Offline Scheduling Algorithm for MapReduce

Mayuresh Kunjir and Ergys Ristani

Abstract
MapReduce is a popular distributed computing framework. Job scheduling on MapReduce poses unique challenge of utilizing all resources optimally while respecting precedence constraints and machine affinity constraints for all jobs. We try to formulate this problem as a mixed integer program for the offline case – when all the jobs arrive at time instance 0. Our schedule utilizes all resources optimally with data locality imposed for maximum possible tasks.

1 Introduction

Scheduling jobs optimally on parallel machines is a well studied problem in the operating systems community. The MapReduce [5] framework is a recent advancement in the area of data intensive parallel computing systems. It facilitates splitting computations across multiple nodes in a cluster. This raises a number of research challenges in optimizing the running time of the workload by scheduling jobs at suitable locations and at appropriate times. Unlike in operating systems and networks, scheduling policies in MapReduce is still a work in progress. We particularly target a scenario of repetitive workloads, where the same set of jobs is executed periodically. We try to find an optimal schedule using a linear program that considers a resource utilization profile of previously executed jobs.

Outline. We introduce the basics of the MapReduce framework in the following subsection which will be necessary to understand the problem. We present a motivation example also. In Section 2, we summarize the related work in this area. We then give our problem formulation in Section 3 as a mixed integer program. Our experimental analysis using this program is described in Section 4. The last two sections outline future work and conclusions.

1.1 MapReduce

MapReduce allows splitting a data processing job across a number of tasks. The tasks can be of two types: (a) map tasks or (b) reduce tasks. Each map task works on a split of data, where filtering-kind of operations are carried out. The reduce tasks are used to aggregate mappers’ output to produce the final result. The data transfer from mappers to reducers takes place in a phase called shuffle. Mappers partition their output in a number of splits equal to the number of reducers. Each reducer copies one split from every mapper during shuffle phase. Once all the data is copied, reducers start aggregating the data.
A point to note here is that the aggregation can’t start until all mappers have finished their execution.

Hadoop [2] is a popular implementation of the MapReduce framework. In Hadoop, a JobTracker (a special master node) is tasked with the responsibility of splitting a job into a number of map and reduce tasks and schedule them across the cluster. Nodes which run these tasks are called TaskTrackers. Every TaskTracker is capable of running certain map tasks and reduce tasks parallelly on dedicated resources, which we will call slots. For instance, if a TaskTracker has 2 map slots and 1 reduce slot, then it can run 2 map tasks and 1 reduce task concurrently in its dedicated slots. All the slots are required to be homogeneous in the current MapReduce implementation. Each map task accesses one block of data which may or may not be located at the corresponding TaskTracker machine. If multiple tasks are scheduled at a node, they get executed in a non-preemptive manner, the so called waves of execution.

Hadoop provides two schedulers for JobTracker – the Fair Share Scheduler and the Capacity Scheduler. Both take the current availability status of nodes (which is communicated by all the slave nodes to master by a heartbeat message) and takes the job priorities into account to find out a schedule using several heuristics. The schedulers try to use all the available resources optimally, but they can’t provide any optimization guarantees because of the following reasons:

1. Data might be skewed. That is, some tasks need to process more data than the others.
2. Some nodes might be slower than the others.
3. Some nodes might have to fetch their input over the network, while others get it locally.

These are not easy problems to solve. Data skew is unavoidable and we also can’t assume all the nodes to be homogeneous. But we can certainly improve on repetitive workloads where jobs are executed periodically. The idea is to profile the workload’s resource utilization over a period of time and then come up with an improved schedule for the same workload. We state the problem formally in the next section.

1.2 Motivating Example

To motivate the problem further, let’s consider an example workload of three jobs – ReplaceWords, WordCount and Sort. The ReplaceWords program reads a set of documents and replaces occurrences of a set of words with specified alternatives. In a MapReduce setting, this can be performed as a map-only job since there is no aggregation required here. The WordCount program also reads a set of documents but needs to count occurrences of all the words occurring in the documents. Each map task accesses a document and sends local word counts of that document across reducers. Reducers aggregate these local word counts to produce a global word count. The Sort program is a reduce-heavy job whose mappers just read data and partition it for reducers. Reducers have to do the actual sorting on the data. These jobs are typical use-cases of the MapReduce framework.
Figure 1: An example schedule for three jobs – ReplaceWords, WordCount and Sort

