

Original citation:

Li, Zhenyu, Davis, James A. and Jarvis, Stephen A. (2017) An efficient task-based all-reduce for machine learning applications. In: Machine Learning on HPC Environments, ACM New York, NY, USA, 12-17 Nov 2017. Published in: Proceedings of the Machine Learning on HPC Environments (MLHPC'17)

Permanent WRAP URL:

<http://wrap.warwick.ac.uk/95878>

Copyright and reuse:

The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher's statement:

© ACM, 2017. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the Machine Learning on HPC Environments (MLHPC'17) (2017)
<http://doi.acm.org/10.1145/3146347.3146350>

A note on versions:

The version presented here may differ from the published version or, version of record, if you wish to cite this item you are advised to consult the publisher's version. Please see the 'permanent WRAP url' above for details on accessing the published version and note that access may require a subscription.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

An Efficient Task-based All-Reduce for Machine Learning Applications

Zhenyu Li

Zhenyu.Li@warwick.ac.uk
Department of Computer Science
University of Warwick
Coventry, UK

James Davis

J.Davis.4@warwick.ac.uk
Department of Computer Science
University of Warwick
Coventry, UK

Stephen Jarvis

S.A.Jarvis@warwick.ac.uk
Department of Computer Science
University of Warwick
Coventry, UK

ABSTRACT

All-Reduce is a collective-combine operation frequently utilised in synchronous parameter updates in parallel machine learning algorithms. The performance of this operation - and subsequently of the algorithm itself - is heavily dependent on its implementation, configuration and on the supporting hardware on which it is run. Given the pivotal role of all-reduce, a failure in any of these regards will significantly impact the resulting scientific output.

In this research we explore the performance of alternative all-reduce algorithms in data-flow graphs and compare these to the commonly used reduce-broadcast approach. We present an architecture and interface for all-reduce in task-based frameworks, and a parallelization scheme for object-serialization and computation. We present a concrete, novel application of a butterfly all-reduce algorithm on the Apache Spark framework on a high-performance compute cluster, and demonstrate the effectiveness of the new butterfly algorithm with a logarithmic speed-up with respect to the vector length compared with the original reduce-broadcast method - a 9x speed-up is observed for vector lengths in the order of 10^8 . This improvement is comprised of both algorithmic changes (65%) and parallel-processing optimization (35%).

The effectiveness of the new butterfly all-reduce is demonstrated using real-world neural network applications with the Spark framework. For the model-update operation we observe significant speed-ups using the new butterfly algorithm compared with the original reduce-broadcast, for both smaller (Cifar and Mnist) and larger (ImageNet) datasets.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures; Distributed architectures;** • **Computing methodologies** → **Machine learning;**

KEYWORDS

Butterfly All-Reduce; Data-flow Frameworks; Apache Spark; Synchronous Model Training;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MLHPC'17, November 12–17, 2017, Denver, CO, USA

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-5137-9/17/11...\$15.00

<https://doi.org/10.1145/3146347.3146350>

ACM Reference Format:

Zhenyu Li, James Davis, and Stephen Jarvis. 2017. An Efficient Task-based All-Reduce for Machine Learning Applications. In *Proceedings of MLHPC'17*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3146347.3146350>

1 INTRODUCTION

In aggregating a given input vector from different processes, with a user-defined associative and commutative reduction function, the all-reduce operation is able to distribute the combined result to all participating processes. This collection of data is key to a number of high performance computing and data analytic applications.

A simplified view of all-reduce is to split it into two parts: *reduce* and *broadcast*. The reduce process collects a vector from participating processes and combines these into a single value. The broadcast process then takes this result and distributes it to all processes involved.

In machine learning, all-reduce operations are commonly seen in synchronous parameter updates of the distributed Stochastic Gradient Descent (SGD) optimization, which is used extensively in, for example, neural networks, linear regressions and logistic regressions. One such example is AlexNet on ImageNet [13], where every step performs a reduction of weights with an estimated size of 200MB for large model trainings. Moritz [17] reports that with SparkNet the weight-update of AlexNet takes around 20 seconds on a 5-node EC2 cluster, while performing a single mini-batch gradient computation only takes about 2 seconds. In such cases, the overall runtime in distributed neural network training is dominated by communication, highlighting the need for a more efficient all-reduce implementation.

Machine learning algorithms and data-processing applications are commonly implemented on data-analytic frameworks, such as batch-processing and stream-processing frameworks, that are designed for static data and continuous data respectively (see Subsection 2.1). Deep learning libraries, such as Caffe [11], Theano [24] and Torch [25] are single machine implementations, but many have been used with batch-processing frameworks in a distributed environment. Tensorflow [1], DistBelief [6] and Project Adam [4], on the other hand, are distributed deep learning frameworks.

Modern data-analytic frameworks, regardless of batch-processing or stream-processing, share two basic traits: (i) task-based execution that separates memory and computation; and (ii) applications defined in terms of data transformations in data-flow graphs. All-reduce is frequently expressed as a simple *reduce-broadcast* data-flow graph but, as we demonstrate, this is not efficient and is limited by bandwidth at the root process. More efficient all-reduce

algorithms, such as butterfly/distance-doubling and doubling-and-halving [23], use many-stage many-to-many communications, which themselves are highly complex for them to be expressed in a data-flow graph. All-reduce also depends on a number of factors, including the size of the vector, the size of the cluster, network latency, bandwidth, topology, etc., and a hybrid strategy is required for optimal performance.

