An Elastic, Parallel and Distributed Computing Architecture for Machine Learning

by

Anthony Zhenyu Li

Thesis

Submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

September 2019

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgments

The completion of this thesis would not have been possible without the help and support from all the staffs and colleagues at the University of Warwick.

To start, I would like to express my sincere gratitude to Prof. Stephen Jarvis, for providing me with the opportunity to undertake a PhD and guiding my research and professional development over the past four years. I would also like to thank my advisor James Davis, who offered great help in developing my academic writing, and together, we co-authored several publications.

I would also like to thank all members of the Warwick Institute for the Science of Cities (WISC), who made life more enjoyable as a researcher, among which, my office mates John Rahilly, David Purser, Ian Tu in particular, for their friendships and support inside and outside of work. A special thank you goes to our administrator Yvonne Colmer for looking after everybody at WISC both professionally and personally.

Lastly, I would like to thank my family and my girlfriend Elena for their continued love and support.

Declarations

Parts of this thesis have been previously published by the author in the following:

[50] Zhenyu Li, James Davis, and Stephen Jarvis. Optimizing machine learning on Apache Spark in HPC environments. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 95–105, Dallas, TX, USA, 2018. IEEE. ISBN 978-1-7281-0180-4. doi: 10.1109/MLHPC.2018.8638643. URL `http://dx.doi.org/10.1109/MLHPC.2018.8638643`

[48] Zhenyu Li and Stephen Jarvis. MapRDD: Finer grained resilient distributed dataset for machine learning. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR'18, pages 3:1–3:9, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5703-6. doi: 10.1145/3206333.3206335. URL `http://doi.acm.org/10.1145/3206333.3206335`

[49] Zhenyu Li, James Davis, and Stephen Jarvis. An efficient task-based All-Reduce for machine learning applications. In *Proceedings of the Machine Learning on HPC Environments*, MLHPC'17, pages 2:1–2:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5137-9. doi: 10.1145/3146347.3146350. URL `http://doi.acm.org/10.1145/3146347.3146350`

Abstract

Machine learning is a powerful tool that allows us to make better and faster decisions in a data-driven fashion based on training data. Neural networks are especially popular in the context of supervised learning due to their ability to approximate auxiliary functions. However, building these models is typically computationally intensive, which can take significant time to complete on a conventional CPU-based computer. Such a long turnaround time makes business and research infeasible using these models. This research seeks to accelerate this training process through parallel and distributed computing using High-Performance Computing (HPC) resources.

To understand machine learning on HPC platforms, theoretical performance analysis from this thesis summarises four key factors for data-parallel machine learning: convergence, batch size, computational and communication efficiency. It is discovered that a maximum computational speed-up exists through parallel and distributed computing for a fixed experimental setup.

This primary focus of this thesis is convolutional neural network applications on the Apache Spark platform. The work presented in this thesis directly addresses the computational and communication inefficiencies associated with the Spark platform with improvements to the Resilient Distributed Dataset (RDD) and the introduction of an elastic non-blocking all-reduce. In addition to implementation optimisations, the computational performance has been further improved by overlapping computation and communication, and the use of large batch sizes through fine-grained control. The impacts of these improvements are more prominent with the rise of massively parallel processors and high-speed networks.

With all the techniques combined, it is predicted that training the ResNet50 model on the ImageNet dataset for 100 epochs at an effective batch size of 16K will take under 20 minutes on an NVIDIA Tesla P100 cluster, in contrast to 26 months on a single Intel Xeon E5-2660 v3 2.6 GHz processor.

Due to the similarities to scientific computing, the resulting computing model of this thesis serves as an exemplar of the integration of high-performance computing and elastic computing with dynamic workloads, which lays the foundation for future research in emerging computational steering applications, such as interactive physics simulations and data assimilation in weather forecast and research.

Sponsorships and Grants

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).

Acronyms

AVX Advanced Vector Extensions.

BSP Bulk Synchronous Parallel.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DA Data Analytics.

DAG Directed Acyclic Graph.

DL Deep Learning.

DMA Direct Memory Access.

FLOPS Floating-point Operations Per Second.

FPGA Field Programmable Gate Arrays.

FPU Floating-Point Unit.

GPU Graphical Processing Unit.

HPC High-Performance Computing.

IP Internet Protocol.

IPoIB Internet Protocol over InfiniBand.

JVM Java Virtual Machine.

LARS Layer-wise Adaptive Rate Scaling.

MAP Maximum A-posteriori Probability.

ML Machine Learning.

MLE Maximum Likelihood Estimate.

MMX MultiMedia eXtension.

MPI Message Passing Interface.

PCI Peripheral Component Interconnect.

RDD Resilient Distributed Dataset.

RDMA Remote Direct Memory Access.

SGD Stochastic Gradient Descent.

SIMD Single Intruction Multiple Data.

SPMD Single Program Multiple Data.

SSE Streaming SIMD Extensions.

SSP Stale Synchronous Parallel.

TCP Transmission Control Protocol.

WRF Weather Research and Forecasting.

YARN Yet Another Resource Negotiator.

Chapter 1

Introduction

Machine learning is playing an increasingly important role in all aspects of today's society, for example, making strategies in business, fraud detection in banking services, robotics in manufacturing and explanatory explorations, research and discovery in all branches of science, self-driving cars and AI-assistants in our homes and mobile devices. It helps to make quicker and better data-driven decisions and replaces repetitive and redundant tasks with a virtual workforce.

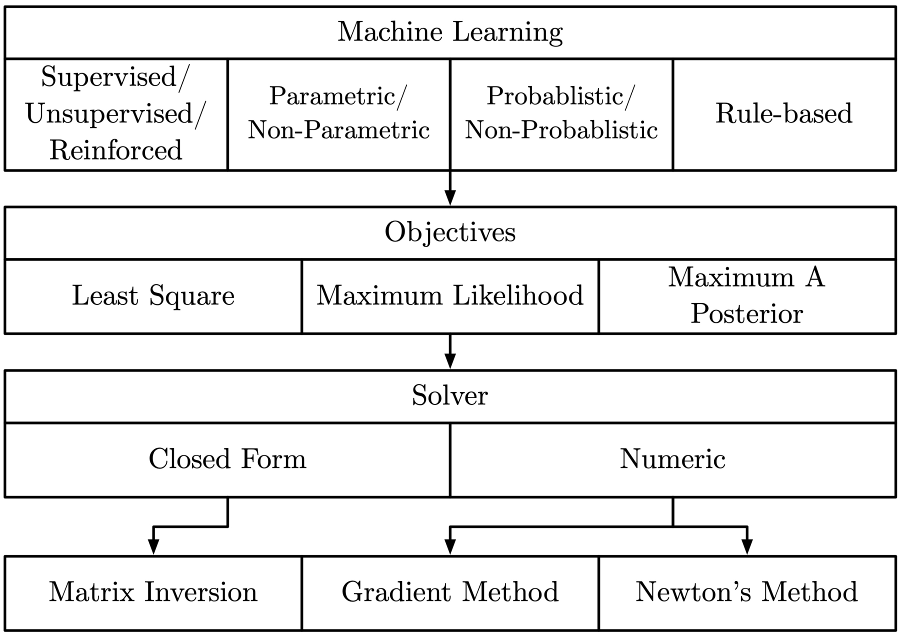


Figure 1.1: A taxonomy for machine learning algorithms

Machine learning is the process of learning an unknown function $f(X) \rightarrow Y$ that transforms input X to output Y. There are different categories of machine learning algorithms: supervised/unsupervised/reinforced, parametric/non-parametric, probabilistic/non-probabilistic and rule-based, etc., as shown in Figure 1.1. These categorisations are not mutually exclusive. Parametric models are the main focus of this research, which assumes a certain form of the unknown function $f(X)$, for example, $Y = aX + b$ in linear regression, where a and b are the parameters. A parametric model places constraints on the problem, which limits its complexity and could result in a poor fit if the assumption of the structure of the unknown function $f(X)$ is wrong; however, it simplifies the problem and makes it easier to learn and interpret.

Different algorithms define different objectives and the most common objective is to minimise the sum of squared errors, which is as known as the 'least squares' method. For a linear least squares problem, there is a closed form analytic solution that involves matrix inversion which has a best–case complexity of $O(n^{2.373})$ for a matrix of size $n \times n$ with the optimised Coppersmith—Winograd algorithm [80]. But the matrix may not be invertible, in which case the analytic solution does not exist. For a non-linear least squares problem, there is no analytic solution in general. As a result, numerical methods are favoured for solving large least squares problems.

Gradient descent and its variants (i.e. batch gradient descent, mini-batch gradient descent and stochastic gradient descent) are the standard methods for solving the least squares problem numerically and it is implemented in the common machine learning libraries (e.g., Caffe [35], Pytorch, Tensorflow [1]). The alternatives are Newton's method and its derivatives, the Iteratively re-weighted least squares method and the limited-memory BFGS method. In this research, we study the Stochastic Gradient Descent (SGD) method extensively as it offers a faster computation compared with a second order optimiser such as the L-BFGS method. It is also more suitable for working with large datasets compared with the classical gradient descent method, the difference between the two is that SGD only takes one sample at each iteration instead of the entire dataset, but in practice, a small mini-batch of samples is used due to computational efficiency limitations.

1.1 Motivation

Neural network algorithms are a type of parametric model, which is generally regarded as the 'universal function approximator'. The perceptron algorithm is the simplest form of a neural network, which takes a weighted sum of the inputs and feeds it to an activation function Φ with a threshold Θ, and produces an output of form $\Phi(\Theta, \sum_{i=0}^{n} w_i x_i)$ where w_i is the weight for input x_i. The multilayer perceptron, as the name suggests, consists of multiple layers of perceptrons, which can be regarded as multiple stages of processing of information, and it is the most common type of deep learning architecture.

Convolutional neural networks are the state-of-the-art method for object detection tasks in computer vision. These neural network models generally have millions of parameters, and the more classes of objects to be trained, the more training data is required. It often takes days, months, even years, to train such a model on a single computer even with accelerated co-processors. Table 1.1 lists the most prominent neural network models in recent years in chronological order, which includes the computational costs, number of parameters and the training time on an Intel Xeon E5-2660, NVIDIA Tesla K80 and an NVIDIA Tesla P100 processor (note: the training time for NVDIA GPUs are estimated from the Tensorflow Benchmark [76]). It is shown that these models take years to train on a high-performance CPU, or days to months on even the most advanced graphical co-processors.

A turnaround time of months or years is not feasible for business or research development. The iterative development model provides a way of mitigating risks by putting the development process under a constant 'requirement-design-implement-evaluation' loop, a long turnaround time renders it impossible. The waterfall development model is the alternative, but this takes a much higher risk as there is no guarantee of how well the model will perform on the given problem. For projects that are highly dependent on the machine learning model, it can also cause stagnation of the entire operation. Therefore, a long turnaround time leads to a high risk of project failure and the acceleration of the training process is greatly needed.

Table 1.1: A compilation of neural network models in terms of computational cost (multiply-add operations), number of parameters and training time.

	AlexNet [41]	VGG16 [26]	InceptionV3 [75]	ResNet50 [69]
year	2012	2014	2015	2015
multiply-adds (million)	724	15500	5000	3900
parameters (million)	61	138	25	25.5
Time (months, ImageNet, 100 epochs, Intel Xeon E5-2660 2.6 GHz)	5	103	33.6	26.4
Time (days, (ImageNet, 100 epochs, NVIDIA K80)	3	47	53	31
Time (days, (ImageNet, 100 epochs, NVIDIA P100)	0.7	14	4.6	3.6

1.2 Parallel & Distributed Machine Learning

To accelerate gradient–based training, the standard method is to deploy the same program on a plurality of computers simultaneously with different input and aggregate the results, by *data parallelism* or *model parallelism* (an introduction to data and model parallelism can be found in Section 2.1.2). This can be achieved with simple batch scripting with networking libraries, or utilising existing distributed computing frameworks such as Message Passing Interface (MPI), or data processing frameworks such as Hadoop Map-Reduce and Apache Spark. Cluster machines are a type of resource, commonly encountered in the field of High-Performance Computing (HPC), that are often comprised of powerful computers connected via high-speed interconnects. Leveraging these resources provides a valuable opportunity to improve the training times of distributed machine learning techniques.

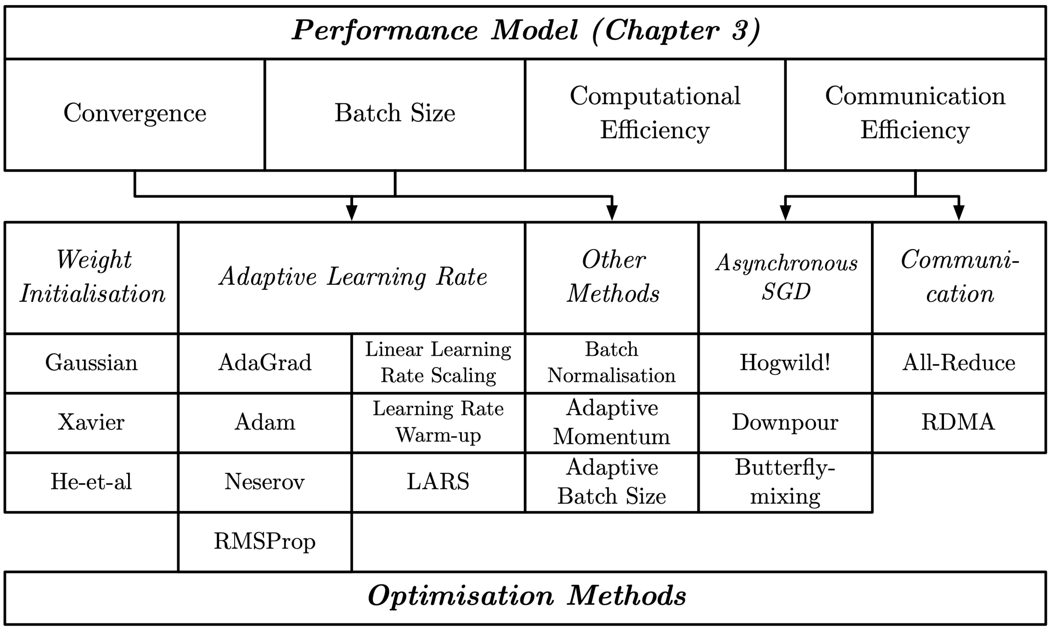


Figure 1.2: A taxonomy of the performance of distributed stochasitc gradient descent

The performance of stochastic gradient descent (SGD) on a standalone machine is mainly concerned with the convergence, which aims at reducing the number of training samples to reach as high as accuracy as possible. A lot of research contributes to the convergence of stochastic gradient descent with the use of momentum and adaptive learning rate. For example, AdaGrad [15], Adam [38], Nesterov [60], and RMSProp [78].

For distributed SGD, communication and synchronisation penalties have to be taken into account. Strong scaling of distributed training is a difficult problem, as the cost of synchronisation and communication overtakes the computational cost as the number of processors increases. Performance evaluations on distributed training are usually empirical, which are not transferable from one machine to another; therefore users cannot make an informed decision on the optimal settings. We provide a mathematical tool/performance model in Chapter 3 for estimating the time, cost and speed-up that allows users to predict likely performance on their cluster machines. We summarise four performance factors in distributed stochastic gradient descent: (i) convergence rate; (ii) batch size; (iii) computational efficiency; (iv) communication efficiency; the relationship to the taxonomy is shown in Figure 1.2.

The communication cost can be reduced, either by using more efficient communication algorithms (e.g. *butterfly* and *ring* all-reduce), hardware acceleration (e.g. zero memory-copy via remote direct memory access) or asynchronous methods (e.g. the Hogwild algorithm [65] and the Downpour algorithm [12]).

Due to the continuous growth of processor speed, the use of bigger batch sizes becomes inevitable as predicted by our performance model (see Equation 3.7). However, an increase in the batch size leads to a slow-down of convergence as they are inversely correlated; therefore, it is hard for them to be increased simultaneously. A number of techniques have been employed for large batch size training to mitigate the effect of growing batch size on the convergence rate, which includes: (i) linear learning rate scaling; (ii) learning rate warm-up; (iii) layer-wise adaptive rate scaling (LARS); (iv) incremental or adaptive batch size.

The learning rate in SGD controls the step size of each update, which starts from a base value and decreases gradually as the current position approaches the saddle point (i.e. the minimum position) to avoid divergence. As observed empirically, a large batch size has regulatory effects and increasing the batch size is equivalent to decreasing the learning rate [73], as such, a larger batch size can be employed as the training progresses. Successes have been demonstrated in scaling up the batch size to 64K-100K in [3] [21] [13] [73] and [81]. The problem with existing methods for dynamic batch-sizing is the **changing problem size**, in contrast to static partitioning on distributed platforms. Static partitioning poses a limit for the problem size due to scaling effects that result in coarse batch size control (for example: doubling the batch size every 10, 20 epochs). By adopting elastic computing, restriction on the problem size can be lifted which allows for the dynamic batch-sizing method in a fine-grained manner.

1.3 Elastic & Cloud Computing

A dynamic batch size in stochastic gradient descent results in a changing problem size that is similar to the changing workload encountered in data analytic applications, which poses a new challenge in distributed machine learning. The biggest difference between data analytics and scientific computing is the difference in workload, therefore resulting in necessarily different computing platforms. Data analytics generally has a data-dependent dynamic workload/problem size, while scientific computing typically has a constant problem size. A typical case for dynamic workload can be demonstrated by the simple *'Map-Reduce'* data-flow, where data elements pass through a *Map* function and the intermediate results are subsequently combined by a *Reduce* function. The *Map* stage typically requires more processing resources than the *Reduce* stage.

Scientific computing platforms (i.e. simple batch scripting, Message Passing Interface (MPI)) do not support applications of a dynamic nature, and there are three main challenges: (i) dynamic allocation; (ii) dynamic coordination; (iii) task migration. Scientific applications are statically partitioned and elastic allocation is not supported by the cluster scheduler, because the large proportion of such applications have a constant problem size throughout the life-span of the application.

In contrast to scientific computing platforms, **elastic computing** platforms are designed for adapting to the change in workload or problem size during the execution of the application. The most common software platforms for elastic computing are data-flow frameworks and parameter servers; they have a master-slave architecture that allows for dynamic coordination, but they have severe drawbacks in performance and overhead. For frameworks using parameter servers, such as Tensorflow, cluster resources are not only statically allocated but also must be manually defined, which is not practical for large-scale deployments.

Cluster computers have long been scarce resources available only to large organisations. Thanks to the advert of cloud computing, cluster computing resources are now widely available as a service, and are highly popular in machine learning. **Cloud computing** allows customers to hire computing resources whenever needed. Providers such as Amazon EC2 and Microsoft Azure can deliver high-performance computing resources on demand, which allows small business and home developers to use super-computing resources at an affordable price. This gives more incentives and driving force behind the need for an elastic computing platform for machine learning which currently does not exist.

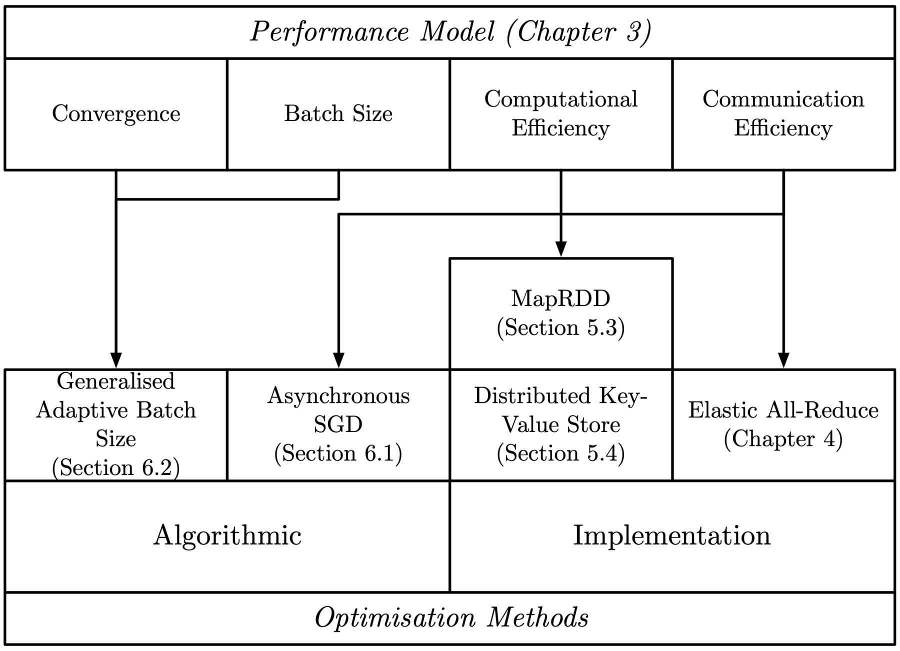
The performance of distributed applications can be explained by two notions of scalability in HPC: strong and weak scaling. **Strong scaling** refers to how the solution time changes with the number of processors for a *fixed total problem size*, as defined by Amdahl's law [4]. **Weak scaling** refers to how the solution time changes with the number of processors for a *fixed problem size per processor* (i.e. the total problem size changes), as defined by Gustafson's law [24]. An introduction to Amdahl's law, Gustafson's law and scalability of high-performance computing, in general, can be found in Section 2.3. For traditional scientific computing, the total problem size is fixed and therefore Amdahl's law/strong scaling applies. For data dependent applications, such as batch processing of static data, stream processing of continuous data or emerging new applications with unknown total problem size, the workload changes and so does the amount of computing resources, and as such weak scaling applies.

Neural network applications are more akin to scientific computing as they mainly consist of matrix–matrix multiplications as explained in Section 2.1.2. The batch size defines the total problem size for SGD training, which is typically constant throughout the process. For a constant batch size, scientific platforms are a better candidate for neural network applications. However, with the rise of using dynamic batch-sizing (i.e. incremental or adaptive) for large batch-size training, the total problem size changes dynamically, for which data-flow platforms for elastic computing are a better candidate. The implementation and experimentation in this research are performed on the Apache Spark framework - an elastic data-flow framework, which is further introduced in Section 2.3.4.

1.4 Thesis Contributions

In this research, we choose Apache Spark as the base platform due to the dynamic nature of the architecture: (i) It allows 'closure' functions (i.e. functions without side-effects) to be executed anywhere in the cluster (ii) It can work with elastic cluster schedulers such as YARN and Mesos, and it has its own standalone scheduler that can work independently; (iii) It is based on master-slave architecture, which allows elastic and dynamic resource allocation and coordination. However, there are well-known performance bottlenecks for Spark: (i) No persistent memory; (ii) Synchronous and exhaustive memory management; (iii) Inefficient data-flow pattern for communications. We will explain how we overcame these issues in Chapters 4 and 5 for communication and memory management respectively. We also choose Convolution Neural Network (CNN) as an exemplar for general machine learning, because training a CNN is both computational and communication intensive, as such, the methods used in training CNNs are expected to be transferable to training other parametric models.

The research presented in this thesis makes the following contributions, which correspond to the performance factors in distributed stochastic gradient descent, and are mapped in Figure 1.3:



Performance Model (Chapter 3)
Convergence
Batch Size
Computational Efficiency
Communication Efficiency
MapRDD (Section 5.3)
Generalised Adaptive Batch Size (Section 6.2)
Asynchronous SGD (Section 6.1)
Distributed Key-Value Store (Section 5.4)
Elastic All-Reduce (Chapter 4)
Algorithmic
Implementation
Optimisation Methods

Figure 1.3: A map of the research contributions to performance factors demonstrated in the research presented in this thesis.

1. A computational performance model for data-parallel distributed stochastic gradient descent, which allows predictions for running time and cost of distributed training and can be used as a mathematical tool to evaluate optimisation methods analytically. We provide a road-map for systematic improvements in accordance to the performance factors as shown in Figures 1.2 and 1.3 for existing methods and also for the contributions of this research respectively. We provide an analysis for the performance of distributed SGD in terms of strong and weak scalability in HPC, which demonstrates a maximum computational speed-up for a fixed experimental setup.

2. A new general architecture and interface for all-reduce in elastic 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. We propose a parallelisation scheme that enables automatic parallelisation of vector computation and serialisation, which reduces overheads in object-serialisation and computation by 80-90%. We demonstrate: (i) A novel application of the *butterfly* all-reduce algorithm for the Apache Spark

framework that is efficient for very large vector reduction, exhibiting a 9× speed-up compared to the *reduce-broadcast* method for vector lengths of 10^8 on a high-performance cluster; (ii) An up to 18× further speed-up for the *butterfly all-reduce* algorithm by zero-copy via remote direct memory access; (iii) 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 all-reduce* algorithm compared with the original *reduce-broadcast* method on small (CIFAR and MNIST) and large (ImageNet) datasets.

3. The design and implementation of the new *MapRDD*, which exploits the record-wise relation between the parent and child datasets during *map* transformations, and permits random-access to individual records in the child dataset through computing the chain of dependent records. We present the implementation of a new MemoryStore for the new MapRDD, which organises the dataset at the record level, and manages data sampling and data transfers asynchronously. We use the ImageNet dataset to demonstrate that the initial data loading can be eliminated by comparing the sampling performance with the original MapPartitionsRDD and the new MapRDD; the CPU processing cycles and memory usage can be reduced by more than 90%, allowing other applications to be run simultaneously. We train AlexNet [41] with the ImageNet dataset on an NVIDIA Tesla K80 GPU. We demonstrate that the data sampling and data transfer can be totally overlapped with training on the GPU with the new asynchronous MemoryStore. We demonstrate a 4× speedup for up to 20% of the ImageNet dataset, in GPU training with the new MapRDD and the new MemoryStore, compared with the original MapPartitionsRDD. We also show a constant training step time with the new MapRDD, regardless of the size of the partition, for up to 1.3 million records in the ImageNet dataset.

4. The design of a distributed key-value store for management of the machine learning model, which allows for memory persistence and flexibility. The correctness of computation is ensured by incremental versioning and checksum, and the fault tolerance is ensured by checkpoints. We show that the resulting hybrid computing model (all-reduce + key-value store) is functionally equivalent to the parameter server architecture but more computationally efficient.

5. A new asynchronous SGD algorithm with non-blocking all-reduce, for reducing the communication cost by overlapping computation and communication. Through experiments on AlexNet-ImageNet, we demonstrate comparable convergence rate with the new algorithm compared to the standard synchronous method, using the Student-t statistic analysis. We show that a further 2×

speedup is obtainable with respect to the synchronous method using the same problem configuration, as predicted by the performance model developed in this thesis.

6. A new dynamic fine-grained batch size control method for large batch size SGD training. We propose a generalised polynomial control formula and we devise six control experiments for each of the hyper-parameters. Through experiments on AlexNet-ImageNet, the new dynamic batch-sizing method consistently demonstrates a faster convergence and higher validation accuracy than the static method with equivalent batch size. The results of the control experiments show that the validation accuracy is sensitive to the initial batch size, which leads a trade-off between the accuracy and computational speed.

1.5 Thesis Overview

The remainder of the thesis is structured as follows:
Chapter 2 provides general background on: (i) Neural networks algorithms and their computational and space complexities in Section 2.1; (ii) Parameter fitting for parametric models and the stochastic gradient descent algorithm in Section 2.2; (iii) High-performance computing and the implementation of distributed machine learning in Section 2.3.
Chapter 3 presents the theoretical analysis on the computational performance and scalability of data-parallel SGD. Equations are derived for running time, running cost and speed-up for both synchronous and asynchronous SGD, as well as the equations for maximum speed-up. This corresponds to contribution 1.
Chapter 4 introduces and compares different 'all-reduce' algorithms (e.g. *butterfly*, *ring* and *reduce-broadcast*) for parameter aggregation in distributed SGD and how it is implemented in Spark and other distributed platforms, and describes an adaption of 'all-reduce' for elastic task-based systems. The new elastic all-reduce method shows significant speed-up via algorithmic changes and optimisation through parallel processing and remote direct memory access, compared with the original 'reduce-broadcast' method in Spark. This corresponds to contribution 2.
Chapter 5 provides in-depth view for RDD - immutable memory abstract for Apache Spark and discussions of its drawbacks in the context of machine learning, and proposes two alternative designs and implementations: (i) *MapRDD* for efficient sub-sampling immutable data; (ii) A distributed key-value store to overcome the memory persistence and flexibility issues of RDD. This corresponds to contributions 3 and 4.
Chapter 6 presents two algorithmic improvements for SGD: (i) An asynchronous

SGD using non-blocking all-reduce in Section 6.1; (ii) A fine-grained batch size control method for large batch size training in Section 6.2. This chapter includes convergence and running time analysis for both methods, which demonstrates: (i) Comparable validation accuracy and double speed-up using asynchronous SGD compared with the standard method; (ii) Higher validation accuracy and comparable speed-up using dynamic batch-sizing compared with the standard method. This corresponds to contributions 5 & 6.

Chapter 7 concludes the thesis, and discusses the limitations of this research. We provide an analysis of the future trends of machine learning and high-performance computing developments and outline future research on an elastic high-performance computing platform for computational steering applications.

Chapter 2

Background

This chapter provides a general introduction to neural network algorithms, the backbone to machine learning - stochastic gradient descent, and their implementation on a parallel and distributed computing systems.

2.1 Background - Deep Learning

Machine learning is the process of learning an unknown function $f(X) \rightarrow Y$ that transforms input X to output Y. Machine learning algorithms can be categorised into supervised (classification), unsupervised (clustering) and reinforced learning. Supervised learning takes an input and tries to predict the correct label; the quality of classification depends on the quality of the training data. High quality labelled data is hard to come by and costly to produce, as it may require experts to manually label the data. In the absence of labelled data, unsupervised learning tries to group data with similar attributes together to form a concept, which can be later defined by a human expert. Reinforced learning is a distinct machine learning paradigm from supervised or unsupervised learning that allows an agent to learn from trial-and-error, but it requires an interactive environment where rewards and penalties can be given to the agent.

Supervised learning allows for parameters to be adjusted in order to meet the target values, and it is the most common task in computer vision, speech recognition, etc. Classical supervised learning algorithms include decision trees, rule-based classifiers, statistical and probabilistic classifiers (e.g., linear classifier, Naïve–bayes, Bayesian belief networks, Gaussian processes, etc.) and the popular neural network algorithms.

Neural networks are a parametric supervised method that is the most popular machine learning model, for its ability to approximate any function $f(X) \rightarrow Y$. Its variants such as Convolutional Neural Networks (CNNs) and Recurrent Neural

Networks (RNNs) have gained huge success in computer vision and natural language processing tasks.

Unlike traditional classifiers that require hand-extracted features, CNNs are an 'end-to-end' approach, which take the input and produce the output in a single algorithm. It is comprised of a feature extraction phase and a classification phase: the former consists of multiple stages of convolution layers, and the latter consists of fully connected multilayer perceptrons. This is often called a **deep learning** architecture due to this multi-stage structural learning nature.

The rest of this chapter provides an introduction to the perceptron algorithm and convolutional neural network algorithm, including how it is implemented in linear algebra, and the computational and space complexity analysis.

2.1.1 Perceptron & Backpropagation

The perceptron algorithm is the simplest neural network algorithm, which takes the weighted sum of the inputs ($\sum_{i=0}^{n}(x_i w_i)$) as an argument for an activation function φ with a threshold Θ, as illustrated in Figure 2.1. A multilayer perceptron consists of multiple layers of neurons: an input layer, an output layer, and at least one hidden layer.

The back-propagation algorithm is a class of automatic differentiation algorithms that provides the derivatives ($\frac{\delta E}{\delta w_{i,j}}$) of the prediction error (E) with respect to individual parameters ($w_{i,j}$), where $w_{i,j}$ indicates the weight for input i in layer j, using the chain rule in an iterative and recursive fashion. This transforms the learning process to an optimisation problem that can be solved by numerical methods such as gradient descent, as explained in Section 2.2.

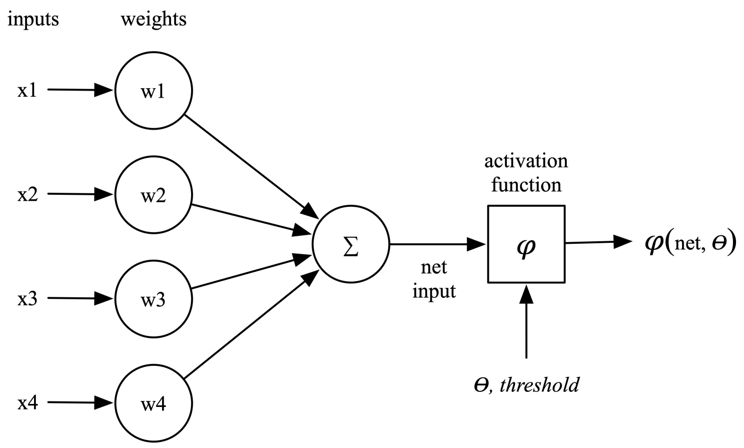


Figure 2.1: An illustration of a neuron in the perceptron algorithm

2.1.2 Convolutional Neural Network

A Convolutional Neural Network (CNN) is an artificial neural network architecture with multiple convolutional layers built upon a traditional multilayer perceptron. In signal processing, an individual signal itself is not meaningful; therefore classification on these signals result in poor performance. For example, classifying individual pixels/dots of an image. The motivation of a convolutional layer is to extract features by applying filters, as known as kernels, to the input signals and then perform classification on the extracted feature maps.
A CNN consists of the following types of layers:

1. Input layer: a layer containing the input signals.
2. Convolutional layer: applies kernels/filters to extract features from the input feature maps and learns the weights to these filters.
3. Pooling layer: down-samples the dimensions of the previous layer and also reduces the chance of over-fitting; an overlap pooling layer may produce an output with the same dimensions.
4. Fully-connected layer: scores the input feature maps by weighted sum, which requires 'all-to-all' connections from the neurons in the previous layer.
5. Output layer: a fully-connected layer with the same number of neurons as the output class.

