Communication Reduction Techniques in Numerical Methods and Deep Neural Networks

Sicong Zhuang

Advisors: Dr. Marc Casas Guix
Dr. Eduard Ayguadé Parra

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

August 2019

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Acta de calificación de tesis doctoral

Curso académico:

Nombre y apellidos

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada _____

_____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a	(Nombre, apellidos y firma) Secretario/a	
(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

_____, _____ de _____ de _____

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretario de la Comisión Permanente de la Escuela de Doctorado

Barcelona a _____ de _____ de _____

Abstract

Parallelism sees an ever-growing adoption in an ever-expanding number of fields. Modern HPC systems are designed to be massively parallel in mind where an immense amount of computational units are available. Yet it proves to be difficult to design parallel algorithms because not all the regions of the program can be parallelized. Furthermore, many problems can only be split into sub-problems with inter-dependencies. The negative impact of communication is not negligible beyond merely a couple of processes.

This thesis provides communication-reduction solutions on three problems in the field of numerical methods and deep learning. We first set out to speed up one of the iterative Krylov methods, the Conjugate Gradient Methods. This work intends to fuse iterations together and thus defer the need for synchronization at the end of the fuse phase. This approach also impedes the application of some error correction routines. This thesis explores the possibility to fuse iterations and its implication on the convergence of the entire algorithm. Empirical evidences from the experiments indicate that it achieves speedups without hampering the convergence of the algorithm.

We then move on to DNN training with multiple GPUs in which the up-to-date parameters stored and updated on the hosting CPU has to be constantly transferred to each GPU at the beginning of each batch. We propose to use a dynamic scheme that compresses the parameters to a lower precision on the CPU side before the transfer takes place. This way it cuts the amount of data transfer while the training on the GPU sides uses lower precision parameters. The compress rate is guided by a heuristics metric and the approach proceeds to increment the precision of the parameters accordingly. It provides competitive accuracy while outperforms our baseline in terms of training time.

We eventually strive to improve the training of DNNs in a distributed-memory system with model parallelism using the message passing paradigm. By replicating the neurons on each process once every two layers, we essentially cut the communication in half during both the forward- and backward-propagation at the cost of a 25% increase in floating point computation. The trade-off turns out to pay off according to our experiments and this approach is able to offer significant speedups over the baseline approach where we naïvely split the neurons at each layer.

Acknowledgements

I would like to extend my gratitude and appreciation to my advisors Dr. Marc Casas and Dr. Eduard Ayguadé, without their patience and guidance this thesis would not have been possible. I would also like to thank Dr. Cristiano Malossi and Dr. Panagiotis Hadjidoukas from IBM Zürich Lab who, during the course of our collaboration, gave me valuable feedbacks and insights that lead to the success of my work. As well as for their hospitality during my month-long stay in Zürich that warmed a lone traveler's heart.

It has been a long five-year journey. One does not simply navigate through those times alone, not to mention the place I call home is ten thousand km away. I want to thank those who were or still are by my side giving me support, talking me through things when the getting is tough. My appreciation goes to my bestie Kallia, along with Rajiv, Danai, Tomasz and David with whom we shared invaluable moments and countless chuckles during the good part of my PhD. To Ariel, a close friend, that we get along well enough to share an apartment with, that is not easy to come by. To Ilia and Klaudia who we shared a close connection despite the little time we spent. Last but not least, a group of people of all ages and from all walks of life yet have enough synergy to form a close group that I came across over the final months of my PhD: Alex, Asaf, Ettore, John "El Cucho" Osorio, Louis "LeDur" Ledoux, Robin and Tamara. A group that offers laughters and stories that makes an otherwise stressful conclusion of a PhD a lot more entertaining.

This thesis has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and by the IBM/BSC Deep Learning Center Initiative.

Table of contents

1

Introduction

Modern HPC systems make extensive use of their massive amount of CPU core count and their peripheral accelerators (GPU, FPGA, ASIC etc.) to achieve a high performance [1, 2, 3, 4]. In order to effectively utilize such systems, algorithm designers need to parallelize their problems either by hand or relying on compiler or runtime system support. The problems need to be meticulously split into smaller chunks that can be executed on the individual computational units simultaneously.

Not all parallelization problems are created equal. For some, denoted as *embarrassingly parallel problems*, the task is relatively simple because they can be easily solved in a divide and conquer fashion. Each component is inherently independent in that it does not require the computational results from its counterparts.

While on the other hand, others oftentimes have non-parallelizable sections that create interleaving parallel-sequential-parallel patterns during the execution where synchronization is required. It is also a commonplace that parallel sections possess dependencies in which case communication inevitably occur. Such problems are ubiquitous in numerical linear algebra and other fields of computational mathematics like matrix multiplication, matrix decomposition, eigenvalue solvers, mathematical optimization problems just to name a few [5, 6, 7, 8].

As peripherals, the various types of accelerators are connected through external buses like PCIe, NVlink [9] etc. Necessary data has to be transferred from the host CPUs to the accelerators before carrying out any meaningful computation. It is prominent among iterative-based numerical methods and with the rise of deep neural networks [10, 11, 12, 13].

Imbalanced synchronization and communication may force the involved computational units to waste its computing resources. The scale of the parallel system is the primary impact factor of the efficiency of the communication for the following reasons.

- The physical proximity of the communicating nodes determines the quality of the communication. In a distributed system with nodes scattered at different physical locations, the communication imbalance could create serious bottlenecks.

- The need to send data back and forth is alleviated on a shared-memory system where all the computational units have access to entire memory region. Nevertheless, such systems are inherently limited by size. Another type of underlying memory hierarchy is distributed-memory systems where each node is in possess of a portion of the entire memory. The acquisition of contents from other memory regions has to be resolved by passing messages which could raise contention on the bus system.

1.1 Thesis Objectives and Contributions

This thesis strives to alleviate the communication by reducing either the occurrences of communication points or the quantity of data in the domain of iterative numerical methods and deep neural networks, while in the meantime retaining the quality of the results the algorithms produce.

1.1.1 Communication Reduction in Conjugate Gradient Method

The conjugate gradient method solves a linear system in an iterative manner. Conventionally, synchronization is needed at the end of each iteration in a parallel implementation for some bookkeeping tasks such as checking the convergence and applying the residual replacement strategy. We propose the *Iteration-Fusing Conjugate Gradient* which fuses some of the iterations by removing the inter-iteration synchronization points within those fused iterations and moving the bookkeeping tasks to the end of the last iteration from the fusion. Also we use a task-based parallel programming model to split numerical kernels into subkernels to relax data-dependencies. By carrying out these two optimizations, our approach allows computations belonging to different iterations to overlap if there are no specific data or control dependencies between them. The main contributions of this approach are:

- The Iteration-Fusing Conjugate Gradient (IFCG) approach, which aims at aggressively overlapping different iterations. IFCG is implemented by means of two algorithms: IFCG1 and IFCG2.
- A task-based implementation of the IFCG1 and IFCG2 algorithms that automatically overlaps computations from different iterations without the need for explicit programmer specification on what computations should be overlapped.
- A comprehensive evaluation comparing IFCG1 and IFCG2 with the most relevant state-of-the-art formulations of the CG algorithm. IFCG1 and IFCG2 provide parallel performance improvements up to 42.9% and 41.5% respectively and average improvements of 11.8% and 7.1% with respect to the state-of-the-art techniques and show similar numerical stability.
- A demonstration that under realistic system noise regimes IFCG algorithms behave much better than previous approaches. IFCG algorithms achieve an average 18.0% improvement over the best state-of-the-art techniques under realistic system noise regimes.

1.1.2 Communication Reduction in Training Deep Neural Network Models

We describe and evaluate a method to accelerate the training of DNNs by reducing the cost of data transfers across heterogeneous high-end architectures integrating multiple GPUs. By relying on DNNs tolerance to data representation formats smaller than the commonly used 32-bit Floating Point (FP) standard [14, 15], we describe how to dynamically adapt the size of data sent to GPU devices without hampering the quality of the training process. Our solution is designed to efficiently use the incoming bandwidth of the GPU accelerators. It relies on an adaptive scheme that dynamically adapts the data representation format required by each DNN layer and compresses network parameters before sending them over the parallel system. This scheme enables DNNs training to progress in a similar rate as if the 32-bit FP format was used. This work makes the following contributions:

- It proposes the *Adaptive Weight Precision (AWP)* algorithm, which dynamically adapts the numerical representation of DNN weights during training. AWP relies on DNNs' tolerance for reduced data representation formats. It defines the appropriated data representation format per each network layer during training without hurting network accuracy.
- It proposes a new *Approximate Data Transfer (ADT)* procedure to compress DNN's weights according to the decisions made by the AWP algorithm. ADT relies on both thread- and SIMD-level parallelism and is compatible with architectures like IBM's POWER or x86. ADT is able to compress large sets of weights with minimal overhead, which enables the large performance benefits of our approach.
- It evaluates ADT and AWP on two high-end systems: The first is composed of two x86 Haswell multicore devices plus four Tesla GK210 GPU accelerators and the second system integrates two POWER9 chips and four NVIDIA Volta V100 GPUs. Our evaluation considers the Alexnet [16], the VGG [17] and the Resnet [18] network models applied to the ImageNet ILSVRC-2012 dataset [19]. Our experiments report average performance benefits of 6.18% and 11.91% on the x86 and the POWER systems, respectively. Our solution does not reduce the quality of the training process since networks final accuracy is the same as if they had been trained with the 32-bit Floating Point format.

1.1.3 Communication Reduction in Deep Learning Model Parallelism

This work describes a novel approach *Altsplit* aims at accelerating the training of DNNs and improving the scalability of the current *model parallelism* approach by reducing the communication occurrences during both the forward- and backward- propagation phases. It achieves so by alternating the splitting and the replication of the neurons in successive layers in a distributed-memory system. We compare this approach with a *baseline* approach, where the neurons of all the layers are split, on two HPC clusters with high-end CPUs (x86 Xeon and POWER9). Our experiments see an average performance benefits of 66.12% and 57.16% respectively on both clusters.

1.2 Thesis Structure

This thesis has the following structures. Chapter 2 equips the reader with sufficient knowledge regarding the status quo and state-of-the-art research on the fields this thesis strives to improve upon. Chapter 3 gives a brief coverage of the various software, hardware and datasets used throughout the thesis. The three contributions are presented in Chapter 4, Chapter 5 and Chapter 6 respectively. Chapter 7 offers a conclusion to the thesis.

2

Background

This section provides a description on the state-of-the-art and the current challenges presented in the field of parallel computing and deep learning in order to grasp the works carried out from this thesis. Section 2.1 prepares the reader with the acquaintances of the various forms of modern parallel systems. Section 2.2 provides background on the parallel programming models on modern parallel systems. Subsequently, section 2.3 and 2.4 present an introduction to the two application domains this work deals with, namely, parallel numerical algorithms and DNNs.

2.1 Modern Parallel Systems

Exascale supercomputers are expected to come into operation near 2020. In order to reach that, major improvements need to be achieved including the energy and power, memory and storage, concurrency and locality and resiliency [20]. Nevertheless, the mainstream trend stays with the massive parallelism with accelerators approach. This section provides a background on both types of systems and an introduction to some major parallel programming models.

With the roll-out of 7 and 5 nm node, the VLSI (very large scale integration) manufacturing technology is rapidly approaching its end due to its physical limitation. Furthermore, the power and energy consumption, as a consequence the heat dissipation, on a modern VLSI chip has become a major limiting factor in processor design. All the above impede the efforts to push a single-core processor to go faster. In response, industry turns its attention into designing multi-core multiprocessor systems which exploit parallelism at the application level. Figure 2.1 illustrates a typical multi-core multiprocessor system [21]. Each of the processors (processor 0 and 1) packs two separate cores with their own L1 and L2 caches. The two processors are connected via a system bus which also connects to the RAM. All the cores thus are able to access to the entire memory region. Nonetheless, since all the access to the memory and communication among processors are conducted on the system bus, the system is limited in its scale in that the inevitable contention on the bus system while the number of processors grows eventually renders a large-scale system unresponsive.

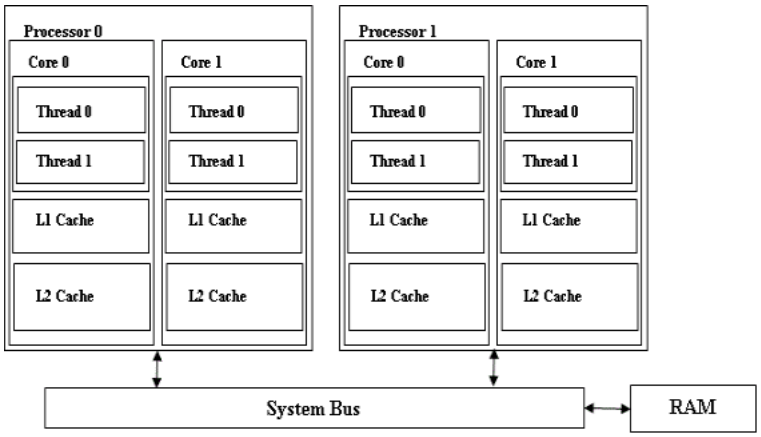

Fig. 2.1 A typical chip multithreaded, multi-core, multiprocessor system

With the ever-growing demand of computation power, a large amount of such processors are grouped close together into a computer cluster with high speed interconnection system to further scale the system. All the cores belonging to the same node in the cluster shares the resources (memory system, last-level cache etc.) whereas cross-node resources are private to their respective nodes. This essentially segregates the memory system into various regions not directly accessible to all the cores. This type of memory system is known as distributed-memory system. Accessing remote contents is possible by sending them as message to the requesting node which implies that accessing to different memory regions incurs distinct latency. This offloads the task of ensuring the performance of the program to the programmer because careless handling of the physical location and topology of the nodes can easily saturate the interconnection system and cause different processors to have imbalanced accessing time to the same data.

Since the dawn of the artificial intelligence, GPU, due to its immense data parallelism capability, has accelerated its transformation from a peripheral device used in niche domains to a general-purpose mass adoption. Along with the re-configurability of FPGAs and domain-specific ASICs, heterogeneous systems contribute to a significant portion of computation power on some modern parallel systems. As external devices, to be able to exploit their parallelism, data has to be transferred from the CPU to the device and vice versa in the face of any synchronization or communication.

2.2 Parallel Programming Models

Parallel programming models can roughly be categorized into two class: one for shared-memory systems and the other for distributed-memory systems. The classification is due to the distinct ways these models deal with the underlying memory system.

2.2.1 Shared-Memory Programming Model

OpenMP [22, 23] is a standard programming model on a shared-memory system. It is an application programming interface (API) that supports shared memory multiprocessing programming in C, C++, and FORTRAN.

OpenMP (prior to version 3.0) uses a fork-join model for its parallel executions as seen in Figure 2.2 [24]. All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered. The master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

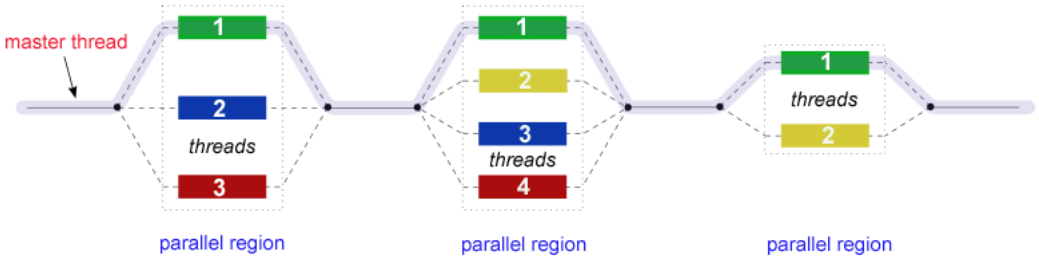

Fig. 2.2 OpenMP uses a fork-join model

2.2.2 Task-based Parallel Programming Model

A fork-join model creates parallel regions each of which is dedicated to solving a specific problem whereas the program logic is executed in sequential on the master thread. It introduces inefficiencies because the parallel regions can be far between and the sequential execution in-between keeps all the other threads idle.

Task-based parallel programming model sets off to tackle this problem. In a task-based approach the problem is ideally re-factorized and decomposed into smaller functions called tasks that have clear set of inputs and outputs from which data dependencies can be derived unambiguously. Therefore, tasks can be launched and executed in parallel as long as they don't have data dependencies among each other and hardware resources are available. A dedicated runtime system is in charge of building the dependency graph and orchestrating the task scheduling. Figure 2.3 illustrates a typical task dependency graph which is a DAG (directed acyclic graph) that the runtime system uses to check the pending dependencies. There are many task-based parallel programming models, the most prominent of which includes the OpenMP (version 3.0 onwards) [23], Clik++ [25], TBB [26] and OmpSs [27].

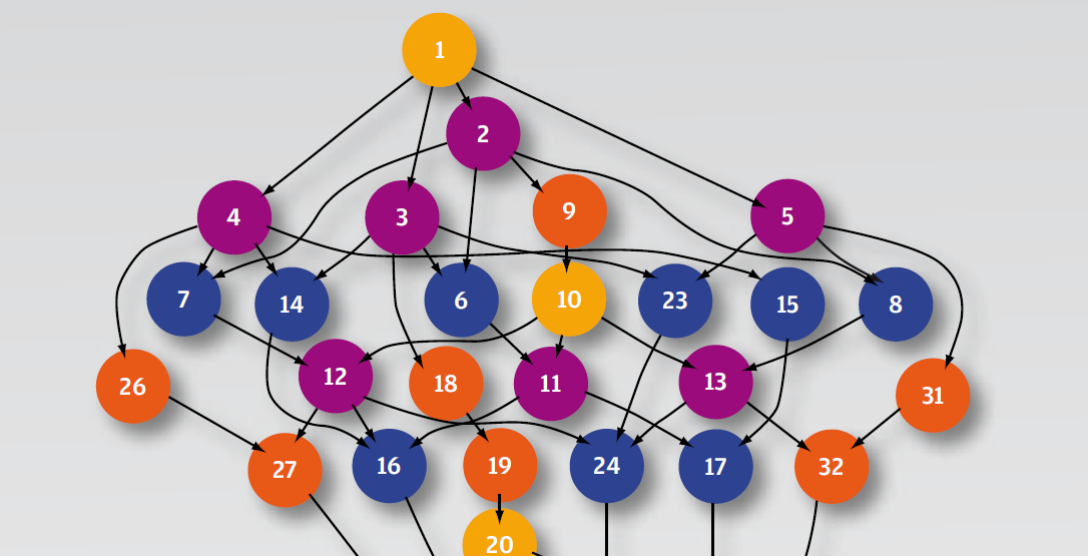

Fig. 2.3 A typical task dependency graph

2.2.3 Distributed-Memory Programming Model

MPI (Message Passing Interface) is the main parallel programming paradigm for distributed memory environments [28]. It is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. It primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. MPI programs always work with processes, but programmers commonly refer to the processes as processors.

MPI library functions include, but are not limited to, point-to-point rendezvous-type send/receive operations, choosing between a Cartesian or graph-like logical process topology, exchanging data between process pairs (send/receive operations), combining partial results of computations (gather and reduce operations), synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session, current processor identity that a process is mapped to, neighboring processes accessible in a logical topology, and so on [29]. Point-to-point operations come in synchronous, asynchronous, buffered, and ready forms, to allow both relatively stronger and weaker semantics for the synchronization aspects of a rendezvous-send.

2.3 Numerical Methods For Systems of Linear Equations

Systems of equations are used to represent physical problems that involve the interaction of various properties. The variables in the system represent the properties being studied, and the equations describe the interaction between the variables. The system is easiest to study when the equations are all linear. Often the number of equations is the same as the number of variables, for only in this case is it likely that a unique solution will exist. Although not all physical problems can be reasonably represented using a linear system with the same number of equations as unknowns, the solutions to many problems either have this form or can be approximated by such a system. In fact, this is quite often the only approach that can give quantitative information about a physical problem. The problem is to find the vectors of unknown x in $Ax = b$

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

where $x \in \Re^n$, $b \in \Re^m$ and $A \in \Re^m$ x $\Re^n$.

Two classes of numerical methods are available for solving the system:

- Direct Methods that provide the exact solution x by a finite sequence of operations.
- Iterative Methods that start with a first approximation $x^{(0)}$ and compute in an iterative manner a sequence of approximations $x^{(i)}$, in the hope to obtain increasingly better results, without ever reaching x.

2.3.1 Direct Methods

A common way to obtain the exact numerical results of systems of linear equations is using matrix factorization.

LU factorization of a matrix is the factorization of a given square matrix into two triangular matrices, one upper triangular matrix and one lower triangular matrix, such that the product of these two matrices gives the original matrix. The LU factorization method comes handy whenever it is possible to model the problem to be solved into matrix form. Conversion to the matrix form and solving with triangular matrices makes it easy to do calculations in the process of finding the solution [30, 31].

For

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

we have

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}$$

and

$$U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}$$

such that $A = LU$. The system of equations $Ax = b$ can thus be solved by the following steps:

1. Factorize the matrix A so that $LUx = b$
2. Solve the equation $Ly = b$ for y by forward substitution
3. Solve the equation $Ux = y$ for x by backward substitution

Cholesky factorization is a faster method if the matrix A is symmetric positive-definite. In which case A can be factorized into $A = LL^T$ where where L is a lower triangular matrix with real and positive diagonal entries, and L^T denotes the transpose of L [31].

2.3.2 Iterative Methods

In the absence of rounding errors, direct methods would yield an exact solution yet iterative methods are indispensable when facing linear problems involving a large number of variables (in the order of millions or more), where direct methods would be prohibitively expensive even with the best available computing power [31].

Krylov methods are among the most successful iterative methods and see a wide application in numerical linear algebra. Krylov subspace methods solves a linear system $Ax = b$ by forming a basis of the sequence of successive matrix powers times the initial residual, $\{b, Ab, A^2b, \ldots, A^mb\}$. The approximations to the solution are then formed by minimizing the residual over the subspace formed. The prototypical method in this class is the conjugate gradient method (CG) [32] which assumes that the system matrix A is symmetric positive-definite. For symmetric (and possibly indefinite) A one works with the minimal residual method (MINRES) [33]. In the case of not even symmetric matrices methods, such as the generalized minimal residual method (GMRES) [34] and the biconjugate gradient method (BiCG) [35], have been derived.

Parallel CG

Driven by the ongoing transition of hardware towards the exascale regime, research on the scalability of Krylov subspace methods on massively parallel architectures has recently garnered attention in the scientific computing community [36]. The system operator A is oftentimes sparse for many applications and in turn rather inexpensive to apply in terms of computational and communication

cost, the main bottleneck for efficient parallel execution is typically not the sparse matrix-vector product (*spmv*), but rather the communication overhead due to global reductions in dot product computations and the related global synchronization bottleneck [37].

Much effort has been put on an efficient parallel version of the CG algorithm. One of the most prominent is the introduction of pipelined CG [38]. It aims at hiding global synchronization latency by overlapping the communication phase in the Krylov subspace algorithm by the application of the *spmv*. Hence, idle core time is reduced by simultaneous execution of the time-consuming synchronization phase and independent compute-bound calculations. Other efforts includes the enlarged CG [39] and [40, 41].

2.4 Deep Supervised Learning and Its Parallelization

Deep learning uses multi-layer neural networks to carry out wide range of tasks such as image recognition [16], video classification [42], various NLP (natural language processing) tasks [43, 44, 45, 46] and art generation [47, 48] just to name a few. A neural network consists of layers of neurons which are themselves a mathematical model of a linear classifier. Figure 2.4 depicts the inner workings of a neuron. The neuron takes a vector of inputs x, first calculate its weighted sum $y = \sum_{i=1}^{n} x_i w_i$ and apply a non-linear step function $o = \delta(y)$. An MLP (multi-layer perceptron) is thus comprised of multiple layers each of which is constituted of multitudes of neurons with their respective set of weights per input.