The Message-Passing-Interface (MPI) includes optimized functions for all-reduce. However, there are fundamental design differences between MPI and task-based frameworks, which mean that MPI cannot be used directly in batch-processing and stream-processing (see Subsection 2.1). As a result, research towards more efficient all-reduce on task-based frameworks is needed.

This paper explores novel, efficient implementations of all-reduce on task-based frameworks. The contributions of this paper include:

- A new general architecture and interface for all-reduce in task-based frameworks, demonstrated via implementation on the Apache Spark framework, the design and results of which are directly transferable to other task-based frameworks;
- A parallelization scheme that enables automatic parallelization of vector computation and serialization, which reduces overheads in object-serialization and computation by 80-90%;
- A novel application of the *butterfly* all-reduce algorithm for the Apache Spark framework that is efficient for very large vector reduction, exhibiting a 9x speed-up compared to the *reduce-broadcast* method for vector lengths of 10^8 on a high-performance cluster;
- A demonstration of the effectiveness of the *butterfly* all-reduce algorithm on real-world neural network applications, where we observe significant speed-ups of model updates using the butterfly algorithm compared with the original *reduce-broadcast* method on small (Cifar and Mnist) and large (ImageNet) datasets.

The remainder of the paper is organized as follows: Section 2 provides background on data-analytic frameworks, all-reduce algorithms and the application of all-reduce in machine learning; Section 3 describes the design and implementation of a new butterfly all-reduce in Spark, which is tested using benchmark tests and through real-world applications in Section 4; Section 5 concludes that the *butterfly* all-reduce has significant performance impact in model updates for distributed machine learning compared with the original reduce-broadcast method. Through this research and further improvements which we propose, we believe that the speed of synchronous training can potentially match asynchronous training in future implementations.

2 BACKGROUND

The goal of this work is to improve the speed at which data can be processed in a task-based framework. However, there are a number of different frameworks and programming models that exist for the purpose of data processing, each with their own advantages and disadvantages. Understanding the design decisions behind the usage of each is key to incorporating new algorithmic approaches to existing processes. We describe several popular frameworks,

and highlight the challenges faced in designing and developing a suitable implementation of the all-reduce operation.

2.1 Data Processing Frameworks

Batch processing is a computing model in which a series of tasks are executed independently with no interaction, and is often employed for processing large volumes of static data. MapReduce [7] is one of the earliest distributed batch-processing frameworks, where each element in the dataset is passed through a *Map-Reduce* pipeline. Data-flow pipelines subsequently developed, providing flexible and sophisticated processing models, include Directed-Acyclic-Graph (DAG) engines in Dryad [10] and Spark [27], as well as the latest Stateful-Directed-Graph engines such as Naiad [18].

Stream processing on the other hand focuses on the real-time processing of continuously generated data. Modern stream processing frameworks such as Apache Storm [26] and Apache Flink [3], share the same task-based data-flow architectures, which can be viewed as batch-processing frameworks with event-triggered processing mechanisms.

The parameter server architecture [22] is a distinct distributed computing model in which globally shared variables are managed by a group of servers that are accessible by a group of stateless workers, which was designed for topic modelling and later applied to neural network training in DistBelief [6] and Project Adam [4]. A special type of tasks called PS tasks in the TensorFlow [1] framework also carries out the same role of parameter servers seen in the other frameworks. Updating a shared variable collaboratively by all workers (i.e., all-reduce) is equivalent to *reduce-broadcast* but with a larger bandwidth at the root process, which depends on the number of machines used as parameter servers. The network bandwidth bottleneck can be partially mitigated by offloading the computation to parameter servers, thus reducing traffic volume by sending the smaller inputs rather than the larger outputs. However, this demands more computing resources on the server side and it is not always possible if the computation contains inter-dependencies.

The Message Passing Interface (MPI) operates in a Single Program, Multiple Data (SPMD) fashion, where each statically allocated process runs a copy of the same program but operates on its own set of data. Running in parallel, these processes can communicate data between one another via the communication interface as and when needed. Task-based frameworks on the other hand, separate memory and computation, running tasks anywhere in the cluster either serially or in parallel. Resource allocation in task-based frameworks is therefore elastic and dynamic, allowing the overall size of the system to grow or shrink according to demand. In addition, tasks can relocate from one machine to another in cases of failure or resource re-allocation.

