SHM-Learn-Dist: An optimized GPU-Powered C++ Distributed Deep Learning Library

Souham Biswas
Illinois Institute of Technology
sbiswas7@hawk.iit.edu

Abstract

To handle training over large datasets, distributed training approaches are very popular. There are mainly two types of distributed training – data-parallel & model-parallel. While data-parallel approaches are easier to implement and include greater support, model-parallelism is usually used to train large neural nets too big to fit on one GPU. In this paper, the workings of the SHM-Learn-Dist library is described which yields training performance boosts in a distributed setting. Support currently exists for data-parallel training.

1. Introduction

Data parallel training is particularly useful in cases where there is huge diversity in data or the mapping to be learnt is complex. The main idea behind data parallel training involves the following two modes –

- Gradient Averaged Data Parallelism
- Weight Averaged Data Parallelism

We shall discuss about both forms of data parallelism and give some pointers about the use cases of each.

Model parallelism is especially useful when we are training very big neural networks of the order of billions of parameters. Here, the model parameters may be distributed across different nodes. It is illustrated in Fig. 1.0.

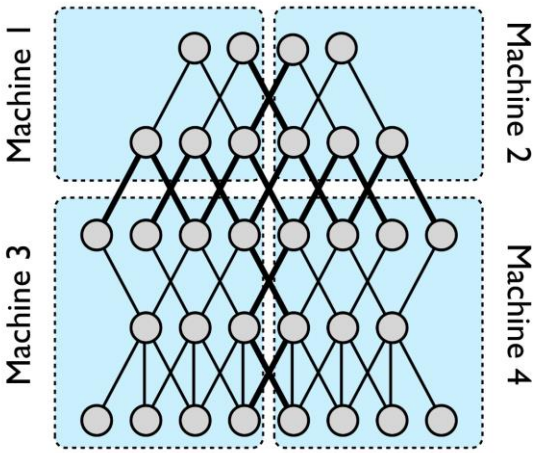

Fig. 1.0 – Model Parallelism illustration[1]

As is clearly evident, there is significant communication involved within every layer here.

The organization of the paper shall be in the following way –

- Section 2 shall explore the 2 different types of data parallelisms mentioned previously.
- In section 3, we delve in to understanding the fundamental effects of data parallelism on training by observing the parameters of a single neuron which is subject to distributed training.
- Section 4 elicits the exact mathematical explanation of the memory-realignment algorithm which contributes greatly to the speedup and presents the GPU kernels and their invocation patterns.

- Section 5 presents the results of some parameter searches for finding the optimal distributed setting to train on the CIFAR-10 dataset are presented.

- In the 6th section, performance and scaling plots of distributed training experiments with SHM-Learn-Dist are presented and compared with other toolkits.

- Finally in section 7, the future work is mentioned which is planned.

2. Types of Data Parallelism

Data parallel trainings are of 2 types namely -

1. Gradient Averaged
2. Weight Averaged

The main difference between these two methods is that in gradient averaging, every worker gets the same set of weights from the parameter server, computes gradients which are averaged element-wise by the parameter server and applied on the same set of weights which each worker started with. The update equation is presented in Eq. 2.1 –

$$w = w - \eta\left(\frac{1}{N_w}\sum_{i=1}^{N_w} g_i\right)$$

Eq. 2.1 Weight update equation in gradient averaged data parallelism.

Here, N_w is the number of workers and g_i is the gradient from the i^{th} worker; η is the learning rate.

In weight averaging, every worker starts with its own set of weights which vary between workers. The workers compute *and apply the gradients* to get new weights which are sent to the parameter server. The parameter server averages these weights element-wise and sends them back to the workers who continue the process. The update equation for the weight averaged technique is presented in Eq. 2.2 –

$$w = \frac{1}{N_w}\sum_{i=1}^{N_w}(w_i - \eta g_i)$$

Eq 2.2 Weight update equation in weight averaged data parallelism

Upon simplifying, it becomes Eq. 2.3 –

$$\Rightarrow w = \left(\frac{1}{N_w}\sum_{i=1}^{N_w} w_i\right) - \eta\left(\frac{1}{N_w}\sum_{i=1}^{N_w} g_i\right)$$

Eq. 2.2 Simplified weight update equation in weight averaged data parallelism

Here, w_i is the set of network parameters of the i^{th} model; the remaining symbols have the same meanings as explained previously.

The main difference intuitively observable here is that the weight averaged technique promotes diversity in the weights along with the gradients; whereas in the gradient averaged technique, only the gradients are averaged.

3. Single Neuron Distributed Training

An experiment script was written which simulates distributed training of one single neuron with a sigmoid activation function which is trying to learn the following simple function as stated in Eq. 3.1.

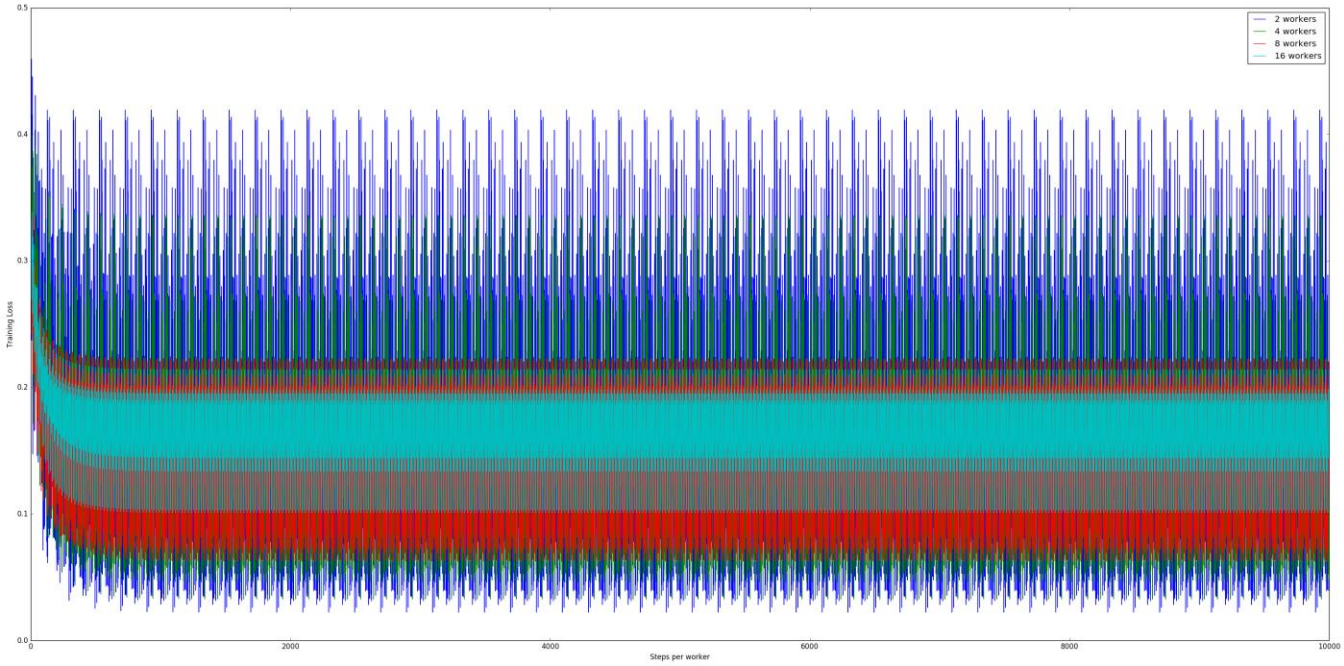

Fig. 2.1 Single Sigmoid Neuron distributed training loss plots over 2, 4, 8 & 16 nodes.

$$f(x) = \begin{cases} 0 \; if \; x < 0.5 \\ 1 \; if \; z \geq 0.5 \end{cases}$$

Eq. 3.1 Function subject to distributed learning by a single sigmoid Neuron

The distributed training was done over 10,000 steps per worker for different numbers of workers for each data parallel training strategy. The learning rate was kept constant at 0.5. Stochastic Gradient Descent was used as the optimization algorithm with no regularization for simplicity coupled with a sigmoid activation function.

