Partition Balancing in Distributed Nested Data-Parallel Systems

Frank Austin Nothaft
Department of Electrical Engineering and Computer Science, University of California, Berkeley
fnothaft@eecs.berkeley.edu

Michael Linderman
Carl Icahn School of Medicine at Mount Sinai
michael.linderman@mssm.edu

Abstract

Map-reduce frameworks such as Apache Hadoop and Spark provide the abstraction of a large, flat array that is processed in parallel across many machines. While this simple programming model has enabled the broad adoption of data-parallel distributed frameworks, these systems cannot express irregular parallel computation, and their performance is impacted by data imbalance across nodes. In this paper, we present a distributed framework for implementing *nested data parallel* (NDP) computation. Unlike previous NDP systems described by Blelloch and Bergstrom et al. that relied on the use of a *flattening* transformation at compile-time, we present a cost model that is used to select between multiple partitioning strategies at run-time. Through this, we provide a user-tunable means for trading node-to-node imbalance versus communication when executing distributed NDP programs.

1. Introduction

The use of map-reduce as a flat data-parallel (FDP) programming model for distributed systems has grown rapidly since it was introduced in Google's seminal 2004 paper [8]. The development of the open-source Apache Hadoop system enabled the use of this programming model outside of Google. Modern map-reduce systems such as Apache Spark [15] have refined the programming model further by reducing the dependency of the framework on disk via in-memory caching. While this refinement has enabled the use of map-reduce for iterative workloads, the programming model remains confined to computation that can be expressed via flat data-parallel operations. Specifically, Spark presents users with the abstraction of a resilient distributed dataset (RDD), which appears as a flat array that is distributed across compute nodes in a cluster [14].

To expand the algorithms that could be expressed as a data-parallel computation, Blelloch introduced the nested data-parallel (NDP) model [6]. In this programming model, users are provided the abstraction of an array whose elements are a nested level of arrays and data-parallel operators are applied on the nested arrays. A primary complication in the implementation of the NDP model is that the nested arrays frequently do not have uniform size. Several approaches have been suggested for balancing work in NDP programs, including the compile-time vectorization of NDP programs for execution on single-instruction multiple-data (SIMD) machines [6, 7] and flattening for multiple-instruction multiple-data (MIMD) machines [3]. Additionally, dynamic work-stealing approaches have been implemented [4].

In this paper, we introduce a distributed programming framework for NDP algorithms. Our implementation is built on top of Apache Spark. We track the structure of the nested arrays at runtime and choose between two strategies (*uniform* and *segmented*) for partitioning data across machines based on the estimated cost of each strategy. In the segmented partitioning strategy, all values in a single nested segment are co-located on a single compute node, and most computation can proceed without communication. The uniform strategy provides perfect load balance across all nodes, but many operations will need to communicate to execute. To choose between these strategies at runtime, we provide a cost model that evaluates the performance tradeoff of communication overhead versus node-to-node imbalance. Since the partitioning strategy is chosen at runtime, we can leverage knowledge of the nested array structure.

This work introduces the following contributions:

1. We introduce a dynamically optimized distributed method for implementing NDP algorithms that is amenable to "cloud computing" platforms.
2. We demonstrate that the current best-known `scan` algorithm has $O(\frac{n \log p}{p})$ performance on bulk-synchronous systems, and provide a "block-parallel" relaxation of the `scan` operator that has $O(\frac{n}{p})$ performance.
3. We demonstrate a model-based approach for balancing partition load in distributed systems.

2. Background

Understanding the system we describe in the rest of the paper requires an understanding of the NDP processing model, as well as the design of current MapReduce-style distributed computing frameworks. In this section, we compare and contrast the FDP and NDP programming models, and discuss the system design of the Spark MapReduce framework. While the problem of implementing scalable distributed FDP frameworks is well studied [8, 15], this paper builds upon those systems to focus on the implementation of scalable distributed NDP platforms.

2.1 Data-Parallel Programming Models

In data-parallel frameworks, processing proceeds in parallel across all of the elements in a dataset. We can further subdivide data-parallel programming models into flat and nested data parallelism:

Flat data-parallelism: In a FDP system, the parallel collection contains single elements which are then processed in parallel. On

a collection containing elements of type T, a few basic operations include:

- `map`, which applies a function $F : T \mapsto U$ to all elements of the collection, and returns a new collection containing elements of type U
- `reduce`, which successively applies a function $F : T, T \mapsto T$ to pairs of elements from the collection until all elements have been reduced down into a single scalar of type T

FDP may also be known as *regular* data parallelism, as the data parallel computation proceeds over the single items in the dataset (all parallel computation occurs on single elements, i.e., "collections" of size 1). Because there are no dependencies between elements in the dataset, FDP can implemented efficiently on a broad range of computing platforms, including SIMD and MIMD architectures, graphics processing units (GPUs) [12], and distributed systems [8].

