Job Scheduling with Minimizing Data Communication Costs

[Extended Abstract]

Trevor Clinkenbeard*
David R. Cheriton School of Computer Science
Faculty of Mathematics, University of Waterloo
tclinken@uwaterloo.ca

Anisoara Nica
SAP SE
Waterloo, Ontario, Canada
Anisoara.Nica@sap.com

ABSTRACT

The research presented in this paper analyzes different algorithms for scheduling a set of potentially interdependent jobs in order to minimize the total runtime, or makespan, when data communication costs are considered. On distributed file systems, such as the HDFS, files are spread across a cluster of machines. Once a request, such as a query, having as input the data in these files is translated into a set of jobs, these jobs must be scheduled across machines in the cluster. Jobs consume input files stored in the distributed file system or in cluster nodes, and produce output which is potentially consumed by future jobs. If a job needs a particular file as input, the job must either be executed on the same machine, or it must incur a time penalty to copy the file, increasing latency for the job. At the same time, independent jobs are ideally scheduled at the same time on different machines, in order to take advantage of parallelism. Both minimizing communication costs and maximizing parallelism serve to minimize the total runtime of a set of jobs. Furthermore, the problem gets more complex when certain jobs must wait for previous jobs to provide input, as is frequently the case when a set of jobs represents the steps of a query on a distributed database.

Keywords

Distributed databases; Job scheduling

1. INTRODUCTION

Previous research has studied specific cases of minimizing data communication costs. Golab et al. [2] used the theory of graph partitioning to find a near-optimal algorithm to place files in a distributed file system in order to minimize data communication cost given an a known workload. In contrast, the goal of this paper is not merely to minimize data communication cost, but to consider the cost of data communication in order to minimize the total runtime of a set of dynamic jobs. We also assume that input files may already reside in some cluster nodes, and our solutions attempt to schedule new jobs considering the current state of distributed filesystem.

Apache Spark is a popular distributed processing paradigm which implements the efficient Sparrow scheduler [4]. The Sparrow scheduler is designed to efficiently schedule jobs with processing times on the order of 100ms. Thus, Sparrow relies on randomness and unsophisticated heuristics, in order to schedule jobs in real time with very low latency. In contrast, here we aim to design a dynamic scheduler for a set of heterogeneous, including both large and small, jobs. Our algorithms take significantly longer to schedule jobs, but produce more optimal scheduling, a trade-off which favors scheduling jobs with relatively long processing times.

The inputs to our problem are a set $\mathcal{M}$ of machines, a set $\mathcal{J}$ of atomic jobs, and a set $\mathcal{D}$ of data files. Each job $j_i \in \mathcal{J}$ has an estimated processing time r_i, a set of input files $I_i \subset \mathcal{D}$, and an output file $o_i \in \mathcal{D}$. Each file $d_i \in \mathcal{D}$ has a (possibly estimated) size t_i. Some jobs in $\mathcal{J}$ are already scheduled on a machine in $\mathcal{M}$. Let the set of unscheduled jobs be $u\mathcal{J}$ (e.g., the set of jobs of currently unscheduled requests). Our goal is to output a function $m : u\mathcal{J} \rightarrow \mathcal{M}$, mapping each unscheduled job to a machine on which it is to be executed. In the case where jobs are interdependent, i.e. $o_i \in I_k$ for some jobs $j_i, j_k \in \mathcal{J}$, we must also produce an order in which to process jobs on each machine. The goal of the output is to minimize total runtime.

The first step in developing our scheduling algorithms was to interpret the set of jobs, files, and dependencies as a directed acyclical graph. Vertices represent files and jobs, while dependencies are represented by arcs. All jobs depend on their input files, and all output files depend on the jobs which produce them. A job's processing time, r_i, provides the weight of the arc to the job's output file. The file sizes, t_i, along with the average network communication speed, give potential weights for the rest of the arcs (these weights are only assigned if the input file is not already stored on the same machine as the job is executed on).

Based on this general outline for a graph, the first algorithm implemented was a greedy algorithm based on the well-known critical path method [3]. Using the arc weights assigned, repeatedly select the job with the longest weighted chain of dependents (i.e. the job at the head of the critical path). Greedily schedule this job on a machine which minimizes the current makespan and ideally balances the load

*The student author and the adviser Dr. Anisoara Nica worked on this project during Fall 2014 co-op term at SAP Waterloo, Canada.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2764943.

between different machines. Repeat this process until all jobs are scheduled.

This algorithm is much more efficient than arbitrary or random scheduling algorithms in all our experiments. However, it is very greedy, and does not fully consider the structure of the directed acyclical graph. Other algorithms explored used linear programming, implemented using SAP SciPlex software.

