I/O Bottleneck Investigation in Deep Learning Systems

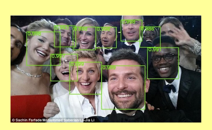
Sarunya Pumma,[1,2] Min Si,[2] Wu-chun Feng,[1] and Pavan Balaji[2]

[1]Virginia Tech, [2]Argonne National Laboratory

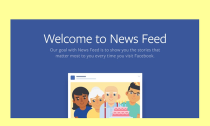
Motivation

Deep Learning & Challenges

Robotics
Asimo
(Honda)



Facial Recognition
Deep Dense Face Detector
(Yahoo Labs)

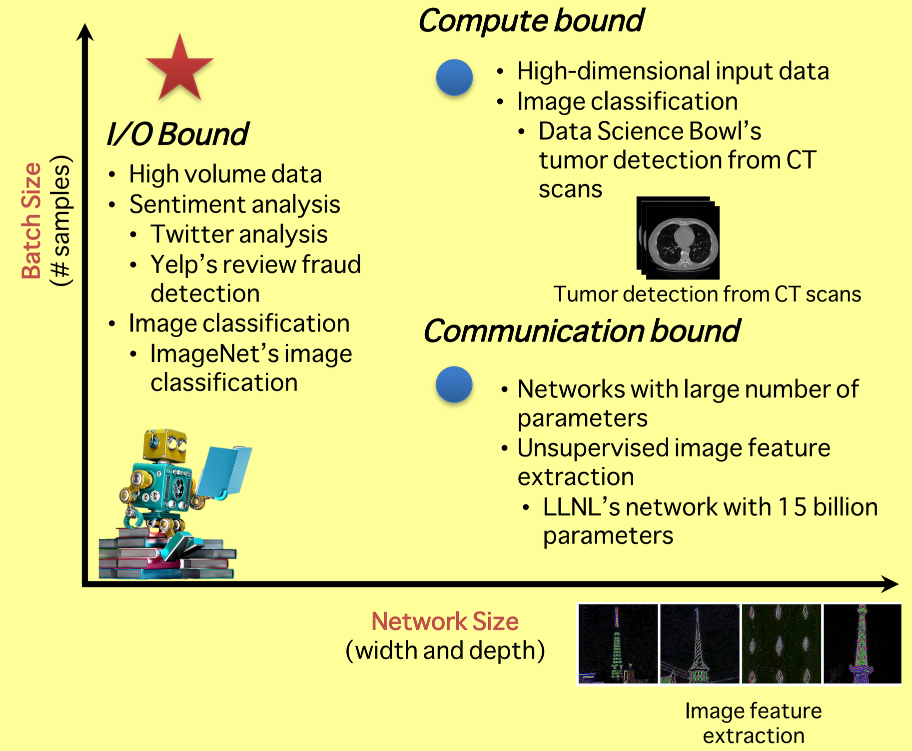


Offline & Online Data Analytics
Real Time News Feed
(Facebook)

In the past decade ...

- 10 – 20x improvement in *processor* speed
- 10 – 20x improvement in *network* speed
- Only 1.5x improvement in *I/O* performance

I/O will eventually become a bottleneck for most computations

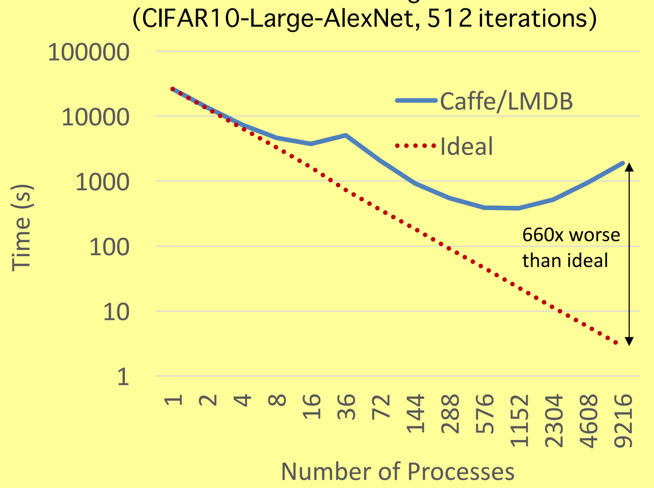


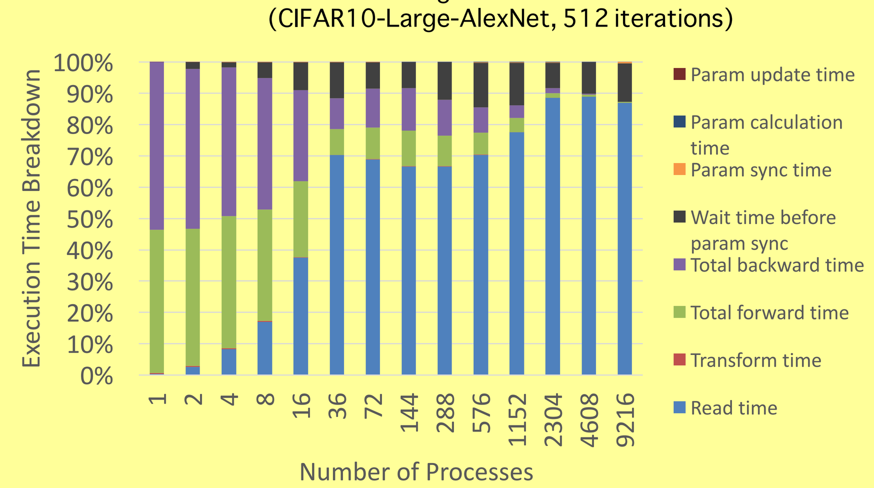
I/O Bound
- High volume data
- Sentiment analysis
  - Twitter analysis
  - Yelp's review fraud detection
- Image classification
  - ImageNet's image classification

