1 Background and Experimental Methodology

In this work, we implement the AllReduce collective communication primitive that is frequently used in ML Systems for performing parameter update after parallel gradient compute. PyTorch Distributed, a distributed training module was utilized for the network functions built on the *gloo* backend. The nodes communicate over a TCP connection using synchronous APIs. The system setup for the implementation was based on 16 cores of Intel Xeon Silver 4114 on the c220g5 cluster at CloudLab. The code for our implementation has been captured in this repository.

2 Task 1: Implementation

We implemented the following two algorithms for AllReduce presented in prior work [1] and analyzed the performance of each in the subsequent tasks.

2.1 Ring AllReduce

Given n-slices of the tensor to be exchanged between n-nodes, Ring AllReduce relies on the simple principle that we can have every node send a slice and receive a different slice in each cycle. Number of exchanges is proportional to number of workers i.e., 2*(N-1) where N is the number of nodes. We introduced an optimization here to have all the alternate nodes perform send in parallel followed by receive in parallel in the next cycle.

```
# Reduce-scatter loop
for i in range(1, world_size):
    s = time.time()
    if (rank % 2) == 0:
        # Send a tensor to the previous machine
        #print((rank + i) % world_size)
        dist.send(t[(rank + i) % world_size], dst=(rank + world_size - 1) % world_size)
        # Receive a tensor from the next machine
        dist.recv(recv_buffers[i-1], src=(rank + 1) % world_size)
    else:
        # Receive a tensor from the next machine
        dist.recv(recv_buffers[i-1], src=(rank + 1) % world_size)
        # Send a tensor to the previous machine
        dist.send(t[(rank + i) % world_size], dst=(rank + world_size - 1) % world_size)
    e = time.time()
    total_time += e - s
    # Accumulate value in t. At the end of the for loop, t will hold the reduced value
    t[(rank + i + 1) % world_size] += recv_buffers[i-1]
# All-gather loop
for i in range(1, world_size):
    s = time.time()
    if (rank % 2) == 0:
        # Send a tensor to the next machine
        dist.send(t[(rank + 1 - i + world_size) % world_size], dst=(rank + 1) % world_size)
        # Receive a tensor from the previous machine
        dist.recv(t[(rank - i + world_size) % world_size], src=(rank + world_size - 1) % world_size)
    else:
        # Receive a tensor from the previous machine
        dist.recv(t[(rank - i + world_size) % world_size], src=(rank + world_size - 1) % world_size)
        # Send a tensor to the next machine
        dist.send(t[(rank + 1 - i + world_size) % world_size], dst=(rank + 1) % world_size)
```

Fig 1. Implementation of Ring AllReduce

2.2 Recursive halving/doubling AllReduce

Recursive halving/doubling uses bi-directional exchange of tensors between nodes. AllReduce is implemented in two steps in this approach:

1. BDE Reduce Scatter is intended to aggregate the tensor slices from all nodes. Each node owns a single slice of the reduced vector at the end of this step. The code is presented in Fig. 2.
2. BDE AllGather ensures that all nodes have all parts of the reduced tensor in (1). The code is presented in Fig. 3.

```python
# BDE Reduce-Scatter

def bde_reduce_scatter(rank, x, tensor_left, tensor_right, rank_left, rank_right):

    if rank_left == rank_right:
        return
    rank_size = rank_right - rank_left + 1
    tensor_size = tensor_right - tensor_left + 1
    tensor_mid = (tensor_left + tensor_right) // 2
    rank_mid = (rank_left + rank_right) // 2
    if rank <= rank_mid:
        partner = rank + (rank_size/2)
    else:
        partner = rank - (rank_size/2)
    partner = int(partner)
    if rank <= rank_mid:
        dist.send(x[tensor_mid+1:tensor_right+1], dst=partner)
        recv_buffer =  copy.deepcopy(x[tensor_left:tensor_mid+1])
        dist.recv(recv_buffer, src=partner)
        x[tensor_left:tensor_mid+1] = x[tensor_left:tensor_mid+1] + recv_buffer
    else:
        recv_buffer = copy.deepcopy(x[tensor_mid+1:tensor_right+1])
        dist.recv(recv_buffer, src=partner)
        x[tensor_mid+1:tensor_right+1] = x[tensor_mid+1:tensor_right+1] + recv_buffer
        dist.send(x[tensor_left:tensor_mid+1], dst=partner)

    if rank <= rank_mid:
        bde_reduce_scatter(rank, x, tensor_left, tensor_mid, rank_left, rank_mid)
    else:
        bde_reduce_scatter(rank, x, tensor_mid+1, tensor_right, rank_mid+1, rank_right)
```

