A Case for Abstract Cost Models for Distributed Execution of Analytics Operators

Rundong Li[1](✉), Ningfang Mi[2], Mirek Riedewald[1], Yizhou Sun[3], and Yi Yao[2]

[1] CCIS, Northeastern University, Boston, USA
{rundong,mirek}@ccs.neu.edu
[2] ECE, Northeastern University, Boston, USA
{ningfang,yyao}@ece.neu.edu
[3] Department of Computer Science, UCLA, Los Angeles, USA
yzsun@cs.ucla.edu

Abstract. We consider data analytics workloads on distributed architectures, in particular clusters of commodity machines. To find a job partitioning that minimizes running time, a cost model, which we more accurately refer to as makespan model, is needed. In attempting to find the simplest possible, but sufficiently accurate, such model, we explore piecewise linear functions of input, output, and computational complexity. They are abstract in the sense that they capture fundamental algorithm properties, but do not require explicit modeling of system and implementation details such as the number of disk accesses. We show how the simplified functional structure can be exploited by directly integrating the model into the makespan optimization process, reducing complexity by orders of magnitude. Experimental results provide evidence of good prediction quality and successful makespan optimization across a variety of cluster architectures.

1 Introduction

With the ubiquitous availability of clusters of commodity machines and the ease of configuring them in the Cloud, there is growing interest in executing data analytics workloads in distributed environments such as Hadoop MapReduce and Spark. For effective use of resources, a job needs to be *partitioned* into tasks running in parallel on different workers. We will use the term *worker* to refer to a single processing unit, i.e., a single physical or virtual core. Hence a k-core machine would support up to k concurrent workers.

Given an analytics operator, our goal is to find a partitioning and degree of parallelism that minimizes total running time of the computation, also referred to as **makespan** of the corresponding set of tasks. Furthermore, we want to quantify the tradeoff between makespan and degree of parallelism. This is useful for identifying cases where a "good" makespan can be achieved with significantly fewer resources. For example, knowing that 36 concurrent workers achieve a makespan of 29.0 min, but 18 achieve 29.2 minutes, the user might decide to accept the small delay for the benefit of having 18 workers available for another application.

© Springer International Publishing AG 2017
L. Bellatreche and S. Chakravarthy (Eds.): DaWaK 2017, LNCS 10440, pp. 149–163, 2017.
DOI: 10.1007/978-3-319-64283-3_11

To analytically derive optimal parameter settings, the makespan model should have simple functional structure. Arguably the simplest approach with any hope for being practically useful is to estimate running time T of a task as a linear combination of input size (I), output size (O), and (asymptotic) number of computation steps (C):

$$T = c_0 + c_1 I + c_2 O + c_3 C. \tag{1}$$

This model is *abstract* in the sense that it reflects algorithm properties, not implementation or system aspects. The latter are captured by the parameters—representing fixcosts (c_0), data transfer rates (c_1, c_2), and processing speed (c_3)—learned through linear regression from a training set of workloads executed on the same system in advance. By learning from training runs, the parameters represent averages over a large number of low-level processing steps. Hence they automatically account for underlying processing complexities [5]. To apply Eq. 1, I, O, and C need to be expressed as functions of the *partitioning* parameters, e.g., number of tasks. (Note that the resulting function might not be linear in those parameters!) This requires human expertise, but is strictly easier than for traditional DBMS cost models. Consider the map phase of the MapReduce sort implementation, for which in Sect. 3 we derive task duration as $c_{m_0} + c_{m_1}(N/m) + c_{m_2}(N/m)\log(N/m)$. All that was needed to obtain this formula were (1) input and output size per task (N/m) and (2) complexity of sorting.

DBMS cost models are also linear in the sense that they are based on the sum of the number of operations, weighted by per-operation cost. However, they are significantly more complex than our approach, because they express cost at a lower level of abstraction. Beyond input size, output size, and asymptotic computation cost, the DBMS approach estimates the actual number of system-level operations such as random and sequential I/O. Those depend on implementation details of the underlying system. For example, a map task for sorting might perform sorting and partitioning completely in memory, or write multiple temp files that are merged on disk in one or more passes. Moreover, since DBMS cost models are concerned with resource usage, not running time or makespan, they do not take resource bottlenecks into account.

Machine learning models [2] could be trained directly for makespan prediction, but behave as "blackboxes", i.e., makespan optimization cannot exploit model structure. Intuitively, with all existing cost models, *finding the job partitioning that minimizes makespan requires trial-and-error style exploration of parameter combinations.* The search space can be very large, as it includes parameters controlling (i) number of tasks, (ii) degree of parallelism during execution, and (iii) problem-specific partitioning parameters. For matrix multiplication, there are 10 important such parameters (Sect. 4), requiring exploration of a 10-dimensional space of combinations. Our approach reduces complexity to three dimensions, because for the other seven we can derive optimal settings analytically. Assuming 4 values explored in each of those 7 dimensions, our approach reduces optimization cost by a factor of $4^7 \approx 16,000$.

