International Journal of Engineering Trends and Technology
ISSN: 2231 – 5381 /doi:10.14445/22315381/IJETT-V69I4P232

Volume 69 Issue 4, 236-246, April 2021
© 2021 Seventh Sense Research Group®

Parallel and Scalable Deep Learning Algorithms for High Performance Computing Architectures

Sunil Pandey[#1], Naresh Kumar Nagwani[#2], Shrish Verma[#3]

[#1]Research Scholar, [#2]Associate Professor

Department of Computer Science and Engineering, NIT Raipur 492010, CG, India

[#3]Professor, Department of Electronics and Communication Engineering, NIT Raipur 492010, CG, India

[1]sys_admin@nitrr.ac.in, [2] nknagwani.cs@nitrr.ac.in, [3] shrishverma@nitrr.ac.in

Abstract *— This paper elucidates the state-of-the-art design of parallel and scalable deep learning algorithms for high-performance computing (HPC) architectures. The paper starts with an application-focused introduction to deep learning. The HPC architectures discussed next include multicore processors and multi systems, which are representatives of the shared and distributed parallel programming paradigms, respectively. Followed by this is a discussion of the computational challenges inherent in deep learning. A review of research in deep learning and HPC has been carried out, and a short summary in the tabular form was provided. Open research directions in the field have been highlighted. Key steps in the deep learning algorithm development process for HPC are then discussed, followed by the possible outcomes. One section each has been dedicated to convolutional neural networks and the high-performance computing environment. The materials and methods used in a computational experiment in deep parallel learning have been described next. The experiment involves the design and development of a parallel algorithm and program for compute-intensive deep learning primitive and its performance testing. The results and the performance of the deep learning parallel program have been discussed. The paper ends with the concluding remarks in the conclusions.*

Keywords — *Parallel, Scalable, High Performance Computing, Multicore, Compute Cluster, Shared Parallel, Distributed Parallel, Deep Learning Algorithms.*

I. INTRODUCTION

Deep learning is a young discipline in the field of machine learning and is a very exciting and active research area at present. Deep learning can be thought of as an extension of the field of artificial neural networks since, at their core, deep learning networks can be thought of as very large artificial neural networks in many cases. Deep learning networks are capable of automatically learning features and patterns at multiple levels of abstraction. Deep learning has seen many early successes in a range of applications. Examples include but are not limited to "Automatic Colorization of Gray-Scale Images" [1-4], "Automatically Adding Sounds to Silent Movies" [5], "Automatic Translation of Text" [6-8], "Image Classification and Object Detection in Photographs" [9-12], "Automatic Handwriting and Text Generation" [13][14], "Automatic Image Caption Generation" [15]. Deep learning networks and algorithms have also been used for predictive IoT data analytics [16], in medical diagnosis [17], to develop a strategy for increased video consumption [18], for the reduction of genome sequence errors [19], for processing of remote sensing images [20], etc. It can be observed that that the applications highlighted above are all applications in the field of Artificial Intelligence characterized by large datasets.

Common examples of deep learning networks include "deep multilayer feed-forward neural networks," "deep convolutional neural networks," and "deep recurrent neural networks" [21]. Deep learning networks and algorithms are typically applied to AI problems that involve large dataset mining and big data analysis [22].

A typical deep learning network, the deep multilayer feed-forward neural network is a very large artificial neural network (ANN) having several layers of neurons and several neurons in each layer. Massive amounts of data are run through the system to train this ANN. As deep learning networks are very large neural networks, they are characterized by a very large number of adaptable parameters or weights which are tuned in the training phase. Parameter training is done in the training phase using specialized mathematical algorithms, such as the feed-forward error backpropagation algorithm for multilayer feedforward neural networks. Since the number of parameters involved in such networks is very large, the training phase of deep learning networks is typically very compute-intensive and can become prohibitively so on high-dimensional data [23]. It is primarily due to the compute-intensive nature of the deep learning algorithms that there arises a need for the development of specialized, efficient parallel, and distributed algorithms and codes for high-performance computing architectures.

II. BUILDING BLOCKS OF HIGH-PERFORMANCE COMPUTING SYSTEMS

Multicore, Multiprocessor, and Multi-systems can be regarded as the basic or foundational building blocks of contemporary high-performance computing systems. The above can also be regarded as representing increasing levels of parallelism in a modern multisystem high-performance computing environment.

A multicore processor [24] is a single processor which contains two or more complete functional processing units, also called the processor cores or the CPU cores. Such chips are now the focus of two major processor manufacturers, i.e., Intel and AMD. Fig. 1 shows the high-level architecture of a typical multicore processor.

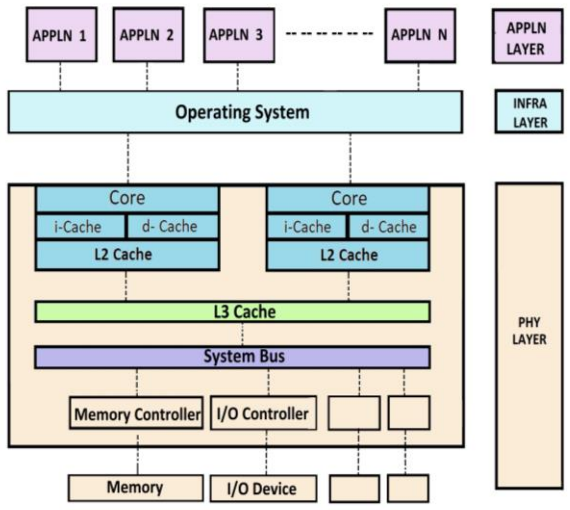

Fig. 1: Multicore HPC Architecture

A multiprocessor system refers to a system in which two or more single-core or multi-core processors have access to the same memory bank [25]. If the processors have equal access to the system memory, then such a system is called a symmetric multiprocessor system. Fig. 2 shows the architectural elements of a symmetric multiprocessor system.