Due to these differences, there exist two primary reasons why MPI all-reduce cannot be used directly in task-based data-analytic frameworks:

- MPI all-reduce takes a pre-defined data-type and operation as inputs, but the design of data-analytic frameworks permits users to define the data-type and operation.
- Synchronization may cause deadlocks in a task-based framework. Since MPI cannot interact with the task scheduler, and

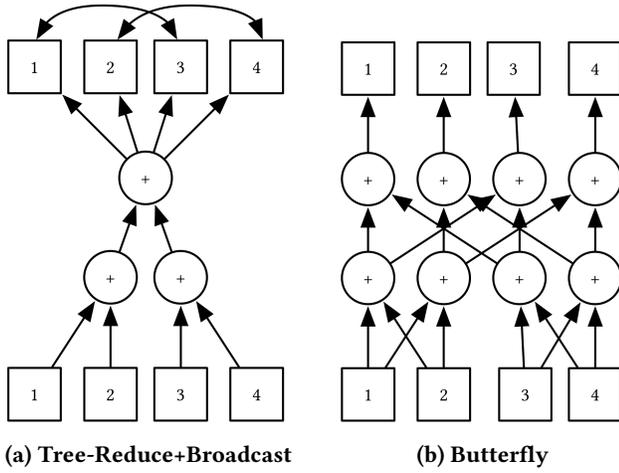


Figure 1: All-Reduce Algorithms

tasks run asynchronously, synchronization operations can cause the application to hang.

As a consequence, the introduction and implementation of blocking operations such as all-reduce can not simply be translated from one high-performance computing framework to another.

2.2 All-Reduce Algorithms

The performance of an all-reduce algorithm depend on many factors, including: (i) the size of the vector; (ii) the size of the cluster; (iii) the nearest number of nodes in a power-of-two; (iv) the network latency and bandwidth and, (v) the network topology (e.g., ring, mesh, torus, hyper-cube, dragonfly, etc.). Classical implementations include the *butterfly* and the *binary-tree* algorithms. This research focuses on *butterfly* algorithms, as algorithmically these take the least number of steps and, they are amenable to modern high-performance interconnects with high bandwidth. A number of previous studies have already made extensive comparisons of all-reduce algorithms for the Message-Passing-Interface (MPI) [20] [21] [23].

Apache Spark implements a simple variant of the *reduce-broadcast* algorithm for all-reduce, which is illustrated in Figure 1a. The *reduction* phase (i.e., bottom half) is a binary-tree reduction process that takes $\lg p$ steps, where p is the number of processes. The *broadcast* phase (i.e., top half) is a process of one-to-all transfer of the initial random data block (default size 4 Mega-Byte), followed by all-to-all shuffle of the rest of the data blocks.

The *butterfly* algorithm is illustrated in the right-hand-side of Figure 1b. In the first step each process exchanges the vector and performs a reduction with a process distance of 1 (i.e., with the neighbouring process), and with each subsequent step the distance doubles. The algorithm takes $\lg p$ steps to complete, where p is the number of processes.

2.3 Theoretical Performance

We compare the *reduce-broadcast* and *butterfly* all-reduce algorithms through theoretical cost estimations using the work of Thakur [23]. Let there be p nodes, with each producing a vector of n bytes after an initial local reduction. γ is the computational cost per byte of locally executing one operation with two operands, and ζ is the serialization or de-serialization cost per byte through a serialization algorithm. Network communication is modelled as linear time by $\alpha + n\beta$, where α is the latency/start-up time per message and β is the transfer time per byte.

Binary-tree reduction takes $\lg p$ steps and, in each step, vectors are fetched and combined by the reduction task; the cost is therefore:

$$T_{tree,red} = \lg p(\alpha + n\beta + 2n\zeta + n\gamma) \quad (1)$$

The communication cost for broadcasting n bytes in $block_size$ blocks is:

$$T_{broadcast} = \frac{n}{block_size}(\alpha + block_size\beta + 2block_size\zeta) \quad (2)$$

The total cost of *reduce-broadcast* is therefore the sum of $T_{tree,red}$ and $T_{broadcast}$.

For *butterfly* all-reduce, there are the same number of steps as a binary tree reduce ($\lg p$), but all nodes fetch and combine in parallel. The cost of *butterfly* all-reduce, assuming a node count of a power-of-two, is therefore:

$$T_{butterfly} = \lg p(\alpha + n\beta + 2n\zeta + n\gamma) \quad (3)$$

In comparison, butterfly all-reduce should be superior if the vector is small, or the bandwidth is large enough such that the linear cost model is still valid.

2.4 Butterfly All-Reduce in Apache Spark

In the early stages of development, it was proposed to implement *butterfly* all-reduce on Spark. However, the idea was rejected because 'the butterfly pattern introduces complex dependency that slows down the computation' [8], and as a result the *reduce-broadcast* approach was adopted as an alternative.

As a result, users employ the less efficient *reduce-broadcast* method provided by Spark, or more efficient custom self-contained Java implementations if available: For example, butterfly mixing [28] is an implementation of *butterfly* all-reduce used by the BIDdata [19] project, which attempts to accelerate incremental optimization algorithms (such as gradient descent) by performing gradient computations at intermediate butterfly stages. However, these are bespoke solutions that assume parallel tasks as MPI processes, which can potentially hang as previously described.

As seen in Sub-section 2.3, *butterfly* all-reduce has a significant performance impact from a theoretical standpoint. Therefore, we seek to implement *butterfly* all-reduce as a shared variable instead of as data-set transformations, to avoid the 'complex dependency' while maintaining good performance.