But can a simple abstract makespan models capture the complexities of a distributed system, in particular resource **bottlenecks** during execution? Our

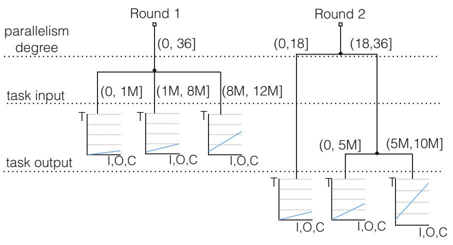


Fig. 1. Schematic illustration of piecewise linear models for a 2-round computation. The model for round 1 is partitioned on task input size only. The model for round 2 is partitioned on both parallelism degree and task output size.

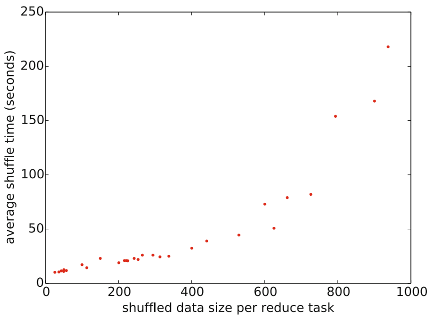


Fig. 2. Shuffle time vs. Data size (MB) for round 2 of the matrix product algorithm

experiments show that for a *piecewise* linear model (Fig. 1), it only took a small number of linear pieces to be sufficiently accurate. The reason for this lies in the way resources are consumed. Consider a network link that can transmit data at a certain rate. While below capacity, doubling the amount of data transmitted approximately doubles transfer time. Once link capacity is exceeded, data is held longer in buffers, effectively decreasing transfer rate. Figure 2 shows a typical observation for a MapReduce program, where the time for shuffling data across the network increases more rapidly after about 600 MB.

The model pieces also provide insights about bottlenecks. For example, for the reduce phase of sorting (Sect. 3), model training for a cluster of quad-core machines determined that three pieces were needed when all four cores were used. Input coefficient c_1 had value 5.5, 9.9, and 12 for “small”, “medium”, and “large” input size, respectively. For executions using only two cores per machine, the model created only two such pieces with c_1 equal to 4.4 for “small”, and 4.9 for “large” inputs. This reflects the I/O-dominated nature of sorting. With four cores competing for access to the data, larger input size stresses I/O and memory bus more than when only two cores are used. By discovering this behavior *automatically* from training data, our model can predict the effect of problem partitioning and parallelism degree.

For an initial proof-of-concept, this paper focuses on relatively “regular” problems—sorting and matrix product. This will be extended in future work through correction factors for skew.

2 Piecewise Linear Model Structure and Training

Let w denote the number of available workers, $p \leq w$ the degree of parallelism of the computation, and n the number of tasks executed during a round of computation. This implies that there will be $\lceil n/p \rceil$ *waves* of tasks in that round. If each task takes time T, then makespan of the round will be $\lceil n/p \rceil T$. This represents an idealized execution, which we show in experiments to be sufficiently accurate, as long as task interactions and bottlenecks are taken into account.

Interaction effects occur when tasks executed in parallel on a multicore processor compete for resources, e.g., memory bus and local disk(s). This competition for scarce resources in effect causes lower rate of data transfer and local computation experienced by the tasks. It can be represented by partitioning the model into $k \geq 1$ ranges $(p_0 = 0, p_1]$, $(p_1, p_2]$,$\ldots$, $(p_{k-1}, p_k = w]$ of degrees of parallelism, each with a different linear model. *Bottlenecks* appear not only when multiple tasks compete for resources. The local computation of a task might also get delayed by I/O wait time caused by its own I/O operations, requiring a different model for different ranges of input and output size.

The result of partitioning the design space is a family of piecewise linear models, each with its own (c_0, c_1, c_2, c_3) combination. We say that this model *covers* the corresponding partition defined by a range of parallelism degrees, input, and output size. The partitioning can be determined in a fully data-driven manner from the training data, e.g., by minimizing the residual sum of squares [20] or by using a model tree [15]. For parallelism degree, we propose a simplified approach where the partitioning is defined by multiples of the number of worker *machines*: For a cluster consisting of k-core machines, the interval endpoints defining a piece based on parallelism degree are a subset of $\{p_i = i \cdot w/k \mid i = 1, 2, \ldots, k\}$. This creates ranges that correspond to a degree of parallelism of 1 to k per physical machine. Figure 1 illustrates the overall structure of the proposed models. For each round of the computation, there is a separate piecewise linear model. A piece is a linear model as defined in Eq. 1, which covers a partition identified by a range of degrees of parallelism, input sizes and output sizes. For illustration purposes, the models in the figure are shown in 1-dimensional space.

Following common practice in machine learning, models are trained based on a set of representative instances of the given problem. As a training instance is executed, task running times for each round are measured. To train the models for a round, we use the *average* task running time, input size (I), and output size (O) for this round. Given these values, computation cost (C) is derived based on the formula expressing computation cost in terms of input and output size.

3 Makespan Model for Sorting

Sorting plays a central role in data analysis, therefore we first demonstrate how to apply abstract piecewise linear makespan models to the classic sort algorithm in Hadoop MapReduce.

