Goals for future from last year

1. Finish Scaling up. I want a kilonode program.
2. Native learning reductions. Just like more complicated losses.
3. Other learning algorithms, as interest dictates.
4. Persistent Demonization

Goals for future from last year

1. Finish Scaling up. I want a kilonode program.

Some design considerations

- Hadoop compatibility: Widely available, scheduling and robustness
- Iteration-firendly: Lots of iterative learning algorithms exist
- Minimum code overhead: Don't want to rewrite learning algorithms from scratch
- Balance communication/computation: Imbalance on either side hurts the system

Some design considerations

- Hadoop compatibility: Widely available, scheduling and robustness
- Iteration-firendly: Lots of iterative learning algorithms exist
- Minimum code overhead: Don't want to rewrite learning algorithms from scratch
- Balance communication/computation: Imbalance on either side hurts the system
- Scalable: John has nodes aplenty

Current system provisions

- Hadoop-compatible AllReduce
- Various parameter averaging routines
- Parallel implementation of Adaptive GD, CG, L-BFGS
- Robustness and scalability tested up to 1K nodes and thousands of node hours

Basic invocation on single machine

```
./spanning_tree
../vw --total 2 --node 0 --unique_id 0 -d $1
--span_server localhost > node_0 2>&1 &
../vw --total 2 --node 1 --unique_id 0 -d $1
--span_server localhost
killall spanning_tree
```

Command-line options

- `--span_server <arg>`: Location of server for setting up spanning tree
- `--unique_id <arg> (=0)`: Unique id for cluster parallel job
- `--total <arg> (=1)`: Total number of nodes used in cluster parallel job
- `--node <arg> (=0)`: Node id in cluster parallel job

Basic invocation on a non-Hadoop cluster

- Spanning-tree server: Runs on cluster gateway, organizes communication

  ```
  ./spanning_tree
  ```

- Worker nodes: Each worker node runs VW

  ```
  ./vw --span_server <location> --total <t> --node
  <n> --unique_id <u> -d <file>
  ```

Basic invocation in a Hadoop cluster

- Spanning-tree server: Runs on cluster gateway, organizes communication

  ```
  ./spanning_tree
  ```

- Map-only jobs: Map-only job launched on each node using Hadoop streaming

  ```
  hadoop jar $HADOOP_HOME/hadoop-streaming.jar
  -Dmapred.job.map.memory.mb=2500 -input <input>
  -output <output> -file vw -file runvw.sh -mapper
  ´runvw.sh <output> <span_server>´ -reducer NONE
  ```

  - Each mapper runs VW
  - Model stored in <output>/model on HDFS
  - runvw.sh calls VW, used to modify VW arguments

mapscript.sh example

//Hadoop-streaming has no specification for number of mappers, we calculate it indirectly

```
total=<total data size>
mapsize=`expr $total / $nmappers`
maprem=`expr $total % $nmappers`
mapsize=`expr $mapsize + $maprem`
```

`./spanning_tree` //Starting span-tree server on the gateway
//Note the argument min.split.size to specify number of mappers

```
hadoop jar $HADOOP_HOME/hadoop-streaming.jar
-Dmapred.min.split.size=$mapsize
-Dmapred.map.tasks.speculative.execution=true -input
$in_directory -output $out_directory -file ../vw -file
runvw.sh -mapper runvw.sh -reducer NONE
```

Communication and computation

- Two main additions in cluster-parallel code:
    - Hadoop-compatible AllReduce communication
    - New and old optimization algorithms modified for AllReduce

Communication protocol

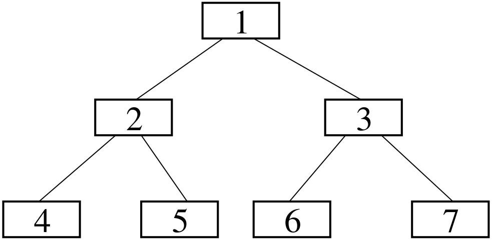
- Spanning-tree server runs as daemon and listens for connections
- Workers via TCP with a node-id and job-id
- Two workers with same job-id and node-id are duplicates, faster one kept (speculative execution)
- Available as mapper environment variables in Hadoop
  - `mapper=`printenv mapred_task_id | cut -d "_" -f 5``
  - `mapred_job_id=`echo $mapred_job_id | tr -d ´job_´``

Communication protocol contd.

- Each worker connects to spanning-tree sever
- Server creates a spanning tree on the n nodes, communicates parent and children to each node
- Node connects to parent and children via TCP
- AllReduce run on the spanning tree

AllReduce

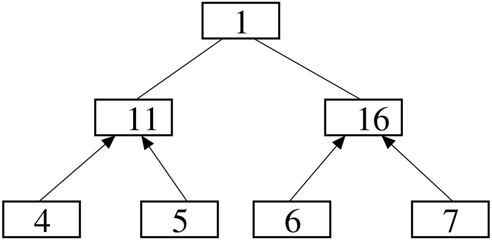
- Every node begins with a number (vector)



- Extends to other functions: max, average, gather, . . .

AllReduce

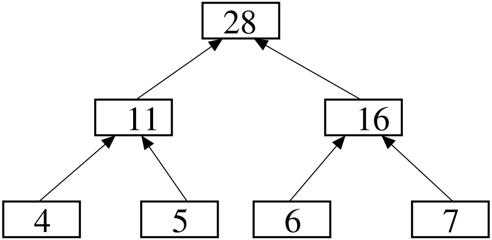
- Every node begins with a number (vector)



- Extends to other functions: max, average, gather, . . .