Figure 2: A better schedule for three jobs – ReplaceWords, WordCount and Sort

Figure 1 shows a schedule of these jobs. We have three map slots and one reduce slot available. ReplaceWords has 9 map tasks that get scheduled first. WordCount with 3 map tasks and 1 reduce task gets scheduled later. The 3 map tasks and 2 reduce tasks of Sort get scheduled last. It can be noticed that the reduce slot doesn’t start allocating any tasks in the beginning and towards the end, map slots sit idle when there are only reduce tasks running in the system. An alternate schedule is shown in Figure 2 where all the tasks of WordCount get scheduled first, Sort goes next and ReplaceWords in the end. This is in fact the optimal schedule given by our program. This schedule utilizes the slots better and thus reduces the workload completion time significantly.

2 Related Work

In this classic paper on scheduling [8], LP-based solutions for scheduling jobs to minimize total completion time on a single processor as well as multiple processors are provided. It provides a polynomial-time approximation algorithm for each of the LP formulations which guarantees reasonable performance. [4] uses the formulations suggested in [8] on MapReduce systems. However, it doesn’t consider precedence relations between map and reduce tasks and hence doesn’t capture the use-case of MapReduce correctly.

There has been a lot of work to develop on-line scheduling algorithm for the MapReduce framework. A good background of the current state of scheduling in Hadoop is provided in [11]. It stresses on the need of a new speculative task scheduler and comes up with a new scheduler called LATE. It uses Hadoop’s
estimates for remaining job execution time and speculatively runs jobs that are estimated to finish farthest in time. [10] builds on the same ideas and it considers rack locality among other things to come up with a better fair share scheduler for Hadoop. [9] is another online scheduling algorithm which runs in epochs. At every epoch, resource allocation for each job is changed based on its progress. It, however, doesn’t factor in data locality.

Perhaps the closest work to ours is [6]. It first identifies the effect of various parameters like data locality, slot-to-core ratio and the heartbeat interval on Hadoop and formulates scheduling as a constraint satisfaction problem. They require enhancements in the current MapReduce execution model though. Resource profiles are used in [7] to schedule for better utilization. It suggests usage of dedicated ‘job slots’ where tasks from a single job can be scheduled. This enables them to provide performance guarantees on jobs.

3 Problem Formulation

In a MapReduce setting, we are given a workload of jobs. For each job, its profile provides us the following information:

1. Number of map tasks
2. Number of reduce tasks
3. Processing time of each task
4. Machine affinity (data locality) of each task

A slot configuration for all the machines in the cluster is provided. The optimization goal for the scheduler in this setting can be either average job completion time or total workload completion time. This scheduling problem is \textit{NP-Complete}. Appendix A shows a reduction from the 3-PARTITION problem. In this section we give a mixed integer programming formulation for this problem. All the parameters are listed below.

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( J )</td>
</tr>
<tr>
<td>( \tilde{M} )</td>
</tr>
<tr>
<td>( T^M_i/R_i )</td>
</tr>
<tr>
<td>( P^M_i/R_i )</td>
</tr>
<tr>
<td>( S_k )</td>
</tr>
<tr>
<td>( A_{ijk} )</td>
</tr>
<tr>
<td>( O )</td>
</tr>
</tbody>
</table>

The list of variables is presented here:
Variables

- \(s_{M/R}^{i,j} \): start time for the map/reduce tasks \((i, j)\) \(i \in J, j \in T_i^{M/R}\)
- \(f_{M/R}^{i,j} \): finish time for the map/reduce tasks \((i, j)\) \(i \in J, j \in T_i^{M/R}\)
- \(x_{ijkl}^{M/R} \): true if map/reduce task \((i, j)\) \(i \in J, j \in T_i^{M/R}\) runs in slot \(l\) of machine \(k\)
- \(y_{ijkl}^{M/R} \): true if map/reduce task \((i_1, j_1)\) \(i_1 \in J, j_1 \in T_i^{M/R}\) starts in slot \(l\) of machine \(k\) before task \((i_2, j_2)\) \(i_2 \in J, j_2 \in T_i^{M/R}\), \((i_1, j_1) \neq (i_2, j_2)\)
- \(z_{ijkl}^M \): true if map task \((i, j)\) \(i \in J, j \in T_i^M\) runs in machine \(k \in \tilde{M}\)