3.1 Round-Time Estimation for Map and Reduce Phase

To apply Eq. 1, I, O, and C need to be expressed in terms of parameters controlling the problem partitioning in each round of computation. In the first round, the map phase, each map task reads records and emits them. The record sets are partitioned and sorted by key, then transferred to the reduce tasks. Each reduce task merges the pre-sorted runs it receives from different map tasks, then emits the records. Our goal is to set number of map tasks, m, number of reduce tasks, r, and parallelism degrees p_m and p_r for map and reduce phase, respectively, to minimize makespan.

With N denoting input size, each map task receives $I = N/m$ input and writes it all out. Since $I = O$, the two separate terms $c_1 I$ and $c_2 O$ collapse to a single term $c_{m_1}(N/m)$, i.e., there is a single coefficient capturing the aggregate of data reading and writing time. (As a by-product, fewer model coefficients allow for smaller training data.) Computational complexity is $(N/m)\log(N/m)$ for sorting. (Note how this abstracts away system details such as the number of disk page accesses.) Hence map task time is modeled as $T_{map} = c_{m_0} + c_{m_1}(N/m) + c_{m_2}(N/m)\log(N/m)$. Given a degree of parallelism p_m, the map phase requires $\lceil m/p_m \rceil$ waves, resulting in round time

$$RT_{map} = T_{map} \cdot \lceil m/p_m \rceil = (c_{m_0} + c_{m_1}(N/m) + c_{m_2}(N/m)\log(N/m))\lceil m/p_m \rceil.$$

Since a reduce task pulls and merges pre-sorted files, then simply reads and emits all its records in order, it follows that all costs are linear in the reduce task's input size, i.e., $I = O = C = N/r$. (Again, system details such as the number of passes for merging of files are abstracted away.) Hence the corresponding terms in Eq. 1 collapse. Analogous to the map phase, there will be $\lceil r/p_r \rceil$ waves, resulting in round time

$$RT_{reduce} = (c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_r \rceil.$$

3.2 Exploiting Model Structure for Optimization

Consider finding optimal number of reduce tasks, r, and parallelism degree p_r:

$$\underset{r,p_r}{\text{argmin}} \quad RT_{reduce} = (c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_r \rceil. \tag{2}$$

Lemma 1. *Model $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p \rceil$ covering parallelism-degree range $(p_l, p_h]$ and task input range $(s_l, s_h]$ is minimized by setting $p = p_h$ and $r = \min\{\lceil r_l/p_h \rceil \cdot p_h; r_h\}$, where $r_l = \lceil N/s_h \rceil$ and $r_h = \lfloor N/(s_l+1) \rfloor$.*

Proof. For task input size N/r, range $(s_l, s_h]$ of input sizes implies that the model is valid for reduce task number r in range $r_l \leq r \leq r_h$ with $r_l = \lceil N/s_h \rceil$ and $r_h = \lfloor N/(s_l+1) \rfloor$. Consider any (r, p) in the valid range, i.e., $r_l \leq r \leq r_h$ and $p_l < p \leq p_h$. For any r, $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p \rceil$ is minimized by selecting the greatest possible value for p, i.e., $p = p_h$. Hence we need to find the value of r that minimizes $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_h \rceil$.

Case 1: the range of possible values for r contains a multiple of p_h. We show that the smallest such multiple minimizes time. Formally, the case condition states that there exists an integer $k \geq 1$ such that $r_l \leq kp_h \leq r_h$. For any such

Algorithm 1. Find p_r and r that minimize RT_{reduce} of sort

```
Input: $N$; $M$ = set of models $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_r \rceil$, each covering some range
    $(p_l, p_h]$ of parallelism degrees and some range $(s_l, s_h]$ of reduce-task input sizes
1: for all model $m \in M$ do // $m$ covers $(p_l, p_h]$ and $(s_l, s_h]$
2:     $t \leftarrow$ time returned by model $m$ when setting $p_r = p_h$ and $r = \min\{\lceil \lceil \frac{N}{s_h} \rceil / p_h \rceil \cdot p_h; \lfloor \frac{N}{s_l+1} \rfloor\}$
3:     Keep track of smallest $t$
4: Return minimal time $t$ and its $(p_r, r)$ combination
```

k, consider all $r \in [r_l, r_h]$ that satisfy $(k-1)p_h < r \leq kp_h$. The latter implies $\lceil r/p_h \rceil = k$ and therefore $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_h \rceil = k(c_{r_0} + c_{r_1}(N/r))$. This formula is minimized by selecting the greatest possible r in $(k-1)p_h \leq r \leq kp_h$, i.e., $r = kp_h$. Then $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_h \rceil = k(c_{r_0} + c_{r_1}\frac{N}{kp_h}) = kc_{r_0} + c_{r_1}N/p_h$. This formula is minimized by setting k to the smallest possible value that satisfies the case condition $r_l \leq kp_h \leq r_h$, i.e., $k = \lceil r_l/p_h \rceil p_h$.