In the linear programming algorithms, variables are used to represent the projected finish time of each machine. The obvious objective is to minimize the maximum of these machine finish times, the makespan. This works optimally for scheduling just one set of jobs. However, if we wish to schedule more jobs in the future, it is also important to balance the load across machines. For example, consider a multinode cluster in which a large number of jobs are already scheduled to run on machine m. Now assume we wish to schedule a new, smaller, set of independent jobs. If the only objective is to minimize makespan, these jobs can be scheduled anywhere on the cluster except on machine m, since the makespan depends only on m's finish time. However, we would prefer to balance these jobs effectively across the other machines which cannot be achieved if only makespan is minimized. Thus, we experimented with different objective functions, such as minimizing a weighted sum of the makespan and some other linear functions based on the load balance. Unfortunately, we are restricted to using linear functions. The greedy algorithm has no such restrictions, so it can use a more effective load balancing objective function.

The first linear programming algorithm implemented ignored the possibility of inter-job dependencies. In this case, the ordering of jobs on a particular machine is irrelevant. By using boolean variables x_{ik} representing job i being scheduled on machine k, and introducing the appropriate constraints, it is possible to find a completely optimal solution.

Introducing inter-job dependencies complicates the problem significantly. It is no longer feasible to find a completely optimal solution. Instead, we split the graph of jobs into "layers." We repeatedly select a set S of unscheduled jobs which don't depend on any other unscheduled jobs. These jobs can be scheduled in a nearly optimal way using linear programming, once constraints are added to ensure that a new job cannot start on a particular machine until that machine is finished executing the jobs already scheduled there.

While breaking the jobs into "layers" greatly simplifies the problem, and the above algorithm performs well in most cases, inter-job dependencies introduce an important technicality. A machine may be starved as its next scheduled job waits for data from another machine. However, in the meantime, the machine can load other data necessary for its waiting jobs. This is difficult to represent using linear programming, because we had previously been adding all load times to the runtime of a job. With this consideration in mind, though, a job could potentially load an input file from another machine, without causing any additional delay to later jobs scheduled on the machine. By adding additional constraints to find the order of jobs on a machine, this problem can be avoided with a more expensive but slightly more optimal linear programming solution.

2. RESULTS AND DISCUSSIONS

To test the algorithms, nine machines were used. 75 files of size 100MB were generated and placed on the HDFS initially.

Table 1: Average Total Runtimes (in seconds): (1) first n_1 jobs are randomly scheduled; (2) n_2 jobs are scheduled with one of the scheduling algorithms *Random*, *Greedy*, and *Linear Programming* given the state of the cluster having n_1 jobs already scheduled.

n_1	n_2	*Random*	*Greedy*	*Linear Programming*
0	20	362.6	162.0	167.5
0	40	415.1	270.3	275.6
20	20	421.2	254.8	327.2

A set of random jobs and dependencies were randomly generated, using random graph generation techniques inspired by Cordeiro et al. [1]. Jobs are created with a randomly assigned constant processing time, uniformly distributed between 30 and 60 seconds. Each job produces a 100MB output file. On average, a given job consumes a given file with probability 0.1, but jobs are ordered to prevent any cyclical dependencies.

Table 1 contains preliminary testing results, giving the average total runtime in seconds. The first n_1 jobs are randomly scheduled, then an additional n_2 jobs are scheduled, considering the state of the cluster with the already scheduled n_1 jobs. In general, the linear programming algorithm and the greedy algorithm perform similarly for smaller sets of jobs. Both algorithms are much better than a random scheduling. The greedy algorithm is better for load balancing, though, as seen in the case where half of the jobs are already scheduled arbitrarily. These scheduled jobs are likely to be scheduled unevenly across machines, so the greedy algorithm is favored due to its load balancing capabilities.

In the future, we plan to run more extensive tests on larger sets of jobs. The linear programming algorithms are promising, in that they can be less greedy than the other algorithm tested. However, utilizing linear programming to effectively balance jobs across many machines is still an open problem. Lastly, certain real life complications were ignored in this study. In practice, machines have memory constraints, and they can get rid of unused files to free up space. Furthermore, we have made the assumption that each job has constant processing time, regardless of the machine it is run on. In reality, some machines are faster than others, so it is not always this simple. These issues further complicate the scheduling problem, and can be the subject of further research.

3. REFERENCES

[1] D. Cordeiro, G. Mounie, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. Random graph generation for scheduling simulations. In *SIMUTools*, Torremolinos, Malaga, Spain, March 2010.

[2] L. Golab, M. Hadjieleftheriou, H. Karloff, and B. Saha. Distributed data placement to minimize communication costs via graph partitioning. In *SSDBM*, Aalborg, Denmark, June 2014.

[3] J. Kelley and M. Walker. Critical-path planning and scheduling. In *Proceedings of the Eastern Joint Computer Conference*, 1959.

[4] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Scalable scheduling for sub-second parallel jobs. Technical Report UCB/EECS-2013-29, U.C. Berkeley, April 2013.