The loss plots so obtained from subjecting the single sigmoid neuron to gradient averaged distributed training are presented in Fig. 2.1.

As is clearly evident, the convergence is slower with more workers; however, the loss is less noisy with more workers which corresponds with our observations from distributed CIFAR-10 experiments presented in section 6.

The noisy loss plot can be attributed to the fact that the function being learnt is too simple to be scaled across multiple compute nodes. Hence, visually illustrating that data-parallelism is most useful on complex mappings and/or on large datasets.

It is not unusual to notice that some optimization algorithms perform better with certain distribution schemes. For example, asynchronous Stochastic Gradient Descent scales better with lesser nodes whereas synchronous Stochastic Gradient Descent scales better with more nodes[2].

4. Mathematical explanation of Memory Re-alignment algorithm

An algorithm to minimize the number of matrix multiply operations by combining the weights and biases efficiently in to a single matrix was

presented in the previous paper[3]. In this section, the exact mathematical explanation of its working and the GPU kernels along with their invocation patterns shall be presented.

Two types of re-alignment procedures are implemented namely `ShiftRight` and `ShiftLeft` which are explained in sections 4.1 & 4.2 respectively.

4.2 ShiftRight Memory Realignment Matrix transform-

In this scheme, the following transform is done, as shown in Fig. 4.2.1 –

$$\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & a_0 & a_1 \\ 1 & a_2 & a_3 \end{bmatrix}$$

Fig. 4.2.1 ShiftRight matrix transform illustration

Recall[3] that this when represented in the row-major memory layout as in the case of NVIDIA GPUs, actually looks like what is shown in Fig. 4.2.2 below –

$$[a_0, a_1, a_2, a_3] \rightarrow [1, a_0, a_1, 1, a_2, a_3]$$

Fig. 4.2.2 ShiftRight transformation actual memory layout illustration

The way this is done has been explained at a high level in [3]; here, the actual implementation details are presented. Let us make the following assumptions about the variable names used to explain the approach –

- `rows` – The original number of rows in the input matrix.
- `cols` – The original number of columns in the input matrix.
- `d_mat` – GPU pointer to the memory location holding the original matrix in a row-major format as depicted on the LHS in Fig. 4.2.2.
- `d_helper` – GPU pointer to the memory location to hold the values which may be damaged due to Write-After-Read race conditions[3].

Firstly, we initialize the `d_helper` pointer to a memory location with a size of $\frac{rows(rows-1)}{2}$. Next, we populate the `d_helper` array with values at the affected indices[3]. Ideally, we'd require $\frac{rows(rows-1)}{2}$ threads to do this job as each thread simply performs a copy operation. However, to satisfy the constraint of launching threadblocks with number of threads being a multiple of the GPU warp size[4] (32), the number threadblocks of size $4 \times warp\_size$ to be launched will be as shown in Eq. 4.2.1.

$$num\_threadblocks = \left\lceil \frac{rows(rows-1)}{2 \times (4 \times warp\_size)} \right\rceil$$

Eq. 4.2.1 Number of threadblocks needed to populate `d_helper`.

And we have Eq. 4.2.2 -

$$threadblock\_size = 4 \times warp\_size$$

Eq. 4.2.2 Size of each threadblock to populate `d_helper`

This operation in done by the following GPU kernel–

```
__global__ void ShiftRight_PopulateHelper_GPUKernel(float
*d_mat, float *d_helper, int damaged_elems, int rows, int
cols) {
  int idx = (blockDim.x * blockIdx.x + threadIdx.x) %
damaged_elems;
  int i = floor(0.5f * (sqrt((float)1 + 8 * idx) - 1.0f)) +
1;
  int j = idx - i * (i - 1) / 2;
```

```
  int read_idx = j + i * cols;
  d_helper[idx] = d_mat[read_idx];
}
```

Mapping each thread to its respective element in the original matrix which its copying to the `d_helper` array, if `idx` is the global threadIdx of a particular thread and assuming the 2D index (in row, column form) from where it shall read in the original matrix `d_mat` as (i, j), we have the following relations as shown in Eq. 4.2.3 and Eq. 4.2.4 –

$$i = \left\lfloor \left(\frac{1}{2} \sqrt{1 + 8 \times idx} \right) + 1 \right\rfloor$$

Eq. 4.2.3 Index row of value in `d_mat` from which thread with global index `idx` reads.

$$j = idx - \frac{i(i-1)}{2}$$

Eq. 4.2.4 Index column of value in `d_mat` from which thread with global index `idx` reads.

Notice that the calculation of j utilizes the calculated value of i in the previous step. Finally, mapping them to unrolled 1D indices respective to `d_mat`, the final read index `read_idx` is given by Eq. 4.2.5 –

$$read\_idx = j + i \times cols$$

Eq. 4.2.5 1D unrolled read index in `d_mat` from where `d_helper` is populated by thread at global index `idx`

Invocation of the GPU kernel is done in the following way –

```
ShiftRight_PopulateHelper_GPUKernel << < num_threadblocks,
    threadblock_size >> >
    (d_mat, d_helper, reqd_threads,
     rows, cols);
```

Following this, another GPU kernel is launched which first, naively does a ShiftRight transform as mentioned in [3], destroying some values in `d_mat` in the process. Secondly, the damaged values are repopulated with the values in `d_helper` which was populated in the previous step. The GPU Kernel concerning this operation is presented below.

```
__global__ void ReAlignMemory_ShiftRight_GPUKernel(float
*d_mat, float *d_helper, int total_size, int cols, int
thread_chunk_size) {
  extern __shared__ float read_vals[];
  int shared_mem_idx, read_idx, read_idx_row, write_idx;
  int row_linear_idx = blockIdx.x * cols;
  int read_idx_base = row_linear_idx
    + (threadIdx.x * thread_chunk_size) % cols;
  int row_last_linear_idx = row_linear_idx + cols;
  for (read_idx = read_idx_base; read_idx <
row_last_linear_idx;
       read_idx++) {
    read_idx_row = read_idx / cols;
    shared_mem_idx = read_idx - row_linear_idx;
    if (read_idx >= read_idx_row * (1 + cols)) {
      read_vals[shared_mem_idx] = d_mat[read_idx];
    }
    else {
      read_vals[shared_mem_idx] = d_helper[read_idx - cols *
read_idx_row
        + (read_idx_row - 1)
        * read_idx_row / 2];
    }
  }
  __syncthreads();
  for (read_idx = read_idx_base; read_idx <
row_last_linear_idx;
       read_idx++) {
    write_idx = (read_idx + ceil((float)read_idx / cols)) +
!(read_idx % cols);
    d_mat[write_idx] = read_vals[read_idx - row_linear_idx];
    if ((write_idx - 1) % (cols + 1) == 0) {
      d_mat[write_idx - 1] = 1.0f;
    }
  }
}
```

Here, each threadblock populates it's shared memory space selectively with values from the original `d_mat` matrix and the `d_helper` array based on whether the final write index of the value in the transform could potentially be damaged by the Write-After-Read race condition. Once this is done, the values from the shared memory array are simply written to their new locations in `d_mat` and the 'empty' spaces are padded with 1s. The whole process is illustrated in Fig. 4.2.3.

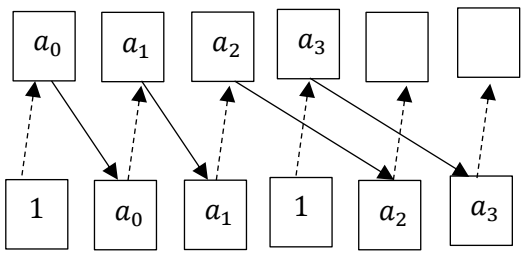

Fig. 4.2.3 Visual illustration of the ShiftRight transform as represented in memory[3].