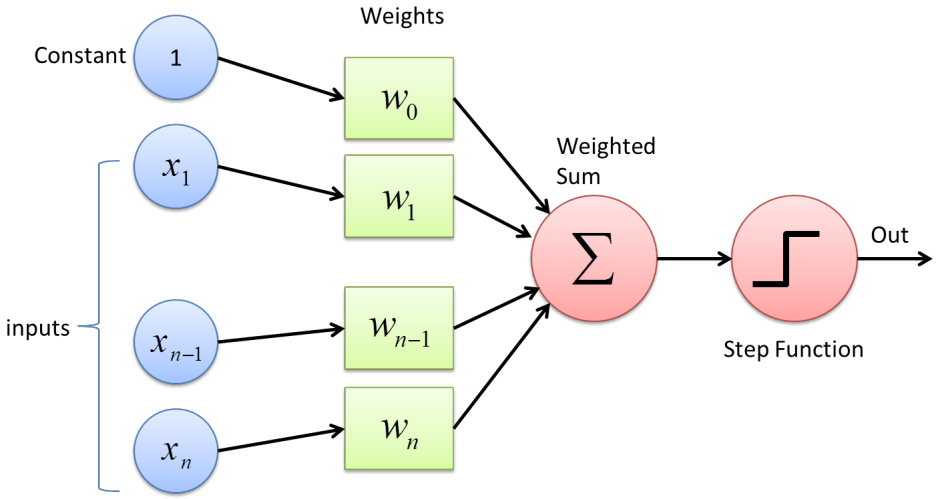

Fig. 2.4 The workings of a neuron

This thesis has its focus on one particular branch of the deep learning, namely, deep supervised learning. It utilizes a MLP (multi-layer perceptron) or CNN (convolutional neural network) to learn a function that maps an input to an output based on example input-output pairs. It infers a function from labeled training data consisting of a set of training examples. The dataset in supervised learning

are a pair of input and a ground truth (the desired output). The neural network learns by adjusting all the weights from each neuron according to its gradient $w' = w - \mu\Delta w$ to the lost yielded by a loss function that measures the difference of the output of the neural network and the ground truth. A typical loss function is MSE (mean square error), cross entropy etc.

2.4.1 Parallelism in Deep Learning

Effectively training a neural network demands an immense amount of data. This inevitably raise the need for parallelization. Currently, there are two paradigms:

- Data parallelism aims to run the training on data batches simultaneously.
- Model parallelism aims to split the neural network itself to available computation units.

Data Parallelism

The most common way in the data parallelism paradigm is the use of a parameter server [49]. The dataset is split into data shards that are consequently fed to each and every computation units respectively. The parameters (weights w, biases b etc.) are stored separately in dedicated units called parameter servers. At the beginning of each batch of the training, all the computation units involved in the training request a up-to-date copy of the parameters from the servers. They carry on with the training and at the end of the batch send their respective gradient Δw, Δb back to the servers. The servers then are responsible for updating the parameters with regard to the gradients. Figure 2.5 illustrates a schematic of two parameter servers and three trainers.

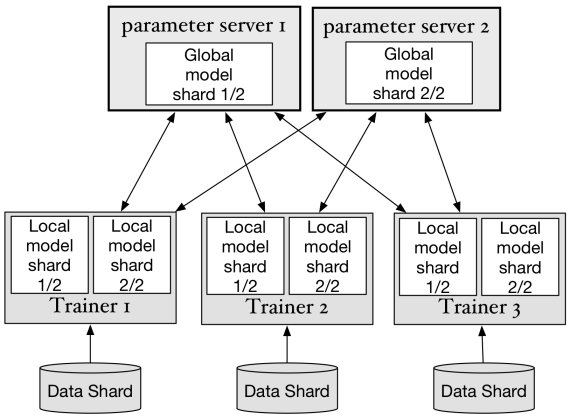

Fig. 2.5 Two parameter server and three trainers

Model Parallelism

Model parallelism is also called network parallelism. It can be seen as a orthogonal to data parallelism. This strategy divides and distributes part of the network to different machines. Figure 2.6 shows the

difference between data and model parallelism. Model parallelism is suitable for models that are too large to fit into one machine but this comes at a cost of incurring additional communication during one batch of training [50].

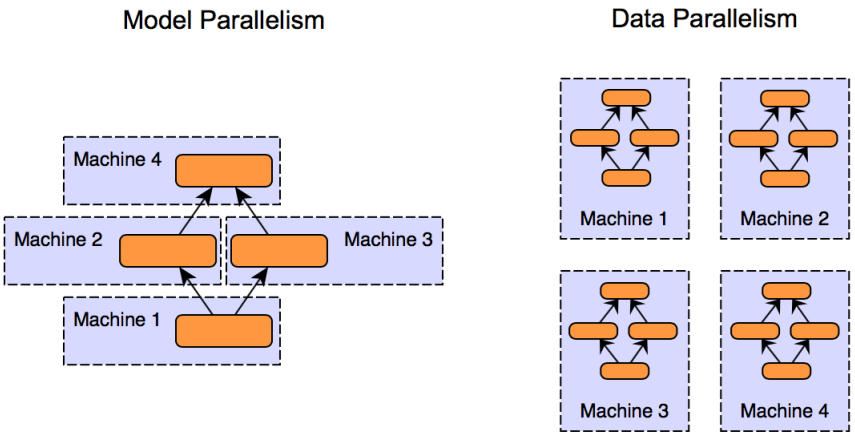

Fig. 2.6 The difference between data and model parallelism

Nevertheless, the DNN architecture creates layer interdependencies, which, in turn, generate communication that determines the overall performance. Fully connected layers, for instance, incur all-to-all communication (as opposed to allreduce in data parallelism), as neurons connect to all the neurons of the next layer [51].

3

Experimental Setup

This chapter introduces the experimental setup for the works. First a description of the three HPC clusters and their characteristics on which we run our experiments is given. Then the chapter provides information regarding the task-based parallel programming model and the two deep learning frameworks used in the work. Finally we give a description of the datasets used.

3.1 Hardware Platforms

Our experiments are run on four HPC clusters with distinct characteristics. They are the MareNostrum III and its upgrade MareNostrum IV supercomputer, MinoTauro supercomputer and CTE-POWER supercomputer at Barcelona Supercomputing Center (BSC). The results from the Iteration-Fusing Conjugate Gradient algorithm is evaluated on the MareNostrum III supercomputer. The evaluations of the deep neural networks are from the MinoTauro and CTE-POWER supercomputers. Their configurations are as follows:

- *MareNostrum III*: It consists of 3,065 compute nodes in total. Each node is IBM System X server iDataPlex dx360 M4, composed of two 8-core Intel Sandy Bridge processors E5 - 2.60Hz, 20 MB of shared last-level cache. There are eight 4GB DDR3 DIMM's running at 1.6 GHz (a total of 32GB per node and 2BG per core).

- *MareNostrum IV*: An upgrade to the *MareNostrum III* supercomputer. It consists of 3,456 compute nodes with a grand total of 165,888 processor cores and 390 TB of main memory. Each node is Lenovo system composed of SD530 Compute Racks, 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz for a total of 48 cores, 33 MB of shared last-level cache, 96 GB of main memory 1.880 GB/core, 12x 8GB 2667Mhz DIMM, 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter.

- *MinoTauro*: It is a heterogeneous cluster with 38 bullx R421-E4 servers. Each server with 2 Intel Xeon E5 U2630 v3 (Haswell) 8-core processors, (each core at 2.4 GHz,and with 20 MB

L3 cache), 2 K80 NVIDIA GPU Cards 128 GB of Main memory, distributed in 8 DIMMs of 16 GB DDR4 @ 2133 MHz - ECC SDRAM 1 PCIe 3.0 x8 8GT/s, Mellanox ConnectX ®3FDR 56 Gbit.

- *CTE-POWER*: Another heterogeneous cluster with 52 compute nodes. Each of which is equipped with 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and 4 threads/core, total 160 threads per node). 512 GB of main memory distributed in 16 dimms x 32 GB @ 2666MHz. 4 NVIDIA V100 (Volta) GPUs with 16 GB HBM2. Single Port Mellanox EDR.

3.2 OmpSs Programming Model

OmpSs is a programming model composed of a set of directives and library routines that can be used in conjunction with a high level programming language in order to develop concurrent applications. This programming model is an effort to integrate features from the StarSs programming model family, developed by the Programming Models group of the Computer Sciences department at Barcelona Supercomputing Center (BSC), into a single programming model [27].

OmpSs is based on tasks and dependences. Tasks are the elementary unit of work which represents a specific instance of an executable code. Dependences let the user annotate the data flow of the program, this way at runtime this information can be used to determine if the parallel execution of two tasks may cause data races. In OmpSs the task construct also allows the annotation of function declarations or definitions in addition to structured-blocks [52]. When a function is annotated with the task construct each invocation of that function becomes a task creation point.

The task construct allows expressing data-dependences among tasks using the *in*, *out* and *inout* clauses (standing for input, output and input/Output respectively). They allow to specify for each task in the program what data a task is waiting for and signaling is readiness.

Each time a new task is created, its *in* and *out* dependences are matched against those of existing tasks. If a dependency, either RaW, WaW or WaR, is found the task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime. Tasks are scheduled for execution as soon as all their predecessor in the graph have finished (which does not mean they are executed immediately) or at creation if they have no predecessors.

3.3 Deep Learning Frameworks

From a functionality-wise perspective, to construct a neural network boils down to stringing together a sequence of simple arithmetic functions. The derivatives of such functions have to be computed when applying the backpropagation process. Since the surge of deep learning, neural networks are getting deeper which means stacking up more layers. Each layer would have basically the same set of

arithmetic functions. It soon becomes a redundant and error-prone procedure for a programmer to build a neural network from ground up every time.

Software libraries that implement common arithmetic functions, perform auto-differentiation set off to alleviate the hassle. Such libraries are called deep learning frameworks and has become a common practice to build neural networks. Common frameworks are Keras [53], Theano [54], TensorFlow [55], PyTorch [56] etc. These frameworks are usually build neural networks by constructing a computational graph that defines the type and sequence of operations prior to the actual computation. In order to facilitate the development and debugging of building neural networks, modern deep learning frameworks also encapsulate and expose APIs of the entire neural networks etc.

3.3.1 Tensorflow

TensorFlow is an open source software library for numerical computation using data flow graphs [55]. The graph nodes represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them. This flexible architecture enables you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code. TensorFlow also includes TensorBoard, a data visualization toolkit.

TensorFlow also takes a graph-based approach to construct neural networks in which each node in the graph represents an op. An op is a function that runs on the desired devices. Its functionality can thus be extended by adding customized ops using its C++ interface and API.

3.3.2 KANN

KANN is a standalone and lightweight library in C for constructing and training small to medium artificial neural networks such as multi-layer perceptrons, convolutional neural networks and recurrent neural networks (including LSTM and GRU) [57]. It implements graph-based reverse-mode automatic differentiation and allows to build topologically complex neural networks with recurrence, shared weights and multiple inputs/outputs/costs. In comparison to mainstream deep learning frameworks such as TensorFlow, KANN is not as scalable, but it is close in flexibility, has a much smaller code base and only depends on the standard C library. In comparison to other lightweight frameworks such as tiny-dnn, KANN is still smaller, times faster and much more versatile, supporting RNN, VAE and non-standard neural networks that may fail these lightweight frameworks.

3.4 Datasets

3.4.1 SuiteSparse Matrix Collection

The SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection), is a large and actively growing set of sparse matrices that arise in real applications [58]. The Collection is widely used by the numerical linear algebra community for the development and

performance evaluation of sparse matrix algorithms. It allows for robust and repeatable experiments: robust because performance results with artificially-generated matrices can be misleading, and repeatable because matrices are curated and made publicly available in many formats. Its matrices cover a wide spectrum of domains, include those arising from problems with underlying 2D or 3D geometry (as structural engineering, computational fluid dynamics, model reduction, electromagnetics, semiconductor devices, thermodynamics, materials, acoustics, computer graphics/vision, robotics/kinematics, and other discretizations) and those that typically do not have such geometry (optimization, circuit simulation, economic and financial modeling, theoretical and quantum chemistry, chemical process simulation, mathematics and statistics, power networks, and other networks and graphs).

3.4.2 CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6,000 images per class [59]. There are 50,000 training images and 10,000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. Figure 3.1 illustrates the classes in the dataset, as well as 10 random images from each.

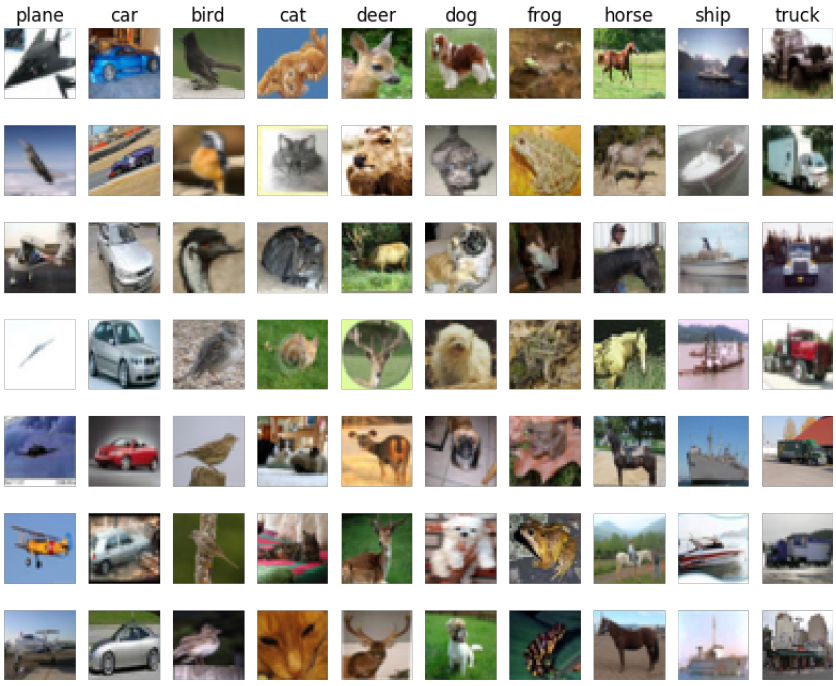

Fig. 3.1 Classes in CIFAR-10

3.4.3 ImageNet ILSVRC 2012 Challenge

ImageNet is an image dataset organized according to the WordNet hierarchy. Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a "synonym set" or "synset" [19]. There are more than 100,000 synsets in WordNet, majority of them are nouns (80,000+). ImageNet aims to provide on average 1000 images to illustrate each synset. Images of each concept are quality-controlled and human-annotated. In its completion, ImageNet will offer tens of millions of cleanly sorted images for most of the concepts in the WordNet hierarchy.

The ImageNet Large Scale Visual Recognition Challenge or ILSVRC for short is an annual competition helped between 2010 and 2017 in which challenge tasks use subsets of the ImageNet dataset. The goal of the challenge was to both promote the development of better computer vision techniques and to benchmark the state of the art. The annual challenge focuses on multiple tasks for image classification that includes both assigning a class label to an image based on the main object in the photograph and object detection that involves localizing objects within the photograph.

We use the dataset released from the 2012 challenge. The validation and test data for this competition consists of 150,000 photographs, collected from flickr and other search engines, hand labeled with the presence or absence of 1000 object categories. The 1000 object categories contain both internal nodes and leaf nodes of ImageNet, but do not overlap with each other. A random subset of 50,000 of the images with labels is released as validation data. The remaining images are used for evaluation and are released without labels at test time. The training data, the subset of ImageNet containing the 1000 categories and 1.2 million images.

4

Communication Reduction in Conjugate Gradient Method

4.1 Introduction

Many relevant High Performance Computing (HPC) applications have to deal with linear systems derived from using discretization schemes like the finite differences or finite elements methods to solve Partial Differential Equations (PDE). Typically, such discretization schemes produce large matrices with a significant degree of sparsity. Direct methods like the LU or the QR matrix factorizations are not applicable to such large matrices due to the significant number of steps they require to fully decompose them. Iterative methods are a much better option in terms of computational cost and, in particular, Krylov subspace methods are among the most successful ones. The basic idea behind Krylov methods when solving a linear system $Ax = b$ is to build a solution within the Krylov subspace composed of several powers of matrix A multiplied by vector b, that is, $\{b, Ab, A^2b, ..., A^mb\}$.

The fundamental linear operations involved in Krylov methods are the sparse matrix-vector (SpMV) product, the vector-vector addition and the dot-product. The performance of the sparse matrix-vector product is strongly impacted by irregular memory access patterns driven by the irregular positions of the sparse matrix's non-zero coefficients. As such, SpMV is typically an expensive memory-bound operation that benefits from large memory bandwidth capacity and also from high-speed interconnection networks. The vector-vector additions involved in Krylov methods typically have strided and regular memory access patterns and benefit a lot from resources like memory bandwidth and mechanisms like hardware pre-fetching to enhance their performance. Finally, the dot-product kernels involve expensive parallel operations like global communications and reductions that constitute an important performance bottleneck when running Krylov subspace methods [20].

Taking into account the performance aspects of the most fundamental linear algebra kernels of Krylov subspace methods, there are some natural improvements that are described in detail in the literature. For example, reducing the number of global reductions required by Krylov methods is a well-known option [60, 61]. Indeed, several variations of the Conjugate Gradient (CG) algorithm

have been suggested to reduce the number of global dot-products to just one [62, 63, 64, 65]. There is also work focused on reducing the number of global synchronizations targeting other subspace Krylov methods (e. g. BiCG and BiCGStab) [66, 67, 68, 69]. S-step Krylov methods also aim at reducing the number of global reductions [70, 71, 72, 73]. Besides reducing the number of global synchronization points, another alternative to boost Krylov subspace methods performance is to overlap the two most expensive kernels (SpMV and dot-product) either with other computations or between them. Indeed, overlapping the two dot-products of the CG algorithm with the residual update has already been proposed [74], as well as an asynchronous version of the CG algorithm to overlap one of the global reductions with the SpMV and the other with the preconditioner [1]. A variant of the CG algorithm that performs the two global reductions per iteration at once and also hides its latency by overlapping them with the SpMV kernel has also been proposed [38]. Despite this extensive body of work devoted on improving the CG algorithm, performance enhancements brought by state-of-the-art approaches are still far from providing good scalability results [38].

In this chapter we propose the *Iteration-Fusing Conjugate Gradient (IFCG)* method, a new formulation of the CG algorithm that outperforms the existing proposals by applying a scheme that aggressively overlaps iterations, which is something not considered by previous work. Our approach does not update the residual at the end of each iteration and splits numerical kernels into subkernels to relax data-dependencies. By carrying out these two optimizations, our approach allows computations belonging to different iterations to overlap if there are no specific data or control dependencies between them. This chapter provides two algorithms that implement the IFCG concept: IFCG1, which aims at hiding the costs of global synchronizations and IFCG2, which starts computations as soon as possible to avoid idle time.

From the programming perspective, there are several ways to enable the overlap of different pieces of computation during a parallel run. For example, such overlaps can be expressed at the parallel application source code level by using sophisticated programming techniques like pools of threads or asynchronous calls [75]. Other alternatives conceive the parallel execution as a directed graph where nodes represent pieces of code, which are named tasks, and edges represent control or data dependencies between them. Such approaches require the programmer to annotate the source code in order to express such dependencies and let a runtime system orchestrate the parallel execution. In this way, the maximum available parallelism is dynamically extracted without the need for specifying suboptimal overlaps at the source code level. Approaches based on tasks are becoming important in the parallel programming area. Indeed, commonly used shared memory programming models like OpenMP include advanced tasking constructs [23] and it is also possible to run task-based workloads on distributed memory environments [52].

This chapter adopts this task-based paradigm and applies it to the IFCG1 and IFCG2 parallel algorithms. Specifically, this chapter improves the state-of-the-art by doing the following contributions:

- The Iteration-Fusing Conjugate Gradient (IFCG) approach, which aims at aggressively overlapping different iterations. IFCG is implemented by means of two algorithms: IFCG1 and IFCG2.
- A task-based implementation of the IFCG1 and IFCG2 algorithms that automatically overlaps computations from different iterations without the need for explicit programmer specification on what computations should be overlapped.
- A comprehensive evaluation comparing IFCG1 and IFCG2 with the most relevant state-of-the-art formulations of the CG algorithm: Chronopoulos' CG [62], Gropp's CG [1], Pipelined CG [38] and a basic Preconditioned CG method. All 6 CG variants are implemented via a task-based programming model to provide a fair evaluation. IFCG1 and IFCG2 provide parallel performance improvements up to 42.9% and 41.5% respectively and average improvements of 11.8% and 7.1% with respect to the state-of-the-art techniques and show similar numerical stability.
- A demonstration that under realistic system noise regimes IFCG algorithms behave much better than previous approaches. IFCG algorithms achieve an average 18.0% improvement over the best state-of-the-art techniques under realistic system noise regimes.

This chapter is structured as follows: In section 4.2 we describe in detail some state-of-the-art approaches that motivate the IFCG algorithms. Section 4.3 contains a detailed description of the IFCG1 and IFCG2 algorithms. Section 4.4 compares the numerical stability of IFCG1 and IFCG2 with other relevant state-of-the-art techniques and explains how task-based parallelism is applied to IFCG1 and IFCG2 and how they are executed in parallel. Section 4.6 shows a comparison of IFCG1 and IFCG2 against other state-of-the-art techniques when run on a 16-core node composed of two 8-core sockets. It also discusses other important aspects like the inter-iteration overlap achieved by IFCG1 and IFCG2 and a comparison of the system jitter tolerance of these algorithms against state-of-the-art approaches. Finally, section 4.7 contains several conclusions on this work and describes future directions.

4.2 The Preconditioned and The Pipelined CG Algorithms

We describe the basic Preconditioned Conjugate Gradient Algorithm and one of its most important evolutions, the Pipelined Conjugate Gradient [38], which aims at improve CG's performance by reducing the cost of its global reductions.

4.2.1 Preconditioned Conjugate Gradient

The fundamental Preconditioned Conjugate Gradient (PCG) algorithm is a Krylov subspace method that iteratively builds a solution in terms of a basis of conjugate vectors built by projecting the

maximum descent direction, i.e. the gradient, into the closest conjugate direction. PCG is shown in **Algorithm 1**. Performance-wise, steps 4 and 8 are important bottlenecks since they involve a global reduction. Pre-conditioning the vector r_{i+1} (carried out by step 7) is also typically expensive.

Algorithm 1 PCG

1: $r_0 := b - Ax_0; u_0 := M^{-1}r_0; p_0 := u_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: $\quad s := Ap_i$
4: $\quad \alpha := (r_i, u_i)/(s, p_i)$
5: $\quad x_{i+1} := x_i + \alpha p_i$
6: $\quad r_{i+1} := r_i - \alpha s$
7: $\quad u_{i+1} := M^{-1}r_{i+1}$
8: $\quad \beta := (r_{i+1}, u_{i+1})/(r_i, u_i)$
9: $\quad p_{i+1} := u_{i+1} + \beta p_i$
10: **end for**
11: Inter-iteration synchronization

4.2.2 Pipelined Conjugate Gradient

The Pipelined Conjugate Gradient (Pipelined CG) [38] is an alternative formulation of the PCG algorithm aiming at i) reducing the cost of the two PCG's reduction operations per iteration by concentrating them into a single double reduction point and ii) hiding the cost of this double reduction by overlapping it with other PCG kernels: SpMV and the preconditioner. The Pipelined CG formulation is mathematically equivalent to PCG and, indeed, both techniques would give the exact same solution if they operated with infinite precision. However, when operating under realistic scenarios, i.e. 32- or 64-bits floating point representations, Pipelined CG exhibits worse numerical accuracy than PCG since the way Pipelined CG builds the basis of conjugate vectors is more sensitive to round-off errors, which ends up having an impact on the basis' orthogonality.

The Pipelined CG technique is detailedly shown in **Algorithm 2**. The two reductions are computed at the beginning of each iteration (lines 3-4), which makes it possible to combine them. Additionally, this double reduction operation is overlapped with two costly computations: the application of the preconditioner to vector w_i (line 5) and a sparse matrix-vector product (line 6). It is important to state that, although the potential of Pipelined CG in terms of overlapping computations and hiding reduction costs is remarkable, the algorithm still has limitations. For example, the update of the z vector in line 12 needs the whole n_i vector and the β_i scalar to be carried out. However, such restriction can be relaxed by breaking down the z update into several pieces that only have to wait for their n counterpart and the β_i scalar to be carried out. In this way, some pieces of the z vector can be updated without the need for waiting until the whole n_i vector is produced.