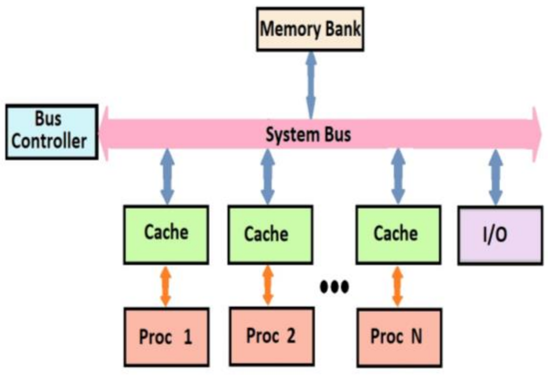

Fig. 2: Symmetric Multiprocessor Architecture

A multisystem refers to a group of symmetric multiprocessor systems connected via a high-speed data network. Due to the network latency, the processors of one system cannot share a memory with the processors of another system.

Such systems are appropriate for problems that do not require frequent communication between processors, i.e., the loosely coupled problems, while tightly coupled problems are more appropriate for symmetric multiprocessor machines. Fig. 3 shows the architectural details of a typical multisystem.

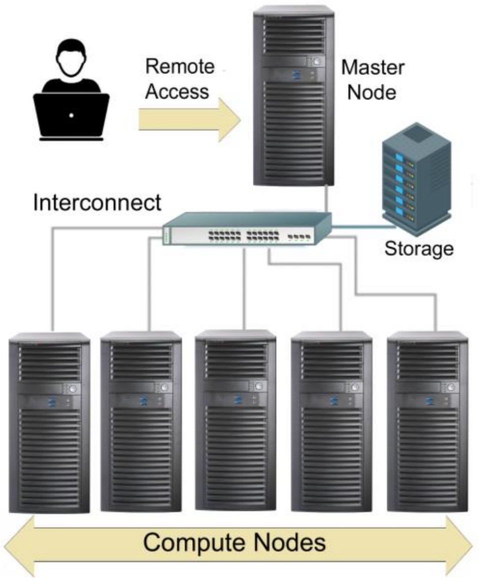

Fig. 3: Multisystem HPC Architecture

III. COMPUTATIONAL CHALLENGES OF DEEP LEARNING

The computational challenges inherent in deep learning are best highlighted by means of an illustrative example. Fig. 4 shows a multilayer feed-forward neural network. If, for example, there is a deep multilayer feed-forward neural network with 100 neurons in each of its 10 hidden layers for use as a classifier in lung cancer detection with the dataset comprising of 10,000 high-resolution low-dose CT scan images, then there are 1000 neurons, and 1,00,000 parameters to be adjusted in every cycle, not counting the output layer neurons or the pre-processing required by the high-resolution images in the feature extraction phase.

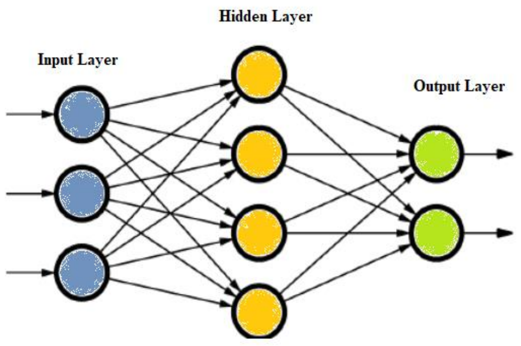

Fig. 4: Typical Multilayer Feed Forward Net [23]

From the perspective of computation, training a large deep neural network, as explained in the example above, is not a trivial task, and training deep networks of such size on typical desktops and workstations has been observed to take inordinate times for completion.

Further on, the system resources are bogged down by the compute-intensive deep learning training programs, rendering the desktop or the workstation practically useless for other tasks during the period that these training programs are running.

This simple observation indicates to us that for true deep learning for real-world applications, as against toy models used for technology demonstration purposes, new fast and efficient algorithms and/or customized software and hardware solutions are an essential requirement. Conventional algorithms, software, and hardware that is typically designed and developed for serial execution prove to be unsuitable for such computation-intensive real-world tasks.

IV. LITERATURE REVIEW

Many researchers are working on accelerating deep learning applications for achieving high computational efficiencies on different architectures and platforms. Some of the literature which is valuable in the context of deep learning and high-performance computing is briefly summarized in Table 1.

TABLE I
Review of Research on Deep Learning Acceleration

Author	Title	Research
"LeCun, Y., et. al. (2015)"	"Deep Learning"	Analysis of Methods and Applications Deep Multilayer NNs + CNN + RNN
"Chen, X.W., et. al. (2014)"	"Big Data Deep Learning: Challenges and Perspectives"	Analysis of challenges in using deep learning for big data
"Najafabadi, M.M., et. al. (2015)"	"Deep learning applications and challenges in big data analytics."	Analysis of challenges in using deep learning for big data
"Sun, X.H., et.al. (2008)"	"Scalable Computing in the Multicore Era"	Analysis of speedup models on multicore architecture
"Giles M.B., et. al. (2014)"	"Trends in HPC for engineering calculations."	Analysis of key developments on the hardware side, recent past + near future
"Angelov, P., et. al. (2016)"	"Challenges in Deep Learning"	Analysis of challenges in deep learning
"Dean, J., et. al. (2012)"	"Large Scale Distributed Deep Networks"	Development of two algorithms – Downpour SGD + Sandblaster for large-scale distributed training
"J. Hauswald *et. al.* (2015)"	"DjiNN and Tonic: DNN as a service and its implications for future warehouse-scale computers."	Open infrastructure for Deep neural networks as a service in Warehouse scale computers
"Bouache, M., et. al. (2016)"	"Deep Learning GPU-Based Hardware Platform Hardware and Software Criteria and Selection"	Hardware + Software Criteria and Selection
"Le, Q.V., et. al. (2011)"	"On Optimization Methods for Deep Learning"	Limited memory BFGS + Conjugate gradient (CG) algorithms for pre-training deep networks
"Hegde, V., et. al. (2016)"	"Parallel and Distributed Deep Learning"	CNN + SGD + ADMM + Downpour SGD
"Keuper, J., et. al. (2015)"	"Asynchronous Parallel Stochastic Gradient Descent A Numeric Core for Scalable Distributed Machine Learning Algorithms" [32]	SGD
"Vanhoucke, V., et. al. (2011)"	"Improving the Speed of Neural Networks on CPUs" [33]	Code speedup on x86 CPUs
"Gupta, S., et. al. (2015)"	"Deep Learning with Limited Numerical Precision" [34]	16-bit wide fixed-point number representation
"Chetlur, S., et. al. (2014)"	"cuDNN: Efficient Primitives for Deep Learning" [35]	High Performance Library Development for GPU based deep learning
"Delong, A., et al."	"Practical Guide to Matrix Calculus for Deep Learning" [36]	Matrix calculus for deep learning algorithms
"Baoyuan Liu, et. al. (2015)"	"Sparse Convolutional Neural Networks"[37]	Sparsely connected CNNs

