Packing the Most Onto Your Cloud

Ashraf Aboulnaga Ziyu Wang Zi Ye Zhang

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{ashraf, z8wang, zy2zhang}@cs.uwaterloo.ca

ABSTRACT

Parallel dataflow programming frameworks such as Map-Reduce are increasingly being used for large scale data analysis on computing clouds. It is therefore becoming important to automatically optimize the performance of these frameworks. In this paper, we deal with one particular optimization problem, namely scheduling sets of Map-Reduce jobs on a cluster of machines. We present a scheduler that takes job characteristics into account and finds a schedule that minimizes the total completion time of the set of jobs. Our scheduler decides on the number of machines to assign to each job, and it tries to pack as many jobs on the machines as the machine resources can support. To enable flexible assignment of jobs onto machines, we run the Map-Reduce jobs in virtual machines. Our scheduling problem is formulated as a constrained optimization problem, and we experimentally demonstrate using the Hadoop open source Map-Reduce implementation that the solution to this problem results in benefits up to 30%.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Design, Experimentation, Management, Performance

1. INTRODUCTION

Cloud computing makes massive clusters of machines available for solving large computation and data analysis problems. A key challenge in harnessing the computing power of these clusters is how to write and execute distributed programs at such a massive scale. To address this challenge, parallel dataflow programming frameworks that automatically deal with problems of distribution have been developed, a very popular one being Map-Reduce. In the Map-Reduce framework, several tuning and optimization decisions need to be made. For example, deciding on the number of machines to use for a given job, or deciding how many of these machines will apply the *reduce* function. As Map-Reduce gets adopted more widely for data analysis tasks, it will become increasingly important to optimize the execution of Map-Reduce jobs automatically.

CloudDB'09, November 2, 2009, Hong Kong, China.

In this paper, we focus on a specific type of optimization: scheduling and resource allocation for sets of Map-Reduce jobs. Given a set of Map-Reduce jobs to be executed on a cluster of machines, our goal is to schedule the jobs on the machines to minimize the jobs' total completion time. In achieving this goal, we also achieve a secondary goal of improving the CPU and I/O utilization of the cluster. This is important because if resource utilization is low, the massive scale of the cloud could lead to a massive waste of resources. For example, a computation using 1000 machines at 40% utilization is effectively wasting the resources of 600 machines.

In designing our scheduler, we rely on two observations. First, we observe that executing the jobs sequentially on the full cluster does not result in the best schedule. This strategy minimizes the sum of the execution times of the jobs, but it does not minimize the total time required for the whole set of jobs (which also includes wait times), and it may result in poor utilization of the cluster resources. A better strategy is to allow different jobs to execute concurrently on parts of the cluster. Our second observation is that different Map-Reduce jobs have different resource (CPU and I/O) requirements. By understanding these resource requirements we can pack more jobs onto cluster nodes, further minimizing total completion time and improving resource utilization. We refer to this as *overcommitting* the resources of the cluster nodes.

To facilitate flexible scheduling and simplify the overcommitting of resources, we run the Map-Reduce nodes in *virtual machines*. Thus, our scheduling problem becomes one of scheduling the virtual machines of the different jobs on the physical machines of the cluster. We believe that using virtual machines for the Map-Reduce nodes has several advantages, and we have found that this decision has made the implementation of our scheduler much simpler. However, we stress that our scheduling algorithm can be implemented in any scheduler and does not depend on using virtual machines.

We formulate the scheduling problem as a constrained optimization problem that our scheduler solves. The optimization problem requires mathematical models of the performance of different jobs. We construct these performance models by running the jobs on different numbers of cluster nodes and fitting a regression model to the observed performance, extrapolating from the observed cluster sizes to unobserved cluster sizes. Such experiment-driven performance modeling is gaining wide acceptance as a way to build robust performance models for complex systems (e.g., [2]). For this experiment-driven modeling approach to be practical, we assume that each job will be run multiple times by users of the cluster, so that the cost of experiment-driven model building can be amortized over a large number of runs. Different runs can have different parameters and input data sets, but we assume that the job characteristics (e.g., size of the input data) will not vary significantly between runs, so that the performance models remain valid across runs.

When our scheduler decides to run a job on a set of cluster nodes, it assumes that the input data for this job should also be placed on these nodes to avoid the overhead of shipping the data to the computation. There are situations in which the data placement is fixed and cannot be changed, so this assumption will not hold. Extending our scheduler to deal with such rigid data placements is a subject of future work. For this paper, we assume that we can place the data on the same nodes as the computation.