Computational & Space Complexity

In a CNN, the convolutional layers and the fully-connected layers are the most important layer types, because they take up the majority of the computation and storage. Here we provide an analysis for the computational and space complexity for the convolutional layers and the fully-connected layers.

The convolution layer performs a dot product of the 3D filter matrix of size $f \times f \times d$ to all the elements in the 3D input matrix of size $m \times m \times d$, where f is the dimension of the filter matrix, m is the dimension of the input matrix and d is the depth for the input matrix, with a stride of s. The resulting 2D matrix has a size of $(\frac{m-f}{s}+1) \times (\frac{m-f}{s}+1)$ if no zero-padding is used. A simple example for a 2D convolution (depth is 1) for a 3×3 matrix and a 2×2 kernel with zero padding is illustrated in Figure 2.2. For n selected filters, the output layer has a size of $(\frac{m-f}{s}+1) \times (\frac{m-f}{s}+1) \times n$. Each of the output values is calculated by dot product with a complexity of $f \times f \times d$, the computational complexity is therefore $(\frac{m-f}{s}+1) \times (\frac{m-f}{s}+1) \times n \times f \times f \times d$.

Applying a filter across the input matrix can be inefficient as implemented in modern computers, because the memory is scattered and not contiguous. In practice, the memory is rearranged (flattened) so that the filter matrix and the corresponding patch in the input matrix are arranged as a 1-dimensional vector in a matrix, such that the convolution becomes a matrix multiplication $A \times B = C$, where A is the input matrix with a size of $(\frac{m-f}{s}+1)^2 \times (f \times f \times d)$ and B is the weight matrix containing all the learnable parameters with a size of $(f \times f \times d) \times n$. The method increases computational efficiency, but at the same time, increases the space complexity for the input matrix A from $m \times m \times d$ to $(\frac{m-f}{s}+1)^2 \times (f \times f \times d)$. This is illustrated in Figure 2.3 with the same example used in Figure 2.2.

For a fully-connected layer with an input size of m and an output size of n, which can also be represented by a matrix multiplication $A \times B = C$, where A is the input matrix with a size of $1 \times m$, B is the weight matrix with a size of $m \times n$, and C is the output matrix with a size of $n \times 1$. The computational complexity is therefore $m \times n$.

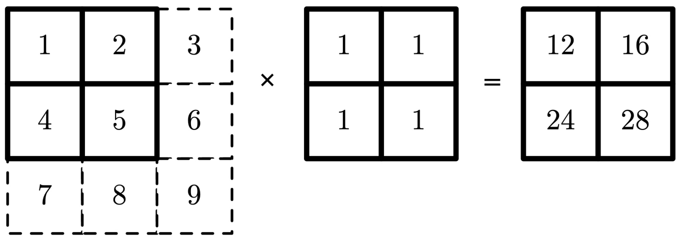


Figure 2.2: An illustration of a simple 2D convolution operation for a 3×3 matrix and a 2×2 kernel with zero padding and a stride of 1.

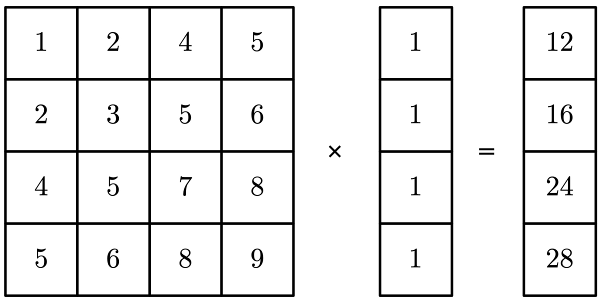
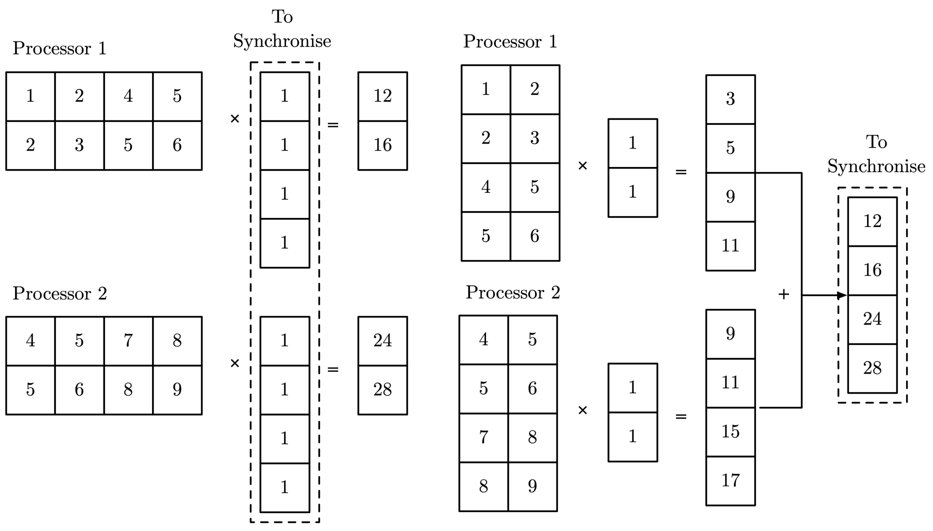


Figure 2.3: An illustration of a flatten convolution operation for a 3×3 matrix and a 2×2 kernel with zero padding and a stride of 1.

Data Parallel vs. Model Parallel

As seen above, the most computational and storage intensive layers are the convolutional and fully-connected layers, which can be represented as matrix multiplications



(a) Data parallelism (b) Model paralellism

Figure 2.4: Data and model parallelism for neural networks.

of $A \times B = C$, where the dimensions of A, B and C are $m \times k$, $k \times n$ and $m \times n$ respectively. To speed-up matrix multiplication through parallelism, there are generally two ways of doing this: splitting matrix A or B, which is the difference between data parallel and model parallel respectively.

For data parallelism, assuming the input matrix A is split by rows into p parts, each worker holds a partition of A and output matrix C of size $\frac{m}{p} \times k$ and $\frac{m}{p} \times n$ respectively. A gradient matrix of size $k \times n$ is produced by the local partition of A and C, which needs to be aggregated, and the communication cost is $k \times n$. This is illustrated in Figure 2.4a with the same example used in Figure 2.3.

For model parallelism, assuming the weight matrix B is split by row into p parts (size is now $\frac{k}{p} \times n$), each worker produces a partition of the output matrix C of size $m \times n$, which needs to be aggregated. In the backward pass, a partition of the error matrix of size $m \times \frac{k}{p}$ is produced and also needs to be assembled. A total of $m \times n + m \times k \times (1 - \frac{1}{p})$ elements needs to be exchanged. This is illustrated in Figure 2.4b with the same example used in Figure 2.3.

For convolutional layers, the first dimension of matrix A is much greater than the second dimension, i.e. $m >> k$, therefore data parallelism is favoured. For a fully-connected layer, matrix A is a vector, i.e. $m = 1$ and $n + k - \frac{k}{p} < k \times n$, therefore model parallelism is favoured.

Mixed data-model parallelism has been suggested in [39], where data parallelism is employed for convolutional layers and model parallelism is employed for fully-connected layers. The switch from data parallelism to model parallelism incurs an extra cost of communication in synchronising the outputs of the last convolutional layer. The mixed parallelism was based on the observation that convolutional layers take up the majority of the computation and fully-connected layers take up the majority of the storage. However, this is not true for most recent models such as GoogLeNet [74], Inception V3 [75] and ResNet50 [69], in which the fully-connected layers only take up 14%, 8% and 4% of the total number of parameters respectively. Therefore, mixed parallelism does not provide much benefit for newer models.

2.1.3 Applications and Datasets

The general application of Convolutional Neural Networks (CNN) is a pattern matching and signal processing, and it has been heavily employed in computer vision, speech processing and natural language processing applications. For example, object recognition, object detection, object tracking, speech recognition, text classification, etc.

This research mainly studies CNNs in the context of object recognition. The task of object recognition is to label images of known objects. There are three datasets widely adopted for benchmarking: MNIST [45], CIFAR [42] and ImageNet [31]. The MNIST dataset consists of handwritten digits from 0 to 9. CIFAR contains 60,000 tiny 32×32 colour images in 10 classes. ImageNet contains 1.2 million colour images in 1,000 classes in various sizes and resolutions. Both MNIST and CIFAR are small and easy tasks, mainly used for sanity checks. ImageNet is the de-facto dataset for evaluating new models.

There are other new additions to the open datasets available for computer vision, speech recognition, as well as other machine learning tasks. For example, Google Open Images, a dataset containing 9 million images in 6,000 classes (with image-level labels and multiple objects in a single image); LibriSpeech, a dataset consisting of clean text and audio from audio books; YouTube-8M, a large dataset of labelled videos for video understanding research. A comprehensive list of these open datasets can be found in [61].

2.2 Background – Model Fitting Algorithms

2.2.1 Parameter Fitting and the Least Squares Problem

The need for parameter fitting and subsequent regression and optimisation arose from parametric models. Parametric models assume a certain form or distribution for the underlying input (X), output (Y) or the unknown transform function ($f : X \to Y$), which are associated with a set of parameters, and the process of fitting the parameters to training data is called 'parameter fitting'. The objective of the algorithm defines the approach to parameter fitting. Common methods include: (i) Least squares, by minimising the sum of squared errors between the predicted value and the observed value; (ii) Maximum Likelihood Estimate (MLE), by maximising value of the likelihood function; (iii) Maximum A-posteriori Probability (MAP) estimate, by maximising the posterior probability.

Given a model function $f(x, \alpha) \to y$, the residual error is defined as $y - f(x, \alpha)$, and the least squares method finds the optimal parameters that minimise the sum of squared residuals:

$$R = \sum_{i=0}^{n} (y_i - f(x_i, \alpha))^2 \tag{2.1}$$

If $y = \alpha x + \beta$ and the residual error $\epsilon = y - \alpha x - \beta$ can be modelled by the normal distribution such that $\epsilon \sim \mathcal{N}(0, \sigma^2)$, then the least squares method is identical to the Maximum Likelihood Estimate (MLE).

As mentioned in Chapter 1, for a linear least squares problem, there is a closed form solution that involves matrix inversion which has a complexity of $O(n^{2.373})$ for a matrix of size $n \times n$ with the optimised Coppersmith—Winograd algorithm [80]. But, the matrix may not be invertible, in which case the analytic solution does not exist. For a non-linear least squares problem, there is no analytic solution in general. As a result, numerical methods are favoured for solving large least squares problems.

Gradient descent and its variants (i.e. batch gradient descent, mini-batch gradient descent and stochastic gradient descent) are the standard methods for solving the least squares problem numerically and are implemented in the common machine learning libraries (e.g., Caffe [35], Pytorch, Tensorflow [1]). The alternatives are Newton's method and its derivatives Iteratively reweighted least squares method and the limited-memory BFGS method. In this research, we study the Stochastic Gradient Descent (SGD) method extensively as it offers a faster computation compared with a second order optimiser such as the L-BFGS method [6]. It is also more suitable for working with large datasets compared with the classical gradient descent method: the difference between the two is that SGD only takes one sample at each iteration instead of the entire dataset, but in practice, a small mini-batch of samples is used

Algorithm 1 Standard Mini-batch SGD

```
1: $w \leftarrow parameters\ of\ the\ objective function$
2: $\eta \leftarrow learning\ rate$
3: procedure SGD($w$, $\eta$)
4:     repeat
5:         Randomly select examples.
6:         Compute gradient $V_{(t)} = \beta V_{(t-1)} + (1 - \beta) \bigtriangledown Q(w)$
7:         Update parameters $w$ with learning rate $\eta$, $w = w - \eta V_{(t)}$ .
8:     until minimum is reached
```

due to computational efficiency limitations. A more extensive comparison between first and second order optimisation methods can be found in Section 2.2.8.

2.2.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimisation method that has been proven effective in many machine learning applications. SGD is in favour for large datasets, over the original Gradient Descent method, because it uses small samples to approximate the gradient of the loss function (i.e. the derivative of the loss function with respect to the parameters), rather than the entire sample population in the original gradient descent method. The SGD optimises objective functions that are in the form of summations (as shown in Equation 2.2, where Q is the objective function with parameter w, and Q_i is the value of the objective function evaluated at the ith observation). Hence, it can be used to solve the least squares problem defined by Equation 2.1. Algorithm 1 shows the standard batch SGD algorithm. In each iteration, the gradient of the objective function is computed based on a batch of randomly selected examples from the dataset, after which the parameter w is updated with the equation shown in Line 7 of Algorithm 1.

$$Q(w) = \frac{1}{n} \sum_{i=1}^{n} Q_i(w) \tag{2.2}$$

There are several variations of the standard SGD algorithm based on two concepts: momentum and adaptive learning rate. The SGD + momentum method incorporates the past gradients into the update equations, as seen in Line 6 of Algorithm 1. The Adaptive Gradient (Adagrad) algorithm [15] is a method that decreases the learning rate monotonically based on the accumulated sum of past gradients to reduce fluctuation. RMSprop [78] is a further variant to the Adagrad method that incorporates a decay on the accumulated sum of past gradients to address issues with the learning rate dropping too low. Adam [38] is the state-of-art variant to the SGD algorithm that combines both the momentum methods and the adaptive learning rate methods.

2.2.3 SGD - Hyper-parameters

The learning rate, momentum and batch size are called hyper-parameters whose values are set before the training process and should be differentiated from 'parameters/weights' of the underlying machine learning model. The *learning rate* determines how far a step is in the direction of the gradient; a small learning rate makes a smooth but slow learning process, while a large learning rate makes the learning process unstable and could potentially lead to a numerical breakdown. The *momentum* decides how much of the previous gradient is used in the calculation to keep the direction of descent steady in the same direction. The *batch size* is the number of samples processed in each iteration ranging from 1 to the size of the training set (it would be identical to the traditional gradient descent when the training batch is the entire training set). A small batch size enables faster learning and less generalisation error but is less stable, since the gradient is evaluated more frequently. A large batch size has the opposite effect of a small batch size (i.e. slower learning and more generalisation), but allows for high degrees of data parallelism to take advantage of computing power (data and model parallelism are introduced in Section 2.1.2 and high-performance computing is introduced in Section 2.3). Optimising the hyper-parameters for fastest descent is difficult as they vary from case to case and the best values are often found experimentally.

The *learning rate* is the most important hyper-parameter as the convergence rate is sensitive to the changes in learning rate. A systematic method for finding the optimum learning rate was proposed in [70]. The idea is to start with a very small value for the learning rate and gradually increase it, recording the training loss at every iteration until it diverges. This way, the optimum value for the learning rate is found where the loss is minimum.

The *momentum* often takes a constant value (e.g. a common value is 0.9). A similar range test for momentum [71] has been further exploited following the success for learning rate in [70]. However, the training loss keeps decreasing as the momentum increases: therefore, an optimum momentum cannot be found using the same method.

The *batch size* often takes a constant small value (e.g. 32, 64, etc.) for fast convergence and regularisation effects (less over-fitting). Increasing the batch size is only useful for improving computational efficiency when data-parallelism is applied. However, the increase in batch size causes a slow-down in convergence. Large batch size training is further discussed in Section 2.2.7.

2.2.4 Parameter Initialisation

Besides the hyper-parameters discussed above, the initial values for the weights/-parameters of a parametric model are crucial to SGD training. Ideally, the values for the parameters should be set as close to the final solution as possible; however, this is hard without prior knowledge of the representation of the underlying data. For a simple linear model such as $y = ax + b$, a and b can be approximated by taking the average of observations. For a multi-stage model with millions of parameters such as the multi-layer perceptron and the convolutional neural network, there is no easy way to approximate the values. In the case of neural networks, inadequate initialisation can easily lead to vanishing or exploding gradients. The problem of vanishing or exploding gradients is associated with the activation function as illustrated in Figure 2.1, which has a non-uniform gradient. For example, the sigmoid function has a maximum gradient at $x = 0$ and a gradient of 0 at positive and negative infinity. Depending on the values of the weights and activation function, the gradients can become progressively bigger or smaller in the backward gradient calculations, resulting in large or small updates to the weights. This, in turn, causes the learning process to be unstable or saturated. Therefore, a good initialisation is more important than the rate of convergence.

A naïve way to initialise the weights in a neural network is by random initialisation with values drawn from a uniform distribution or a Gaussian distribution. As the problem of vanishing or exploding gradients is highly associated with the activation function, heuristics should be taken into account in the initialisation. The Xavier [19] and He et al [25] methods are examples of such a method that incorporates the number of neurons of the layer in the random number generation so that the weights are either too big or too small for the *tanh* and *ReLU* functions respectively. This does not completely solve the vanishing gradient or the exploding gradient problem but improves the convergence rate to a significant extent.

2.2.5 Validation, Regularisation & Generalisation

In learning a machine learning model, the dataset is partitioned into 3 subsets: the training set, the validation set and the testing set. The training set contains a large proportion of the original dataset, which is used as examples to train the model. A validation set is a smaller subset that is used to approximate the performance of the learned model on unobserved data during the training process. The testing set is used on a learned model to evaluate its performance on real-world applications.

There are two metrics for validation: loss and accuracy. Loss is the value computed by the loss function that describes the difference between the predicted value and the expected value and accuracy is only concerned with the percentage of correct labelling. The smaller the loss and the higher the accuracy the better.

Over-fitting is the situation where the model performs well on the training set but not on the validation set. The signs for over-fitting are one of the following:

1. Training loss decreases, validation loss increases, accuracy *decreases.*

2. Training loss decreases, validation loss increases, accuracy *increases.*

The first scenario is commonly encountered, but the second one is counter-intuitive: the accuracy increasing with the validation loss indicates that some of the classes are over-fitted and subsequently the accuracy increases.

The *generalisation error* is a measure of over-fitting that is used to describe how well the learned model can predict previously unseen data. It is defined as the difference between the expected and empirical error. The more over-fitting occurs, the larger the generalisation error. Regularisation is a technique used to reduce over-fitting and improve generalisation of a model, which works by adding a regularisation term to the loss function. The most common regularisers are the L1-norm and L2-norm regulariser, which are the sum of *absolute magnitude* and the square root of the sum of *squared magnitude* of the coefficients respectively. In practice, it has been widely observed that using a small batch size in SGD also reduces generalisation errors and over-fitting, while the use of a barge batch size has the opposite effect.

2.2.6 Asynchronous Stochastic Gradient Descent

Due to the cost of synchronisation at the end of the training step for data-parallel training (a discussion of data-parallelism and model parallelism can be found in Section 2.1.2), the synchronous SGD does not scale with large clusters; therefore research has been attempting to speed up SGD by removing the synchronisation. Hogwild [65] is a lock-free SGD method for shared memory architectures. The idea is that several processes update the parameters asynchronously without locking, so that the processes can be a few steps out-of-sync, but can still converge in spite of losses in accuracy. This is later referred to as the Stale-Synchronous-Parallel (SSP) model. The Downpour SGD described in the Distbelief [12] deep learning library is a similar method to the Hogwild algorithm but implemented on parameter servers, where the parameters are stored in remote servers. Tensorflow [1] is the successor to the Distbelief library and it uses a parameter server like architecture. DeepSpark [37] is a realisation of the parameter server approach on the Apache Spark framework, where the driver process acts as the parameter server, but is inferior to the parameter server implementation since the parameters can only be stored in a single node and this puts more stress on the network bandwidth.

Butterfly mixing [85] is a communication algorithm (based on the *butterfly all-reduce*) used by the BIDdata [62] project, which attempts to accelerate incremental optimisation algorithms (such as stochastic gradient descent) by performing gradient computations at intermediate communication stages. This is an asynchronous optimisation method, in essence, due to the use of unsynchronised weights in the gradient computation, not unlike the Hogwild [65] algorithm and the Stale-Synchronous-Parallel (SSP) model described above. This creates a long lag in synchronisation and leads to greater inaccuracy and this lag must satisfy $k \sim O(n^{1/4})$ according to the Hogwild algorithm [65], where k is the number of steps out-of-sync and n is the number of samples in the training set. For example, $k = \log_2 p$ and $k = 2(p-1)$ for *butterfly all-reduce* and *chunked ring all-reduce* respectively, and k must be less than 32 for 1 million samples. As a consequence, butterfly-mixing forbids bandwidth-optimised algorithms such as the *chunked ring all-reduce* or the use of a large number of workers.

SparkNet [58] proposes a naïve parallelisation scheme for synchronous data-flow frameworks such as Apache Spark, which simply reduces the frequency of synchronisation; it is equivalent to using a larger batch size and by doing so sacrifices the rate of convergence in return.

2.2.7 Large Batch-size SGD Training

As mentioned in Chapter 1, the performance of stochastic gradient descent mainly depends on the convergence rate. For distributed stochastic descent, it may also depend on the batch size and communication efficiency. An analysis of distributed SGD is provided in Chapter 3. The use of a larger batch size effectively increases the total problem size and subsequently the computational efficiency for data-parallel training as explained in Section 2.1.2.

Generally, as the batch size increases, the convergence rate decreases; as such, the convergence rate and the batch size cannot be increased at the same time. This is due to the generalisation effects explained in Section 2.2.5. A typical relation between the convergence rate and the batch size is demonstrated in Figure 2.5; it may differ from problem to problem, but the convergence rate is always smaller for larger batch sizes. In a distributed computing environment, a larger batch size is needed to compensate for the overhead in synchronisation and communication, which in turn slows down the convergence. As a result, the benefits of distributed machine learning diminishes.

A number of adaptive learning rate techniques have been deployed to mitigate

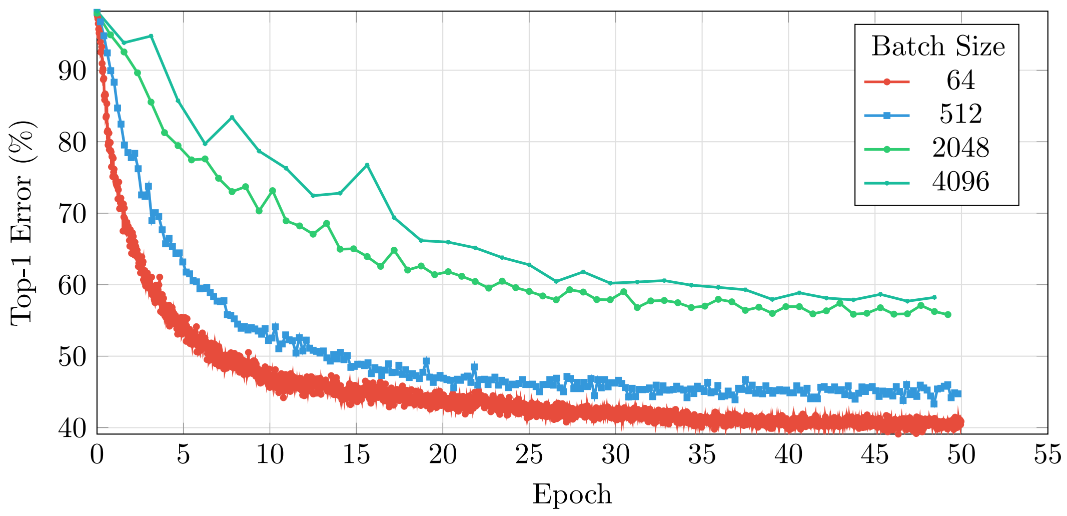


Figure 2.5: Typical validation error vs. epoch: an example using AlexNet on a 5% subset of the ImageNet dataset.

the slow-down of convergence for large batch size training. *Linear Learning Rate Scaling* [39] states that the learning rate should be increased by a factor of k if the batch size is increased by a factor of k. This method works for up to a batch size of 2,048 for AlexNet. It was observed that large learning rates cause the learning to be unstable [27]. Another technique called *Learning Rate Warm-up* is proposed to complement the linear scaling method by gradually increasing the learning rate to the target value from a small initial setting [21]. The *Layer-wise Adaptive Rate Scaling* (LARS) method [82] is an attempt to address this problem for neural network models in particular. It works by adjusting the learning rate based on the ratio between the norm of the weights and the gradients of each layer.

As seen in the techniques above, a large batch size requires a large learning rate, but a large learning rate causes SGD to be unstable. Part of the problem is that a slight change in the parameter values in the deep neural network amplifies as it propagates to the rest of the network, which is referred to as **Internal Covariate Shift**. *Batch normalisation* [32] is a technique that allows for the use of higher learning rates by normalising the mean and variance for each layer inputs using mini-batch statistics, which effectively stops the propagation of covariate shifts at the end of each layer.

Recent research has demonstrated successes in large batch size training with a combination of techniques (e.g. adaptive learning rate, adaptive batch size, adaptive momentum, batch normalisation) [13] [73] [57] [21] [3] [81], which reduces the loss in accuracy while increasing the batch size. The effect of increasing batch size has been further explained by Bayes Theorem [72], and it has been demonstrated that an optimum batch size exists for fixed hyper-parameters, which is proportional to the learning rate and the size of the training data.

2.2.8 Newton's Method In Optimisation

As mentioned earlier in Section 2.2.1, Newton's method is a second order optimisation method as an alternative to the gradient descent method. The difference between first order and second order methods is whether a first order or second order derivative of the objective function (i.e. the function to be optimised) is used. Newton's method uses second order Taylor expansion to find a stationary point of a function f at $x_n+\Delta x$. By setting the values of the derivatives to zero, it is found that $\Delta x = -\frac{f'(x_n)}{f"(x_n)}$ or $x_{n+1} = x_n - \frac{f'(x_n)}{f"(x_n)}$. A generalised form of the equation for higher dimensions is shown in Equation (2.3), where $Hessf(x)$ is the Hessian matrix containing second order derivatives of the function f, and $\bigtriangledown f(x_n)$ is the vector containing first order derivatives of the function.

$$x_{n+1} = x_n - [Hessf(x)]^{-1} \bigtriangledown f(x_n) \tag{2.3}$$

The Hessian matrix provides more accurate information about the curvature of the objective function, which allows for a more direct route to the root of the function, therefore, fewer steps are required compared with gradient descent. However, it is costly to compute (involving matrix inversion) and store the Hessian matrix. Limited-memory BFGS [6] is a quasi-Newton method that approximates the Hessian matrix using limited computer memory. The other disadvantage of L-BFGS is that the entire training set must be enumerated at each iteration (like the gradient descent method), which is time and memory consuming for large datasets.

Due to the limitations mentioned above, second-order methods are not widely adopted in practice, and the stochastic gradient descent method is preferred for simplicity and computational efficiency.

2.3 Background - High-performance Computing

High-performance computing refers to the use of aggregated computing resources to solve large-scale problems, which usually arise from scientific simulations or big data analytics. In the case of deep learning with convolutional neural networks, high-performance computing is a resource for accelerating the training process in a data-parallel or model-parallel fashion as described in Section 2.1.2.

Parallel computing is employed in high-performance computing and is a type of computation that breaks down a computational task into many sub-tasks, where the sub-tasks are executed simultaneously and the results combined upon completion. Modern computers with multiple processors and multiple processor cores can benefit from parallelism by executing sub-tasks on these processing resources in parallel.

Distributed computing is parallel computing on multiple networked computers rather than a single computer. Each has its own private memory space and network communication is required to exchange information and combine results. Distributed computing is used for solving a problem that could use more processing power or memory capacity than a single computer; however, it comes at the expense of slower network communication than shared memory on a single computer.

There are two ways to increase the computing capacity of a parallel and distributed computing system. **Vertical scaling (scaling up)** refers to the addition of more computing resources (e.g. processors, memory and storage) to a single computer. **Horizontal scaling (scaling out)** refers to the addition of more computing nodes to a distributed system.

Speedup is a performance measure widely adopted in parallel computing, which is the relative speed of two systems to solve the same problem. For the same workload, the speedup is defined as the ratio of the serial and parallel execution time of the task as shown in Equation (2.4). The theoretical speedup for parallel computing is formulated by Amdahl's law [4] and Gustafson's law [24].

Amdahl's law predicts the speedup with the number of processors for a fixed total problem size, as shown by Equation (2.5), where t is the original proportion of execution time for the parallel part of the code and s is the speedup for the parallel part of the code; therefore, the maximum speedup is limited by $\frac{1}{1-t}$. Amdahl's law is usually referred to as the **strong scaling** of a parallel and distributed system.

Gustafson's law complements Amdahl's law by predicting the speedup with the number of processors for a fixed execution time of the whole task (or a fixed problem size per processor), as shown by Equation 2.6, s and t have the same meaning as before in Equation (2.5). Gustafson's law does not pose an upper bound on the maximum speedup as it allows programmers to change the total problem size in order to fully exploit the computing power; a larger problem size can be solved in the same amount of time if faster computing resources are available. Gustafson's law is usually referred to as the **weak scaling** of a parallel and distributed system.

$$Speedup, S = \frac{T_{serial}}{T_{parallel}} \tag{2.4}$$

$$Speedup, S = \frac{1}{(1-t)+\frac{t}{s}} \tag{2.5}$$

$$Speedup, S = 1 - t + st \tag{2.6}$$

In addition to speedup, a useful measure in parallel computing is efficiency - E - which measures the ratio of execution time on a uni-processor system and the total execution time of all processors in a parallel system. Efficiency can be expressed as Equation (2.7), where T_{serial} is the serial execution time and $T_{overhead}$ is the total execution time outside of T_{serial} in a parallel system which includes scheduling, communication and synchronisation costs. The isoefficiency function defines the growth rate of problem size with the number of processors to maintain the same efficiency (E) [22]. Isoefficiency analysis provides another measure of scalability of a parallel application: A linear isoefficiency function indicates high scalability; in contrast, a quadratic or an exponential isoefficiency function indicates poor scalability.

$$Efficiency, E = \frac{T_{serial}}{T_{serial} + T_{overhead}} \tag{2.7}$$

The rest of this section compares the existing distributed systems for machine learning which leads to the discussion of the design for an elastic system and the introduction to the Apache Spark framework, and heterogeneous architectures using accelerator technologies.

2.3.1 Characteristics of a Machine Learning Application

Machine learning/deep learning is comprised of training and validation. The purpose of training is to build a machine learning model, which is the most time-consuming step, after which the model can be evaluated in validation or deployed in production. The accuracy of the machine learning model is calculated by the difference between the predicted value and the actual value. Machine learning applications improve their predictions through statistical analysis that fits parameters to the given observations. The input can be in static or streaming form and is usually beyond the memory capacity of a single computer. Since datasets can be too big to reside in main memory, especially for audio, images and videos, a batch of random samples is used in each

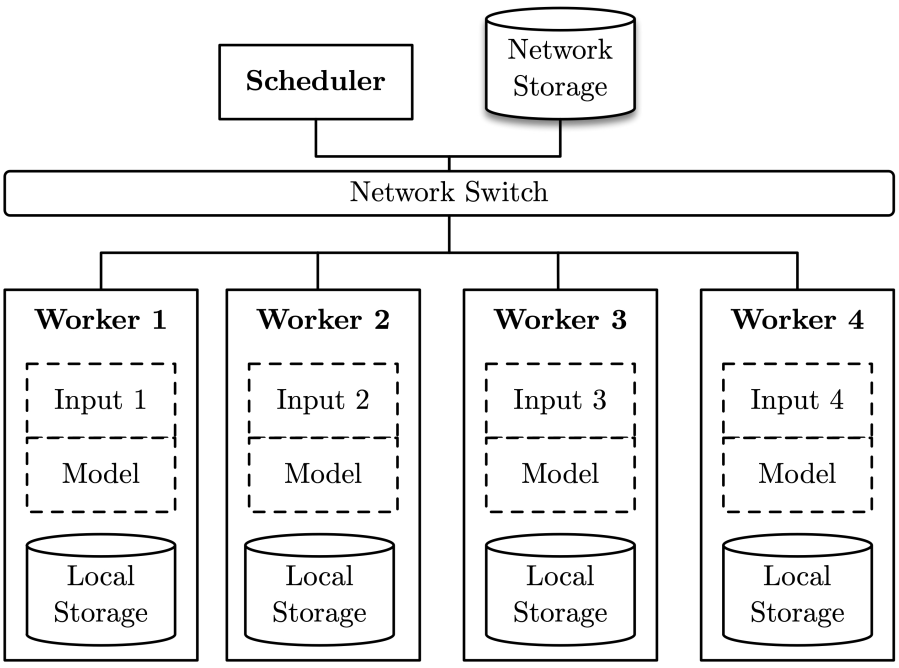


Figure 2.6: A distributed machine learning system.

training iteration. A complete enumeration of the dataset is called an epoch, and usually, more than 1 epoch is needed to train the deep learning model to the desired accuracy. In the context of object recognition with convolution neural networks, the basic operations of a neural network algorithm rely on linear algebra (e.g. dot products and matrix multiplications), which are compute intensive. The machine learning model itself is relatively small compared with the input dataset and can reside in main memory. However, unlike the input data, the machine learning model is volatile and must persist over iterations. This leaves us with a technique that is both compute intensive and data intensive.

2.3.2 System Implementation for Distributed Machine Learning

In a distributed setting, a machine learning system is comprised of a scheduler, a database/file system and a plurality of worker nodes, which are connected through a network. Each worker keeps a partition of the immutable training input and a mutable copy of the machine learning model (Figure 2.6).

There are two main platforms for distributed computing: the message passing interface and data-flow frameworks. In Section 2.3.1, we discussed the characteristics of a deep learning application, and now we can discuss how deep learning applications can fit into existing distributed frameworks.