Case 2: the range of possible values for r *does not* contain a multiple of p_h. Then there exists an integer $k' \geq 1$ such that $(k'-1)p_h < r_l \leq r_h < k'p_h$. This implies $\lceil r/p_h \rceil = k'$ for *all* values of r in $(r_l, r_h]$, and hence $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_h \rceil = k'(c_{r_0} + c_{r_1}(N/r))$. This formula is minimized by selecting the greatest possible r, i.e., $r = r_h$.

To put the solutions for both cases together, notice that for case 1 $\lceil r_l/p_h \rceil p_h \leq r_h$ and for case 2 $r_h \leq \lceil r_l/p_h \rceil p_h$. Hence, in general, $(c_{r_0} + c_{r_1}(N/r)) \cdot \lceil r/p_h \rceil$ is minimized by $r = \min\{\lceil r_l/p_h \rceil \cdot p_h; r_h\}$, completing the proof.

Lemma 1 forms the foundation for Algorithm 1. Instead of exhaustively exploring (r, p_r) combinations, optimization cost is linear in the number of model pieces. Using more linear pieces improves model accuracy, but increases optimization cost—a directly tunable tradeoff.

To understand how the optimization process takes task interactions and bottlenecks into account, consider first the special case where a single makespan model M covers all parallelism degree values $p_r \in (0, w]$, and all reduce-task input sizes $s \in (0, s_h]$, where $s_h > N$. The for-loop in Algorithm 1 would be executed once, returning $p_r = w$ and $r = \min\{w; N\} = w$. (Note that $N/s_h < 1$ and we assume $N \geq w$, i.e., the number of workers does not exceed the number of input records.) Stated differently, the algorithm determines that the problem should be partitioned into w tasks—one per worker—and all tasks should be executed in a single wave in parallel.

Now consider a cluster of $w/2$ dual-core machines and assume that when using both cores on a worker, the memory bus on the worker slows down data transfer rate from memory to core, causing the cores to wait for data. During model construction, our approach would automatically determine from the training data that two different linear models are needed: one covering parallelism degree $p_r \in (0, w/2]$, and the other $p_r \in (w/2, w]$. The for-loop in Algorithm 1 now compares predicted makespan for two configurations (p_r, r): $(w/2, w/2)$ for the model covering $p_r \in (0, w/2]$ and (w, w) for the model covering $p_r \in (w/2, w]$.

Stated differently, if the memory-bus bottleneck leads to a severe slowdown, the optimal solution may be to use only half of the cores—one per machine—and execute the reduce phase in a single wave of $w/2$ concurrently executed tasks. This perfectly captures the intuition that if the memory bus is the bottleneck (and not the CPU), then it may be better to only use one of the two cores per machine.

4 Dense Matrix Product

The second test case for our approach, dense matrix multiplication, represents a more challenging workload with high data transfer costs, but also significant CPU load in some rounds due to the large number of multiplications and additions. Furthermore, matrix partitioning increases total cost due to data replication. Dense matrix multiplication was identified as an important computation problem in a recent UC Berkeley survey on the parallel computing landscape [3]. Also note that the closed-form solution to the linear regression problem $y = X\beta + \epsilon$, given by the ordinary least squares estimator $\hat{\beta} = (X^T X)^{-1} X^T y$, involves the product of matrices that are often dense.

4.1 Makespan Model for Block-Wise Matrix Multiplication

Dense matrix-matrix multiplication can be parallelized by partitioning each matrix into blocks. We discuss the makespan model for the MapReduce implementation. (The approach for Spark is analogous.) As illustrated in Fig. 3, input matrix U with dimensions $N_0 \times N_1$ is partitioned into $B_0 \cdot B_1$ blocks, each of size N_0/B_0 by N_1/B_1; V (with dimensions $N_1 \times N_2$) is partitioned into $B_1 \cdot B_2$ blocks, each of size N_1/B_1 by N_2/B_2. Each block from U will be multiplied with the B_2 corresponding blocks from V, for a total of $B_0 \cdot B_1 \cdot B_2$ block-pair multiplication tasks. Note that each U block is duplicated B_2 times, each V block B_0 times. The data duplication (map: round 1) and local multiplication (reduce: round 2) form the **multiplication job (m-job)**. If $B_1 > 1$, then each block-pair product represents only a partial result. In that case an **aggregation job (a-job)** needs to read and re-shuffle these partial results (map: round 3) and sum them up (reduce: round 4).

For $i \in \{1, 2, 3, 4\}$, let p_i and n_i denote degree of parallelism and number of tasks, respectively, in round i. From the analysis above follows $n_2 = B_0 B_1 B_2$. Similar to the sort program, an analysis of input, output, and computation in each round results in the following round time estimators: (Note that rounds 3 and 4 are executed if and only if $B_1 > 1$.)