"Ionescu, C., et. al. (2015)"	"Matrix Backpropagation for Deep Networks with Structured Layers" [38]	Matrix formulation of Backpropagation Algorithm
"Zhang, Y., et. al. (2013)"	"Optimized Deep Learning Architectures with Fast Matrix Operation Kernels on Parallel Platform" [39]	Design of fast matrix operation kernels on parallel platforms for deep learning

V. OPEN RESEARCH DIRECTIONS

Contemporary large-scale deep learning models need a significant amount of computational power to reach acceptable performance levels on medium and large-size datasets, especially in real-time and online environments. However, the amount of data available is growing very rapidly in terms of volume, velocity, and variety. Thus, there is a requirement for a class of deep learning algorithms that can be trained efficiently on big data. HPC architectures and infrastructures coupled with specially designed parallel and distributed algorithms and codes capable of efficient deep learning computations and performance-optimized on such architectures and infrastructures are also a key requirement for handling this kind of data [26].

The models and data available for deep machine learning applications have grown significantly over the last few years. High-performance computing architectures in combination with the right deep learning codes are capable of accelerating the performance and enabling us to make sense of such large data sets through recognition of latent patterns and inherent knowledge.

The focus of most of the current literature and public implementations of deep machine learning algorithms is generally on either cloud-based, i.e., warehouse-scale [27][28] or on small-scale GPU environments [29][30][31]. These implementations do not scale well in high-performance computing environments because of inefficient data movement and network communication within the multisystem computers, originating from significant imbalances in the level of parallelism. Furthermore, the application of deep machine learning to extreme-scale scientific data remains by and large unexplored. In order to leverage high-performance computing for deep machine learning applications, advancements are required in both algorithms and their scalable, parallel, and distributed versions. Also, work is disjointed; no structured performance and scalability studies at different levels of CPU parallelism appear to have been carried out, and not much benchmarking data/benchmarks are available.

The computation-intensive nature of deep learning algorithms leads to the following two clear research directions:

1. The design of fast and computationally efficient serial algorithms for training deep learning networks, and
2. The design of parallel and scalable algorithms for training deep learning networks exploiting recent advances in high performance computing architecture and hardware

Deep learning is being used for the solution to many hard problems. It is clear that the training of deep learning networks is a problem that is well-suited for HPC ecosystems. Using parallel and distributed algorithms on HPC hardware reduces the training time of deep learning networks. The scaling property of the parallel and distributed algorithms helps in the application of deep learning to the ever-increasing volume of the data sets and assists in the progress of deep learning.

Parallel and scalable algorithms for deep learning enable many new applications and provide the capabilities for experimenting with bigger models, larger datasets, and more ideas within a given timeframe reducing the artificial intelligence research cycle and product development times.

VI. KEY PHRASES IN PARALLEL DEEP LEARNING APPLICATION DEVELOPMENT

Key phases towards the research and development of parallel and scalable deep learning algorithms for HPC architectures are mentioned in Table 2. The phases contain sub-tasks which are described in what follows.

The HPC architectures considered in this paper, i.e., multicore processors and compute clusters, represent the proven and time-tested shared and distributed memory parallel programming paradigms, respectively.

TABLE II
The Key Phases

Phase-A	Understanding the specific deep neural network algorithm and the HPC environment
Phase-B	Analysis and design of high-performance deep learning algorithm or computational primitive
Phase-C	Programming for multisystem implementation
Phase-D	Performance and scalability evaluation
Phase-E	Comparative evaluation of performance with other platforms

Phase-A: Understanding the Specific Deep Neural Network Algorithm and the HPC Environment

This is the learning and understanding phase. In this phase, selection of the deep learning network algorithm, or the computational primitive thereof, which is to be taken up for parallelization, e.g., deep multilayer feed-forward neural network, deep convolutional neural network, deep recurrent neural network, etc. is done. Literature is closely followed. This phase comprises the following two subtasks:

Understanding the Algorithmic Implementation Details of Selected Deep Neural Network or Computational Primitive (Phase A1)

Widely used deep learning networks, e.g., deep multilayer feed-forward neural network, deep convolutional neural network, deep recurrent neural network, etc., are studied with a focus on their mathematical, algorithmic, and algorithmic implementation details before selecting the computationally intensive deep learning algorithm or the computational primitive to be parallelized.

Understanding the Building Blocks of HPC architecture, Programming and Performance Considerations (Phase A2)

This phase involves a closer look at the HPC architecture, parallel programming models, constructs, programming languages, tools, libraries, and functions available for implementing the selected deep learning algorithm or computational primitive. Necessary programming and code optimization considerations also demand a closer look in this phase.

Phase-B: Analysis and Design of Parallel Deep Learning Algorithm / Computational Primitive

This phase comprises the following subtasks:

Analysis of the Serial Algorithm (Phase B_1)

An in-depth analysis of the serial algorithm of the selected deep neural network or computational primitive needs to be carried out in this phase.

Design of the HPC Algorithm (Phase B_2)

In this subtask, the selected deep neural network or computational primitive shall be partitioned into functions, procedures, and subroutines, with an eye on identifying significant parallel computation or distributed computation opportunities.

Analysis of the HPC Algorithm (Phase B_3)

A preliminary analysis of the parallel or distributed HPC algorithm shall be done in this subtask.

Phase-C: Programming for HPC Implementation

This phase shall involve programming the selected deep neural network, e.g., deep multilayer feed-forward neural network, deep convolutional neural network, deep recurrent neural network, etc., or the selected computational primitive for the HPC architecture considered. Debugging and testing shall also be carried out alongside the development in this phase.

