Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis
Tal Ben-Nun and Torsten Hoefler
ETH Zurich

Presenter: Derrick Blakely
https://qdata.github.io/deep2Read

March 6, 2019
Outline

1. Background
2. Parallel Computing and Communication
3. Neural Network Concurrency
4. Parameter Sharing and Consistency
5. Frameworks
6. Conclusion
Deep Learning is Using More Nodes

- Median
- 25th/75th Percentile
- Min/Max

# Background

**Presenter:** Derrick Blakely  
[https://qdata.github.io/deep2Read](https://qdata.github.io/deep2Read)  
(University of Virginia)  

**Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis**  
Tal Ben-Nun and Torsten Hoefler  
ETH Zurich  
March 6, 2019  

---

**Graphs:**

1. **Number of Nodes**
   - Y-axis: Number of Nodes
   - Graphs show trends in the use of nodes from pre-2013 to present, with notable nodes such as Titan Supercomputer, DistBelief, and Project Adam.

2. **Reported Experiments [%]**
   - Y-axis: Percentage of experiments using single versus multiple nodes.
   - The percentage of experiments using multiple nodes has increased over time, indicating a trend towards more complex and resource-intensive tasks.
More GPUs, More MPI

- **CPU**
- **GPU**
- **FPGA**
- **Specialized**

Reported Experiments [%]

Year:
- Pre-2010
- 2010
- 2011
- 2012
- 2013
- 2014
- 2015
- 2016
- 2017-Present

- **MPI**
- **MapReduce**
- **Spark**
- **Sockets**
- **RPC**

Reported Experiments
Outline

1. Background

2. Parallel Computing and Communication

3. Neural Network Concurrency

4. Parameter Sharing and Consistency

5. Frameworks

6. Conclusion
Critical path: causes the earliest possible completion time
Depth $D$: time needed to execute critical path
$T_1 = W$: sequential execution; same as total work
$T_\infty = D$: with unlimited processors
Average parallelism: $T_1 / T_\infty = W / D$
Parallel Computing Basics: Granularity

- Granularity: $G \approx \frac{T_{\text{comp}}}{T_{\text{comm}}}$
- Alternatively:

$$G = \frac{\min_{n \in V} w(n)}{\max_{e \in E} c(e)}$$
Granularity: \( G \approx \frac{T_{\text{comp}}}{T_{\text{comm}}} \)

Alternatively:

\[
G = \frac{\min_{n \in V} w(n)}{\max_{e \in E} c(e)}
\]
Parallel Computing Basics: Granularity

- **Granularity**: $G \approx \frac{T_{\text{comp}}}{T_{\text{comm}}}$
- Alternatively:
  
  $G = \frac{\min_{n \in V} w(n)}{\max_{e \in E} c(e)}$

```c
    c = 0;
    For (i=0; i<16; i++)
        c = c + A[i]
```

![Diagram](image)
Goals

- Maximize parallelism
- Minimize communication and load imbalance
- Tradeoff: high parallelism, low communication
- Tradeoff: high load balance vs low communication
- High parallelism and high load balance are often compatible
Want to perform the following AllReduce:

\[ y = x_1 \oplus x_2 \oplus ... \oplus x_{P-1} \oplus x_P \]

- \( P \): num processing elements
- \( x_i \): length-\( m \) vector of data items stored on a processing element
- \( \gamma \): size of a data item (bytes)
- \( G \): computation cost per byte
- \( L \): network latency
Naive: Linear-Depth Reduction

\[ T = \gamma mG(P - 1) + L(P - 1) \]

Average Parallelism: \( T_1 / T_\infty = W / D = (P - 1)/(P - 1) = 1 \)
Tree and Butterfly AllReduce

\[ T = 2\gamma mG \log P + 2L \log P \]

\[ T = \gamma mG \log P + L \log P \]
Ring (or Pipeline) AllReduce
Ring (or Pipeline) AllReduce

Process 1

Process 2

Process 3

Process 4
Ring (or Pipeline) AllReduce
Ring (or Pipeline) AllReduce

Diagram showing a network of processes connected in a ring. Each process contains a grid with numbers 1 to 4.
AllReduce

Ring:

\[ T = 2\gamma mG(P - 1)/P + 2L(P - 1) \]

Reduce-Scatter-Gather:

\[ T = 2\gamma mG(P - 1)/P + 2L \log P \]

Lower bound:

\[ T \geq 2\gamma mG(P - 1)/P + L \log P \]
Outline

1. Background
2. Parallel Computing and Communication
3. Neural Network Concurrency
4. Parameter Sharing and Consistency
5. Frameworks
6. Conclusion
## Within-Layer Concurrency