$$
\begin{aligned}
RT_1 &= (c_{1_0} + c_{1_1}(N_0N_1 + N_1N_2)/n_1 + c_{1_2}(N_0N_1B_2 + N_1N_2B_0)/n_1) \cdot \lceil n_1/p_1 \rceil, \\
RT_2 &= (c_{2_0} + c_{2_1}(\frac{N_0N_1}{B_0B_1} + \frac{N_1N_2}{B_1B_2}) + c_{2_2}\frac{N_0N_2}{B_0B_2} + c_{2_3}\frac{N_0N_1N_2}{B_0B_1B_2}) \cdot \lceil B_0B_1B_2/p_2 \rceil, \\
RT_3 &= (c_{3_0} + c_{3_1}N_0N_2B_1/n_3) \cdot \lceil n_3/p_3 \rceil, \\
RT_4 &= (c_{4_0} + c_{4_1}N_0N_2B_1/n_4 + c_{4_2}N_0N_2/n_4) \cdot \lceil n_4/p_4 \rceil.
\end{aligned}
$$

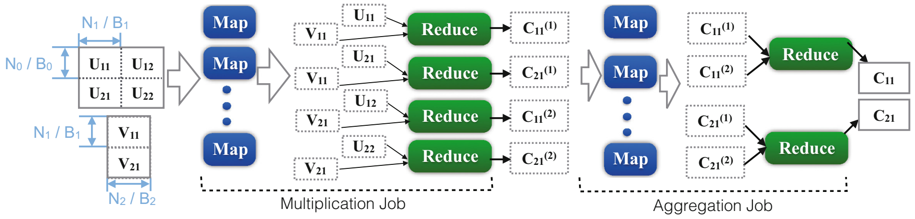


Fig. 3. Block-wise parallel matrix multiplication in 4 rounds. U is partitioned into 2×2 blocks, V into upper and lower half, i.e., $(B_0, B_1, B_2) = (2, 2, 1)$.

4.2 Optimal Partitioning

The problem partitioning that minimizes estimated makespan is defined as $\text{argmin}_{B_0,B_1,B_2,p_1,p_2,p_3,p_4,n_1,n_3,n_4} RT_1 + RT_2 + RT_3 + RT_4$. With traditional cost models, this would require trial-and-error based exploration of a *10-dimensional search space*. Using our approach, we can show, like for sorting, that the optimal setting for parallelism degree is $p = p_h$ for a model covering range $(p_l, p_h]$. The optimal task number n is $\min\{\lceil \frac{n_l}{p_h} \rceil \cdot p_h; n_h\}$. Here n_l and n_h denote the lower and upper extreme of the range of possible choices for the corresponding n_i so that task input and output size are in the range covered by the model piece for round i. Hence the optimization problem simplifies to

$$\underset{B_0,B_1,B_2}{\text{argmin}}\ RT_1 + RT_2 + RT_3 + RT_4, \tag{3}$$

where n_1, n_3, n_4, p_1, p_2, p_3, and p_4 are all computed directly as discussed above. This reduces optimization cost by *orders of magnitude*, from search in 10 dimensions to 3 dimensions. (Note that optimization cost is linear in the total number of linear pieces, across all rounds.)

5 Experiments

The main purpose of the experiments is to provide a *proof of concept* that abstract piecewise linear makespan models with a "small" number of pieces are accurate *enough* to rank "good" above "bad" data partitionings. Accuracy comparisons to traditional cost models, in particular DB optimizer cost formulas and blackbox models, are not included. Our abstract models trade off prediction accuracy (hence will be less accurate than a carefully designed and tuned traditional model), to gain in terms of two unique properties: (1) Make it easier to specify the model for a given data analytics operator, and (2) enable more efficient running-time optimization algorithms by exploiting the simple model structure. Note that in all experiments, the piecewise linear models had between 1 and 7 pieces per round.

Table 1. Cluster specifications

Name	#Machines	#Cores per machine	#Workers	Memory per machine	Software
9h36	10	4	36	8 GB	Hadoop 1.2
2h24	3	12 (virtual)	24	47 GB	Hadoop 2.4
20h160	21	8	160	64 GB	Hadoop 2.4
Emr10	11	1 (virtual)	10	3.75 GB	Hadoop 2.6
6s12	7	2	12	8 GB	Spark 1.6.1
Emr12s	7	2 (virtual)	12	7.5 GB	Spark 1.6.1

5.1 Basic Setup

We show representative results on six different systems with diverse properties. They include in-house clusters (9h36,2h24,6s12), a research cluster (20h160) provided by CloudLab [22] and two (Emr10,Emr12s) on Amazon Web Services. For details see Table 1.

For simplicity, in most experiments on Hadoop, the number of map tasks is left at the Hadoop default value, i.e., total map input size divided by Hadoop Distributed File System (HDFS) block size. Only for small data sets whose size is smaller than the product of desired parallelism degree and HDFS block size, we set the number of map tasks equal to the desired parallelism degree.

5.2 Sorting

We present measurements on clusters 9h36 and Emr10. All piecewise models for 9h36 are partitioned into ranges $(0, 18]$ and $(18, 36]$ on parallelism degree. Possible partitioning on task input and output size is determined automatically as discussed in Sect. 2. We create 15 different data sets with 100 million to 2.7 billion randomly generated records of type Long (8 bytes per record), and for each data set we use various numbers of waves (up to 10) in the reduce phase. In total, there are 54 problem instances. A subset of 41 of these is used for model training, the others for testing.

