annCharm++: Artificial Neural Network Implementation using Charm++

Luis Remis, Aditya Deshpande, Boyan Li, Shangquan Wu
Computer Science Department
School of Engineering
University of Illinois at Urbana Champaign
{remis2, ardeshp2, boyanli3, swu55}@illinois.edu

***Abstract*—In ANN training procedure, the weight of each neuron is updated using a back-propagation method. Back-propagation makes use of the difference between output and target to modify the weight of neuron backwards. We aim to explore the potential parallelism underlying in this seemingly sequential training process using Charm++, in a seek for obtaining a trained neural network in short time and in a scalable design. After describing the methodology followed by the authors, this paper shows promising results when it comes to execution time and scalability, with a reduction in time by about $6\times$ when moving from 1 to 8 processors.**

I. Introduction

Artificial Neural Networks are used for many tasks in Computer Vision [1], Machine Learning and Speech Recognition [2]. With the advent of Deep Learning, the size of these networks is only getting bigger and they are difficult to train on a single CPU commodity system. Currently parallelism of GPUs is used to train these large neural networks. In our project, we wish to explore if we can leverage the distributed object model of Charm++ [3] to efficiently implement ANNs. The most basic neural network has 3 Layers: (i) 1000 Input Layer Units, (ii) 500 Hidden Layer Units, (iii) 10 Output Units (one per digit). The input is 900 (=30 $\times$ 30, image size) dimensional vector. Thus the total weights are: $900 \times 1000 + 1000 \times 500 + 500 \times 10 \sim 1.3$million. The training time for this achieved earlier with GPU parallelism is 24hrs [4]. There are also some MPI implementations of the same problem [5].

We believe that this is a good setting to leverage parallelism through Charm++ and achieve better performance.

II. Basic Algorithm

In our work we consider fully connected neural networks, with one input and output layer and one or more hidden layers. As you can see in Figure 1, it has L layers, the 1^{st} layer ($l = 1$) is **input layer**, the layers after that $l = 2 \cdots L-1$ are **hidden layers** and last layer ($l = L$) is the **output layer**.

Each edge represents a weight that needs to be applied to the data transferred between the two neurons. The neural network starts with randomly initialized weights and when the training algorithm (here we use backpropogation) converges, we will have learnt the correct weights for our classification problem. There are two phases to the training algorithm: (i) **Forward Phase** and (ii) **Backward Phase**.

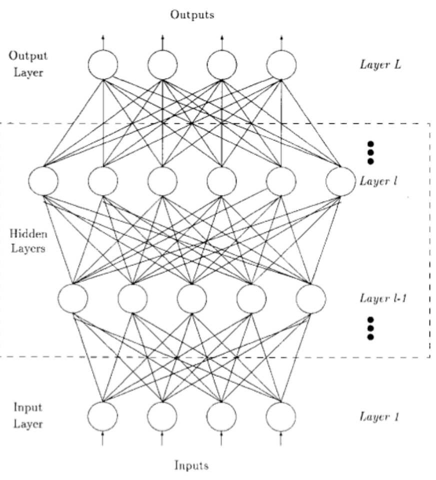

Fig. 1. Illustration of Fully Connected Neural Networks (from [5]).

A. Forward Phase

- The i^{th} neuron in input layer sends the data corresponding to dimension i in the input feature vector to next layer.
- The i^{th} neurons in the next layers (l), wait for recieving data from all j neurons in previous layer ($l-1$) and then apply the weight/sigmoid and send it to next layer.
- The output layer, once it recieves data, computes the error. It also computes weight update terms, which are to be sent back into the network in backward phase.

The data computation (a_i's /activations) at each neuron can be captured in the equation:

$$a_i(l) = F(\sum_{j=1}^{n_{l-1}} w_{ij} \times a_j(l-1) + \theta), F : non-linearity$$

Also, for output layer we compute weight update term (δ's).

$$\delta_i(L) = (oracle_i - a_i(L)) \times (1 - a_i(L))$$

B. Backward Phase

- Now, $\delta's$ in layer l, depend on $l+1$. Note that δ_L are already computed, so we start from δ_{L-1} till δ_1.
- Weights at each neuron are update using δ, $w_{ji} = w + \delta_j \times a_i(l)$ at each neuron.

- When we reach first layer, all weights are updated and we can resume forward phase again.