MPI and Scientific Applications

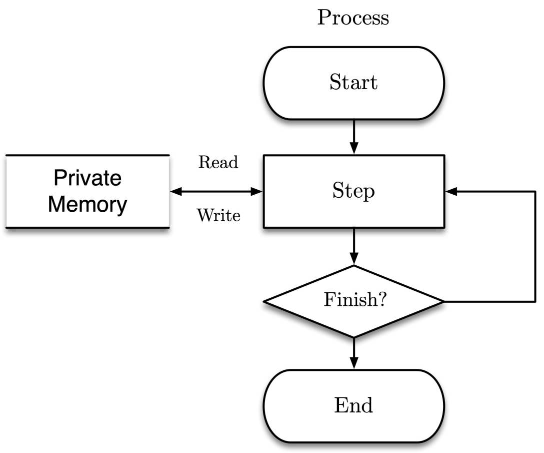
A typical scientific application takes several input parameters and generates a large computer model for physics or similar simulations. The computations often involve linear algebra (e.g., matrix–matrix operations) and require fast mutable memory.

To this end, the MPI framework fits the requirements for scientific applications, since every MPI process has its own local memory (as illustrated in Figure 2.7a) and communication is reduced to a minimum via message-passing. However, programmers are responsible for memory management and there is no native support for data beyond the memory capacity.

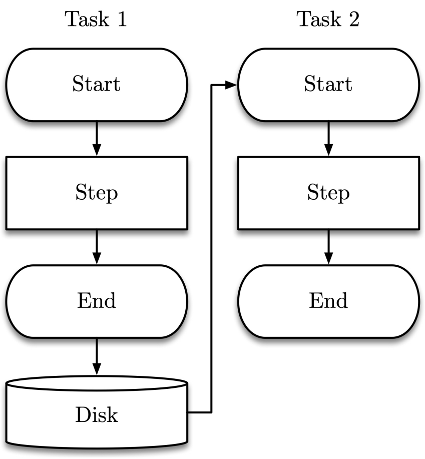
Data-flow and Big Data

A typical big data analytic application takes a large dataset as input and feeds it into a data processing pipeline, after which the data is reduced into a smaller output.

To this end, the data-flow paradigm suits big data analytic applications. Since it is time costly to move large amounts of data across the network, the compute function is instead sent to the worker nodes. The pair of a compute function and a small chunk of the input data forms a task, which transforms the input data into intermediate data and saves it on the disk; this solves the problem of not having enough main memory to hold the data. The intermediate data is passed onto the next stage for further processing until it is reduced into the final output, and each stage of the data processing pipeline is executed in lock-step. The workers do not have persistent memory across different tasks or stages. This is illustrated in Figure 2.7b.



(a) MPI Processes



(b) Analytic Tasks

Figure 2.7: A comparison between MPI processes and analytic tasks.

Parameter Servers

Parameter servers [47] provide a different approach to memory persistence and the fault-tolerance problem, and are employed in the TensorFlow [1] and MXNet [8] frameworks. Parameter servers work by keeping mutable variables in a remote server, away from error-prone workers. It is not an ideal solution since the variables are not local to the workers, and this adds extra latency and traffic to the network. It is therefore not the best approach going forward for larger scale and more advanced high performance compute clusters.

Static vs. Elastic computing

The greatest difference between data analytics and scientific computing comes from the change of workload and therefore resulting in different computing architectures (e.g. MPI, Data-flow, and Parameter servers). Data analytics generally has a data-dependent dynamic workload/problem size, while scientific computing generally has a constant problem size.

A typical case for dynamic workload can be demonstrated by the simple '*Map-Reduce*' data-flow, where data elements pass through a *Map* function and the intermediate results are subsequently combined by a *Reduce* function. The *Map* stage typically requires more processing resources than the *Reduce* stage.

Neural network applications are more akin to scientific computing as it mainly consists of matrix–matrix multiplications as explained in Section 2.1.2. The batch size defines the total problem size for SGD training, which is typically constant throughout the process. For a constant batch size, scientific platforms should be a better candidate for neural network applications. However, with the rise of using dynamic batch-sizing (i.e. incremental or adaptive) for large batch-size training, the total problem size changes dynamically, for which data-flow platforms for elastic computing is a better candidate. The implementation and experimentation of this research are performed on the Apache Spark framework – an elastic data-flow framework, which is further introduced in Section 2.3.4.

The cluster scheduler is the decisive factor for a static or dynamic problem size. Common cluster schedulers include Slurm and Moab for high-performance computing clusters, and YARN and Mesos for data analytic clusters.

Resource allocation in Slurm and Moab is static, and works by submitting a batch script with the job description that describes the amount of resource requested in terms of memory, processor cores and time, and each node executes the same script when the resources are allocated. An example of a job script is shown in Listing 2.1, in which the number of nodes, the number of tasks per node, the memory per processor and the time are defined statically.

Listing 2.1: An example of a batch script for SLURM

```
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=16
#SBATCH --mem-per-cpu=3882
#SBATCH --time=08:00:00
srun ./program
```

Resource allocation in YARN and Mesos is dynamic, the design of YARN and Mesos consists of agents for negotiating resources and they provide a framework for agents, such as the 'ApplicationMaster' in YARN, whose working mechanism is described below. Unlike the batch script shown in Listing 2.1, ApplicationMaster is able to negotiate and interact with the ResourceManager; this is what enables dynamic resource allocation. The Spark framework has built-in support for dynamic resource allocation for YARN and Mesos, as well as in standalone mode. Additional workers can be requested via the *SparkContext* class, and any worker will automatically be freed when it is idle for a certain amount of time.

> For the actual handling of the job, the ApplicationMaster has to request the ResourceManager via AllocateRequest for the required number of containers using ResourceRequest with the necessary resource specifications such as node location, computational (memory/disk/cpu) resource requirements. The ResourceManager responds with an AllocateResponse that informs the ApplicationMaster of the set of newly allocated containers, completed containers as well as current state of available resources. (Apache Hadoop Main 2.6.4 API)

The Message Passing Interface (MPI) and data-flow engines, the two major distributed computing models, are associated with Slurm/Moab and YARN/Mesos respectively. Apache Spark is an example of such a data-flow engine.

For MPI programs, dynamic worker addition and deletion during execution is not possible for the following reasons: (i) MPI programs run in a Single Program Multiple Data (SPMD) fashion and communicators must be created collectively (i.e. each worker must call the same function); therefore communicators cannot be created during computation; (ii) the memory space is private to the process (i.e. each process possess a distinct copy of a variable); (iii) stopping one worker causes the entire job to fail; (iv) technically, it is difficult to connect workers from different job requests. One simple solution to these problems above is to stop-start the application every time the worker pool changes, which is practically impossible if this happens every few iterations.

Data-flow engines usually have a master-slave architecture that allows slaves to be added and removed during the execution of the application. Each worker executes a ‘closure’ function that takes an input and produces an output without side-effects. The ‘closure’ function carries the state of variables which guarantees synchronisation. This model enables the program to be executed anywhere in the cluster, and it was designed for applications that grow or shrink over time.

Overall, the design of elastic schedulers is more scalable and flexible than static batch schedulers and subsequently the associated computing platforms. Elastic computing models have the capability to replace static batch computing models and it is the way forward towards larger scale distributed computing platforms.

2.3.3 Survey of Data Analytics Frameworks

Modern data analytics frameworks arose from the *MapReduce* [11] paradigm for batch processing large amounts of data, which is a simple fixed data processing pipeline consisting of a *Map* and a *Reduce* stage, leading to a different approach of ‘moving compute to data’. After *MapReduce*, research has focused on improving the flexibility of the pipeline, and as a result, Directed-Acyclic-Graph (DAG) and Timely-Graph have been adopted in Dryad [33] and Naiad [59] respectively. Apache Spark is the mainstream data-flow framework, which is considered *MapReduce2.0*. The core of the Spark framework is the DAG engine for scheduling and the Resilient Distributed Dataset (RDD) [84] for in-memory analytics.

Data-flow for batch processing does not satisfy various analytical application use cases, such as stream processing and graph analytics. As a result, there have been new frameworks: (i) Storm [79], Flink [7], IBM Infosphere and Microsoft StreamInsight for stream processing; (ii) Pregel [53], GraphLab [51] and PowerGraph [20] for graph analytics. However, the fundamental mechanisms are similar: An application consists of smaller short-term tasks which carry out a function which may or may not produce side-effects on the state of the data.

2.3.4 Apache Spark & Resilient Distributed Dataset

Apache Spark [84] is a mainstream distributed data-flow framework, which is chosen as the subject of study in this work, as it is capable of in-memory processing, and therefore more suitable for machine learning purposes. It includes two core components: (i) a Directed-Acyclic-Graph (DAG) engine and (ii) a Resilient Distributed Dataset (RDD). The RDD is an in-memory distributed data abstraction that allows fast in-memory computation, which is useful for data to be reused repeatedly. A Spark program is expressed as RDD transformations in a DAG graph; the DAG engine will then generate a physical execution plan, and schedule tasks onto available processors.

The design of RDD and the DAG data-flow engine is inefficient for deep learning and machine learning in general, for a number of reasons in regard to the memory management and data-flow model.

In regard to the Resilient Distributed Dataset (RDD), firstly, it is immutable, meaning changes to the contents are done through transformations that generate yet another RDD and require more memory. Secondly, there is no random access to the records of an RDD, because the state of the dataset is undetermined, so the entire RDD must be computed. As a consequence, RDDs are not suitable to hold machine learning models as they are volatile during training. Access to only a few records in a dataset is inefficient as it causes the entire dataset and the dependent datasets to be computed. We have made substantial improvements with regards to the problems mentioned above in Section 5.2.

In regard to the data-flow model, there is a barrier between stages in a data-flow pipeline, as such overlapping two dependent stages in the same pipeline is not possible (but it is possible to overlap two stages that are not dependent).

Fundamentally, workers in a data-flow framework carry out ‘tasks’, which is a type of short-term process that executes a ‘closure’. As opposed to a normal process that runs throughout the life-span of the program, a data-flow program consists of small tasks that can generate no side-effect outside of the ‘closure’ function. In other words, the workers have no persistent memory. However, it is a common practice to workaround this problem by attaching data to a long-running process, and doing so generates side-effects outside of the closure function, which is not what it was designed to do.

Communication between stages of a data-flow pipeline is through a process called ‘shuffle’, which is not explicitly directed. The sending task saves the shuffle data on the disk and informs the driver process; the receiving task then inquires about the location of the shuffle data from the driver process. Since the task allocation is not pre-determined, it can be anywhere in the cluster, so it is not possible to eagerly transmit data (it is technically possible, but an incorrect prediction of the destination incurs a performance penalty).

Finally, there are limited forms of data-flow communication patterns. For example, an ‘all-reduce’ operation can only be expressed as a ‘reduce-broadcast’ in a data-flow, which can be inefficient for collective communications. We have also improved Spark communication for the ‘reduce-broadcast’ pattern by adapting the MPI ‘all-reduce’ function for elastic task-based systems, which is described in Chapter 4.

2.3.5 Improving Communication for Map-Reduce

From MapReduce to Spark, the shuffling performance has been the main subject of research, which is the data exchange between two stages in a data pipeline. Most research on this subject is concerned with the disk I/O and file system performance. [10] has identified random I/O as a major bottleneck in Spark, and solutions were proposed to reduce the number of open files in order to mitigate the amount of random I/O overhead. Specifically, a set of partition files are created per CPU core instead of per mapper task (there can be more than 1 task assigned to each CPU core). This research also attempted to reduce the amount of shuffle data by columnar compression, but without significant results.

Disk I/O can also be improved through high-performance parallel file systems used on High-Performance Computing (HPC) clusters. One example is the Intel Hadoop Adapter for Lustre [43], which demonstrated that the integrated solution improves Hadoop shuffle with the removal of the extra data copy stage since the Lustre file system appears as a single storage image to all nodes and supports parallel file I/O. Other research experiments with Map-Reduce applications on in-memory file systems like Tachyon and Triple-H. It has been shown that Tachyon is 5$\times$ faster than HDFS for primitive operations [34].

Other studies seek to improve network performance by taking advantage of the latest advances in interconnect technologies (e.g., Remote Direct Memory Access (RDMA) via InfiniBand). [52] is one of the early designs to replace the transport socket layer with RDMA, and demonstrated the possibility for a low latency and high throughput transport layer for Spark. With a similar plug-able RDMA shuffle module, [9] focuses on more efficient pairwise data exchange algorithms for all-to-all communications. The design also features an in-memory buffer for shuffle data instead of compulsory disk persistence in Spark, which is one of the factors that contributes to low latency. However, the implementation was a proof of concept and completely independent from Spark. [46] also presents a similar architecture but with a focus on reusing the off-heap buffer to avoid redundant data copy.

2.3.6 Heterogeneous Architectures

There are two ways to increase the computing capacity of a parallel and distributed computing system as explained at the beginning of the chapter. So far we have discussed horizontal scaling (adding more computing nodes), the other is vertical scaling (adding more processors), and co-processors/accelerators are one such resource. There are three major types of co-processors: manycore CPU, Graphical Processing Unit (GPU) and Field Programmable Gate Array (FPGA) cards. These co-processors are architecturally different, but there is one thing in common: they are pluggable cards currently connected to the main CPU through the Peripheral Component Interconnect (PCI).

To understand the differences between these architectures and their impacts on the computational performance of machine learning, we briefly introduce how processors work. A generic processor pipeline consists of 4 stages: fetch, decode, execute and write-back; the pipeline runs based on a clock, the faster the clock rate, the faster the pipeline is executed. Floating-point arithmetic is performed by the Floating-Point Unit (FPU) of the processor, a CPU may have multiple FPUs integrated or added, which enables multiple floating-point operations per cycle. Vector processing is a way to increase floating-point operations per cycle, which operates in a Single Instruction Multiple Data (SIMD) fashion; MMX, SSE and AVX are such SIMD extensions to the x86-64 architecture. Subsequently, the floating-point arithmetic performance can be measured by FLoating-point Operations Per Second (FLOPS), which is defined by Equation (2.8).

$$FLOPS = \#cores \frac{cycles}{second} \frac{FLOP}{cycle} \tag{2.8}$$

Manycore CPU co-processors, such as the Intel Xeon Phi, are a composition of 60-72 CPU cores, each equipped with vector processors which run at a lower clock rate (megahertz instead of gigahertz). Graphical processing units (GPUs) are a type of massively-parallel computing resource, which differ from traditional CPU architecture by having many more smaller but specialised computing cores (NVIDIA GPUs typically have 1000+ computing cores). Unlike the general-purpose CPU cores, GPU cores have a much smaller instruction set, and are therefore more energy and computationally efficient. Both CPU and GPU co-processor cores are not able to execute general-purpose software such as the operating system, therefore, the operating system must run on the main CPU, resulting in a heterogeneous architecture – the computational tasks are offloaded from the CPU to the co-processor via PCI as illustrated in Figure 2.8.

The difference between manycore CPU and GPU architectures are the changes in the number of cores and clock rate in Equation 2.8. Both architectures are successful in terms of the FLOPS performance: the Intel Xeon Phi, NVIDIA Tesla K80 and P100 produce tera-FLOPS (10^{12}) double-precision performance.

FPGAs are a completely different architecture that allow for programmable circuits, unlike fixed circuits in the CPU and GPU. This allows for a specialised circuitry that performs more efficient or fused operations (such as fused multiple-add) without going through multiple processor cycles. However, programming FPGAs requires knowledge in electronic circuit design, unlike general-purpose programming languages. FPGAs are also limited by the clock rate, which is often lower than CPUs.

Machine learning and scientific computing are comprised of linear algebraic operations as explained in Section 2.1.2, where the elementary multiplications and additions are independent of each other, resulting in many-way parallelism. This could largely benefit from the manycore CPU and GPU architectures. With the rising use of co-processors in machine learning, this research is conducted in the context of heterogeneous architectures using co-processor cards.

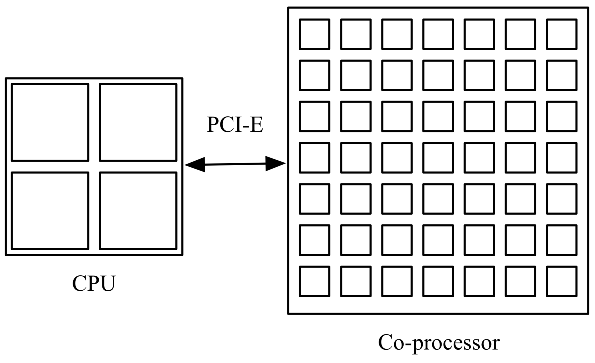


Figure 2.8: A heterogeneous computer system comprised of a CPU and a co-processor connected by the PCI.

Chapter 3

Performance Modelling

Analytic modelling for running time and cost is crucial to the understanding of the scalability and cost-effectiveness of Distributed Stochastic Gradient Descent (SGD). Existing performance evaluations on distributed SGD are experimental, which are not transferable from one setup to another. Precise modelling at a microscopic-scale is difficult, depending on numerous variables (e.g. processor speed, hierarchical data transfer, network speed, machine learning model complexity, machine learning model parameter size, etc.). Such work had been performed on a multi-GPU architecture on a single node [68]. Our objective is to model the running time and running cost in terms of 4 macro variables: number of epochs, batch size, processing speed (samples per second) and network speed, which provides a high-level overview for all different setups.

3.1 Modelling Synchronous SGD

Assuming a Bulk Synchronous Parallel (BSP) computing model, the computation is broken down into super–steps, where a barrier–style synchronisation is applied at the end of each super–step. The wall–clock time of each super–step for every process is identical due to synchronisation. The BSP model is a simplistic parallel computing model, which is the closest fit to the execution model for Apache Spark where the jobs are also executed in super–steps, except that the jobs are broken down into smaller tasks on Spark.

Let $T_{compute,total}$ be the total processing time and $T_{comm,total}$ be total communication time across all nodes including the idling time. The wall–clock time on a cluster machine with N nodes can be expressed as Equation 3.1, and the running cost can be expressed as Equation 3.2, assuming the input/output and synchronisation overheads are included in $T_{compute,total}$ and $T_{comm,total}$.

$$Time = (T_{compute,total} + T_{comm,total}) \times \frac{1}{N} \tag{3.1}$$

$$Cost = (T_{compute,total} + T_{comm,total}) \times \frac{cost}{time} \tag{3.2}$$

The computation cost can be modelled by Equation 3.3, where $\#samples = \#epochs \times dataset\_size$ and γ is the processing speed (samples per second), assuming γ is constant. The communication cost with respect to the number of nodes N can be approximated with a linear function (i.e. $T_{comm,single} = \frac{\#samples}{B}(\alpha + \beta N)$, where B is the global batch size, $\frac{\#samples}{B}$ is the total number of iterations, N is the number of workers and α and β are constant coefficients), because the number of steps of weight aggregation is bounded by the number of workers (i.e. $\#steps \propto N$ for *ring all-reduce* and $\#steps \propto \log_2 N$ for *butterfly all-reduce*, see Section 4.1.1 for details). Subsequently, the total communication cost of N nodes is $N \times T_{comm,single}$ as shown in Equation 3.4.

By substituting $T_{compute,total}$ and $T_{comm,total}$ in Equations 3.1 and 3.2, and assuming the cost over time is proportional to the processing speed (i.e. $\frac{cost}{time} \propto \gamma$), expressions are derived for the running time and cost in Equations 3.5 and 3.6 respectively. Subsequently, the expression for computational speed-up ($\frac{T(N=1,\alpha=0,\beta=0)}{T(N)}$) can be derived in Equation 3.7.

By neglecting the scheduling and synchronisation costs (i.e. $T_{overhead} = T_{comm,total}$ in Equation (2.7)), the efficiency can be expressed as $E = \frac{T_{compute,total}}{T_{compute,total}+T_{comm,total}}$. By substituting Equations (3.3) and (3.4), efficiency can be derived as Equation (3.8).

$$T_{compute,total} = \frac{\#samples}{\gamma} \tag{3.3}$$

$$T_{comm,total} = \frac{\#samples}{B} N(\alpha + \beta N) \tag{3.4}$$

$$Time \propto \#epochs \times \frac{B + \alpha\gamma N + \beta\gamma N^2}{B\gamma N} \tag{3.5}$$

$$Cost \propto \#epochs \times \frac{B + \alpha\gamma N + \beta\gamma N^2}{B} \tag{3.6}$$

$$Speedup = \frac{BN}{B + \alpha\gamma N + \beta\gamma N^2} \tag{3.7}$$

$$E = \frac{B}{\gamma N(\alpha + \beta N) + B} \tag{3.8}$$

3.2 Modelling Asynchronous SGD

For asynchronous stochastic gradient descent where the computation is overlapped with communication, the computation cost is taken away from the communication (i.e. $T_{comm,blocked} = max(T_{comm,total} - T_{compute,total}, 0)$), Equations (3.1) and (3.2) become Equations (3.9) and (3.10) and subsequently Equations (3.11) - (3.13) (all symbols have the same meanings as before).

By neglecting the scheduling and synchronisation costs, $T_{overhead}$ becomes $T_{comm,blocked}$ in Equation (2.7). Therefore, $E = 1$ for $T_{comm,total} \leq T_{compute,total}$; and $E = \frac{T_{compute,total}}{T_{compute,total}+(T_{comm,total}-T_{compute,total})} = \frac{T_{compute,total}}{T_{comm,total}}$ for $T_{comm,total} > T_{compute,total}$. By substituting Equations (3.3) and (3.4) into $E = \frac{T_{compute,total}}{T_{comm,total}}$, efficiency can be derived as Equation (3.14).

$$Time = max(T_{compute,total}, T_{comm,total}) \times \frac{1}{N} \tag{3.9}$$

$$Cost = max(T_{compute,total}, T_{comm,total}) \times \frac{cost}{time} \tag{3.10}$$

$$Time \propto \#epochs \times max(\frac{1}{\gamma N}, \frac{\alpha + \beta N}{B}) \tag{3.11}$$

$$Cost \propto \#epochs \times max(1, \frac{\alpha\gamma N + \beta\gamma N^2}{B}) \tag{3.12}$$

$$Speedup = min(N, \frac{B}{\alpha\gamma + \beta\gamma N}) \tag{3.13}$$

$$E = \frac{B}{\gamma N(\alpha + \beta N)} \tag{3.14}$$

3.3 Assumption Validation

The performance model has been developed based on the assumption that the processing speed per node (γ, samples per second) is constant regardless of the batch size, and the communication cost is linear with respect to the number of workers for a fixed problem size, i.e. $T_{comm,total} \propto N$.

To validate the assumption regarding the processing speed, we train AlexNet on the ImageNet dataset for 20 iterations for each batch size setting, on an Intel Xeon E5-2660 (2.6 GHz) CPU and an NVIDIA K80 GPU. Figure 3.1 shows the computation time for increasing batch size for CPU and GPU. The processing speed (γ) is represented by the gradient of the curve, which verifies the assumption that the processing speed per node (γ) is constant regardless of the batch size, for up to a batch size of 400 samples in our experiment. The processing speed γ is found to be 10 and 260 samples per second for the CPU (Intel Xeon E5-2660 2.6 GHz) and GPU (NVIDIA Tesla K80) respectively.

To validate the assumption regarding the communication cost, we test the 'all-reduce' implementation described in Chapter 4 for both the *butterfly* and *chunked ring* algorithms, over TCP and RDMA communication protocols, different number of workers (N) and different vector sizes (i.e. $10^6, 10^7, 10^8$), for 20 iterations. Figures 3.2a - 3.3b show the average all-reduce time for these experiments. The curve can be affected by a lot of factors, such as the all-reduce algorithm, the system load, the system implementation, the network topology, the underlying hardware architecture, etc. However, a linear function is still the best approximation: we have fitted linear, logarithmic, power and polynomial equations to the data, and we found a linear equation results in the least residual squared errors, which supports that a linear approximation is the best fit for the communication cost with respect to the number of workers (N), even though the curve is not perfectly linear. For AlexNet with 65 million parameters, we can extrapolate the values for α and β from Equation 3.5 as follows: $\alpha = 2.0$ and $\beta = 0.024$ for *chunked ring all-reduce* over RDMA, $\alpha = 0.8$ and $\beta = 0.03$ for *butterfly all-reduce* over RDMA.

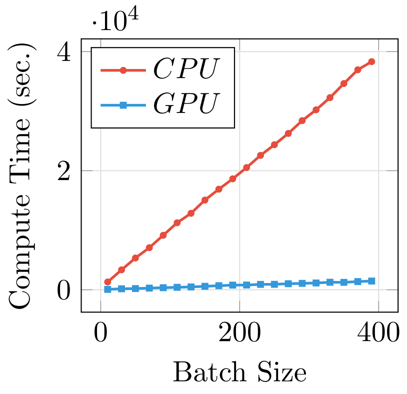
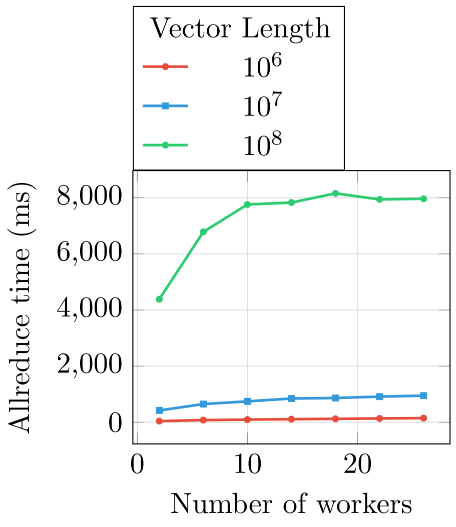
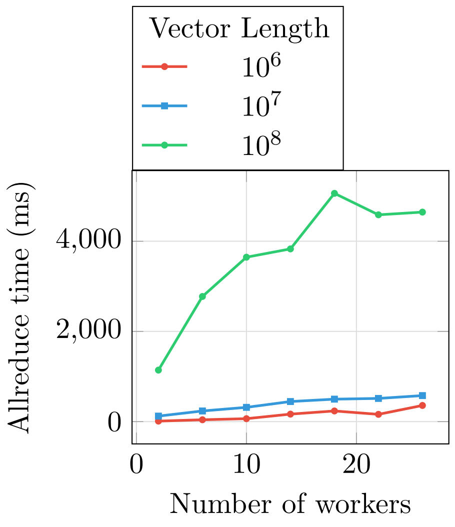


Figure 3.1: Computation time for AlexNet-ImageNet over different batch size on a Intel Xeon E5-2660 2.6 GHz CPU and a NVIDIA K80 GPU.

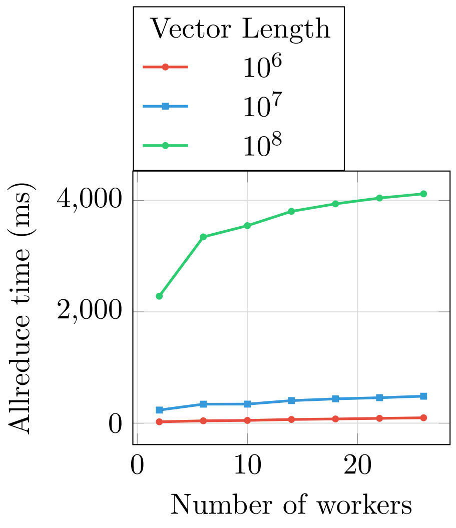


(a) Ring-TCP

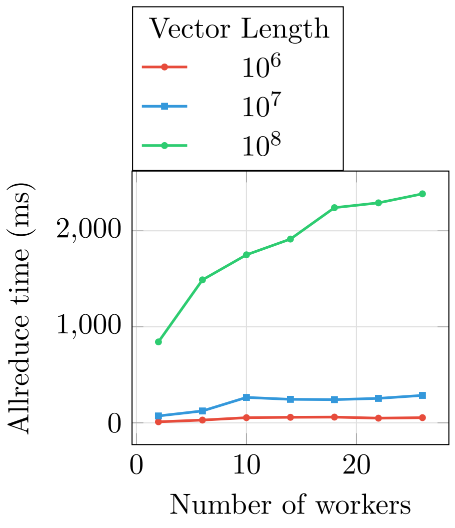


(b) Butterfly-TCP

Figure 3.2: Allreduce time against number of workers for a vector length of 10^6, 10^7 and 10^8 over TCP.



(a) Ring-RDMA



(b) Butterfly-RDMA

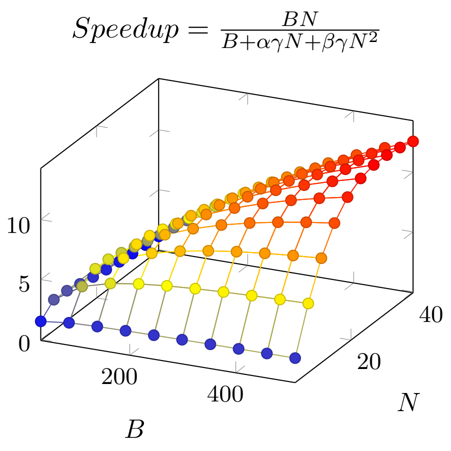
Figure 3.3: Allreduce time against number of workers for a vector length of 10^6, 10^7 and 10^8 over RDMA.

3.4 Strong and Weak Scaling for SGD

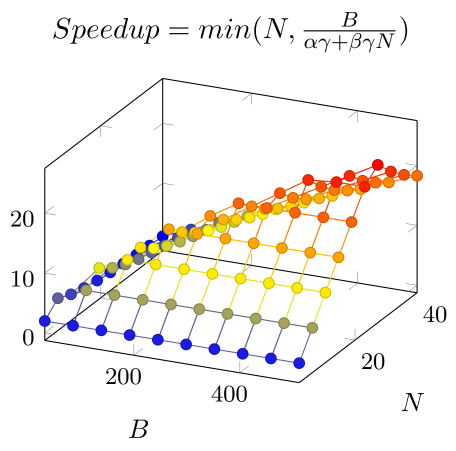
In Section 2.3, the concepts of strong and weak scaling were introduced for High-Performance Computing (HPC), which are measures of scalability for HPC that describe the change in solution time with the number of processors for a fixed total problem size and for a fixed problem per processor. The same concepts can be applied to synchronous and asynchronous SGD training using Equations (3.7) and (3.13), and the values for computational speed-up are plotted against the number of workers (N) and global batch size (B) in Figures (3.4a) and (3.4b) for synchronous and asynchronous respectively.

The axis along variable N represents **strong scaling** for SGD – the change of speed-up with the number of workers for a fixed global batch size (i.e. total problem size). It is shown that the speed-up is a concave function for both synchronous and asynchronous SGD, which has a peak value and after which is penalised as the number of workers keeps increasing.

A vertical cut through the axes B and N represents **weak scaling** for SGD - the change of speed-up with the number of workers for a constant batch size per worker (i.e. constant $\frac{B}{N}$ or a fixed problem size per processor). For synchronous SGD, it is shown that the speedup slows down and converges to a limit. For asynchronous SGD, the speed-up increases linearly as it reaches the peak value and starts to decline. For both synchronous and asynchronous SGD, a maximum speed-up exists for a constant problem size per processor.



(a) synchronous



(b) asynchronous

Figure 3.4: Speedup as a function of number of workers (N) and global batch size (B). Plotted with coefficients $\alpha = 0.2$, $\beta = 0.03$ and $\gamma = 10$

3.5 Maximum Theoretical Speed-up

As shown above, a maximum speed-up exists for a fixed problem size or a fixed problem size per processor (i.e. strong and weak scaling) and is yet to be determined. The minimum runtime with respect to number of workers (N) can be found by finding $\frac{dT}{dN} = 0$ for Equations (3.5) and (3.11). Once the minimum runtime is found, the maximum speed-up can be found by $\frac{T_{serial}}{T_{parallel,minimum}}$.

For synchronous training, by differentiating Equation 3.5 with respect to N (i.e. $\frac{dT}{dN} = 0$), we found the number of workers for maximum speed-up as shown in Equation (3.15). For asynchronous training, the maximum number of workers can be found by equating the computation time and the communication time (i.e. $\frac{1}{\gamma N} = \frac{\alpha+\beta N}{B}$), which is shown in Equation (3.16). Subsequently, the maximum speed-up can be found by substituting Equations (3.15) and (3.16) into Equations (3.5) and (3.11) respectively, which is given by Equations (3.17) and (3.18). The problem size per processor ($\frac{B}{N}$) can also be found by dividing B by Equation (3.15) or (3.16), which states that $\frac{B}{N} \sim \Omega(\sqrt{B})$, this implies the problem size per processor has to increase with the batch size to achieve maximum speed-up.