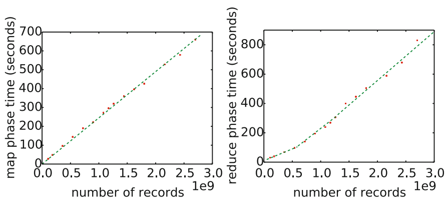


Fig. 4. Sorting: measured round time vs. Input size on 9h36 for Map (left) and Reduce (right) phase

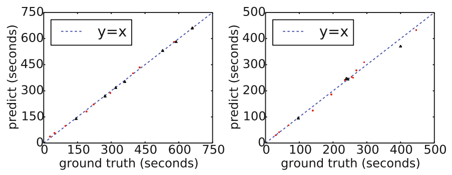


Fig. 5. Sorting: predicted vs. Measured round time on 9h36 for Map (left) and Reduce (right) phase

Figure 4 presents measurements of the relationship between input size and round time. In particular, the y-axis reports the true value for RT_{map} and RT_{reduce}, computed as the product of average *measured* task running time and number of waves in the round. Degree of parallelism is set to the number of workers for all runs. The dotted green line shows a piecewise linear model fitted to the data.

Figure 5 compares predicted and measured round time of map and reduce phase for sorting on cluster `9h36`, using either all or only half of the available cores. The red dots are for training cases, while the green triangles are for test cases. All individual times and the overall trend are captured very accurately, as the *relative errors* are mostly around 1%, and never exceed 5%.

Table 2. Degree of parallelism vs. Measured and predicted makespan on `9h36`.

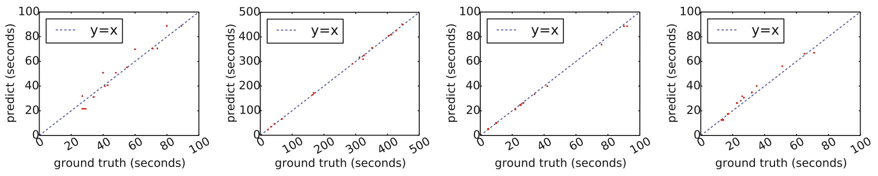
Number of records	Degree of parallelism = 18		Degree of parallelism = 36	
	True (sec)	Prediction	True (sec)	Prediction
1.17E + 9	790	601.96	698	564.21
1.26E + 9	835	657.36	723	629.59
1.62E + 9	1056	842.00	1050	833.66
1.80E + 9	1146	928.18	1112	926.13
2.43E + 9	1558	1254.39	1524	1288.04
2.70E + 9	1751	1408.24	1741	1465.02

Table 2 shows that our models significantly underestimate true makespan. This is caused by tasks starting and/or finishing later than others, delaying job completion. However, this bias is consistent, allowing the model to capture the trend correctly, no matter if all cores or only half of them is used per machine. For large inputs, it identifies the I/O-related bottleneck: doubling the number of cores used per machine results in virtually no improvement of makespan when data size reaches 1.6 billion records.

5.3 Matrix Multiplication

All models are partitioned into parallelism-degree ranges based on multiples of the number of machines in the cluster; partitioning on input and output size is determined automatically as discussed in Sect. 2. The training set consists of 104 problem instances, covering 12 different matrix-size combinations (square matrices from $10k \times 10k$ to $30k \times 30k$ and also extreme rectangular ones up to $200 \times 4 \cdot 10^6$), each with 3 to 20 (B_0, B_1, B_2)-combinations. We test the model on 57 independent problem instances, covering 10 different matrix sizes in the same broad range. As Fig. 6 shows, predicted and true round times are very close.

Like for sorting, our model significantly underestimates true makespan, but can still correctly separate "good" from "bad" problem partitionings. In all cases



(a) M-Job: Map (b) M-Job: Reduce (c) A-Job: Map (d) A-Job: Reduce

Fig. 6. Matrix product: predicted vs. measured round time on `9h36`. The test cases (red dots) are near the perfect-prediction line (blue dotted line). (Color figure online)

Table 3. Ranking quality: predicted vs. true makespan (in sec) for matrix product (Hadoop MapReduce, (a)∼(c) are synthetic data, (d) and (e) are real data)

(a) $15,000 \times 15,000$ matrices on `9h36`

			prediction		ground truth	
B_0	B_1	B_2	makespan	rank	makespan	rank
6	1	6	305.40	1	400.00	1
3	3	4	330.87	2	434.00	2
4	1	4	345.89	5	440.00	3
3	3	3	350.39	7	445.67	4
3	4	3	333.55	3	448.00	5
5	1	5	356.48	9	452.00	6
3	2	3	344.69	4	453.00	7
2	6	3	348.85	6	471.00	8
4	2	4	353.48	8	479.00	9
2	9	2	385.02	11	485.00	10
2	6	2	380.85	10	497.00	11
2	4	2	403.84	13	505.00	12
2	8	2	410.55	14	525.00	13
2	7	2	446.67	15	548.00	14
4	1	8	401.43	12	556.00	15
2	2	2	614.17	16	656.00	16
1	18	1	638.41	17	713.00	17
1	36	1	941.19	18	1,290.00	18