Delta computation uses the following equation:

$$\delta_i(l) = [\sum_{j=1}^{j=n_{l+1}} \delta_j(l+1)w_{ji}(l)] \times [a_i(l) \times (1-a_i(l))]$$

III. Decomposition to Chares

We use a three-level structure to decompose neural network for Charm++ implementation. The top level is a Layer, the second level is a chare and the bottom level is a neuron. In a neural network, there is an input layer, an output layer and several hidden layers. In our implementation, each Layer is defined as a chare array and we store proxies for each layer to communicate between them. Note that, we could have used a 2D-chare array instead, but since each layer may have different neurons, this leads to wasteful allocation of few chares. Also, we conclude that it would be easy to program and structure the application if we had separate chare arrays for every layer rather than a sparse 2D-chare array. In our application, Every chare contains certain number of neurons. The number of neurons is a parameter to be determined and will define our granularity, which will allow us to experiment about performance according to the granularity, as it will be described later.

Based on the description above, neurons on a specific layer will be in different chares, where each chare will be part of the Layer (a chare array). In other words, each chare contains a number of neurons, called neurons groups, and that chare is responsible for all the operations over this neurons. Moreover, for neurons in different layers, none of them will be in the same chare. Since errors and activation value are sent between layers, this characteristic makes it possible to use a Group to send and receive errors and activation values in batch, which will significantly increase the communication performance. This feature will come as part of future implementations.

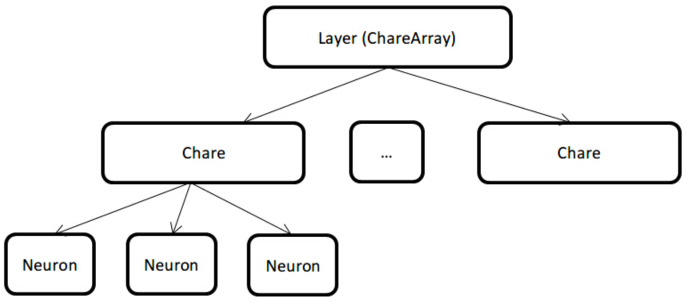

Fig. 2. Granularity

IV. Batch Training vs. Single Example Training

Batch methods processes large amounts of data instead of one small set before executing the following step. Batch processing can benefit from parallel processing because different steps that do not have dependencies can run on different together. Since each image is processed separately, we reduce synchronization required at the end by accumulating the error several times before back propagating it.

Due to added synchronization, the overhead time of single processing is larger than the batch processing. In single example training in ANNs, each image can be processed only after the previous one is done and weights have been updated (as shown by many back arrows in left part of Figure 3). When using batch processing, we club the updates together and send them only after the batch completes (as shown by single back arrow in right part of Figure 3). Even if this may affect the speed of convergence of the ANN, by having a large set of images for the training we may mitigate this effect.

The idea of using this architecture for training the neural network meant performing a cumulative calculation of the error, which can introduce errors and though incrementing the necessary time for the network to converge. For our first approach to the problem, we decide to code our application with this tool in mind with a parameter which sets the frequency for the backward propagation but we did not implement the function to perform this cumulative error calculation.

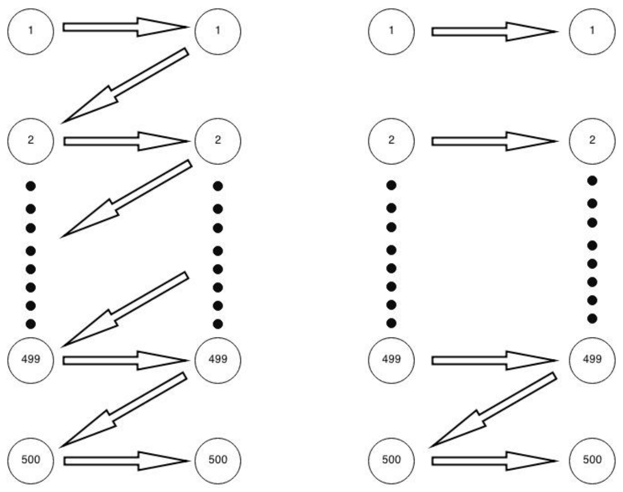

Fig. 3. Low synchronization (backward arrows) required in Batch Training.

V. Simplified Architecture of the Application

The input parameters of our application are:

Neurons in Input layer (900 for our digits dataset)
Neurons in Output layer (10)
Hidden Layers (1-3)
Neurons Per Hidden Layer (100-500)
Neurons Per NeuronGroup (grainsize, 2-64)

Chare array (size = # Neurons in Layer / # Neurons Per Group) for each layer is created, and its proxy stored in global array. We use a reduction to identify end of constructor, run method of each chare array called thereafter.