The objective of our program minimizes the total makespan\(^1\), which is equal to minimizing the latest finish time among all tasks\(^2\):

\[
\min \max \{f_{M/R}^{i,j} \mid i \in J, j \in T_i^{M/R}\} \quad (1)
\]

The start times of map and reduce task should all be non-negative numbers:

\[
s_{M/R}^{i,j} \geq 0 \quad (2)
\]

The finish time of any reduce task should be at least its start time plus its processing time:

\[
f_{R}^{i,j} \geq s_{R}^{i,j} + P_{R}^{i,j} \quad (3)
\]

Similarly, the finish time of any map task should be at least its start time plus its processing time. However, due to the locality of data, a map task that doesn’t run where its data is located suffers from the overhead of data transfer. We are using a single number for overhead, but in reality overheads for intra-rack transfer and inter-rack transfers will differ considerably. The program can be generalized to account for this scenario.

\[
f_{M}^{i,j} \geq s_{M}^{i,j} + P_{M}^{i,j} + O(1 - \sum_k A_{ijkl} z_{ijkl}^{M}) \quad (4)
\]

MapReduce allows reduce tasks of a particular job to start as early as one of its map tasks has finished. However due to the non-linearity of such a constraint, we relax the constraint as follows: for any job, all its reduce tasks should start after all its map tasks have finished. As an effect, shuffle phase of MapReduce jobs is not considered in this formulation.

\[
\min \{s_{R}^{i} \mid j \in T_i^R\} \geq \max \{f_{M}^{i} \mid j \in T_i^M\} \quad (5)
\]

We also constrain each task to run only on one machine and only on one of its slots:

\[
\sum_{kl} x_{ijkl}^{M/R} = 1 \quad (6)
\]

\(^1\)Other objectives like average completion time or number of tardy jobs can be considered as well.

\(^2\)If we consider latest finish time of all reduce tasks only, map-only jobs will get scheduled in the end sub-optimally.
We impose a precedence constraint for any pair of tasks running in the same slot. We make this constraint linear as explained in [1]. Such a constraint enforces that for a pair of tasks, one should start after the other has finished:

\[ f_{i_2j_2}^M \leq s_{i_1j_1}^M \text{ or } f_{i_1j_1}^M \leq s_{i_2j_2}^M \]  

(7)

Jobs might have a precedence relation. Our formulation allows introducing such constraints as follows: If job \( i_1 \) preceds job \( i_2 \), none of its map tasks can start before all tasks of job \( i_2 \) have finished.

\[ \min \{ s_{i_1j}^M \mid j \in T_{i_1} \} \geq \max \{ f_{i_2j}^{M/R} \mid j \in T_{i_2}^{M/R} \} \]  

(8)

The mixed integer program is presented here:

\[
\begin{align*}
\min & \max \{ f_{ij}^{M/R} \mid i \in J, j \in T_i^{M/R} \} \\
\text{subject to} & \\
& s_{ij}^{M/R} \geq 0 \quad \forall i \in J, \forall j \in T_i^{M/R} \\
& f_{ij}^R \geq s_{ij}^R + P_{ij}^R \quad \forall i \in J, \forall j \in T_i^R \\
& f_{ij}^M \geq s_{ij}^M + P_{ij}^M + O(1 - \sum_k A_{ijk}z_{ijk}^M) \quad \forall i \in J, \forall j \in T_i^M \\
& z_{ijk}^M = \sum_{kl} x_{ijkl}^M \quad \forall i \in J, \forall j \in T_i^M, \forall k \in \tilde{M} \\
& \min \{ s_{ij}^R \mid j \in T_i^R \} \geq \max \{ f_{ij}^M \mid j \in T_i^M \} \quad \forall i \in J \\
& \sum_{kl} x_{ijkl}^{M/R} = 1 \quad \forall i \in J, \forall j \in T_i^{M/R} \\
& f_{i_2j_2}^M \leq s_{i_1j_1}^M \text{ or } f_{i_1j_1}^M \leq s_{i_2j_2}^M \quad \forall i_1, i_2 \in J, j_1 \in T_{i_1}^M, j_2 \in T_{i_2}^M
\end{align*}
\]

4 Experimental Analysis

We have conducted a set of experiments using the Gurobi solver for our randomly generated scenarios. We have generated and tested our scenarios by picking parameters from the ranges described in Table 1.