Phase-D: Performance and Scalability Evaluation

The following shall be performed using the selected large dataset(s). The dataset shall be from large-scale pattern recognition domains like imaging.

Performance Evaluation of Serial Vs. HPC Program (Phase-D_1)

The HPC program for the deep neural network, e.g., deep multilayer feed-forward neural network, deep convolutional neural network, deep recurrent neural network, etc., has to be run by varying the numbers of nodes, and this data has to be compared with the equivalent serial implementation on the selected dataset. The main parameter to be noted for performance evaluation are the execution times of the corresponding programs.

Scalability Assessment of the HPC Program (Phase-D_2)

In this subtask, the scalability of the system has to be analyzed as well as empirically evaluated by running the HPC program for the deep neural network, e.g., the deep multilayer feed-forward neural network, deep convolutional neural network, deep recurrent neural network, etc. by varying the numbers of nodes which participate in the computation. The clock times with different numbers of nodes shall be noted. If the graph between execution time and the number of nodes shows a linear decreasing trend with a good slope, the algorithm and its implementation are scalable. Otherwise, necessary improvisations to the algorithm and/or the source code will be necessitated to make the program scale better.

Comparative Evaluation of Performance and Scalability with other Software (Phase-E)

In this task, the performance and scalability of the multisystem program for the deep neural network, e.g., deep multilayer feed-forward neural network, deep convolutional neural network, deep recurrent neural network, etc., are compared with other software implementations, if comparable.

VII. POSSIBLE OUTCOMES

Following are the possible outcomes of a sincere effort in the design and development of parallel and distributed deep learning algorithms for HPC architectures

1. Optimized high-performance deep learning codes for HPC architectures.

2. The HPC codes can be used to handle big data and test out larger models in deep learning, which may not be practicable on desktops, workstations, etc.

3. The HPC codes can also be used in deep learning application development.

4. Performance and scalability benchmarks shall become available for researchers and practitioners.

5. Depending on the performance of the codes on the large dataset used for performance and scalability evaluation, new application spin-offs are possible.

VIII. UNDERSTANDING DEEP NEURAL NETWORKS: CONVOLUTIONAL TYPE

A CNN has primarily comprised of the following four layers: the convolutional layer(s), pooling or sub-sampling layer(s), non-linear layers, and fully connected layer(s).

Each feature of a layer in a CNN receives its inputs from a small neighborhood of features located in a local receptive field of the previous layer. Using local receptive fields, it is possible to extract fundamental visual features, such as edges, corners, etc. These fundamental features are combined in the higher layers.

The convolutional layers of a CNN are the feature extractors. The weights of the convolutional filter kernels are tuned during the training process. The convolutional layers are able to extract local features because the receptive fields in the hidden layers of the CNN are restricted to small localized regions.

CNNs have found applications in computer vision, image recognition, pattern recognition, speech signal recognition, natural language processing (NLP), and video analysis. In CNNs, the weights of the convolutional layer, which is used in the feature extraction stage, and the weights in the fully connected layers used for the classification task are adapted during the training process. The network structures of CNNs lead to optimal memory and computation complexity requirements. They also give superior performance in applications like image and speech recognition, where the input is locally correlated.

Training and evaluation of CNNs require large computational resources. The large requirements of computational resources are sometimes met by graphical processing units (GPUs), Digital Signal Processors (DSPs), Field Programmable Gate Arrays (FPGAs), or other specialized silicon architectures which have been optimized for high throughput and lower power while they execute the characteristic patterns of CNN computation.

CNNs have achieved the highest correct detection rates (CDRs) in image and pattern recognition applications - 99.77% on the MNIST database of handwritten digits [40], 97.47% on the NORB 3D objects dataset [41], and 97.6% on ~5600 images of more than 10 objects [42]. They have not only given better performance compared to other detection algorithms but have also surpassed humans in object classification tasks involving fine-grained categories such as the particular breed of dog or bird species [43].

Complex CNN architectures are built for difficult classification problems by stacking up the four layers, i.e., the convolution layers, pooling or sub-sampling layers, non-linear layers, and fully connected layers in different permutations and combinations. Fig. 5 shows a typical CNN architecture for the image classification problem.

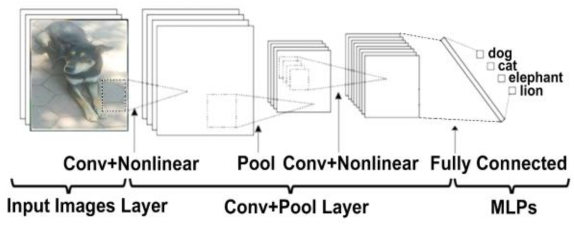

Fig. 5: CNN Architecture for Image Classification

Convolution Layers: The convolution operation is the feature extractor that is used for the extraction of the different features of the input image. The initial convolutional layers are used for the extraction of the low-level image features, also called the fundamental image features, i.e., the edges, lines, corners, etc. The subsequent layers are used for the extraction of the higher-level features.

Fig 6(a) explains the process of three-dimensional convolution, which is used in CNNs. The input is of size M x N x 3, which corresponds to an RGB image having the red, blue, and green channels, respectively. It is convolved with H filters, each of size k x k x 3 separately. Each filter is comprised of three separate k x k x 1 convolutional kernels. The convolution operation of one input image with one convolutional filter produces one output feature. With H independent filters, H number of output features are produced.

Beginning from the top-left corner of the input, each of the filters is shifted from left to right, one element at a time. When the top-right corner is reached, the filter is moved one element in the downward direction and back to the left corner in the horizontal direction. Once again, the filter is shifted from left to right, one element at a time. The process is continued until, eventually, the filter reaches the bottom-right corner. At each receptive field, pointwise multiplications between the image matrix value and the corresponding convolutional filter kernel are performed and their sums computed. These sums replace the local receptive field in the output.