The operation showed above are in-place in nature. The invocation of the kernels are shown in the code snippet presented below which performs the whole transformation –

```
void ReAlignMemory_ShiftRight(float *d_mat, float *d_helper,
int rows, int cols, int max_threadblock_size) {
  int org_size = rows * cols;
  int reqd_threads = rows * (rows - 1) / 2;
  int threadblock_size = GPU_WARP_SIZE *
GPU_WARP_DISPATCHERS * 2;
  if (threadblock_size > max_threadblock_size)
    threadblock_size = max_threadblock_size;
  int num_threadblocks =
my_ceilf_division_FCLayer(reqd_threads, threadblock_size);
  int thread_chunk_size = my_ceilf_division_FCLayer(cols,
max_threadblock_size);
  ShiftRight_PopulateHelper_GPUKernel << < num_threadblocks,
    threadblock_size >> >
    (d_mat, d_helper, reqd_threads,
     rows, cols);
  reqd_threads = my_ceilf_division_FCLayer(cols,
thread_chunk_size);
  threadblock_size = my_ceilf_division_FCLayer(reqd_threads,
GPU_WARP_SIZE)
    * GPU_WARP_SIZE;
  ReAlignMemory_ShiftRight_GPUKernel << < rows,
    threadblock_size,
    sizeof(float) * cols >> >
    (d_mat, d_helper,
     org_size, cols,
     thread_chunk_size);
}
```

4.3 ShiftLeft Memory Realignment Matrix transform-

In this scheme, the following transform is done, as shown in Fig. 4.3.1 –

$$\begin{bmatrix} 1 & a_0 & a_1 \\ 1 & a_2 & a_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix}$$

Fig. 4.3.1 ShiftRight matrix transform illustration

This, when represented in the row-major memory layout as in the case of NVIDIA GPUs, actually looks like what is shown in Fig. 4.3.2 below –

$$[1, a_0, a_1, 1, a_2, a_3] \rightarrow [a_0, a_1, a_2, a_3]$$

Fig. 4.3.2 ShiftLeft transformation actual memory layout illustration

This is usually carried out in the back-propagation step to compute the derivatives to be back-propagated to the previous layer.

To illustrate, suppose we have a given layer and we are training with a batch size of n. Let us further assume that the layer in consideration has m_o activations feeding to the next layer and takes in m_i activations from the previous layer. Therefore, the derivatives received from the layer in front during back-propagation will have the dimension $n \times m_o$ and the parameter matrix of the layer including the bias[3] will have the dimension $m_o \times (m_i + 1)$.

Let us call the parameter matrix as W (dimension = $m_o \times (m_i + 1)$) and the derivative matrix backpropagated from the frontal layer as D (dimension = $n \times m_o$). It is desired that the derivatives to be backpropagated to the previous layer (for the previous layer to compute its gradients) have the dimension $n \times m_i$. For this to happen, we need to perform the operation $W \cdot D$. However, the dimension of W being $m_o \times (m_i + 1)$ prevents that from happening. Therefore, the +1 due to the extra bias terms needs to be removed. This is done by doing a `ShiftLeft` transform prior to the matrix multiply on matrix W. To explain the process, the various definitions of variable names as described in 4.2 are borrowed here.

To populate the helper array to carry out this transform, a similar approach as mentioned in 4.2 is employed. Mapping each thread to its respective element in the original matrix which its copying to the `d_helper` array, if `idx` is the global threadIdx of a particular thread and assuming the 2D index (in row, column form) from where it shall read in the original matrix `d_mat` as (i, j), we have the following relations as shown in Eq. 4.3.1 and Eq. 4.3.2 –

Name: Souham Biswas **CWID**: A20365242 **Email**: sbiswas7@hawk.iit.edu

$$i = \left\lfloor \left(\frac{1}{2} \sqrt{1 + 8 \times idx} \right) - 1 \right\rfloor$$

Eq. 4.3.1 Index row of value in `d_mat` from which thread with global index `idx` reads.

$$j = cols - \left(idx - \frac{i(i-1)}{2} \right) - 1 + i$$

Eq. 4.3.2 Index column of value in `d_mat` from which thread with global index `idx` reads.

The final read_idx value which is a 1D unrolled index is computed using Eq. 4.2.5. The actual GPU kernel is presented below –

```
__global__ void ShiftLeft_PopulateHelper_GPUKernel(float
*d_mat, float *d_helper, int damaged_elems, int rows, int
cols) {
  int idx = (blockDim.x * blockIdx.x + threadIdx.x) %
damaged_elems;
  int i = floor(0.5f * (sqrt((float)1 + 8 * idx) - 1.0f));
  int j = cols - (idx - i * (i - 1) / 2) - 1 + i;
  int read_idx = j + i * cols;
  d_helper[idx] = d_mat[read_idx];
}
```

After populating the helper array, the actual writing and transform is performed in a similar manner to section 4.2, but only in the opposite direction. The GPU kernel to perform this shift is presented below.

```
__global__ void ReAlignMemory_ShiftLeft_GPUKernel(float
*d_mat, float *d_helper, int total_size, int cols, int
thread_chunk_size) {
  extern __shared__ float read_vals[];
  int shared_mem_idx, read_idx, read_idx_row, write_idx;
  int row_linear_idx = blockIdx.x * cols;
  int rows = total_size / cols;
  int read_idx_base = row_linear_idx
    + (threadIdx.x * thread_chunk_size) % cols + 1;
  int read_idx_lateral;
  int row_last_linear_idx = row_linear_idx + cols;
  for (read_idx = read_idx_base; read_idx <
row_last_linear_idx;
       read_idx++) {
    read_idx_row = read_idx / cols;
    shared_mem_idx = read_idx - row_linear_idx - 1;
    if (read_idx < ((read_idx_row + 1) * (cols - 1))
        || blockIdx.x == (rows - 1)) {
      read_vals[shared_mem_idx] = d_mat[read_idx];
    }
    else {
      read_idx_lateral = row_linear_idx + cols -
shared_mem_idx - 2;
      read_vals[shared_mem_idx] = d_helper[read_idx_lateral
        - cols * read_idx_row
        + (read_idx_row - 1)
        * read_idx_row / 2 + read_idx_row];
    }
  }
  __syncthreads();
  for (read_idx = read_idx_base; read_idx <
row_last_linear_idx;
       read_idx++) {
    read_idx_row = read_idx / cols;
    shared_mem_idx = read_idx - row_linear_idx - 1;
    write_idx = row_linear_idx + shared_mem_idx -
read_idx_row;
    d_mat[write_idx] = read_vals[shared_mem_idx];
  }
}
```

The invocation of these kernels can be understood with the help of the following code snippet which does the whole job –

```
void ReAlignMemory_ShiftLeft(float *d_mat, float *d_helper,
int rows, int cols, int max_threadblock_size) {
  int org_size = rows * cols;
  int reqd_threads = rows * (rows - 1) / 2;
  int threadblock_size = GPU_WARP_SIZE *
GPU_WARP_DISPATCHERS * 2;
  if (threadblock_size > max_threadblock_size)
    threadblock_size = max_threadblock_size;
  int num_threadblocks =
my_ceilf_division_FCLayer(reqd_threads, threadblock_size);
  int thread_chunk_size = my_ceilf_division_FCLayer((cols -
1), max_threadblock_size);
  ShiftLeft_PopulateHelper_GPUKernel << < num_threadblocks,
    threadblock_size >> >
    (d_mat, d_helper, reqd_threads,
     rows, cols);
  reqd_threads = my_ceilf_division_FCLayer((cols - 1),
thread_chunk_size);
  threadblock_size = my_ceilf_division_FCLayer(reqd_threads,
GPU_WARP_SIZE)
    * GPU_WARP_SIZE;
  ReAlignMemory_ShiftLeft_GPUKernel << < rows,
    threadblock_size,
    sizeof(float) * (cols - 1) >> >
    (d_mat, d_helper,
     org_size, cols,
     thread_chunk_size);
}
```

The execution pattern is very similar to the `ShiftRight` transformation as explained in section 4.2.