Figure 3.5 is a plot of theoretical maximum speed-up as a function of the batch size (i.e. problem size) for synchronous and asynchronous SGD. It shows that the maximum speed-up is approximately proportional to the square root of the batch size (i.e. $S_{max} \propto \sqrt{B}$), which implies that the benefits from increasing the batch size recedes as the batch size gets larger, as $\frac{dS}{dB} \propto \frac{1}{2\sqrt{B}}$. It also shows that the asynchronous SGD is nearly twice as fast as the synchronous SGD for the same problem size.

$$N_{max,sync} = \sqrt{\frac{B}{\beta\gamma}} \tag{3.15}$$

$$N_{max,async} = -\frac{\alpha}{2\beta} + \sqrt{((\frac{\alpha}{2\beta})^2 + \frac{B}{\beta\gamma})} \tag{3.16}$$

$$S_{max,sync} = \frac{B}{\alpha\gamma + 2\sqrt{B\beta\gamma}} \tag{3.17}$$

$$S_{max,async} = min(N_{max,async}, \frac{B}{\alpha\gamma + \beta\gamma N_{max,async}}) \tag{3.18}$$

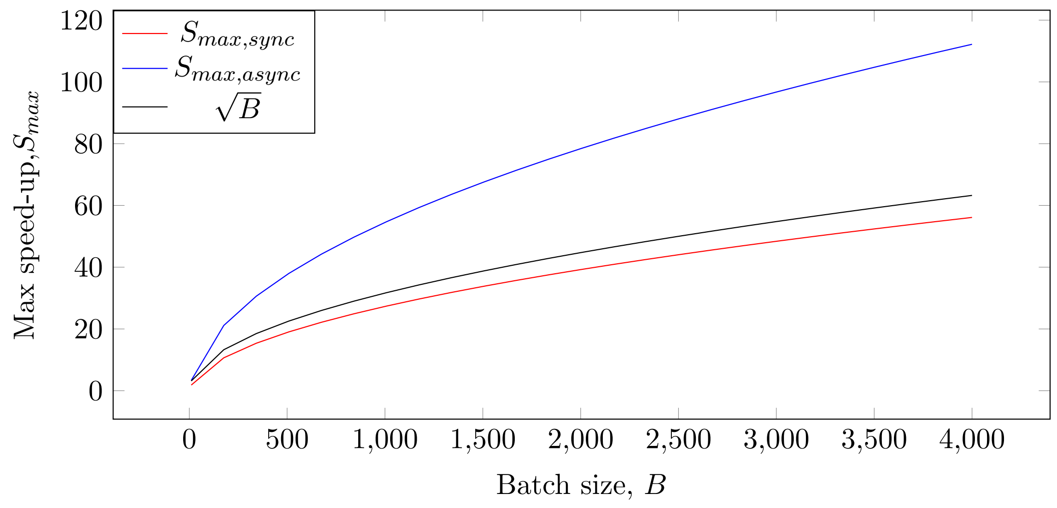


Figure 3.5: Theoretical maximum speedup as a function of the global batch size (B). Plotted with batch size $\gamma = 10$, and coefficients $\alpha = 0.2$ and $\beta = 0.03$

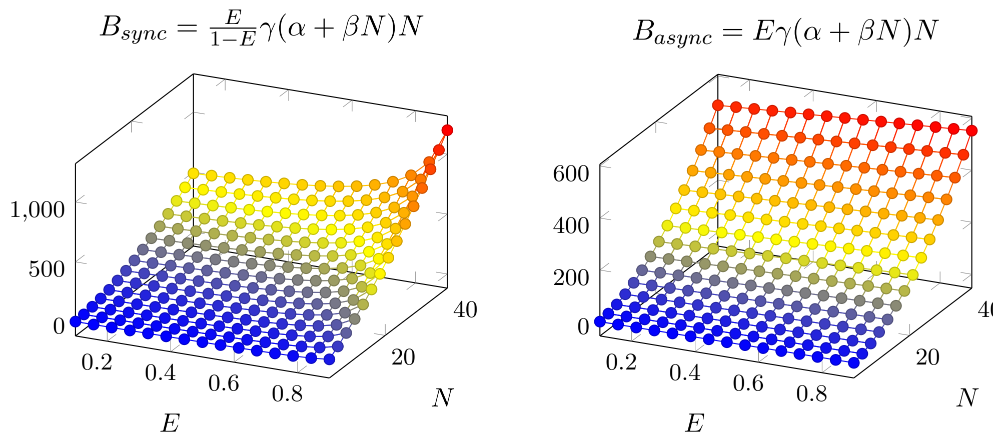
3.6 Isoefficiency Analysis

As introduced in Section 2.3, isoefficiency is another measure of scalability for HPC, which measures the rate of change of problem size with respect to the number of processors to maintain the same efficiency. The efficiency as function of batch size (B) and number of nodes (N) has been found in Equations (3.8) and (3.14) for synchronous and asynchronous SGD respectively. By arranging Equations (3.8) and (3.14), the batch size/problem size can be expressed as Equations (3.19) and (3.20).

As shown by Figures 3.6a and 3.6b, the batch size/problem size (B) is a quadratic function with respect to the number of nodes (N), and $B_{sync} = \frac{1}{1-E} B_{async}$. This indicates poorer scalability for the SGD algorithm compared with the linear case - $B \propto N$. As the efficiency analysis neglected the scheduling and synchronisation overheads, the actual efficiency is even lower. However, for asynchronous SGD, it is possible to reach an efficiency of 100%, for a batch size up to $\gamma(\alpha + \beta N)N$ when $T_{comm,total} \leq T_{compute,total}$.

$$B_{sync} = \frac{E}{1-E}\gamma(\alpha + \beta N)N \tag{3.19}$$

$$B_{async} = E\gamma(\alpha + \beta N)N \tag{3.20}$$



(a) synchronous (b) asynchronous

Figure 3.6: Global batch size (B) as a function of efficiency (E) and number of nodes (N). Plotted coefficients $\alpha = 0.2$, $\beta = 0.03$ and $\gamma = 10$.

3.7 Summary

This section explores analytical performance for distributed data-parallel SGD in terms of the number of epochs, batch size and the computational and communication speed, where equations for running time and computational speed-up are derived.

The scalability of distributed SGD is analysed using the concepts of strong and weak scaling from High-Performance Computing (HPC). It is shown that a maximum speed-up exists for both synchronous and asynchronous SGD, for a fixed total problem size (i.e. strong scaling) and a fixed problem size per processor (i.e. weak scaling).

The maximum theoretical speed-up is found to be a function of the batch size, by differentiating the equations for the running time with respect to the number of workers. We show that the maximum speed-up is approximately proportional to the square root of the batch size (i.e. $S_{max} \propto \sqrt{B}$), which implies that the benefits from increasing the batch size recedes as the batch size gets larger, as $\frac{dS}{dB} \propto \frac{1}{2\sqrt{B}}$. We also show that the problem size per processor ($\frac{B}{N}$) is found to be $\frac{B}{N} \sim \Omega(\sqrt{B})$, this implies that the problem size per processor has to increase with the batch size to achieve maximum speed-up.

The scalability of SGD is also analysed through isoefficiency - the rate of change of problem size with respect to the number of nodes to achieve the same efficiency. It has been demonstrated that isoefficiency is a quadratic function for SGD (i.e. $B \propto N^2$), which indicates poorer scalability for the SGD algorithm compared with the linear case - $B \propto N$. However, for asynchronous SGD, it is possible to reach an efficiency of 100%, for a batch size up to $\gamma(\alpha + \beta N)N$ when $T_{comm,total} \leq T_{compute,total}$.

Lastly, the running cost for distributed data-parallel SGD is shown to be constant for the same number of epochs and compute-to-communication ratio, since Equation 3.2 can be written as $Cost \propto T_{compute,total}(1+\frac{T_{comm,total}}{T_{compute,total}})$ and $T_{compute,total}$ is constant. Therefore, training on a single computer will have the lowest running cost, and the running cost is constant as long as the computation–communication ratio is constant.

Chapter 4

Communication

High-Performance Computing introduced in Section 2.3 is a sub-category of distributed and parallel computing. The main difference between distributed and non-distributed computing is the need for between-node communication between workers, which is the bottleneck for large-scale machine learning training. Communication efficiency is key to high-performance computing and machine learning applications.

Neural networks suffer the most from the communication overhead as explained in Section 2.1.2: (i) The weight matrix or the output matrix must be synchronised for data–parallel and model–parallel training respectively; (ii) Often contain millions of parameters. For convolutional neural networks, data-parallelism is favoured for convolutional layers and model-parallelism is favoured for fully-connected layers. As the computation and parameter distribution shift to convolutional layers, data-parallelism has become the main parallelism for convolutional neural networks. To synchronise the weights of a machine learning model, an operation called 'all-reduce' is introduced, which collects and combines a vector from all processes and delivers the results back to the processes.

SparkNet [58] is an implementation of convolutional neural networks on the Apache Spark platform, which has demonstrated that 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. The overall runtime in distributed neural network training is dominated by communication, highlighting the need for a more efficient all-reduce implementation.

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 [77], 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 optimised 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. 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 assumes all processes run in parallel and synchronisation may cause deadlocks in a task-based framework. Since MPI cannot interact with the task scheduler, and tasks run asynchronously, synchronisation operations can cause the application to hang.

- 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.

As a consequence, the introduction and implementation of blocking operations such as all-reduce cannot simply be translated from one high-performance computing framework to another, and a more efficient all-reduce is needed for the Apache Spark framework.

4.1 Background

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.

4.1.1 All-Reduce Algorithms

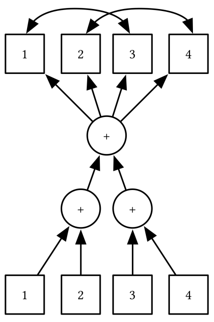
The performance of an all-reduce algorithm depend on many factors, including: (i) the size of the vector; (ii) the size of the cluster; (iii) the nearest number of nodes in a power-of-two; (iv) the network latency and bandwidth and, (v) the network topology (e.g., ring, mesh, torus, hypercube, dragonfly, etc.). Classical implementations include the *butterfly* and the *binary-tree* algorithms. *Ring all-reduce* has been popular in neural network and machine learning applications due to the large size of the parameters and limited network bandwidth on cloud computers. This research focuses on the *butterfly all-reduce* and *ring all-reduce* algorithms, as algorithmically, one takes the least number of steps (*butterfly*) and one takes the least bandwidth (*ring*) so that they represent algorithms with optimal latency and bandwidth respectively. A number of previous studies have already made extensive comparisons of all-reduce algorithms for the Message-Passing-Interface (MPI) [63] [64] [77].

Apache Spark implements a simple variant of the *reduce-broadcast* algorithm for all-reduce, which is illustrated in Figure 4.1a. The *reduction* phase (i.e., bottom half) is a binary-tree reduction process that takes $\log_2 p$ 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 an all-to-all shuffle for the rest of the data blocks.

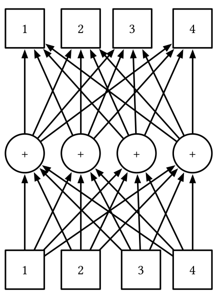
Parameter servers [47] also implement a variant of *reduce-broadcast* by 'pushing' and 'pulling' data to and from the servers. Instead of reducing the traffic volume through tree reduction in Spark, it employs more servers that store a portion of the memory, resulting in a *scatter-gather* pattern as shown in Figure 4.1b, where the 'push' and 'pull' operations are equivalent to the 'scatter' and 'gather' operations. This effectively increases the bandwidth at the server side, but the total amount of traffic across the network is unchanged which can cause congestion. The ratio between the number of servers and workers must accommodate to the size of the data for better performance, which is difficult to adjust from case to case.

The *butterfly* algorithm is illustrated in the middle of Figure 4.1c. 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 $\log_2 p$ steps to complete, where p is the number of processes.

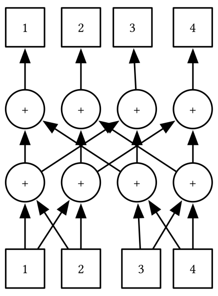
The *ring all-reduce* algorithm can be understood simply as forwarding the vector in a circle as illustrated in Figure 4.1d, and every worker will have the reduced vector after $p-1$ steps. A bandwidth optimised version of the algorithm is similar, but instead of forwarding the entire vector, only $\frac{1}{p}$ of the vector is forwarded at each step. Each worker will possess the reduced partition of the original vector after $p-1$ steps, and this process is called 'the scatter' phase. The reduced partitions are forwarded to the next worker as before, and after another $p-1$ steps, each worker will have the full reduced vector, and this processed is known as the 'gather' phase. The bandwidth optimised *ring* algorithm is also known as the *chunked ring* algorithm, which reduces the total data transmission by a factor of $\frac{2}{p}$, but doubles the number of steps (i.e. $2(p-1)$).



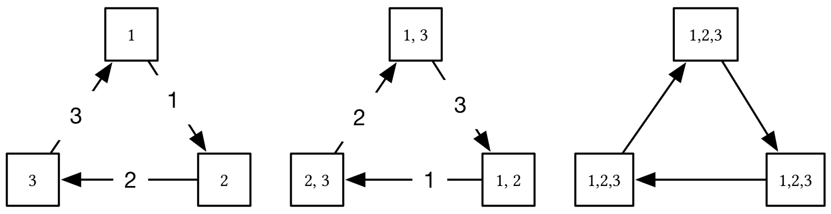
(a) Reduce-Broadcast



(b) Scatter-Gather



(c) Butterfly



(d) Ring

Figure 4.1: All-Reduce Algorithms

4.1.2 Theoretical Performance

We compare the *reduce-broadcast*, *butterfly all-reduce* and *ring all-reduce* algorithms through theoretical cost estimations using the work of Thakur [77]. Let there be p processes and a single process per node (i.e. $p = N$), with each producing a vector of b bytes after an initial local reduction. g is the computational cost per byte of locally executing one operation with two operands, and z is the serialisation or de-serialisation cost per byte through a serialisation algorithm. Network communication is modelled as linear time by $u + bv$, where u is the latency/start-up time per message and v is the transfer time per byte.

Binary-tree reduction takes $\log_2 p$ steps and, in each step, vectors are fetched and combined by the reduction task [23, Chapter 4.1]; the cost is therefore:

$$T_{tree\_reduce} = \log_2 p(u + bv + 2bz + bg) \tag{4.1}$$

The communication cost for broadcasting b bytes in *block_size* blocks is:

$$T_{broadcast} = \frac{b}{block\_size}(u + v \cdot block\_size + 2 \cdot z \cdot block\_size) \tag{4.2}$$

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 ($\log_2 p$), but all processes fetch and combine in parallel [23, Chapter 4.3]. The cost of *butterfly* all-reduce, assuming a process count of a power-of-two, is therefore:

$$T_{butterfly} = \log_2 p(u + bv + 2bz + bg) \tag{4.3}$$

For the original *ring* all-reduce, it takes $p - 1$ steps and b bytes are being exchanged and combined at each step, the cost of which is shown in Equation 4.4. For the *chunked ring* algorithm, the number of steps is doubled and the amount of data at each step is factored by $\frac{1}{p}$, the cost of which is shown in Equation 4.5.

$$T_{ring} = (p - 1)(u + bv + 2bz + bg) \tag{4.4}$$

$$T_{ring,chunked} = (p - 1)(2u + \frac{2bv}{p} + \frac{4bz}{p} + \frac{bg}{p}) \tag{4.5}$$

In comparison, the *butterfly all-reduce* is more efficient than the *reduce-broadcast* and *ring all-reduce*; however, it is comparable to the *chunked ring all-reduce.* By equating Equations (4.3) and (4.5), $T_{ring,chunked} > T_{butterfly}$ when Equation (4.6) is satisfied. Equation (4.6) is true when all coefficients are non-zero and it leads to predicates as shown in Table 4.1, which suggests that *chunked ring all-reduce* is more efficient if the cost is bounded by bandwidth (v), computation (g) and serialisation (z), while *butterfly all-reduce* is more efficient if it is bounded by latency (u). Since v, z and g are multiplied by the size of the vector b, the *butterfly all-reduce* will have more advantages over small vectors, and vice versa. For multi–threaded processes with c processor cores, the cost for computation (g) and serialisation (z) will be further reduced by a factor of $\frac{1}{c}$, which reduces the advantages of the *chunked ring all-reduce* for large vectors.

$$u(2p - 2 - \log_2 p) + b(v + 2z)(2 - \frac{2}{p} - \log_2 p) + bg(1 - \frac{1}{p} - \log_2 p) > 0 \qquad (4.6)$$

Table 4.1: Conditions for which the coefficients in Equations (4.6) are positive

Term	Positive Condition
u	$p > 2$
$b(v + 2z)$	$1 < p < 2$
bg	$0.5 < p < 1$

4.1.3 Butterfly All-Reduce in Apache Spark

In the early stages of development, it has been 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’ [18], and as a result the *reduce-broadcast* approach was adopted as an alternative.

Consequently, users employ the less efficient *reduce-broadcast* method provided by Spark, or more efficient custom self-contained Java implementations if available. However, these are bespoke solutions that assume parallel tasks as MPI processes, which can potentially hang as previously described.

As seen in Section 4.1.2, *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.

4.1.4 All-Reduce in Machine Learning

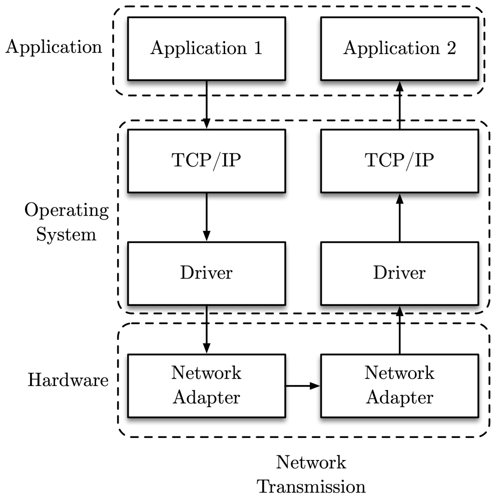
Many machine learning algorithms can be formulated as an optimisation problem to search for the best model, and Stochastic Gradient Descent (SGD) is a popular algorithm for solving the optimisation problem over a large dataset. A distributed implementation of SGD averages the model weights across the 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 50× reduction in communication volume, and a 1.3× 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.

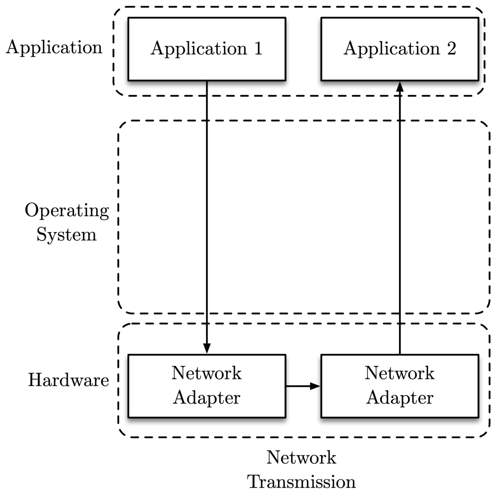
Kylix [86] is another self-contained Java implementation of all-reduce that attempts to optimise all-reduce for power-law graph data that commonly presents itself in web graphs and social networks, for example. The idea of Kylix is to use heterogeneous-degrees at different layers of a butterfly network, and it is shown that the communication volume in the lower-layer is typically much less than the top layer. Experimental results show a 5× speed-up of Kylix with respect to the binary *butterfly* algorithm in a selection of different test scenarios.

4.1.5 Remote Direct Memory Access

Direct Memory Access (DMA) allows for peripheral devices such as the graphical processors and network adaptors to gain access to the main memory directly, which is enabled by the sharing of the electrical bus between the CPU and the DMA-enabled device. Remote Direct Memory Access (RDMA) is similar to DMA but involves data transmission through the network. Traditionally, data-flow between the main memory and device memory goes through the CPU and the operating system, generating redundant memory copies in the process. With DMA or RDMA, the main memory can be accessed directly by the device, bypassing the CPU and the operating system. In the case of network communication, this can be demonstrated by comparing the traditional TCP/IP and RDMA. Figure 4.2a depicts the normal data flow from the application space of a local computer to a remote computer, in which the memory is copied from the application space to the operating system (i.e. TCP/IP and driver), and from the operating system to the device memory; the reverse process is carried out on the remote side. With RDMA, the operating system is bypassed, memory is copied directly between the application and the device as illustrated by Figure 4.2b. In short, the RDMA memory transfer is totally silent generating zero memory copy and no CPU and operating system interrupts and enabling high data transfer throughputs.



(a) TCP/IP



(b) Remote Direct Memory Access

Figure 4.2: Comparison between usual network transmission and remote direct memory access.

To allow for (R)DMA, a special computer resource is required – pinned memory. Pinned memory differs from normal application memory by the non-pageable and non-swappable characteristics. To understand this, we must understand how virtual memory in modern computer architectures works. Virtual memory is a technology that allows for multiple processes to occupy the same linearly contiguous memory space, creating an image of a uniform memory layout of physically scattered memory pieces. To access the data pointed by a virtual memory address, the virtual memory address must be translated to a physical address that points to the correct location in the main memory as illustrated in Figure 4.3. Data pointed by virtual memory can also be in the secondary storage, by *swapping* memory between the main memory and the secondary storage as required, expanding the memory space beyond the physical memory capacity.

For memories to be directly accessible by a peripheral device, they must be physically addressable and present in the main memory, since the DMA controller bypasses the memory translation in the CPU; therefore pinned memory must be used for DMA. However, allocating too much pinned memory degrades system performance as physical memory will run out for non-pinned memory and causing them to be swapped to secondary storage, subsequently increasing I/O operations. This makes pinned memory a scarce resource on a computer. For further details about DMA, please refer to [36].

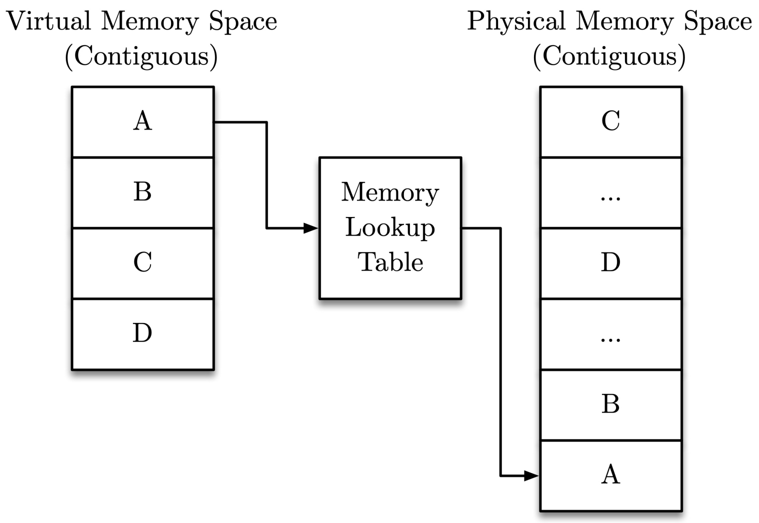

Virtual Memory Space (Contiguous)
A
B
C
D
Memory Lookup Table
Physical Memory Space (Contiguous)
C
...
D
...
B
A

Figure 4.3: Memory translation for virtual memory system.

4.2 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. Sections 4.2.1 and 4.2.2 introduce the 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 optimisation are identified and are detailed further in Sections 4.2.3, 4.2.4 and 4.2.5.

4.2.1 All-Reduce Architecture

In contrast to the static parallel processes of an MPI application, 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 serially or in 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 4.4 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 2. 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*).

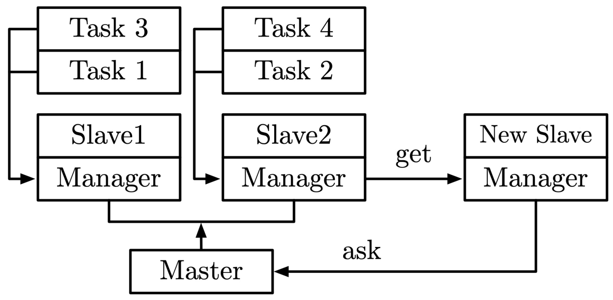


Figure 4.4: Architecture of task-based all-reduce

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

4.2.2 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 Section 4.1.3), 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. `Submit`(vector): Submits a vector for reduction. The function does not block;
3. `Get`: Gets the globally reduced vector. Blocks 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

Algorithm 2 Multi-threaded implementation of the all-reduce manager

```
1: $reduced\_vector \leftarrow empty\ vector$
2: $local\_submissions \leftarrow new\ queue$
3: $reduced\_vector\_lock \leftarrow new\ lock$
4: $new\_vector\_signal \leftarrow new\ semaphore$
5: $global\_reduction\_start\_signal \leftarrow new\ semaphore$
6: $global\_reduction\_finish\_signal \leftarrow new\ semaphore$
7: procedure BackgroundThread
8:     while wait for signal from master process do
9:         $global\_reduction\_start\_signal.signal()$
10: function Init(key, num_tasks, func)
11:     Register(key, num_tasks, func)
12:     Start new thread(LocalReduction)
13:     Start new thread(GlobalReduction)
14:     return
15: function Submit(new_vector)
16:     $local\_submissions.add(new\_vector)$
17:     $new\_vector\_signal.signal()$
18:     return
19: function Get( )
20:     $global\_reduction\_finish\_signal.wait()$
21:     return $reduced\_vector$
22: function LocalReduction
23:     while $new\_vector\_signal.wait()$ do
24:         $reduced\_vector\_lock.lock()$
25:         $reduced\_vector \leftarrow Reduce(reduced\_vector, new\_vector)$
26:         $reduced\_vector\_lock.unlock()$
27:         Remove($local\_submissions$, $new\_vector$)
28:         Signal master process
29:     return
30: function GlobalReduction
31:     $global\_reduction\_start\_signal.wait()$
32:     $reduced\_vector\_lock.lock()$
33:     Apply all-reduce algorithm (e.g., butterfly)
34:     $reduced\_vector\_lock.unlock()$
35:     $global\_reduction\_finish\_signal.signal()$
36:     return
```

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 serialisation and computation.

4.2.3 Parallel Processing

Figure 4.5 depicts the scheme by which the data is processed in a parallelised 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., serialisation–upload–get–deserialisation–reduction) simultaneously and asynchronously.

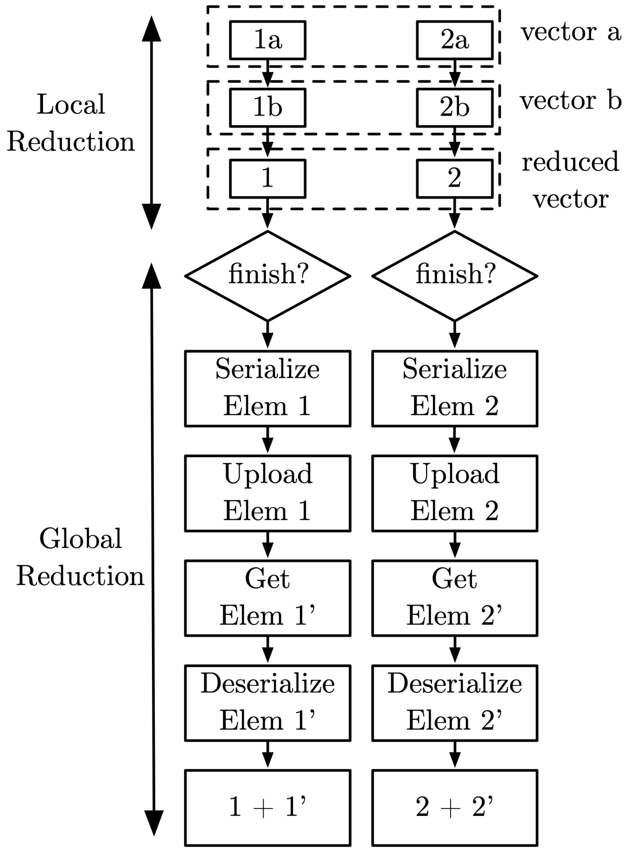


Figure 4.5: Internal Mechanism of the all-reduce process. Elem 1: first element/-partition in the local vector. Elem 1': first element/partition in the exchanged vector.

Applying the cost analysis described in Section 4.1.2, the cost of parallel *butterfly all-reduce* and *chunked ring all-reduce* become Equations 4.7 and 4.8 respectively,

$$T_{butterfly,par} = \log_2 p(u + bv + \frac{2bz}{c} + \frac{bg}{c}) \tag{4.7}$$

$$T_{ring,chunked} = (p-1)(2u + \frac{2bv}{p} + \frac{4bz}{cp} + \frac{bg}{cp}) \tag{4.8}$$

where c is the number of available processor cores on each node, and other symbols have the same meaning as in Section 4.1.2. In comparison, object serialisation 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 threefold:

- *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 utilise 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 Section 4.3.2, object serialisation is the dominant cost factor, but there is no parallel implementation of the generic serialiser. To speed up object serialisations of arbitrary type, users must implement a custom parallel serialisation method, which involves low-level byte manipulations that are 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 parallelisation 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.

4.2.4 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 twofold: (i) all-reduce vectors are relatively small in size compared to the input dataset, and

(ii) submitted/exchanged vectors are combined into a single vector, resulting in a memory usage that does not grow as the number of tasks increases.

4.2.5 RDMA Optimisation

Remote Direct Memory Access (RMDA) is a hardware feature in high-performance computer networks such as Infiniband, which is previously introduced in Section 4.1.5. RDMA has the advantage of reducing the number of memory copies by interfacing directly between the network card and the main memory, which in turn reduces latency and frees up the processor. We compare the experimental results for TCP-IPoIB/RDMA and Ring/Butterfly all-reduce in Section 4.3.4.

There are several issues to enabling RDMA in Spark. Spark uses the Netty TCP library for network communications; replacing Netty requires a significant change of code. Verbs is the low-level interface designed for RDMA communications. To enable RDMA, one solution is to use Java Sockets Direct Protocol (deprecated) or its successor the rsocket/librspreload [67] that automatically translates socket to rsocket/verbs calls; the other solution is to use the DiSNI/jVerbs library [30] developed by IBM. Both options provide a socket-like interface for the verbs interface. However, rsocket is not yet a standard protocol and it is only available on Linux systems; as such it is not a universal solution. For cloud services (e.g. AWS, Google, Azure), only Azure provides RDMA on the Windows system and this has been the decisive factor for the choice of computer system and software library.

Automatic translation from the socket to rsocket interface seems to be the default option, but socket and rsocket cannot co-exist in the same application in this way. For analytic applications, data may be stored in network-attached or off-site storage, which will require a normal socket connection. This means automatic interface translation is not an option. On the other hand, RDMA and sockets/messages are two different communication methods. Mapping the socket interface to RDMA cannot fully utilise the advanced features of RDMA. For example, in RDMA, the receiver can lazily fetch the remote memory and any sub-portion of the memory whenever it wants, whereas the receiver can only fetch the entire message once in sockets.

For the reasons above, the DiSNI is favoured for Apache Spark. The design of RDMA networking is similar to a non-blocking socket application; the main difference is that the sender only provides the information of the local memory, and subsequently, the receiver can read the remote memory into its own memory space.

4.3 Results

4.3.1 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 4.2. Notable features of this hardware include Intel Xeon CPUs and an Infiniband interconnect.

Table 4.2: Hardware and 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 (IB QDR)
Software	Centos/Linux-2.6, Hadoop 2.7, Spark-2.1.1

We evaluate the performance of all-reduce by comparing 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.

4.3.2 Empirical Performance

Figure 4.6 reports the average all-reduce time against the vector size on 32 executors, and Figure 4.7 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.

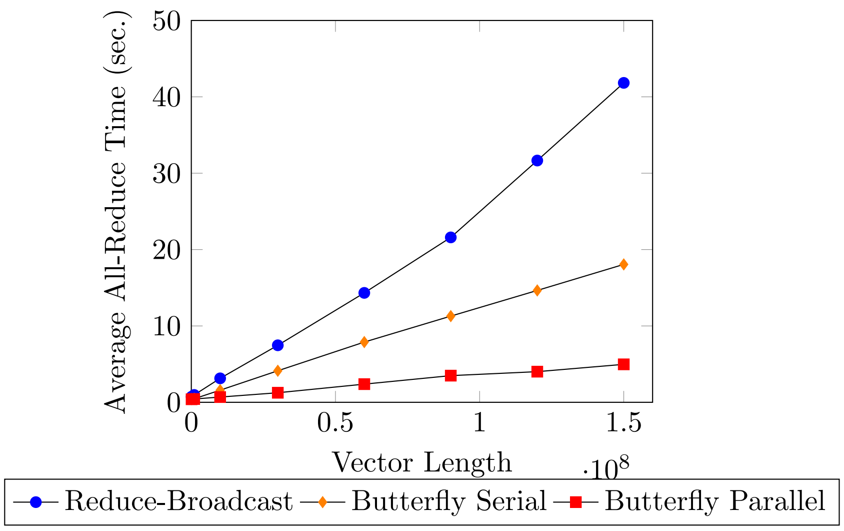


Figure 4.6: Average All-Reduce Performance on 32 Executors for a Single Iteration

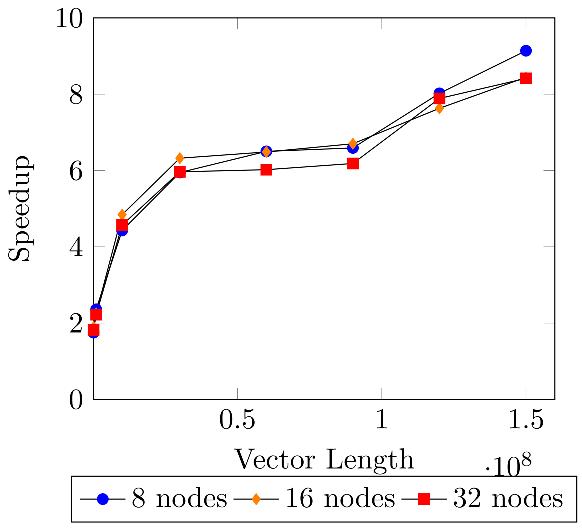


Figure 4.7: Speed-up of Parallel Butterfly w.r.t Tree-Reduce+Broadcast on 8, 16, 32 nodes

Reduce-Broadcast and Vector Length

It is observed that the gradient of *reduce-broadcast* starts to grow as the vector length reaches 10^8. The same is reflected in Figure 4.7, where the speed-up should have saturated at $7\times$ 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.

Butterfly All-Reduce and Cluster Size

Even though the *butterfly* algorithm minimises the number of steps in the all-reduce, it is still susceptible to network bandwidth limits and contention. In contrast to the *reduce-broadcast* method, we have not seen an increase in steepness in overall all-reduce time in Figure 4.6 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 4.3. 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.