Before BDE Reduce Scatter

Node 0	Node 1	Node 2	Node 3
$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(3)}$
$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(3)}$
$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(3)}$
$x_3^{(0)}$	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(3)}$

After BDE Reduce Scatter

Node 0	Node 1	Node 2	Node 3
$x_0^{(0:3)}$			
	$x_1^{(0:3)}$		
		$x_2^{(0:3)}$	
			$x_3^{(0:3)}$

Fig. 2: Implementation of BDE Reduce Scatter

```python
# BDE AllGather

def bde_all_gather(rank, x, tensor_left, tensor_right, rank_left, rank_right):
    if rank_left == rank_right:
        return

    rank_size = rank_right - rank_left + 1
    tensor_size = tensor_right - tensor_left +1
    tensor_mid = (tensor_left + tensor_right) // 2
    rank_mid = (rank_left + rank_right) // 2

    if rank <= rank_mid:
        partner = rank + (rank_size/2)
    else:
        partner = rank - (rank_size/2)
    partner = int(partner)

    if rank <= rank_mid:
        bde_all_gather(rank, x, tensor_left, tensor_mid, rank_left, rank_mid)
    else:
        bde_all_gather(rank, x, tensor_mid+1, tensor_right, rank_mid+1, rank_right)

    if rank <= rank_mid:
        dist.send(x[tensor_left:tensor_mid+1], dst=partner)
        recv_buffer =  x[tensor_mid+1:tensor_right+1]
        dist.recv(recv_buffer, src=partner)
    else:
        recv_buffer = x[tensor_left:tensor_mid+1]
        dist.recv(recv_buffer, src=partner)
        dist.send(x[tensor_mid+1:tensor_right+1], dst=partner)
```

Before BDE AllGather

Node 0	Node 1	Node 2	Node 3
x_0			
	x_1		
		x_2	
			x_3

After BDE AllGather

Node 0	Node 1	Node 2	Node 3
x_0	x_0	x_0	x_0
x_1	x_1	x_1	x_1
x_2	x_2	x_2	x_2
x_3	x_3	x_3	x_3

Fig. 3: Implementation of BDE AllGather

3 Task 2: Tensor Size variation

We measured the time to complete the AllReduce operation with 16 nodes and varying tensor sizes. Each element of the tensor generated using *torch.rand(TENSOR_SIZE)* is of the type float32 (that consumes 4bytes) and TENSOR_SIZE values were determined correspondingly. The code was instrumented to measure time taken for the AllReduce operation in each rank to arrive at the same tensor across all nodes. The measured time includes the communication (send, receive) and compute operations within the algorithm. We observed that there is a small variation in this time measured across ranks as illustrated in Fig. 4.

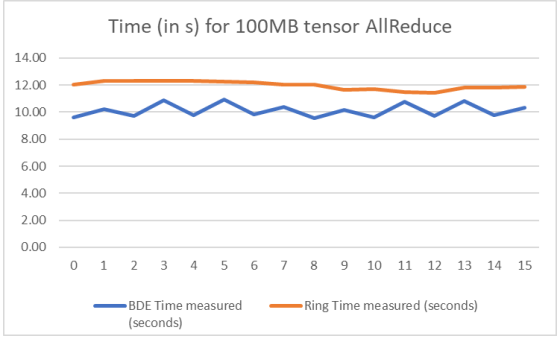


Fig. 4. Variation in measured time for 100MB AllReduce measured across ranks

For this reason, for all subsequent measurements we used the average of measured times across ranks for analysis. Fig. 5 illustrates the measurements and trends across varying tensor sizes for the two implementations. The recursive halving/doubling algorithm using BDE performs faster than ring in all cases as expected.