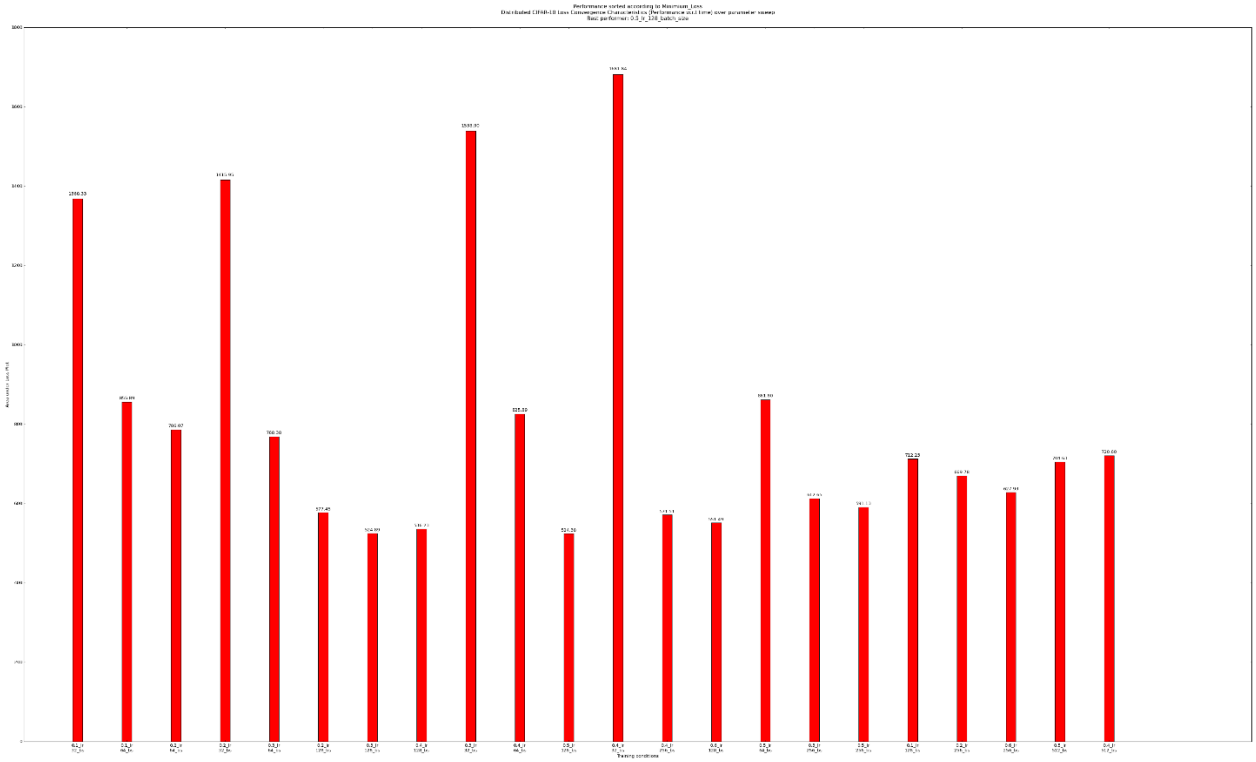

Fig. 5.1 Loss-plot area vs training-hyperparameter histogram for parameter search over 1 worker

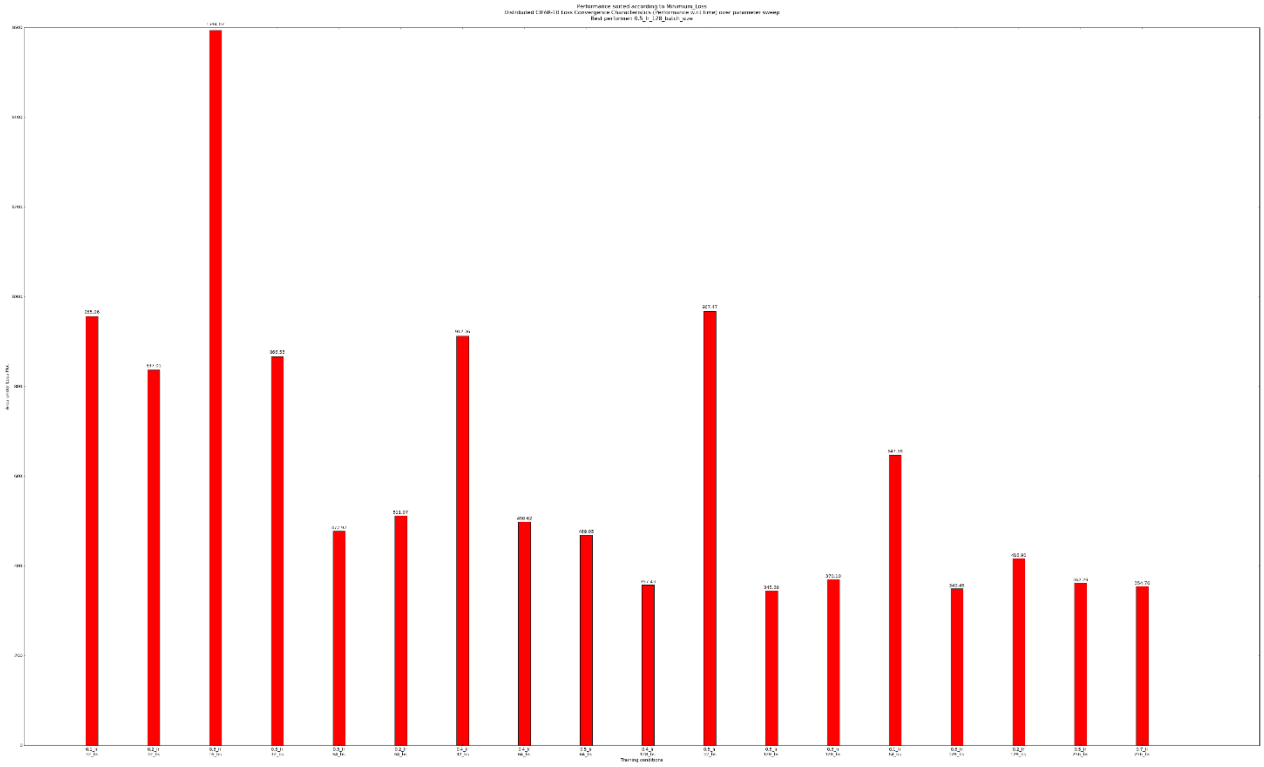

Fig. 5.2 Loss-plot area vs training-hyperparameter histogram for parameter search over 2 workers