4.3 Iteration-Fusing Conjugate Gradient

In this section we present the Iteration-Fusing Conjugate Gradient (IFCG) approach, which breaks down some of the Pipelined CG computations into smaller pieces to relax data dependencies and

Algorithm 2 Pipelined CG

1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: $\gamma_i := (r_i, u_i)$
4: $\delta_i := (w_i, u_i)$
5: $m_i := M^{-1} w_i$
6: $n_i := Am_i$
7: **if** $i > 0$ **then**
8: $\beta_i := \gamma_i / \gamma_{i-1}; \alpha_i := \gamma_i / (\delta_i - \beta_i \gamma_i / \alpha_{i-1})$
9: **else**
10: $\beta_i := 0; \alpha_i := \gamma_i / \delta_i$
11: **end if**
12: $z_i := n_i + \beta_i z_{i-1}$
13: $q_i := m_i + \beta_i q_{i-1}$
14: $s_i := w_i + \beta_i s_{i-1}$
15: $p_i := u_i + \beta_i p_{i-1}$
16: $x_{i+1} := x_i + \alpha_i p_i$
17: $r_{i+1} := r_i - \alpha_i s_i$
18: $u_{i+1} := u_i - \alpha_i q_i$
19: $w_{i+1} := w_i - \alpha_i z_i$
20: **end for**
21: Inter-iteration synchronization

reduce idle time. Also, IFCG enables the overlap of different iterations by removing inter-iteration barrier points. We present two algorithms that implement the IFCG approach: The first one (IFCG1) improves the Pipelined CG formulation by letting different iterations to overlap as much as possible while the second one (IFCG2) aims at increasing performance even more by splitting Pipelined CG's single synchronization point into two and exploiting additional opportunities to reduce idle time. While both IFCG1 and IFCG2 algorithms apply the IFCG approach, they aim at increasing performance by targeting different goals.

4.3.1 IFCG1 Algorithm

IFCG1 is an evolution of the Pipelined CG algorithm described in section 4.2.2. IFCG1 aims at increasing the potential for overlapping different pieces of computation by breaking down the Pipelined CG kernels into smaller pieces or subkernels. Each subkernel just needs a subset of the data required by the whole kernel. For example, as mentioned a few paragraphs above, the update of the z vector in line 12 of **Algorithm 2** requires the whole n_i vector and the β_i scalar. Instead of considering the update of z as a single operation, IFCG1 breaks it down into N pieces in a way that instead of computing the whole $z_i := n_i + \beta_i z_{i-1}$ it computes N updates of the form $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$ where $z_i = \{z_{i1}, z_{i2}, ..., z_{iN}\}$ and i refers to the ith iteration. In this way, the computation of z_{ij} only depends on a subset n_{ij} of the n_i vector and the scalar β_i. The only operation that can not always be broken down into pieces is the computation of the preconditioning vector w_i (step 5 of **Algorithm 2**). While some preconditioning schemes can be decomposed into pieces (e. g. Block-Jacobi preconditioning with incomplete Cholesky factorization within the blocks) some others do not admit a straightforward decomposition (e. g. multi-grid preconditioning), although preconditioners that can be decomposed are often applied [38].

Algorithm 3 IFCG1

1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: **for** $j = 1 \ldots N$ **do** ▷ The computation is split in N blocks
4: $\gamma_{ij} := (r_{ij}, u_{ij})$
5: $\delta_{ij} := (w_{ij}, u_{ij})$
6: $m_{ij} := M^{-1} w_{ij}$
7: $n_{ij} := A_j m_i$
8: **end for**
9: $\gamma_i := \sum_{j=1}^{N} \gamma_{ij}; \delta_i := \sum_{j=1}^{N} \delta_{ij}$ ▷ Global reduction
10: **if** $i > 0$ **then**
11: $\beta_i := \gamma_i / \gamma_{i-1}; \alpha_i := \gamma_i / (\delta - \beta_i \gamma_i / \alpha_{i-1})$
12: **else**
13: $\beta_i := 0; \alpha_i := \gamma_i / \delta_i$
14: **end if**
15: **for** $j = 1 \ldots N$ **do**
16: $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$
17: $q_{ij} := m_{ij} + \beta_i q_{(i-1)j}$
18: $s_{ij} := w_{ij} + \beta_i s_{(i-1)j}$
19: $p_{ij} := u_{ij} + \beta_i p_{(i-1)j}$
20: $x_{(i+1)j} := x_{ij} + \alpha_i p_{ij}$
21: $r_{(i+1)j} := r_{ij} - \alpha_i s_{ij}$
22: $u_{(i+1)j} := u_{ij} - \alpha_i q_{ij}$
23: $w_{(i+1)j} := w_{ij} - \alpha_i z_{ij}$
24: **end for**
25: **end for**

Besides breaking down linear algebra kernels into pieces, the second innovative aspect of the IFCG approach is the elimination of inter-iteration synchronizations to check algorithm's convergence. Instead of checking for convergence at the end of each iteration, IFCG only checks it once every n iterations. The number of iterations between two checks is called the *FUSE* parameter.

We apply these two approaches (decomposition of linear kernels and elimination of inter-iterations checks) across the whole Pipelined CG algorithm, which ends up producing the IFCG1 algorithm (**Algorithm 3**). IFCG1 can potentially overlap steps 4-7 of iteration i with steps 16-23 of iteration $i-1$. Also, each repetition of steps 16-23 depends on just one of the N repetitions of steps 6-7, significantly relaxing data-dependencies between the algorithm's main kernels.

4.3.2 IFCG2 Algorithm

The IFCG2 algorithm splits Pipelined CG and IFCG1's single synchronization point, which is composed of two reductions, into two synchronization points composed of a single reduction operation each. IFCG2 aims at updating the s_i and p_i vectors, which only depend on one of the two reductions and on some data generated by iteration $i-1$, as soon as possible. The IFCG2 algorithm is detailedly shown in **Algorithm 4**. The global reductions producing δ_i and γ_i are run in separate steps. Also, the updates on vectors s_i and p_i do not need to wait for the reduction producing δ_i to finish as they can be overlapped with it. Computing q_i and n_i is left after the second reduction since these computations require m_i and we want the reductions to be overlapped as much as possible with the most expensive computational kernel, the preconditioning of vector ω_i (step 6 of **Algorithm 4**).

Algorithm 4 IFCG2

$r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
for $i = 0 \ldots i_{max}$ **do**
  for $j = 1 \ldots N$ **do**
    $\gamma_{ij} := (r_{ij}, u_{ij})$
    $\delta_{ij} := (w_{ij}, u_{ij})$
    $m_{ij} := M^{-1} w_{ij}$ ▷ The most expensive step
  end for
  $\gamma_i := \sum_{j=1}^{N} \gamma_{ij}$ ▷ Global reduction on γ_i
  if $i > 0$ **then**
    $\beta_i := \gamma_i / \gamma_{i-1}$
  else
    $\beta_i := 0$
  end if
  for $j = 1 \ldots N$ **do** ▷ AXPYs that only depend on β_i
    $s_{ij} := w_{ij} + \beta_i s_{(i-1)j}$
    $p_{ij} := u_{ij} + \beta_i p_{(i-1)j}$
  end for
  $\delta_i := \sum_{j=1}^{N} \delta_{ij}$ ▷ Global reduction on δ
  if $i > 0$ **then**
    $\alpha_i := \gamma_i / (\delta_i - \beta_i \gamma_i / \alpha_{i-1})$
  else
    $\alpha_i := \gamma_i / \delta_i$
  end if
  for $j = 1 \ldots N$ **do**
    $q_{ij} := m_{ij} + \beta_i q_{(i-1)j}$
    $n_{ij} := A_j m_i$
    $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$
    $x_{(i+1)j} := x_{ij} + \alpha_i p_{ij}$
    $r_{(i+1)j} := r_{ij} - \alpha_i s_{ij}$
    $u_{(i+1)j} := u_{ij} - \alpha_i q_{ij}$
    $w_{(i+1)j} := w_{ij} - \alpha_i z_{ij}$
  end for
end for

There is an interesting trade-off between the IFCG1 and IFCG2 algorithms: While the first one is focused on reducing the cost of the two global reductions by overlapping them with computations, which implies delaying the update of the s_i and p_i vectors, the second tries to run these updates as soon as possible, which requires splitting the single synchronization point composed of two reductions into two parallel dot-products. As such, the IFCG1 formulation aims at reducing the cost of reduction operations while the IFCG2 aims at starting the computations as soon as possible to avoid idle time. IFCG1 and IFCG2 algorithms are thus two complementary approaches that constitute an evolution of the Pipelined CG algorithm aiming at increasing performance. Besides the parallel programming and performance aspects, which are detailedly discussed in sections 4.4.2 and 4.6, it is also important to verify that both IFCG algorithms have similar numerical stability properties as state-of-the-art approaches like Pipelined CG.

4.4 Characteristics of The IFCG Algorithms

4.4.1 Numerical Stability of the IFCG Algorithms

The main issue with the numerical stability of IFCG algorithms is the same as the one displayed by many other Krylov-based methods: The way the residual vector r is computed. It is usually done by just updating the residual of iteration i from the one in iteration $i-1$ via expressions like $r_i = r_{i-1} - \alpha Ap_{i-1}$. However, by doing so, residual r_i may deviate from the true residual $b - Ax_i$. State-of-the-art approaches use a residual replacement strategy to prevent the updated residual r_i to deviate from the true residual. The remedy is to periodically replace the updated r_i by $b - Ax_i$ [38, 76, 77]. The frequency of such replacement is a trade-off between convergence speed and accuracy and some sophisticated strategies exist [77, 78] to deal with it. In the case of IFCG and IFCG2 we do the $r_i = b - Ax_i$ replacement every *FUSE* iterations to avoid hurting the overlap between different iterations.

We run some experiments considering several sparse matrices obtained from the Florida Sparse Matrix Collection [58]. These experiments involve parallel executions of the IFCG1, IFCG2, Pipelined CG and Preconditioned CG algorithms on a 16 cores NUMA node composed of two 8-core sockets. More specific details on the parallel implementations and the precise experimental setup can be found in sections 4.4.2 and 4.5, respectively. Also, Table 4.1 contains a description of the matrices considered in the experiments.

Figure 4.1 displays the evolution of the relative residual $||b - Ax_i||_2/||b||_2$ on matrix *consph* considering the IFCG1, IFCG2, Pipelined CG and Preconditioned CG methods. Data concerning IFCG1 and IFCG2 are expressed in a coarser-grain pattern than Pipelined and Preconditioned CG's data points since we calculate the relative residual every 100 iterations (i. e. *FUSE*=100). We can see that the convergence of the IFCG1 and IFCG2 algorithms is the same as Pipelined CG. Interestingly, Figure 4.1 also displays how the basic Preconditioned CG algorithm has better convergence properties than Pipelined CG, which is consistent with previously reported numerical results [38, 79]. We omit

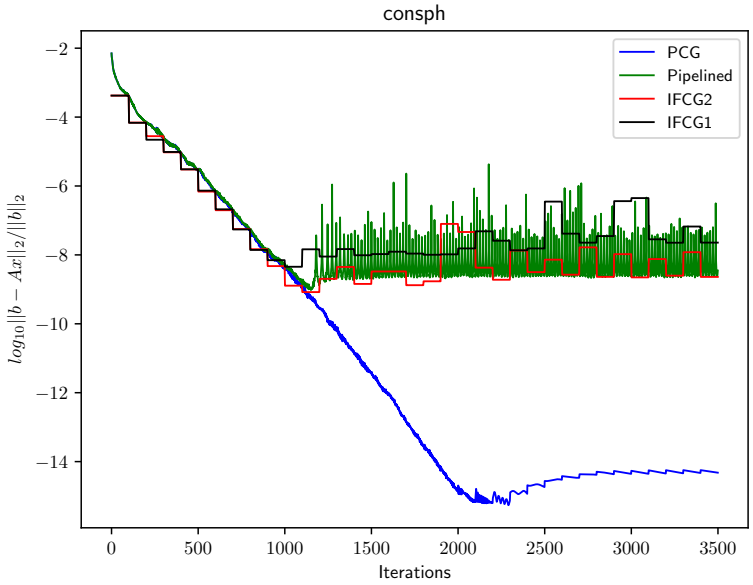

Fig. 4.1 Convergence of the Preconditioned CG, Pipelined CG, IFCG1 and IFCG2 algorithms. Data regarding IFCG1 and IFCG2 is reported every 100 iterations since $FUSE = 100$.

results concerning the rest of the matrices in Table 4.1 since they exhibit the exact same behavior as the one observed with *consph*. Our experiments show how the numerical behavior in terms of the relative residual $||b - Ax_i||_2/||b||_2$ achieved by IFCG1 and IFCG2 matches the state-of-the-art.

Name	Dimension	Nonzeros	Nonzeros%
G3_circuit	1585478	7660826	0.0003%
thermal2	1228045	8580313	0.0006%
ecology2	999999	4995991	0.0005%
af_shell8	504855	17579155	0.0068%
G2_circuit	150102	726674	0.003%
cfd2	123440	3085406	0.02%
consph	83334	6010480	0.087%

Table 4.1 Matrices used for experiments

4.4.2 Parallel Execution of the IFCG Algorithms

The IFCG algorithms have been carefully designed to hide the impact of their global synchronization points by overlapping them with other numerical kernels. Also, IFCG algorithms aim at relaxing data-dependencies between these different kernels by breaking them down into several subkernels that just require a reduced input data set to carry on. These features can significantly improve performance but, in order to exploit them, the IFCG algorithms must run in parallel and enforce the overlap of the

different computational kernels as much as possible. Therefore, we need to specify at the source code level a parallel scheme that meets these requirements and there are several ways to do so.

One option is to statically specify at the application source code level the way different kernels overlap with each other, which should be done by means of sophisticated parallel programming techniques like pools of threads or active waiting loops that trigger work once its input data is ready. However, the optimality of these techniques depends a lot on the parallel hardware where the parallel execution takes place. Therefore, a static approach is not practical since it needs to be adapted to each parallel execution scenario. In this paper we follow a dynamic approach that conceives the parallel execution as a directed acyclic graph where the nodes represent pieces of code (also known as tasks) and the edges are control or data dependencies between them. This approach requires from the programmer to specify the pieces of code or tasks that run in parallel by means of annotations that contain their input or output dependencies. A runtime system orchestrates the parallel run by considering tasks' input or control dependencies and scheduling them into the available parallel hardware once all dependencies are satisfied. The most important shared-memory programming models, like OpenMP, have support for this kind of task-based parallelism and there is also support for running task-based workloads on distributed memory environments [52].

4.4.3 Task-based Formulations of the Pipelined CG and IFCG algorithms

The Pipelined CG, IFCG1 and IFCG2 algorithms can be easily formulated in terms of tasks by just looking at each one of the steps in algorithms 2, 3 and 4 and considering them tasks. Indeed, by means of the #pragma annotations provided by OpenMP it is possible to specify that each one of these steps is a task as well as which are its data dependencies. Control dependencies are typically expressed in terms of sentinels. Importantly, IFCG1 and IFCG2 have many more tasks per iteration than Pipelined CG. Indeed, steps 3-6 and 12-19 of Pipeline CG (**Algorithm 2**) are split into N substeps in **Algorithms 3 and 4**, which implies that we have N tasks in IFCG1 and IFCG2 per each Pipeline CG task. The only exception to this rule is the preconditioning step which is typically split depending on whether or not the chosen preconditioner allows a decomposition in terms of tasks.

Figure 4.2 shows two iterations of the Pipelined CG, IFCG1 and IFCG2 algorithms represented in terms of task graphs. Parameter N is equal to 3, which means that many of the Pipelined CG tasks appearing in the task graph are broken down into 3 tasks by the IFCG1 and IFCG2 methods. In the case of the Pipelined CG algorithm the task named *DOT* represents steps 3 and 4 of **Algorithm 2** while tasks named *Precond* represents step 5, which in this particular case is divided into several tasks. Tasks α and β represent the computations done within step 8. Finally, the task designated as *AXPY* represents steps 12-19. Similarly, the task graph representations of the IFCG1 and IFCG2 methods represent all the steps displayed by algorithms 3-4.

By comparing the center and the left hand side task graphs in Figure 4.2 we can observe how by removing the iteration barrier and breaking the computation routines into blocks we expose much more parallelism to the hardware. Indeed, Pipelined CG has a limited potential for overlapping tasks

belonging to the same iteration and cannot overlap tasks from different iterations at all since its inter-iteration barrier prevents it from doing so. In contrast, IFCG1 displays a much more flexible parallel pattern that can easily overlap tasks belonging to different iterations. IFCG2, by further extracting two AXPYs operations (s, p) that only depend on β, is able to create even more concurrency. The implications and analysis of these varying level of parallelism shown by the different algorithms are explained in Section 4.6.

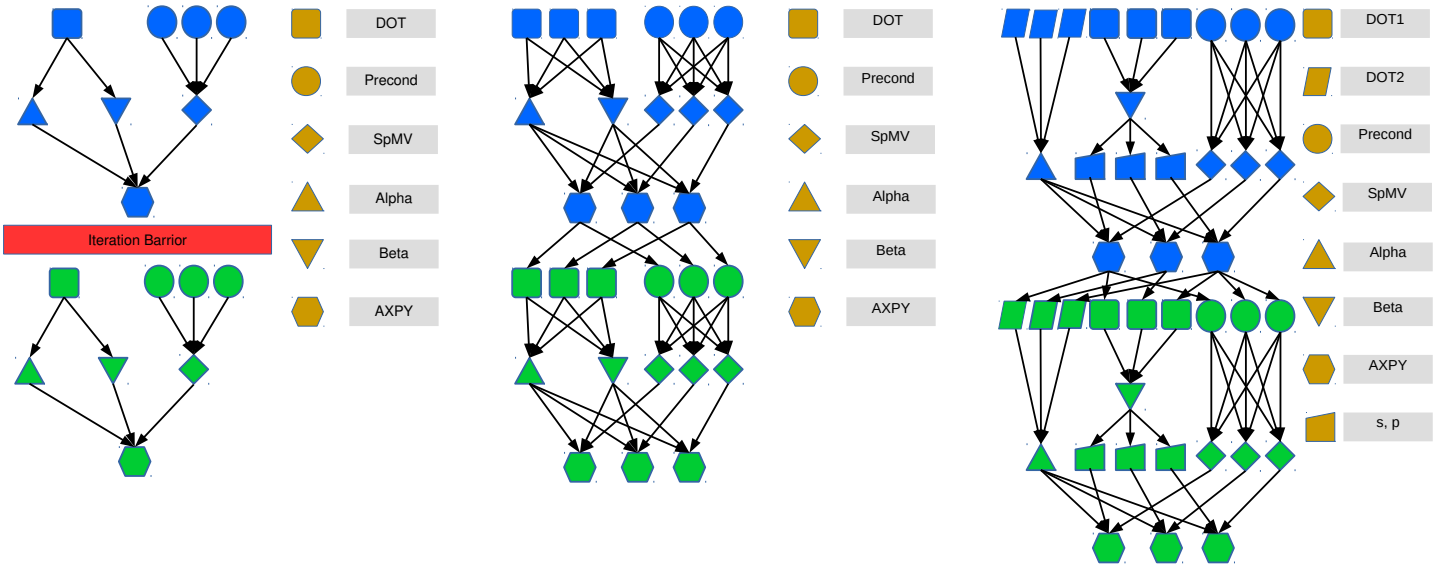

Fig. 4.2 Graphs of tasks representing two Iterations of Pipelined CG (left), IFCG1 (center) and IFCG2 (right), $N = 3$.

4.5 Experimental Setup

We conduct our parallel experiments on a 16-cores node composed of two 8-core Intel Xeon® processors E5-2670 at 2.6 GHz and a 20 MB L3 shared cache memory with SuSe Linux OS. All the algorithms we consider in the evaluation (IFCG1, IFCG2, Preconditioned CG, Pipelined CG [38], Chronopoulos CG [62] and Gropp CG [1]) are implemented using the OpenMP4.0 programming model running on top of the Nanos++ (v0.7a) parallel runtime system [80]. We use the Intel's MKL [81] library to compute the fundamental linear algebra kernels involved in our experiments. All the aforementioned algorithms are implemented with the Block-Jacobi preconditioner with incomplete Cholesky factorization within the blocks. The block size N is set to 64 throughout the experiments and the convergence threshold is $||b - Ax_i||_2/||b||_2 < 10^{-7}$

We consider 7 Symmetric and Positive Definite (SPD) matrices from The University of Florida Sparse Matrix Collection [58]. We use two matrices from circuit simulation problems (*G3_circuit* and *G2_circuit*), two matrices from unstructured Finite Element Method schemes (*thermal2*, *consph*), one matrix from material engineering problems (*af_shell8*) and one matrix from a computational

fluid dynamics problem (*cfd2*). In Table 4.1 we show a more precise description of all considered matrices in terms of their dimensions and sparsity. The considered matrices cover a wide range of dimensions (from 72,000 up to 1,585,478 rows and columns) and sparsity degrees, which makes them representative of the typical problems faced by the CG method and its variants.

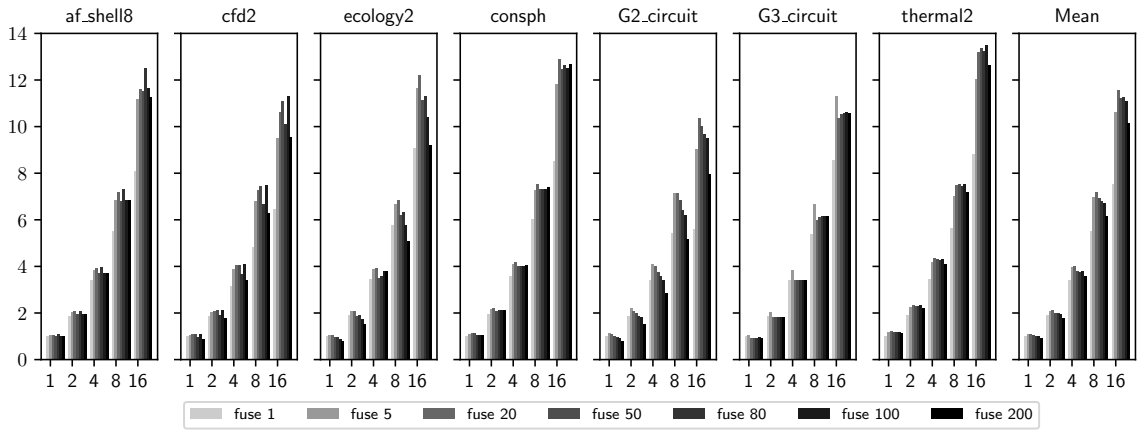

Fig. 4.3 Impact of the *FUSE* parameter on IFCG1. The y-axis represents the achieved speedups with respect to the *FUSE*=1 configuration running on 1 core while x-axis represents core counts.

4.6 Evaluation

In this section we provide a comprehensive evaluation of the IFCG1 and the IFCG2 algorithms and we compare them in terms of performance with the 4 state-of-the-art techniques mentioned in Section 4.5. We first carry out a sensitivity study of the *FUSE* parameter to determine its optimal value. We then compare the performance of IFCG1 and IFCG2 running with this optimal *FUSE* value against the 4 state-of-the-art methods mentioned above. We demonstrate that IFCG1 and IFCG2 achieve a significant degree of overlap between iterations, which provides them with much better performance results than their competitors. Finally, we compare the noise tolerance of IFCG1 and IFCG2 against other CG variants. We consider two different noise regimes, both of them close to realistic noise scenarios, and we demonstrate that the IFCG algorithms are much more tolerant to system noise than state-of-the-art approaches.