AllReduce

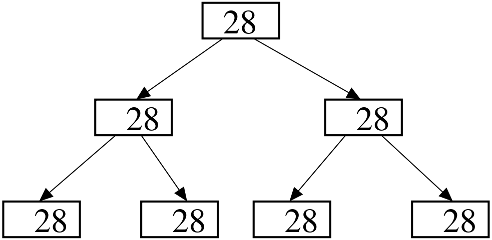
- Every node begins with a number (vector)



- Extends to other functions: max, average, gather, ...

AllReduce

- Every node begins with a number (vector)
- Every node ends up with the sum



- Extends to other functions: max, average, gather, ...

AllReduce Examples

- Counting: $n =$ `allreduce(1)`
- Average: `avg = allreduce(`n_i`)/allreduce(1)`
- Non-uniform averaging: `weighted_avg = allreduce(`$n_i w_i$`)/allreduce(`w_i`)`
- Gather: `node_array = allreduce(`$\{0, 0, \ldots, \underbrace{1}_{i}, \ldots, 0\}$`)`

AllReduce Examples

- Counting: $n = \texttt{allreduce(1)}$
- Average: `avg = allreduce(`n_i`)/allreduce(1)`
- Non-uniform averaging: `weighted_avg = allreduce(`$n_i w_i$`)/allreduce(`w_i`)`
- Gather: `node_array = allreduce(`$\{0, 0, \ldots, \underbrace{1}_{i}, \ldots, 0\}$`)`
- Current code provides 3 routines:
  - `accumulate(<params>)`: Computes vector sums
  - `accumulate_scalar(<params>)`: Computes scalar sums
  - `accumulate_avg(<params>)`: Computes weighted and unweighted averages

Machine learning with AllReduce

- Previously: Single node SGD, multiple passes over data
- Parallel: Each node runs SGD, averages parameters after every pass (or more often!)
- Code change:

```
 if(global.span_server != "") {
     if(global.adaptive)

accumulate_weighted_avg(global.span_server,
params->reg);
     else
          accumulate_avg(global.span_server,
params->reg, 0);
}
```

- Weighted averages computed for adaptive updates, weight features differently

Machine learning with AllReduce contd.

- L-BFGS requires gradients and loss values
- One call to AllReduce for each
- Parallel synchronized L-BFGS updates
- Same with CG, another AllReduce operation for Hessian
- Extends to many other common algorithms

Communication and computation

- Two main additions in cluster-parallel code:
  - Hadoop-compatible AllReduce communication
  - New and old optimization algorithms modified for AllReduce

Hybrid optimization for rapid convergence

- SGD converges fast initially, but slow to squeeze the final bit of precision
- L-BFGS converges rapidly towards the end, once in a good region

Hybrid optimization for rapid convergence

- SGD converges fast initially, but slow to squeeze the final bit of precision
- L-BFGS converges rapidly towards the end, once in a good region

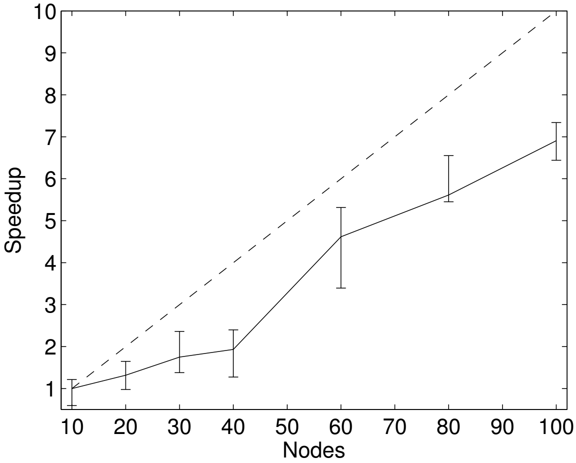
Hybrid optimization for rapid convergence

- SGD converges fast initially, but slow to squeeze the final bit of precision
- L-BFGS converges rapidly towards the end, once in a good region

- Each node performs few local SGD iterations, averaging after every pass
- Switch to L-BFGS with synchronized iterations using AllReduce
- Two calls to VW

Speedup

- Near linear speedup



Hadoop helps

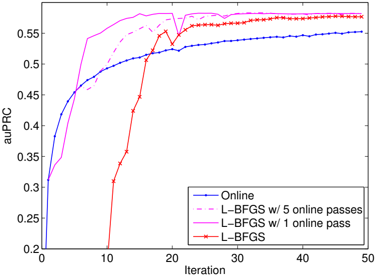
- Naïve implementation driven by slow node
- Speculative execution ameliorates the problem

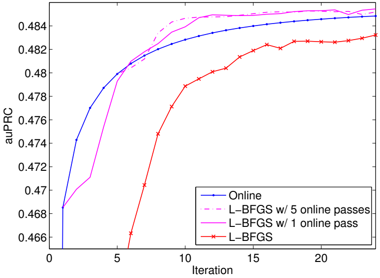
Table: Distribution of computing time (in seconds) over 1000 nodes. First three columns are quantiles. The first row is without speculative execution while the second row is with speculative execution.

	5%	50%	95%	Max	Comm. time
Without spec. exec.	29	34	60	758	26
With spec. exec.	29	33	49	63	10

Fast convergence

- auPRC curves for two tasks, higher is better





Conclusions

- AllReduce quite general yet easy for machine learning
- Marriage with Hadoop great for robustness
- Hybrid optimization strategies effective for rapid convergence
- John gets his kilonode program