Table 4.3: Per Stage Time for Vector Length of 1.5×10^8 for Parallel Butterfly All-Reduce

Nodes	8	16	32
Time (seconds)	0.95	0.93	0.99

Breakdown Analysis

Figure 4.8 reports the breakdown of costs in all-reduce, which is summed over 10 runs and averaged across 32 slave nodes. The overheads are split into 5 metrics:

1. Start-Up: Starting up of tasks, including task delivery, serialisation/deserialisation, etc.;
2. Compute: Compute cost of the reduction function;
3. Send Overhead: Object serialisation (for all), and disk I/O for Spark Shuffle (for reduce-broadcast only);
4. Receive Overhead: Object deserialisation;

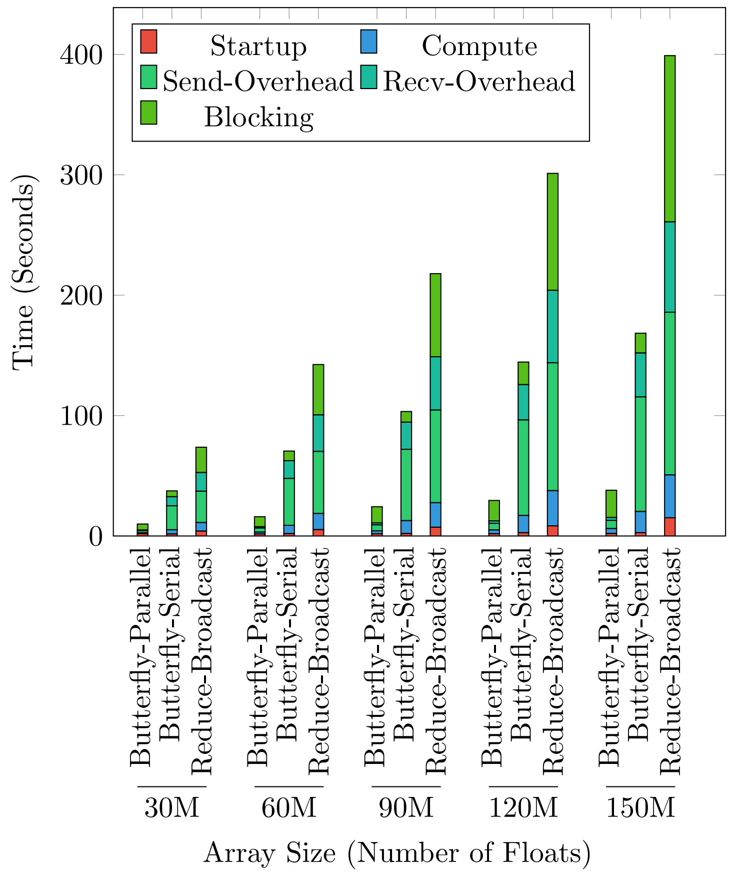


Figure 4.8: Breakdown of overheads in all-reduce of a large array size for 10 iterations on a 32-node cluster

5. Blocking: Block time during network transmission of data (for all), and final stage object deserialisation at the driver process (for reduce-broadcast only);

By comparing the breakdown components of *serial-butterfly all-reduce* and *reduce-broadcast*, the network block time in *serial-butterfly all-reduce* is reduced by 84%, whilst the cost of computation and object serialisation are almost identical. The *parallel-butterfly all-reduce* further optimises the compute and object serialisation by making use of all available CPU cores. Compute time is reduced by 80-90%, and object serialisation (i.e., send overhead + receive overhead) is also reduced by 80-90%, with respect to the serial version. Overall, algorithmic changes (i.e., *butterfly all-reduce* against *reduce-broadcast*) and parallel-processing contributes to 65% and 35% of the overall speed-up.

4.3.3 Applications - Neural Network

As described at the beginning of the chapter, Neural networks are a typical example of where the overall performance suffers due to the network exchange of parameters at each iterative step. CIFAR, MNIST and ImageNet are three popular datasets in neural network research for object recognition, which are also used as examples in SparkNet [58]. 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 4.4.

CIFAR and MNIST are relatively small datasets compared with ImageNet, and so the neural-net models are therefore simpler. The model weights for CIFAR and MNIST are only 0.2% and 0.6% of the size of ImageNet. Nevertheless, a 2.3× speed-up is observed for CIFAR and MNIST, and a more notable 7.4× speed-up is observed for ImageNet. The all-reduce times and speed-ups match the projections seen in Figures 4.6 and 4.7.

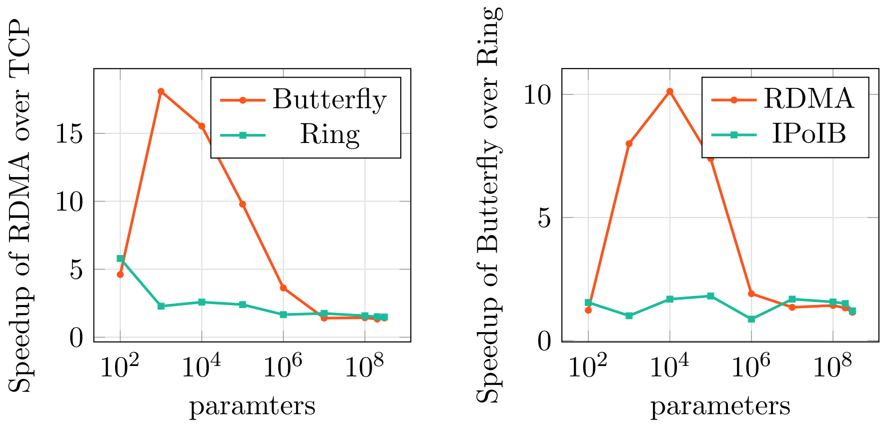
Table 4.4: All-reduce time in real-world neural network applications across 32 nodes. Original: Reduce-broadcast. New: Butterfly all-reduce.

Dataset	Neural Net	Weight size – log length	Original (sec.)	New (sec.)
CIFAR [42]	cuda-convnet [40]	5.2	0.356	0.154
MNIST [45]	LeNet [44]	5.6	0.447	0.184
ImageNet [31]	AlexNet [41]	7.8	17.9	2.4

4.3.4 Ring vs. Butterfly All-reduce

We have seen how *butterfly all-reduce* performs in comparison with *reduce-broadcast* over TCP, on artificial data and on real-world applications. Here we further explore the *chunked ring all-reduce* and *butterfly all-reduce* over TCP-IPoIB and RDMA communication protocols. The objective is to establish whether bandwidth or latency should be the priority when selecting an algorithm for aggregating a large number of parameters in neural network training. We run *chunked ring all-reduce* and *butterfly all-reduce* 20 times for the length of parameters (single precision floating point numbers) ranging from $10^2 - 10^8$.

Figure 4.9a reports the speedup of RDMA over TCP-IPoIB. It shows that RDMA achieves a speedup as large as 18× on the *butterfly all-reduce*, and as the number of parameter increases, the advantage of RDMA decreases. On the other hand, RDMA has negligible effect for *chunked ring all-reduce* as the speedup remains close to unity. This implies that zero memory copy has a greater impact on bandwidth-



(a) Speedup of RDMA over TCP (b) Speedup of Butterfly over Ring

Figure 4.9: Comparison of all-reduce runtime on 32 workers for the *butterfly* and *chunked ring* (abbr. *ring*) algorithm over RDMA and IPoIB.

bounded algorithms such as the *butterfly all-reduce* which is more copy-heavy. The total amount of memory copy is $\log_2(p) \times b$ for *butterfly all-reduce* and $\frac{2b(p-1)}{p}$ for *chunked ring all-reduce*, where p is the number of processes and b is the size of the vector. RDMA is also proven more effective on moderately-sized vectors: if the vector size is too small, the advantage of zero memory copy is less significant; if the vector size is too large, resource for *pinned memory* is quickly consumed and degrades system performance. *Pinned memory* is a scarce memory resource on a computer, which is non-pageable and non-swappable unlike the *non-pinned memory* and is necessary for RDMA to work as explained in Section 4.1.5.

Figure 4.9b reports the speedup of *butterfly all-reduce* over *chunked ring all-reduce*. It is shown that *butterfly all-reduce* is favoured over *chunked ring all-reduce* for a parameter size between 10^3 and 10^5 over RDMA. The performance is comparable for these two algorithms over TCP-IPoIB, but *chunked ring all-reduce* is favoured due to a lower computational cost as analysed in Section 4.1.2.

For real–world applications, we see a clear advantage of the *butterfly all-reduce* algorithm over RDMA for models with a small number of parameters. For example, the *butterfly-RDMA* would be 7-8 times faster than the *chunked ring-RDMA* by extrapolation from Figure 4.9b, for the Cuda-convnet model [40] and LeNet model [44] for CIFAR [42] and MNIST [45] datasets, with $10^5 - 10^6$ parameters. For models with large number of parameters, such as the AlexNet [41] for the ImageNet [31] dataset, the *butterfly all-reduce* does not have a clear advantage over the *chunked ring all-reduce*, and the *butterfly all-reduce* poses higher computational stress on the processor. Therefore, the *chunked ring all-reduce* algorithm is expected to perform better in practice under system load and for large vectors over TCP. Everything considered, there is no single winner for every application and the choice of the algorithm has to be considered case to case.

4.4 Summary

In this chapter, we explore novel, efficient all-reduce algorithms and their implementation in elastic task-based frameworks. We present an architecture and interface for all-reduce in task-based frameworks and a parallelisation scheme for object serialisation and computation. Testing of the new *butterfly all-reduce* and *chunked ring all-reduce* is conducted using the Apache Spark framework.

The effectiveness of the *butterfly all-reduce* is demonstrated by a logarithmic growth in speed-up with respect to the vector length compared with an existing *reduce-broadcast* method. A $9\times$ 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* is also tested with respect to the naïve *reduce-broadcast* method on model-updates of neural network applications. A $2\times$ and a $7\times$ speed-up are observed for the CIFAR and MNIST datasets, and the ImageNet dataset, respectively. We predict a stable performance of the *butterfly* algorithm for larger cluster sizes.

We also compare the *butterfly all-reduce* and *chunked ring all-reduce* over TCP/IPoIB and RDMA communication protocols. We show a clear advantage of the *butterfly all-reduce* algorithm over RDMA for moderately-sized vectors, but RDMA is only effective on copy-heavy algorithms and vectors with moderate size. We conclude that there is no single winner for every application and the choice of the algorithm has to be considered from case to case.

To further reduce the cost of communication, one possible direction is data compression methods to reduce the storage volume of the parameters. For example, by dropping near-zero values in [2], or by using mixed/lower precision floating point numbers in [56]. One other direction is by overlapping computation and communication, which is further explored in Section 6.1.

Chapter 5

Memory Management

Besides inter-process communication as discussed in Chapter 4, computational efficiency is another key element to the performance of machine learning. The computational pipeline in machine learning consists of two parts: data pre-processing and data consumption, as shown in Figure 5.1. Data pre-processing is comprised of the data loading and transformation of the input data, which can later be consumed by the machine learning algorithm. In this process, static (i.e. input data) and volatile (i.e. model parameters) data are generated. Efficient handling of the processing pipeline is closely associated with the overall computational efficiency, which is a key factor in the performance model.

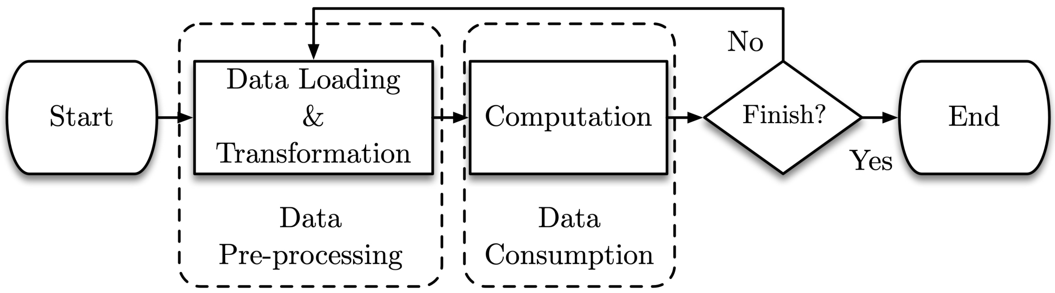


Figure 5.1: Processing pipeline for machine learning

The data consumption is dominated by linear algebraic calculations for neural networks, as explained in Section 2.1.2. Linear algebra is the core to high-performance scientific computing which has been under constant improvement; as such, the data consumption throughput is relatively efficient.

The data pre-processing, on the other hand, is often underestimated but contributes to computational inefficiency. This is especially prominent for heterogeneous architectures, where the data consumption is performed by faster accelerator cards and the data pre-processing is performed by the slower host processor, resulting in an increased proportion of processing time in the data pre-processing. The consumption of physical memory also increases the pressure on the input/output throughput, which indirectly impacts the overall performance of the system.

The Resilient Distributed Dataset (RDD) is the core distributed memory concept in Apache Spark that is responsible for the representation and computation of datasets. Section 5.1 covers a detailed introduction to the Resilient Distributed Dataset for Apache Spark and why it is inefficient for both immutable and mutable datasets. The remainder of the chapter introduces a new *MapRDD* (see Section 5.2) and a new distributed key-value store (see Section 5.3) for better handling of immutable and mutable data respectively.

5.1 Resilient Distributed Dataset

The *Resilient-Distributed-Dataset (RDD)* memory abstraction is the key concept underpinning the Spark framework. In this section, we explore in detail the underlying structures and working mechanisms of RDD, and its application in machine learning and heterogeneous environments.

5.1.1 Parallelism

The fundamental unit of an RDD is a partition that describes a subset of the dataset, rather than the elements in the partition. When a *map* function is applied to the dataset, tasks are created for each partition. Therefore, the number of tasks is the number of partitions, and so is also the level of parallelism. Memory management is also organized in terms of partitions; as such, a data-block unit belongs to a single partition.

This detail turns out to be crucial to understanding the performance difference between *map()* and *mapPartitions()* transformations, as it had been recognized that there are discrepancies between the two [66] [54]. As shown in lines 3 and 11 of Listing 5.1, the user-function is applied to the entire partition as a single task, where *iter* is an iterator for the elements in the partition, and the difference is whether the user-function takes an element or an iterator as input, but they create the same number of tasks/threads. As demonstrated by Lester Martin [54], *map()* transformations can lead to slower performance than *mapPartitions()* transformations, if some helper objects are created for every element, but parallelism does not contribute to the performance difference.

Listing 5.1: Code snippet from RDD.scala: map() and mapPartitions() Syntax

```
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[U, T](this, (context, pid, iter) =>
        iter.map(cleanF))
}
def mapPartitions[U: ClassTag](
    f: Iterator[T] => Iterator[U],
    preservesPartitioning: Boolean = false): RDD[U] =
       withScope {
    val cleanedF = sc.clean(f)
    new MapPartitionsRDD(
      this,
      (context: TaskContext, index: Int, iter: Iterator[T])
         => cleanedF(iter),
      preservesPartitioning)
}
```

5.1.2 Dependencies and Computations

A partition is the basic unit of an RDD, and dependencies describe the relationships between partitions of the parent and the child RDD. There are two types of dependencies: Narrow-Dependency and Shuffle-Dependency. For narrow dependencies, a child partition depends on a small number of partitions from the parent RDD. For shuffle dependencies, on the other hand, a child partition depends on a large number of partitions in the parent RDD.

The computation of an RDD is delegated to the MemoryStore or the DiskStore through the process of unrolling, in which the MemoryStore or the DiskStore iterates through the elements in a given partition, which is a chained-action that causes all the dependent partitions to be computed if not already. As illustrated in Figure 5.2, the computation of partition 1 in RDD3 causes the materialization of partitions in the parent RDDs 1 and 2. In a sense, the Spark framework is essentially a distributed memory system.

The mechanism of the MemoryStore or DiskStore during computations is shown in Algorithm 3. Depending on whether the memory or the disk is used, the partition is either unrolled until the maximum memory is reached or written directly to disk, and this process is synchronous. What is interesting is how it handles the data that exceeds the memory limit. If disk is used, it must first write the entire content to the disk, then returns a memory-mapped image of the file. Else, the memory store must release the references to the previously unrolled elements. If the

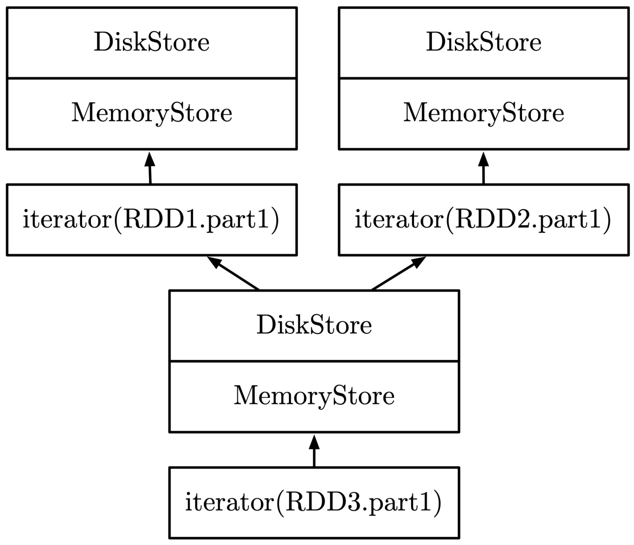


Figure 5.2: Computation of partition 1 in RDD3 that depends on partition 1 in RDD1 and RDD2

user still keeps a reference to the data (i.e., memory cannot be reclaimed by the garbage collector), an out-of-memory error is raised.

5.1.3 Persistence and Checkpointing

Persisting an RDD keeps a copy of the RDD in memory or on the disk while maintaining its lineage/dependencies. Checkpointing an RDD truncates the lineage graph of the RDD and saves the RDD on a reliable file system (Hadoop File System or local file system). The two are not rival concepts but rather complementary: checkpointing was introduced to resolve the lineage issue with RDD even though it can be entirely re-computed from scratch, it is useful when the lineage is too long or dependent on too many RDDs, such that re-computation would take a long time. Checkpointed RDDs are saved on a reliable location that is naturally resilient by replication. Checkpointed RDDs can also be read after the application terminates and carried on in the next application, whereas persisted RDDs will be removed.

For iterative algorithms, a new RDD is created at every iteration, resulting in a long string of dependencies that must be truncated using checkpoints. This is further discussed in Section 5.3.

5.1.4 Sampling

Having shown how the MemoryStore and DiskStore handle data, we can now understand why sampling data from an RDD is inefficient. By invoking RDD.sample(), a

Algorithm 3 Simplified illustration for unrolling an RDD partition

```
 1: source ← iterator of partition p
 2: output ← iterator of partition p
 3: if Use Memory then
 4:     while source.hasNext() & Enough Memory do
 5:         output.add(source.next)
 6:         Reserve memory if needed
 7:     if !source.hasNext() then
 8:         Return completely unrolled output iterator
 9:     else
10:         Return partially unrolled output iterator
11: else
12:     Unroll source iterator to file
13:     Return file stream of the memory-mapped file
```

new RDD is created by iterating through the entire parent RDD. Although drawing a sequence from a probability distribution is expensive, and efforts have been made to minimize it by using a method called Gap-Sampling [16], a much greater cost comes from the materialization of the entire parent RDD and the memory pressure when there is not sufficient physical memory to hold the data as discussed above.

The root of the problem is the granularity of the RDD (i.e., in partitions instead of records) and the sequential-access (i.e., as opposed to random-access). Sampling only requires a subset of the dependent partition; therefore it is not efficient to compute the entire partition.

There is no easy solution to the problem, because there exists no explicit relation between the records in the parent and the child dataset, nor even between the parent-child partitions in the case of a *Shuffle-Dependency.* The state of the child dataset is entirely undetermined.

5.1.5 RDDs on Accelerators

Machine learning using accelerators, such as Graphical Processors (GPUs), has become the main trend in recent years. Accelerated clusters have scaled-up and concentrated processing power, as opposed to a scaled-out cluster with less computationally intensive nodes. This has several significant implications on the practicalities of RDDs. As the compute-to-memory ratio is higher, applications run on fewer nodes with less main memory. With less main memory comes higher memory pressure, and data is more likely spilled to disk storage. Since the total device memory must be less than or equal to the main memory, there is even more stress on the device memory. This results in the starvation of accelerators.

There exist GPU implementations of the RDD abstraction [28] [83], which takes care of the data management between the CPU and the GPU, and the mapping of data to GPU kernels. However, it does not solve the fundamental issues concerning how the data is loaded, nor does it improve sampling efficiency. Moreover, mapping data to GPU kernels is very restrictive for programmers, since users cannot utilize the interface provided by the machine learning libraries. It is more practical to provide a handle to the GPU data, and leave the choice of programming interface and library to programmers.

5.2 Immutable Data - MapRDD

The main immutable data in machine learning is the input dataset that is in the form of text, image, audio or video, etc., which needs to be loaded from the storage and transformed into the in-memory data structure that is expected by the machine learning algorithm. This type of immutable data can be represented as a Resilient Distributed Dataset (RDD) as introduced in Section 5.1. Section 5.1 explored in-depth how an RDD works, and the issues for the use of RDDs on accelerators and machine learning applications. We found that the RDD is inefficient for machine learning due to the coarse granularity and the synchronous sequential-access of the dataset. With this knowledge, our goal is efficient handling of data that exceeds physical memory capacities for both homogeneous and heterogeneous architectures, with an application for stochastic processes. We present the design of the new **MapRDD**, which exploits the implicit relations between data records in *map()* transformations; we describe the design of *MapRDD* in the remainder of this section.

5.2.1 New MapRDD vs. MapPartitionsRDD

As explained in Section 5.1.4, the dataset granularity is limited by the non-explicit relation between the parent and the child datasets. There is, however, an implicit relation between the records by the *map()* transformation due to the syntax of the *map()* function $map : f(A) \to B$.

In the current implementation of Spark, both *map()* and *mapPartitions()* transformations produce a MapPartitionsRDD, in which the data granularity is kept at the partition level, such that it is consistent with other data transformations (such as *sortByKey()*, *groupByKey()*, *cogroup()*, etc.).

We introduce a new *MapRDD* that exploits the implicit relations of $map : f(A) \to B$. Figure 5.3 shows a layout of the architectural differences of the original RDD implementation and the new asynchronous *MapRDD* implementation. From the top down, they are: (I) User interface, an iterator that draws items from the dataset; (II) Memory abstraction that describes the dataset; (III) Memory/Disk

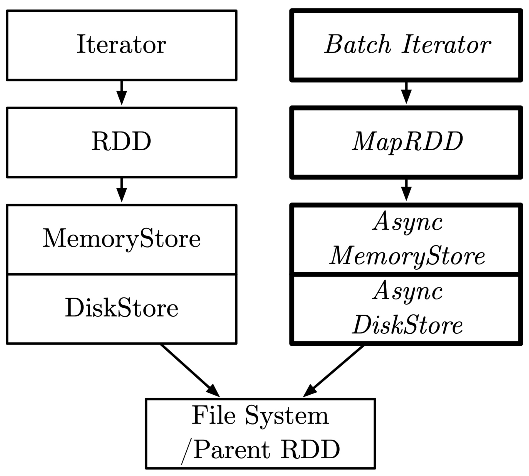


Figure 5.3: Overall Architecture. Left: Original RDD Implementation; Right: New SampleRDD Implementation

Store that manages data objects in memory and on disk; (IV) Parent dataset, or the underlying file system at the source level. The new *MapRDD* is an extension to the original RDD at all levels except for the file system. It preserves backward capabilities of the original RDD; as such the new RDD can be the base/parent RDD of the original RDDs.

5.2.2 Random Access and Sampling

The RDD supports an interface that iterates the elements in a one-by-one manner, rather than in a randomly-accessible fashion. The primary reason for this is that the state of the dataset is undetermined. The other reason for an iterator interface is that the iterated records can be safely discarded and recycled by the garbage collector; whereas in a randomly-accessible collection (e.g., arrays), memory cannot be recycled as each of the records is referenced.

With the implicit relation of *map* transformations, the size of a child *MapRDD* is known to be the same as its parent. Therefore, random-access to individual records is possible by applying the transformation function to the chain of dependent records.

We have seen in Section 5.1.4 how sampling is inefficient by iterating the entire dataset. With the random-access made possible by record-wise granularity in the new *MapRDD*, it is now possible to draw sample records randomly without materializing the complete dataset.

In addition, we extend the *iterator* interface to draw batches of records. It not only permits direct sampling from the current dataset, which not only bypasses the creation of a child dataset, but also provides opportunities for the sampling

Listing 5.2: Simplified implementation of parallel sampling

```
def parallelSampling(partitionSize, sampler) : Array[Int] =
   {
    (0 until partitionSize).par.map(i => {
        if (sampler.sample()) (i, 1)
        else (i, 0)
    }).filter(e => e._2 > 0)
}
```

algorithm and data loading to be carried out asynchronously.

5.2.3 Parallel Sampling for Large Partitions

The sampling process consists of a series of independent tests from a probability distribution with known parameters. The cost of computing the probabilities is relatively expensive and therefore Spark has been seeking algorithmic accelerations. An example of this is Gap-Sampling [16], as mentioned previously.

Sequential sampling is implemented in Spark, since the number of tasks in Spark is determined by the number of partitions (as explained in Section 5.1.1), and each task takes a single processor by default. For heterogeneous architectures, the dataset is partitioned by the number of accelerators; therefore there are far fewer but larger partitions. Sequential sampling large partitions is not efficient due to the imbalance in the number of partitions and the number of CPU cores.

With the size of the child dataset known in the new *MapRDD*, a parallel sampling algorithm can be implemented as shown in Listing 5.2. The *parallelSampling()* function takes the partition size and the sampler as arguments, and produces a parallel collection of indices from 0 to the partition size; for each of the indices, the sampler is invoked to decide if the index should be sampled (i.e. 1 for positive, 0 for negative); a final set of sample indices is produced by filtering the sampler outputs (i.e. greater than 0).

5.2.4 Asynchronous MemoryStore

The MemoryStore and the DiskStore are core components of the Spark framework, where the computation and the memory management of the RDD take place; as described in Section 5.1.2. We have also discussed how a synchronous MemoryStore can be in-efficient for modern computer architectures and for datasets that exceed physical memory capacities, especially for stochastic applications.

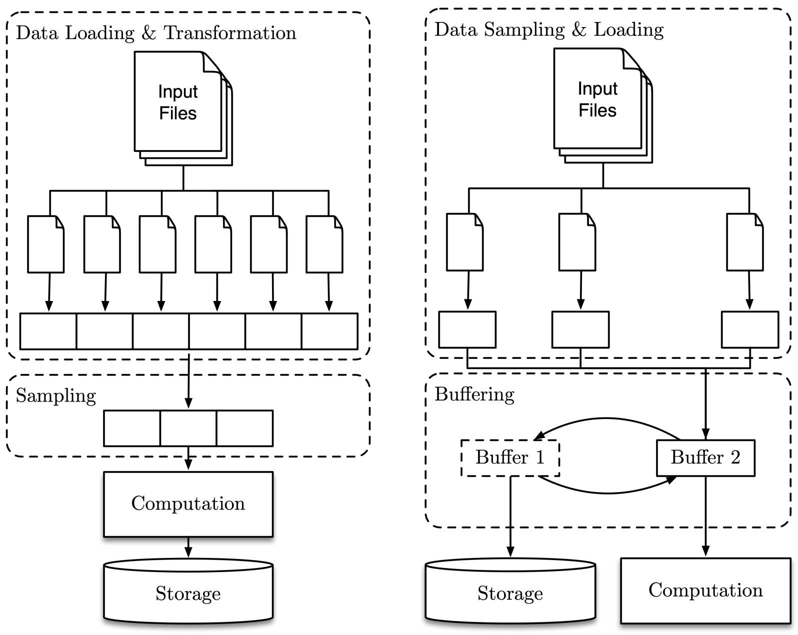
Algorithm 4 illustrates a simplified implementation of an asynchronous

Algorithm 4 Asynchronous implementation of the MemoryStore

```
1: $AllRecords \leftarrow Collection[Id, Dependency]$
2: $Saved \leftarrow Map[Id, FilePath]$
3: $Buffered \leftarrow Map[Id, Record]$
4: $NextBatch \leftarrow wait\ for\ ReadThread$
5: procedure ReadThread
6:     loop
7:         Wait for signal
8:         $batch \leftarrow Sample(AllRecords)$
9:         $toRead \leftarrow batch \cap Saved.elements$
10:        $inMemory \leftarrow batch \cap Buffered.elements$
11:        $toCompute \leftarrow batch - toRead - inMemory$
12:        $Read(toRead)$
13:        $Compute(toCompute)$
14:        $Buffered.add(toCompute, toRead)$
15:        $NextBatch \leftarrow batch$
16:        $Upload(NextBatch)$
17:        Signal WriteThread
18: procedure WriteThread
19:     loop
20:         Wait for signal
21:         $toWrite \leftarrow getRecordsOutsideBufferSize(Buffered)$
22:         $Write(toWrite)$
23:         $Saved.add(toWrite)$
24:         $Buffered.remove(toWrite)$
25: function nextBatch
26:     $ret \leftarrow NextBatch$
27:     Signal ReadThread
28:     return $ret$
```

MemoryStore. The workhorses of the MemoryStore are the *ReadThread* and the *WriteThread* that run asynchronously in the background while the executor computes the user-function on the next batch. As the user-function invokes the *nextBatch()* function, it immediately returns the pre-fetched batch, and signals the *ReadThread* to prepare the next batch. The *ReadThread* first samples a list of records to be computed, and cross-references any records that may have been buffered in memory or saved to disk. It then computes the record and its dependent records from the parent dataset, and reads the saved records from disk. Lastly, the *ReadThread* signals the *WriteThread* before setting the value of the *NextBatch*. The *WriteThread* in turn checks if the buffer has exceeded its limit and writes any unsaved records to the disk.

5.2.5 Everything Put Together



(a) RDD (b) MapRDD

Figure 5.4: Data Sampling with Regular RDD (left) and MapRDD (right).

We compare how the original RDD and the new *MapRDD* work in Figure 5.4. Originally, the input data must be fully loaded in the main memory, and a small sample is selected to be consumed by the training process, as shown in Figure 5.4a. With *MapRDD*, only the sparsely selected inputs are loaded into a buffer in the main memory, shown in steps 1-3 of Figure 5.4b. *MapRDD* adopted a double-buffering strategy, while the contents in one buffer is being used for training, the other is saved in storage and recycled for the next batch, as shown in steps 4 and 5 of Figure 5.4b.

5.2.6 Evaluation

We evaluate the new MapRDD on the ImageNet [31] dataset with AlexNet [41] on Caffe. The dataset consists of 1.3 million resized images (256 $\times$ 256 pixels), which is 19GB uncompressed. The experiment setup is explained in Section 5.2.6, and the results are evaluated in terms of overall runtime, CPU utilisation and GPU utilisation; these results can be found in Sections 5.2.6, 5.2.6 and 5.2.6 respectively.

Experimental Setup

The experiments are carried out on a standalone workstation with a NVIDIA Tesla K80 card (only a single GPU is used), so that there is no model synchronisation overhead; the specification of which is listed in Table 5.1.

The user code is a modified version of the SparkNet [58] toolkit. The main loop implemented with the original Spark API is shown in Listing 5.3, and the new implementation with the new MapRDD is shown in Listing 5.4. The main differences are the main loop inside the *foreachPartition* structure in line 4, and the new batch interface in line 5 of Listing 5.4.

We run the experiments with various data sizes and memory cache settings, which are listed in Table 5.2; the batch size is set to the default value (i.e. 256) in the reference to Caffe training model [14]; all experiments are repeated 5 times. In the original Spark implementation, the size of a partition cannot exceed 4GB, as the indexing limit is set to the maximum value of an 32-bit integer. Since we are consolidating the data into a single partition for the GPU, we are only using 20% of the ImageNet dataset due to the limit of memory capacity. For caching methods, the 'Memory & Disk' mode uses memory as much as possible until it spills to the disk; the 'Disk-Only' mode does not cache in memory, to simulate a short-of-memory situation; the 'Async' mode is the new mode that saves data asynchronously in the new MapRDD.