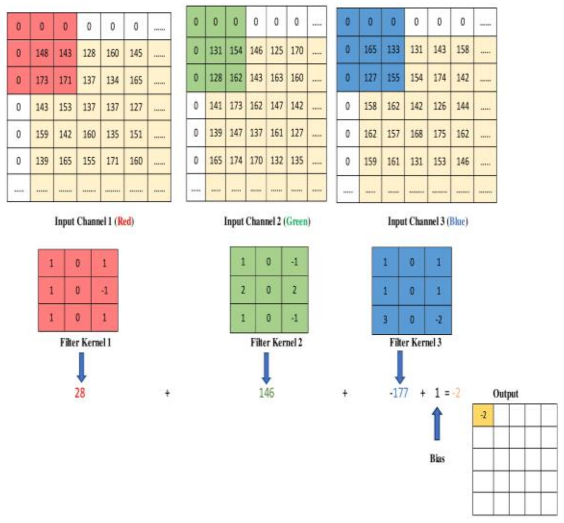

Fig. 6(a): Three Dimensional Convolution Process

Pooling Layers: The pooling / sub-sampling layer is used for dimension reduction of the feature space making the features robust to noise and distortion. Two common pooling operations are in use. They are the max-pooling and the average pooling operations, respectively. In max pooling, the selected region or window of the input image is replaced by its maximum value, while in the case of average pooling, the selected region or window of the input image is replaced by its average value. The window or region shifts from the top left corner of the input image to its bottom right corner in a manner similar to the movement of the convolutional filter and kernels. In both these operations, the input image is partitioned into non-overlapping two-dimensional regions. Fig 6(b) demonstrates the max and average pooling operations in a CNN.

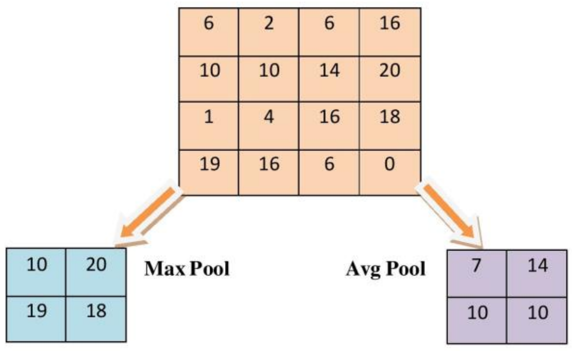

Fig. 6(b): Max and Average Pooling Operations

Nonlinear Layers: In order to signal the distinct identification of likely features which are present in the hidden layers, the CNNs make use of specific nonlinear functions. Commonly used functions include the rectified linear unit (ReLU), hyperbolic tangent, absolute hyperbolic tangent, and the sigmoid functions. Since this function operates on an element-by-element basis, the size of the input and the output are the same.

The mathematical equations of these non-linear functions are as follows:

Rectified Linear Unit (ReLU):

$$y = \max(0, x) \tag{1.1}$$

Hyperbolic Tangent:

$$y = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{1.2}$$

Absolute Hyperbolic Tangent:

$$y = abs(\tanh(x)) = \left| \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \right| \tag{1.3}$$

Sigmoid Function:

$$y = \frac{1}{1 + e^{-\lambda x}} \tag{1.4}$$

Fig. 6(c) shows the output when a ReLU function operates on a selected 4 x 4 region of an image.

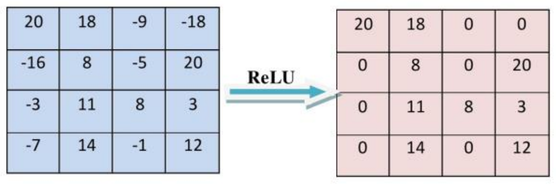

Fig. 6(c): ReLU operation on a selected image region

Fully Connected Layers: Fully connected layers are typically used in the last layers of a CNN. The fully connected layers comprise multiple artificial neurons, also called perceptrons. The artificial neurons first compute the weighted sum of the previous layer of features, add a bias term and pass this output through a nonlinear function which is typically the sigmoid function. During the training phase, the weights are modified as per a training algorithm so that the error or cost, which is a measure of the difference between the expected and the actual outputs, is minimized. The backpropagation algorithm is the most commonly used training algorithm.

Fig. 6(d) shows the "computational" architecture of an artificial neuron in the fully connected layer.

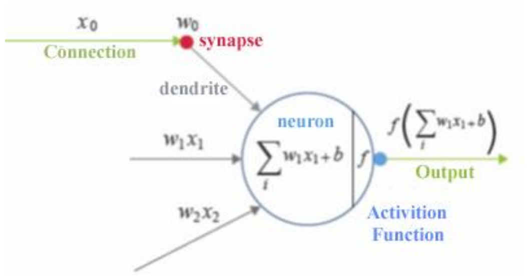

Fig. 6(d): Architecture of an Artificial Neuron

Fig. 6(e) shows the adjustment of weights of the neurons of the fully connected layer during the training phase. The weights are adjusted so as to minimize the error.

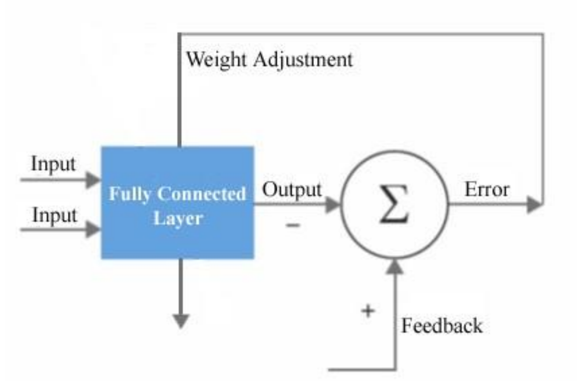

Fig. 6(e): Weight adjustment during the training phase

IX. UNDERSTANDING THE HIGH-PERFORMANCE COMPUTING ENVIRONMENT

The building blocks of high-performance computing systems have already been discussed in section II. In this section, the computing environment for high-performance computing systems is discussed.

The focus is primarily on the software side. Multicore and multisystem models are the two high-performance computing systems that have been considered in this work. In this context, Java is a programming language that has in-built features for the explicit programming of multicore as well as multi systems. Besides, Java has the advantage of being a modern “simple, general-purpose, object-oriented, interpreted, robust, secure, architecture-neutral, portable, high-performance, and multithreaded” computer programming language with automatic memory management

Java Multithreading: Multithreading refers to the simultaneous execution of several threads. A thread is defined as a lightweight process and is the smallest unit of processing.