Nested data-parallelism: An NDP system extends the collection structure of an FDP system. Instead of a flat collection, the collection is divided into *segments* or *nests*, which are sub-groupings of elements. Operations may then execute across the whole collection, or across the segments inside of the collection. A full introduction to NDP can be found in the text by Blelloch [6]. Blelloch justifies the utility of the NDP model by noting that NDP reduces the asymptotic runtime of some graph and linear algebra algorithms by a factor of $\log n$. A good demonstration of the utility of the nested vector model comes from implementing a matrix-vector multiply. Here, we assume that we have a matrix A which has been packed into a nested vector by splitting each row i into it's own segment, and a dense vector v. We can implement the matrix-vector multiply by:

```
def matrix-vector-multiply(A: SegmentedArray[Int],
                                    v: Array[Int]
  ): Array[Int] = {
  A.map-with-index((point: Int, idx: Int) => {
    point * v(idx)
  }).segmented-reduce(_ + _)
}
```

Here, we use the `map-with-index` function to multiply each point in the matrix by the corresponding point in the vector. This can execute in parallel per element. The `segmented-reduce` function then sums up all of the elements from each row. The naïve implementation of a `segmented-reduce` is parallel per segment, but an optimized implementation is parallel per element in the whole collection.

2.2 Data-Parallel Distributed Computing

This work builds upon the infrastructure provided by the Apache Spark distributed data-parallel computing framework [2, 15]. Spark was designed for "cloud computing" platforms, where machines may be unreliable, and where network performance may preclude the use of traditional distributed message passing systems such as the Message Passing Interface (MPI). Unlike Apache Hadoop, where data is shuffled to/from disk between all processing stages [1, 13], Spark has an in-memory processing model. The in-memory processing model leads to large ($100\times$) performance for iterative jobs implemented using Spark instead of Hadoop [15].

Spark's programming model is implemented on top of the *resilient distributed dataset* (RDD) abstraction [14]. The RDD abstraction presents a view of an array which is chopped into *partitions* that are distributed over the computers within the Spark cluster. Programs enqueue data-parallel transformations on the Spark master, which are then interpreted into a directed-acyclic graph (DAG) for execution; this approach is similar to the DAG scheduling approach pioneered in Dryad [11]. Spark executes the DAG whenever a disk shuffle is required, or if a newly enqueued operation would create a cycle in the DAG (i.e., an iterative computation is scheduled).

To execute a stage, the Spark master serializes the user defined function (UDF) which is being applied, transmits this function to all worker nodes, and then applies the computation. While the general application programming interface (API) that Spark provides is data-parallel across all elements, the API transformations are implemented by applying the transformation iteratively across all elements of a partition—Spark also exposes this parallel-by-partition interface via the `mapPartitions` call. If data must be moved between partitions after an execution stage, a disk shuffle will occur. Disk shuffles are analogous to an execution stage with interleaved computation and all-to-all communication followed by a barrier before the next stage. This process is similar to how the results of `map` phases are moved to reducers in both MapReduce and Hadoop [8]—the main distinction is that Spark allows successive `map` phases, and that shuffles do not occur after all stages.

3. Implementation

3.1 Characterizing and Modeling Imbalance

3.2 Partitioning Strategies

Given the performance model introduced in §3.1, we introduce two separate strategies for data partitioning:

- `segmented`: In this strategy, each segment/nest of the nested array is allocated a single partition of the RDD. Unless the size of each segment is identical, the size of each partition may vary.
- `uniform`: In this strategy, each partition of the RDD is allocated the same number of values; therefore, the partition sizes are uniform. In this strategy, a single segment may be split across one or more partitions.

If the number of segments equals the number of partitions, and all segments have identical length, both strategies are identical and reduce to the `segmented` strategy.