4.6.1 Optimizing the *FUSE* Parameter.

As explained in previous sections, by removing the convergence check at the end of each iteration and just checking for convergence once every *FUSE* algorithmic steps, we let computations to overlap across different iterations. However, the algorithm may keep running once the threshold is met since convergence is only checked once every several iterations, which has an impact over the total execution time. If this extra time is larger than the benefits obtained from increasing the overlap across

iterations, IFCG1 and IFCG2 will perform poorly. On the contrary, if we restrict the *FUSE* Parameter too much, that is, if we check for convergence too often, the potential for overlap will be undermined.

In Figure 4.4 we show the impact of the *FUSE* parameter on the scalability of the IFCG1 algorithm when applied to the 8 matrices described in section 4.5. We consider the *FUSE* parameter to be 1, 5, 20, 50, 80, 100 and 200. For each matrix we show the speedup achieved by varying the *FUSE* value and running IFCG1 on 1, 2, 4, 8 and 16 cores over the execution with $FUSE = 1$ on 1 core. In the x-axis we represent the total number of cores involved in the parallel execution while in the y-axis we show the speedup achieved by each technique. The input matrices and the experimental setup are described in Section 4.5.

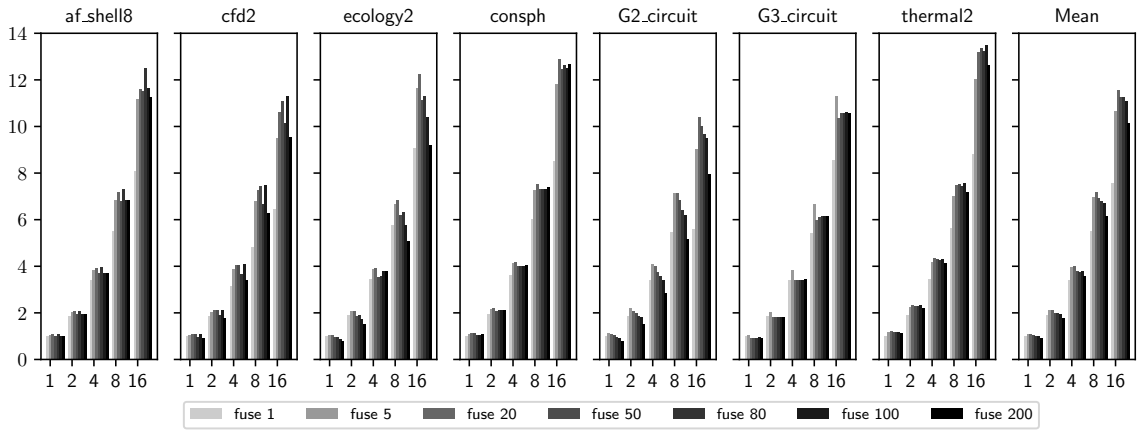

Fig. 4.4 Impact of the *FUSE* parameter on IFCG1. The y-axis represents the achieved speedups with respect to the *FUSE*=1 configuration running on 1 core while x-axis represents core counts.

When running on a single core we achieve speedups of 1.12x, 1.12x and 1.09x over the $FUSE = 1$ configuration when *FUSE* is set to 5, 20 and 50, respectively. This modest speedups are due to the reduction of overheads brought by checks for convergence, i. e. computing $Ax_i - b$, which is done once every *FUSE* iterations. These small benefits decrease for large *FUSE* values due to the extra iterations the algorithm carries out. When the experiments are run on larger core counts the benefits of increasing the *FUSE* value are very significant. Indeed, we achieve average speedups of 10.44x, 11.15x and 10.99x when *FUSE* is set to 5, 20 and 50 and IFCG1 runs on 16 cores with respect to the sequential run with $FUSE = 1$. In general, the benefits of increasing *FUSE* stall at 20 and start to decline when *FUSE* reaches the 200 value. Matrix-wise, results are very consistent since IFCG1 reaches optimal or very close to optimal performance when $FUSE = 20$ for 5 matrices: *cfd2*, *ecology2*, *consph*, *G2_circuit* and *thermal2*. Just for the *af_shell8* and *G3_circuit* matrices the *FUSE* optimal value is different from 20 (80 in the first case and 5 in the second) although the speedups in these optimal points (12.5x and 11.33x respectively), are very close to the ones achieved by the $FUSE = 20$ configuration (11.62x and 10.38x). In general, a *FUSE* value of 20 is the best one for the IFCG1 algorithm. By conducting the same analysis for IFCG2 we find its optimal *FUSE* parameter to be 20 as well.

4.6.2 Evaluation of the IFCG1 and IFCG2 algorithms against state-of-the-art techniques

This section provides an evaluation of the parallel speedups achieved by the IFCG1 and the IFCG2 algorithms and compares them with 4 state-of-the-art techniques: Preconditioned CG (PCG), Pipelined CG [38], Chronopoulos CG [62] and Gropp CG [1]. Both IFCG1 and IFCG2 run with $FUSE = 20$, which is the configuration that provides the best performance on average, as shown in section 4.6.1. Figure 4.5 provides a comparison in terms of speedup considering all 6 CG variants. The x-axis represents the number of cores involved in the parallel run while the y-axis shows the speedups achieved by the different techniques taking the execution time of the Preconditioned CG algorithm on a single core as reference. The experimental setup is described in Section 4.5.

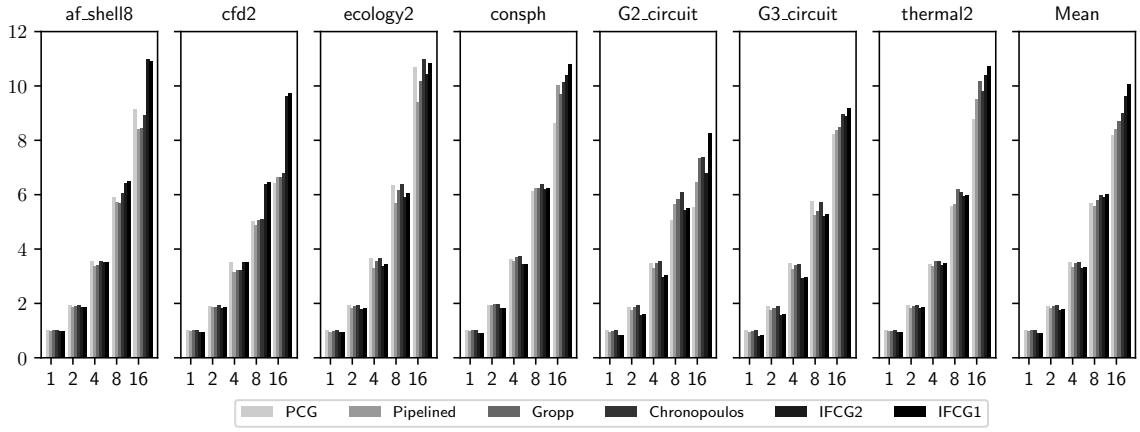

Fig. 4.5 Speedup of all considered CG versions with respect to PCG running on 1 core. The y-axis represents the speedups achieved by the different techniques while x-axis represents core counts.

The most dramatic improvements are achieved when applying IFCG1 and IFCG2 to the *af_shell8* and *cfd2* matrices. For these two matrices IFCG1 running on 16 cores achieves speedups of 10.92x and 10.96x while IFCG2 reaches speedups of 9.72x and 9.62x, respectively. These results are much better than the speedups achieved by the other considered techniques. Indeed, the speedups achieved by the Preconditioned, Pipelined, Gropp and Chronopoulos variants of the CG algorithm are 9.13x, 8.41x, 8.46x and 8.91x in the case of *af_shell8* and 6.41x, 6.63x, 6.66x and 6.8x in the case of *cfd2*, respectively. In the case of the *cfd2* matrix the performance improvements achieved by IFCG1 and IFCG2 are 42.9% and 41.5% better than Chronopoulos, the best state-of-art-technique. IFCG1 provides the highest performance in almost all the cases. The only exception is *ecology2*. In this case, the best speedup on 16 cores is achieved by the Chronopoulos CG (10.98x) although IFCG1 provides a very close speedup of 10.82x when run on 16 cores. This represents a case where techniques proposed in this paper are not better than the state-of-the-art since the input matrix makes the linear system easily scalable (all CG variants achieve speedups close to 10x with respect to PCG running on a single core when solving the *ecology2* on 16 cores).

Besides individual observations, the average speedup over the PCG algorithm running in a single core of both IFCG1 and IFCG2 is significantly better than the one achieved by the other CG versions. Indeed, we can observe from Figure 4.5 that IFCG1 and IFCG2 reach an average speedup when run on 16 cores of 10.06x and 9.64x, respectively. The other variants achieve speedups of 8.20x (PCG), 8.40x (Pipelined), 8.70x (Gropp) and 8.99x (Chronopoulos) when run on 16 cores. On average, IFCG1 and IFCG2 provide 11.8% and 7.1% performance improvements over the best state-of-the-art technique (Chronopoulos CG). Table 4.2 lists the iteration counts for all considered matrices and CG variants. Due to the residual check done once every *FUSE* iterations the IFCG1 and IFCG2 algorithms are bound to take more iterations than the other approaches and indeed they take 16 more iterations on average than the other CG variants. This overhead is effectively compensated by overlapping adjacent iterations, as Figure 4.5 demonstrates.

	PCG	**Chronopoulos**	**Pipelined**	**Gropp**	**IFCG**	**IFCG2**
af_shell8	676	676	676	676	680	680
cfd2	563	563	563	563	580	580
ecology2	678	678	678	678	680	680
consph	912	911	926	911	940	940
G2_circuit	430	430	430	430	440	440
G3_circuit	428	428	428	428	480	480
thermal2	2076	2076	2077	2076	2080	2080
Average	794	794	796	794	810	810

Table 4.2 Iteration counts of all considered methods and matrices. *FUSE* = 20 for IFCG1 and IFCG2

4.6.3 Visualizing The Overlap Pattern

The main reason behind the good behavior of IFCG1 and IFCG2 in terms of performance is their capacity to overlap different iterations and this section aims to provide a visual proof of this overlap. Figure 4.6 displays three 16-core runs composed of 19 iterations. On top of Figure 4.6 we represent a Pipelined CG execution while IFCG1 is shown in the middle and IFCG2 at the bottom. In the y-axis we represent the 16 threads involved in the parallel run while in the x-axis we show time. The three represented algorithms are applied to the *af_shell8* matrix. In all three views the iterations are marked by distinct colors and all are trimmed for the same time duration (the duration of the pipelined CG since it takes the longest time). The white gaps in Figure 4.6 represent either idle time or system software activity.

Boundaries between iterations are clearly marked by white gaps in the Pipelined CG representation of Figure 4.6. Computations belonging to the same iteration are clearly executed in isolation by the Pipelined CG algorithm while this lock-step execution mode is not present in the IFCG1 and IFCG2 representations. For these two cases computations belonging to different iterations are overlapped in a way that only their color identifies the iteration they belong to. There are some small regions represented in white in the IFCG1 and IFCG2 parallel runs that are overlapped with iterations, which account for system software activity. The idle time is almost completely eliminated. The white areas overlapped with the first iteration of IFCG1 and IFCG2 represent computations belonging to previous

iterations while the large white areas that appear after the 19 iterations mean that the parallel execution has already finished.

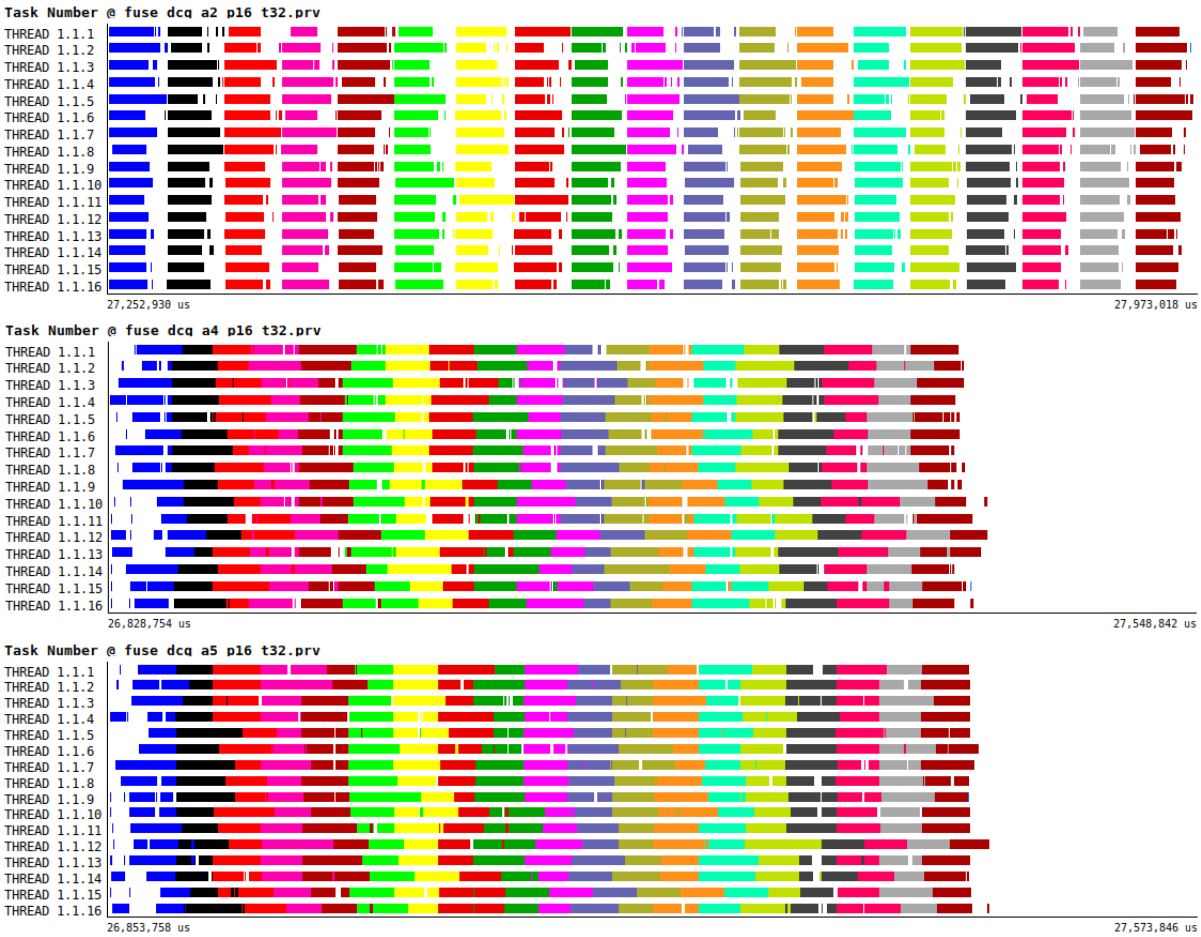

Fig. 4.6 Visualization of 19-iteration runs on 16 cores of Pipelined CG (top), IFCG1 (middle) and IFCG2 (bottom). The input matrix is *af_shell8*

4.6.4 Tolerance to System Noise

HPC infrastructures frequently get their performance severely degraded by system noise or jitter, which is caused by factors like OS activity, network sharing effects or other phenomena [82]. Although the effects of system jitter may be negligible as long as they are kept at the local scale, parallel operations like reductions or synchronizations are known to strongly amplify its effects by propagating jitter across the whole parallel system [83]. Since algorithms IFCG1 and IFCG2 presented in this paper perform much less reductions or synchronization operations than the Preconditioned CG or the Chronopoulos CG algorithms, they are much more tolerant to jitter effects. To evaluate this additional advantage of IFCG algorithms, this section compares the performance of these 4 algorithms (Preconditioned CG, Chronopoulos CG, IFCG1 and IFCG2) on a noisy regime. The Gropp and

Pipelined versions of CG are not considered in this section since, as Figure 4.5 demonstrates, their behavior is between the one displayed by PCG and Chronopoulos.

We run the 4 algorithms mentioned above on 16 cores considering the input matrices and the experimental setup described in Section 4.5 and we inject uniformly distributed random noise with an amplitude of 10μs and frequencies of 8kHz and 2kHz. Such noise regimes are close to the measured ones on real systems (1kHz and 25μs [84]) and produce, on average, overheads of 8% ($8 \cdot 10^3 \cdot 10^{-5} = 0.08$) and 2% in sequential computations, respectively. Therefore, any extra overhead suffered by parallel applications under these noise regimes is brought by amplification effects due to parallel synchronization or reduction operations. Parallel executions may also filter out noisy events that take place during idle execution phases.

In Figure 4.7 we show the elapsed time running on 16 cores of the Preconditioned CG, Chronopoulos CG, IFCG1 and IFCG2 under a noiseless, a 10μs-2kHz and a 10μs-8kHz noise regimes. The y-axis displays the execution time normalized to the Preconditioned CG execution without noise and the x-axis shows the obtained results per matrix plus their average values. On average, the Preconditioned and the Chronopoulos CG algorithms suffer degradation of 19.0% and 14.6% of their execution time, respectively under the 10μs-8kHz noise regime. They are much larger than the 8% degradation expected to be suffered by purely sequential applications, which implies that noise is amplified by parallel operations like reductions or synchronizations. In contrast, IFCG1 and IFCG2 suffer much milder degradation of just 6.2% and 6.9%, respectively, when exposed to 10μs-8kHz noise. IFCG1 has a 1.18x speedup over Chronopoulos under the 10μs-8kHz, that is, it runs 18.0% faster. Interestingly, both IFCG algorithms run faster under this noisy regime than their state-of-the-art counterparts under the noiseless regimes. In Figure 4.7 we also show results considering the 10μs-2kHz scenario. For this case, the Preconditioned and the Chronopoulos CG algorithms suffer degradation of 6.1% and 5.1% of their execution time, respectively. In contrast, IFCG1 and IFCG2 suffer milder degradation of just 1.1% and 1.7%, respectively.

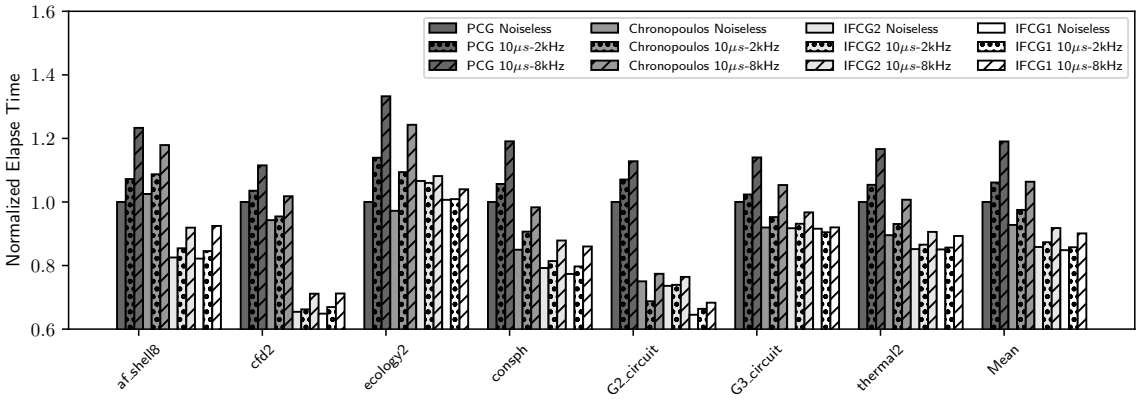

Fig. 4.7 Behavior of different variants of CG running on 16 cores under noiseless, 10μs-2kHz and 10μs-8kHz noise regimes.

4.7 Conclusions

This chapter presents the IFCG1 and IFCG2 algorithms, which are variants of the CG algorithm where most of the inter-iteration barriers are removed and linear kernels are split into several subkernels. The main difference between IFCG1 and IFCG2 is that the first one aims at hiding parallel reduction costs while the second one avoids idle time by starting the execution of the linear subkernels as soon as possible. The *FUSE* parameter specifies how often both IFCG1 and IFCG2 check for convergence. To maximize the performance of these algorithms the *FUSE* parameter needs to be set up to the optimal value by means of an exhaustive search. This parameter is not input dependent and we find its optimal value to be 20.

To compare the performance of IFCG1 and IFCG2 against other relevant variants of the CG algorithm, we consider 8 matrices from the Florida Sparse Matrix Collection [58] with varying sparsity degrees and dimensions. We find that IFCG1 and IFCG2 achieve significant performance improvements with respect to the state-of-the-art due to their flexibility to overlap computations belonging to different iterations. We also show how reducing the number of global synchronization points makes IFCG1 and IFCG2 much less sensitive to system noise perturbations than their state-of-the-art counterparts. Also, both IFCG1 and IFCG2 display the same numerical stability properties as the most relevant previous techniques.

5

Communication Reduction in Deep Neural Network Training

5.1 Introduction

The use of Deep Neural Networks (DNNs) is becoming ubiquitous in areas like computer vision (e.g., image recognition and object detection) [16, 85], speech recognition [86], language translation [87], and many more [88]. DNNs provide very competitive pattern detection capabilities and, more specifically, Convolutional Neural Networks (CNNs) classify very large image sets with remarkable accuracy [89]. Indeed, DNNs already play a very significant role in the large production systems of major IT companies and research centers, which has in turn driven the development of advanced software frameworks for the deep learning area [55] as well as DNN-specific hardware accelerators [90, 91]. As an example, deep learning solutions are being coupled with physical computational models for solving pattern classification problems in the context of large-scale climate simulations [92]. Despite all these accomplishments, deep learning models still suffer from several fundamental problems: the neural network topology is determined through a long and iterative empirical process, the training procedure has a huge cost in terms of time and computational resources, and the inference process of large network models incurs considerable latency to produce an output, which is not acceptable in domains requiring real-time responses like autonomous driving.

The DNN training process typically relies on the backpropagation procedure [93], which requires solving an optimization problem aimed at discovering the values of network weights that better fit the training data. A possible way to carry out the backpropagation process is the Gradient Descent (GD) method [94], which aims at fitting the weights to the training data by considering, at each iteration, the steepest descent direction in terms of an error function. A popular variant of the GD procedure is the Stochastic Gradient Descent (SGD) method [95], which computes the gradient against several randomly chosen samples at each iteration. Today's common practice to train DNNs is to split the data set into several subsets, called batches, and let each iteration of SGD to compute a descent direction or gradient that contains contributions of all the samples belonging to the same batch. SGD converges

faster than GD since it updates network parameters at the end of each batch once all samples are processed.

To tackle the large amount of Floating Point computations required to train a DNN, GPUs are usually employed [96]. They exploit the large amount of data-level parallelism of deep learning workloads. Although GPUs and other hardware accelerators have been successfully employed to boost the training process, data exchanges involving different accelerators may incur significant performance penalties.

This chapter describes and evaluates a method to accelerate the training of DNNs by reducing the cost of data transfers across heterogeneous high-end architectures integrating multiple GPUs. By relying on DNNs tolerance to data representation formats smaller than the commonly used 32-bit Floating Point (FP) standard [14, 15], this chapter describes how to dynamically adapt the size of data sent to GPU devices without hurting the quality of the training process. Our solution is designed to efficiently use the incoming bandwidth of the GPU accelerators. It relies on an adaptive scheme that dynamically adapts the data representation format required by each DNN layer and compresses network parameters before sending them over the parallel system. This scheme enables DNNs training to progress in a similar rate as if the 32-bit FP format was used. This chapter makes the following contributions:

- It proposes the *Adaptive Weight Precision (AWP)* algorithm, which dynamically adapts the numerical representation of DNN weights during training. AWP relies on DNNs' tolerance for reduced data representation formats. It defines the appropriated data representation format per each network layer during training without hurting network accuracy.

- It proposes a new *Approximate Data Transfer (ADT)* procedure to compress DNN's weights according to the decisions made by the AWP algorithm. ADT relies on both thread- and SIMD-level parallelism and is compatible with architectures like IBM's POWER or x86. ADT is able to compress large sets of weights with minimal overhead, which enables the large performance benefits of our approach.