Threads use a shared memory model. There is no separate memory allocation which saves memory and the context switching between the threads is also faster compared to a process. Threads execute independently of each other. A multi-threaded program running on a single-core chip needs to interleave the threads. However, in multicore high-performance computing architectures, the threads can be distributed across the available cores enabling true parallel processing. Fig. 7 shows the events and states of Java threads.

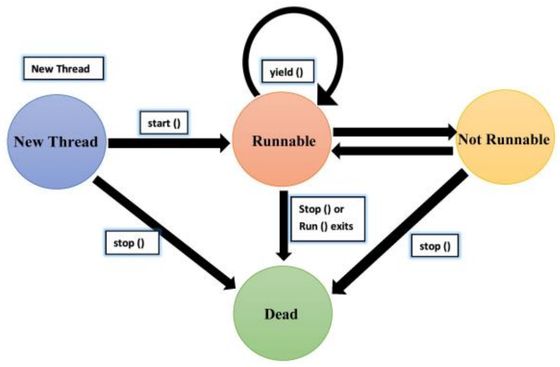

Fig 7: Events and States of Java Threads

Java Remote Method Invocation: Distributed computing can be treated as partitioning an application into separate computing agents, which are then distributed and executed on a network of computers. However, all the computing agents work in unison to execute tasks cooperatively. Java provides a feature called remote method invocation (RMI), which enables programs to invoke Java objects on remote machines.

Two programs, one for the server and the other for the client, are written in the case of an RMI application. A remote object is created in the server program, and a reference to that object is made available for the client program using the RMI registry. The client program makes requests for the remote server objects and invokes its methods. The components of the RMI architecture include the transport layer, the stub, the skeleton, and the remote reference layer (RRL).

The transport layer enables a network connection between the client and the server programs. Its role is to manage the existing connections and also to set up new connections when required. A stub is a representation of the remote object at the client. It is the gateway to the client program. The client-side stub communicates with the server-side skeleton for passing requests to the remote object. The remote reference layer manages the references made by the client to the remote object.

Client-side calls to the remote object are received by the stub, which passes this request to the RRL. On receiving the client-side requests, the RRL invokes a method called invoke of the remote object, which passes this request to the corresponding RRL on the server-side. The server-side RRL passes the request to the Skeleton, which invokes the corresponding object on the server. The result is passed back to the client. Fig. 8 illustrates the architecture of RMI.

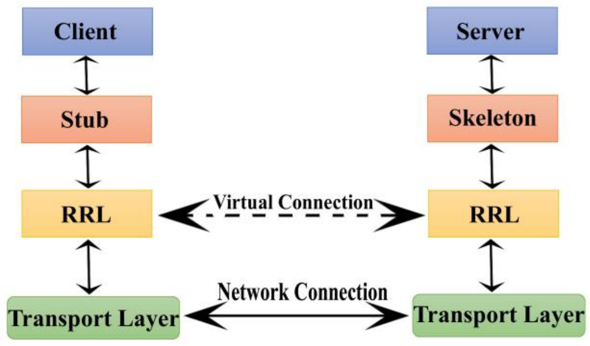

Fig 8: The RMI Architecture

If a client invokes a method on a remote object which requires parameters, then these parameters are first bundled into a message before sending them over the network. The parameters may be primitive type parameters or object type parameters. If the parameters are of primitive type, they are assembled, and a header is attached to the bundle. If the parameters are objects, then they are serialized. This procedure is called marshaling. On the server-side, the packed parameters are unbundled, and after this step, the method called by the client is invoked. This procedure is called unmarshalling.

The RMI registry is a namespace containing the names of all the server objects. When the server creates an object and needs to register it, it makes use of the bind or rebinds methods. The objects are registered in the RMI registry using a unique name called bind name.

For invoking a remote object on a server, the client needs a reference of the remote object, which it fetches from the registry using its bind name through the lookup method.

X. MATERIALS AND METHODS

This section contains the details of the materials and methods used in a computational experiment which demonstrates the performance gains made through parallelization of a deep learning primitive. The deep learning primitive chosen in this experiment is the image convolution operation in deep convolutional neural networks. The convolution operation takes place in the convolutional layers of the convolutional neural networks. The operation is repetitively performed on all the input images of a domain-specific image dataset. It is a very compute-intensive operation. The image convolution *S* between an image *I* and a kernel *K* is defined as follows

$$S_{ij} = (I * K)_{ij} = \sum_{a=0}^{m-1}\sum_{b=0}^{n-1} I_{i+a,j+b}K_{a,b} \tag{2}$$

In this computational experiment, the image convolution operation is performed on a high definition 1280 px by 720 px two-dimensional grayscale image using 128 kernels of size 11 x 11. A multicore program was written to perform this operation in parallel. The program can be run on one or more cores of a multicore processor machine. The program is written in the Java programming language version 8. The program was executed on a dual-socket AMD server with one AMD Opteron (tm) Processor 6136 family. The processor has 8 CPU cores. The current speed of the processor is 2400 MHz. The server has a total of 8GB of DDR3 random access memory, having a speed of 1333 MT/s. Fig. 9 shows the test image, a 1280 pixel x 720-pixel grayscale image of a dog named Gugu on a hexagonal paver block background.

Fig. 9: 1280 x 720-pixel test image of Gugu

XI. RESULTS AND DISCUSSION

Table III shows the results of the computational runs when the program has been executed using 1, 2, 4, and 8 cores of the processor. Ten computational runs were performed for each of the core counts. The net execution time on each core has been calculated as the average of the ten computational runs.

