Parallel Programming CS-375 Project Final Report

1st Shams Arfeen
sa05169@st.habib.edu.pk

2nd Nofil Haroon
mh04314@st.habib.edu.pk

3rd Muhammad Sabihul Hasan
mh04387@st.habib.edu.pk

***Abstract*—Deep learning applications have been at the heart of the modern HPC industry and research sectors. At the core of it are the parallel architectures, their compilers and parallel libraries. This research deals with prediction function in deep neural nets, an "embarrassingly" parallel problem, and its parallel implementations with OpenMP and OpenACC libraries along with MPI based cluster computing approach.**

***Index Terms*—parallel programming, deep learning, MPI, OpenMP, OpenACC, GPU computing, cluster computing**

I. Introduction

In machine learning, pushing data points through a network is the task of making prediction out of a given input vector. This vector, say an array of floats, is passed through all the layers of neurons up to the output layer which will represent the answer of its prediction. In face detection for instance, the input layer may be pixel values of an image, and as for the prediction result, the value in an output neuron may represent probability of the image matching to a particular person.

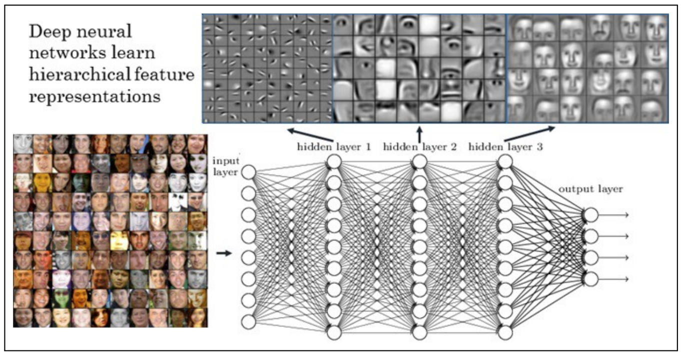

Fig. 1. Face Detection with Deep Learning

This mechanism can be simplified further by taking a feed-forward network with no cycles and one which has no connection that skips any layer. If further it is assumed that the network is fully-connected, then calculation of subsequent layers in the network is just taking a dot product of previous layer V_L with the weight matrix of interconnecting edges W_L.

$$V_{L+1} = W_L . V_L$$

Although a layer cannot be calculated before its previous layers, there is excessive parallelism that can be exploited within individual layers. In the coming sections, we shall present various different approaches to parallelize this task, specifically using OpenMP, OpenACC and MPI libraries along with a combined approach.

II. Serial Implementation of a FeedForward Network

A. Implementation Details

The generic algorithm is repeatedly taking for each layer its weighted dot product and then applying an activation function of some kind. We are using **tanh()** function of **math.h** header although, other sigmoids can also be used. The total number of layers here is referred to as "depth" of the network whereas the "breadth" of the network is the number of neurons in each layer.

The simplified version of **feedforward** function consists of 3 nested loops; **L1**, **L2** and **L3**. The first loop k iterates over individual layers, starting from input layer at $k = 0$ then intermediate layers up to the final output layer. The nested loops of i and j form the SAXPY form of dot product which are responsible for multiplication of $(k-1)^{\text{th}}$ vector and weight matrix to update the k^{th} layer.

B. Serial Code Listing of Prediction function

```
void feedforward() {
    // L1
    for ( int k=1; k < DEPTH ; k++ ) {
    // L2
    for ( int i=1 ; i < BREADTH ; i++ ) {
    // L3
    for ( int j=0; j < BREADTH; j++ ) {
    neuron[k][i] += weight[k-1][j][i] * neuron
        [k-1][j];
    }
    neuron[k][i] = tanh( neuron[k][i] );
    }
    neuron[k][0] = 1;
    }
}
```

We hardcode one neuron in every layer to value 1 to ensure our network can approximate functions which may not pass through origin. The performance results of the serial code are summarized in the following graphs.