***Name**: Souham Biswas **CWID**: A20365242 **Email**: sbiswas7@hawk.iit.edu*

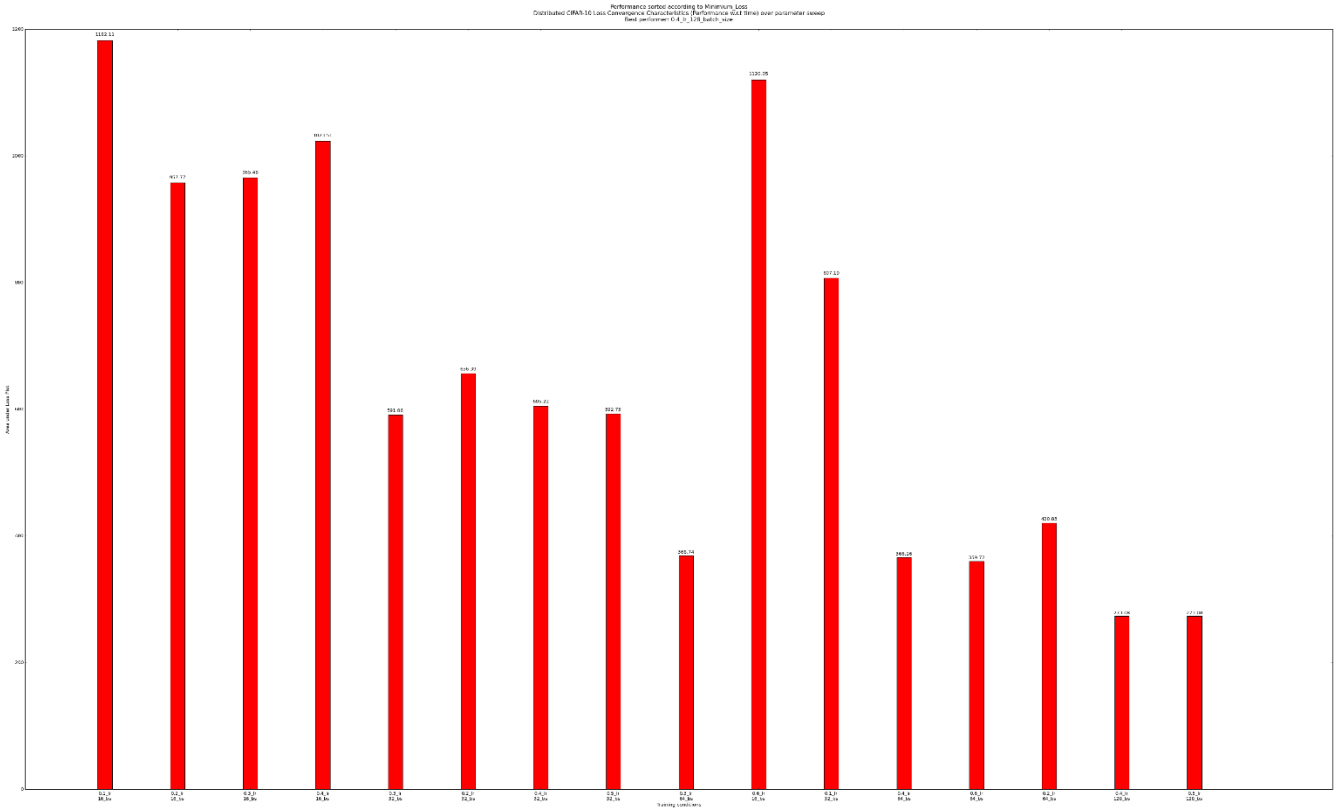

Fig. 5.3 Loss-plot area vs training-hyperparameter histogram for parameter search over 4 workers.

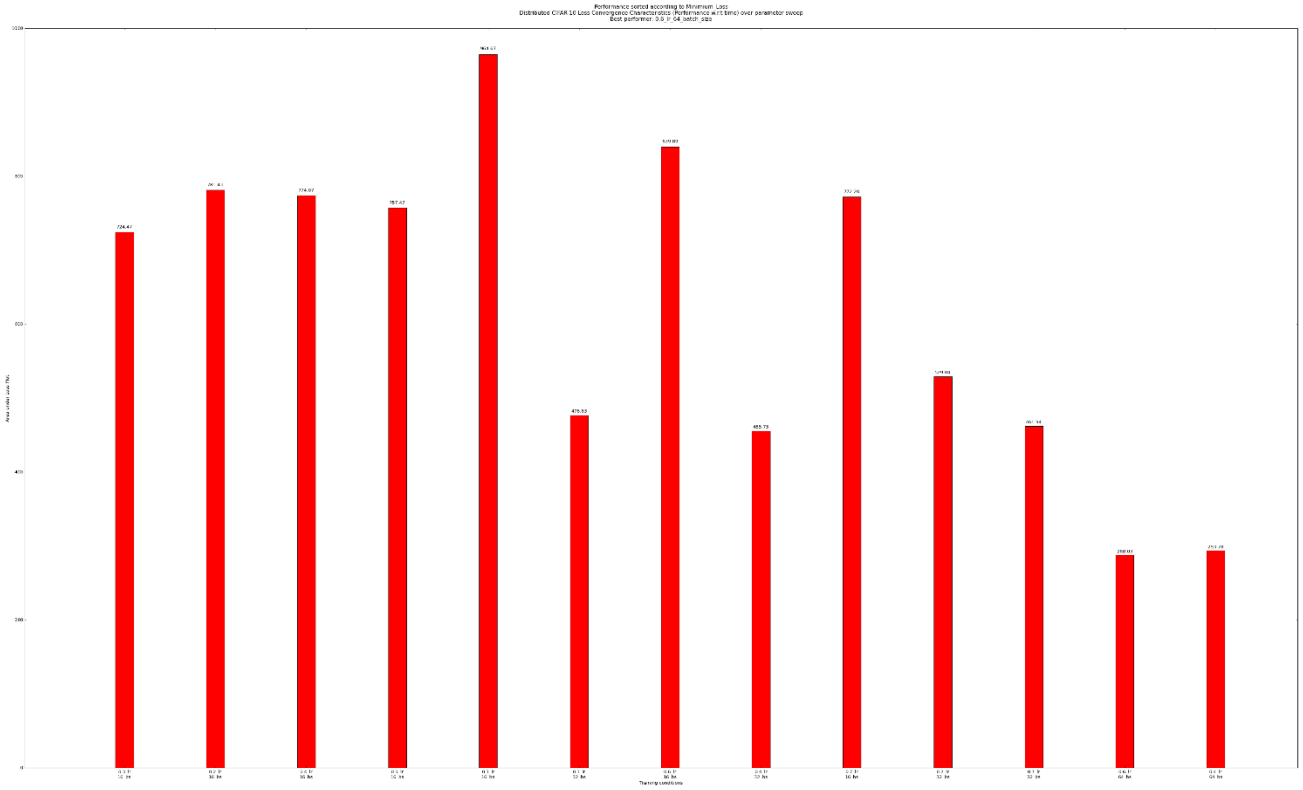

Fig. 5.4 Loss-plot area vs training-hyperparameter histogram for parameter search over 8 workers.

5. CIFAR-10 distributed training parameter search

To arrive at optimal hyper-parameters to use for distributed training over the CIFAR-10 dataset, a hyper-parameter search was conducted over all the training schemes involving 1, 2, 4 & 8 nodes. The results are shown in Fig. 5.1, Fig. 5.2, Fig. 5.3 & Fig. 5.4 for 1, 2, 4 & 8 workers respectively.

In a naïve metric of deciding the "best performer", we could solely use the area under the loss curve as the metric. However, this leads to the loss curve with the minimum time to always win regardless of the final loss it reached, no matter how big it is. Therefore, a new metric to enumerate the "best performer" was devised.

Under the new scheme, firstly, the loss curves whose minimum losses aren't less than or equal to a certain threshold (1.8 in this case) are eliminated. Then, the curve with the minimum area under curve from the remaining curves, is chosen as the "winner". Presented in Fig. 5.1, Fig. 5.2, Fig. 5.3 & Fig. 5.4 are the histogram plots of areas under the loss plots sorted according to the minimum loss attained by the curve (Lowest to Highest); the best performer in each case is mentioned in the plot title.

Training was performed such that a constant 307,200 images are “seen” in each case to keep the training data quantity constant in each case. The number of iterations & epochs are automatically adjusted according to the batch size being used to enforce this constraint. The number 307,200 was chosen to ensure that integral values for the number of iterations & epochs were obtained over the batch size search space.

Experiments were conducted over a batch size search space over the values of 16, 32, 64, 128, 256, 512 and a learning rate space of 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9. Fig. 5.5 – 5.8 show the visualization matrices showing the areas of loss plots with different training conditions. “Empty” spots in the visualizations (with value of 0) correspond to diverged training runs. Do note that 1 worker diverges always with a 16 batch size.