In this paper, we use the Hadoop open source Java implementation of Map-Reduce [1]. Scheduling and resource allocation for Hadoop is becoming an active area of research, because the effectiveness of a Hadoop cluster and the efficiency with which it uses computing resources is directly linked to the effectiveness of the Hadoop scheduler (e.g, [5]). To the best of our knowledge, no Hadoop scheduler that minimizes the total completion time of a set of jobs has been proposed. Total completion time is an important metric for users of a throughput-oriented system like Hadoop, so scheduling based on this metric is of great benefit.

2. CLUSTERS OF VIRTUAL MACHINES

For our scheduler implementation we chose to run each Hadoop node in a virtual machine (VM), and we use Xen as the virtualization environment. Our scheduler decides the number of VMs to allocate to each Hadoop job, which jobs to run with each other, and how to allocate the VMs to the physical machines of the cluster. The set of VMs for each job runs a separate instance of the Hadoop Distributed File System (HDFS), and one VM for every job serves as the Hadoop master node for this job. The data for each job is stored in the HDFS instance running in the VMs of this job.

Using virtual machines for executing sets of Hadoop jobs simplifies the deployment of these jobs onto the cluster. Deployment consists of copying a VM image pre-loaded with Hadoop (i.e., a Hadoop *virtual appliance*) onto the different physical machines of the cluster and giving each VM a separate "identity" (e.g., separate hostname and IP address). Another advantage of using virtual machines is that the mapping of jobs to physical machines is flexible. To share a physical machine among multiple Hadoop jobs, we simply run the VMs of these jobs together on the physical machine. The virtual machine monitor partitions the resources of the physical machine among the VMs, and we have found that it adds very little overhead. One area where virtualization is not effective is if multiple VMs share the same physical disk. In this case, the virtual machine monitor is not able to guarantee performance isolation, and the disk activities of the different VMs interfere with each other. Our scheduler avoids these negative effects by ensuring that each VM has its own physical disk.

3. JOB CHARACTERISTICS

Our scheduler relies on two observations about Hadoop jobs. The first observation is that the benefit of allocating more nodes to a Hadoop job starts diminishing at some point as the number of nodes increases. To illustrate this, we use an example with four different Hadoop jobs: sorting a large number of records (job s-srt in our experiments), k-means clustering of records (job s-kmns), building an inverted index for Wikipedia (job s-inv), and computing pairwise document similarity over Wikipedia (job s-dsim). Figure 1 shows the execution time of these jobs for different numbers of Hadoop nodes. It is clear that the benefit of extra nodes decreases as the number of nodes increases. Thus, while allocating more nodes to a job always reduces its run time, partitioning the nodes among jobs and running them concurrently may reduce total completion time.

The second observation is that different jobs have different resource (I/O and CPU) intensities. Our scheduler tries to take advantage of the different resource characteristics of different jobs by packing jobs onto cluster nodes to maximize the resource utilization of these nodes. In our implementation, this is done by running the VMs of these jobs together on the same nodes. In our scheduler, we characterize jobs as *CPU intensive* or *I/O intensive*. We consider a job to be CPU intensive if its CPU consumption is above 60% when running on 6 VMs, and we consider it to be I/O intensive if its I/O consumption is above 100 MB/s when running on 6 VMs. We use this simple classification mechanism for a simple decision: whether or not we can schedule two jobs on the same physical machine. Our scheduler will try to schedule CPU intensive jobs on the same machines as I/O intensive jobs since the latter do not consume too much CPU. For this decision, we only need a coarse-grained classification of jobs and our simple classification mechanism is sufficiently accurate.

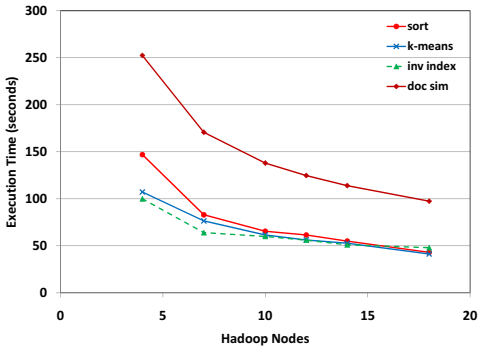

Figure 1: Execution Time of Different Jobs

Our Hadoop VM image has one virtual disk, and we assign at most one I/O intensive VM (with its one virtual disk) to each physical disk on our machines. Thus, a physical machine with k disks could support up to k I/O intensive VMs. The number of virtual CPUs in the virtual appliance is configured so that the physical CPUs (or cores) of the machine are evenly divide among the k VMs. Since I/O intensive VMs have CPU capacity to spare, it may be possible to reduce the completion time of the of jobs by scheduling extra CPU intensive VMs onto physical machines that are running I/O intensive VMs. We call this *overcommitting* the CPU resources of the physical machine. In our scheduler, we can schedule one "extra" CPU intensive VM for every r "primary" (possibly I/O intensive) VMs that run on a machine ($r \leq k$). In our experiments we use physical machines with two disks each, a very typical configuration ($k = 2$). We set $r = k = 2$ which means that we can have one extra VM per physical machine.