- It evaluates ADT and AWP on two high-end systems: The first is composed of two x86 Haswell multicore devices plus four Tesla GK210 GPU accelerators and the second system integrates two POWER9 chips and four NVIDIA Volta V100 GPUs. Our evaluation considers the Alexnet [16], the VGG [17] and the Resnet [18] network models applied to the ImageNet ILSVRC-2012 dataset [19]. Our experiments report average performance benefits of 6.18% and 11.91% on the x86 and the POWER systems, respectively. Our solution does not reduce the quality of the training process since networks final accuracy is the same as if they had been trained with the 32-bit Floating Point format.

Many proposals describe how data representation formats smaller than the 32-bit Floating Point IEEE standard can be applied to deep learning workloads without harming their accuracy [14, 97, 98]. This chapter presents the first approach that uses reduced data formats to minimize data movement

during DNN training. This chapter is particularly relevant from the high-performance computing perspective since it proposes a methodology to accelerate DNNs training in heterogeneous high-end systems, which are extensively used to run deep learning workloads [96].

This chapter is structured as follows: Section 5.2 describes our first contribution, the Adaptive Weight Precision algorithm (AWP). Section 5.3 details the Approximate Data Transfer (ADT) procedure. Section 5.5 describes the experiments we conduct to evaluate AWP and ADT on three state-of-the-art neural networks. Finally, Section 5.6 summarizes the main conclusions of this chapter.

5.2 The Adaptive Weight Precision (AWP) Algorithm

The Adaptive Weight Precision (AWP) algorithm relies on the tolerance of DNNs to data representation formats smaller than the 32-bit Floating Point standard. Indeed, previous work indicates that, unlike scientific codes focused on solving partial differential equations or large linear systems, neural networks do not always require 32-bit representation during training [14, 97]. Even more, adding stochastic noise to certain variables during the learning phase improves DNNs accuracy [99, 100, 101]. Nevertheless, when facing unknown scenarios in terms of new workloads or parameter settings, the data representation requirements of DNNs are non-trivial to be determined and, to make things more complicated, they may change as the training phase progresses.

Algorithm 5 Adaptive Weight Precision (AWP) Algorithm

```
1: BitsPerLayer := [B_0, B_1, ..., B_NumLayers] ▷ List storing the number of bits corresponding to the data representation of each layer
2: IntervalCounter := [0, 0, ..., 0] ▷ List storing the number of times the relative change rate fails to meet the threshold per layer
3: for batch := 0 ... NumBatches do
4:     Apply backpropagation to batch
5:     for layer := 0 ... NumLayers do
6:         δ := (|W_batch,layer| − |W_batch−1,layer|) / |W_batch−1,layer|
7:         if δ < T then
8:             IntervalCounter_layer += 1
9:         end if
10:        if IntervalCounter_layer == INTERVAL then
11:            BitsPerLayer_layer += N
12:            IntervalCounter_layer := 0
13:        end if
14:    end for
15: end for
```

The AWP algorithm dynamically determines data representation requirements per each network layer by monitoring the evolution of the l^2-norm of the weights. AWP identifies the number of bits that are needed to represent DNNs weights and guarantees the progress of the training process. AWP assigns the same data representation format to all weights belonging to a certain network layer. The training starts with a relatively small data representation that is independently increased for each layer.

Algorithm 5 displays a pseudo-code description of AWP. Once the backpropagation process has been applied to a given batch, AWP iterates over all network layers. The algorithm computes per each batch and network layer the l^2-norm of all its weights' values and derives the relative change

rate δ of the l^2-norm with regard to the previously processed batch. For the batch i, the change rate is defined as $\delta_i = (|W_i| - |W_{i-1}|)/|W_{i-1}|$, where W_i is the vector of weights of a certain layer while batch i is processed. Every time the change rate is below a given threshold T for a certain layer, the algorithm accounts for it by increasing the $IntervalCounter$ parameter. The algorithm increases N bits of precision if the change rate is below T during a certain number of batches defined by the parameter $INTERVAL$ and sets the $IntervalCounter$ parameter of the corresponding layer to zero. Section 5.5.1 describes how we determine the values of parameters T, $INTERVAL$, and N.

5.3 The Approximate Data Transfer (ADT) Procedure

The Approximate Data Transfer (ADT) procedure compresses network's weights before they are transferred to the GPUs. In the context of DNNs training on heterogeneous multi-GPU nodes, CPU multicore devices are typically responsible for orchestrating the parallel run and updating DNN parameters. Once the process of a batch starts, the updated parameters including the weights W are sent to each GPU. If the set of parameters does not fit in GPUs' main memory, they are sent on several phases as the different GPUs need them. The different samples of each batch are evenly distributed across all GPUs. Therefore, each GPU computes its contribution to the gradient ΔW by processing its corresponding set of samples. The CPU multicore device subsequently gathers all contributions to the gradient and combines them to update the weights $W \leftarrow W - \mu(\frac{1}{n}\sum_i^n \Delta W_i)$, where μ is the learning rate.

Data movement involving different GPU devices increases as the network topology becomes more complex or the number of training samples grows, which can saturate the system bandwidth and become a major performance bottleneck. This paper mitigates this issue by compressing network weights before they are sent to the GPU devices. The AWP algorithm described in Section 5.2 determines, for all weights belonging to a particular network layer, the number of bits to send. In this context, to efficiently compress and decompress network weights, ADT uses of two procedures that constitute its fundamental building blocks. These procedures are complementary and applied either before or after data transfers to GPU devices.

- **Bitpack** compresses the weights discarding the less significant bits on the CPU side;
- **Bitunpack** converts the weights back to the IEEE-754 32-bit Floating Point format on the GPUs.

Figure 5.1 provides an example including a multicore CPU and two GPU devices to describe the way both Bitpack and Bitunpack procedures operate. All neural network parameters (weights and biases) are updated at the CPU level, which is where the Bitpack procedure takes place. We do not apply the Bitpack procedure to the network biases since we do not observe any significant performance benefit from compressing them. Since each output neuron requires just one bias parameter, the total number of them is significantly smaller than the total number of weights. At the beginning of

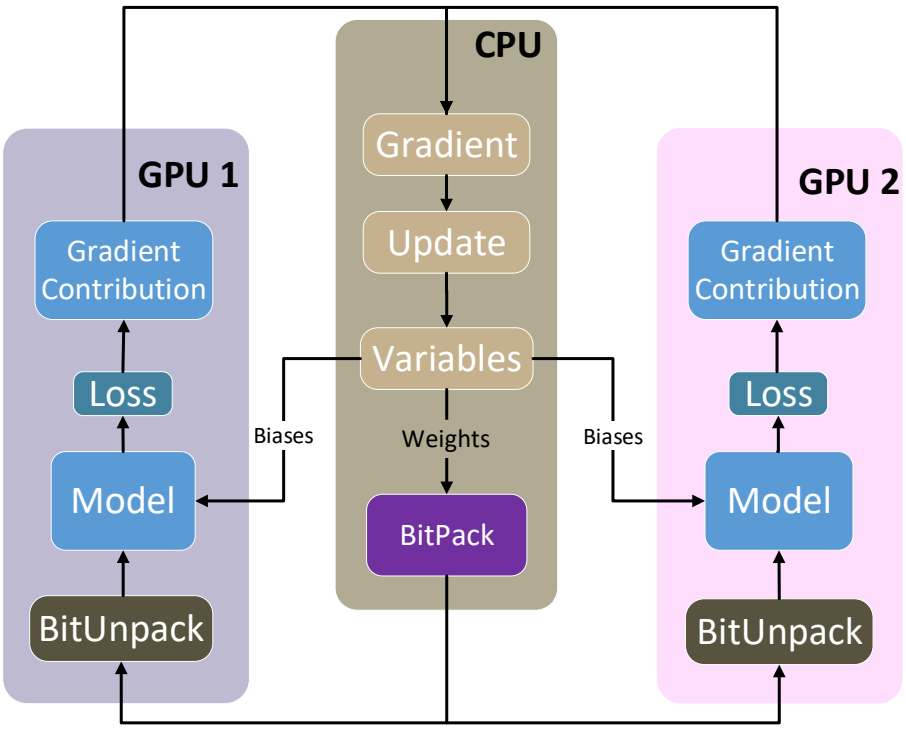

Fig. 5.1 The ADt on a 2-GPU system. Variables include: weights which go through the ADt procedure and biases which are sent directly to the GPUs to build the network model together with the unpacked weights.

each SGD iteration the compressed weights are sent to each GPU together with the biases and the corresponding training samples. Each GPU uncompresses the weights, builds the neural network model and, finally, computes its specific contribution to the gradient. These contributions are sent to the CPU, which gathers all of them, computes the gradient and updates network parameters.

The Bitpack operation runs on CPU multicore devices. To boost Bitpack we use OpenMP [22] and Single-Instruction Multiple Data (SIMD) intrinsics. OpenMP is used to run Bitpack on several threads. The use of SIMD instructions allows Bitpack to operate at the SIMD register level, which avoids incurring large performance penalties in the process of producing the reduced-size weights. We implement two versions of Bitpack. One version uses Intel's AVX2 [102] instruction set and the other one relies on AltiVec [103]. Bitpack can be implemented on top of any SIMD instruction set architecture supporting simple byte shuffling instructions at the register level. The Bitunpack procedure runs on the GPUs.

It can be trivially parallelized since each weight is mapped to a single 32-bit FP variable, which means that GPUs can process a large amount of weights simultaneously and efficiently build the DNN model. In fact, Bitunpack incurs negligible overhead as Section 5.5.6 shows.

ADT manipulates the internal representation of network weights by discarding some bits. We use the standard 32-bit IEEE-754 single-precision Floating Point format [104] (1 bit sign, 8 bits exponent and 23 bits mantissa) for all the computation routines. The Bitpack method considers network weights as 32-bit words where rounding to N bits means discarding the lowest $32 - N$ bits.

Algorithm 6 High Level Pseudo-code Version of Bitpack

```
1: W                                  ▷ Array of 32-bit Floating Point values containing weights
2: Pw                                 ▷ Array containing the reduced precision weights
3: RoundTo                            ▷ Number of bytes to keep per weight
4: POffset := 0                       ▷ Indicates the current size (in bytes) of Pw
5: for weight in W do
6:     Pw[POffset : POffset+RoundTo] := weight[0 : RoundTo] ▷ Copy most significant RoundTo
   bytes to Pw
7:     POffset := POffset + RoundTo
8: end for
```

5.3.1 Bitpack

A high-level version of the Bitpack procedure in terms of pseudo-code is illustrated by Algorithm 6. The algorithm requires a couple of arrays: the input array W, which contains all the weights of a certain network layer, and an output array Pw, which stores the compressed versions of these weights. The algorithm goes through the entire W input array, per each weight, copies the most significant *RoundTo* bytes to the output array Pw. Our Bitpack implementation manipulates data at the byte granularity. We do not observe significant performance benefits when operating at finer granularity in the experiments we run. The AWP algorithm described in Section 5.2 determines the data representation format per each network layer. The number of bits of the chosen format is rounded to the nearest number of bytes that retains all of its information (E.g., if AWP provides the value 14, *RoundTo* will be set to 2 bytes). The Pw array is sent to the GPUs once the Bitpack procedure finishes compressing network weights.

Deep networks usually contain tens or even hundreds of millions of weights [16, 17, 105], which makes any trivial implementation of Algorithm 6 not applicable in practice. We mitigate compression costs by observing that Algorithm 6 is trivially parallel since processing one weight just requires the *RoundTo* parameter. Algorithm 7 shows how to parallelize the Bitpack procedure by using OpenMP threads. Each thread takes care of a certain portion of the array storing network weights.

5.3.2 Single Instruction Multiple Data Bitpack

Since all weights within one layer are processed in the same way by the Bitpack procedure, we can leverage Single Instruction Multiple Data (SIMD) instructions to vectorize it. Most state-of-the-art architectures implement SIMD instruction set: IBM's AltiVec [103], Intel's Advanced Vector Extensions (AVX) [102], and ARM's Neon [106]. In our experiments we use Intel's AVX2 [102], which implements a set of SIMD instructions operating over 256-bit registers, and IBM's AltiVec

Algorithm 7 Bitpack with OpenMP

```
1: W                                  ▷ Array of 32-bit Floating Point values containing weights
2: Pw                                 ▷ Array containing the reduced precision weights
3: RoundTo                            ▷ Number of bytes to keep per weight
4: NumThreads                         ▷ Number of OpenMP threads
5: #pragma omp parallel for
6: for weight in W do
7:     POffset := Corresponding position in Pw
8:     Pw[POffset : POffset+RoundTo] := weight[0 : RoundTo]   ▷ Copy the most significant
   RoundTo bytes to Pw
9: end for
```

instruction set [103], which has SIMD instructions operating over 128-bit registers. Section 5.4 describes the specific details of our evaluation considering both x86 and POWER architectures.

Figure 5.2 shows the byte-level operations of SIMD-based Bitpack applied to eight 32-bit weights and implemented with AVX2. The *RoundTo* parameter is set to 3, which implies discarding the last 8 bits of each weight since the target data representation is 24-bit long. First, eight 32-bit Floating Point weights are loaded to a 256-bit register. In the next step, we use *_mm256_shuffle_epi8* to shuffle the least significant eight bits of each weight to the least significant bits of their respective 128-bit lane (see the grey area of Figure 5.2 Step 2) and pack the rest of the bits together. Afterwards we use *_mm256_permutevar8x32_epi32* to do the same operation across the two 128-bit lanes. Finally, we use *_mm256_maskstore_epi32* to just store the resulting 192 bits to the target array. Not all AVX2 instructions operate over the entire 256-bit register. Instead, many of them conceive the register as two 128-bit lanes and operate on them separately. This is the reason way we can not carry out Steps 2 and 3 by using a single AVX2 instruction.

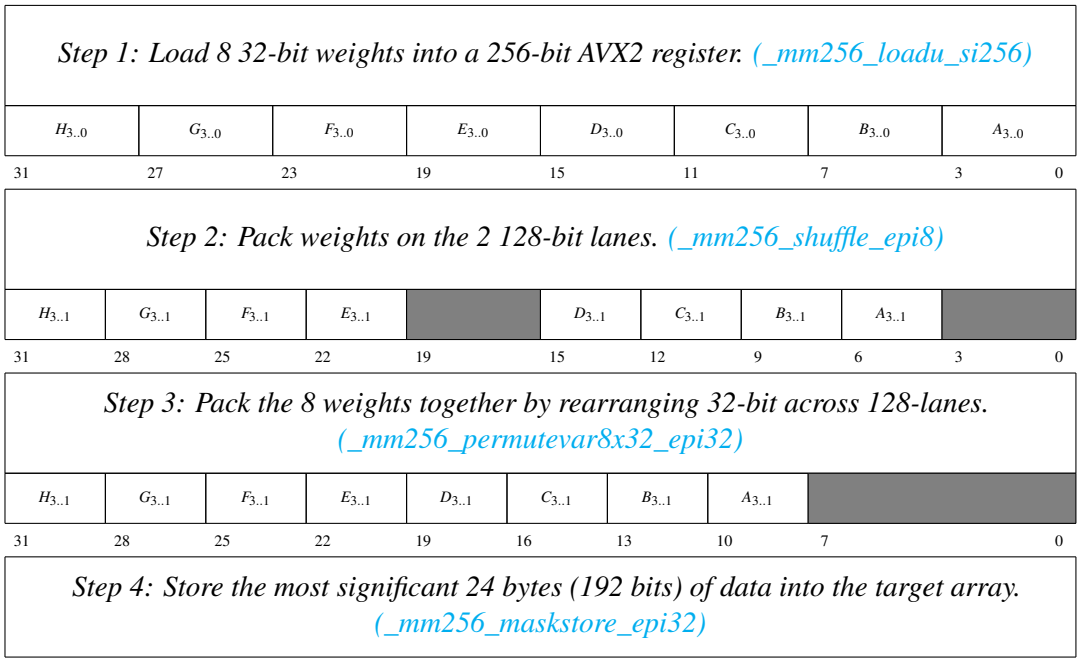

Fig. 5.2 Bitpack implemented with AVX2, RoundTo=3

Algorithm 8 Bitpack with OpenMP + AVX2

```
1:  W                                  ▷ Array of 32-bit Floating Point values containing weights
2:  Pw                                 ▷ Array containing the reduced precision weights
3:  RoundTo                            ▷ Number of bytes to keep per weight
4:  #pragma omp parallel for
5:  for weights in W do
6:      _mm256_loadu_si256             ▷ Load 8 32-bit weights
7:      _mm256_shuffle_epi8            ▷ Compress at each 128-bit lane
8:      _mm256_permutevar8x32_epi32    ▷ Shuffle the compressed weights into the most
    significant bits
9:      _mm256_maskstore_epi32         ▷ Store compressed weights to the target array
10: end for
```

Algorithm 9 Bitunpack on GPU

```
1:  Pw                                 ▷ Array containing compressed weights
2:  W                                  ▷ Array of 32-bit Floating Point values containing weights
3:  RoundTo                            ▷ The number of bytes that are going to be kept
4:  for UnitId := 0 . . . NumUnit do
5:      Distribute W and Pw across all the computation units in the GPU
6:      POffset := 0
7:      for weight in W do
8:          weight := Pw[POffset : POffset+RoundTo] ≪ (4 - RoundTo) * 8
9:          POffset := POffset + RoundTo
10:     end for
11: end for
```

Algorithm 8 summarizes our implementation of the Bitpack procedure with AVX2. It exploits two-level parallelism: first, the input array of weights is distributed across several threads. Second,

within each thread, the compression of each eight 32-bit weights subset is performed at the register level by means of byte shuffling instructions. This sophisticated procedure exploiting parallelism at both thread and SIMD register levels uses all the available hardware and avoids costly memory accesses.

5.3.3 Bitunpack

Once data in reduced-size format reaches the target GPU, the Bitunpack procedure immediately restores them into their original IEEE-754 32-bit Floating Point format. We display pseudo-code describing this process in Algorithm 9. Bitunpack reads the reduced-sized weights from array *Pw* and assigns additional bits to them. Bitunpack gives zero values to these additional bits. We distribute the Bitunpack process across the whole GPU, which enables an extremely parallel scheme exploiting GPUs manycore architecture.

The Bitunpack routine is developed using CUDA [107]. Our code runs in parallel on *N* CUDA threads and the CUDA runtime handles the dynamic mapping between threads and the underlying GPU compute units. Since each thread involved in the parallel run targets a different portion of the *Pw* array, our Bitunpack procedure exposes a large amount of parallelism able to exploit the large number of compute units integrated into high-end GPU devices.

5.4 Experimental Setup

The experimental setup considers a large image dataset, three state-of-the-art neural network models and two high-end platforms. The following sections describe all theses elements in detail.

5.4.1 Image Dataset

We consider the ImageNet ILSVRC-2012 dataset [19]. The original ImageNet dataset includes three sets of images of 1000 classes each: training set (1.3 million images), validation set (50,000 images) and testing set (100,000 images). Considering 1000 classes makes the training process around 170 hours long, which is prohibitively expensive since our large experimental campaign considers different network models, batch sizes and hardware platforms. To reduce the execution time of our experiments we consider a subset of 200 classes for both the training and the validation dataset, which keeps the training time under manageable margins. For the rest of this paper, we refer to the 200 classes dataset as ImageNet200. Since it is a common practice [17], we evaluate the ability of a certain network in properly dealing with the ImageNet200 dataset in terms of the top-5 validation error computed over the validation set.

Table 5.1 Neural network configurations: The convolutional layer parameters are denoted as "conv<receptive field size>-<number of channels>". The ReLU activation function is not shown for brevity. The building blocks of Resnet and the number of times they are applied are shown in a single cell.

Alexnet	**VGG**	**Resnet-34**
input(224x224 RGB image)		
conv11-64	conv3-64	conv7-64
maxpool		
conv5-192	conv3-128	conv3-64 conv3-64 x3
maxpool		
conv3-384	conv3-256 conv3-256	conv3-128 conv3-128 x4
maxpool		
conv3-384	conv3-512 conv3-512	conv3-256 conv3-256 x6
maxpool		
conv3-256	conv3-512 conv3-512	conv3-512 conv3-512 x3
maxpool		avgpool
FC-4096		
FC-4096 FC-4096	FC-4096	
FC-200		
softmax		

5.4.2 DNN Models and Training Parameters

We apply the AWP algorithm along with the ADT procedure on three state-of-the-art DNN models: a modified version of Alexnet [16] with an extra fully-connected layer of size 4096, the configuration A of the VGG model [17] and the Resnet network [18]. All hidden layers are equipped with a Rectified Linear Units (ReLU) [16]. The exact configurations of the three neural networks are shown in Table 5.1. The Alexnet model is composed of 5 convolutional layers and 4 fully-connected ones, VGG contains 8 convolutional layers and 3 fully-connected ones and Resnet is composed of 33 convolutional layers and a single fully-connected one.

We use momentum SGD [108] to guide the training process with momentum set to 0.9. The training process is regularized by weight decay and the L_2 penalty multiplier is set to 5×10^{-4}. We apply a dropout regularization value of 0.5 to fully-connected layers. We initialize the weights using a zero-mean normal distribution with variance 10^{-2}. The biases are initialized to 0.1 for Alexnet and 0

for both VGG and Resnet networks. For the Alexnet and VGG models we consider training batch sizes of 64, 32 and 16. To train the largest network we consider, Resnet, we consider batch sizes of 128, 64 and 32. The 16 batch size incurs in a prohibitively expensive training process for Resnet and, therefore, we do not use it in our experimental campaign.

For Alexnet we set the initial learning rate to 10^{-2} for the 64 batch size and decrease it by factors of 2 and 4 for the 32 and 16 batch sizes, respectively. In the case of VGG we set the initial learning rate to 10^{-2} for the 64, 32 and 16 batch sizes, as in the state-of-the-art [17]. In the case of Resnet the learning rate is 10^{-2} for the batch size of 32 and 0.1 for the rest. For all network models we apply exponential decay to the learning rate throughout the whole training process in a way the learning rate decays every 30 batches by a factor of 0.16, as previous work suggests [105]. For Resnet we obtain better results by adapting precision at the Resnet building blocks level [18] instead of doing so in a per-layer basis.

5.4.3 Implementation

Our code is written in Python on top of Google Tensorflow [55]. Tensorflow is a data-flow and graph-based numerical library where the actual computation is carried out according to a computational graph constructed beforehand. The computational graph defines the order and the type of computations that are going to take place. It supports NVIDIA's NCCL library.

To enable the use of both Bitpack and Bitunpack routines, we integrate them into Tensorflow using its C++ API. Tensorflow executes the two routines before sending the weights from the CPU to the GPU and right after receiving the weights on the GPU side, respectively. The Bitpack routine is implemented using the OpenMP 4.0 programming model. There are two versions of this routine using either Intel's AVX2 or AltiVec instructions, as explained in Section 5.3. Bitunpack is implemented using CUDA 8.0 and CUDA 10.0 respectively on the two platforms [107].

5.4.4 Hardware Platforms

We conduct our experiments on two clusters featuring the x86 and POWER architectures. The x86 machine is composed of two 8-core Intel Xeon ®E5-2630 v3 (Haswell) at 2.4 GHz and a 20 MB L3 shared cache memory each. It is also equipped with two Nvidia Tesla K80 accelerators, each of which hosts two Tesla GK210 GPUs. It has 128 GB of main memory, distributed in 8 DIMMs of 16 GB DDR4 @ 2133 MHz. The 16-core CPU and the four GPUs are connected via a PCIe 3.0 x8 8GT/s. The operating system is RedHat Linux 6.7. Overall, the peak performance of the two 8-core sockets plus the four Tesla GK210 GPUs is 6.44 TFlop/s.

The POWER machine is composed of two 20-core IBM POWER9 8335-GTG at 3.00 GHz. It contains four NVIDIA Volta V100 GPUs. Each node has 512 GB of main memory, distributed in 16 DIMMS of 32 GB @ 2666 MHz. The GPUs are connected to the CPU devices via a NVIDIA NVLink 2.0 interconnection [9]. The operating system is RedHat Linux 7.4. The peak performance of the two 20-core sockets plus the four V100 GPUs is 28.85 TFlop/s.

5.5 Evalutation

In this section we evaluate the capacity of the AWP algorithm and the ADT procedure to accelerate DNNs training. We show how our proposals are able to accelerate the training phase of relevant DNN models without reducing the accuracy of the network.