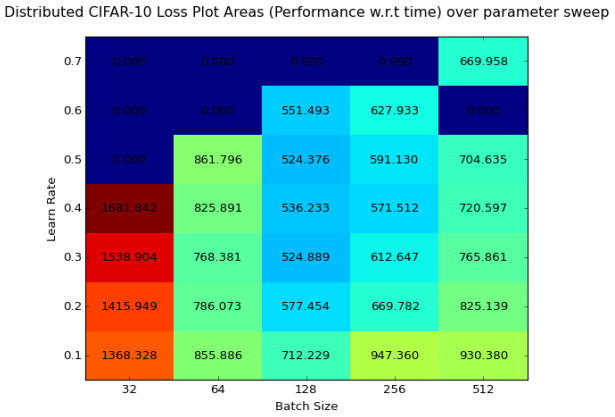

Fig 5.5 Area under Loss plots for parameter search over 1 worker.

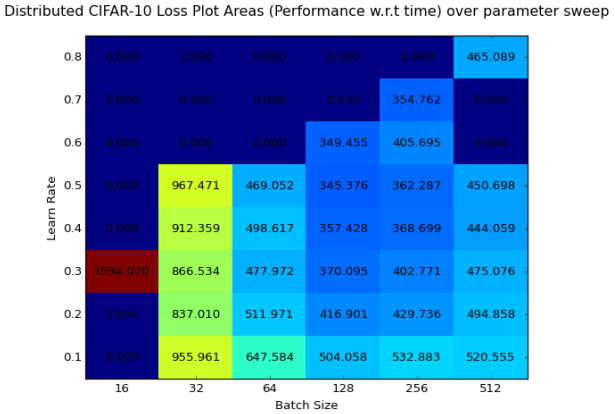

Fig 5.6 Area under Loss plots for parameter search over 2 workers.

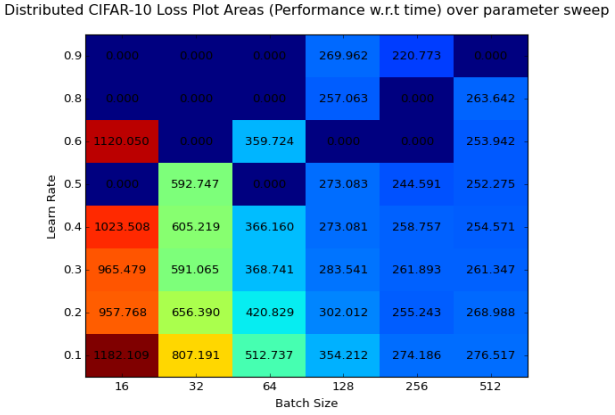

Fig 5.7 Area under Loss plots for parameter search over 4 workers.

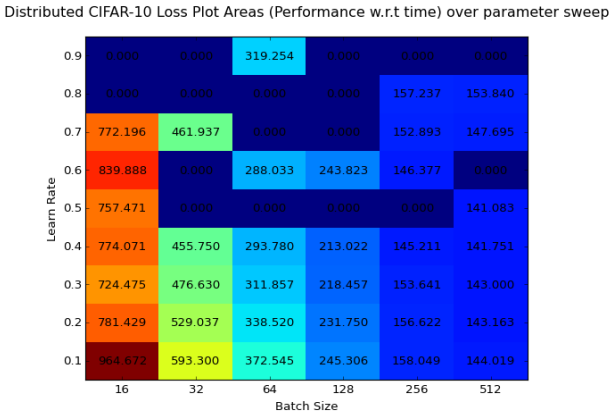

Fig 5.8 Area under Loss plots for parameter search over 8 workers.

The neural network architecture deployed here is the AlexNet[5]. The final hyper-parameters obtained for the distributed training over 1, 2, 4 & 8 nodes are given in Table 5.1.

#Workers	Batch Size	Learn Rate
1	128	0.5
2	128	0.5
4	128	0.4
8	64	0.6

Table 5.1 Hyper-parameter search results

Therefore, these optimum hyper-parameters shall be used to evaluate the performance of the SHM-Learn-Dist framework posited in this paper.

The training was done on NVIDIA Tesla K80 GPUs.

6. SHM-Learn-Dist Performance Plots

The performance plots of the posited framework over the CIFAR-10 dataset are presented here. The hardware utilized in this work involves a cluster with each node equipped with an NVIDIA K80 GPU with 12 GB GPU memory per GPU, 2 Intel Haswell architecture E5-2620 v3 processors with 6 cores per CPU and 384 GB of RAM.

Training was done using gradient averaged data parallel training. The distribution back-end was written in MPI[6]. Currently, this implementation does not include GPUDirect based inter-node GPU-GPU communication and all the gradient and weight data exchanged between GPUs on different nodes passes through the host. This is so because the pre-installed mvapich2[6] version on the cluster was built without the support for CUDA-aware MPI. However, efforts are being made to rebuild the MPI version with CUDA support to further accelerate inter-node communication.

The distribution scheme in terms of MPI ranks is as follows –

1. Rank 0 initializes random weights and distributes the weights to the workers.
2. Each worker computes gradients for the weights on their next respective batch of images.
3. Rank 0 gathers gradients from each of the workers.
4. Rank 0 computes average of the gradients.
5. Rank 0 applies the averaged gradients to it's own weights.
6. Rank 0 broadcasts the weights to the workers.
7. Process continues from step 2.

The gather and broadcast operations utilized are illustrated in Fig. 6.1 –

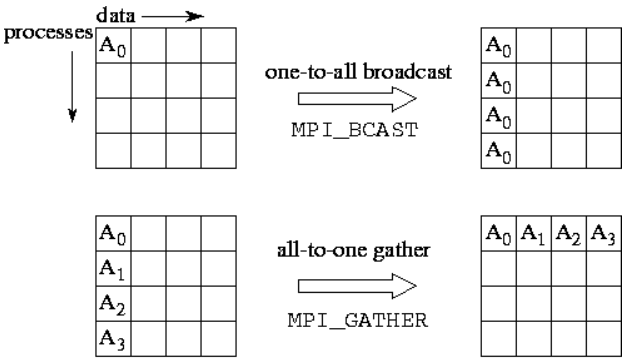

Fig. 6.1 Visual illustration of the MPI gather and broadcast operations leveraged for distributed training.

Although multiple GPUs exist on a single node, only one GPU per node is utilized. Multi-GPU support for each node is currently on the roadmap.

Name: *Souham Biswas* ***CWID***: *A20365242* ***Email***: *sbiswas7@hawk.iit.edu*