1) *Machine Specifications:*

- Processor : Intel(R) Core(TM) i3 CPU 540 @ 3.07GHz 3.06 GHz
- RAM : 6.0 GB
- Operating System : 64-bit Windows 10

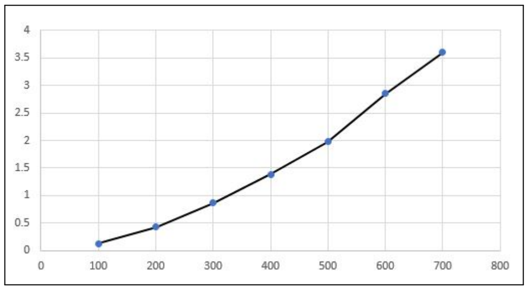

Fig. 2. Time in sec (Y) Vs. Breadth of network (X)

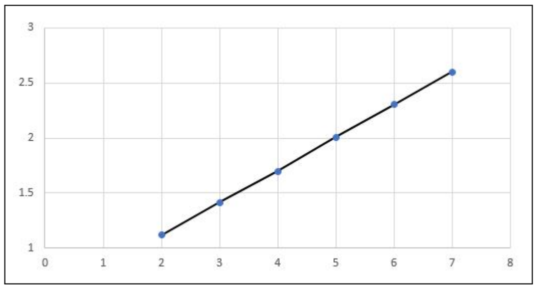

Fig. 3. Time in sec (Y) Vs. Depth of network (X)

C. Interpretation of Results

As can also be seen from code, the relation from graph shows that breadth is proportional to square of time while the depth is directly proportional. Therefore, our target will be to focus on parallelizing breadth as it is our main bottleneck here.

III. Parallelization of code using OpenMP Directives

A. Implementation Details

The approach here is to distribute the iterations of SAXPY form of loop structure in the dot product calculation. In order to spawn OpenMP threads, the three groups are allocated to a threaded block using the ***omp parallel*** directive. The partitioning of data into threads is controlled in the external loop in each of the 3 groups of nested loops by the thread ID. The data range in the external loop is set by controlling the value of the iterator **i**.

B. Performance Results and Explanation

To determine the performance of the parallelized code, the code is executed for multiple threads (2, 4, 8, 16) on the following processor:

1) Machine Specifications:

- Processor : AMD Ryzen 5 3600 6 cores 12 threads (CPU @ 3.60 GHz)
- RAM : DDR4 ADATA (8 + 8) GB
- Operating System : Ubuntu 20.04 LTS

To ensure consistency, the parameters(no. of neurons, no. of layers, input/output neurons) for the code were the same for each execution. The time taken for the code to execute using multiple threads is displayed below:

TABLE I
Time Taken by N threads

No. of threads	Time Taken (seconds)
1	**4.980**
2	**4.160**
4	**2.789**
8	**2.054**
16	**1.231**

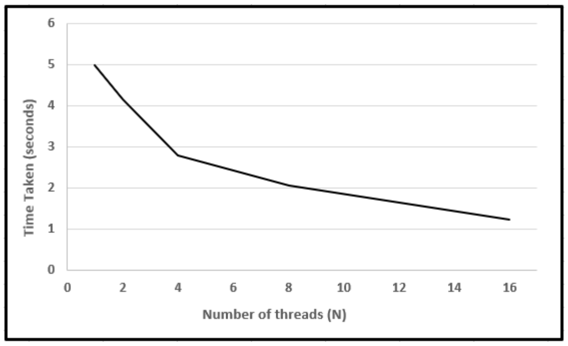

Fig. 4. Time Taken by N threads

The results and the graph above show that, as per our expectation, the total execution time of the code is the least for the code that utilizes the most number of threads. Calculating the speed up of using N threads using the following formula:

$$speedup = \frac{sequential\ time}{parallel\ time}$$

TABLE II
Speedup of N threads over a single thread.