5.5.1 Methodology

Our experimental campaign considers batch sizes of 64, 32 and 16 for the Alexnet and VGG models and 128, 64 and 32 for the Resnet network. For each model and batch size, the *baseline* run uses the 32-bit Floating Point precision for the whole training. The data represention formats we consider to transfer weights from the CPU to the GPU are: 8-bit (1 bit for sign, 7 bits for exponent), 16-bit (1 bit for sign, 8 for exponent, 7 for mantissa), 24-bit (1 bit for sign, 8-bits for exponent and 15 bits for mantissa) and 32-bits (1 bit for sign, 8 bits for exponent and 23 bits for mantissa). We train the network models with dynamic data representation by applying the AWP algorithm along with the ADT procedure. We denote this approach combining ADT and AWP as A^2DTWP. For each DNN and batch size, we select the data representation format that first reaches the 35%, 25% and 15% accuracy thresholds for Resnet, Alexnet and VGG, respectively, and we denote this approach as *oracle*. For the case of the *oracle* approach, data compression is done via ADT. The closer A^2DTWP is to *oracle*, the better is the AWP algorithm in identifying the best data representation format.

During training we sample data in terms of elapse time and validation error every 4000 batches. The total number of training batches corresponding to the whole ImageNet200 dataset are 16020, 8010, 4005 and 2002 for batch sizes 16, 32, 64 and 128, respectively. The values of AWP parameters T, $INTERVAL$, and N are determined in the following way: In the case of T we monitor the execution of several epochs until we observe a drop in the validation error. We then measure the average change, considering all layers, of weights' l^2-norm during this short monitoring period. The obtained values of T are -5×10^{-2}, -2×10^{-3} and -2×10^{-5} for Alexnet, VGG and Resnet, respectively. We set the $INTERVAL$ parameter to 4000 for both AlexNet and VGG and 2000 for Resnet. These values correspond to a single batch (for the ImageNet200 dataset and batch sizes 64 and 128) and avoid premature precision switching due to numerical fluctuations. We set N to 8 since the smallest granularity of our approach is 1 byte. AWP initially applies 8-bit precision to all layers. We use ImageNet200 in Sections 5.5.2, 5.5.3, 5.5.4, 5.5.5, and 5.5.6. Section 5.5.7 uses ImageNet1000.

5.5.2 Evaluation on Alexnet

The evaluation considering the Alexnet model on the x86 system is shown in Figure 5.3, which plots detailed results considering batch sizes of 32 and 16, and Figure 5.5, which shows the total execution time of the *oracle* and A^2DTWP policies normalized to the *baseline* for the 64, 32 and 16 batch sizes on both the x86 and the POWER systems. The two top plots of Figure 5.3 depict how the validation error of the *baseline*, *oracle*, and A^2DTWP policies evolves over time for the 32 and the 16 batch

sizes until the 25% accuracy is reached. The two bottom plots provide information regarding the performance improvement of both *oracle* and *A²DTWP* over the 32-bit *baseline* with regard to a certain validation error. Such performance improvement is computed by looking at the time required by the *oracle* and *A²DTWP* techniques to reach a certain validation error with respect to the *baseline*.

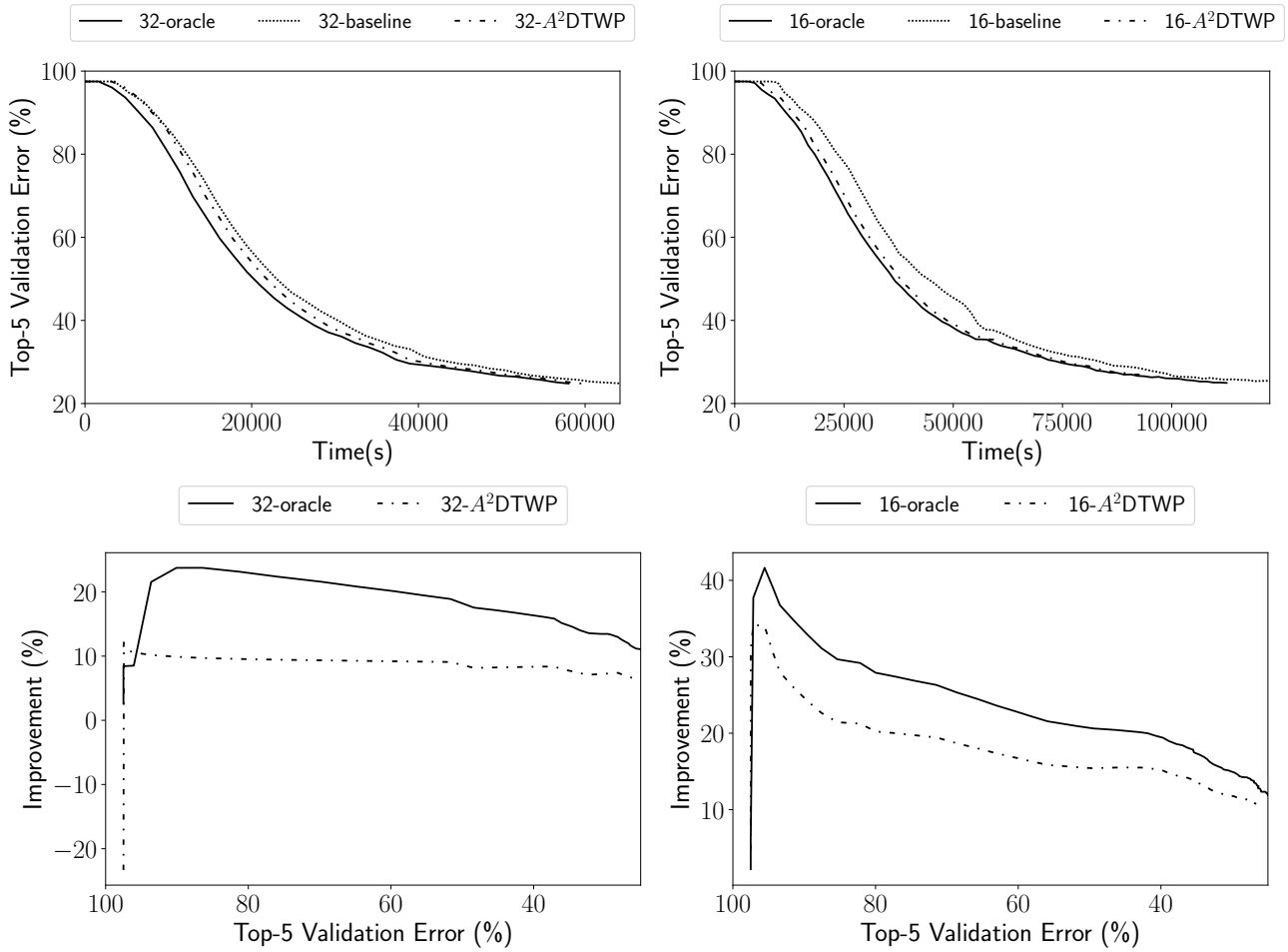

Fig. 5.3 Alex training considering 32 and 16 batch sizes. The two upper plots show the top-5 validation error evolution of *baseline*, *oracle* and *A²DTWP*. The two bottom figures provide information on the performance improvement of *oracle* and *A²DTWP* against *baseline* during the training process. Experiments run on the x86 system.

It can be observed in the upper left-hand side plot of Figure 5.3 how the *oracle* and the *A²DTWP* approaches are 10.82% and 6.61% faster than the baseline, respectively, to reach the 25% top-5 validation error when using a 32 batch size. The upper right-hand side plot shows results considering a 16 batch size. The improvements achieved by the *oracle* and *A²DTWP* approaches are 11.52% and 10.66%, respectively. This demonstrates the efficiency of the ADT procedure in compressing and decompressing the network weights without undermining the performance benefits obtained from

sending less data from the CPU device to the GPU. It also demonstrates the capacity of AWP to quickly identify the best data representation format per layer.

The two bottom plots of Figure 5.3 provide information on performance improvement of *oracle* and A^2DTWP over the *baseline* during the training process. For the 32 batch size, *oracle* reaches a peak improvement of 24.11% when the 90% validation error is reached and steadily declines from that point although it keeps a significant improvement of 10.82% over the *baseline* once the 25% top-5 validation error is reached. A^2DTWP falls in-between the *baseline* and the *oracle* and keeps its improvement above 7.03% until it reaches the 27% top-5 validation error. Once it reaches the 25% validation error A^2DTWP is 6.51% faster than the *baseline*. In conclusion, the A^2DTWP policy is able to provide performance improvements that are close to the ones achieved by the best possible accuracy. For the 16 batch size, the performance benefits of the *oracle* policy reach a 41.64% peak at the 94% validation error point. The A^2DTWP policy reaches its maximum performance benefit, 34.21%, when the validation error is 97%. At the 25% validation error point, the *oracle* and the A^2DTWP policies reach 13.00% and 10.75% performance improvement, respectively. Overall, results considering the Alexnet network for batch sizes 32 and 16 confirm that A^2DTWP, which combines both the AWP algorithm and the ADT procedure, successfully delivers very similar performance benefits to the best possible accuracy.

Figure 5.5 shows the normalized execution time of the *oracle* and A^2DTWP policies with respect to the 32-bit FP *baseline* on the x86 and the POWER systems. The top chart reports performance improvements of 10.75%, 6.51%, and 0.59% for batch sizes 16, 32 and 64 in the case of Alexnet runnig on the x86 system. For the 64 batch size, the marginal gains of A^2DTWP over the *baseline* are due the poor performance of the 8-bits format employed by A^2DTWP at the beginning of the training process. This format does not contribute to reduce the validation error for the 64 batch case, which makes the A^2DTWP policy to fall behind the *baseline* at the very beginning of the training process. Although A^2DTWP eventually increases its accuracy and surpases the *baseline*, it does not provide the same significant performance gains for Alexnet as the ones observed for batch sizes 16 and 32.

A^2DTWP performance improvements on the POWER system in the case of Alexnet are 18.61%, 14.25% and 10.01% with respect to the *baseline* for batch sizes 16, 32 and 64, respectively. The POWER system achieves larger performance improvements than x86 since the Bitpack procedure can be further parallelized over the 40 cores of the POWER9 multicore chips than the 16 cores available in the Haswell multicore devices of the x86 system. This mitigates the costs of weigths' compression and thus provides larger performance improvements.

5.5.3 Evaluation on VGG

Figure 5.4 shows results for batch sizes 64 and 32 when using the VGG architecture on the x86 system. The upper figures display the temporal evolution of the validation error until the 15% top-5 validation error is reached. Like in Alexnet, both the A^2DTWP and the *oracle* policies outperform the *baseline*. In the case of batch size 64, both *oracle* and A^2DTWP display a similar evolution in terms

of validation error, which translates to very close performance improvement over the baseline. They maintain an overall improvement of over 13.00% against the *baseline* during most of their training. The A^2DTWP technique outperforms the baseline by 12.88% when reaches 15% of top-5 validation error while the *oracle* policy achieves the same improvement. For batch size 32 the final improvement achieved by A^2DTWP over the baseline is 5.02%. This improvement is not as large as the one achived for the 64 batch size since the AWP algorithm does not identify a numerical precision able to beat the *baseline* until the 57% validation error is reached, as it can be seen in the bottom right hand side plot of Figure 5.4.

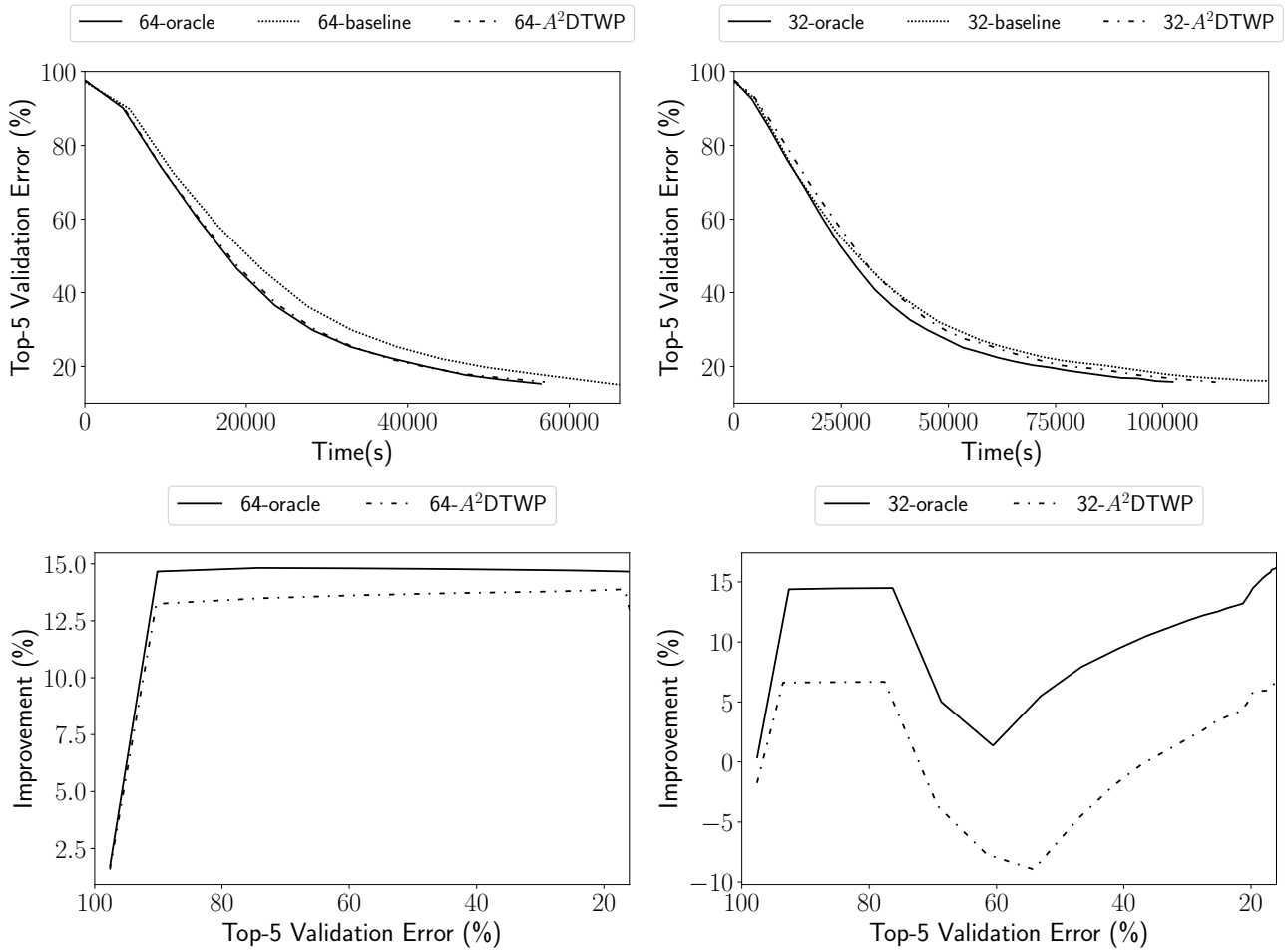

Fig. 5.4 VGG training considering 64 and 32 batch sizes. The two upper plots show the top-5 validation error evolution of *baseline*, *oracle* and A^2DTWP. The two bottom figures provide information on the performance improvement of *oracle* and A^2DTWP against *baseline* during the training process. Experiments run on the x86 system.

Figure 5.5 shows the normalized execution time of A^2DTWP and *oracle* with respect to the *baseline* for VGG considering batch sizes of 16, 32 and 64 on the x86 and POWER systems. When applied to the VGG model on the x86 system, A^2DTWP outperforms the 32-bit Floating Point *baseline*

by 12.88%, 5.02% and 7.31% for batch sizes 64, 32 and 16, respectively. Despite the already described issues suffered by the A^2DTWP technique when applied to the 32 batch size, this approach achieves very remarkable performance improvements over the baseline in all considered scenarios.

The performance improvements observed when trying VGG on the POWER system are even higher. A^2DTWP outperforms the *baseline* by 28.21%, 20.19% and 11.13% when using the 16, 32 and 64 batch sizes, respectively. The performance improvement achieved on the POWER system are larger than the ones observed for x86 since the Bitpack procedure can be parallelized over 40 cores when running on the POWER system. We observe the same behavior for Alexnet, as Section 5.5.2 indicates.

5.5.4 Evaluation on Resnet

We display the normalized execution time of the A^2DTWP and the *oracle* policies when applied to the Resnet model using batch sizes of 128, 64 and 32 in Figure 5.5. In the case of Resnet we do not show detailed plots describing the evolution of the validation error during training because its behavior is very close to some previously displayed scenarios like VGG. On the x86 system, A^2DTWP beats the 32-bit Floating Point baseline by 4.94%, 4.39% and 3.11% for batch sizes of 128, 64 and 32, respectively, once a top-5 validation error of 30% is reached. The relatively low performance improvement achieved in the case of 32 batch size is due to a late identification of a competitive numerical precision, as it happens in the case of VGG and batch size 32.

The performance gains on the POWER system display a similar trend as the ones achieved on x86. While they show the same low improvement for the 32 batch size, 2.12%, A^2DTWP achieves 6.92% and 11.54% performance gains for batch sizes 64 and 128, respectively. A^2DTWP achieves the largest performance improvement with respect to the 32-bit *baseline* when run on the POWER system due to the reasons described in Sections 5.5.2 and 5.5.3.

5.5.5 Average Performance Improvement

The average performance improvement of A^2DTWP over the *baseline* considering the Alexnet, VGG and Resnet models reach 6.18% and 11.91% on the x86 and the POWER systems, respectively. As we explain in previous sections, A^2DTWP obtains larger improvements on the POWER system than on x86 since the ADT procedure can be further parallelized over the 40 cores of the POWER9 multicore devices. In contrast, the two Haswell devices of the x86 system offer just 16 cores for ADT.

The combination of the AWP algorithm and the ADT procedure properly adapts the precision of each network layer and compresses the corresponding weigths with a minimal overhead. The large performance improvement obtained while training deep networks on two high-end computing systems demonstrate the effectiveness of A^2DTWP.

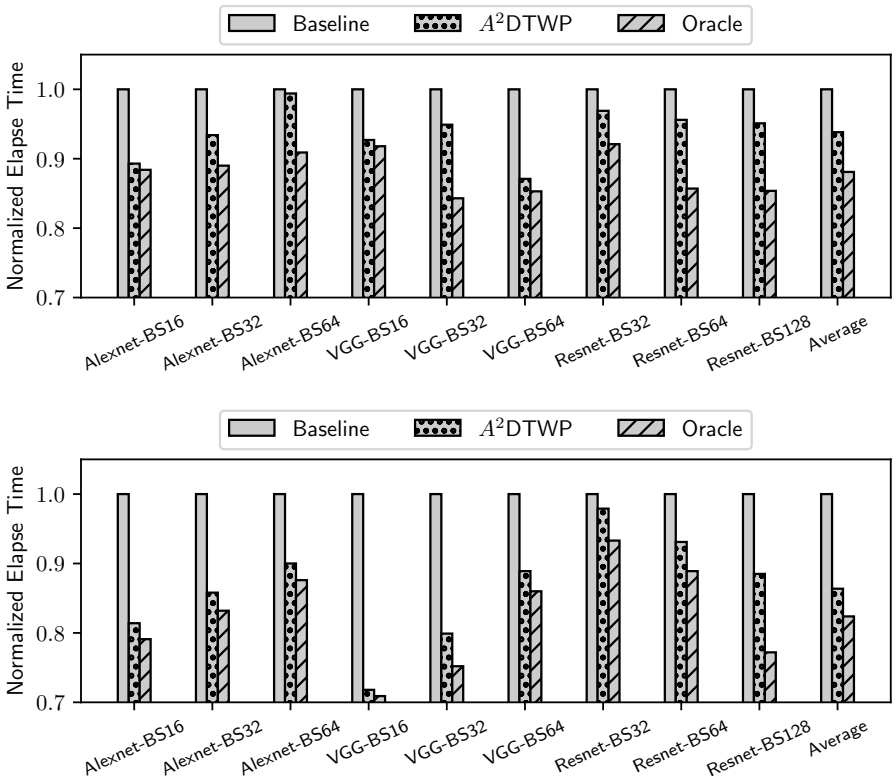

Fig. 5.5 Normalized execution times of the *A^2DTWP* and the *oracle* policies with respect to the baseline. Results obtained on the x86 system appear in the upper plot while the evalution on the POWER system appears at the bottom.

5.5.6 A^2DTWP Performance Profile

This section provides a detailed performance profile describing the effects of applying A^2DTWP when training the VGG network model with batch size 64 on the x86 and POWER systems described in section 5.4.4. To highlight these effects we also show a performance profile of applying 32-bit Floating Point format during training. The main kernels involved in the training process and their corresponding average execution time in milliseconds are shown in Tables 5.2 and 5.3. Each kernel can be invoked multiple times by different network layers and it can be overlapped with other operations while processing a batch. Tables 5.2 and 5.3 display for all kernels the average execution time of their occurrences within a batch when run on the x96 and the POWER systems, respectively.

Results appearing in Table 5.2 show how time spent transferring data from the CPU to the GPU accelerators when applying A^2DTWP on the x86 system, 52.27 ms, is significantly smaller than the cost of performing the same operation when using the 32-bit configuration, 153.93 ms. This constitutes a 2.94x execution time reduction that compensates the cost of the operations involved in the ADT routine, Bitpack and Bitunpack, and in the AWP algorithm, the l^2-norm computation. On POWER we observe a similar reduction of 3.20x in the time spent transferring data from the CPU to the GPUs when applying A^2DTWP. These reductions in terms of CPU to GPU data transfer time are due to a close to 3x reduction in terms of weights size enabled by A^2DTWP. The average execution time of operations where the A^2DTWP technique plays no role remains very similar for the 32-bit Floating Point baseline and A^2DTWP in both systems, as expected. Tables 5.2 and 5.3 indicate that performance gains achieved by A^2DTWP are due to data motion reductions, which validates the usefulness of A^2DTWP.

Tables 5.2 and 5.3 also display the overhead associated with AWP and ADT in terms of milliseconds. The AWP algorithm spends most of its runtime computing the l^2-norm of the weights, which takes a total of 3.88 ms within a batch on the x86 system. On POWER, the cost of computing the l^2-norm of the weights is 0.93 ms. The other operations carried out by AWP have a negligible overhead. The two fundamental procedures of ADT are the Bitpack and Bitunpack routines, which take 19.71 and 4.51 ms to run within a single batch on the x86 system. For the case of POWER, Bitpack and Bitunpack take 10.51 and 1.11 ms, respectively. Overall, measurements displayed at Table 5.2 indicate that AWP and ADT constitute 1.05% and 6.60% of the total batch execution time, respectively, on x86. On the POWER system, AWP and ADT constitute 0.54% and 6.82% of the total batch execution time according to Table 5.3. Figures 5.3, 5.4 and 5.5 account for this overhead in the results they display.

5.5.7 Experiments with ImageNet1000

We run experiments considering ImageNet1000 to confirm they display the same trends as executions with ImageNet200. Network parameters are the same as the ones described in Section 5.4. *AWP* parameters are the ones described in Section 5.5.1. The experimental setup of the evaluation considering ImageNet1000 is the same as the one we use for ImageNet200. We consider batch sizes that produce

Table 5.2 Performance profiles of both the A^2DTWP and the 32-bit Floating Point approaches expressed in milliseconds on the x86 system. We consider the VGG network model with batch size 64.

	32-bit FP	**A^{2}DTWP**
Data Transfer CPU→GPU	153.93	52.27
Data Transfer GPU→CPU	68.51	73.55
Convolution	128.72	126.13
Fully-connected	33.51	34.17
Gradient update	54.39	52.86
AWP (l^2-norm)	N/A	3.88
ADT (Bitpack)	N/A	19.71
ADT (Bitunpack)	N/A	4.51

Table 5.3 Performance profiles of both the A^2DTWP and the 32-bit Floating Point approaches expressed in milliseconds on the POWER system. We consider the VGG network model with batch size 64.