(b) $15,000 \times 15,000$ matrices on `2h24`

			prediction		ground truth	
B_0	B_1	B_2	makespan	rank	makespan	rank
4	1	6	247.93	1	325.03	1
2	4	3	267.90	4	366.53	2
2	3	4	257.58	3	384.64	3
3	2	4	248.79	2	388.53	4
2	6	2	290.78	5	408.92	5
1	12	2	356.74	6	455.77	6
1	24	1	765.48	7	574.52	7

(c) $10,000 \times 60,000$ matrices on `20h160`

			prediction		ground truth	
B_0	B_1	B_2	makespan	rank	makespan	rank
4	10	4	124.57	1	186	1
4	8	5	128.14	2	204	2
2	20	4	132.53	3	205	3
4	5	8	134.07	4	205	3
2	8	10	137.51	5	206	5
1	20	8	141.89	6	211	6
12	1	12	171.08	7	238	7

(d) 68×2458285 matrices (1990 US Census data) on `9h36`

			prediction		ground truth	
B_0	B_1	B_2	makespan	rank	makespan	rank
1	18	1	30.93	1	95.00	1
1	36	1	37.97	2	107.25	2
3	2	3	59.05	4	127.00	3
2	9	2	51.12	3	128.5	4
3	4	3	64.20	5	132.25	5
6	1	6	85.72	6	145.00	6

(e) 481×191779 matrices (KDD Cup 1998 data) on `9h36`

			prediction		ground truth	
B_0	B_1	B_2	makespan	rank	makespan	rank
1	18	1	24.12	1	94	1
1	36	1	30.89	2	103	2
3	2	3	39.19	4	109	3
2	9	2	37.43	3	112	4
3	4	3	44.61	5	121	5
6	1	6	48.37	6	144	6

our approach would find a near-optimal configuration. Table 3 confirms this for both synthetic and real data sets (from the UCI Machine Learning Repository [13]). There our technique is applied to the step where the data matrix is multiplied with its own transpose during linear regression analysis. Table 4 confirms that this observation also holds for Spark.

Note that for both real data sets (Table 3d, e), our model correctly discovers that setting (B_0, B_1, B_2) to $(1, 18, 1)$ results in lower makespan than $(1, 36, 1)$. We confirmed that due to I/O bottlenecks, it is better to only use half of the available cores per machine, even though round 2 performs a huge number of arithmetic operations (more than $11 \cdot 10^9$ for the Census data).

Table 4. Ranking quality: predicted vs. true makespan (in sec) for matrix product (synthetic data, Spark)

(a) $800 \times 80,000$ matrices on `6s12`

			prediction		ground truth	
B_0	B_1	B_2	makespan	rank	makespan	rank
2	2	3	73.81	1	88.8	1
2	3	2	74.08	3	90.67	2
1	12	1	73.88	2	96	3
1	3	4	87.84	4	101	4
1	6	2	100.10	7	101.4	5
1	4	3	96.79	6	104	6
3	1	4	133.95	9	109.5	7
1	6	1	92.80	5	113	8
2	1	3	154.30	11	134	9
1	2	3	134.22	10	141	10
1	3	2	131.48	8	154	11

(b) 6000×6000 matrices on `Emr12s`

			prediction		ground truth	
B_0	B_1	B_2	makespan	rank	makespan	rank
3	1	4	144.73	1	149.5	1
2	2	3	152.50	2	152	2
2	3	2	156.63	3	162	3
1	2	6	171.79	5	170.5	4
1	3	4	166.27	4	171	5
1	4	3	180.81	6	173.5	6
1	6	2	184.95	7	195	7
2	1	3	251.14	9	254	8
1	2	3	268.33	8	268.5	9
1	3	2	277.20	11	277	10
1	1	6	266.92	10	304	11
1	6	1	365.17	12	362	12

6 Related Work

Structured cost models that capture execution details are essential for query optimization in relational DBMS [16], and they can be highly accurate when tuned [23]. This motivated similar approaches for MapReduce and other distributed data analysis systems [11,14,19,21,24]. As an alternative to structured cost models, blackbox-style machine learning techniques were explored for a variety of performance prediction problems [2,5,6,8,10]. For all previous cost models, the effect of problem-partitioning parameters on makespan is relatively complex, hence makespan minimization would have to rely on trial-and-error style exploration of possible parameter settings. For dense matrix multiplication, this corresponds to a 10-dimensional space of $(B_0, B_1, B_2, p_0, p_1, p_2, p_3, n_1, n_3, n_4)$ combinations. (Note that Ernest [19] could possibly derive optimal settings for all p_i, $i = 0, \ldots, 3$, reducing complexity to 6 dimensions.) In contrast, our approach sacrifices some prediction accuracy to simplify model structure. This enables analytical derivation of optimal settings for most parameters, reducing complexity to 3 dimensions for dense matrix multiplication.

We use dense matrix multiplication to showcase model design and makespan optimization for an analytics operator with a demanding I/O and CPU profile. Previous work explored a variety of performance-related aspects for matrix multiplication on parallel architectures. This includes load balancing [9], minimizing communication cost [1,4,12,17], and optimizing for memory hierarchy [7,18].