2.5 All-Reduce in Machine Learning

Many machine learning algorithms can be formulated as an optimization problem to search for the best model, and Stochastic Gradient Descent (SGD) is a popular algorithm for solving the optimisation problem over a large dataset. A distributed implementation of SGD

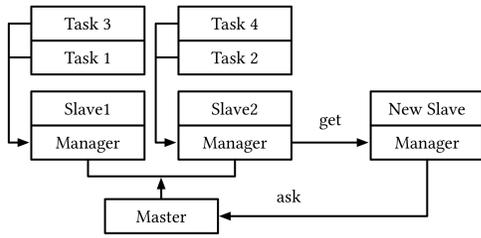


Figure 2: Architecture of task-based all-reduce

averages the model weights across the cluster to incorporate different training examples, which itself is an all-reduce operation.

In many cases, real-world data is very sparse, and much research takes advantage of this fact to accelerate communications. One solution to accelerate the model-update process (i.e., all-reduce) has been to drop 99% of near-zero values and exchange sparse indices of the remaining 1% [2]; this is, in many respects, a compression method. Such an approach has been shown to demonstrate a 50x reduction in communication volume, and a 1.3x speed-up in model training in a neural machine translation system. By dropping the near-zero values, accuracy is lost and the rate of convergence of SGD is degraded. As such, it is only applicable where the values are highly skewed and the lost indices have low-significance.

Kylix [29] is another self-contained Java implementation of all-reduce that attempts to optimize all-reduce for power-law graph data that commonly presents itself in web graphs and social networks, for example. The idea of Kylix is to use heterogeneous-degrees at different layers of a butterfly network, and it is shown that the communication volume in the lower-layer is typically much less than the top layer. Experimental results show a 5x speed-up of Kylix with respect to the binary *butterfly* algorithm in a selection of different test scenarios.

2.6 Asynchronous SGD

Model updates with all-reduce in SGD is a synchronous process that works best for fast convergence, which also limits the speed and scalability of distributed learning. There have been other attempts to accelerate SGD by giving up the synchronous nature and exploiting the tolerance of SGD to noise. Butterfly mixing [28] is just such an example, performing gradient update at intermediate stages of a *butterfly* all-reduce instead of at the end of the all-reduce process.

SparkNet [17] presented a more straight-forward approach by synchronizing the weights every few steps. Project Adam [4] and Tensorflow [1] use a so-called Stale Synchronous Model, where the element updates in a vector are performed by atomic operations without synchronization across all processes. As a result, the values used to compute the weights may not be current and could have been modified by other processes, hence the term ‘stale’.

3 METHODOLOGY

We present an architecture and interface for *butterfly* all-reduce in task-based frameworks, demonstrated through implementation in Apache Spark, the current mainstream task-based data-flow batch-processing framework. Subsections 3.1 & 3.2 introduce the

Algorithm 1 Multi-threaded implementation of the all-reduce manager

```

1: reduced_vector ← empty vector
2: local_submissions ← new Queue
3: procedure LOCALREDUCTION
4:   repeat
5:     new_vector ← Wait for new submission
6:     Lock reduced_vector for reduction
7:     reduced_vector ← Reduce(reduced_vector, new_vector)
8:     Release reduced_vector
9:     Remove(local_submissions, new_vector)
10:  until Global Reduction Is Signalled
11: end procedure
12: function GLOBALREDUCTION
13:   Wait for local reduction to finish
14:   Apply all-reduce algorithm (e.g., butterfly)
15: end function
16: function SUBMIT(new_vector)
17:   local_submissions.add(new_vector)
18:   Signal local reduction thread
19: end function
20: function GET
21:   Wait until global reduction ends
22:   return reduced_vector
23: end function

```

proposed general architecture and user interface used within this work, the design and implementation of which are portable to other task-based batch-processing or stream-processing frameworks. In addition, other opportunities for optimizations are identified and are detailed further in Subsections 3.3 & 3.4.

3.1 All-Reduce Architecture

In contrast to the static parallel processes of a MPI applications, tasks in batch-processing or stream-processing can be allocated dynamically across the cluster. The number of machines available can grow or shrink, with tasks able to run in either serial or parallel and migrate from one machine to another. For a collective operation to function in such a system, the number of participating tasks must be defined prior to the all-reduce action and resume only once the number of committed tasks is reached.

Figure 2 illustrates the architectural structure of this approach. A master process is in charge of task scheduling and maintaining a list of processes participating in the all-reduce. A multi-threaded implementation of the all-reduce manager is presented in Algorithm 1. Each slave process has an independent manager for all-reduce results, with the tasks submitting a vector to their manager as they end; to preserve the data, the managers stay alive within their slave processes. Once all of the participating tasks have finished the all-reduce process can begin, storing the combined results in the all-reduce manager for retrieval by tasks in the next stage. If a task is migrated from one machine to another, whether it is due to task failure or resource re-allocation, a copy of the all-reduce data will be sent to the new slave (*ask* and *get*).