	32-bit FP	**A^{2}DTWP**
Data Transfer CPU→GPU	39.12	12.21
Data Transfer GPU→CPU	17.34	17.87
Convolution	69.78	71.21
Fully-connected	12.66	13.51
Gradient update	41.29	42.98
AWP (l^2-norm)	N/A	0.93
ADT (Bitpack)	N/A	10.51
ADT (Bitunpack)	N/A	1.11

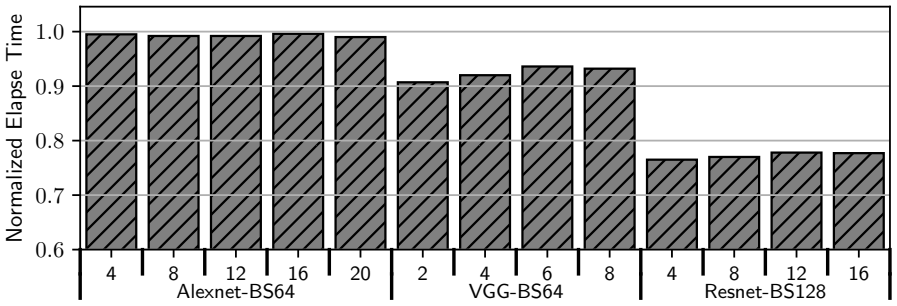

Fig. 5.6 Normalized execution time of A^2DTWP with respect to *baseline* considering the Imagenet1000 data set. Training for Alexnet, VGG and Resnet considers up to 20, 8, and 16 epochs, respectively.

the fastest 32-bit FP training for each one of the network models: 64, 64, and 128 for Alexnet, VGG and Resnet, respectively.

Figure 5.6 displays results corresponding to the experimental campaign with ImageNet1000 on the x86 system. In the x-axis we display different epoch counts for each one of the three models: 4, 8, 12, 16, and 20 epochs for Alexnet; 2, 4, 6, and 8 for VGG; and 4, 8, 12, and 16 epochs for Resnet. The y-axis displays the normalized elapsed time of A^2DTWP with respect to the the 32-bit Floating Point *baseline* per each model and epoch count. For the case of Alexnet with batch size 64, A^2DTWP is slightly faster than the *baseline* as it displays a normalized execution time of 0.995, 0.992, 0.992, 0.996, and 0.990 after 4, 8, 12, 16 and 20 epochs, respectively. Figure 5.5 also reports small gains for the case of Alexnet with batch size 64, which confirms that experiments with ImageNet1000 show very similar trends as the evaluation with ImageNet200. When applying A^2DTWP to VGG with 64 batch size, it displays a normalized execution time of 0.907, 0.920, 0.936, and 0.932 with respect to the *baseline* after running 2, 4, 6 and 8 training epochs, respectively. For the Resnet example, we observe normalized execution times of 0.765, 0.770, 0.778, and 0.777 for A^2DTWP after 4, 8, 12, and 16 training epochs, respectively, which constitutes a significant performance improvement.

In terms of validation error, both A^2DTWP and *baseline* display very similar top-5 values at the end of each epoch. For example, for the case of VGG, the Floating Point 32-bit *baseline* approach displays a validation error of 88.04% after 2 training epochs while A^2DTWP achieves a validation error of 89.97% for the same epoch count, that is, an absolute difference of 1.93%. After 4, 6, and 8 training epochs absolute distances of top-5 validation errors between A^2DTWP and *baseline* are 3.09%, 0.47%, and 0.71%, respectively. Top-5 validation error keeps decreasing in an analogous way for both *baseline* and A^2DTWP as training goes over more epochs, although A^2DTWP is significantly faster. Our evaluation indicates that A^2DTWP can effectively accelerate training while achieving the same validation error as the 32-bit FP *baseline* when considering ImageNet1000.

5.6 Conclusions

This chapter proposes A^2DTWP, which reduces data movement across heterogeneous environments composed of several GPUs and multicore CPU devices in the context of deep learning workloads. The A^2DTWP framework is composed of the AWP algorithm and the ADT procedure. AWP is able to dynamically define the weights data representation format during training. This chapter demonstrates that AWP is effective without any deterioration on the learning capacity of the neural network. To transform AWP decisions into real performance gains, we introduce the ADT procedure, which efficiently compresses network's weights before sending them to the GPUs. This procedure exploits both thread- and SIMD-level parallelism. By combining AWP with ADT we are able to achieve a significant performance gain when training network models such as Alexnet, VGG or Resnet. Our experimental campaign considers different batch sizes and two different multi-GPU high-end systems.

This chapter is the first in proposing a solution that relies on reduced numeric data formats to mitigate the cost of sending DNNs weights to different hardware devices during training. While our evaluation targets heterogeneous high-end systems composed of several GPUs and CPU multicore devices, techniques presented by this chapter are easily generalizable to any context involving several hardware accelerators exchanging large amounts of data. Taking into account the prevalence of deep learning-specific accelerators in large production systems [91], the contributions of this chapter are applicable to a wide range of scenarios involving different kinds of accelerators.

6

Communication Reduction in Model Parallelism of Deep Neural Networks

6.1 Introduction

Deep Neural Networks (MLPs, CNNs, RNNs etc.) have seen a mass adoption into the industry in recent years [86, 87, 88]. DNNs provide very competitive pattern detection capabilities and, more specifically, Convolutional Neural Networks (CNNs) classify very large image sets with remarkable accuracy [89].

As DNNs are gaining traction in more and more fields, the needs to accelerate the otherwise notoriously slow training has become a prominent topic in the HPC (high performance computing) community. (references) Furthermore, with the ever-increasing size of the datasets and the ever-growing complexity of the DNN architecture [18, 85, 109], nowadays it takes HPC clusters to train DNNs to reach a competitive accuracy [50]. A simple yet prevalent method to accelerate the training is to use *data parallelism* [49, 51] in which the input data are distributed onto various available computational units (CPUs, GPUs, FPGAs etc.) [96, 110, 111] and the training on different portion of the data are being carried out simultaneously. Nevertheless, it does not tackle the problem of the architectural complexity of the DNNs where the memory capacity of a computational unit is not sufficient to hold the parameters of the entire network. It is then natural to develop ways to distribute the network onto multiple computational units. *Model parallelism* is thus the parallelism paradigm to this end [50, 51].

Unlike *data parallelism* where the trainings on portions of data have no inter-dependencies, *model parallelism* inevitably introduces dependencies among the computational units. As a consequence, communication will have to occur so that each computational unit is updated with the contribution from the rest of the units. This impedes the network to scale on the current massively parallel systems with the message passing paradigm.

This chapter describes a novel approach *Altsplit* to accelerate the training of DNNs and improving the scalability of the current *model parallelism* approach by reducing the communication occurrences

during both the forward- and backward- propagation phases. It achieves so by alternating the splitting and the replication of the neurons in successive layers in a distributed-memory system. We compare this approach with a *baseline* approach, where the neurons of all the layers are split, on two HPC clusters with high-end CPUs (x86 Xeon and POWER9). Our experiments see an average performance benefits of 66.12% and 57.16% respectively on both clusters.

This chapter is structured as follows: Section 6.2 describes our *baseline* and *Altsplit* approaches. Section 6.3 provides information on the HPC clusters we run experiments on as well as the implementation details. We evaluate our approach in Section 6.4. Section 6.5 offers the conclusion to this chapter.

6.2 Communication Reduction in Model Parallelism of DNN

6.2.1 State-of-the-Art Approach

We consider model parallelism to accelerate the training of DNN on a distributed-memory system using MPI. Figure 6.1 illustrates the state-of-the-art approach to it which also serves as the baseline of this chapter. It depicts a 5-hidden-layer all-connected DNN split on two MPI processes and the 4 neurons per layer are evenly assigned to each MPI process. The input and output layer are omitted and we assume they are replicated in every MPI process. The black arrows in the figure indicate all-to-all communications between the two MPI processes. During both the forward- and backward-propagation, each neuron from the current layer needs the outputs of the entire neurons from the preceding layer. All-to-all communication must be performed across all the MPI processes. In the figure, prior to the computation of each layer 4 all-to-all communication need to occur so that the output information from the preceding layer can get to be fully propagated to the respective portions of the current layer to all the MPI processes. Albeit some numerical rounding errors introduced due to the non-deterministic nature of the all-to-all communication, this approach guarantees that the output information from the preceding layer is consistent and identical to all the MPI processes prior to the computation of their respective portions of the current layer.

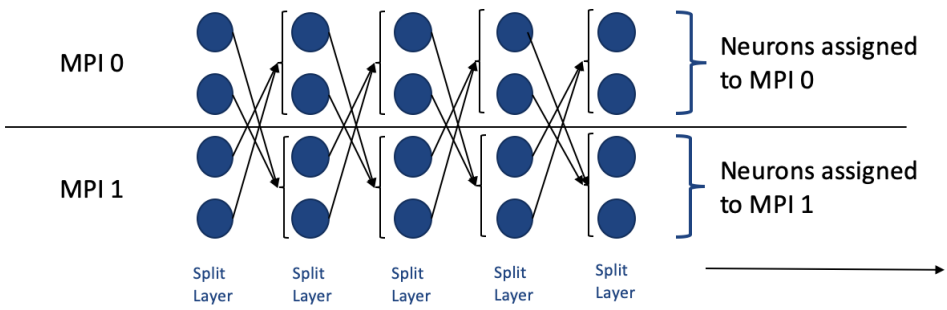

Fig. 6.1 State-of-the-art model parallelism scheme

We first show the sequential training of a DNN using matrix operations in Algorithm 10. We denote bs as the batch size, L as the number of layers and N_l as the number of neurons in layer l. Matrix of layer l is denoted as $\boldsymbol{A}_l[m,n]$ whereas m and n stand for the number of rows and columns respectively of the matrix and $(\boldsymbol{A}_l)^T[m,n]$ denotes the transpose of the matrix $\boldsymbol{A}_l$ and [m,n] represents the dimension of the transposed matrix. $\nabla\boldsymbol{A}_l[m,n]$ represents the gradients of matrix $\boldsymbol{A}_l$. ϕ is an element-wise non-linear function (tanh, relu etc.). Only the operations on the forward- and backward-propagation phases for the hidden layers are shown in detail since they are the regions of interest with regard to the subsequent parallel versions of the algorithm.

Algorithm 10 Sequential DNN

1: **for** $l = 1 \dots L$ **do** ▷ Forward-propagation
2: $\quad \boldsymbol{Y}_l[bs,N_l] = \boldsymbol{O}_{l-1}[bs,N_{l-1}] * (\boldsymbol{W}_l)^T[N_{l-1},N_l]$
3: $\quad \boldsymbol{O}_l[bs,N_l] = \phi(\boldsymbol{Y}[bs,N_l])$
4: **end for**
5: **for** $l = L \dots 1$ **do** ▷ Backpropagation
6: $\quad \nabla\boldsymbol{W}_l[bs,N_l] = \nabla\boldsymbol{W}_{l+1}[bs,N_{l+1}] * \boldsymbol{W}_{l+1}[N_{l+1},N_l]$
7: $\quad \nabla\boldsymbol{W}_l[bs,N_l] = \nabla\boldsymbol{W}_l[bs,N_l] * (\partial\boldsymbol{O}_l[bs,N_l]/\partial\boldsymbol{Y}_l[bs,N_l])$
8: **end for**
9: Update parameters

The state-of-the-art model parallelism of a DNN is shown in Algorithm 11. Besides the notations we introduced in Algorithm 10, we give some additional notations due to the introduction of parallelism. *size* denotes the number of MPI processes, *rank* denotes the ID of the current MPI process and N_l represents the number of neurons assigned to each MPI process which is equivalent to $N_{l\_total}/size$ (here we assume that $N_{l\_total}/size$ is divisible). This algorithm allocates some extra space for holding the information gathered across all the MPI threads:

- The weight matrix of each hidden layer $\boldsymbol{W}_l$ should be of dimension $[N_l, N_{l-1} * size]$.
- An extra matrix to store the outputs from the preceding hidden layer from all the MPI threads $\boldsymbol{O}_{l-1}[bs, N_{l-1} * size]$.
- An extra matrix to store the gradients of the succeeding layer from all the MPI threads $\nabla\boldsymbol{W}_l[bs, N_l * size]$.

At the beginning of the forward-propagation phase of each hidden layer an *MPI_Allgather* precedes the computation to gather outputs from all local portions from the preceding layer. Subsequently, each MPI thread needs to perform a MPI_Allreduce with the *sum* operation on $\nabla\boldsymbol{W}_l[bs, N_l * size]$ and extract its respective gradients from it during the backpropagation phase.

Algorithm 11 State-of-the-art approach to model parallelism of DNN

for all $p \in MPI\_Processes$ **do** ▷ Forward-propagation
  for $l = 1 \dots L$ **do**
    MPI_Allgather on $\boldsymbol{O}_{l-1}[bs, N_{l-1} * size]$ from all local $\boldsymbol{O}_{l-1}[bs, N_{l-1}]$
    $\boldsymbol{Y}_l[bs, N_l] = \boldsymbol{O}_{l-1}[bs, N_{l-1} * size] * (\boldsymbol{W}_l)^T[N_{l-1} * size, N_l]$
    $\boldsymbol{O}_l[bs, N_l] = \phi(\boldsymbol{Y}[bs, N_l])$
  end for
end for
for all $p \in MPI\_Processes$ **do** ▷ Backpropagation
  for $l = L \dots 1$ **do**
    $\nabla \boldsymbol{W}_l[bs, N_l * size] = \nabla \boldsymbol{W}_{l+1}[bs, N_{l+1}] * \boldsymbol{W}_{l+1}[N_{l+1}, N_l * size]$
    MPI_Allreduce_sum on $\nabla \boldsymbol{W}_l[bs, N_l * size]$
    Extract $\nabla \boldsymbol{W}_{l,p}[bs, N_l]$ from $\nabla \boldsymbol{W}_l[bs, N_l * size]$ according to *rank*
    $\nabla \boldsymbol{W}_{l,p}[bs, N_l] = \nabla \boldsymbol{W}_{l,p}[bs, N_l] * (\partial \boldsymbol{O}_{l,p}[bs, N_l] / \partial \boldsymbol{Y}_{l,p}[bs, N_l])$
  end for
end for
for all $p \in MPI\_Processes$ **do**
  Update parameters
end for

6.2.2 The Altsplit (Alternate Split) Approach

We propose the *Altsplit* approach which splits or replicates the layers alternately. The scheme is illustrated in Figure 6.2 with the same 5-hidden-layer DNN as in Figure 6.1. The first hidden layer is split across MPI processes whereas the next layer is replicated on the MPI processes with the same initialization values. Subsequent layers are constructed with alternating splits and replications. Therefore, we cut the amount of communication by half during the entire training compared to the *state-of-the-art* approach by triggering communication every other layer while at the cost of replicating layers on each MPI process. As a consequence, the floating point computation is increased by 25% (twice every other layer).

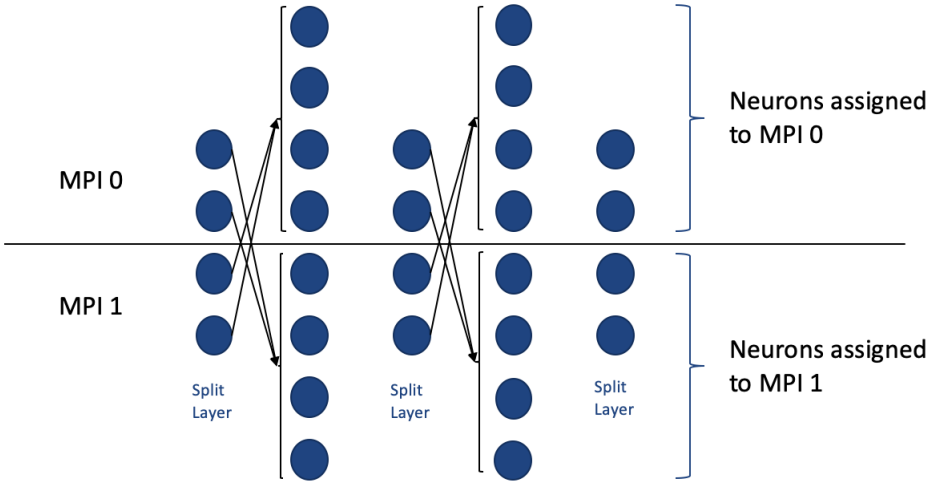

Fig. 6.2 The *Altsplit* scheme

Algorithm 12 illustrates the *Altsplit* approach to model parallelism. Unlike the state-of-the-art approach, there is no need for extra storage in *Altsplit*. If the preceding layer during the forward-propagation phase is a split, an MPI_Allreduce with the *sum* operation must be performed on $\boldsymbol{Y}_{l-1}[bs, N_l]$. Similarly, the same routine must be called upon on $\nabla \boldsymbol{W}_l[bs, N_l]$ while in the backpropagation phase if the succeeding layer is a split.

Algorithm 12 *Altsplit* approach to model parallelism of DNN

for all $p \in MPI\_Processes$ **do** ▷ Forward-propagation
  for $l = 1 \ldots L$ **do**
    $\boldsymbol{Y}_l[bs, N_l] = \boldsymbol{O}_{l-1}[bs, N_{l-1}] * (\boldsymbol{W}_l)^T[N_{l-1}, N_l]$
    if $l - 1 == SPLIT$ **then**
      MPI_Allreduce_sum on $\boldsymbol{Y}_{l-1}[bs, N_l]$
    end if
    $\boldsymbol{O}_l[bs, N_l] = \phi(\boldsymbol{Y}[bs, N_l])$
  end for
end for
for all $p \in MPI\_Processes$ **do** ▷ Backpropagation
  for $l = L \ldots 1$ **do**
    $\nabla \boldsymbol{W}_l[bs, N_l] = \nabla \boldsymbol{W}_{l+1}[bs, N_{l+1}] * \boldsymbol{W}_{l+1}[N_{l+1}, N_l]$
    if $l + 1 == SPLIT$ **then**
      MPI_Allreduce_sum on $\nabla \boldsymbol{W}_l[bs, N_l]$
    end if
    $\nabla \boldsymbol{W}_{l,p}[bs, N_l] = \nabla \boldsymbol{W}_{l,p}[bs, N_l] * (\partial \boldsymbol{O}_{l,p}[bs, N_l] / \partial \boldsymbol{Y}_{l,p}[bs, N_l])$
  end for
end for
for all $p \in MPI\_Processes$ **do**
  Update parameters
end for

6.3 Experimental Setup

6.3.1 Hardware Platforms

We conduct our experiments on two clusters featuring the x86 and POWER architectures. The x86 machine is composed of two 24-core Intel Xeon ®E5-2630 v3 (Haswell) at 2.4 GHz and a 20 MB L3 shared cache memory each. It is also equipped with two Nvidia Tesla K80 accelerators, each of which hosts two Tesla GK210 GPUs. It has 128 GB of main memory, distributed in 8 DIMMs of 16 GB DDR4 @ 2133 MHz. The 16-core CPU and the four GPUs are connected via a PCIe 3.0 x8 8GT/s. The operating system is RedHat Linux 6.7. Overall, the peak performance of the two 8-core sockets plus the four Tesla GK210 GPUs is 6.44 TFlop/s.

The POWER machine is composed of two 20-core IBM POWER9 8335-GTG at 3.00 GHz. It contains four NVIDIA Volta V100 GPUs. Each node has 512 GB of main memory, distributed in 16 DIMMS of 32 GB @ 2666 MHz. The GPUs are connected to the CPU devices via a NVIDIA NVLink 2.0 interconnection [9]. The operating system is RedHat Linux 7.4. The peak performance of the two 20-core sockets plus the four V100 GPUs is 28.85 TFlop/s.

6.3.2 Implementation

We build the baseline and our approach on top of KANN [57] which is a deep learning framework written in C/C++. Section 3.3.2 provides its information in detail.

It builds a computational graph prior to carrying out the actual computations and the operation and the computation of its derivative are performed in the same node in the graph. Furthermore, we observe that all the MPI calls in both approaches are performed either right before or after a matrix-matrix multiplication. We thus insert appropriate MPI calls inside the computational nodes that are responsible for carrying out the multiplication and its derivative computation according to their relative order to the multiplications.

Due to the fact that MPI all-to-all communication possess non-deterministic behavior since the order the messages arriving to each MPI process may vary, the two approaches may show minor differences in the accuracy. Apart from that we make sure that the initial values of each layer are identical.

We use OpenMPI [112, 113] as our MPI implementation. The version we use on the x86 machine is 1.10.0 and the version on the POWER machine is 3.0.0.

6.4 Evalutation

We conduct extensive experiments on various aspects of *Altsplit*. We demonstrate its scalability in Section 6.4.1 and show that it is applicable in different machines by providing results on two HPC clusters in Section 6.4.2. In Section 6.4.3 we visualize the *Altsplit* and the *baseline* approach to get more insights.

6.4.1 Parallelism Scalability

We run *Altsplit* and *baseline* side-by-side on the x86 clusters up to 16 nodes (640 MPI threads). We use a total of 4 configurations of MLP networks:

- 16,000-neuron-per-layer, 3-layer, batch size of 512, denoted as 16000.3.512
- 16,000-neuron-per-layer, 3-layer, batch size of 1024, denoted as 16000.3.1024
- 16,000-neuron-per-layer, 5-layer, batch size of 512, denoted as 16000.5.512
- 16,000-neuron-per-layer, 5-layer, batch size of 1024, denoted as 16000.5.1024

We measure the elapse time of runs of 50 batches (51 batches but excluding the result from the first batch to minimize system noises). We execute them on the dataset from Cifar-10 [59]. We use 40 MPI threads per node which are mapped to 40 distinct physical cores. Hence, the measurements are taken in a stride of 40 MPI threads.

Figure 6.3 shows the results in 4 plots. Each plot depicts the performance of *Altsplit* against *baseline* in terms of elapse time normalized with regard to *baseline* with varied number of MPI threads, from 80 MPI threads (2 nodes) all the way up to 640 MPI threads (16 nodes).

We can see from the top left plot (80 MPI threads) that *Altsplit* goes slower than *baseline*. Nevertheless, starting from 160 MPI threads (top right) and beyond *Altsplit* begins to gain track and consistently outperforms *baseline*. More specifically, *Altsplit* runs 18.3%, 39.32% and 66.11% faster than their respective *baseline* in average.