No. of threads	Speedup
2	**1.20**
4	**1.79**
8	**2.42**
16	**4.05**

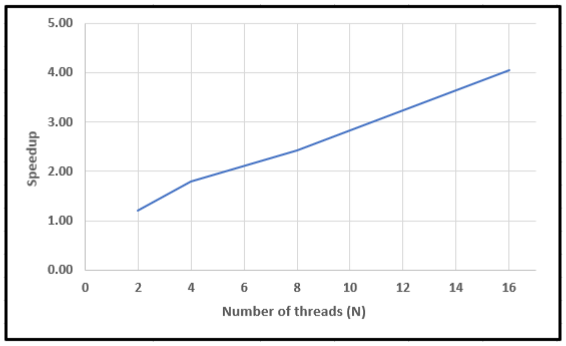

Fig. 5. Speedup of N threads over a single thread.

IV. Parallelization of code using OpenACC Directives

A. Implementation Details

An important attribute of our parallel algorithm is that there is no thread diversion and this property makes it exploitable with an SIMD based GPU. The strategy here is to upload the arrays on the GPU memory just when they are initialized and applying ***acc parallel loop*** directives around parallelizable L2 loop. In theory, this should only be efficient for a large problem size as the GPU cores are of low clock rate and benefit only on a long term parallelism.

B. Performance Results and Explanation

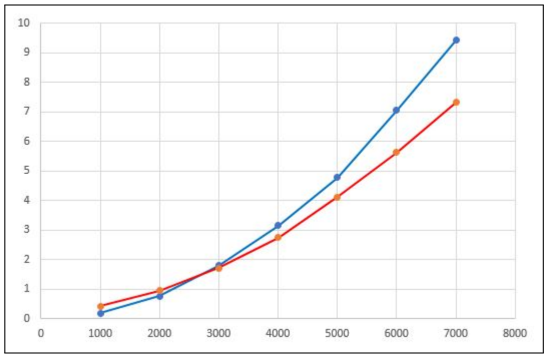

Fig. 6. OpenACC (Orange) Vs. Serial (Blue) Execution Time

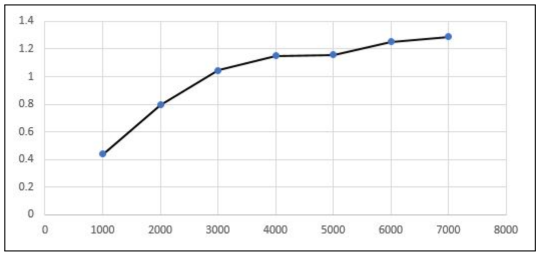

Fig. 7. OpenACC Vs. Serial Speedup against Breadth (X)

1) Machine Specifications:

- Processor : Intel(R) Xeon(R) CPU @ 2.20GHz
- RAM : 13.0 GB
- Operating System : Ubuntu 18.04.5 LTS
- GPU : NVIDIA Tesla T4

TABLE III
Data size Vs. Execution time

Breadth	Serial Time	OpenACC Time
1000	**0.184**	**0.42**
2000	**0.749**	**0.942**
3000	**1.784**	**1.704**
4000	**3.152**	**2.735**
5000	**4.767**	**4.114**
6000	**7.039**	**5.618**
7000	**9.415**	**7.315**

V. Parallelization of code using MPI Broadcast

A. Implementation Details

This approach involves dividing contiguous blocks of iterations in the SAXPY dot product form among different MPI processes. We divide the iterations contiguously as the **MPI_Bcast()** is capable of synchronizing contiguous elements in memory of different MPI processes. Doing so would result in calculation of dot product vector in contiguous chunks in different nodes. In order to combine all the chunks, the processes would then call **MPI_Bcast()** thereby, synchronizing the entire dot product result and then continue to the next layer.

B. Performance

1) Machine Specifications:

- Processor : Intel(R) Core(TM) i3 CPU 540 @ 3.07GHz 3.06 GHz
- RAM : 6.0 GB
- Operating System : Ubuntu 64-bit

TABLE IV
Execution time Vs. No. of MPI processes

No. of processes	Time Taken (seconds)
1	**1.28**
2	**1.32**
3	**1.42**
4	**2.03**
5	**1.89**