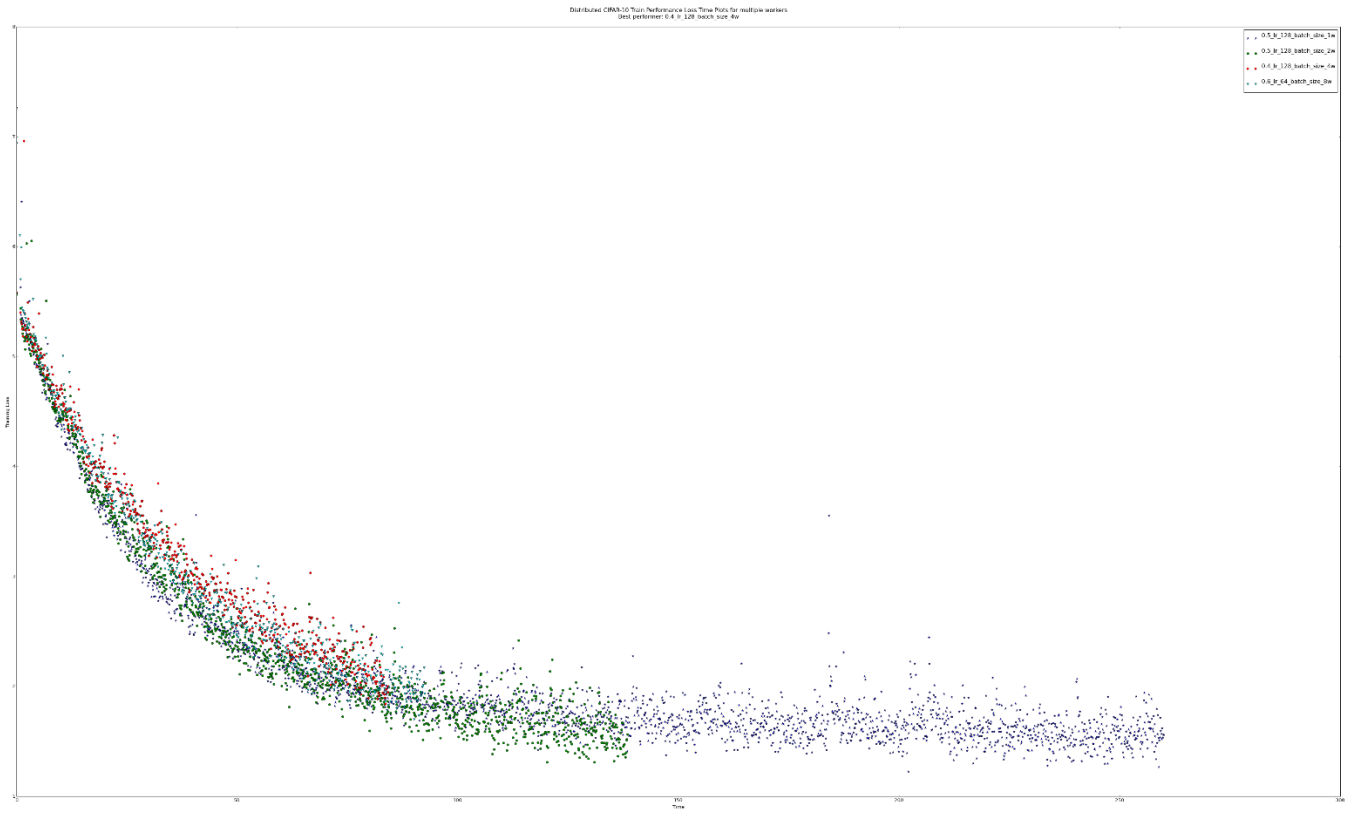

Fig. 6.2 Loss Plots over different workers

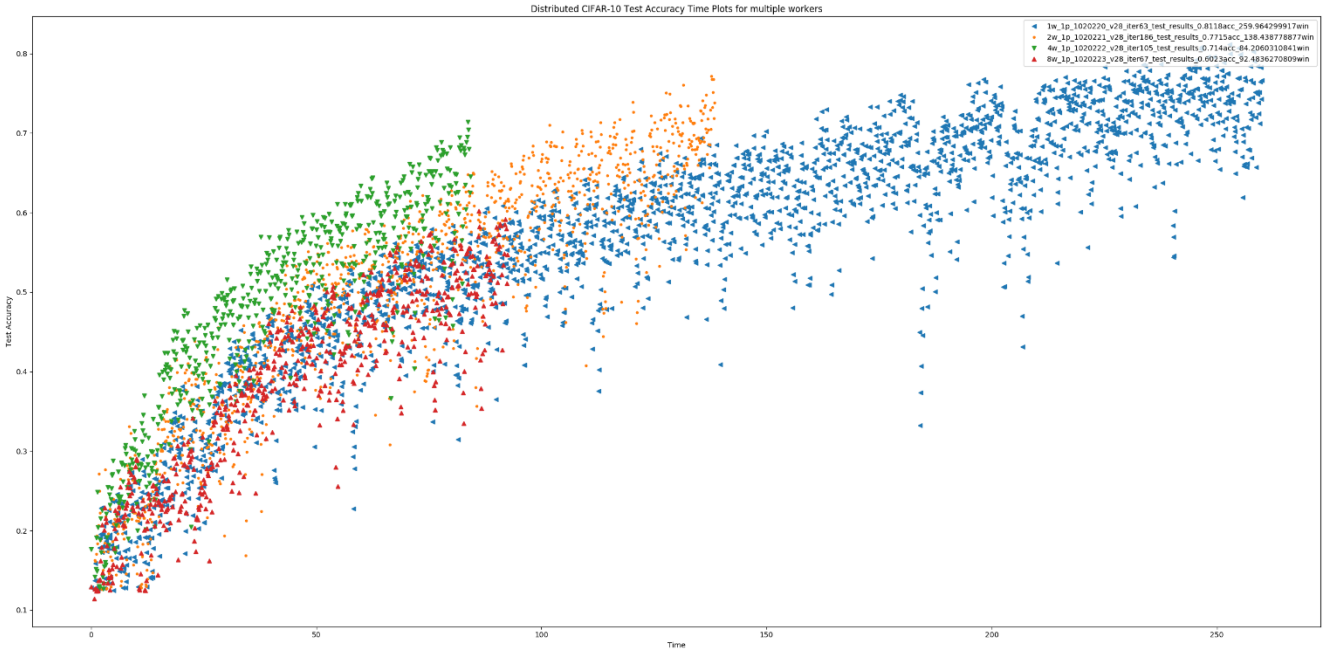

Fig. 6.3 Test Accuracies over different workers

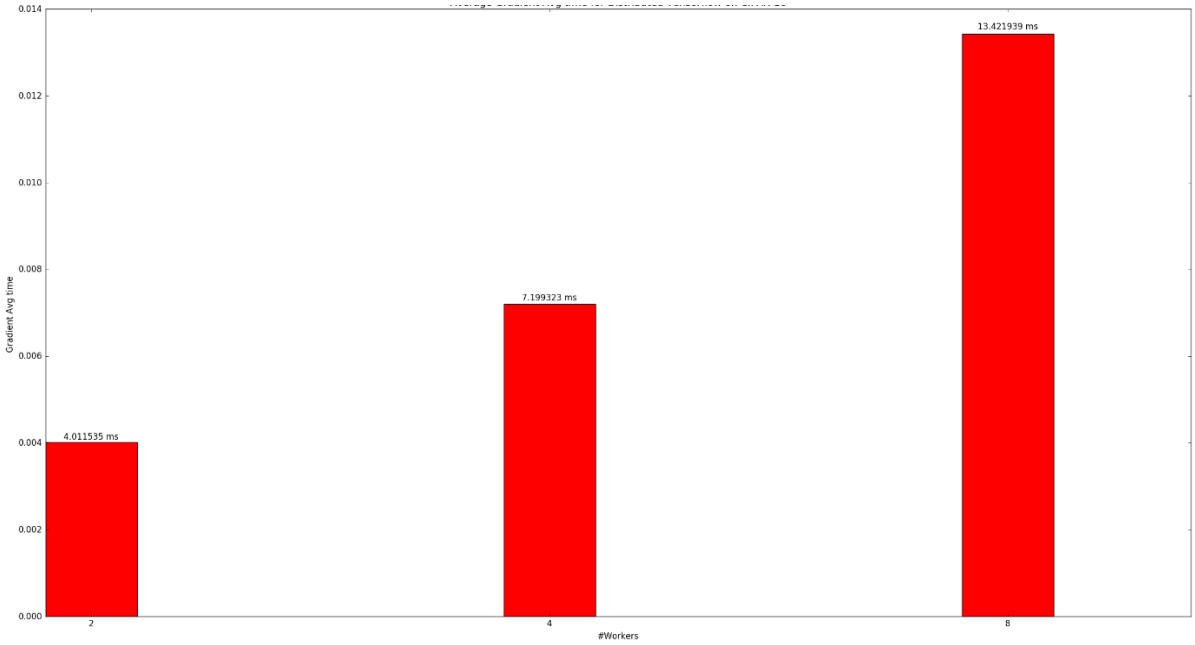

Fig. 6.4 Average time taken to compute mean of gradients at parameter node

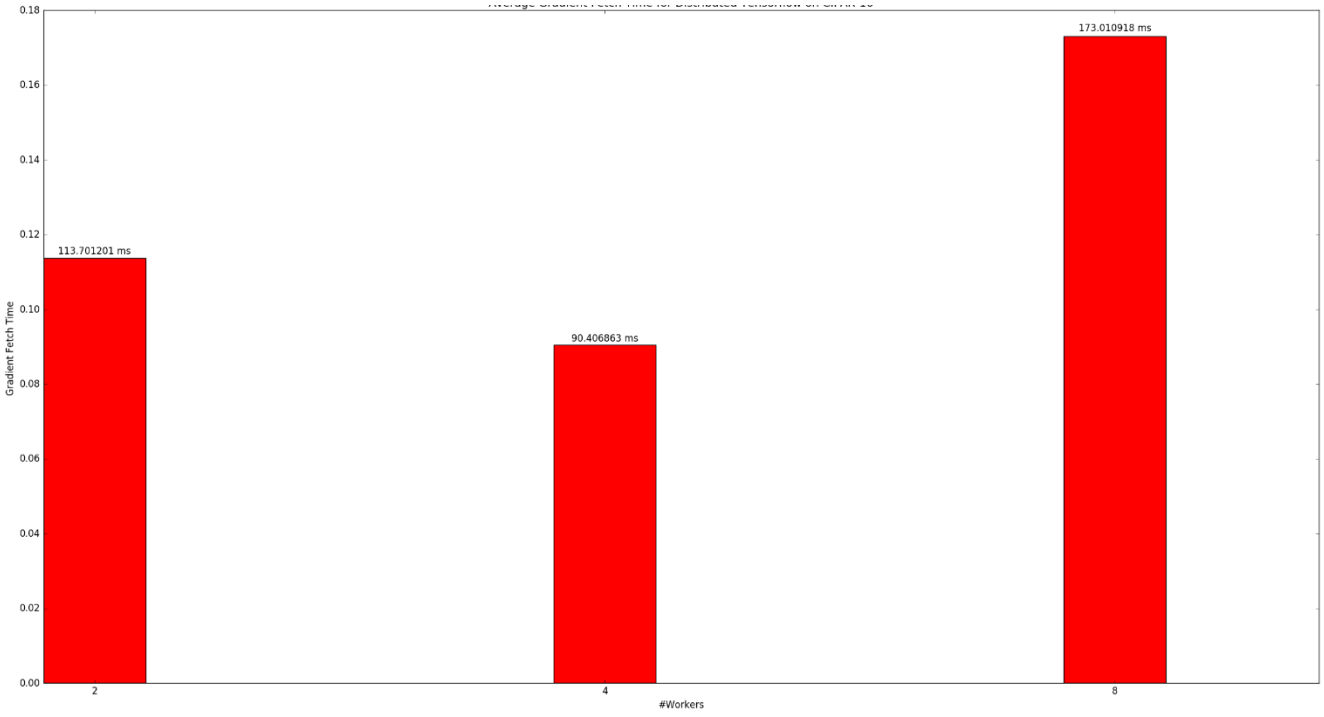

Fig. 6.5 Average time taken by parameter node to collect the gradients from each worker node.

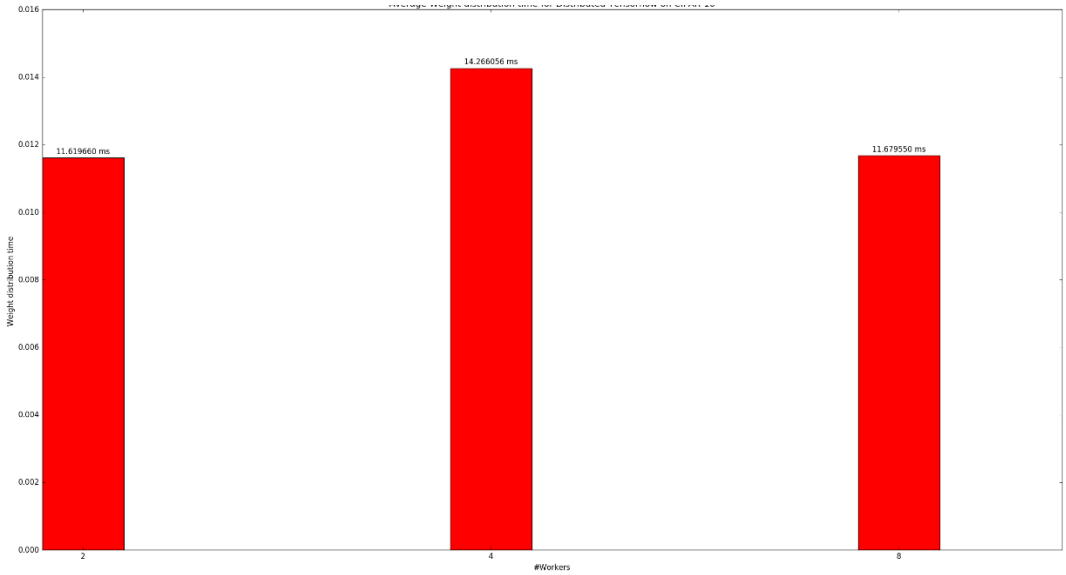

Fig. 6.6 Average time taken to distribute weights from the parameter node to the worker nodes.

Fig. 6.2 presents the progression of the L2 loss of the distributed training with time. It can be seen here that the 'length' of each plot on the graph vaguely corresponds to the number of workers and that the final loss to which they converge also map to the number of workers.

However, the progression of test accuracies over 10,000 test images with time as presented in Fig. 6.3 shows something rather interesting. The 4 worker trainer converged to around 71% accuracy but took only 84.2 seconds, whereas the 2 worker trainer converges to 77.15% accuracy at 138.44 seconds. The 1 worker trainer might haver converged to 81.8% accuracy but it also took a much longer time of around 260 seconds. It is to be noted here that the amount of training data provided to each setting is exactly the same. The 8 worker trainer will not be considered as its final accuracy and the time it takes are both poor as compared to the other settings.

Judging by these results, it can be stated that the 4 worker trainer seems to outperform the others given that it has the highest accuracy percentage gain per unit time over the same quantity of training data. It is followed by the 2 worker and the 1 worker trainers. The 8 worker trainer seems to perform poorly because the problem does not scale to larger nodes; in other words, the problem is not complex enough (as demonstrated in section 3).