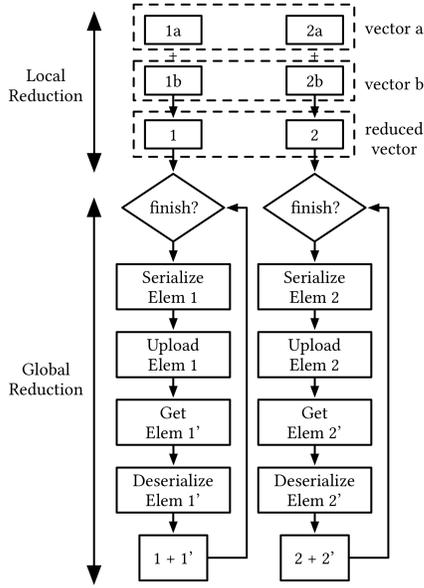


Figure 3: Internal Mechanism of the all-reduce process.
Elem 1: first element/partition in the local vector. Elem 1':
first element/partition in the exchanged vector.

The resulting architecture is suitable for any task-based framework (e.g., batch-processing or streaming-processing), with or without dynamic allocation.

3.2 User Interface

To incorporate the use of all-reduce algorithms other than reduce-broadcast, a simple interface is provided to operate on a shared variable, rather than applying dataset transformations in a data-flow. This is due to the potential use of hybrid schemes with different all-reduce algorithms which, as expressed in Subsection 2.4), are too complex to be efficiently expressed in a data-flow diagram. The API methods are as follows:

- (1) `Init(key, numTasks, func)`: Creates a shared variable for the given key with the number of tasks and a reduction function, the context of all-reduce is maintained by the returned handle;
- (2) `Commit(vector)`: Commits a vector for reduction, the function does not block;
- (3) `Get`: Get the globally reduced vector, block until completion;

In addition to information about the number of tasks, users must also supply a reduction function and all-reduce data in the form of a vector object. The format of the inputs to the function is that of a pair of elements in the vector (i.e., in the form of $C_k \leftarrow A_k + B_k$, instead of $C \leftarrow A + B$), where the elements can simply be sub-vectors in the original vector. The reason for this explicit format is that the reduction function cannot be applied to the sub-elements in parallel, even if a collection type is detected by reflection. By providing the data in this manner, the all-reduce module is able to exploit parallelism to speedup the object-serialization and computation.

3.3 Parallel Processing

Figure 3 depicts the scheme by which the data is processed in a parallelized fashion to speedup the all-reduce operation. As the vector is submitted to the all-reduce manager, the elements are partitioned based on the number of cores available on the node. As the algorithm starts, each partition of the vector goes through the pipeline (i.e., serialization-upload-get-deserialisation-reduction) simultaneously and asynchronously.

Applying the cost analysis described in Subsection 2.3, the cost of parallel butterfly all-reduce becomes

$$T_{butterfly,par} = \lg p(\alpha + n\beta + \frac{2n}{c}\zeta + \frac{n}{c}\gamma) \quad (4)$$

where c is the number of available processors on each node, and other symbols have the same meaning as in Subsection 2.3. In comparison, object serialization and computation are serial in Spark, which poses performance limitations as the vector size grows for larger-scale model training in machine learning. The reasons why it is not parallel are three-fold:

- *Map* and *reduce* have their origin in functional languages, where a function is applied on elements of arbitrary type, and are not forced to be a vector type. Spark preserves such syntax for general usage;
- Parallelisation of the *map* and *reduce* stages is at the object-level, and not at the vector-element level. This is achieved by running multiple tasks in parallel in Spark, which is acceptable if there are enough tasks to occupy the processors. However, in the case of all-reduce, there are far fewer objects for reduction (i.e., one combined vector per node) to allow enough parallel tasks to fully utilize all processors on each node;
- Users can write a parallel version of the reduction function to take advantage of the multi-level resources, but the computation itself is rarely the primary cost factor. As we will see in a demonstration of the neural network training in Sub-section 4.2.3, object serialization is the dominant cost factor, but there is no parallel implementation of the generic serializer. To speed up object serializations of arbitrary type, users must implement a custom parallel serialization method, which involves low-level byte manipulations that is too technical and error-prone even for the most skilled programmers. We solve this conundrum by forcing an input of a vector type, which allows the framework to take care of parallelization without additional user code.

In other words, our vector-based user-interface and parallel-processing scheme provides a finer-grained parallelisation to fully exploit all processing resources, in contrast to the coarse-grained parallelisation in Spark.