TABLE III
Results of the Computational Runs

	CPU Cores	**1**	**2**	**4**	**8**
Execution Time (s)	Run01	84.497	48.447	26.302	14.401
	Run02	87.353	48.816	25.497	14.567
	Run03	86.299	48.956	26.349	14.276
	Run04	85.600	49.568	25.290	14.378
	Run05	83.734	48.799	25.286	13.843
	Run06	84.826	49.434	25.777	14.049
	Run07	85.173	48.974	24.479	14.178
	Run08	90.199	49.031	25.422	14.325
	Run09	86.308	48.666	25.086	14.461
	Run10	85.136	49.943	24.278	14.323
Total Time (s)		859.125	490.664	253.766	142.801
Avg Time (s)		85.9125	49.0664	25.3766	14.2801

The average computer time for one image for each core count can be seen to decrease from approximately 86 seconds when utilizing a single processor core to approximately 14 seconds when utilizing all the 8 processor cores. This is the time taken for performing convolutions using 128 convolution kernels of size 11 x 11 on a grayscale high definition digital image of size 1280 pixels x 720 pixels.

This has been visualized in Fig. 10, which is a plot of the average compute time versus the number of cores on which the multicore program has been executed.

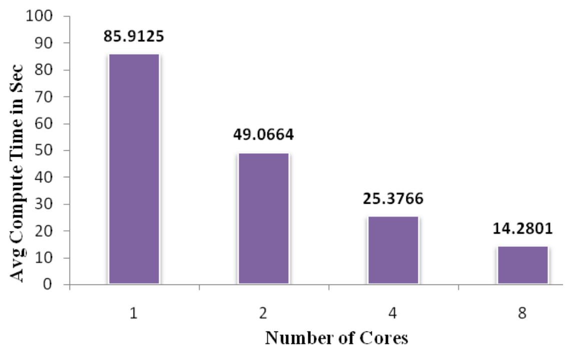

Fig. 10: Average compute time per image versus the number of cores

In parallel computing, the overall speedup is defined as the ratio of the old execution time and the new execution time. The overall speedup between uni-core performance and octa-core performance is calculated to be nearly 6 for this computational experiment. This means that the multicore program has completed the program execution and produced the output on 8 numbers of processor cores 6 times faster than on a single processor core.

Table IV shows the overall speedups obtained in the case of the multicore program considered.

TABLE IV
Overall speedups with core count

Core Counts	**Overall speedups**			
	1	**2**	**4**	**8**
1	1	*1.751*	*3.385*	*6.016*
2	-	1	*1.934*	*3.436*
4	-	-	1	*1.777*
8	-	-	-	1

It can be seen from the above table that the overall speedup is 1.75 between uni-core performance and dual-core performance. It is calculated to be 3.38 between unicore and quadcore performance and 6 between unicore and octa-core performance for this computational experiment.

This data has been visualized in Fig. 11, which shows the overall speedups versus the number of cores.

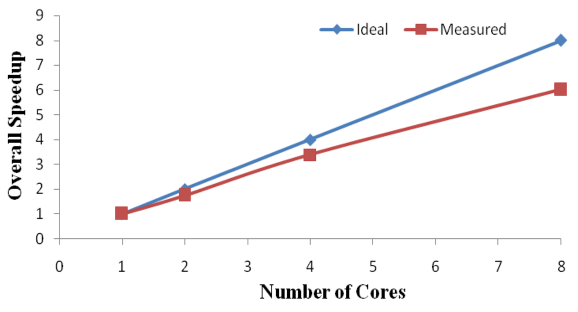

Fig. 11: Plot showing the ideal and actual overall speedups versus the number of cores

In the ideal scenario, that is, if the program was perfectly scalable, the overall speedup, which is an important performance metric, should have been in the ratio of the core counts. In the actual scenario, the overall speedup obtained is lower, and from the graph, the gap between the ideal and the actual speedups can be seen to widen with increasing core counts. However, it can be said that the program scales well on the processor considered.

This computational experiment demonstrates that the compute-intensive primitives of deep learning are amenable to parallelization. Significant performance gains of nearly 600% have been obtained by developing a multicore parallel program for the convolution operation and executing the same by increasing the core counts. The program can be seen to scale well with increasing core counts on the server processor considered in the present work.

XII. CONCLUSIONS

In this paper, an attempt has been made to elucidate the state-of-the-art developments in the design of parallel and scalable deep learning algorithms for HPC architectures. Common deep learning algorithms have been highlighted. An overview of the HPC architectures, including multicore processors and multisystem, has been provided. The computational challenges inherent in deep learning necessitating the use of HPC have been brought out. A review of research literature in deep learning acceleration has been summarized in tabular form. Open research directions have been identified. Key steps in the design and development of parallel and distributed deep learning algorithms for HPC architectures have been discussed, followed by the possible outcomes. A detailed discussion on CNNs and HPC environment has been made. Finally. A multicore program for a compute-intensive deep learning primitive has been designed and developed, and its performance tested.

REFERENCES

[1] Zezhou Cheng, Qingxiong Yang, and Bin Sheng, "Deep Colorization, in 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, (2015) 415-423.

[2] Zhang R., Isola P., and Efros A.A., Colorful Image Colorization, European Conference on Computer Vision, Springer, 9907 (2016)

[3] Larsson G., Maire M., and Shakhnarovich G., Learning Representations for Automatic Colorization, Computer

Vision(ECCV), Lecture Notes in Computer Science - Springer, 9908, (2016).

[4] Hwang, Jeff, and You Zhou., Image Colorization with Deep Convolutional Neural Networks, Stanford University,

[5] Andrew Owens et al., Visually Indicated Sounds, in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, (2016) 2405 - 2413.

[6] I Sutskever, O. Vinyals, and Q.V. Le., Sequence to Sequence Learning, in Proc. Advances in Neural Information Processing Systems 27 (2014) 3104–3112.

[7] K. Cho et al., Learning phrase representations using RNN encoder-decoder for statistical machine translation, in Proc. Conference on Empirical Methods in Natural Language Processing, (2014) 1724–1734.

[8] Zhang Jiajun and Zong Chengqing., Deep Neural Networks in Machine Translation: An Overview," IEEE Intelligent Systems, 30(5) (2015)16-25.

[9] A. Krizhevsky, I. Sutskever, and G. Hinton., ImageNet classification with deep convolutional neural networks, in NIPS Proceedings, (2012).

[10] A. G. Howard., Some improvements on deep convolutional neural network-based image classification, in International Conference on Learning Representation (ICLR), Banff, Canada, (2014).