We identified 3 different behaviors in the existing layers. Input layers, which only send information, middle layers which both send and receive information, and output layers, which only receive information. This behavior is valid during the forward phase. For the backward phase, we have a similar situation, but with output layer only sending and input layer only receiving. Charm++ provides all the tools to implement this flow control. We make use of Structured DAGGER to encode the both the forward and the backpropogation training procedure, controlling the life cycle of a neuron. Charm++ features taught in course were be used to efficiently perform ANN training and testing phases.

An overview of our algorithm is best seen in the run method at https://github.com/luisremis/annCharm/blob/master/neuralNetwork.ci.

VI. Results

Our code can be found at https://github.com/luisremis/annCharm/. We first started with 200 different images for training and 100 images for testing. These images only contained zeros and ones. In this way, we could check the correctness of the algorithm.

After preparing the data set for its processing in and tuning parameters in our implementation, we started running test to see and analyze the convergece, which means to see if we correctly predict whether an unseen image (during training): is zero or a one. By training with a interleaved set of 20 zeros and 20 ones, repeated 100 times (though meaning a set of 2000 images in total for training), the ANN proved to be working correctly by converging after 1500 iterations, as it can be seen in Figure 4. This figure shows the last 1700 iterations, where we see the accuracy of our predictions. The blue line corresponds to the value of output neuron corresponding to zero and orange line for the output neuron corresponding to one. We can see that the values rise and fall after 20 images, as per our data and thus, we are correctly classifying the input digits.

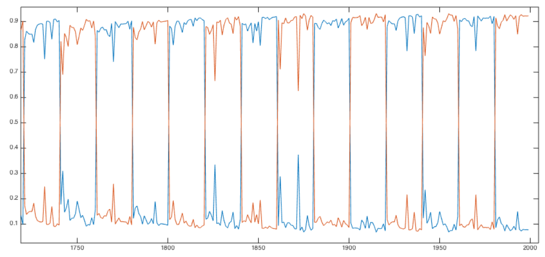

Fig. 4. Last 1700 iterations of the 2000 images data set

These results allowed us to verify the correctness of the implementation, so we could continue with the analysis of performance and scalability.

A. Changing the GrainSize

In order to understand the behaivour of our implementation when changing the grain size, and after making sure of the correctness of the output, we carried out test using all fixed parameters and moving the grain size from 2 to 64 neurons per chare.

The parameters for these test were:

Neurons in Input layer = 1024
Neurons in Output layer = 64
Hidden Layers = 2
Neurons Per Hidden Layer = 128
Neurons Per NeuronGroup = { 2, 4, 8, 16, 32, 64 }

Figure 5 depics how the execution time decrease in our application when varying the grain size. Because of the nature of this highly communicative application, the utilization of the processors tends to be low when assigning a small number of neurons per chare, meaning higher execution time. As the work done by each neuron may increase of decrease depending of the activation function, the tuning of this parameter (neurons per chare) allows the application to adapt easily and have better utilization according to the needs.

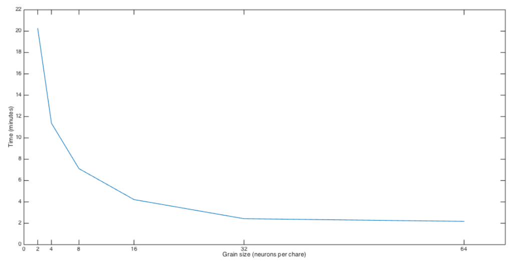

Fig. 5. Changing the Grainsize

B. Changing the number of processors

We run our application with the same parameters as above, using 64 neurons per chare, with different number of processors to analyze how the application scale. The results were better than the expected, performing almost six times faster when comparing the execution time using 1 and 8 processors. These results can be seen in Figure 6. Further test will be made with larger neural networks and more number of processors, but permissions to run in more than 16 processors are needed for these test.

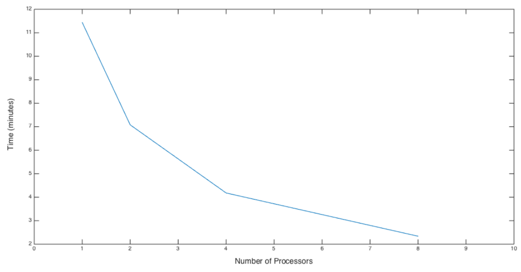

Fig. 6. Changing the number of processors

C. Bigger numbers range

After obtaining good results and behaviors as predicted, we could now continue with a entire data set (MNIST of 60,000 images) containing numbers from 0 to 9. From initial test we learned that having a sequence of 20 zeros and 20 ones produced good results, so we changed the larger set of images to follow the same sequence, but using numbers from 0 to 9. Of course, for training the network to be able to recognize numbers from bigger range, we need more iterations. Figure 7 shows the convergence of the network as the program run. The blue line represents the average percent of the positive predictions of the last 100 iterations, and the red line represents the average accuracy of the correct predictions. After 20000 iterations, the neural network is very well trained, and the accuracy of the predictions continue to grow as iterations move forward. It took us only about $4hrs$ to run this experiment (on $60K$ images dataset).