Table 5.1: System Configuration

CPU	Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz
Memory	DDR4, Capacity:128 GB, Speed: 2400MHz
Primary Disk	Samsung SSD 850 PRO 512GB
Secondary Disk	TOSHIBA HDWE160 (6TB, 7200RPM)
Accelerator	NVIDIA Tesla K80

Listing 5.3: Main Training Loop with current Spark API

```
val trainRDD = new MapPartitionsRDD()
for (i <- 0 until iters) {
  val sampleRDD = trainRDD.sample()
  sampleRDD.foreachPartition(
    trainIt => {
      solver.step(trainIt)
    })
}
```

Table 5.2: Experimental Settings

Experiment	Data Size	Cache Method	Iterations	Batch Size
1	5%	Memory & Disk	20	256
2	10%	Memory & Disk	20	256
3	15%	Memory & Disk	20	256
4	20%	Memory & Disk	20	256
5	5%	Disk Only	20	256
6	10%	Disk Only	20	256
7	15%	Disk Only	20	256
8	20%	Disk Only	20	256
9	5%	Async	20	256
10	10%	Async	20	256
11	15%	Async	20	256
12	20%	Async	20	256
13	50%	Async	20	256
14	100%	Async	20	256

Listing 5.4: Main Training Loop with new MapRDD

```
val trainRDD = new MapRDD()
trainRDD.foreachPartition(
    batchIt => {
        for (i <- 0 until iters) {
            solver.step(batchIt.nextBatch)
        }
    }
})
```

Overall

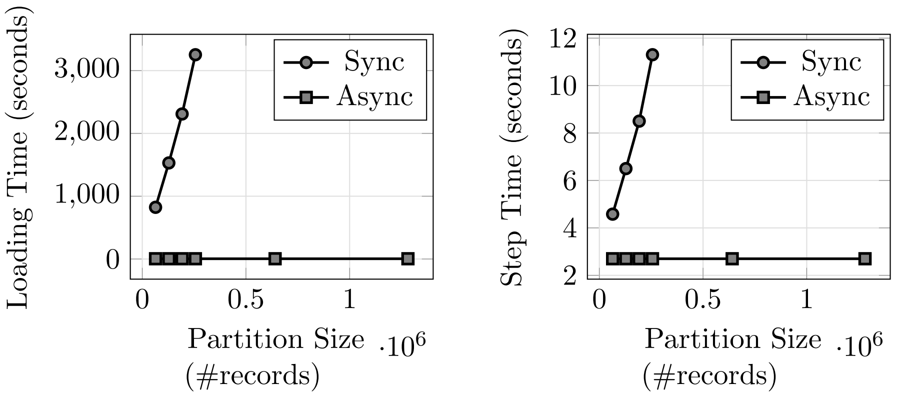
Table 5.3 lists the runtime results corresponding to the experimental settings in Table 5.2. The loading time includes all the time spent from the initialisation of the application until the first training step; the average step time is the averaged runtime for each training step. Figures 5.5a and 5.5b are direct comparisons of the loading time and step time for the synchronous method in ‘Memory & Disk’ mode and the asynchronous method with the new MapRDD.

For the first set of experiments (i.e., 1-4) that run in ‘Memory & Disk’ mode, the initialisation takes significant time (i.e., more than 13 minutes for 5% of the dataset) until the training finally begins, which could have been used to train for 180-280 steps. As shown in Figures 5.5a and 5.5b, both the loading time and the

Table 5.3: Overall Runtime

Experiment	Loading Time (sec.)	Average Step Time (sec.)
1	823.4	4.6
2	1530	6.5
3	2309.5	8.5
4	3252.9	11.3
5-8	failed	failed
9	4.2	2.7
10	4.2	2.7
11	4.2	2.7
12	4.2	2.7
13	4.2	2.7
14	4.2	2.7

step time increase near linearly as the size of the partition increases; the gradient starts to grow after 15% of the ImageNet dataset (i.e., 256k records), caused by the memory pressure; this is discussed in Section 5.2.6.



(a) Average loading time (b) Average step time

Figure 5.5: Average loading and step time across different partition sizes for synchronous (experiments 1-4) and asynchronous (experiments 9-14) methods.

For the third set of experiments (i.e., 9-14) in asynchronous mode, the loading time is almost negligible compared with the loading time in the 'Memory & Disk' mode; a 1.7-4.2× speedup is observed in training steps for up to the partition size limit of 4GB. Both the loading time and the step time are constant in spite of the increase in data size (see Figures 5.5a and 5.5b). This demonstrates that the loading time can be totally avoided by lazy-loading of data records; the asynchronous sampling and memory transfers by the new MemoryStore (see Algorithm 4) are effective, which kept the step time constant, even for a full size dataset such as the ImageNet on a single machine.

For the experiments run in ‘Disk Only’ mode (i.e., 5-8), they failed with the output size exceeding the maximum value of the integer type (i.e., 2^{32}) while trying to write the partition to the disk; this is because the images are expanded 4 times in size as every pixel byte is converted to a 4 byte floating-point number, and a single file cannot exceed the limit of 4GB (by the limit of 2^{32} bytes). The memory usage is also reflected in Section 5.2.6. This implies that the dataset must be split into small partitions, or the application would fail. For large data items, such as high-resolution images and videos, a partition may contain very few items limited by the size of 4GB, but in large quantity. Sampling from millions of partitions and mapping these partitions to devices is not efficient. Since the new MemoryStore (see Algorithm 4) manages data in a per-record fashion, it no longer poses a size limit on the partitions, and it is therefore more suitable for managing large items.

CPU Resource Utilisation

Table 5.4 lists the peak memory and CPU usage during loading and training, corresponding to the experiments listed in Table 5.2.

In terms of CPU memory, peak memory usage is much higher for the synchronous method than the asynchronous counterpart, as expected. As shown in Figure 5.6a, the committed memory (i.e., size of the JVM heap) during training steps increases rapidly as the partition size increases for the synchronous method (i.e., experiments 1-4), whilst the memory usage of the asynchronous method (i.e., experiments 9-14) is near constant. For experiment 4, the size of the heap of the Java Virtual Machine has almost reached the limit of the physical memory capacity, which cannot grow any further, therefore causing the loading and training to slow down as seen in Section 5.2.6. In our experiments, the peak memory usage of the asynchronous method is reduced by 96% during training steps compared with the synchronous method.

In terms of CPU processing cycles, the usage is stable for both synchronous and asynchronous methods. During loading, the synchronous method takes up a significant amount of CPU cycles (as much as 70%), which is freed up by the asynchronous method (to only 6%). During training steps, the parallel sampling algorithm (see Section 5.2.3) makes better use of the free CPU cycles (i.e., CPU utilisation rises from 6% to 11%) while the majority of the computation is delegated to the GPU.

Table 5.4: CPU Resource Utilisation

Experiment	Peak Memory (Loading) (GB)	Peak Memory (Training) (GB)	CPU (Loading) (%)	CPU (Training) (%)
1	48	58	70	6
2	48	76	70	6
3	51	88	70	6
4	50	89	70	6
5-8	failed	failed	failed	failed
9	2	2.5	6	11
10	2.5	2.5	6	11
11	2.5	2.5	6	11
12	2.5	3.5	6	11
13	2.5	3.5	6	11
14	3.5	3.5	6	11

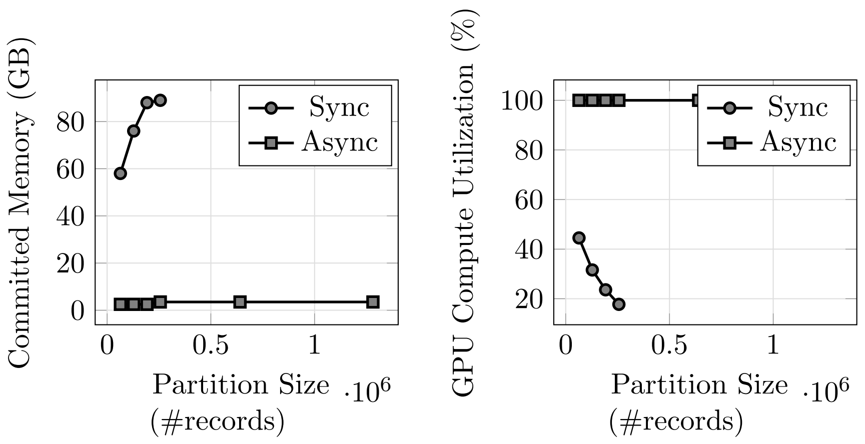
Table 5.5: GPU Resource Utilisation

Experiment	Average block time per step (sec.)	Average compute time per step (sec.)	GPU(%)
1	2.58	2.07	44.52%
2	4.48	2.07	31.60%
3	6.7	2.07	23.60%
4	9.6	2.07	17.74%
5-8	failed	failed	failed
9	0	2.07	100.00%
10	0	2.07	100.00%
11	0	2.07	100.00%
12	0	2.07	100.00%
13	0	2.07	100.00%
14	0	2.07	100.00%

GPU Resource Utilisation

Table 5.5 lists the average block time, the compute time, and the GPU utilisation during training steps corresponding to the experiments listed in Table 5.2. Figure 5.6b draws direct comparisons of GPU utilisation during training steps for the synchronous (i.e., experiments 1-4) and asynchronous (i.e., experiments 9-14) methods.

The average compute time per training step is the same for both synchronous and asynchronous methods across different sizes of the dataset, as the batch size is constant. For the synchronous experiments, the block time (mainly consisting of data sampling and data transfer) contributes to the low GPU utilisation, which drops exponentially as the partition size increases. For asynchronous experiments, the block time is negligible and the GPU functions near 100% of the time, because the data sampling and data transfer on the CPU is entirely overlapped with the compute time on the GPU.



(a) Commited memory (b) GPU compute utilisation

Figure 5.6: Average committed memory and GPU compute utilisation during training, across different partition sizes, for synchronous (experiments 1-4) and asynchronous (experiments 9-14) methods

5.3 Mutable Data - Distributed Key-Value Store

One of the biggest obstacles for Apache Spark is the lack of persistent memory support for mutable data because of the functioning programming paradigm. In the context of machine learning, this is mainly comprised of the weights/parameters of a machine learning model that updates and synchronises at every iteration as seen in Figure 5.1. So far we have been using Resilient Distributed Datasets (RDD) for immutable data; however, this is not feasible for volatile memory such as the weights of a machine learning model for the following reasons:

1. A new RDD is created every time the weights change. The changes in the weights can be represented as a series of RDDs, which will quickly consume all physical memory and causes data to be dumped onto slow secondary storage. The RDD can also be checkpointed, in which case the previous history of the RDD (i.e. the lineage) will be truncated; consequently, the main advantage of the RDD (i.e. the lineage) will be disregarded and the use of RDDs is in a disadvantage compared with private memory space used in MPI.

2. For data-parallel training, a replica of the same weights/parameters is required by every worker which is not well represented by RDD, as RDD is designed to represent a single image of the whole data. For this reason, an RDD consists of replicas of the same variable cannot be re-partitioned.

For the two reasons above, we conclude that the RDD is not a suitable representation for model replicas; therefore a new system is needed for the efficiency and scalability of the management of memory replicas in a distributed system. There are three different scenarios to be considered:

1. Bootstrap: initialisation of mutable variables when an executor is added;

2. Consistency: the copy of the variable is exactly the same on every executor at any given time;

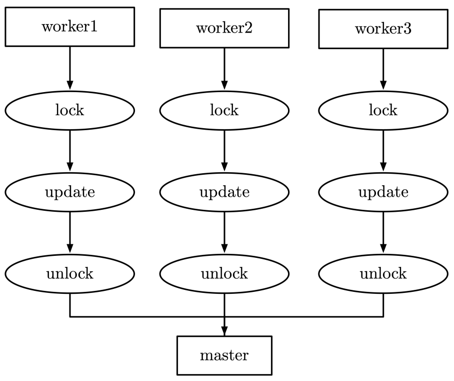
3. Out-of-sync: an executor may be left out due to dynamic allocation of tasks, as such the data held by this process is out-of-sync;

4. Exit: executor holding the data exits;

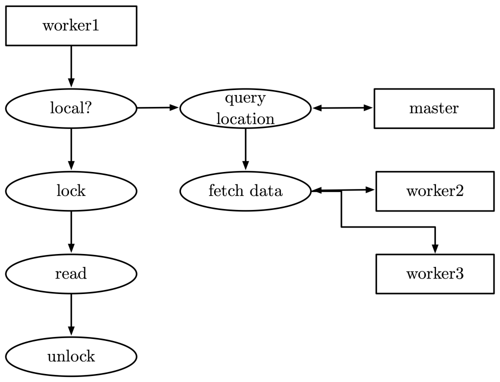
Since the Spark framework is based on a master/slave architecture and only the driver process on the root node survives the life span of the application, therefore it is the only reliable place for the persistent data to be kept. However, collecting/broadcasting data from/to all the workers is a costly operation that is bounded by the network bandwidth, which is identical to how the parameter server architecture works.

We propose a new proof-of-concept distributed key/value store in which the master process only acts as the coordinator which keeps a copy of the metadata of the data stored on each node, whilst the actual data is stored distributively across the cluster. Upon initialisation (i.e. bootstrap scenario), the executor invokes the *Get(key, state)* method in Algorithm 5; if the requested data is not present in the local memory, the worker will consult the master process about the location of the data and try to fetch it remotely (see Lines 15 and 17 of Algorithm 5). During execution, a global *state* of each iteration is passed to the executors and this is compared with the local *state* of the value by incorporating the *state* in the calculation of the hash of the key-value pair as shown in Line 12 of Algorithm 5. If the hash is present, the local copy of the value will be used; otherwise, an 'out-of-sync' scenario is detected, and a copy of the memory will be fetched from the remote workers as before during initialisation. When a key-value pair is updated, the *Update(key, state, value)* method is invoked, which removes the previous copy of the value from the master process and inserts the updated copy. To ensure that the value corresponding to a given key and state is unique, a checksum is performed and passed to the master, where it will be compared against duplicate copies in the cluster and an error will be raised if different copies of the same variable exist. As the worker process exits, it will inform the master process to destroy all the data it owns.

The interactions between the master and the workers are illustrated in Figures 5.7a and 5.7b. To update a key-value pair, each worker goes through the 'lock-update-unlock' procedure and informs the master of the location and checksum of the variable, which only requires communication between the worker and the master. To read a key-value pair, the worker will first check if it is present locally: if yes, no communication is required; if no, the variable is fetched from remote workers as described above.



(a) Updating a key-value pair



(b) Reading a key-value pair

Figure 5.7: Updating (left) and reading (right) a key-value pair.

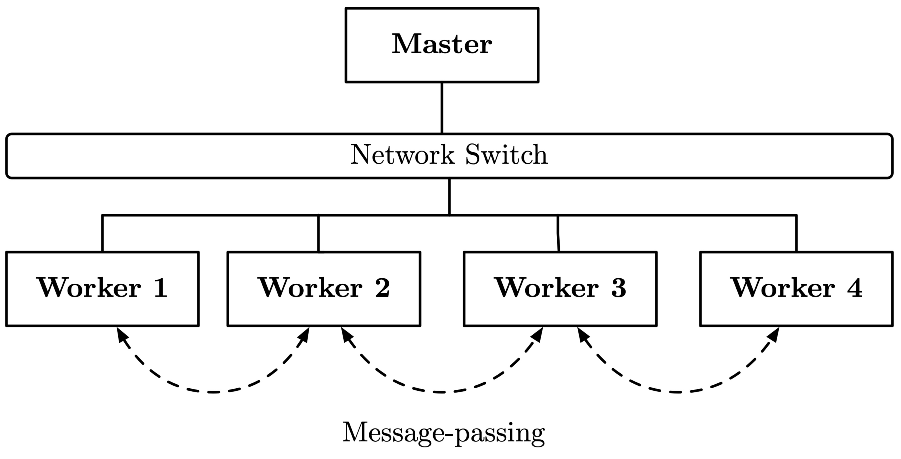
Algorithm 5 Distributed memory algorithm - client side

```
 1: $store \leftarrow ConcurrentHashMap$
 2: procedure UPDATE(key, state, v)
 3:     $hash \leftarrow computeHash(key, state)$
 4:     $tellMaster(remove(hash))$
 5:     $newHash \leftarrow computeHash(key, state + 1)$
 6:     $checksum \leftarrow computeChecksum(key, state + 1)$
 7:     $tellMaster(add(newHash, checksum))$
 8:     $store.put(newHash, v)$
 9: procedure GET(key, state)
10:     $hash \leftarrow computeHash(key, state)$
11:     $v \leftarrow null$
12:     if $store.contains(hash)$ then
13:         $v \leftarrow store.get(hash)$
14:     else
15:         $locations \leftarrow askMaster(hash)$
16:         while $v$ is null or $locations$ is not empty do
17:             $v \leftarrow fetch(random(locations))$
18:         $store.put(hash, v)$
19:     return $v$
20: procedure REMOVE(key, state)
21:     $hash \leftarrow computeHash(key, state)$
22:     $tellMaster(remove(hash))$
23:     $store.remove(hash)$
```

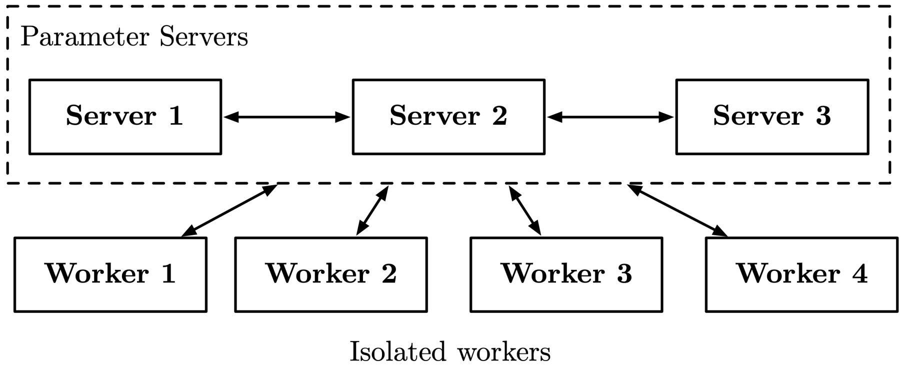
5.3.1 Consistency and Fault Tolerance

The idempotence property ensures the same result is produced for the same input parameter in functional programming, such as the Scala language used by Spark. To ensure this, the value for any variable must be unique and unchanging. This is accomplished by the versioning and checksum in our system. Versioning is implemented by a unique key-state combination after each update. The checksums of all the variables are gathered and stored by the master, which will be checked for each update, to makes sure a unique value corresponds to the same key-state pair, despite duplicated copies.

In our design, a variable is recoverable as long as a single copy survives amongst all nodes (master and workers); therefore two strategies are adopted for fault tolerance: duplication and checkpointing. For data-parallel machine learning, a copy of the weights/parameters is present on every worker; therefore the variable is recoverable unless all workers fail when a catastrophic failure occurs. Checkpointing writes the variable to permanent storage, which persists outside the life-span of the



(a) Hybrid all-reduce + key-value



(b) Parameter Servers

Figure 5.8: A comparison between the hybrid model (all-reduce & key-value store) and the parameter server architecture. The hybrid model requires communication between workers, whereas the parameter server architecture only requires communication between the servers and the workers.

application. In the situation of a catastrophic failure, the application can be resumed from the latest checkpoint of the variable.

5.3.2 Comparison with the Parameter Server Architecture

The combination of all-reduce and key-value store in this research is functionally equivalent to the parameter server architecture used in Tensorflow [1] and MxNet [8]. Both serve the purpose of persistent storage of the volatile variables (i.e. the parameters) for elastic computing.

The use of all-reduce and key-value store creates a hybrid computing model of traditional high-performance computing and elastic computing as shown in Figure 5.8a, in which all-reduce ensures efficient inter-process communication and the key-value store using master-slave model enables flexible dynamic horizontal scaling of the application.

Parameter servers are created with flexibility as a priority, where the variable is stored across a group of servers which can then be fetched by any workers as shown in Figure 5.8b. The updates to the variable are aggregated at the servers (i.e. pushing) and the combined results are broadcasted to the workers (i.e. pulling), which is similar to the 'reduce-broadcast' in Apache Spark that is discussed in Chapter 4. Unlike the original 'reduce-broadcast' method, the use of multiple servers effectively increases the bandwidth at server nodes and 'reduce-broadcasts' become 'scatter-gather' operations as shown in Figure 4.1b in Chapter 4; however, the total volume of traffic is unchanged and the ratio of the number of servers and workers has a significant impact on the communication performance.

In comparison, the hybrid model of all-reduce and key-value store offers the same flexibility as the parameter server architecture, while keeping an optimum performance as the MPI.

5.4 Summary

Data pre-processing as part of the pipeline for machine learning is often underestimated but plays an important role in computational efficiency. Managing data efficiently improves the processor utilisation and overall computational efficiency, and subsequently shortens the turnaround time.

The Resilient Distributed Dataset (RDD) is the core memory concept in Apache Spark, and this chapter has explored in depth how the RDD works and how it is largely obsolete in present-day machine learning applications. We identified that the source of deficiency originates from the coarse granularity and synchronous sequential-access of the dataset.

We present the new *MapRDD*, an extension to the Resilient Distributed Dataset (RDD) for *map* dataset transformations, and the new complementary asynchronous MemoryStore. Individual records in the child *MapRDD* can be accessed randomly and lazily. The data sampling and data transfers are managed asynchronously.

Through the experiments on the ImageNet dataset over different caching methods and data size settings, it is demonstrated that: (I) The initial data loading phase is redundant and can be completely avoided; (II) Sampling on the CPU can be entirely overlapped with the training on the GPU to achieve near full occupancy; (III) CPU cycles and memory usage can be reduced by more than 90% to allow other applications to run simultaneously; (IV) Constant training step time can be achieved, regardless of the size of the partition, for up to 1.3 million records in our experiments.

We also identified that RDD as a static memory abstraction is not suitable for volatile memory such as the model parameters in machine learning, because making a persistent copy of the parameters is both time and space consuming and an RDD comprised of replicas cannot be re-partitioned.

We propose the new proof-of-concept distributed key-value store, as an alternative means to manage volatile data efficiently and flexibly. We compare the hybrid model of all-reduce and key-value store with the parameter model architecture for managing volatile data in elastic computing. We show that the new hybrid model offers the same flexibility as the parameter server architecture while maintaining optimal communication performance.

Chapter 6

Algorithmic Improvements

Having optimised the implementation for distributed machine learning on Apache Spark (i.e. communication in Chapter 4 and memory management in Chapter 5), we now turn our attention to algorithmic improvements.

Overlapping the gradient computation and communication – asynchronous computation – is a way to reduce solution time. Increasing the batch size (i.e. the problem size) and subsequently the amount of computing resources (i.e. processors, memory and storage), known as 'weak scaling' in High-Performance Computing, is another way to reduce solution time.

The remainder of this chapter introduces two new methods: (i) An asynchronous SGD method based on non-blocking elastic all-reduce, see Section 6.1; (ii) A generalised fine-grained batch-size control method for large batch size training, see Section 6.2.

6.1 Asynchronous SGD

The inefficiency of distributed SGD compared with non-distributed SGD arises from the need for communication and synchronisation at each iteration for data-parallel training (the difference between data–parallelism and model–parallelism is explained in Section 2.1.2). Reducing the cost of communication and synchronisation directly impacts computational efficiency, and communication efficiency plays a key role in the performance model explained in Chapter 3.

For neural network algorithms, the computation and synchronisation can be overlapped due to the layer-by-layer structure, which is implemented in most neural network libraries. However, in a distributed setting, communicating in small messages incurs extra latency costs, which would be significant for a latency bounded computer network. As such, an asynchronous version of the SGD algorithm is needed.

Neural networks often consist of millions of parameters as shown earlier in

Table 1.1. The aggregation of these parameters is accomplished by the 'all-reduce' function in distributed computing platforms, which is explored in Chapter 4. Even with the communication optimised, machine learning on Spark still suffers from the overhead in synchronisation, in which the time spent for each training step depends on the slowest worker.

Major studies [1] [65] favour a Stale Synchronous Parallel (SSP) scheme for Stochastic Gradient Descent, which offers a trade-off between the training speed and the rate of convergence, relying on shared memory or a parameter server architecture. The bufferfly-mixing algorithm [85] proposed the application of partial weight aggregation and interleaved computation and communication rather than overlapping them, which leads to an 'out-of-sync by k steps', where k is the number of steps of the chosen 'all-reduce' algorithm. This creates a long lag in synchronisation and leads to greater inaccuracy as explained in Section 2.2.6.

Due to the synchronous data-flow design, asynchronous machine learning is not possible on Apache Spark, but a naïve parallelisation scheme has been used in SparkNet [58], to reduce the overhead in synchronisation by a less frequent global summation after a certain number of batches. With the non-blocking elastic all-reduce described in Chapter 4, this opens up possibilities for asynchronous machine learning on Spark.

6.1.1 Asynchronous SGD using non-blocking all-reduce

We modify the original update rule for synchronous stochastic gradient descent shown in Equation 6.1, and we propose two possible alternatives with non-blocking all-reduce, as shown in Equations 6.2 and 6.3. The symbols of the equations have the same meanings as previously defined, where $Q(w)$ is the objective function to be minimised with parameter w, symbol η is the learning rate, subscript i denotes the number of samples, and superscript j denotes the number of iterations.

For method 1, gradients are calculated based on the weights from the previous iteration (denoted by the term $\triangledown Q_i(w^{(j-1)})$ in Equation 6.2), and applied on the weights of the current iteration. For method 2, the weights and gradients are current to the current iteration, but the gradients are generated from the local/non-synchronised version of the weight (denoted by the term $Q_i(w^{(j,local)})$ in Equation 6.3).

This new method proposed works by making the following assumption: the difference between the un-synchronised local weight and the synchronised global weight (i.e. $\delta w = w^j - w^{j,local}$) is small enough, such that the gradient difference $\triangledown Q = \triangledown Q_i(w^j) - \triangledown Q_i(w^{j,local})$ is negligible. One way to minimise the chance of divergence is by regular synchronisation every certain number of iterations, but not as often so that it will become equivalent to the synchronous SGD. Another common interpretation for asynchronous SGD is that it carries more momentum than the synchronous SGD since the weights used for gradient calculation are from the previous iteration.

$$w^{(j+1)} = w^{(j)} - \eta \sum_{i=1}^{n} \bigtriangledown Q_i(w^{(j)})/n \tag{6.1}$$

$$w^{(j+1)} = w^{(j)} - \eta \sum_{i=1}^{n} \bigtriangledown Q_i(w^{(j-1)})/n \tag{6.2}$$

$$w^{(j+1)} = w^{(j)} - \eta \sum_{i=1}^{n} \bigtriangledown Q_i(w^{(j,local)})/n \tag{6.3}$$

Algorithm 6 Asynchronous SGD with non-blocking all-reduce method 1 (corresponding to Equation 6.2)

```
1: $w \leftarrow$ parameters of the network
2: procedure MAIN($w$)
3:     $j \leftarrow$ count of training steps
4:     for $i \leq numIterations$ do
5:         $\Delta w^j \leftarrow -\eta \bigtriangledown Q(w^{(j)})$
6:         AllReduce.Submit($\Delta w^{(j)}$, j)
7:         $\Delta w^{(j-1)} \leftarrow$ AllReduce.Get(j-1)
8:         $w^{(j+1)} = w^{(j)} + \Delta w^{(j-1)}$
9:         $j = j + 1$
```

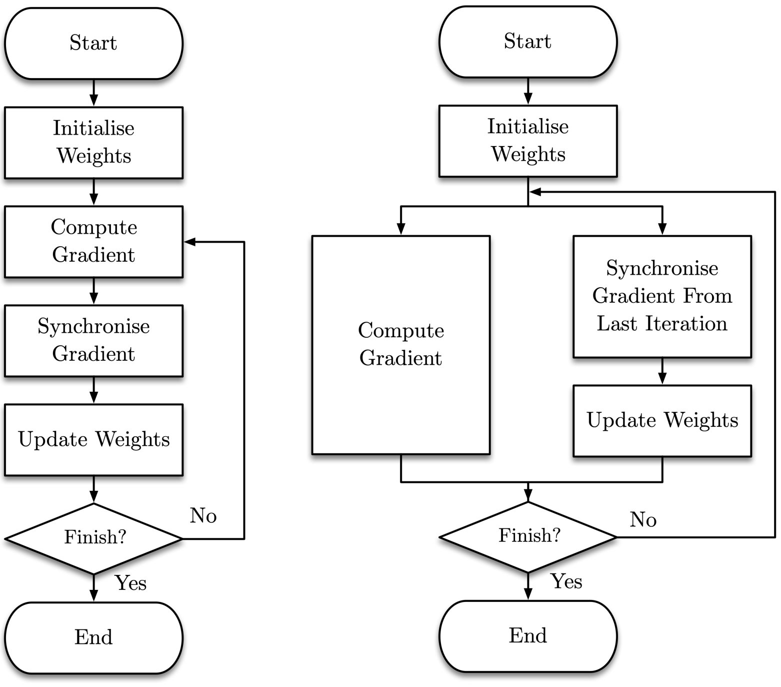
Algorithm 7 Asynchronous SGD with non-blocking all-reduce method 2 (corresponding to Equation 6.3)

```
1: $w \leftarrow$ parameters of the network
2: procedure MAIN($w$)
3:     $j \leftarrow$ count of training steps
4:     for $i \leq numIterations$ do
5:         $\Delta w^{(j,local)} \leftarrow -\eta \bigtriangledown Q(w^{(j,local)})$
6:         $w^{(j)} \leftarrow$ AllReduce.Get(j)
7:         $w^{(j+1)} = w^{(j)} + \Delta w^{(j-1)}$
8:         AllReduce.Submit($w^{(j+1)}$, j+1)
9:         $j = j + 1$
```

The implementations of the two methods are presented in Algorithms 6 and 7, for methods 1 and 2 respectively. For method 1, the gradient is first calculated in Line 5 and submitted for all-reduce in Line 6, after which the combined gradients from the previous iteration are retrieved in Line 7, and the current weight is updated in Line 8. For method 2, a local gradient is generated from the local version of the weights in Line 5, after which the global version of the weights are retrieved in Line 6 and updated with the local gradient in Line 7, which is lastly submitted for all-reduce in Line 8.

6.1.2 Theoretical Speedup

The differences between the synchronous method and the asynchronous method are illustrated in Figure 6.1. For the synchronous method (Figure 6.1a), the computation of gradient, the global weight aggregation and the weight update occur in a sequential fashion. For the asynchronous method (Figure 6.1b), the global weight aggregation and the weight update are overlapped with the gradient computation, which can potentially accelerate the training speed. However, the asynchronous method trades the rate of convergence for training speed, which may or may not result in an overall improvement. Therefore, an experiment is setup to test the rate of convergence of this method (see Section 6.1.3).



(a) Synchronous (b) Asynchronous

Figure 6.1: Comparison of the synchronous and asynchronous SGD methods.

Assuming the rate of convergence is comparable and the settings (e.g., learning rate, batch size, number of workers, etc.) are identical, this method provides a maximum speedup of 2× (as shown in Equations 6.4 and 6.5), in the ideal case where the computation of the gradient is totally overlapped with the global synchronisation of parameters (i.e. $T_{compute} = T_{comm}$ in Equation 6.4).

$$Speedup = \frac{T_{compute} + T_{comm}}{max(T_{compute}, T_{comm})} \tag{6.4}$$

$$\lim_{T_{comp} \to T_{comm}} Speedup = \frac{T_{comp} + T_{comm}}{max(T_{comp}, T_{comm})} = 2 \tag{6.5}$$

This, in turn, depends on: (i) The neural network model, the processor speed and the batch size, which can have an effect on the computation time; (ii) The cluster size, the parameter size and the network speed, which can have an effect on the communication time. The modelling of the relations between processing and communication speed for asynchronous SGD has been analysed earlier in Section 3.2.

6.1.3 Results

Experimental Setup

To evaluate our methods, a real-world neural network deployment was tested on a high-performance cluster, the specification of which is detailed in Table 6.1. The data to be classified is the ImageNet (ILSVRC2012) dataset [31], which contains 1.2 million images of 1000 classes, but only 10% (100 classes) of which was used in our tests. The AlexNet [41] and GoogLeNet [74] models were used for training, and they have a reported top-1 accuracy of 57.1% and 68.7% on classifying the ILSVRC2012 dataset respectively.

Table 6.1: Hardware and 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 and SSD)
Interconnect	Mellanox Technologies MT26428 (IB QDR via IPoIB)
Software	Centos/Linux-2.6, Hadoop 2.7, Spark-2.1.1

Experiment 1: Spark vs. MapRDD/All-Reduce

In the first experiment, we compare the original Spark implementation and the modified MapRDD/All-Reduce implementation (MapRDD and Elastic All-Reduce

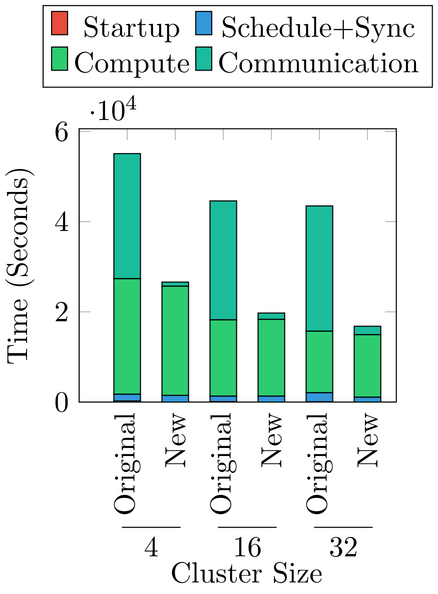
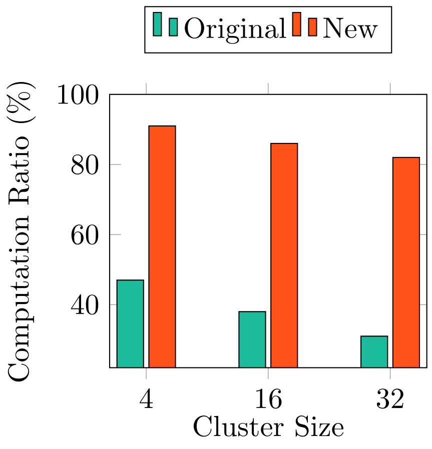


Figure 6.2: Breakdown of costs to reach 20% accuracy on GoogLeNet with 10% of the ImageNet dataset. Original: Spark and RDD implementation. New: MapRDD and all-reduce implementation.