[11] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov., Scalable Object Detection Using Deep Neural Networks, in IEEE Conference on Computer Vision and Pattern Recognition, Columbus, (2014) 2155-2162.

[12] D. Erhan, C. Szegedy, and A. Toshev., Scalable object detection using deep neural networks, in CVPR, (2014).

[13] Alex Graves., Generating Sequences With Recurrent Neural Networks, [Online]. https://arxiv.org/pdf/1308.0850.pdf

[14] Ilya Sutskever, James Martens, and Geoffrey E Hinton., Generating text with recurrent neural networks, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), New York, NY, (2011) 1017-1024

[15] Andrej Karpathy and Li Fei-Fei., Deep Visual-Semantic Alignments for Generating Image Descriptions, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, Massachusetts, (2015) 3128 - 3137.

[16] Ayushi Chahal, Preeti Gulia., Deep Learning: A Predictive IoT Data Analytics Method, International Journal of Engineering Trends and Technology, 68(7) (2020).

[17] P. Seetha Subha Priya, S. Nandhinidevi, M. Thangamani, S. Nallusamy., A Review on Exploring the Deep Learning Concepts and Applications for Medical Diagnosis, International Journal of Engineering Trends and Technology, 68(10) (2020).

[18] Sangeeta, Preeti Gulia., Deep learning-based combating strategy for COVID-19 induced increased video consumption, International Journal of Engineering Trends and Technology, 68(7) 2020.

[19] Ferdinand Kartriku, Robert Sowah, Charles Saah., Deep Neural Network: An Efficient and Optimized Machine Learning Paradigm for Reducing Genome Sequencing Error, International Journal of Engineering Trends and Technology, 68(9) (2020).

[20] Ramya T.E., Marikkannan, M., Investigations on Combinational Approach for Processing Remote Sensing Images Using Deep Learning Techniques, International Journal of Engineering Trends and Technology, 67(8) (2019).

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton., Deep Learning, 521(2015) 436.

[22] X. W. Chen and X. Lin., Big Data Deep Learning: Challenges and Perspectives, IEEE Access, 2 (2014) 514-525.

[23] M.M. Najafabadi et al., Deep learning applications and challenges in big data analytics, Journal of Big Data, (2015) 1.

[24] Xian-He Sun, Yong Chen, and Surendra Byna., Scalable Computing in the Multicore Era, in Proceedings of the Inaugural Symposium on Parallel Algorithms, Architectures, and Programming, Hefei: University of Science and Technology of China Press, (2008).

[25] M. Tanveer, M.A. Iqbal, and F. Azam., Using Symmetric Multiprocessor Architectures for High Performance Computing Environments, International Journal of Computer Applications, 27(9) (2011).

[26] P. Angelov and A. Sperduti., Challenges in Deep Learning, in European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges (Belgium), M. B. Giles and I. Reguly, Trends in high-performance computing for engineering calculations," in Phil.Trans.R.Soc.A, (2016) 27-29.

[27] J. Dean et al., Large Scale Distributed Deep Networks, in 25th International Conference on Neural Information Processing Systems, Lake Tahoe, Nevada, (2012) 1223-1231.

[28] J. Hauswald et al., DjiNN and Tonic: DNN as a service and its implications for future warehouse-scale computers, in Proceedings 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, (2015) 27-40.

[29] M. Bouache and J. Glover., Deep Learning GPU-Based Hardware Platform Hardware and Software Criteria and Selection, in ICS-2016, Istanbul, Turkey,(2016).

[30] Q. Le et al., On Optimization Methods for Deep Learning," in Proceedings of the International Conference on Machine Learning, Washington, (2011)

[31] V. Hegde and S. Usmani., Stanford University, (2016). https://web.stanford.edu/~rezab/dao/projects_reports/hedge_usmani.pdf [Online].

[32] J. Keuper and F.J. Pfreundt., Asynchronous Parallel Stochastic Gradient Descent A Numeric Core for Scalable Distributed Machine Learning Algorithms, in Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, Austin, TX, USA, (2015).

[33] V. Vanhoucke, A. Senior, and M. Z. Mao., Improving the speed of neural networks on CPUs," in Proceedings of the Deep Learning and Unsupervised Feature Learning NIPS Workshop, Granada SPAIN,(2011).

[34] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan., Deep Learning with Limited Numerical Precision, Journal of Machine Learning Research, 37 (2015).

[35] S Chetlur et al., cuDNN: Efficient Primitives for Deep Learning. (2014).[Online]. https://arxiv.org/pdf/1410.0759.pdf

[36] A. Delong. Practical Guide to Matrix Calculus for Deep Learning,http://www.psi.toronto.edu/~andrew/papers/matrix_calculus_for_learning.pdf [Online]

[37] Baoyuan Liu, Min Wang, H. Foroosh, M. Tappen, and M. Penksy., Sparse Convolutional Neural Networks, in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, (2015) 806-814.

[38] C. Ionescu, O. Vantzos, and C. Sminchisescu., Matrix Backpropagation for Deep Networks with Structured Layers, in IEEE International Conference on Computer Vision (ICCV), Santiago, (2015) 2965-2973.

[39] Y. Zhang and S. Zhang., Optimized Deep Learning Architectures with Fast Matrix Operation Kernels on Parallel Platform, in IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, (2013) 71-78.

[40] Ciresan, Dan, Ueli Meier, and Jürgen Schmidhuber., Multi-column deep neural networks for image classification, In IEEE Conference on Computer Vision and Pattern Recognition (2012) 3642-3649.

[41] Ciresan, Dan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jurgen Schmidhuber., Flexible, High-Performance Convolutional Neural Networks for Image Classification, in International Joint Conference on Artificial Intelligence, (2013) 1237–1242.

[42] Lawrence, Steve, C. Lee Giles, Ah Chung Tsoi, and Andrew D. Back., Face Recognition: A Convolutional Neural Network Approach, IEEE Transactions on Neural Networks, 8(1) (1997) 98-113.

[43] Russakovsky, O., Deng, J., Su, H. et al., ImageNet Large Scale Visual Recognition Challenge. Int J Comput Vis, (115) 2015.