3.4 In-Memory Optimisation

In contrast to many task-based frameworks that store intermediate results on disk to release memory pressure and enhance memory tolerance, we keep the update-to-date vector in-memory, which avoids extra I/O overhead. The reason for this is two-fold: (i) all-reduce vectors are relatively small in size compared to the input dataset, and (ii) submitted/exchanged vectors are combined into a

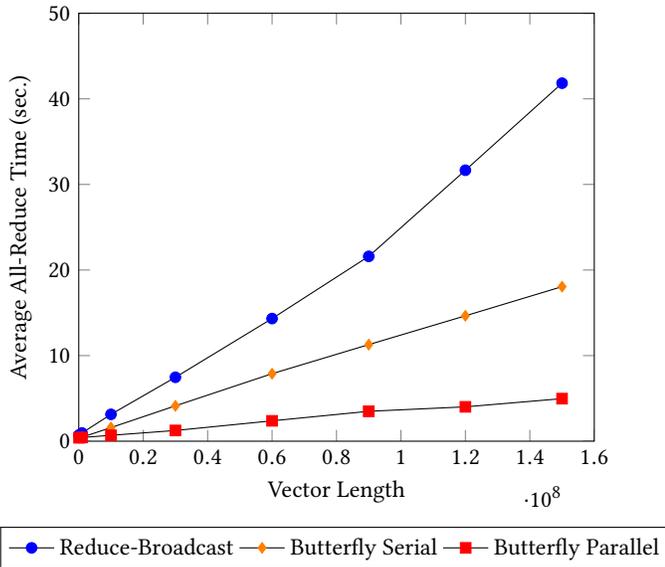


Figure 4: Average All-Reduce Performance on 32 Executors for a Single Iteration

single vector, resulting in a memory usage that does not grow as the number of tasks increases.

4 RESULTS

4.1 Experimental Setup

To evaluate the all-reduce implementations, a simple benchmark and a real-world neural network deployment were tested on a high-performance cluster, the specification of which is detailed in Table 1. Notable features of this hardware include Intel Xeon CPUs, an Infiniband interconnect and the latest install of Apache Spark. We evaluate the performance of all-reduce by comparing

Table 1: Hardware & Software Specification of the Test Cluster

Component	Detail
Nodes	1 Driver Node, 32 Executor Nodes
Cores per Node	20
CPU	Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz
Memory	64GB
Harddisk	Locally Attached (HDD & SSD)
Interconnect	Mellanox Technologies MT26428
Software	Centos/Linux-2.6, Hadoop 2.7, Spark-2.1.1

the resident *reduce-broadcast* and our new implementation of the *butterfly* algorithm. Each executor process runs two tasks in turn, and each task outputs a vector of randomly generated floating point numbers. The length of the vector for reduction ranges from 100,000 to 150,000,000 elements (that is, it has an approximate size of 390KB to 572MB). Experiments are repeated 10 times in 8, 16 and 32 node configurations.

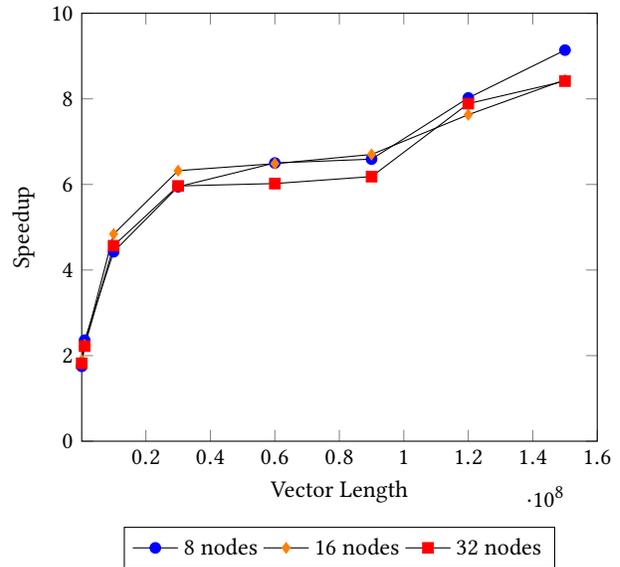


Figure 5: Speed-up of Parallel Butterfly w.r.t Tree-Reduce+Broadcast on 8, 16, 32 nodes

4.2 Empirical Performance

Figure 4 reports the average all-reduce time against the vector size on 32 executors, and Figure 5 reports the relative speed-up of the *parallel-butterfly* algorithm with respect to *reduce-broadcast* in 8, 16 and 32 node configurations. The average all-reduce time exhibits a linear relationship with respect to the vector length. The relative speed-up of the *parallel-butterfly* algorithm exhibits logarithmic growth and becomes saturated at a vector length of 10^7 ; improvements re-gain momentum at 10^8 , signalling traits of the underlying network and supporting protocols.

4.2.1 Reduce-Broadcast and Vector Length. It is observed that the gradient of *reduce-broadcast* starts to grow as the the vector length reaches 10^8 . The same is reflected in Figure 5, where the speed-up should have saturated at 7x for a vector length of $10^7 - 10^8$, but re-surges rapidly after 10^8 . It is evident that the bandwidth bottleneck is reached for the *reduce-broadcast* method at this point.

4.2.2 Butterfly All-Reduce and Cluster Size. Even though the *butterfly* algorithm minimizes the number of steps in the all-reduce, it is still susceptible to network bandwidth limit and contention. In contrast to the *reduce-broadcast* method, we have not seen an increase in steepness in overall all-reduce time in Figure 4 for the *butterfly* all-reduce. Furthermore, the per-stage all-reduce time is stable (i.e., within 0.1 second difference) for the largest vector length of 1.5×10^8 with different cluster setups (i.e., 8, 16 and 32 nodes), as shown in Table 2. As such, we might assume a steady growth in per-stage all-reduce time for the next immediate power-of-2 cluster sizes (i.e., 64, 128 nodes) for vector lengths within 1.5×10^8 .