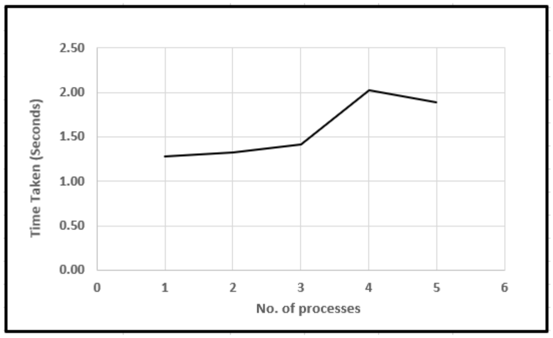

Fig. 8. Execution time Vs. No. of MPI processes

C. Interpretation of Results

As can be seen from table, the efficiency is not increasing and this is due to low number of nodes in the network. We predict that time will show steep decrease with more nodes in the MPI cluster network. A more suitable environment would be a bigger network to exploit parallelism as the number of processes are increased.

VI. A Combined Implementation: MPI and OpenMP

A. Implementation Details

Although we were able to divide the task among cluster nodes in MPI implementation, there is still a bottleneck of performance within individual nodes since they are running serial execution. This approach uses the fact that even after dividing tasks among different nodes, each MPI process is still running in serial and therefore, is not utilizing all of its processing power. By further using **OpenMP parallel** directive against the outer loop L2 of SAXPY form, the iterations can be divided into threads giving even more parallelism.

B. *Performance*

1) Machine Specifications:

- Processor : Intel(R) Core(TM) i3 CPU 540 @ 3.07GHz 3.06 GHz
- RAM : 6.0 GB
- Operating System : Ubuntu 20.04 LTS

TABLE V
EXECUTION TIME VS. NO. OF MPI PROCESSES (WITH OPENMP DIRECTIVES).

No. of processes	Time Taken (seconds)
1	**1.32**
2	**1.35**
3	**1.44**
4	**1.54**
5	**1.65**

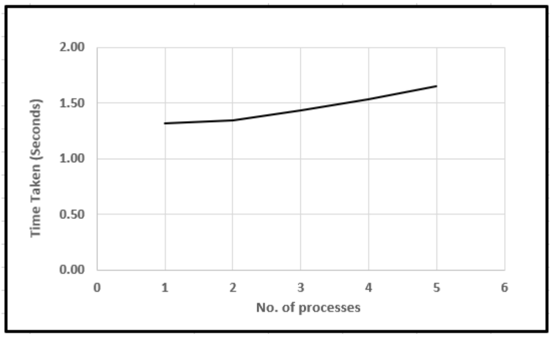

Fig. 9. Execution time Vs. No. of MPI processes (with OpenMP directives).

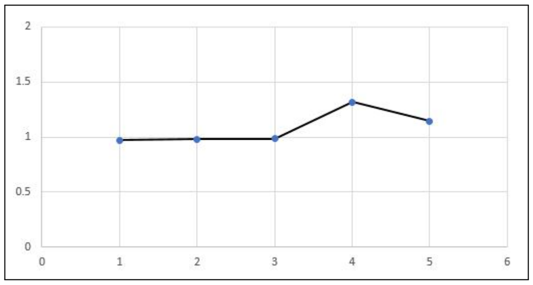

Fig. 10. Speedup of Combined Approach Vs. MPI

C. *Interpretation of Results*

The speedup graph proves that combined approach is better than simple MPI broadcast based approach. This was due to better parallelism within individual MPI processes which in turn spawned multiple OpenMP threads. We predict better speedup for a larger cluster network as our current network is only one computer running all the MPI processes.

CONCLUSIONS

Important conclusions can be drawn from our experiments. Firstly, we presented parallel approaches to address the bottlenecks of prediction in neural networks of large breadth. This can be overcome with OpenMP and OpenACC directives in SAXPY form and tiling based implementations. Secondly, we presented a parallel approach of MPI based implementation which divides iterations of dot product in SAXPY form and then broadcasts to synchronize the results. Lastly, we gave a combined approach of OpenMP and MPI which exceeded the simple MPI approach in performance and we predict that it will show a better speedup when run in a larger cluster network.

REFERENCES

[1] M. J. Quinn, "Parallel Programming in C with MPI and OpenMP," Oregon Stale University, 2004.
[2] M. Nelson, "Neural Networks and Deep Learning," Oregon Stale University, December 2019.