	A	B	C	D	E
1			Measured Time (s)		BDE % faster
2	Size in bytes	(TENSOR_SIZE)	Ring	Recursive H/D BDE	(Ring-BDE)/Ring)
3	1KB	256	0.1098	0.0852	29%
4	10KB	2560	0.1085	0.0892	22%
5	100KB	25600	0.1114	0.0914	22%
6	1MB	256000	0.3576	0.2186	64%
7	10MB	2560000	1.5962	1.1790	35%
8	100MB	25600000	11.9740	10.1329	18%

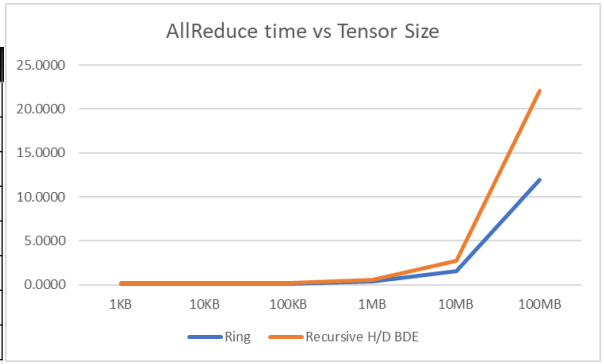


Fig. 5. Effect of varying tensor sizes

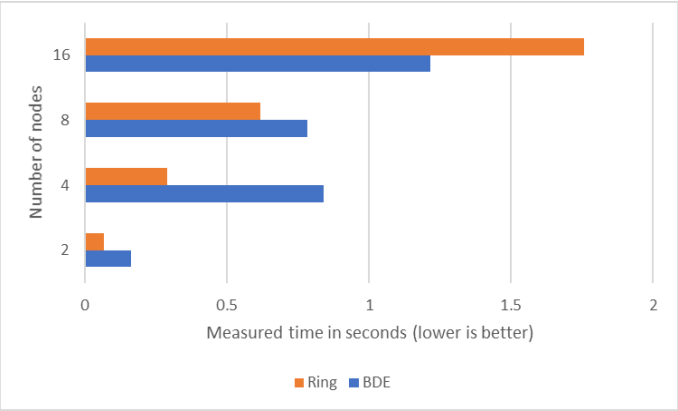
We tried a second flavor of instrumentation (code here) to capture only the network time (send/receive) as opposed to measuring the total time to complete AllReduce. The observed metrics are captured as Experiment 2 in Fig. 6. The approximate compute % composition shown in column E was determined as (B-(C+D))/B. From this we can deduce that the communication time dominates in both these algorithms. Hence, the measurements made in Fig. 5 should be reasonably accurate to compare the communication performance trends of the two algorithms.

	A	B	C	D	E
1	Recursive H/D BDE				
2		Exp1: Total	Experiment 2		
3	Size in bytes	Time	Send	Recv	~Compute %
4	1MB	0.2186	0.0975	0.0727	22%
5	10MB	1.1790	0.5480	0.5546	6%
6	100MB	10.1329	4.8362	4.1124	12%

Fig 6. Compute and Communication composition in measured time

4 Task 3: Number of node variation

For this experiment, keeping tensor size constant at 10MB we varied the number of nodes participating in the communication. The lower bound equations are presented in Table I of [1]. We have the following variables: **p** = 2/4/8/16, **n** = (p-1) i.e., we expect to send/receive p-1 packets with the participating nodes. For the combination of Reduce-scatter and AllGather, the bandwidth factor is given by (2*(p-1)*(p-1))/p. The product of this scaling factor and the measured bandwidth is roughly a constant across varying number of nodes indicative of a theoretical bound.