### Table 4. Asymptotic Work-Depth Characteristics of DNN Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Eval.</th>
<th>Work (W)</th>
<th>Depth (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation</td>
<td></td>
<td>( O(NCHW) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( \nabla w )</td>
<td></td>
<td>( O(NCHW) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( \nabla x )</td>
<td></td>
<td>( O(NCHW) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Fully Connected</td>
<td></td>
<td>( O(C_{out} \cdot C_{in} \cdot N) )</td>
<td>( O(\log C_{in}) )</td>
</tr>
<tr>
<td>( \nabla w )</td>
<td></td>
<td>( O(C_{in} \cdot N \cdot C_{out}) )</td>
<td>( O(\log N) )</td>
</tr>
<tr>
<td>( \nabla x )</td>
<td></td>
<td>( O(C_{in} \cdot C_{out} \cdot N) )</td>
<td>( O(\log C_{out}) )</td>
</tr>
<tr>
<td>Convolution (Direct)</td>
<td></td>
<td>( O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_{x} \cdot K_{y}) )</td>
<td>( O(\log K_{x} + \log K_{y} + \log C_{in}) )</td>
</tr>
<tr>
<td>( \nabla w )</td>
<td></td>
<td>( O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_{x} \cdot K_{y}) )</td>
<td>( O(\log K_{x} + \log K_{y} + \log C_{in}) )</td>
</tr>
<tr>
<td>( \nabla x )</td>
<td></td>
<td>( O(N \cdot C_{out} \cdot C_{in} \cdot H \cdot W \cdot K_{x} \cdot K_{y}) )</td>
<td>( O(\log K_{x} + \log K_{y} + \log C_{in}) )</td>
</tr>
</tbody>
</table>
Data Parallelism

- Simple and efficient
- Must replicate models, possible GPU out of memory
Model Parallelism

- Conserve GPU memory
- Can work well with multiple GPUs on the same system
- Must share minibatch with every worker
- Back-prop requires all-to-all communication
Layer-by-layer Concurrency: Pipelining

- Avoid out of memory errors
- Sparse communication: GPUs only communicate with GPU in front of them
- Have to make sure there is overlap in computation
- Latency linear in number of processors
Graph Parallelism
Outline

1. Background
2. Parallel Computing and Communication
3. Neural Network Concurrency
4. Parameter Sharing and Consistency
5. Frameworks
6. Conclusion
Parameter Sharing and Consistency

Centralized Parameter Sharing: Parameter Server

- $T = 2P \frac{\gamma mg}{s} + 2L$
- Ensures consistency
- “Stragglers” cause poor utilization
Asynchronous Parameter Server

- Better utilization, faster training
- Slow agents cause parameter divergence
Stale Synchronous Parameter Server

- Statistical performance vs hardware performance tradeoff
- Having a parameter server at all can bottleneck training
Decentralized Parameter Sharing

Agent 1

\[ w^{(0)} \]

\[ w^{(1)} \]

\[ w^{(2)} \]

\[ \ldots \]

\[ w^{(T)} \]

Agent m

\[ w^{(1)} \]

\[ w^{(2)} \]

\[ \ldots \]

\[ w^{(T)} \]

Time

- \( T = 2\gamma mG(P - 1)/P + 2L \log P \)
- MPI or NCCL can automatically provide a good AllReduce
- Avoids parameter server bottleneck
- “Straggler” problem
Stale Synchronous Decentralized Parameter Sharing

Parameter Sharing and Consistency

Tal Ben-Nun and Torsten Hoefler ETH Zurich

March 6, 2019 26 / 34
Asynchronous Decentralized Parameter Sharing

Agent 1

Agent r

Agent k

Agent m

\[ w^{(1,1)} \rightarrow w^{(2,1)} \rightarrow w^{(3,1)} \]

\[ w^{(1,r)} \rightarrow w^{(2,r)} \rightarrow w^{(3,r)} \rightarrow w^{(4,r)} \rightarrow w^{(5,r)} \]

\[ w^{(1,k)} \rightarrow w^{(2,k)} \rightarrow w^{(3,k)} \]

\[ w^{(1,m)} \rightarrow w^{(2,m)} \rightarrow w^{(3,m)} \]

Time
Outline

1 Background
2 Parallel Computing and Communication
3 Neural Network Concurrency
4 Parameter Sharing and Consistency
5 Frameworks
6 Conclusion
Neural Net Frameworks

- TensorFlow: allows Parameter Server and AllReduce (MPI, NCCL, TCP/IP)
- PyTorch: AllReduce with MPI, NCCL, or gloo
Why not use Hadoop or Spark?

- Don’t natively support GPU runtimes
- Require JVM—slow
- Designed for fault-tolerance, not speed
- Only support data-parallelism
- NNs have cyclic computation graphs (must revisit working sets)
- Must be synchronous
- TF or PyTorch better optimized for deep learning
Outline

1. Background
2. Parallel Computing and Communication
3. Neural Network Concurrency
4. Parameter Sharing and Consistency
5. Frameworks
6. Conclusion
Ongoing Distributed ML Challenges

1. Optimization and Theory
2. Make algorithms more scalable
3. Better software for distributing ML
4. Develop distributed ML systems
   - Consistency
   - Fault tolerance
   - Communication latency
   - Resource management
Up-and-Coming ML Topics

- Impact of compression and quantization?
- Challenges unique to networks with dynamic control flow?
- Best ways to implement utilize graph parallelism?
- Hierarchical tasks?
- Automated architecture searches?
References