4. MODELING JOB PERFORMANCE

To make scheduling decisions, we need a model of performance for each job that provides an estimate of the execution time of the job given the number of nodes assigned to it. A key design decision we made to reduce the cost of modeling was to base our scheduler on models of performance for *individual jobs*, even though we use these models to make scheduling decisions for *groups* of jobs. We propose using an experiment-driven methodology for modeling performance. In this paper, we use the following simple methodology: For each job, we run the job on the expected data set size several times, varying the number of nodes in each run. In these runs, the job is executed alone on the cluster, and we measure its execution time. We fit a *power regression* formula to the observed points, obtaining a function of the form $F_i(n) = a * n^b$, where n is the number of nodes used by the job, $F_i(n)$ is the estimated execution time of job i on n nodes, and a and b are regression constants.

When we schedule up to k VMs on a physical machine, we assume that these VMs do not interfere with each other. This assumption was validated in our experiments. However, when we overcommit the CPU of a physical machine, the run time of all VMs scheduled on this machine increases. To model this increase, we multiply the execution time of all jobs scheduled on such a machine by a constant penalty factor α. We have found a value of $\alpha = 1.3$ to work well in our experiments.

The power regression model we propose here is one of many possible statistical models. We chose it because it is simple and has a closed form equation that is easy to fit, it matches the behavior of Hadoop job performance as the number of nodes varies, and it was shown to be accurate in our experiments and to produce good schedules. However, there are other possible statistical models, and our scheduler can work with any of them since the performance model is a pluggable component of the scheduler.

5. SCHEDULING ALGORITHMS

Our scheduler is given a set of M Hadoop jobs. Each job may be characterized as CPU intensive or I/O intensive. For each job i, we are given an experimentally determined performance model $F_i(n)$. We are also given the number of physical machines in the cluster P, the number of disks per physical machine k, and the number of primary VMs per extra VM r. We assume that the machines in the cluster are identical. The number of VMs we could schedule in this case without overcommitting is $N = P \times k$.

Our scheduler determines a schedule consisting of a specification of which jobs to run together, how many Hadoop VMs to give to each job, and which jobs should share the same physical machines (including overcommitting). We have developed two scheduling algorithms. The first is limited to running pairs of jobs together on the cluster at any time. This makes the problem formulation and solution simpler, but is limited in its flexibility. The second algorithm can schedule any number of jobs together on the cluster, so it is more flexible but more complex.

5.1 Pair Scheduling Algorithm

When scheduling two jobs i and j on the available nodes, the best achievable completion time, C_{ij} is the minimum of:

- $s_{ij} = F_i(N) + F_j(N)$, the total time of running the two jobs sequentially, giving all nodes to each job.
- c_{ij}, the run time of the two jobs running concurrently on the cluster without overcommitting. In this case, we give x nodes to job i and $N - x$ nodes to job j. The value of x can be determined by finding the root of the equation $F_i(x) - F_j(N - x) = 0$ using any root finding method.
- o_{ij}, the run time of the two jobs running concurrently on the cluster with overcommitting. This option only makes sense if one of the jobs is I/O intensive and the other is CPU intensive. Let i be the I/O intensive job. In this case, the estimated completion time is $\max\left(F_i(N), F_j(P \times \frac{k}{r})\right) \times \alpha$.

To schedule M Hadoop jobs using the pair scheduling algorithm, we determine C_{ij} for every pair of jobs, and we exhaustively search through all possible pairings of jobs to determine the schedule that minimizes the completion time. For the problem sizes we have tried, the exhaustive search was fast enough. However, for larger problem sizes we could formulate the problem as a binary integer programming problem that can be solved to determine the schedule.