We face the following tradeoffs when picking a strategy for partitioning a dataset:

- *How well will load be balanced?* The `uniform` strategy is highly desirable because we are able to perfectly allocate the keys to partitions, and load is balanced. However, in practice, load balance can be improved by oversubscribing partitions to processing elements (see *Tuning Guide* in [2]; typically, an oversubscription factor of 3 is used). If there is oversubscription, we may be able to co-schedule small and large partitions on the same processing element. While we may have imbalance between each partition, we may be able to achieve balance between processing elements.
- *How much overhead does the strategy incur?* While the `segmented` strategy may have increased imbalance, it has low overhead because we can guarantee that all segment-based operations complete without requiring communication between partitions. This is desirable because the block synchronous model common to MapReduce frameworks implies that all communication requires a barrier.

The additional overhead of the `uniform` partitioning model is limited to the `segmented-scan` operation. To implement a `segmented-scan`, we fall back on the block-parallel `scan` implementation that we introduce in §3.3. There are several additional complications due to the segmented nature of the `scan`; specifically, we must track whether the results generated by previous

partitions are related to this segment. This requires us to track additional information during the *pre-scan* phase, and necessitates filtering when performing the *update* phase.

While it may appear that the `segmented-reduce` operation will also incur additional overhead, this overhead is negligible. For the `segmented` partitioning strategy, each partition performs a reduction across its elements, and then sends the results to the master. The `uniform` strategy performs a keyed reduction—we only reduce together values that are from the same segment. Thus, if a partition contains values from s segments, it will compute s reduced values (one per segment) after the initial partition reduction. We label these values with the segment number, collect the values to the driver, and reduce together values with like labels. If we ensure that the reduction operator is associative and commutative, this approach will be correct. Overhead is limited, as generally $n_p \gg s_p$, where n_p is the number of values in partition p, and s_p is the number of segments with values in partition p.

The structured nature of a nested array provides a significant advantage over unstructured data-parallel models, as it enforces a rigid, known, and efficiently quantifiable order across the data. For unordered datasets, a hash function must be used to partition data in parallel across nodes [8, 9]. Even for ordered datasets, finding an even partitioning is expensive; in Spark, sorting requires the calculation of histogram describing the distribution of keys across the ordering [2]. Since the key distribution is described by the array structure, and a nested array structure can be described by an array of lengths, we can statically track the whole ordering on the master node, instead of needing to recalculate the key distribution before repartitioning.

3.3 Block-Parallel Scans

The `scan`[1] operator is challenging to implement in parallel, as the scan operation on array element a_n depends on the values of all elements $a_0 \dots a_{n-1}$. For clarity, we reproduce the definition of a `scan` given by Blelloch [5] here:

Definition 1. The `scan` operator takes an associative operator $\oplus$, and an ordered set of n elements:

$$[a_0, a_1, \dots, a_{n-1}]$$

and returns the ordered set:

$$[a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})]$$

Blelloch proposed a parallel algorithm for computing vector scans [5]; this algorithm has recently been implemented for GPUs [10]. This algorithm is composed of two basic steps:

1. First, perform an "up sweep" by performing a parallel tree-reduction across the vector.
2. "Down sweep" by shifting in the identity value and rotating the tree elements.

A detailed description of this algorithm, as well as a proof of correctness is presented by both Blelloch [5] and Harris [10]. This algorithm completes in $O(\frac{n}{p} + \log p)$ time when evaluating a vector of n elements on p processors. For $n \gg p$, the runtime is approximated as $O(\frac{n}{p})$. Despite the efficient runtime, the *up sweep, down sweep* algorithm has several drawbacks:

- For a vector of type T, the $\oplus$ operator is restricted to $T, T \mapsto T$. This is an unnecessary restriction if we loosen definition 1 to include an additional input variable b_0, which is the additive identity value. If b_0 has type U, this allows $\oplus$ to become $U, T \mapsto U$. We include this in our updated definition 2.
- The *up sweep, down sweep* algorithm only has $O(\frac{n}{p})$ runtime when executing on systems with a parallel random access memory (PRAM) compute model. As described in §2.2, MapReduce computing systems provide a block synchronous model. For this model, the runtime degrades to $O(\frac{n \log p}{p})$, as we have $\frac{n}{p}$ runtime for each of the $\log p$ stages in both the *up sweep* and *down sweep* stages.

To address the issues we have just presented, we introduce a block-parallel scan:

Definition 2. The block-parallel `scan` operator takes an associative *scan* operator $\oplus$ $(U, T \mapsto U)$, and an associative *update* operator $\otimes$ $(U, U \mapsto U)$, an identity value b_0, and an ordered set of n elements:

$$A = [a_0, a_1, \dots, a_{n-1}]$$

The ordered set A is partitioned into p ordered partitions:

$$\begin{aligned} P^0 &= [a_0, a_1, \dots, a_{\frac{n}{p}-1}] \\ P^1 &= [a_{\frac{n}{p}}, a_{\frac{n}{p}+1}, \dots, a_{2\frac{n}{p}-1}] \\ &\dots \\ P^{p-1} &= [a_{(p-1)\frac{n}{p}}, a_{(p-1)\frac{n}{p}+1}, \dots, a_{n-1}] \\ A &= P^0 \cup P^1 \cup \dots \cup P^{p-1} \end{aligned}$$

For notational convenience, we assume that $n \bmod p = 0$.