A	B	C	D	E
Num. nodes	[(2*(p-1)*(p-1))/p]	Measured Time (s)	Bandwidth (MB/s)	B*D
2	1	0.0658	151.9243252	151.9243252
4	4.5	0.2908	34.39278366	154.7675265
8	12.25	0.6178	16.18639868	198.2833839
16	28.125	1.7568	5.692071611	160.0895141

Fig. 7. Performance trends between the two algorithms in varying number of nodes

Fig.8. Measured time for Ring AllReduce

5 Task 4: Theoretical bounds in communication

Based on the data we obtained by varying tensor size, we obtained the following table by using the formulae for alpha and beta:

Size	alpha	beta
1KB	0.001119	0.040646
10KB	0.000879	0.043801
100KB	0.00091	0.044847
1MB	0.00632	0.089595
10MB	0.018965	0.547873
100MB	0.083685	5.047158

Fig 8. Calculated alpha and beta values

As we can see, both alpha and beta seem to have some dependency on the size of message being passed. In beta's case it is more clear cut – at smaller message sizes, bandwidth is probably not being utilized efficiently, so we're seeing a fixed penalty, with beta showing minor increases with increase in tensor size. As the messages get much larger however, we see that beta begins to scale proportionately, as we would expect. Alpha curiously, also seems to scale proportionately with the message size, which should not usually be the case, so this may be resulting from an additional cost that we have not factored out. If we account for the "scaling" of these values, alpha seems to be around 0.001 and beta around 0.05.

We tried to validate our results with the data obtained by varying the number of nodes used, but we found that at lower node counts, ring was anomalously outperforming BDE, leading to negative alpha, which should not be possible. Further, this does not seem to be a one off/anomalous set of readings, as we noticed similar trends while collecting data after limiting bandwidth. This indicates that there is some inefficiency in our implementation of BDE at lower node counts which is not captured by the formula, some kind of constant factor that is overshadowed at when more nodes are used.

6 Task 5: Limiting network bandwidth

For this experiment, we constrained the upload/download network bandwidth to 100Mb/s using Wondershaper across all nodes. The measurements from Task3 were repeated. The trends in performance are presented in Fig. 9. We did not observe appreciable difference from limiting the interface bandwidth. We attributed this as an effect of the Tensor Size which was 10MB (80Mb) in this case. At any given time, each node would only be sending a fraction of the total tensor size thereby no single interface reaches the set limit for throughput.

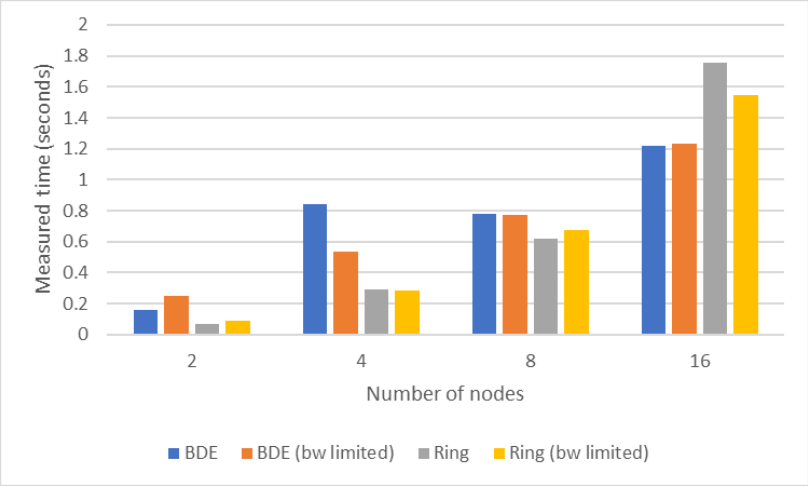


Fig. 9. Variation of measured time with and without won

7 References

[1] Collective communication https://www.cs.utexas.edu/~pingali/CSE392/2011sp/lectures/Conc_Comp.pdf

8 Contributions

Ashwin Poduval

- Implementation of Ring AllReduce
- Task-wise performance analysis and documentation

Rajesh Shashi Kumar

- Implementation of Recursive halving/doubling AllReduce
- Task-wise performance analysis and documentation