5.2 Scheduling Multiple jobs

To be able to schedule more than two jobs concurrently on the cluster, we define our scheduling problem as a constrained non-linear optimization problem. Given M jobs, we define the schedule as a sequence of M *phases* where we run one or more jobs in each phase. A simple schedule in which the jobs are run sequentially will require all M phases, but if we schedule jobs concurrently, then some of the phases will be empty. Our scheduler gets rid of empty phases in a post-processing pass. The inputs to the optimization problem solved by the scheduler are as follows:

- An $M \times 1$ vector, C, where C_j is 1 if job j is CPU intensive and 0 if it is not.
- An $M \times 1$ vector, O, where O_j is 1 if job j is I/O intensive and 0 if it is not.
- The M functions $F_1(n), F_2(n), \ldots, F_M(n)$ that model the performance of the jobs. We define $F_j(0) = 0$ for all j.
- The number of physical machines P, the number of disks per physical machine k, and the number of primary VMs per extra VM r. Each physical machine can support k VMs without overcommitting, for a total of $N = P \times k$ VMs on the cluster.

The output of the scheduler is an execution schedule described by the following three output variables:

- An $M \times M$ matrix, A, where an element $0 \leq A_{ij} \leq k$ specifies that job j will run in phase i on A_{ij} VMs (rows specify phases and columns specify jobs).
- An $M \times 1$ vector, B, where B_j is 1 if job j is I/O intensive and will run with an extra VM on the same physical machine (used for specifying overcommitting).
- An $M \times 1$ vector, B', where B'_j is 1 if job j is CPU intensive and will be run in an extra VM with some I/O intensive job that allows it (used for specifying overcommitting).

The objective function that our scheduler minimizes is the completion time of the full schedule, which is defined as the sum of the maximum job execution times for each phase (the critical path). The objective function is defined as:

$$\mathcal{F} = \sum_{i=1}^{M} \max_{1 \leq j \leq M} \left[F_i(A_{ij}(1 - B_j - B'_j)) + \alpha F_i(A_{ij}(B_j + B'_j)\right]$$

The solution (A, B, and B') must satisfy the following constraints. In these constraints, I is an $M \times 1$ identity vector consisting of 1's, and 0 is an $M \times 1$ zero vector or the scalar 0.

- In each phase we should schedule at most N primary virtual machines: $A(I - B') \leq NI$.
- Every job is scheduled in at least one phase: $A^T I > 0$.
- Every extra (overcommitted) CPU intensive job is scheduled with r I/O intensive jobs that allow overcommitting: $A(B - rB') \geq 0$.
- Only I/O intensive jobs can be run with overcommitted CPU jobs: $O^T B = 0$.
- Only CPU intensive jobs are run as overcommitted jobs: $C^T B' = 0$.

After finding the optimal A, B, and B', the jobs to be run in phase i (row i of matrix A) are scheduled as follows: (1) Schedule I/O intensive jobs ($O_j = 1$) in a round robin manner on the primary VMs. (2) Schedule overcommitted CPU intensive jobs ($C_j = 1$ and $B' = 1$) on extra VMs with I/O intensive jobs that allow that ($B = 1$). (3) Schedule the remaining jobs on primary VMs.

This problem formulation affords us the most flexibility in schedul-

Workload	Jobs
W1	2 × s-srt + 2 × s-kmns + 1 × s-inv
W2	2 × s-srt + 1 × s-kmns + 1 × s-inv + 1 × s-dsim
W3	3 × s-srt + 2 × s-kmns + 1 × s-inv + 1 × s-dsim
W4	1 × s-srt + 1 × b-srt + 2 × b-inv
W5	2 × b-srt + 2 × b-kmns + 1 × b-inv
W6	2 × b-srt + 2 × b-kmns + 2 × b-inv

Table 1: Complex Workloads Used

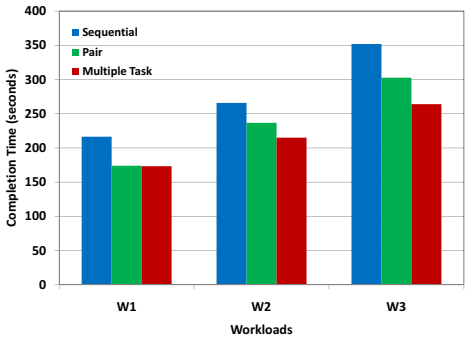

Figure 2: Scheduling for W1–W3

ing, but it is a highly complex non-linear constrained optimization problem. We implemented an integer programming solver for this problem that uses particle swarm optimization [4] and we use it in our experiments with 1500 particles and 200 iterations.