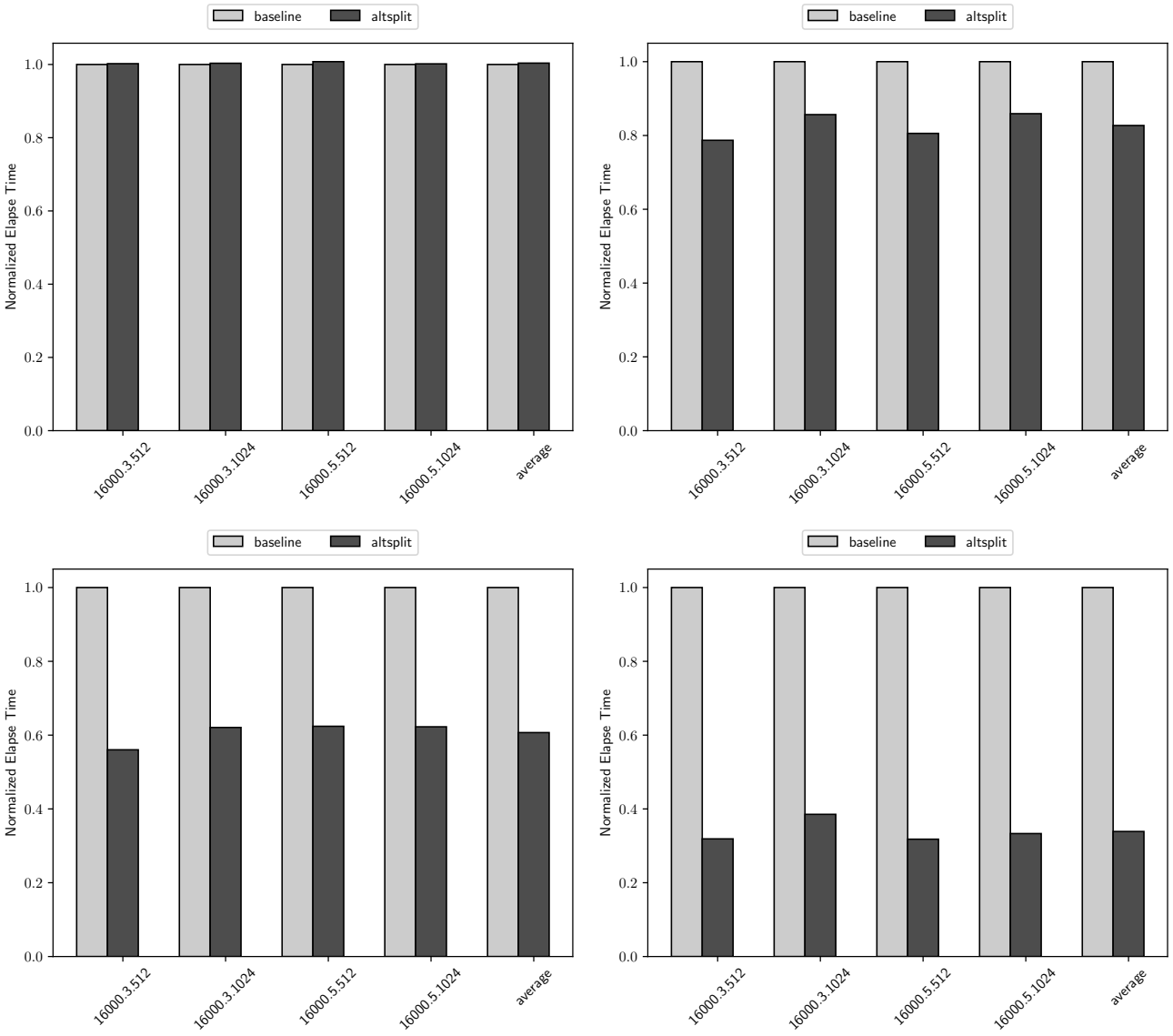

Fig. 6.3 Performance improvements of 16k neurons on x86. Top left: 80 MPI processes, Top right: 160 MPI processes, Bottom left: 320 MPI processes, Bottom right: 640 MPI processes

6.4.2 Network Versatility

We launch a more extensive set of experiment that covers more network configurations. We use two neurons-per-layer numbers: 16k and 32k, two layers: 3 and 5 along with two batch sizes: 512 and 1024 on both machines. We find out that *Altsplit* achieves the greatest performance gain on 640 MPI threads (16 nodes on both machines). Table 6.1 illustrates the results we obtain from of all the configurations.

We observe that *Altsplit* with 16k neurons-per-layer outperforms *baseline* over 66.12% and 55.47% on x86 and POWER9 respectively whereas with 32k neurons-per-layer the figures are 41.10% and

32.64% respectively. A low neuron-per-layer count attributes to a better performance with an average gain in average of 25.02% and 24.51% respectively on x86 and POWER9. Since we are effectively trading communication with additional computation with the *Altsplit* approach. In general *Altsplit* also performs better under smaller batch sizes. This indicates that there is a sweet spot where the benefit of reducing the communication is maximized.

Table 6.1 Performance improvements over the *baseline* on 640 MPI threads

Parameters			Machine	
Neurons	Layers	BS	**x86**	**POWER9**
16k	3	512	68.12%	58.65%
		1024	61.47%	56.12%
	5	512	68.21%	58.43%
		1024	66.68%	55.47%
Average			66.12%	57.16%
32k	3	512	45.28%	32.75%
		1024	37.69%	31.58%
	5	512	43.87%	34.37%
		1024	37.58%	31.91%
Average			41.10%	32.65%

6.4.3 Traces

We run both the *Altsplit* and *baseline* on the x86 cluster with 80 MPI threads and generate running traces. Figure 6.4 illustrates a run with 10 batches (9 are actually shown to exclude the system noises introduced by the first batch). The x-axis represents the elapse time whereas each row from the y-axis is the chronological activity from one of the 80 MPI threads. MPI calls are displayed as the short burst of colored lines in the traces and the blank areas represents computation or other system activities. Their length are proportional to the total elapse time. Two types of MPI all-to-all communication involved in the *baseline* implementation, namely, MPI_Allreduce and MPI_Allgather which corresponds with the trace on the top. Those marked with red are calls from MPI_Allgather while the pink ones are calls from MPI_Allreduce. The duration of the trace of the *Altsplit* is the same as in the *baseline*. One batch in the trace of the *baseline* is marked with two consecutive MPI_Allgather (red) calls followed with two MPI_Allreduce (pink) ones. On the other hand, in the trace of *Altsplit* one batch is marked with two MPI_Allreduce calls with a blank area sparsely dotted with small bursts of MPI calls. The dark blue activities displayed at the end of both traces are MPI_Finalize calls to mark the end of the MPI execution.

We can see that despite the similar duration of MPI_Allreduce calls on both approaches and the fact that it takes a longer time for the *Altsplit* to commence the subsequent batch, the additional

MPI_Allgather incur a significant slow down compared to the blank area in-between the successive MPI_Allreduce calls from the *Altsplit* approach.

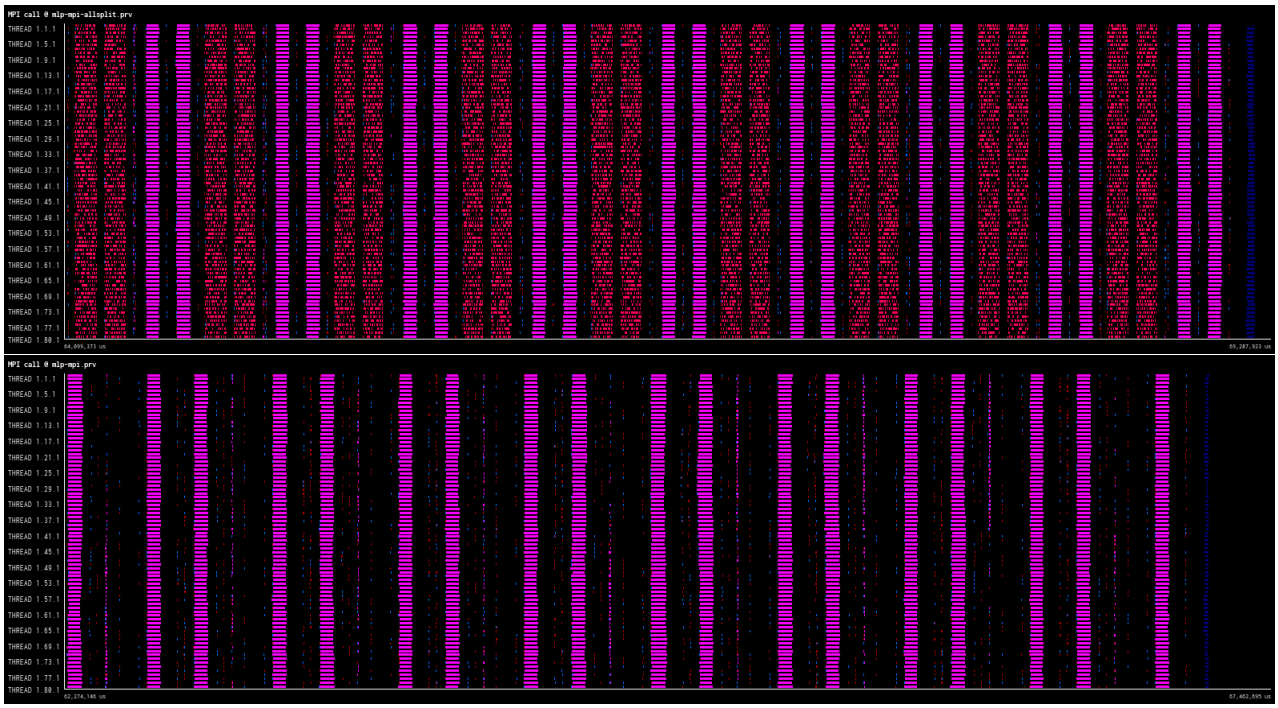

Fig. 6.4 Traces on 80 MPI processes 9 batches. Top: *baseline* approach Bottom: *altsplit* approach

6.5 Conclusions

This chapter presents the *Altsplit* approach that intends to provide a model parallelism of the DNN with less communication. It achieves this by distributing the neurons and replicating them across all the computational units alternately in-between successive layers so that all-to-all communication only occurs every other layers instead of in a lock-step fashion in an *state-of-the-art* approach. Our experiments are conducted taking multiple hyper-parameters into consideration: the number of neurons per layer, the number of layers and batch sizes. We conduct the experiments on two high-end multicore multinode clusters with distinct CPU architecture.

We find that the *Altsplit* approach achieves significant speedups over our baseline *state-of-the-art* approach regardless the underlying cluster. Furthermore, we show that the speedup is retained among various hyper-parameters and we visualize the speedup by generating traces on both approach and shows that the trade-off between additional floating-point computations and reduced communication pays off.

7

Conclusions

This thesis intends to alleviate the bottlenecks brought about by all-to-all communication in the modern HPC systems due to the constant scaling-up of the system size, problem size and the appearances of emerging fields. This proves to be a daunting task because there is not an one-size-fits-all approach to it. Each field possesses its own requirements and patterns on communication. Furthermore, the trade-offs applicable differ from field to field.

This thesis utilizes three trade-off techniques in concrete for the communication reduction purpose.

- Exploiting the resilience towards accumulation of rounding errors and loss of precision of the problem at hand to reduce communication.
- Trading with a decreased computational precision with a marginal deterioration of the accuracy of the problem for reduced amount of communication.
- Trading at the cost of additional computation with reduce amount of communication.

The thesis first takes on reducing communication in one of the Krylov iterative methods, CG. Compare to the various available direct methods the CG method displays greater tolerance towards the accumulation of rounding errors. We thus fuse a certain amount of iterations together which means some efforts to bring down the accumulated error such as the residual replacement strategy has to be deferred to the end of the fused phase. Nevertheless, we show that further incorporating a task-based parallelism approach, significant speedup can be achieved without much hampering to the convergence of the algorithm.

On tackling the problem of accelerating the DNN training with multiple GPUs, we exploited the fact that DNNs are intrinsically tolerant to a lowered precision of both its parameters. Guided by the L2-norm of the weights of each layer, we start by transferring compressed weights with low precision and dynamically increment the precision in a per-layer granularity if needed. Our experiments confirms that this approach greatly reduces the amount of data needed to be transferred from the CPU host to GPUs prior to the beginning of each batch and consequently outperforms the training with full 32-bit floating point weights in terms of the training time.

In order to remove some amount of communication from applying model parallelism to DNNs, we try to replicate every other layers rather than splitting each and every layer so that communication only occurs once every two layers. Despite the higher count of computation inevitably introduced by the approach, our experiments indicate that it is a trade-off well justified. It outperforms our baseline approach by a large margin on two high-end clusters with completely distinct CPU architectures.

7.1 Further Down The Road

With the first exascale clusters on the horizon, the pursuit of more accurate models in computational sciences and the awe-inspiring speed deep learning and data science are taking over multiple industries, the potential benefit for a communication-reduction solution to a particular problem will not diminish any time soon. The techniques this thesis explores all come off as trade-offs of various sorts since communication is closely entangled with other aspects of the algorithm: computation, space, precision etc. A deeper understanding into their synergy on an per-algorithm basis is of great importance. Furthermore, we need some level of generalization to derive techniques that are applicable to multiple fields.

Appendix A

Publications

A.1 Publications Related With The Thesis

- Sicong Zhuang and Marc Casas. Iteration-fusing conjugate gradient. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 21:1–21:10, New York, NY, USA, 2017. ACM
- Sicong Zhuang, Cristiano Malossi and Marc Casas. Reducing Data Motion to Accelerate the Training of Deep Neural Networks (under review)
- Sicong Zhuang, Panagiotis Hadjidoukas, Cristiano Malossi and Marc Casas. Altsplit: Communication Reduction In DNN Model Parallelism (future submission)

A.2 Other Publications

- Ilia Pietri, Sicong Zhuang, Marc Casas, Miquel Moretó, and Rizos Sakellariou. Evaluating scientific workflow execution on an asymmetric multicore processor. In *Euro-Par 2017: Parallel Processing Workshops*, pages 439–451, Cham, 2018. Springer International Publishing

Bibliography

[1] W. Gropp. Update on libraries for blue waters. http://jointlab-pc.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf, 2010.

[2] J. A. Kahle, J. Moreno, and D. Dreps. 2.1 summit and sierra: Designing ai/hpc supercomputers. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 42–43, Feb 2019.

[3] Christopher Zimmer, Don Maxwell, Stephen McNally, Scott Atchley, and Sudharshan S. Vazhkudai. Gpu age-aware scheduling to improve the reliability of leadership jobs on titan. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 7:1–7:11, Piscataway, NJ, USA, 2018. IEEE Press.

[4] Marenostrum 4 technical information. https://www.bsc.es/marenostrum/marenostrum/technical-information, 2019.

[5] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. Communication-avoiding parallel strassen: implementation and performance. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 101, 2012.

[6] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[7] Edgar Solomonik, Grey Ballard, James Demmel, and Torsten Hoefler. A communication-avoiding parallel algorithm for the symmetric eigenvalue problem. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 111–121, 2017.

[8] Edgar Solomonik, Grey Ballard, James Demmel, and Torsten Hoefler. A communication-avoiding parallel algorithm for the symmetric eigenvalue problem. *CoRR*, abs/1604.03703, 2016.

[9] Nvidia Corp. Nvlink fabric. https://www.nvidia.com/en-us/data-center/nvlink/, 2016.

[10] J. Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 585–585, May 2013.

[11] Edgar Solomonik, Grey Ballard, James Demmel, and Torsten Hoefler. A communication-avoiding parallel algorithm for the symmetric eigenvalue problem. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 111–121, New York, NY, USA, 2017. ACM.

[12] Yang You, Zhao Zhang, Cho-Jui Hsieh, and James Demmel. 100-epoch imagenet training with alexnet in 24 minutes. *CoRR*, abs/1709.05011, 2017.

[13] Yang You, James Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. Ca-svm: Communication-avoiding support vector machines on distributed systems. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 847–859, Washington, DC, USA, 2015. IEEE Computer Society.

[14] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1737–1746, 2015.

[15] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J Pai, and Naveen Rao. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1742–1752. Curran Associates, Inc., 2017.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[20] Exascale Mathematics Working Group(EMWG). Applied mathematics research for exascale computing. Technical report, US Department of Energy, 2012.

[21] Software design for multi-core multiprocessor architectures. https://developer.ibm.com/articles/au-aix-multicore-multiprocessor, 2013.

[22] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[23] *Application Program Interface*. OpenMP Architecture Review Board, 2013.

[24] Llnl openmp tutorial. https://computing.llnl.gov/tutorials/openMP, 2019.

[25] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[26] Thread building blocks (tbb). https://www.threadingbuildingblocks.org, 2019.

[27] A.Duran, E.Ayguade, R.M.Badia, J.Labarta, L.Martinell, X.Martorell, and J.Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2), 2011.

[28] Llnl mpi tutorial. https://computing.llnl.gov/tutorials/mpi, 2019.

[29] Mpi. https://en.wikipedia.org/wiki/Message_Passing_Interface, 2019.

[30] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[31] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

[32] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[33] Misha Kilmer and G. W. Stewart. Iterative regularization and minres. *SIAM J. Matrix Anal. Appl.*, 21(2):613–628, October 1999.

[34] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[35] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. Alistair Watson, editor, *Numerical Analysis*, pages 73–89, Berlin, Heidelberg, 1976. Springer Berlin Heidelberg.

[36] Siegfried Cools, Jeffrey Cornelis, Pieter Ghysels, and Wim Vanroose. Improving strong scaling of the conjugate gradient method for solving large linear systems using global reduction pipelining. *CoRR*, abs/1905.06850, 2019.

[37] Jeffrey Cornelis, Siegfried Cools, and Wim Vanroose. The communication-hiding conjugate gradient method with deep pipelines. *CoRR*, abs/1801.04728, 2018.

[38] P.Ghysels and W.Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40, 2014.

[39] L. Grigori, S. Moufawad, and F. Nataf. Enlarged krylov subspace conjugate gradient methods for reducing communication. *SIAM Journal on Matrix Analysis and Applications*, 37(2):744–773, 2016.

[40] Paul R. Eller and William Gropp. Scalable non-blocking preconditioned conjugate gradient methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 18:1–18:12, Piscataway, NJ, USA, 2016. IEEE Press.

[41] Mark Hoemmen. *Communication-avoiding Krylov Subspace Methods*. PhD thesis, Berkeley, CA, USA, 2010. AAI3413388.

[42] Joe Yue-Hei Ng, Matthew J. Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. *CoRR*, abs/1503.08909, 2015.

[43] Jason Lee, Kyunghyun Cho, and Thomas Hofmann. Fully character-level neural machine translation without explicit segmentation. *CoRR*, abs/1610.03017, 2016.

[44] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv e-prints*, page arXiv:1409.0473, Sep 2014.

[45] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[46] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *CoRR*, abs/1611.01576, 2016.

[47] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.

[48] Ahmed M. Elgammal, Bingchen Liu, Mohamed Elhoseiny, and Marian Mazzone. CAN: creative adversarial networks, generating "art" by learning about styles and deviating from style norms. *CoRR*, abs/1706.07068, 2017.

[49] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, Berkeley, CA, USA, 2014. USENIX Association.

[50] Amir Gholami, Ariful Azad, Kurt Keutzer, and Aydin Buluç. Integrated model and data parallelism in training neural networks. *CoRR*, abs/1712.04432, 2017.

[51] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *CoRR*, abs/1802.09941, 2018.

[52] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Implementing ompss support for regions of data in architectures with multiple address spaces. ICS '13.

[53] François Chollet et al. Keras. https://keras.io, 2015.

[54] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[55] Martín Abadi, Paul Barham, Jianmin Chen, et al. 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.

[56] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[57] Kann: A lightweight c library for artificial neural networks. https://github.com/attractivechaos/kann, 2019.

[58] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), 2011.

[59] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[60] John Van Rosendale. Minimizing inner product data dependencies in conjugate gradient iteration. *ICASE-NASA 172178*, 1983.

[61] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[62] A. T. Chronopoulos and C. W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2), 1989.

[63] Yousef Saad. Practical use of some krylov subspace methods for solving indefinite and nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5(1), 1984.

[64] Gérard Meurant. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Computing*, 5(3), 1987.

[65] Eduardo D'Azevedo, Victor Eijkhout, and Charles Romine. Lapack working note 56: Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors. Technical report, Knoxville, TN, USA, 1993.

[66] Laurence Tianruo Yang and Richard P. Brent. The improved bicg method for large and sparse linear systems on parallel distributed memory architectures. PDSECA '02.

[67] Laurence Tianruo Yang and Richard P. Brent. The improved bicgstab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures. ICA3PP '02.

[68] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the gmres algorithm on massively parallel machines. *SIAM Journal on Sci. Computing*, 35(1), 2013.

[69] A. T. Chronopoulos. s-step iterative methods for (non)symmetric (in)definite linear systems. *SIAM Journal on Numerical Analysis*, 28(6), 1991.

[70] E. de Sturler and H. A. van der Vorst. Reducing the effect of global communication in gmres(m) and cg on parallel distributed memory computers. *Appl. Numer. Math.*, 18(4), 1995.

[71] Z. Bai, D. Hu, and L. Reichel. A newton basis gmres implementation. *IMA Journal of Numerical Analysis*, 14(4), 1994.

[72] Wayne Joubert and Graham F. Carey. Parallelizable restarted iterative methods for nonsymmetric linear systems. SIAM PP '91.

[73] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, University of California, 2010.

[74] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel numerical linear algebra. *Acta Numerica*, 2, 1993.

[75] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. PACT '08.

[76] Erin Carson and James Demmel. A residual replacement strategy for improving the maximum attainable accuracy of s-step krylov subspace methods. Technical report, University of California, Berkeley, 2012.

[77] Henk A. Van Der Vorst and Qiang Ye. Residual replacement strategies for krylov subspace iterative methods for the convergence of true residuals. Technical report, 1999.

[78] Erin Carson and James Demmel. A residual replacement strategy for improving the maximum attainable accuracy of *s*-step krylov subspace methods. *SIAM Journal on Matrix Analysis and Applications*, 35(1), 2014.

[79] Siegfried Cools, Wim Vanroose, Emrullah Fatih Yetkin, Emmanuel Agullo, and Luc Giraud. On rounding error resilience, maximal attainable accuracy and parallel performance of the pipelined conjugate gradients method for large-scale linear systems in petsc. EASC '16.

[80] BSC. Programming models group. the nanos++ parallel runtime. https://pm.bsc.es/nanox, 2015.

[81] *Intel Math Kernel Library Reference Manual*. Intel Corporation, 2009.

[82] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. A quantitative analysis of os noise. IPDPS '11, May 2011.

[83] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. SC '10.

[84] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. SC '08.

[85] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[86] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[87] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[88] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, June 2012.

[89] 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.

[90] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[91] Norman P. Jouppi, Cliff Young, Nishant Patil, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[92] Thorsten Kurth, Jian Zhang, Nadathur Satish, et al. Deep learning at 15pf: Supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 7:1–7:11, New York, NY, USA, 2017. ACM.

[93] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[94] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1988.

[95] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23:462–466, 1952.

[96] Yang You, Aydin Buluc, and James Demmel. Scaling deep learning on gpu and knights landing clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 9:1–9:12, New York, NY, USA, 2017. ACM.

[97] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, pages 161–168, 2008.

[98] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *Seventh International Conference on Learning Representations (ICLR)*, 2018.

[99] A. F. Murray and P. J. Edwards. Enhanced mlp performance and fault tolerance resulting from synaptic weight noise during training. *IEEE Transactions on Neural Networks*, 5(5):792–802, Sep 1994.

[100] Chris M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Comput.*, 7(1):108–116, January 1995.

[101] K. Audhkhasi, O. Osoba, and B. Kosko. Noise benefits in backpropagation and deep bidirectional pre-training. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Aug 2013.

[102] Chris Lomont. Introduction to intel advanced vector extensions. intel white paper, 2011.

[103] Linley Gwennap. AltiVec Vectorizes PowerPC. *Microprocessors Report*, 12(6):1–5, May 1998.

[104] Ieee standard for floating point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[105] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[106] Hwajeong Seo, Zhe Liu, Johann Großschädl, and Howon Kim. Efficient arithmetic on arm-neon and its application for high-speed rsa implementation. *IACR Cryptology ePrint Archive*, 2015:465, 2015.

[107] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[108] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[109] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019.

[110] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. A survey of FPGA based deep learning accelerators: Challenges and opportunities. *CoRR*, abs/1901.04988, 2019.

[111] Norman P. Jouppi, Cliff Young, Nishant Patil, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[112] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open mpi: A flexible high performance mpi. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[113] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, Sep. 2006.

[114] Sicong Zhuang and Marc Casas. Iteration-fusing conjugate gradient. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 21:1–21:10, New York, NY, USA, 2017. ACM.

[115] Ilia Pietri, Sicong Zhuang, Marc Casas, Miquel Moretó, and Rizos Sakellariou. Evaluating scientific workflow execution on an asymmetric multicore processor. In *Euro-Par 2017: Parallel Processing Workshops*, pages 439–451, Cham, 2018. Springer International Publishing.

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface.
ASIC	Application Specific Integrated Circuit.
BiCG	Biconjugate Gradient Method.
CG	Conjugate Gradient Method.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DNN	Deep Neural Network.
FPGA	Field Programmable Gate Array.
GMRES	Generalized Minimal Residual Method.
GPU	Graphics Processing Unit.
HPC	High Performance Computing.
MINRES	Minimal Residual Method.
MLP	Multi-Layer Perceptron.
MPI	Message Passing Interface.
NLP	Natural Language Processing.
RNN	Recurrent Neural Network.
TDG	Task Dependency Graph.
VLSI	Very Large Scale Integration.