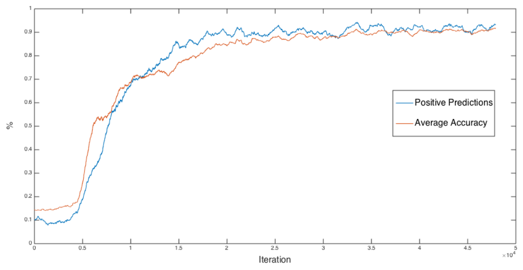

Fig. 7. 60K iterations

D. Projections Analysis

With Projection, we got a closer look at how our application behaves. We generate profiler images of utilization, communication, and time overview with different grain size (10 and 60 neurons per chare) and with different number of processors (4 and 8). First we get the utilization images when application runs in grain size of 10 with 4 and 8 processors. We did not apply any load-balance strategy. As clearly shown in Figure 8 and Figure 9, the workload is naturally balanced among all processors. We believe this is a result of the uniform nature of computations performed by each chare and effectiveness of the Charm RTS migratibility. Overhead in the first processor is caused by the method which executes the loading file from disk. Based on our design, the input layer is allocated in processor 0, which reads input file from disk.

As shown in Figures 8 and 9, with more processors being used, the computation load per processor decreases (denoted by the colored part in lower bar) but idle/communication increases (denoted by the white part). With more processor being used, the computation time is saved but inter-processor communication increases. Large portion of idle is a concern to the nature of the algorithm rather than our design only. Since, ANN is a communication heavy algorithm. Computation of value and error are simple but sending errors and values between neurons takes up most of the time.

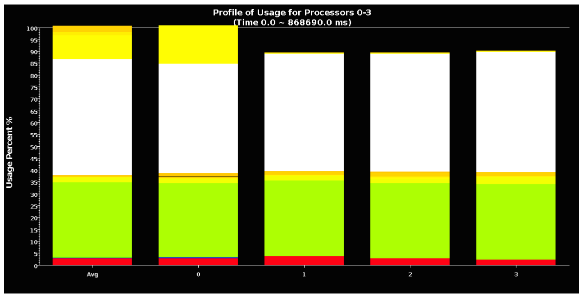

Fig. 8. Usage Profile for 4 processors

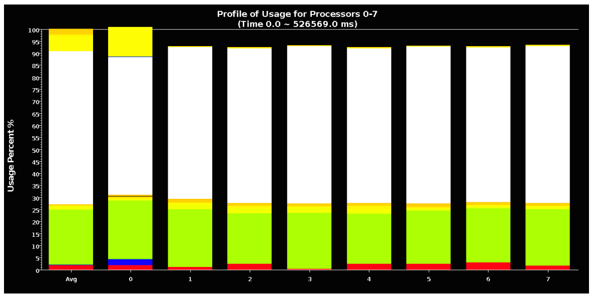

Fig. 9. Usage Profile for 8 processors

Since our application is communication heavy, we next generate communication profiles in projection. These are shown in Figures 10 and 11. Green block denotes messages sent by sendBackward method. Red block denotes messages sent by sendForward method. Blue block denotes messages sent by broadcast method called internally. In each bar of the image, the message number of broadcast equals the sum of messages sent by sendBackward method and sendForward method because both methods are broadcast. The first processor has less messages to send because it does not send back errors. As we double the number of processors by 2, we see that the fraction of messages handled per processor reduces by 2. This again shows that our application achieves good load balance.