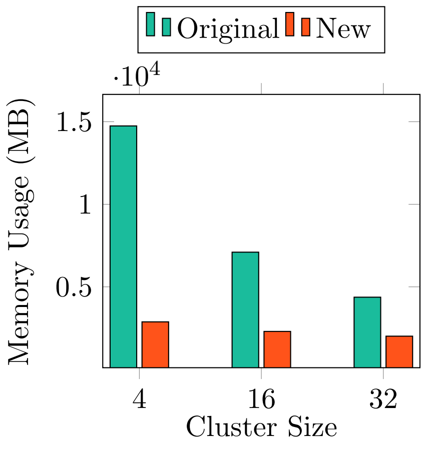
are introduced in Chapters 5 and 4 respectively). We train the first 100 classes of the ILSVRC2012 dataset with a constant batch size of 64 and a varying cluster size (i.e., 4-node, 16-node and 32-node). The candidates in the experiment are algorithmically equivalent; all configurations are the same except for the underlying implementations. We expect changes in the amount of overhead (i.e., startup, scheduling and communication) and the memory usage.

Figure 6.2 reports the breakdown of costs in training the GoogLeNet model to a 20% accuracy. There is a significant reduction in the communication costs, while the other costs (i.e., startup, computation, scheduling and synchronisation) remain comparable. The improvements in the execution time are mainly contributed by the employment of all-reduce described in Chapter 4. This is also reflected in the compute-ratio in Figure 6.3a, which is defined as the proportion of actual computation cost in the total cost; a rise from 31-47% to 82-91% in the compute ratio is observed in our tests.

The MapRDD described in Section 5.2 has a negligible effect on the total execution time, because a small subset of the original dataset was used and the processing time dominates. We expect a more notable impact with larger input and the use of accelerator cards. However, the improvements are still reflected in the amount of memory used in Figure 6.3b. An 80% reduction in the memory usage is observed in the 4-node setting where the memory pressure is highest in our test,



(a) Computation Ratio



(b) Memory Usage

Figure 6.3: A comparison between the original (Spark/RDD) and new (MapRDD/all-reduce) implementations for training the GoogLeNet model on the ImageNet dataset.

and subsequently, a 67% and 54% reduction is observed for 16-node and 32-node settings respectively. The advantage of the MapRDD would be more prominent if the full dataset is used, since only 10% of the dataset was used in our tests. Since the MapRDD only keeps the latest batch of samples in memory, the amount of memory used for storing the input data is invariant, the fluctuations in the memory usage for the MapRDD method reflect only the working memory.

Overall, a 2.0x-2.6× speedup is observed in our experiment (listed in Table 6.2), which increases as the cluster size increases. We expect little speed gain to be further extracted from this cluster since the computation ratio has reached 82-91% as aforementioned. However, since the speedup is mainly contributed by the improvements in communication, a greater speedup is expected if the execution time is communication dominant, which is the case for heterogeneous clusters with accelerator cards (such as Graphical Processing Units, GPUs). We tested GoogLeNet with a single GPU chip on an NVIDIA K80 graphics card using the Caffe and cuDNN library, and the average processing time for a batch size of 64 is 210ms. Assuming a processing time of 210ms per iteration, a speedup between 9.6x-11.2× is to be expected for the same tests carried out by substituting for $T_{compute}$ and $T'_{compute}$ in Equation 6.6 (also listed in Table 6.2).

$$Speedup_{sync} = \frac{T_{startup} + T_{compute} + T_{comm} + T_{sync}}{T'_{startup} + T'_{compute} + T'_{comm} + T'_{sync}} \tag{6.6}$$

Table 6.2: Speedup of Neural Network Training of the new method (MapRDD+AllReduce) with respect to the original implementation.

Cluster Size	CPU Speedup	GPU Speedup (Expected)
4	2.0	11.2
16	2.3	9.6
32	2.6	9.7

Experiment 2: Convergence Analysis for Asynchronous SGD

The second experiment is concerned with the asynchronous method using the non-blocking all-reduce proposed in Section 6.1. As discussed before, this method provides a maximum further speedup of 2× if the compute-to-communication ratio is 1, providing the rate of convergence does not deteriorate faster than the acceleration of the training speed. The compute-to-communication ratio can be manipulated by changing the batch size, the cluster size, the neural network model, etc. but it is based on the assumption that the convergence rate stays the same.

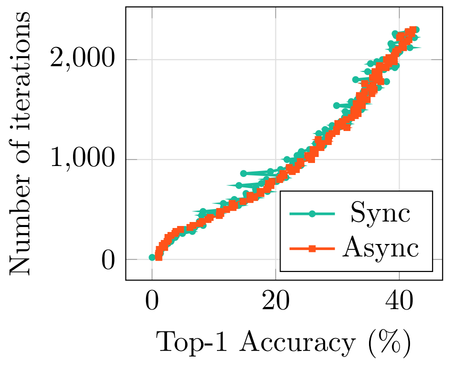
In this experiment, we investigate the rate of convergence of the new asynchronous methods (i.e., Algorithms 6 and 7) with 32 compute nodes and various batch sizes. As the processing power of the accelerators grows, so the execution time will become more communication dominant. The most likely solution to gain speedup is by increasing the size of the batch for each training iteration, to bring the compute-to-communication ratio closer to 1. However, it is important to understand how the rate of convergence will react to the changes in the batch size.

We test both Algorithm 6 and 7. Unfortunately, Algorithm 6 failed after several iterations as the error became too great. Therefore, only the results of Algorithm 7 will be shown in the remainder of this section.

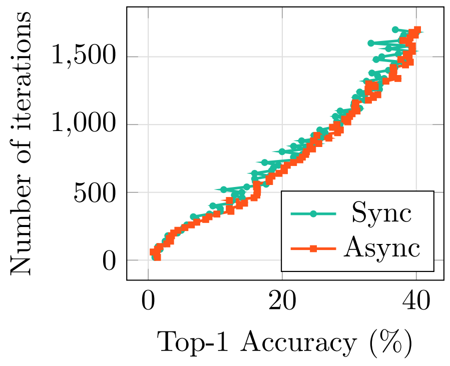
The convergence rate for the asynchronous method and the synchronous method are comparable, as shown in Figures 6.4a and 6.4b.

By comparing the same batch size, the accuracy with respect to the number of iterations, for the synchronous method and the asynchronous method, overlap with each other. It is also observed that the accuracy of the asynchronous method grows more steadily, whilst the accuracy for the synchronous method fluctuates.

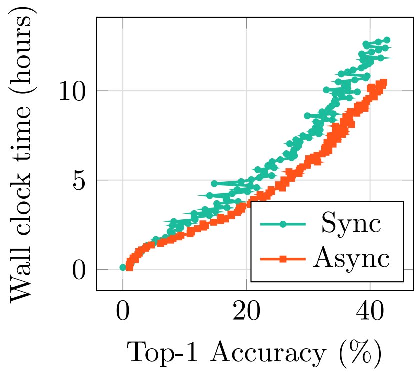
By comparing different batch sizes, it is observed that the rate of convergence for a larger batch size with respect to the number of training steps increases. In the case of accelerated clusters, this implies that faster convergence can be obtained by increasing the batch size without additional wall clock time since more computation can be overlapped with the communication. This can potentially provide more than



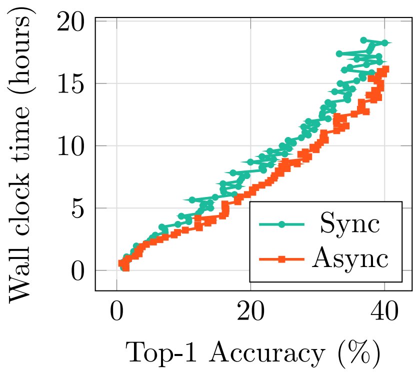
(a) Batch Size 64



(b) Batch Size 128



(c) Batch Size 64



(d) Batch Size 128

Figure 6.4: Top-1 accuracy against the number of iterations/time for training GoogLeNet on the ImageNet dataset.

2× speedup (the maximum speedup derived from Equation 6.5 in this Section) to reach the same accuracy with respect to the synchronous method.

With respect to wall clock time, the asynchronous method provides a 1.0-1.2× speedup over the synchronous method with the same batch size on a homogeneous cluster, as shown in Figures 6.4c and 6.4d. This is contributed to by the overlap between computation and communication. As shown in Table 6.3, the blocked all-reduce and synchronisation costs for the asynchronous method are reduced to zero, in return for a slight increase in the compute time. The increase in compute time is caused by the shared workload of the neural network training and all-reduce, which is not expected in a heterogeneous cluster with accelerator cards where the training is performed by the accelerator card and the all-reduce is performed by the CPU processors.

The amount of actual speedup is also dependent on the compute-to-communication ratio, as well as the convergence rate, as explained earlier in this section. For a

Table 6.3: A comparison of breakdown costs per iteration for the synchronous and asynchronous method (with MapRDD and all-reduce).

Method	Batch Size	Compute per iteration (sec.)	All-reduce per iteration (blocked, sec.)	Sync per iteration (blocked, sec.)	Iterations	Accuracy (%)	Duration (sec.)
Sync	64	16.1	2.4	1.6	2200	40	44220
Async	64	16.4	0.0	0.0	2200	40	36080
Sync	128	32.6	2.4	4.2	1700	40	66504
Async	128	34.2	0.0	0.0	1700	40	58140

homogeneous cluster, the execution time is computation dominant; however, for an accelerated compute cluster, it switches from computation dominant to communication dominant. For example, the compute-to-communication ratio for a batch size of 64 with our experiment setup is around 4, but it drops to 0.05 if an NVIDIA K80 graphics card is used (assuming a single GPU is used and the processing time is approximately 0.2 seconds). This means a batch size of 20×64 is needed for a compute-to-communication ratio of 1, which can be achieved by 20 mini-batches in a single step. It is a question of whether the convergence rate stays the same with a large sample batch (20×64).

Experiment 3: Statistical Analysis

We further evaluate the convergence of the asynchronous SGD method for different training sets, larger batch sizes and larger numbers of epochs, using the AlexNet model - a model with lower computational complexity and shorter training time. We compare the validation accuracy of the synchronous and the asynchronous method on 4 mutually exclusive sub-classes of the ImageNet dataset, the values of the best validation accuracy over 50 epochs of training are shown in Table 6.4. For 'async-strict', no synchronisation is performed after the initial warm-up; for 'async', synchronisation is performed every epoch.

To test if there is a significant difference between the three methods (i.e. sync, async-strict, async), we perform **paired student-t tests** on the results with a confidence level of 95%. A null hypothesis states that there is no significant pair-wise difference between the two sets of data. A p-value of 0.047 and 0.327 is obtained by comparing sync and async-strict and sync and async respectively. The test result suggests a significant difference between sync and async-strict, and no significant difference between sync and async, with a 95% confidence.

The results demonstrate a slight loss of accuracy of the asynchronous method compared with the synchronous method, with an average difference of 1.26%. A regular synchronisation at every epoch is enough to make up the differences, which is

Table 6.4: Best validation accuracy in 50 epochs (%, synchronous vs. asynchronous)

dataset	batch size	sync	async-strict	async
1	512	56.6	55.52	55.94
1	2048	47.88	46.96	48.34
1	4096	46.03	44.08	45
2	512	54.16	52.22	53.03
2	2048	46.1	44.55	45.13
2	4096	42.66	42.55	44
3	512	60.58	60.21	59.2
4	512	44.73	40.47	41.59

negligible compared with the synchronous method (e.g. 6,300 synchronisation steps are required for a batch size of 512 for 50 epochs). The async-SGD method with regular synchronisation at every epoch is effectively equivalent to the synchronous method with a 95% confidence. However, this is subject to change for datasets with different variance and sample size. For example, a classification task with a small variance is easier than one with large variance; subsequently, less frequent synchronisation is needed.

6.2 Generalised Fine-grained Batch-size Control

The batch size is one of the key factors in improving computational efficiency, which is one of the hyper-parameters of the mini-batch stochastic gradient descent method as introduced in Section 2.2.3. The maximum computational speed-up increases indefinitely with the batch size as shown in Section 3.5. However, the convergence rate and batch size have an inverse relation, whereby convergence slows down as batch size increases due to generalisation effects; as such the advantage of large batch size training diminishes, as explained in Section 2.2.7.

The purpose of using a dynamic batch size is to minimise the effect of generalisation and the convergence slow-down for large batch size training (as illustrated in Figure 1.3). The learning rate controls the step size of each update, which starts from a base value and decreases gradually as the current position approaches the saddle point (i.e. the minimum position) to avoid divergence. As observed empirically, a large batch size has generalisation effects and increasing the batch size is equivalent to decreasing the learning rate [73]; as such, a larger batch size can be employed as the training progresses. Successes have been demonstrated in scaling up the batch size to 64K-100K in [3] [21] [13] [73] and [81]. The problem with existing methods for dynamic batch-sizing is the changing problem size, in contrast to static partitioning on distributed platforms. Static partitioning poses a limit for the problem size due to scaling effects that result in coarse batch size control (for example: doubling the batch size every 10, 20 epochs). By adopting elastic computing, restrictions on the problem size can be lifted which allows for the dynamic batch-sizing method in a fine-grained manner.

We further explore a new generalised fine-grained dynamic batch-sizing method based on the existing techniques. There are two questions to be answered about this new method:

1. How do the rate of convergence and validation accuracy change in response to a finer-grained control of the batch size?
2. How to adapt to the changing problem size for maximum speed-up?

6.2.1 Fine-grained Batch Size Control

The first task is to understand how the rate of convergence and validation accuracy change in response to fine-grained control of the batch size as listed above, and this is to be demonstrated through experiments on real-world classification tasks.

We propose a monotonically increasing batch size based on the commonly used monotonically decreasing learning rate, which is simple and easy to interpret: as the step size should decrease as it gets closer to the solution. The alternative is the heuristic method (random walk) [81] which requires extra computations for heuristics and a **non-deterministic batch size** (problem size) which is difficult to implement in a distributed computing environment. Other options include a cyclic batch size to avoid a local minimum for a non-convex problem, similar to a cyclic learning rate [70]; however, such a method is highly experimental.

We choose the polynomial equation as the control function for the batch size, as it can also approximate other functions such as the logarithmic and exponential functions. Equation 6.7 is a generalised polynomial formula for calculating the global batch size, where B_0 and B_{max} are the initial and the maximum global batch size respectively, m and M are the current and target epoch respectively, and P is the power to the polynomial that controls how fast the batch size increases. The effective batch size for the running time and cost calculations (as in Equations 3.5 - 3.13) can be found by integration of dynamic batch size B in Equation 6.7 over the number of epochs m as shown in Equation 6.8, which is also dynamic at different points of training. The final effective batch size at the end of M epochs is given by Equation 6.9. The difference between the current batch size and current effective batch size the term $P+1$ in Equation 6.8, which means that the effective batch size grows $P+1$ times slower.

$$B = B_0 + (B_{max} - B_0)(\frac{m}{M})^P \tag{6.7}$$

$$B_{\mathit{eff}} = \frac{1}{m}\int_{m=0}^{m} B\ dm = B_0 + \frac{B_{max} - B_0}{P+1}(\frac{m}{M})^P \tag{6.8}$$

$$B_{eff,m=M} = B_0 + \frac{B_{max} - B_0}{P+1} \tag{6.9}$$

Different values for initial (B_0) and final batch size (B_{max}), power (P), and the number of epochs (M), could lead to the same effective batch size as given by Equation 6.8, and it is important to understand how these variables affect the convergence rate. As such, the majority of this research is to verify the convergence rate in response to these parameters, and there are five sets of control experiments to consider:

1. Same B_{eff}, B_0, M, different P, B_{max}.
2. Same B_{eff}, P, M, different B_0, B_{max}.
3. Same B_0, P, M, different B_{max}, B_{eff}.
4. Same B_{eff}, B_0, B_{max}, P , different M.
5. Same B_{eff}, B_0, B_{max}, P, M, with constant and decaying learning rate.

The result of the experiments above are presented in Section 6.2.3.

6.2.2 Workload Balancing

The second task is concerned with the challenge of changing problem size in the computational performance of the application. Our solution is to adopt the use of elastic computing which allows for dynamic resource allocation and co-ordinations, as shown in Algorithm 8, where line 9 is the core of the algorithm that determines the number of workers (N) in accordance with the global batch size set by Equation 6.7.

To adjust for the number of workers, there are two options: optimised for speed-up or for running costs.

To optimise for speed-up, we can simply use Equations 3.15 and 3.16 for the maximum speed-up for synchronous and asynchronous SGD respectively, which were previously derived in performance modelling in Section 3.5.

Algorithm 8 Dynamic scaling for SGD

```
1: $w \leftarrow$ parameters of the objectivefunction
2: $\eta \leftarrow$ learning rate
3: $M \leftarrow$ total epochs
4: $D \leftarrow$ size of trainingset
5: procedure SGD($w$, $\eta$)
6:     $m \leftarrow$ currentepoch
7:     repeat
8:         $B \leftarrow$ dynamic global batch size
9:         $n \leftarrow$ dynamic global resource
10:        $b \leftarrow B/n$
11:        $Step(w, \eta, b)$
12:        $m = m + B/D$
13:    until $m \leq M$
```

To optimise for running costs, the computation-to-communication ratio is introduced, which is the ratio between the latency for computation and communication (i.e. $\frac{T_{compute}}{T_{comm}}$); the higher the computation-to-communication ratio the lower the running cost. The computation-to-communication ratio for synchronous SGD is calculated by Equation 6.10, which is derived by dividing Equations 3.3 and 3.4, where γ is the processing speed (samples per second), α and β are the communication coefficients (smaller the better), B is the global batch size and N is the number of workers. The number of workers (N) for a particular computation-to-communication ratio (r) can subsequently be derived from Equation 6.10 as shown in Equation 6.11.

$$\frac{T_{compute}}{T_{comm}}, r = \frac{B}{\alpha\gamma N + \beta\gamma N^2} \tag{6.10}$$

$$N = \frac{-r\alpha\gamma + \sqrt{(r\alpha\gamma)^2 + 4Br\beta\gamma}}{2r\beta\gamma} = \sqrt{\frac{B}{r\beta\gamma}} \tag{6.11}$$

6.2.3 Results - Convergence Analysis

We analyse the convergence of the dynamic batch-sizing method using control experiments designed to understand how different variables affect the convergence rate in Section 6.2.1. In our experiments, we train the AlexNet model on the ImageNet dataset for 50-100 epochs. The solver settings for different batch sizes are listed in Table 6.5, where the 'learning rate warm-up' indicates the number of epochs taken for the learning rate to rise from the initial value of 0.01.

We list the control experiments, of which the hyper-parameters (i.e. B_0, B_{max},

P and B_{eff}) and the best validation accuracy are reported in Table 6.6. The actual batch size for control experiments 1-4 are reported in Figures 6.5a, 6.6a, 6.7a and 6.8a, and the top-1 error curves of the corresponding experiments in Figures 6.5b, 6.6b, 6.7b and 6.8b.

1. Experiment 1 (index: 1 - 4): Same B_{eff}, B_0, M, different P, B_{max}.
2. Experiment 2 (index: 4 - 6): Same B_{eff}, P, M, different B_0, B_{max}.
3. Experiment 3 (index: 6 - 9): Same B_0, P, M, different B_{max}, B_{eff}.
4. Experiment 4 (index: 9 - 11): Same B_{eff}, B_0, B_{max}, P , different M.
5. Experiment 5 (index: 12 - 19): Same B_{eff}, B_0, B_{max}, P, M, with different learning rate policy: (i) fixed learning rate of 0.01; (ii) polynomial decay with an initial value of 0.01 and a power of $P \times 2$; (iii) polynomial growth to the corresponding values in Table 6.5 with an initial value of 0.01 and a power of $P \times 2$.

Table 6.5: Solver settings

batch size	learning rate	LARS learning rate	learning rate warm-up
64	0.01	n/a	0 epoch
512	0.02	2	2.5 epochs
2048	0.02	8	2.5 epochs
4096	0.04	10	2.5 epochs

From experiment 1 with same effective batch size B_{eff} and changing scaling power P, it is demonstrated that as P, increases, the best validation accuracy increases, which suggests that the use of a small batch size at the beginning of the training is more important, as shown by the shapes of the dynamic batch size profile in Figure 6.5a.

For experiment 2 with same effective batch size B_{eff} but changing B_0 and B_{max}, it is demonstrated that the smaller the initial batch size, the higher the best validation accuracy, which matches the findings in control experiment 1. The differences in accuracy are more apparent when the error curves are examined in Figure 6.6b.

For experiment 3 with changing effective batch size B_{eff}, a 2.7% loss in top-1 validation error is observed over an increase of batch size from 64 to 4,096, instead of an 18% loss as previously shown in Figure 2.5. A comparison of the best validation accuracy over different effective batch size can be found in Table 6.7, and it is shown

Table 6.6: Experiment settings and best top-1 validation accuracy for dynamic scaling, with control variables highlighted.

index	B_0	B_{max}	P	B_{eff}	learning rate	epochs	accuracy
1	64	352	**0.5**	256	fixed	50	57.8%
2	64	448	**1**	256	fixed	50	58.2%
3	64	544	**1.5**	256	fixed	50	59.8%
4	64	640	**2**	256	fixed	50	60.2%
5	**16**	736	2	256	fixed	50	60.4%
6	**32**	704	2	256	fixed	50	60.0%
7	32	1472	2	**512**	fixed	50	60.6%
8	32	6080	2	**2048**	fixed	50	59.6%
9	32	12224	2	**4096**	fixed	50	57.5%
10	32	12224	2	4096	fixed	**25**	53.2%
11	32	12224	2	4096	fixed	**20**	50.3%
12	32	704	2	256	**poly-decay**	50	61.0%
13	32	1472	2	512	**poly-decay**	50	60.4%
14	32	6080	2	2048	**poly-decay**	50	59.0%
15	32	12224	2	4096	**poly-decay**	50	57.1%
16	32	704	2	256	**poly-grow**	50	60.2%
17	32	1472	2	512	**poly-grow**	50	59.7%
18	32	6080	2	2048	**poly-grow**	50	58.0%
19	32	12224	2	4096	**poly-grow**	50	58.4%

that dynamic scaling always results in a higher validation accuracy with the same effective batch size.

For experiment 4 with the same effective batch size B_{eff} and changing number of epochs M, a 4% loss of validation accuracy is observed as the number of training epochs is shortened by a half, and yet the validation accuracy is still 10% higher than that of a static batch size (53%(new) - 42%(original)). This implies the opposite is also true: as the number of epochs increases, the effective batch size can also increase. For example, if the number of epochs is doubled from 50 to 100 epochs, the overall effective batch size is multiplied by $(\frac{1}{2})^P$, for the same effective dynamic batch size at the same epoch number.

For control experiment 5 with a changing learning rate, it is demonstrated that a decaying learning rate improves the best validation accuracy for relatively small batch sizes ($B_{\mathit{eff}} = 256$), but it has a negative impact on relatively large batch sizes ($B_{\mathit{eff}} = 512, 2048, 4096$); and a growing learning rate has an opposite effect that

Table 6.7: Best top-1 validation accuracy of static and dynamic batch size for training AlexNet-ImageNet.

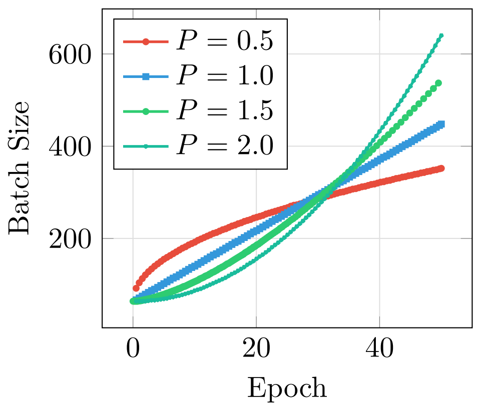
sub-classes	epochs	batch size	static	static-lars	dynamic ($P = 2$)
0-49	50	64	60.9	-	-
0-49	50	256	56.7	-	-
0-49	50	512	56.7	55.7	60.6
0-49	50	2048	48.9	55.0	59.6
0-49	50	4096	42.6	50.5	57.5
50-100	50	64	57.8	-	-
50-100	50	256	54.1		57.8
50-100	50	512	53.5	53.1	57.7
50-100	50	2048	52.8	52.6	57.5
50-100	50	4096	45.8	49.4	56.6
200-300	100	64	55.9	-	-
200-300	100	256	51.0	-	-
200-300	100	16K	-	-	53.9
300-500	100	64	58.2	-	-
300-500	100	256	55.2	-	-
300-500	100	16K	-	-	56.2

improves the validation accuracy for large batch size $B_{\mathit{eff}} = 4,096$. This demonstrates the delicacy of adjusting the learning rate and batch size simultaneously, as the changes in either have similar effects.

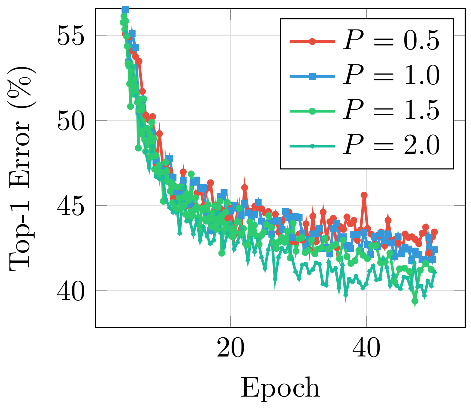
Table 6.7 compares the following methods on independent subsets of ImageNet: (i) standard/static batch size; (ii) Layer-wise Adaptive Rate Scaling (LARS); (iii) dynamic batch-sizing. The dynamic batch-sizing method consistently outperforms the other two methods in terms of best validation accuracy. It is also demonstrates that a bigger effective batch size (up to 16K in our experiments) can be used for a larger training set and a larger number of epochs as predicted by experiment 4.

Overall, the dynamic batch-sizing method consistently produces faster convergence and higher validation accuracy than a static batch size with/without adaptive learning rates. The experiments have also shown consistently that the validation accuracy is sensitive to the initial batch size. The outcome suggests the lower value for B_0 and the higher value for P, the higher the accuracy. However, this penalises the computational efficiency which is analysed in Section 6.2.4, and this also shifts the time distribution to the beginning of the training (i.e. more time is spent on small batch sizes and less time is spent on larger batch sizes). The best choice for these settings for different datasets remains the subject of further research.

Similar to the asynchronous stochastic gradient descent method presented in Section 6.1, the convergence results are subject to change for different training datasets (i.e. sample size and variance). However, the use of large batch size only applies to large sample pools and the ImageNet dataset is currently the largest dataset for object recognition tasks in computer vision. So far, our experiments have used a maximum subset of 200/1000 classes; further validation is required for the full dataset.

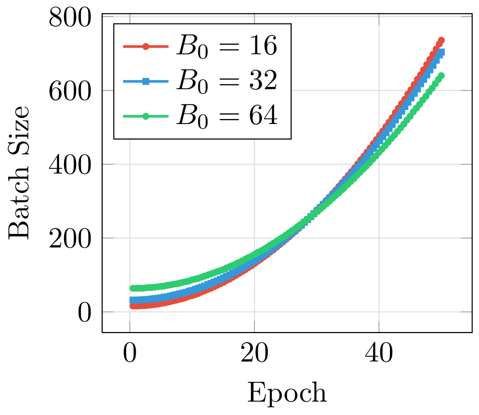


(a) Experiment 1 settings

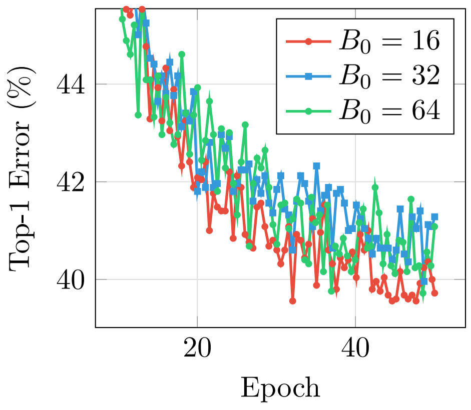


(b) Top-1 error vs. epoch

Figure 6.5: Experiment 1: Same $B_{\mathit{eff}} = 256$, $B_0 = 64$, $M = 50$, different P, B_{max}, for training AlexNet on 50 sub-classes of ImageNet.

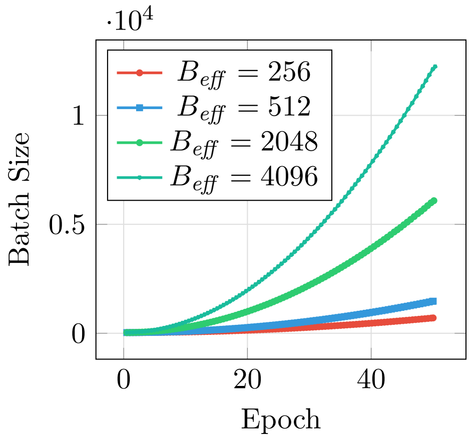


(a) Experiment 2 settings

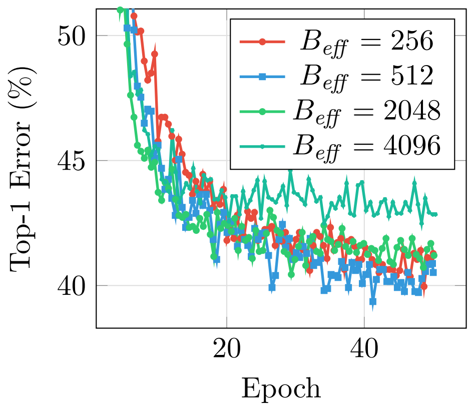


(b) Top-1 error vs. epoch

Figure 6.6: Experiment 2: Same $B_{\mathit{eff}} = 256$, $P = 2.0$, $M = 50$, different B_0, B_{max}, for training AlexNet on 50 sub-classes of ImageNet.

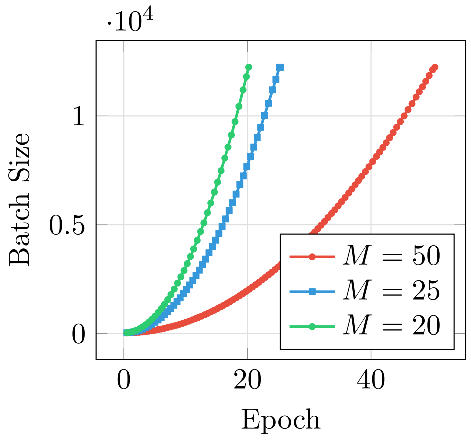


(a) Experiment 3 settings

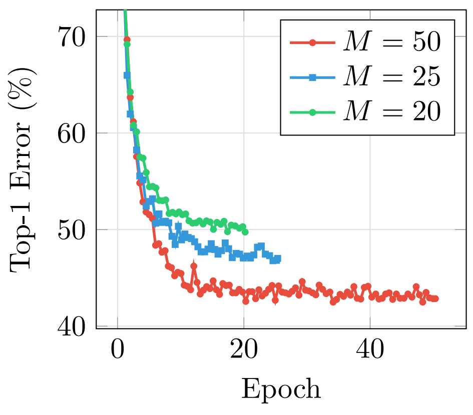


(b) Top-1 error vs. epoch

Figure 6.7: Experiment 3: Same $B_0 = 32$, $P = 2.0$, $M = 50$, different B_{eff}, B_{max}, for training AlexNet on 50 sub-classes of ImageNet.



(a) Experiment 4 settings



(b) Top-1 error vs. epoch

Figure 6.8: Experiment 4: Same $B_{eff} = 4096$, $B_0 = 32$, $P = 2.0$, different M, for training AlexNet on 50 sub-classes of ImageNet.

6.2.4 Results - Performance Analysis

We analyse the computational performance of the dynamic batch-sizing method using the predictive model derived in Chapter 3. For this, measurements for coefficients γ, α and β are required.

In Section 3.3, the processing speed γ has been measured to be 10 and 260 samples per second, for the CPU (Intel Xeon E5-2660 2.6 GHz) and GPU (NVIDIA Tesla K80) respectively for AlexNet-ImageNet. The all-reduce time has also been measured for a vector length of 10^6, 10^7 and 10^8, from which the values for α and β

can be extrapolated. For AlexNet with 65 million parameters, the values for α and β in Equation 3.5 are extrapolated as follows: $\alpha = 2.0$ and $\beta = 0.024$ for *ring all-reduce* over RDMA, $\alpha = 0.8$ and $\beta = 0.03$ for *butterfly all-reduce* over RDMA. With the processing speed γ and the communication coefficients α and β known, the running time, maximum number of workers and maximum speed-up can be predicted.

Tables 6.8 and 6.9 provide the computation-communication ratio and the maximum speed-up for CPU and GPU respectively. The computation-communication ratio is an indication of the running cost, the higher the ratio the lower the cost, as discussed in Section 3.7. The following observations can be drawn:

1. As the batch size increases, both the computation-to-communication ratio and the speed-up increase and this subsequently leads to reduced running time and cost.
2. For dynamic batch-sizing, it is shown that as the scaling power P increases, the maximum speed-up decreases and the computation-to-communication ratio increases. A loss of 15% in maximum speed-up is seen for $B_{\mathit{eff}} = 4,096$ and $P = 2.0$.

The observations above suggest a trade-off between the validation accuracy and the computational efficiency: the validation accuracy increases as seen in Section 6.2.3 as P increases, while the maximum speed-up and computation-to-communication ratio decreases. This can be mitigated by asynchronous SGD, where the speed-up is doubled compared with synchronous SGD and the computation-to-communication ratio is 100%, as shown in Tables 6.10 and 6.11. Yet, a decrease in maximum speed-up is still present as P increases for asynchronous SGD.

Table 6.8: Maximum number of workers (N_{max}), the computation-to-communication coefficient (r) and the maximum speedup (S_{max}) for synchronous training of AlexNet-ImageNet on the CPU (with $\gamma = 10$, $\alpha = 0.8$, $\beta = 0.028$).