7 Conclusions

The goal of this work was to find the "simplest possible" realistic model to predict makespan for distributed execution of data analytics operators. To this end, we proposed abstract models that are piecewise linear functions depending only on input, output, and computation complexity. Our approach has two main benefits. First, it simplifies tying problem-partitioning parameters to model variables (input, output and computation) for user-defined operators, e.g., programs written in MapReduce or Spark. Second, we showed that the linear structure can be exploited for more efficient optimization algorithms. This reduces optimization complexity from a search process in ten dimensions to only three for matrix product; for sorting the optimal solution was directly obtainable in closed form.

Our experiments indicated that a small number of pieces achieves sufficient prediction quality, enabling us to find near-optimal problem partitionings and to identify when a lower parallelism degree delivers the same (or better) makespan.

In future work, we will explore how to extend this approach to workloads that are more heterogeneous in the sense that individual tasks may vary widely in their cost. Moreover, we will consider tuning partitioning parameters along with system parameters external to user programs, by integrating our ideas into optimizers like Starfish [10].

Acknowledgments. This work was supported by a Northeastern University (NU) Tier 1 award, by the National Institutes of Health (NIH) under award number R01 NS091421, by the Air Force Office of Scientific Research (AFOSR) under grant number FA9550-14-1-0160, and by the National Science Foundation (NSF) Career Award #1741634. The content is solely the responsibility of the authors and does not necessarily represent the official views of NU, NIH, AFOSR or NSF. The authors also would like to thank the reviewers for their constructive feedback.

References

1. Agarwal, R.C., Balle, S.M., Gustavson, F.G., Joshi, M., Palkar, P.: A three-dimensional approach to parallel matrix multiplication. IBM J. Res. Dev. **39**(5), 575–582 (1995)
2. Akdere, M., Cetintemel, U., Riondato, M., Upfal, E., Zdonik, S.: Learning-based query performance modeling and prediction. In: Proceedings of the ICDE, pp. 390–401 (2012)
3. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)

4. Ballard, G., Buluc, A., Demmel, J., Grigori, L., Lipshitz, B., Schwartz, O., Toledo, S.: Communication optimal parallel multiplication of sparse random matrices. In: Proceedings of the SPAA, pp. 222–231 (2013)
5. Duggan, J., Cetintemel, U., Papaemmanouil, O., Upfal, E.: Performance prediction for concurrent database workloads. In: Proceedings of the SIGMOD, pp. 337–348 (2011)
6. Duggan, J., Papaemmanouil, O., Çetintemel, U., Upfal, E.: Contender: A resource modeling approach for concurrent query performance prediction. In: Proceedings of the EDBT, pp. 109–120 (2014)
7. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Rev. **46**(1), 3–45 (2004)
8. Ganapathi, A., Kuno, H.A., Dayal, U., Wiener, J.L., Fox, A., Jordan, M.I., Patterson, D.A.: Predicting multiple metrics for queries: Better decisions enabled by machine learning. In: Proceedings of the ICDE, pp. 592–603 (2009)
9. van de Geijn, R.A., Watts, J.: Summa: Scalable universal matrix multiplication algorithm. University of Texas at Austin, Technical report (1995)
10. Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of mapreduce programs. VLDB **4**(11), 1111–1122 (2011)
11. Huang, B., Babu, S., Yang, J.: Cumulon: optimizing statistical data analysis in the cloud. In: Proceedings of the SIGMOD, pp. 1–12 (2013)
12. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distrib. Comput. **64**(9), 1017–1026 (2004)
13. Lichman, M.: UCI machine learning repository (2013)
14. Morton, K., Balazinska, M., Grossman, D.: Paratimer: a progress indicator for mapreduce DAGs. In: Proceedings of the SIGMOD, pp. 507–518 (2010)
15. Quinlan, J.R., et al.: Learning with continuous classes. In: Australian Joint Conference on Artificial Intelligence, vol. 92, pp. 343–348 (1992)
16. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw-Hill, New York (2003)
17. Solomonik, E., Demmel, J.: Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6853, pp. 90–109. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23397-5_10
18. Valsalam, V., Skjellum, A.: A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. In: Concurrency and Computation: Practice and Experience, vol. 14(10), pp. 805–839 (2002)
19. Venkataraman, S., Yang, Z., Franklin, M., Recht, B., Stoica, I.: Ernest: efficient performance prediction for large-scale advanced analytics. In: NSDI, pp. 363–378 (2016)
20. Vieth, E.: Fitting piecewise linear regression functions to biological responses. J. Appl. Physiol. **67**(1), 390–396 (1989)
21. Wang, G., Chan, C.Y.: Multi-query optimization in mapreduce framework. In: Proceedings of the VLDB, pp. 145–156 (2013)

22. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proceedings of the OSDI, pp. 255–270 (2002)
23. Wu, W., Chi, Y., Hacígümüş, H., Naughton, J.F.: Towards predicting query execution time for concurrent and dynamic database workloads. Proc. VLDB **6**(10), 925–936 (2013)
24. Zhang, X., Chen, L., Wang, M.: Efficient multi-way theta-join processing using mapreduce. Proc. VLDB **5**(11), 1184–1195 (2012)