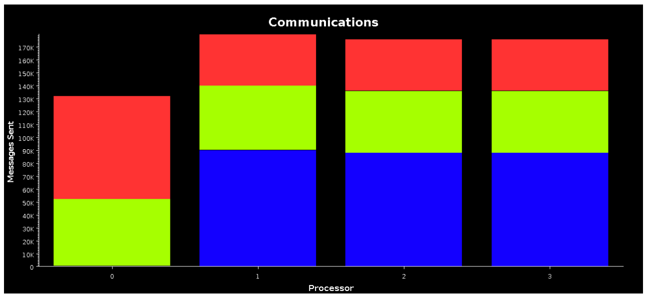

Fig. 10. Communication Profile for 4 processors

We realized that the grain size (number of neurons per chare) will influence the performance of our application. We start changing it in order to get time overview images in projection and verify the behavior. We run the application with

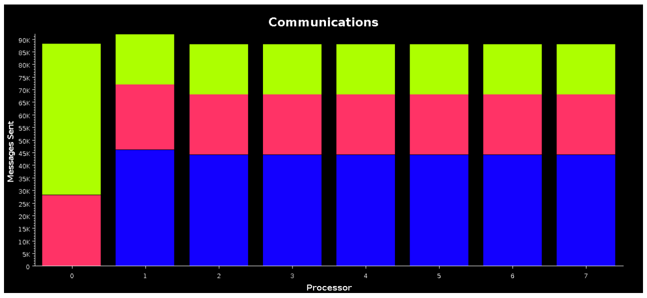

Fig. 11. Communication Profile for 8 processors

grain size 10 and 60 with 4 processors. It clearly reveals that the utilization decreases with assigning more neurons in each chare. Since with more neurons in a chare, less chares are used. Since neurons communicate with each other by the chare they reside, communication decreases with less chares (white portion in Figures 12 and 13).

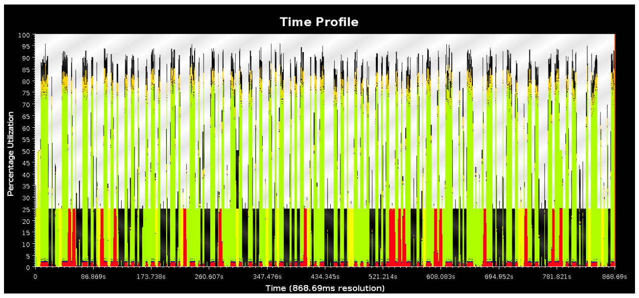

Fig. 12. Time Profile for 4 processors and 10 neurons per chare

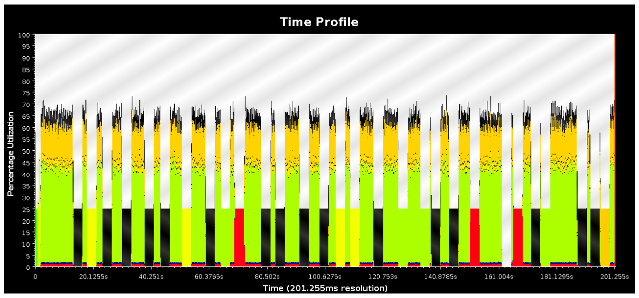

Fig. 13. Time Profile for 4 processors and 60 neurons per chare

VII. Conclusion and Future Work

In our work, we used features in Charm++ to perform parallel training for Artificial Neural Networks. We demonstrated results using a network of about a $\sim$ 1.3 million parameters and on dataset of $60,000$ images (each image being a 900-dim vector). In this large benchmark, we required only 4hrs to obtain the result and also, we were able to train the network with good accuracy. We analyzed our implementation using the Projections tool of Charm++ and showed that our application achieves load balance i.e. the computation, as well as the communication load distributes itself evenly as we change the number of processors. We showed the effect of changing the grainsize parameter using time overview in projection, increasing the grainsize to a certain extent helped overlap computation and communcation by a chare and thus reduced the overall computation time.

As future work, we have not yet fully tested the batch training paradigm and we hope that it will offer more gains in our runtime (at some compromise in accuracy). We can reduce some of our broadcasts during backward phase to communicate smaller data to specific chares and see the impact on the execution time. We believe that different layers should have different granularity. Since output layer has fewer neurons, with the same granularity as other layers, it may end up being the sequential bottleneck. We need to address this issue in our implementation, which was discovered via projections.

Acknowledgment

The authors would like to thank Eric Mikida for mentoring this project.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] H. Lee, Y. Largman, P. Pham, and A. Y. Ng, "Unsupervised feature learning for audio classification using convolutional deep belief networks," in *Advances in Neural Information Processing Systems 22*, 2009, pp. 1096–1104.

[3] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108. [Online]. Available: http://doi.acm.org/10.1145/165854.165874

[4] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big simple neural nets excel on handwritten digit recognition," *CoRR*, vol. abs/1003.0358, 2010. [Online]. Available: http://arxiv.org/abs/1003.0358

[5] S. Suresh, S. Omkar, and V. Mani, "Parallel implementation of back-propagation algorithm in networks of workstations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 1, pp. 24–34, 2005.