In Figure 3 we show an optimal scheduling policy found using our formulation. Four jobs were scheduled in 3 machines. Each machine had 2 map and 1 reduce slots. Job 1 had 6 map tasks and 3 reduce tasks, while the others had 3,1; 3,3; 3,1 map and reduce tasks respectively. The lengths of the jobs appear in the image.

Typically a cluster has about hundreds of nodes and jobs have tasks in the order of thousands. With our current formulation we haven’t been able to scale to such values. We would like to work on it in the future.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jobs</td>
<td>4.5</td>
</tr>
<tr>
<td>Machines</td>
<td>2.3</td>
</tr>
<tr>
<td>Map execution time</td>
<td>2.20</td>
</tr>
<tr>
<td>Reduce execution time</td>
<td>2.10</td>
</tr>
<tr>
<td>Map tasks per job</td>
<td>3.7</td>
</tr>
<tr>
<td>Reduce tasks per job</td>
<td>0.3</td>
</tr>
<tr>
<td>Map slots per machine</td>
<td>0.3</td>
</tr>
<tr>
<td>Reduce slots per machine</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 1: Experiment parameters

![Figure 3: Optimal schedule found using the mixed integer programming formulation. Blocks represent tasks and numbers on them denote task lengths.](image)

5 Future Work

So far in our work we haven’t done any comparison study between our solution and the other approaches used in the literature. Our schedules are guaranteed to use the resources optimally. We also ensure that most of the map tasks process local data by considering overheads in transferring data over the network. It would be interesting to quantify the benefits we provide over baseline approaches used in practice. We would also like to carry out these experiments on real workloads.

Our solution looks at the offline scheduling problem currently, but most of the workloads are ad-hoc in nature. We should be able to adapt for changing workloads. It would be an interesting direction to combine off-line scheduling with heuristics to serve all kinds of jobs on MapReduce.
Another interesting challenge is posed by MapReduce Next Generation [3] which is currently under development. It allows resources to be split heterogeneously among the tasks. In contrast, the current version of MapReduce requires all slots to be homogeneous. Under these new settings, an optimal schedule needs to specify the slot configuration as well. An optimal schedule for a homogeneous slot configuration may not be the best possible schedule if we have the flexibility to change the resource allotment of slots. We would further like to tackle the scheduling problem on these settings.

6 Conclusion

The main contributions of this project are the following:

1. We formulated offline scheduling problem for workload completion time minimization on the MapReduce framework as a mixed integer program.

2. Our scheduler considers job locality and precedence constraints and provides a schedule that utilizes all resources optimally.

We would like to use this work as a strong base to tackle scheduling problems on the next generation of MapReduce. It is part of a larger vision of building an end-to-end system that includes job profiling, better scheduling and query optimization for the new MapReduce framework.

References


A Complexity of MapReduce Scheduling

In this section, we prove that scheduling on MapReduce framework is a NP-Complete problem. [12] gives a proof from 3-PARTITION problem. The scheduling problem the proof uses considers a case of \( N \) slots which can be used by mappers or reducers identically. But in reality, we have dedicated map slots and reduce slots and they can't be used interchangeably. We consider the latter case in our proof. The idea remains the same though. We first define a 3-PARTITION problem instance and then the scheduling problem instance.

3-PARTITION

Given a set \( A \) of \( 3m \) elements \( a_i \in \mathbb{Z}^+ \), \( i = 1, 2, \ldots, 3m \), and a number \( N \in \mathbb{Z}^+ \) s.t. \( N/4 < a_i < N/2, \forall i \) and \( \sum a_i = mN \). The decision version of the problem is to find if \( A \) can be partitioned into \( m \) disjoint sets \( A_1, A_2, \ldots, A_m \) such that \( \sum_{a_i \in A_k} a_i = N, \forall k \). Note that each partition will have exactly 3 elements because of bounds on \( a_i \)s.