B_{eff}	r	N_{max}	S (static)	r (p=1.0)	S (p=1.0)	r (p=2.0)	S (p=2.0)
256	51%	30	10.27	49%	9.85	47%	9.35
512	60%	43	16.03	57%	15.23	53%	14.27
2048	75%	86	36.64	72%	34.52	66%	31.87
4096	81%	121	54.09	78%	50.89	72%	46.83

Table 6.9: Maximum number of workers (N_{max}), the computation-to-communication coefficient (r) and the maximum speedup (S_{max}) for synchronous training of AlexNet-ImageNet on the GPU (with $\gamma = 260$, $\alpha = 0.8$, $\beta = 0.028$).

B_{eff}	r	N_{max}	S (static)	r (p=1.0)	S (p=1.0)	r (p=2.0)	S (p=2.0)
256	100%	1	1.00	68%	1.10	71%	1.18
512	23%	8	1.55	46%	1.61	58%	1.68
2048	37%	17	4.53	40%	4.33	50%	4.16
4096	45%	24	7.40	45%	7.01	52%	6.60

Table 6.10: Maximum number of workers (N_{max}), the computation-to-communication coefficient (r) and the maximum speedup (S_{max}) for asynchronous training of AlexNet-ImageNet on the CPU (with $\gamma = 10$, $\alpha = 0.8$, $\beta = 0.028$).

B_{eff}	r	N_{max}	S (static)	r (p=1.0)	S (p=1.0)	r (p=2.0)	S (p=2.0)
256	100%	19	19.16	100%	18.36	100%	17.42
512	100%	31	30.80	100%	29.21	100%	27.32
2048	100%	72	72.42	100%	68.12	100%	62.78
4096	100%	108	107.50	100%	101.04	100%	92.84

Table 6.11: Maximum number of workers (N_{max}), the computation-to-communication coefficient (r) and the maximum speedup (S_{max}) for asynchronous training of AlexNet-ImageNet on the GPU (with $\gamma = 260$, $\alpha = 0.8$, $\beta = 0.028$).

B_{eff}	r	N_{max}	S (static)	r (p=1.0)	S (p=1.0)	r (p=2.0)	S (p=2.0)
256	100%	1	1.18	100%	1.34	100%	1.45
512	100%	2	2.28	100%	2.32	100%	2.39
2048	100%	8	7.75	100%	7.45	100%	7.15
4096	100%	13	13.40	100%	12.72	100%	11.98

6.3 Summary

Further to implementation optimisations in Chapters 4 and 5, this chapter explores algorithmic improvements to accelerate deep learning training, by further reduction of communication costs using asynchronous methods and by large batch size training using dynamic fine-grained batch-size control.

A new asynchronous SGD algorithm using non-blocking all-reduce was proposed, which reduces the communication cost by overlapping the communication with computation, and we show a $2\times$ theoretical speed-up over the synchronous method. Through experiments of neural network training using AlexNet and GoogLeNet on the ImageNet dataset, we show statistically that the asynchronous method has the same effective convergence as the synchronous method with 95% confidence.

A method of large batch-size SGD training using fine-grained batch-size control was also proposed. Experiments on AlexNet-ImageNet show that the method consistently produces faster convergence and higher validation accuracy than a static batch size with/without adaptive learning rates. It is also shown consistently that the validation accuracy is sensitive to the initial batch size through control experiments. However, small initial values and slow growth in the batch size leads to a trade-off between the computational speed and validation accuracy and causes a shift in time distribution to the initial stage of the training.

These two methods above are implementation-independent and can be easily adopted by other elastic distributed computing systems. However, the convergence results are susceptible to the characteristics of the training dataset, especially the variance and the number of training samples per class (the smaller the variance, the easier to classify). As mentioned in Section 2.1.3, ImageNet is currently the biggest and the most popular dataset to use for computer vision. Google Open Image is a larger dataset with 9 million images, but with multiple image-level labels per image, it is not suitable for object recognition tasks (where here is a single object per image).

Chapter 7

Conclusions and Future Work

Machine learning is a powerful tool that allows us to make better and faster decisions in a data-driven fashion based on training data. Neural networks are especially popular in supervised learning due to their ability to approximate auxiliary functions. However, building these models is computationally intensive, and can take years to complete on a conventional CPU-based computer. Such a long turnaround time makes business and research impossible using these models. This research seeks to accelerate this training process through parallel and distributed computing using High-Performance Computing resources.

To understand the bottlenecks and limitations to distributed machine learning, analysis has been performed on the characteristics of the Convolutional Neural Network (CNN) algorithm and the Stochastic Gradient Descent (SGD) method, and their implementations on a distributed computing platform. A predictive model for data-parallel CNN+SGD has been devised and the theoretical maximum speed-up is expressed as a function of the batch size for a fixed computing setup. This gives an upper limit to which computational acceleration is achievable through parallel and distributed computing, and the predictive model dictates four key factors to achieve the maximum acceleration: the convergence rate, the batch size, the computation efficiency and the communication efficiency. These four performance factors provide insights into which distributed machine learning tasks can be improved through algorithmic and implementation optimisations.

The Message Passing Interface (MPI), data-flow frameworks (i.e. Apache Hadoop and Spark) and the parameter server architecture have all been considered for distributed machine learning. This research is carried out on Apache Spark for its support for distributed datasets and its ability to adapt to a changing problem size for elastic computing. However, computational inefficiency is a well-known problem for Spark, and analysis has shown that this is contributed to by the synchronous computing model of the Resilient Distributed Dataset (RDD) and the inherent limitation of data-flow for communication. Lastly, Spark has a lack of support for persistent mutable memory, which causes significant overhead for iterative algorithms such as SGD for machine learning.

The significance of the computational inefficiency due to the synchronous computing model has been demonstrated by experiments using the ImageNet dataset, where the GPU utilisation fell below 50% for 700k+ images per worker and the memory usage is at maximum capacity. This is overcome by the introduction of *MapRDD* in Section 5.2, which enables asynchronous lazy loading of training samples, resulting in full processor utilisation and a more than 90% reduction in memory usage.

The communication inefficiency using the 'reduce-broadcast' data-flow pattern for synchronising parameters has been proven analytically in comparison with the *butterfly all-reduce* and *chunked ring all-reduce*. This is overcome by the adoption of all-reduce algorithms for elastic systems such as Spark, and its effectiveness has been demonstrated in both small and large neural network models where a 2× and 7× speed-up was observed on LeNet [44] and AlexNet [41] respectively. Further optimisation using zero-copy with Remote Direct Memory Access (RDMA) has been proven to be more effective on bandwidth-bounded algorithms such as the *butterfly all-reduce* which was more copy-heavy; and on moderately-sized vectors: if the vector size is too small, the advantage of zero memory copy is less significant; if the vector size is too large, resource for *pinned memory* is quickly consumed and degrades system performance. It is found that *chunked ring all-reduce* is still favoured for large vectors over TCP. In short, there is no single winner for every application and the choice of algorithm has to be considered case by case.

RDD is the core memory concept for Spark and is only designed for static datasets, as such, any changes would generate redundant memory copies. RDD is also designed to represent a single image of the dataset and not replicas. These two issues have been resolved by the introduction of a new distributed key-value store with check-pointing as described in Section 5.3. This results in a hybrid all-reduce and key-value store model that is both flexible and high-functioning compared with the parameter server architecture.

The research so far improves the standard synchronous SGD in terms of computational and communication efficiency, further improvements are attainable through altering the algorithm itself. Asynchronous SGD, a method that overlaps the computation and communication, is one such method. We explore two possible formulae for asynchronous SGD using non-blocking all-reduce and we demonstrate statistical equivalence to the standard synchronous SGD using AlexNet-ImageNet.

Lastly, a method of fine-grained batch size control for large batch size SGD training is introduced in Section 6.2. The characteristic of this method is the changing problem size, which has been resolved by performance prediction and the hybrid computing model (i.e. all-reduce + key-value store) introduced in this research. Using AlexNet-ImageNet, this method consistently produces faster convergence and higher validation accuracy than a static batch size with/without adaptive learning rates.

Taken together, the work presented in this thesis detailed a systematic approach to tuning the performance of distributed and parallel machine learning. This consists of: (i) the design and implementation of an elastic machine learning system with optimised computational and communication efficiency; (ii) asynchronous SGD with non-blocking all-reduce; (iii) large batch size training with fine-grained batch size control. We demonstrate a hybrid computing model that is able to adapt to a changing problem size as well as maintaining high computational efficiency. Such a system serves as an exemplar for emerging new applications with a non-uniform and dynamic workload. For example, computational steering and data assimilation. As dynamic workloads becoming more and more common, this promotes the need for on-demand computing services such as Amazon, Google, Azure cloud services, and a high-performance elastic computing system is part of the solution to future large-scale distributed and parallel computing.

7.1 Limitations

The primary limitation of this thesis is its focus on the Convolutional Neural Networks (CNN) algorithm. CNN represents a class of machine learning algorithms that is both computation and communication intensive. As we learned, the computations are mainly comprised of matrix–matrix multiplications and these models contain millions of parameters to be communicated. It also represents a larger class of applications that can be represented as solving for the matrix equation $A.X = B$, such as the finite element analysis in scientific simulations or general matrix factorisation. As such, optimisations for system design and implementation in this thesis are transferable to other applications, whereas algorithmic improvements to the Stochastic Gradient Descent (SGD) are only applicable to optimisation problems using SGD.

The secondary limitation is the focus on the Apache Spark platform. As explained in Section 2.3.3, the Apache Spark represents a wide range of data analytics platforms based on a master-slave architecture and a data-flow task execution engine. The majority of this thesis, that is the techniques for memory management (i.e. *MapRDD*) and the algorithmic optimisation methods (i.e. asynchronous SGD), are also transferable to other distributed computing platforms (e.g. the Message Passing Interface).

Another limitation of this thesis is that computational efficiency as its main concern, but not the precision of the gradient descent or the predictive accuracy of the machine learning model. In the process of improving the computational efficiency, the synchronous computing model and the data-flow programming paradigm are broken, and a new hybrid computing model is created. We argue that such a model incorporates the advantages of both high-performance computing and elastic computing and sheds light upon new computing architectures for steering applications (see Section 7.2.2).

Lastly, this research is also limited by the computing resources and data resources available. All experiments are carried out on an Intel Xeon CPU-based computing cluster with 32 nodes; as a consequence, the computational performance on an accelerator-based cluster is only conjectural, and the limited computing power only allows for experiments on simpler models such as AlexNet. The asynchronous SGD and dynamic batch-sizing methods are only tested on the ImageNet dataset [31], and the characteristics of the training data could have an impact on the effectiveness of these two methods. However, the ImageNet dataset used in this research is currently the only dataset that is large and complex enough for representing object recognition problems.

7.2 Future Work

Going forward, there are two major avenues for the continuation of this work: (i) native acceleration for distributed machine learning; (ii) a hybrid high-performance and elastic computing model for steering applications. We discuss the trend of hardware and neural network developments and the opportunity for further acceleration with native codes in Section 7.2.1. We describe the similarities and differences of steering applications compared with distributed machine learning, and the future computing architectures and programming paradigms for steering applications in Section 7.2.2.

7.2.1 Native Elastic Computing

Revisiting the initial problem of long turnaround for neural network algorithms, it is predicted that training the ResNet50 model on the ImageNet dataset for 100 epochs at an effective batch size of 16K takes under 20 minutes on a NVIDIA Tesla P100 cluster (as listed in Table 7.2), in contrast to 26 months on a single Intel Xeon E5-2660 2.6 GHz processor (as listed in Table 1.1).

For further improvements, the trend of development in machine learning models and computer hardware must be analysed. Table 7.2 is a compilation of recent convolutional neural networks and their performance metrics; Table 7.1 lists the growth of network bandwidth for the InfiniBand interconnect technology over the years. The following trends are identified:

1. The computational complexity (i.e. multiply-add operations) increases in general.
2. The communication cost (i.e. the number of parameters) decreases in general.
3. The processing speed has grown explosively, a 222× increase in computing power is seen (from 10 samples per second on an Intel Xeon E5-2660 2.6 GHz CPU to 580 and 2227 samples per second on an NVIDIA Tesla K80 and an NVIDIA Tesla P100 GPU, for AlexNet-ImageNet).
4. The network bandwidth for InfiniBand has increased by 6.25 times from QDR to HDR over 10 years.

The issue discovered above is the disproportionate growth in computing power with respect to the growth in network bandwidth, taken into account the changes in computational complexity and communication costs. This results in a change from computational dominance to communication dominance, and therefore it is harder to gain any speed-up from distributed training (i.e. maximum speed-up of 1 in Table 7.2).

Re-visiting the breakdown in costs in all-reduce from Section 4.3.2, there are five sources for overhead: start-up, compute, object serialisation, object deserialisation, waiting/blocking. The cost for object serialisation/deserialisation remains the same for the same processor speed, and it currently takes up to 25% of the total communication cost for *butterfly-parallel all-reduce* over TCP-IBoIP. The percentage for object serialisation/deserialisation keeps increasing as the waiting time decreases due to the continuous growth in the network speed. This number is expected to reach 40-50% as InfiniBand NDR arrives after 2020, and it is even more significant for Remote Direct Memory Access (RDMA). The impact of removing this object serialisation/deserialisation cost is demonstrated by the measurement of the communication coefficients α and β on the native Message Passing Interface as listed in Table 7.2, where α is reduced by 4× and β is reduced by 10×.

Object serialisation/deserialisation is a process that transforms an in-memory data structure into a portable binary form that can be stored and read correctly by other computers in other architectures, which is necessary for one of the following scenarios:

1. Transportation and translation for Java Virtual Machine (JVM) memory objects.

2. Non-contiguous native/non-native memory structure.

3. Non-uniform data representation for the same data type due to different processor architectures. For example, different byte order/width/standard for different processors (e.g. big-endian and small-endian).

For the use of MPI floating-point arrays in a uniform computing cluster, object serialisation is disregarded as the native memory representation can be read correctly by other computers in the same cluster. For data analytics systems, a non-uniform computing cluster is assumed and the Java Virtual Machine is used for the application to run on different processor architectures. However, the architectural differences between High-Performance Computing (HPC) and Data Analytic (DA) systems are diminishing and the two systems are converging towards a single architecture. This is demonstrated by a survey of commercial systems in Table 7.3; the only difference is how the storage system is attached, either locally or network connected.

The trend of convergence for cluster computer architectures towards a single uniform architecture signals less of a need for cross-platform programming languages. Subsequently, this underpins the possibility of a native elastic computing platform.

To close the gap in the software architecture between HPC and DA platforms as mentioned in Section 2.3.2, it is clear that a master–slave and task execution model must be adopted. The biggest obstacle to this is the contrast of the explicit communication in HPC and the implicit communication in DA. A mixed model proposed in this thesis (i.e. all-reduce, data-flow and key-value store) sheds light on a possible path forward towards a unified system.

Table 7.1: Bandwidth evolution of InfiniBand from the year 2011.

Type	Year	4× Link Bandwith (Gb/s)	12× Link Bandwidth (Gb/s)
QDR	2007	32	96
FDR	2011	56	168
EDR	2014	100	300
HDR	2017	200	600
NDR	after 2020	400	1200

Table 7.2: Comparison of neural network models in terms of computational cost (multiply-add operations), communication cost (number of parameters), processing speed (γ: bigger is better), communication speed using InfiniBand QDR (α and β: smaller is better), maximum synchronous speed-up (S) at batch size (B), distributed processing speed ($\gamma \times S$, samples per second) and training time (hours).

	AlexNet [41]	VGG16 [26]	InceptionV3 [75]	ResNet50 [69]
year	2012	2014	2015	2015
multiply-adds (million)	724	15500	5000	3900
parameters (million)	61	138	25	25.5
γ (P100, 8gpus)	17822	1081	1131	1734
γ (K80, 8gpus)	4642	260	227	387
α (Java, RDMA)	0.8	1.76	0.32	0.32
β (Java, RDMA)	0.028	0.063	0.011	0.011
α (C, MPI)	0.2	0.44	0.08	0.08
β (C, MPI)	0.0028	0.0063	0.0011	0.0011
S(B=512, Java)	1.0	1.0	1.0	1.0
S(B=2048, Java)	1.0	1.0	3.01	2.16
S(B=4096, Java)	1.0	1.38	5.04	3.68
S(B=16K, Java)	1.0	4.08	12.95	9.79
S(B=512, MPI)	1.0	1.0	3.63	2.54
S(B=2048, MPI)	1.0	2.88	10.70	7.77
S(B=4096, MPI)	1.0	5.06	17.56	12.98
S(B=16K, MPI)	3.05	14.33	43.57	33.28
Time (hrs.) (ImageNet, 100 epochs, (P100×8, standalone)	2.0	32.9	31.5	20.5
Time (hrs.) (ImageNet, 100 epochs, (P100×8, B=16K, distributed, **sync**)	0.65	2.29	0.72	0.62
Time (hrs.) (ImageNet, 100 epochs, (P100×8, B=16K, distributed, **async**)	0.33	1.15	0.36	0.31

Table 7.3: List of High-Performance Computing and Data Analytic Systems in 2016

Name	CPU	Accelerator	Interconnect	Storage	Application
Cray XC	Intel Xeon E5	Intel Xeon Phi	Aries	Network Attached	HPC
Cray CS	Intel Xeon E5	Intel Xeon Phi, NVIDIA GPU	InfiniBand FDR	Network Attached	HPC
Cray Urika-GX	Intel Xeon E5	None	Aries	HDD, SSD	Data Analytics
IBM Power S822	Power8	None		HDD, SSD	Data Analytics
IBM Power S822LC	Power8	FPGA, NVIDIA GPU		HDD,SSD	High Performance Data Analytics
IBM Power S824L	Power8	NVIDIA GPU		HDD,SSD	High Performance Data Analytics

7.2.2 Computational Steering

One of the most powerful findings of the research in this thesis is the new computing architecture for adaptive problem sizes using combined High-Performance Computing and elastic computing, and its shared characteristics to scientific computing for solving a large set of linear equations. This opens up new avenues for computational mechanics and fluid dynamics for new **computational steering** applications, which is a class of applications that allows for manual or data interventions that change the computation process.

Interactive Steering

Computational mechanics and fluid dynamics use finite element or finite volume methods for approximating solutions to partial differential equations, in order to calculate forces and stresses in a solid or a fluid. These methods work by partitioning a solid object or fluid volume of arbitrary shapes into small discrete elements or volume cells, called a **mesh** and, subsequently, variation across the element or cell can be estimated using discrete linear/non-linear approximations. For areas with high variations, for example, the corners of an object, a fine mesh/small elements should be used; for areas with small variations, a coarse mesh/large elements should be used; this results in a non-uniform mesh as shown in Figure 7.1.

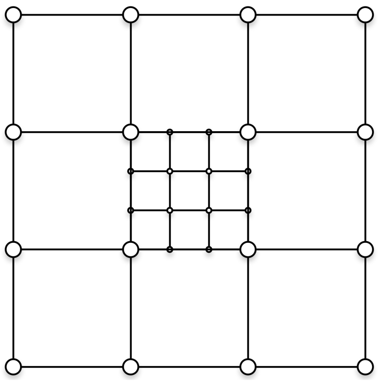


Figure 7.1: An example of non-uniform 2D mesh.

The finite element method can be formulated by solving for a large linear system $A.x = b$, where A is the symmetric and positive-definite matrix that describes the stiffness between every node in the mesh, x is a vector of unknown displacement at every node and b is a vector of known boundary conditions. This is similar to neural networks introduced in Section 2.1.2 for solving $A.X = B$, where A is the matrix of input vectors, X the weights of connections between layers of neurons and

B is the matrix of output vectors.

The conjugate gradient method displayed in Algorithm 9 is often used to solve a system of $A.x = b$ where A is symmetric positive-definite. The algorithm is comprised of matrix-vector multiplications, dot product and scalar-vector multiplications. Every iteration generates new vectors $x_{k+1}, r_{k+1}, p_{k+1}$ that describe the position and residual error at iteration $k + 1$, which need to be synchronised. There are two ways of parallelising the linear algebra calculations: along the first or second dimension of matrix A, which are analogous to *data-parallelism* and *model-parallelism* described in Section 2.1.2.

Algorithm 9 Conjugate gradient algorithm for solving $A.x = b$

```
1: $r_0 = b - Ax_0$
2: $p_0 = r_0$
3: $k = 0$
4: repeat
5:     $\alpha_k = \frac{r_k^T.r_k}{p_k^T.A.p_k}$
6:     $x_{k+1} = x_k + \alpha_k.p_k$
7:     $r_{k+1} = r_k - \alpha_k.A.p_k$
8:     $\beta_k = \frac{r_{k+1}^T.r_{k+1}}{r_k^T.r_k}$
9:     $p_{k+1} = r_{k+1} + \beta_k.p_k$
10:    $k = k + 1$
11: until $r_{k+1}$ is small enough
```

The areas with high variations are often of highest interest for researchers; the ability to zoom in/out and interact with the simulation process is called **interactive steering**. This manual interaction subsequently changes the problem size of the simulation and leads to re-sizing of A, x and b. The techniques developed in this thesis can, therefore, be used for this application: static matrix A and vector b can be represented by *MapRDD* and mutable vectors x, r, p can be represented by logical blocks of key-value pairs. The vector synchronisation can be achieved by all-reduce for parallelising the columns of matrix A, or implicit communications of the key-value store for parallelising the rows of matrix A. However, the main difference compared to the traditional Message Passing Interface implementation is its ability to adapt to the new problem size and increase/decrease the number of resources required.

Data Assimilation

Data assimilation is another type of steering application where real-world observations are combined with predictions to adjust for the outcome. This is particularly useful for physics simulations that are strongly dependent on the initial condition of

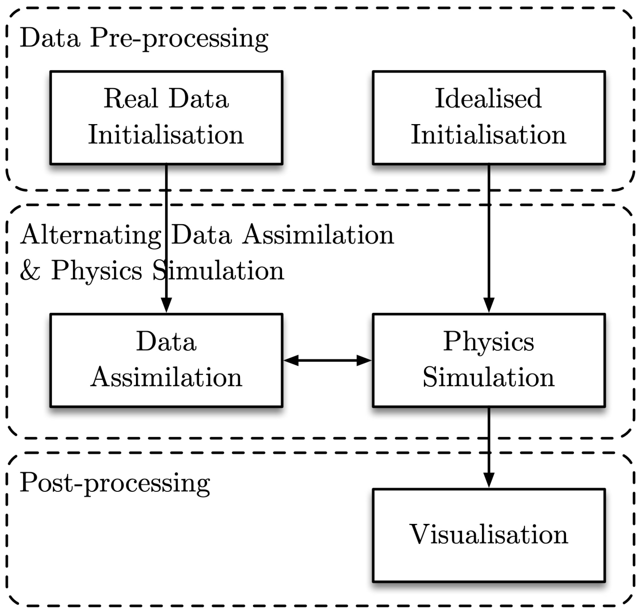


Figure 7.2: An illustration of data assimilation in the Weather Research and Forecast (WRF) system [55].

the problem, such as weather forecasting, as small differences in the initial condition due to measurement granularity and instrumental errors can cause significant divergence from the observations. The data assimilation technique often involves the optimisation of a cost function between the simulated prediction and the observation to improve future predictions. Lastly, these physics simulations generate a large volume of results that need to be analysed. This produces an atypical non-uniform workload of machine learning alongside physics simulations.

The Weather Research and Forecasting (WRF) model [55] is a next-generation mesoscale numerical weather prediction system, which is an exemplar of a data assimilation application. WRF generates atmospheric simulations using real observation or analysis data or idealised conditions. The WRF system consists of 2 major components: a physics simulator and a data assimilation system (3DVAR [5] and 4DVAR [29]). Data assimilation in 3DVAR [5] minimises the cost function shown in Equation 7.1, which describes the sum of background (J_b) and observation (J_o) errors; x^b and y^o are the previous forecast and observation respectively; B and $(E + F)$ are the background and observation error covariance matrices respectively. This is identical to the least square problem in parameter fitting for machine learning outlined in Section 2.2.1.

$$J(x) = J_b + J_o = \frac{1}{2}(x - x^b)B^{-1}(x - x^b) + \frac{1}{2}(y - y^o)(E + F)^{-1}(y - y^o) \qquad (7.1)$$

The WRF system is illustrated in Figure 7.2. The execution path starts from the initialisation, either from the real or the idealised data, which generates the input feed for the WRF physics models. The physics module generates the next forecast, which is fed back to the data assimilation system that combines it with the observed data to generate the input for the next iteration (i.e. alternating numerical simulation and data analysis). This system serves as an exemplar for the integration of scientific computing and machine learning.

Unified platform

Computational steering applications present a challenge of non-uniform workload and a hybrid system of scientific simulation and machine learning, this requires non-conventional computing architectures as outlined in [17]: (i) scalable architecture; (ii) integrated programming and software paradigm; (iii) application study. This thesis presents a partial solution to steering applications, and demonstrates that: (i) an elastic resource negotiator such as YARN and Mesos is necessary to support dynamic workload; (ii) a hybrid data-flow, all-reduce and key-value store architecture for high-performance elastic computing in the context of machine learning. However, this research does not represent the characteristics of all steering applications and further research is needed for a new unified programming paradigm and the migration of existing code and libraries onto new platforms.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL `http://dl.acm.org/citation.cfm?id=3026877.3026899`.

[2] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[3] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.

[5] Dale M Barker, Wei Huang, Yong-Run Guo, AJ Bourgeois, and QN Xiao. A three-dimensional variational data assimilation system for MM5: Implementation and initial results. *Monthly Weather Review*, 132(4):897–914, 2004.

[6] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi,

and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Data Engineering*, 38(4), 2015.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[9] Harunobu Daikoku, Hideyuki Kawashima, and Osamu Tatebe. On exploring efficient shuffle design for in-memory MapReduce. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, page 6. ACM, 2016.

[10] Aaron Davidson and Andrew Or. Optimizing shuffle performance in Spark. *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep*, 2013.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[13] Aditya Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[14] Jeff Donahue. `https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet`, 2016. [Online; Accessed 08-April-2018].

[15] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[16] Erik Erlandson. `http://erikerlandson.github.io/blog/2014/09/11/faster-random-samples-with-gap-sampling/`, n.d. [Online; Accessed 23-January-2018].

[17] Geoffrey Fox, Shantenu Jha, and Lavanya Ramakrishnan. Scalable HPC workflow infrastructure for steering scientific instruments and streaming applications. Technical report, Technical report, 2015.

[18] Spark Github. [WIP][SPARK-1485][MLLIB] Implement Butterfly AllReduce. `https://github.com/apache/spark/pull/506`, 2014. [Online; Accessed 01-August-2017].

[19] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[20] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[21] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[22] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *IEEE Parallel Distributed Technology: Systems Applications*, 1(3):12–21, Aug 1993. ISSN 1558-1861. doi: 10.1109/88.242438.

[23] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[24] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL `http://doi.acm.org/10.1145/42411.42415`.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[27] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.

[28] Sumin Hong, Woohyuk Choi, and Won-Ki Jeong. GPU in-memory processing using Spark for iterative computation. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 31–41. IEEE Press, 2017.

[29] Xiang-Yu Huang, Qingnong Xiao, Dale M Barker, Xin Zhang, John Michalakes, Wei Huang, Tom Henderson, John Bray, Yongsheng Chen, Zaizhong Ma, et al. Four-dimensional variational data assimilation for WRF: Formulation and preliminary results. *Monthly Weather Review*, 137(1):299–314, 2009.

[30] Zurich IBM Research Lab. `https://github.com/zrlio/disni`, n.d. [Online; Accessed 12-April-2019].

[31] ImageNet. `http://www.image-net.org/challenges/LSVRC/2012/`, n.d. [Online; Accessed 01-September-2018].

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference*. Association for Computing Machinery, Inc., March 2007. URL `https://www.microsoft.com/en-us/research/publication/dryad-distributed-data-parallel-programs-from-sequential-building-blocks/`.

[34] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K Panda. Performance characterization and acceleration of in-memory file systems for Hadoop and Spark applications on HPC clusters. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 243–252. IEEE, 2015.

[35] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[36] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly Media, 2005.

[37] Hanjoo Kim, Jaehong Park, Jaehee Jang, and Sungroh Yoon. DeepSpark: A Spark-based distributed deep learning framework for commodity clusters. *arXiv preprint arXiv:1602.08191*, 2016.

[38] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[39] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[40] Alex Krizhevsky. `https://code.google.com/archive/p/cuda-convnet/`, n.d. [Online; Accessed 01-August-2017].

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL `http://dl.acm.org/citation.cfm?id=2999134.2999257`.

[42] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`, n.d. [Online; Accessed 01-August-2017].

[43] Omkar Kulkarni. `http://www.opensfs.org/wp-content/uploads/2013/04/LUG2013_Hadoop-Lustre_OmkarKulkarni.pdf`, 2013. [Online; Accessed 01-July-2019].

[44] Yann Lecun. LeNet-5, Convolutional Neural Networks. `http://yann.lecun.com/exdb/lenet/`, n.d. [Online; Accessed 01-August-2017].

[45] Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. The MNIST dataset. `http://yann.lecun.com/exdb/mnist/`, n.d. [Online; Accessed 01-August-2017].

[46] Hu Li, Tianjia Chen, and Wei Xu. Improving Spark performance with zero-copy buffer management and RDMA. In *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, pages 33–38. IEEE, 2016.

[47] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

[48] Zhenyu Li and Stephen Jarvis. MapRDD: Finer grained resilient distributed dataset for machine learning. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR'18, pages 3:1–3:9, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5703-6. doi: 10.1145/3206333.3206335. URL `http://doi.acm.org/10.1145/3206333.3206335`.

[49] Zhenyu Li, James Davis, and Stephen Jarvis. An efficient task-based All-Reduce for machine learning applications. In *Proceedings of the Machine Learning on HPC Environments*, MLHPC'17, pages 2:1–2:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5137-9. doi: 10.1145/3146347.3146350. URL `http://doi.acm.org/10.1145/3146347.3146350`.

[50] Zhenyu Li, James Davis, and Stephen Jarvis. Optimizing machine learning on Apache Spark in HPC environments. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 95–105, Dallas, TX, USA, 2018. IEEE. ISBN 978-1-7281-0180-4. doi: 10.1109/MLHPC.2018.8638643. URL `http://dx.doi.org/10.1109/MLHPC.2018.8638643`.

[51] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5 (8):716–727, 2012.

[52] Xiaoyi Lu, Md Wasi Ur Rahman, Nusrat Islam, Dipti Shankar, and Dhabaleswar K Panda. Accelerating Spark with RDMA for big data processing: Early experiences. In *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, pages 9–16. IEEE, 2014.

[53] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[54] Lester Martin. `https://martin.atlassian.net/wiki/spaces/lestermartin/blog/2016/05/19/67043332/why+spark+s+mapPartitions+transformation+is+faster+than+map+calls+your+function+once+partition+not+once+element`, 2016. [Online; Accessed 08-April-2018].

[55] J Michalakes, J Dudhia, D Gill, T Henderson, J Klemp, W Skamarock, and W Wang. The weather research and forecast model: software architecture and performance. In *Use of High Performance Computing in Meteorology*, pages 156–168. World Scientific, 2005.

[56] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[57] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. ImageNet/ResNet-50 training in 224 seconds. *arXiv preprint arXiv:1811.05233*, 2018.

[58] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. Sparknet: Training deep networks in spark. *CoRR*, abs/1511.06051, 2015.

[59] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[60] Y. E. Nesterov. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. *Dokl. Akad. Nauk SSSR*, 269:543–547, 1983. URL `https://ci.nii.ac.jp/naid/10029946121/en/`.

[61] Pathmind. `https://pathmind.com/wiki/open-datasets`, n.d. [Online; Accessed 01-March-2020].

[62] BID Data Project. `http://bid.berkeley.edu/BIDdata/overview/`, n.d. [Online; Accessed 01-August-2017].

[63] Rolf Rabenseifner. Optimization of collective reduction operations. In *International Conference on Computational Science*, pages 1–9. Springer, 2004.

[64] Rolf Rabenseifner and Jesper Larsson Träff. 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*, pages 36–46. Springer, 2004.

[65] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[66] Alexey Romanov. `https://stackoverflow.com/questions/21185092/apache-spark-map-vs-mappartitions`, n.d. [Online; Accessed 08-April-2018].

[67] Rsocket. `https://github.com/rsocket/rsocket`, n.d. [Online; Accessed 12-April-2019].

[68] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. Performance modeling and evaluation of distributed deep learning frameworks on GPUs. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 949–957. IEEE, 2018.

[69] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[70] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[71] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[72] Samuel L Smith and Quoc V Le. A Bayesian perspective on Generalization and Stochastic Gradient Descent. *arXiv preprint arXiv:1710.06451*, 2017.

[73] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[74] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015. doi: 10.1109/CVPR.2015.7298594.

[75] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[76] Tensorflow. `https://www.tensorflow.org/guide/performance/benchmarks`, n.d. [Online; Accessed 12-April-2019].

[77] Rajeev Thakur and William D Gropp. Improving the performance of collective operations in MPICH. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 257–267. Springer, 2003.

[78] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[79] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[80] V Vassilevska Williams. Breaking the Coppersmith–Winograd barrier. *n.d.*, 2011.

[81] Peifeng Yin, Ping Luo, and Taiga Nakamura. Small batch or large batch: Gaussian walk with rebound can teach. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1275–1284. ACM, 2017.

[82] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[83] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 273–283. IEEE, 2016.

[84] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 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*, pages 2–2. USENIX Association, 2012.

[85] Huasha Zhao and John Canny. Butterfly Mixing: Accelerating incremental-update algorithms on clusters. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 785–793. SIAM, 2013.

[86] Huasha Zhao and John Canny. Kylix: A sparse Allreduce for commodity clusters. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 273–282. IEEE, 2014.