programming model similar to SQL, called RQL, that uses the notion of deltas (or small updates). REX also supports recursive relations that amount to iterative fixpoint computations.

7.4 Scheduling

Scheduling and resource management is a challenging problem that cannot be perfectly solved. However, there exist many appximate solutions. The Spark default scheduler uses Delay scheduling [34] to manage tasks. In HDFS clusters, tasks are usually scheduled where data are stored. This way, however, a task can wait for a long time before being

launched, resulting in unfairness. To solve this, issue delay scheduling sets a small time window to try and launch a task on local data, and if a timeout occurs, schedules the task on the next free node.

Ousterhout et al. [25] propose Sparrow, an alternative scheduling algorithm. Sparrow is a decentralized scheduling mechanism that uses a power-of-two-choices technique to improve the load-balancing effiency. In Sparrow, there is a fixed number of workers and schedulers, and every scheduler can send tasks to every worker. The scheduling of a job is assigned to a random scheduler, that sends each task probe to 2 random workers. When the worker dequeues a task probe, it asks for the task binary from the scheduler and the corresponding scheduler sends the task to the worker that asks first. Sparrow was very influential for our scheduler design; however, we found that treating load imbalance within the same node instead of interacting with the scheduler performs similarly or better.

Malte Schwarzkopf et al. [27] present Omega, a distributed scheduling mechanism. 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 for making scheduling decisions. In comparison, we chose to not replicate scheduling state between master and proxy schedulers in order to avoid the complication of maintaining all copies coherent, thus introducing additional fail points in the scheduling algorithm.

7.5 Straggler Mitigation

Ousterhout et al. [24] address straggler mitigation by increasing task granularity, resulting in a lot of very small tasks; they measure a 5.2$\times$ improvement in response time to stragglers. SkewTune [19] is a Hadoop extension that 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. [33] describe a way to reduce straggler mitigation 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. KMN [31] is 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. Dolly [3] is a system that proactively clones all the tasks in order to avoid stragglers, considering only the result of first task clone that finishes. Dolly also introduces a new technique, called delay assignment, used to avoid contention in intermediate data.

7.6 Scheduling Under Constraints

Quincy [17] schedules concurrent distributed jobs that share common resources. It 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 [16] uses an alternative way of task scheduling that also considers a task's network and disc utilization requirements, whereas most schedulers consider mainly 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.

Orchestra [10] is 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 scheduling policies and selects a mechanism for transfers based on the data size, the number of nodes, their locations and other factors.

GRASS [4] focuses on approximation algorithms, where there is either a deadline in time, or an error threshold. GRASS is a scheduling technique that combines two policies: (i) the Greedy Speculative scheduling that selects the next task based on the approximation goal, and (ii) the Resource Aware Speculative that schedules a task copy if it saves time. GRASS was implemented in Hadoop and Spark, and achieved up to 47% speedups.

LATE [36] is a 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 [5] mitigates stragglers caused by bad machines, data loss and crossrack traffic in Bing clusters, while taking networking, IO and memory constraints into account.

References

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014.

[2] George Almsi, Paul Hargrove, Ilie Gabriel, and Tnase Yili Zheng. Upc collectives library 2.0. In *Partitioned Global Address Space Programming Models*, 2011.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Networked Systems Design and Implementation*, 2013.

[4] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *Networked Systems Design and Implementation*, April 2014.

[5] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Operating Systems Design and Implementation*, 2010.

[6] Apache Software Foundation. Hadoop.

[7] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Object-oriented Programming, Systems, Languages, and Applications*, 2005.

[10] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *ACM Conference on SIGCOMM*, 2011.

[11] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5, January 1998.

[12] Databricks. Spark survey results, 2015.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[14] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *ACM/IEEE Conference on Supercomputing*, 2006.

[15] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Object-oriented Programming, Systems, Languages, and Applications*, 2007.

[16] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *ACM Conference on SIGCOMM*, 2014.

[17] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Symposium on Operating Systems Principles*, 2009.

[18] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[19] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *ACM SIGMOD International Conference on Management of Data*, 2012.

[20] Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. Rex: Recursive, delta-based data-centric computation. *Proc. VLDB Endow.*, 5(11):1280–1291, July 2012.

[21] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles*, 2013.

[22] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed dataflow computing. In *Networked Systems Design and Implementation*, 2011.

[23] OmpSs. https://pm.bsc.es/ompss, Sep 2015.

[24] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In *Hot Topics in Operating Systems*, pages 14–14, 2013.

[25] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Symposium on Operating Systems Principles*, 2013.

[26] Nikolaos Papakonstantinou, Foivos S Zakkak, and Polyvios Pratikakis. Hierarchical parallel dynamic dependence analysis for recursively task-parallel programs. In *IEEE International Parallel and Distributed Processing Symposium*, 2016.

[27] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems*, 2013.

[28] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13):2110–2121, September 2015.

[29] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *ACM SIGMOD International Conference on Management of Data*, 2016.

[30] Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In *In KDD Workshop on Text Mining*, 2000.

[31] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *Operating Systems Design and Implementation*, October 2014.

[32] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines. *PVLDB*, 8(12):1542–1553, 2015.

[33] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph Gonzalez, and Randy H. Katz. Faster jobs in distributed data processing using multi-task learning. In *SDM*, pages 532–540. SIAM, 2015.

[34] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *European Conference on Computer Systems*, 2010.

[35] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Hot Topics in Cloud Computing*, 2010.

[36] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Operating Systems Design and Implementation*, 2008.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D4.3

Execution Engine v.2

WP 4 – Dependency-aware query execution engine

Nature: Report

Dissemination: Public