4.2.3 Breakdown Analysis. Figure 6 reports the breakdown of costs in all-reduce, which is summed over 10 runs and averaged across 32 slaves. The overheads are split into 5 metrics:

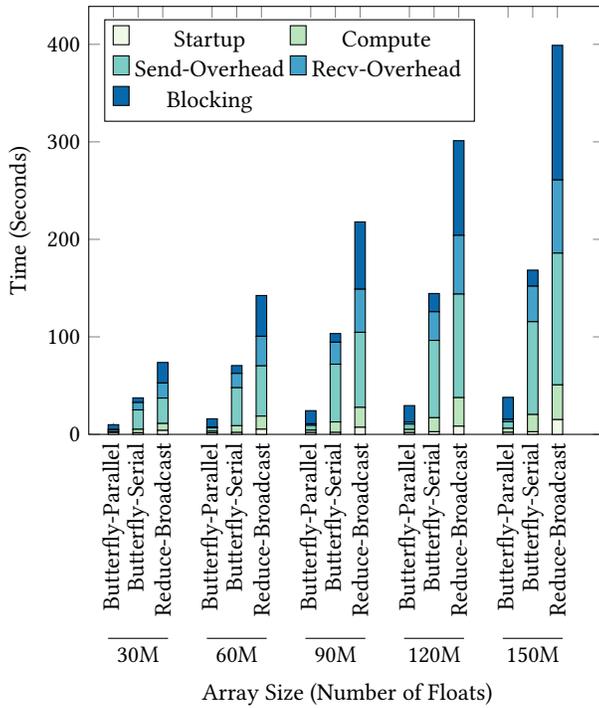


Figure 6: Breakdown of overheads in all-reduce of a large array size for 10 iterations on a 32-node cluster

Table 2: Per Stage Time for Vector Length of 1.5×10^8 for Parallel Butterfly All-Reduce

Nodes	8	16	32
Time	0.95	0.93	0.99

Table 3: All-reduce time in real-world neural network applications across 32 nodes. Original: Reduce-broadcast. New: Butterfly all-reduce.

Dataset	Neural Net	Weight size – log length	Original (sec.)	New (sec.)
Cifar [12]	cuda-convnet [5]	5.2	0.356	0.154
Mnist [16]	LeNet [15]	5.6	0.447	0.184
ImageNet [9]	AlexNet [14]	7.8	17.9	2.4

- (1) Start-Up: Starting up of tasks, including task delivery, serialization/deserialization, etc.;
- (2) Compute: Compute cost of the reduction function;
- (3) Send Overhead: Object serialization (for all), and disk I/O for Spark Shuffle (for reduce-broadcast only);
- (4) Receive Overhead: Object deserialization;
- (5) Blocking: Block time during network transmission of data (for all), and final stage object deserialization at the driver process (for reduce-broadcast only);

By comparing the breakdown components of *serial-butterfly* and *reduce-broadcast*, the network block time in *serial-butterfly* is reduced by 84%, whilst the cost of computation and object serialization are almost identical. The *parallel-butterfly* algorithm further optimizes the compute and object serialization by making use of all available CPU cores. Compute time is reduced by 80-90%, and object serialization (i.e., send overhead + receive overhead) is also reduced by 80-90%, with respect to the serial version.

In summary, algorithmic changes (i.e., *butterfly* against *reduce-broadcast*) and parallel-processing contributes to 65% and 35% of the overall speed-up.

4.2.4 Further Optimization. For *parallel-butterfly*, the major sources of overhead are object serialization (25%) and network blocking (60%). With the impact of object serialization minimized by parallel-processing, network blocking is the only source of further improvements, which can be further reduced by utilizing the native interface of the underlying interconnect architecture such as, for example, Infiniband Verbs/Remote-Direct-Memory Access (RDMA). From this, a theoretical maximum of 2.5x speed-up is obtainable through further optimization or alternative algorithms.

For sparse vector reduction in stochastic gradient descent, there exist other optimization methods such as butterfly mixing [28] and sparse vector compression [2]. Assuming the communication volume can be compressed 50 times by dropping 99% of the near-zero values, an extra 4x speed-up is expected by extrapolation from Figure 4, which can potentially lead to a total of 72x speed-up for vector lengths of 10^8 with respect to the original *reduce-broadcast* method. However, since it also slows down convergence, the actual speed-up for the model to reach the same accuracy may be smaller.

4.3 Applications - Neural Network

Many machine learning algorithms can be formulated as an optimization problem to search for the best model, and Stochastic Gradient Descent (SGD) is a popular algorithm for solving the optimisation problem over a large dataset. For distributed machine learning, a typical parallelisation scheme of SGD averages the weights across the cluster to incorporate updates from different training examples at the end of each step, which requires reduction and re-distribution of the weights (i.e., all-reduce). The overhead in exchanging the model updates limits the scalability of distributed learning, which depends on the complexity of the model. Neural networks are one typical example where the overall performance suffers due to the network exchange of weights at each iterative step.