Fig. 6.4 explicates the gradient averaging time which happens on the parameter node. This seems to linearly scale with increasing number of worker nodes which is expected as more workers mean more gradients to average. Fig. 6.5 shows the gradient collection time from each worker to the parameter node which shows quite an unusual behavior; a similar unusual behavior is observed in Fig. 6.6 which illustrates how the time taken to broadcast the updated weights from the parameter node to the workers change with different workers. It can be seen that the 4 worker trainer is faster than the 8 worker trainer in gathering the gradients; however, the 8 worker trainer is faster than the 4 worker trainer in distributing the weights to the workers.

This strange behavior can possibly be attributed to the internal implementation of the gather and broadcast routines in MPI. These implementations are topology-aware and leverage the physical layout of the nodes to minimize the data communication times using tree-based algorithms[7].

Although similar performance plots for other toolkits were unable to be obtained in time, judging by some readings obtained, SHM-Learn-Dist seems to provide an average of 5x distributed training speedup over tensorflow on the CIFAR-10 dataset with similar correctness.

7. Future Work

Future work involves expanding the library to support CUDA-aware MPI so as to eliminate the performance overhead incurred by the data being passed through the host every time there is an inter-node communication. Single-node multi-GPU support using the NCCL library[8] is on the roadmap to better leverage the power of multiple GPUs at every node.

8. References

1. Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada, United States, December, 2012.

2. Peter H. Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning?. NIPS 2016: ML Systems Workshop

3. Souham Biswas. SHM-Learn: An optimized GPU-Powered C++ Deep Learning Library. Illinois Institute of Technology, Fall 2016 CS 597 report

4. Justin Luitjens, Steven Rennich. CUDA Warps and Occupancy. GPU Tech Conference 2011: GPU Computing Webinar 7/12

5. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. NIPS 2012: Advances in Neural Information Processing Systems 25

6. The Ohio State University. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP and RoCE.

7. Wes Kendall. MPI Broadcast and Collective Communication. MPI Tutorials at http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

8. NVIDIA Corporation. NCCL: Optimized primitives for collective multi-GPU communication. https://github.com/NVIDIA/nccl