6. EXPERIMENTAL EVALUATION

Our experiments use a cluster of machines mounted in the same rack and connected via a 1GB Ethernet switch. Each machine is a Sun Fire X4100 server, with two 2.2GHz dual core AMD Opteron Model 275 x64 processors, 8GB memory, and two 73GB 10K RPM SCSI disks, running SUSE Linux 10.1 with Xen 3.1. For different experiments, we use different numbers of physical machines P, up to $P = 11$. Since every machine has two disks ($k = 2$), we can run up to $N = 22$ primary Hadoop VMs. We give each VM two virtual CPUs, and we allow one extra VM per physical machine ($r = 2$). We use the following jobs in our workloads:

- **sort:** A Hadoop implementation of sorting. We use two variants of sort: (1) a small sort that sorts 500,000 records of 5000 characters each, with a random integer sort key, for a total size of 2.4GB (we call this job s-srt), and (2) a big sort that sorts 21,000,000 records of 1000 characters each, with a random integer sort key, for a total size of 20GB (we call this job b-srt).
- **k-means:** k-means clustering of synthetically generated data. We use two variants: (1) a small variant in which the input data consists of 2000 clusters of 10,000 points each, distributed according to a Gaussian distribution around 0, 1, ..., 1999, for a total size of 353MB (we call this job s-kmns), and (2) a big variant in which the input data consists of 100 clusters of 5,720,000 points each, distributed according to a Gaussian distribution around 0, 1, ..., 99, for a total size of 10GB (b-kmns).
- **inverted index:** Building an inverted index for the XML dump of Wikipedia. We run this job on the first 32MB of Wikipedia data (s-inv) and on the first 5GB of Wikipedia data (b-inv).
- **document similarity:** Computing document similarity for all pairs of Wikipedia documents using the algorithm described in [3]. The input to this algorithm is the inverted index constructed by the s-inv job. We call this job s-dsim.

The **sort** job is available as one of the code examples in the Hadoop distribution. The other jobs are jobs that we implemented. We use these jobs to construct the complex workloads described in Table 1. Figure 2 shows the actual completion time for W1–W3 on $P = 9$ physical machines (18 primary VMs) using three schedules: running the jobs sequentially giving each job 18 VMs, the schedule obtained using our pair scheduling algorithm, and the schedule obtained using our algorithm for scheduling multiple jobs. The figure shows that both our scheduling algorithms are better than the sequential schedule, and that the more flexible multiple-job algorithm is better than the pair scheduling algorithm, saving 24–33% of the completion time compared to sequential execution. For W4–W6, most of the jobs are large and resource intensive, so it is never beneficial to schedule more than two jobs on the cluster at a time. Thus, the pair scheduling algorithm and the multiple-job algorithm give the same schedules. Figure 3 shows the actual completion time for W4–W6 on $P = 10$ physical machines for sequential execution of the jobs in the workload and for the schedule obtained using our pair scheduling algorithm. The scheduler outperforms sequential execution by a margin of 16-30%.

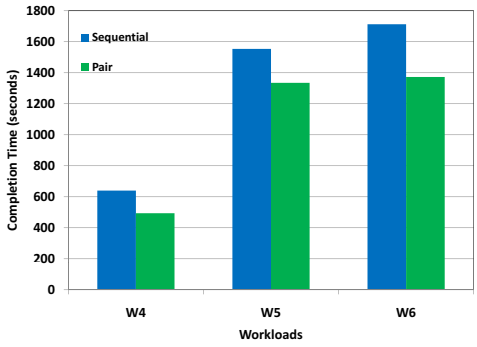

Figure 3: Scheduling for W4–W6

7. CONCLUSION

We presented a scheduler for Map-Reduce jobs whose goal is to minimize the total completion time of a set of jobs. Our scheduler is based on the popular Hadoop implementation of Map-Reduce, and its key highlights are: (1) using virtual machines to run the Hadoop nodes, (2) experiment-driven modeling of job performance, (3) taking advantage of the different characteristics of Hadoop jobs, including their CPU and I/O intensity. We demonstrated that our scheduler can achieve performance gains of 30% or more over a simple sequential execution of the jobs on the whole cluster.

8. REFERENCES

[1] Apache Software Foundation. Hadoop. http://hadoop.apache.org/.

[2] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, August 2009.

[3] T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with MapReduce. In *Proc. Annual Meeting of the Association for Computational Linguistics*, 2008.

[4] X. Hu and R. C. Eberhart. Solving constrained nonlinear optimization problems with particle swarm optimization. In *Proc. World Multiconference on Systemics, Cybernetics and Informatics*, 2002.

[5] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proc. Symp. on Operating System Design and Implementation (OSDI)*, December 2008.