Scheduling

We are given \( N \) map slots and \( N \) reduce slots. We assume \( N \equiv 0 \) without loss of generality. We are given \( 3m \) jobs, \( J_1, J_2, \ldots, J_{3m} \), to schedule, all arriving at time 0. The map tasks and reduce tasks for each task are given as \( M_i = a_i \), \( R_i = \frac{2}{3}N - a_i \), \( \forall i \). Furthermore, let \( \omega = \frac{3(2Tm^2 - T^2m^2 + Tm + m)}{2}, T \in \mathbb{Z}^+ \). The problem is to decide if there is a feasible schedule with a flow time no more than \( \omega \).

Flow Time

Let \( F(S) \) denote flow time of schedule \( S \). Let \( n^S_k \) denote the number of jobs that finish in the \( k \)-th time slot in schedule \( S \). Then the flow time \( F(S) = \sum_{k=1}^{\infty} k n^S_k \). We drop the superscript in the following. It is clear that \( n_1 = 0 \). Let \( N_k = \sum_{j=1}^{\infty} n_j \) denote the number of jobs finished by time \( k \).

Lemma 1. \( N_k \leq 3(k-1) \) and the upper bound is attained only if \( n_j = 3, \forall j \in 2, 3, \ldots, k \).

Proof: The inequality can easily be proved by contradiction. For \( N_k \) to be greater than \( 3(k-1) \), we need to be able to finish reduce processing of all these jobs. But we can’t start any reduce tasks at time instance 0 and therefore won’t be able to serve more than \( 3(k-1) \) reduce tasks in the given time period.

To show when we can achieve the upper bound, let’s use induction. When \( m = 1 \), we can schedule all map tasks of the three jobs at time instance 1 and reduce tasks for them at time instance 2 as they require \( N \) map and \( N \) reduce tasks collectively by definition. So we can finish all three jobs by time 2. Now let’s assume that we have \( n_j = 3, \forall j \in 2, 3, \ldots, t-1 \) or \( N_{t-1} = 3(t-2) \). If we schedule map tasks of 3 new jobs completely at time instance \( t-1 \), we can finish their reduce tasks in time instance \( t \). So total number of jobs finished by time \( t \) will be \( N_t = 3(t-1) \). So this proves that scheduling 3 jobs every instance allows us to achieve an upper bound on the total jobs we can finish.

Corollary 1. \( \sum_{k=1}^{Tm} N_k \leq \frac{3Tm(Tm-1)}{2} \) and the upper bound is attained only if \( N_k = 3(k-1), \forall k \in 1, 2, \ldots, Tm - 1 \).
Lemma 2. $F(S) \geq \frac{3(2Tm^2 - T^2m^2 + Tm + m)}{2}$ and the lower bound is attained only if Corollary 1 holds and $n_k = 0, \forall k > Tm$.

Proof: By definition of flow time,

$$F(S) = \sum_{k=1}^{\infty} kn_k$$

$$= \sum_{k=1}^{\infty} [(Tm + 1) - (Tm + 1 - k)]n_k$$

$$\geq \sum_{k=1}^{\infty} (Tm + 1)n_k - \sum_{k=1}^{Tm}(Tm + 1 - k)n_k$$

$$= (Tm + 1)3m - \sum_{k=1}^{Tm} N_k$$

$$\geq (Tm + 1)3m - \frac{3Tm(Tm - 1)}{2} \ldots \text{Corollary 1}$$

$$= \frac{3(2Tm^2 - T^2m^2 + Tm + m)}{2}$$

From the derivation, it can be observed that the lower bound can be attained only if $n_k = 0, \forall k > Tm$ and $\sum_{k=1}^{Tm} N_k = \frac{3Tm(Tm - 1)}{2}$.

Theorem 1. The MapReduce scheduling problem is NP-Complete.

Proof: Given a schedule, it’s easy to verify in polynomial time whether it is feasible and also it’s flow time. So MapReduce scheduling problem is clearly in NP. We reduce 3-PARTITION to scheduling using the above mapping. If scheduling problem gives a feasible schedule with flow time within the bound, using Lemma 1 and Lemma 2, we can say that 3 jobs get scheduled at each time instance. The number of map tasks for each of these 3 jobs can be considered as a partition for 3-PARTITION instance. In this way, if we take three jobs at every time instance, we get a solution to our 3-PARTITION instance. Since 3-PARTITION is known to be strongly NP-Complete, this reduction proves that MapReduce scheduling is also a NP-Complete problem.