For all $P^k, k \in \{0, \dots, p-1\}$, we compute intermediate *block pre-scans* S^k where:

$$\begin{aligned} S_0^k &= b_0 \\ S_i^k &= S_{i-1}^k \oplus P_{i-1}^k, \forall i \in 1, \dots, p \end{aligned}$$

Note that all sets S_k have size $p+1$, not p.

For all $S^k, k \in \{0, \dots, p-1\}$, we then collect the S_p^k terms into set C, where $C_k = S_p^k$. We then perform the *update* phase across all S^k:

$$\begin{aligned} u^k &= \begin{cases} b_0 & \text{if } k = 0 \\ C_0 \otimes \dots \otimes C_{k-1}, & \text{otherwise} \end{cases} \\ U_i^k &= u_k \otimes S_i^k, \forall i \in \{1, \dots, p\} \end{aligned}$$

We return the ordered set:

$$\begin{aligned} S &= U_0 \cup U_1 \cup \dots \cup U_{p-1} \\ S &= [b_0 \oplus a_0, \dots, b_0 \oplus a_0 \oplus \dots \oplus a_{\frac{n}{p}-1}, \dots, \\ &\quad (b_0 \oplus a_0 \oplus \dots \oplus a_{\frac{n}{p}-1}) \otimes (b_0 \oplus a_{\frac{n}{p}} \oplus \dots \oplus a_{2\frac{n}{p}-1}) \otimes \dots \\ &\quad \otimes (b_0 \oplus a_{(p-1)\frac{n}{p}} \oplus \dots \oplus a_{n-1})] \end{aligned}$$

It is not possible to generally prove that the `scan` operator is equal to the block-parallel `scan` operator for all general $\oplus$ operators. However, it is straightforward to prove equivalence for some common operators. For example, given $T = U =$ integer, and $\oplus = + = \otimes$, it follows that $b_0 = 0$. In this case, all of the b_0 terms drop out, and $\oplus = \otimes$, therefore the block-parallel `scan` operation matches the `scan` operation.

Both the per-block *pre-scan* and the *update* operations from definition 2 have runtime $O(\frac{n}{p})$. On the computing platforms we are targeting, the cost of collecting a small amount of data from each partition is negligible. Therefore, the block-parallel `scan` has runtime $O(\frac{n}{p})$. We are able to use the block-parallel `scan` algorithm to perform efficient parallel `scan`s across all datasets, and to accelerate segmented scans when using the `uniform` partitioning strategy introduced in §3.2.

[1] The `scan` operator is also known as the *all-prefix sum* [5].

4. Performance

5. Discussion

6. Conclusion

References

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

[2] Apache Spark. `http://spark.apache.org`.

[3] BERGSTROM, L., FLUET, M., RAINEY, M., REPPY, J., ROSEN, S., AND SHAW, A. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)* (2013), ACM, pp. 81–92.

[4] BERGSTROM, L., RAINEY, M., REPPY, J., SHAW, A., AND FLUET, M. Lazy tree splitting. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)* (2010), ACM, pp. 93–104.

[5] BLELLOCH, G. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*, J. H. Reif, Ed., 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[6] BLELLOCH, G. E. *Vector models for data-parallel computing*, vol. 356. MIT Press Cambridge, 1990.

[7] BLELLOCH, G. E., AND SABOT, G. W. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing (JPDC) 8*, 2 (1990), 119–134.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI '04)* (2004), USENIX Association, pp. 10–10.

[9] DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D. A., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (1990), 44–62.

[10] HARRIS, M., SENGUPTA, S., AND OWENS, J. D. Parallel prefix sum (scan) with CUDA. *GPU gems 3*, 39 (2007), 851–876.

[11] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)* (2007), vol. 41, ACM, pp. 59–72.

[12] NVIDIA. CUDA programming guide, 2008.

[13] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10)* (2010), IEEE, pp. 1–10.

[14] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)* (2012), USENIX Association, pp. 2–2.

[15] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)* (2010), p. 10.