Cifar, Mnist and ImageNet are three popular datasets in machine learning research, which are also used as examples in SparkNet [17]. We compare the costs of model updates in neural networks with the original *reduce-broadcast* method and the new *butterfly* all-reduce algorithm for these three datasets. The neural-net models and the results for all-reduce are listed in Table 3.

Cifar and Mnist are relatively small datasets compared with ImageNet, and so the neural-net models are therefore small. The model weights for Cifar and Mnist are only 0.2% and 0.6% the size for ImageNet. Nevertheless, a 2.3x speed-up is observed for Cifar and Mnist, and a more notable 7.4x speed-up is observed for ImageNet. The all-reduce times and speed-ups match the projections seen in Figures 4 & 5.

5 CONCLUSION & FUTURE WORK

In this paper we explore novel, efficient all-reduce algorithms and their implementation in task-based, data-analytic frameworks. The aim of this research is to speed up synchronous parameter updates in several machine learning algorithms (for example, linear/logistic regression and neural networks). We present an architecture and interface for all-reduce in task-based frameworks, and a parallelization scheme for object serialization and computation. Testing of the new butterfly all-reduce algorithm is conducted using the Apache Spark framework.

The effectiveness of the *butterfly* algorithm is demonstrated by a logarithmic growth in speed-up with respect to the vector length compared with an existing *reduce-broadcast* method. A 9x speed-up is seen on vector lengths in the order of 10^8 on a 32-node high-performance cluster.

The new butterfly all-reduce algorithm is also tested with respect to the naive *reduce-broadcast* method on model-updates of neural network applications. A 2x and a 7x speed-up are observed for the Cifar and Mnist datasets, and the ImageNet dataset, respectively. We predict stable performance of the *butterfly* algorithm for larger cluster sizes.

By taking advantage of further architectural improvements (for example RDMA) and algorithmic improvements (for example sparse vector compression), we predict that significant speed-ups of all-reduce with respect to the original *reduce-broadcast* method remain obtainable. It is this which motivates our future research.

All-reduce in the context of dynamic scaling of cluster resources is the other thread of our research, and in particular the design of all-reduce algorithms suitable for such architectures. In future research we plan to report on a new hybrid all-reduce algorithm which is in development for architectures with varying number of nodes, network bandwidth and topology.

ACKNOWLEDGMENT

This research is supported by Atos IT Services UK Ltd and by the EPSRC Centre for Doctoral Training in Urban Science and Progress (grant no. EP/L016400/1).

REFERENCES

- [1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA.
- [2] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. *arXiv preprint arXiv:1704.05021* (2017).
- [3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering* 38, 4 (2015).
- [4] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *OSDI*, Vol. 14. 571–582.
- [5] cuda convnet. n.d.. <https://code.google.com/archive/p/cuda-convnet/>. (n.d.). [Online; Accessed 01-August-2017].
- [6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
- [7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
- [8] Spark Github. 2014. [WIP][SPARK-1485][MLLIB] Implement Butterfly AllReduce. <https://github.com/apache/spark/pull/506>. (2014). [Online; Accessed 01-August-2017].
- [9] ImageNet. n.d.. <http://www.image-net.org/>. (n.d.). [Online; Accessed 01-August-2017].
- [10] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.
- [11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).
- [12] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. n.d.. CIFAR-10 dataset. <http://www.cs.toronto.edu/~kriz/cifar.html>. (n.d.). [Online; Accessed 01-August-2017].
- [13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
- [14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates Inc., USA, 1097–1105. <http://dl.acm.org/citation.cfm?id=2999134.2999257>
- [15] Yann Lecun. n.d.. LeNet-5, convolutional neural networks. <http://yann.lecun.com/exdb/lenet/>. (n.d.). [Online; Accessed 01-August-2017].
- [16] Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. n.d.. The MNIST dataset. <http://yann.lecun.com/exdb/mnist/>. (n.d.). [Online; Accessed 01-August-2017].
- [17] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. 2015. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051* (2015).
- [18] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.
- [19] BID Data Project. n.d.. <http://bid.berkeley.edu/BIDdata/overview/>. (n.d.). [Online; Accessed 01-August-2017].
- [20] Rolf Rabenseifner. 2004. Optimization of collective reduction operations. In *International Conference on Computational Science*. Springer, 1–9.
- [21] Rolf Rabenseifner and Jesper Larsson Träff. 2004. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 36–46.
- [22] Alexander Smola and Shравan Narayanamurthy. 2010. An architecture for parallel topic models. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 703–710.
- [23] Rajeev Thakur and William D Gropp. 2003. Improving the performance of collective operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 257–267.
- [24] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). <http://arxiv.org/abs/1605.02688>
- [25] Torch. n.d.. <http://torch.ch/>. (n.d.). [Online; Accessed 01-August-2017].
- [26] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.
- [27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. 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*. USENIX Association, 2–2.
- [28] Huasha Zhao and John Canny. 2013. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 785–793.
- [29] Huasha Zhao and John Canny. 2014. Kylix: A sparse allreduce for commodity clusters. In *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 273–282.