Compute bound
- High-dimensional input data
- Image classification
  - Data Science Bowl's tumor detection from CT scans

Communication bound
- Networks with large number of parameters
- Unsupervised image feature extraction
  - LLNL's network with 15 billion parameters

Deep Learning Scaling





- Caffe/LMDB is 660x worse than ideal for 9216 processes
- Read time takes up 90% of the total training time for 9216 processes
- I/O bottleneck is caused by **five** major problems
  1. Interprocess contention -- results in excessive number of context switches
  2. Implicit I/O inefficiency -- OS fully controls I/O
  3. Sequential data access restriction -- arbitrary database access is not allowed in LMDB
  4. Inefficient I/O block size -- I/O request size is too small to be efficient
  5. I/O randomization -- abundant readers participating in I/O at the same time
- We proposed 6 optimizations that address 5 problems in state of the art I/O subsystem of deep learning

Experiment Information
Dataset: CIFAR10-Large
Network: AlexNet
Batch size: 18,432
Training iterations: 512
Framework: Caffe
Testbed: LCRC Bebop
(Each node: 36 cores Intel Broadwell, 128 GB memory)

LMDB Inefficiencies

Caffe's I/O Subsystem: LMDB

- Uses Lightning Memory-mapped database (LMDB) for accessing the dataset
- B+-tree representation of the data
- Database is mapped to memory using `mmap` and accessed through direct buffer arithmetic
- Virtual memory allocated for the size of the full file
  - Specific physical pages dynamically loaded by the OS on-demand

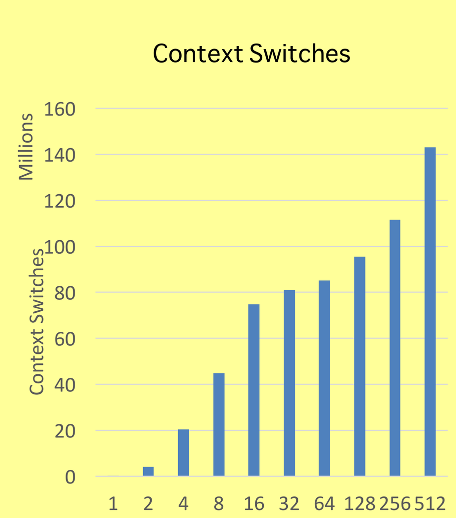
Pros: makes it easy to manipulate complex data structures (e.g., B+ trees) since LMDB can think of it as fully in-memory

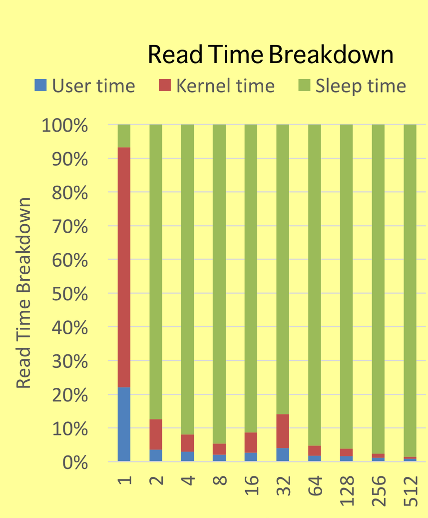
Cons: OS has very little knowledge of the access model and parallelism making it hard to optimize

Problem 1: `Mmap`'s Interprocess Contention

Underlying I/O in `mmap` relies on the CFS scheduler to wake up processes after I/O has been completed

- Processes are put to sleep while waiting for I/O to complete
- I/O completion interrupt is a bottom-half interrupt
  - The handler does not have knowledge about the specific process that triggered the I/O operation
- Every process that is waiting for I/O is marked as runnable
- Every reader is woken up each time an I/O interrupt comes in
- This causes a large number of unnecessary context switches

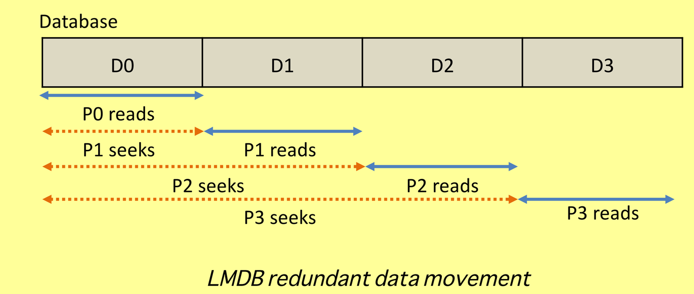




LMDB Inefficiencies (cont.)

Problem 2: Sequential Data Access Restriction

- LMDB data access is sequential in nature due to the B+-tree structure
- There is no way to randomly access a data record
  - All branch nodes associated with the previous records must be read before accessing a particular record
- When multiple processes read the data, they read extra data
- Different processes do different amount of work, causing skew

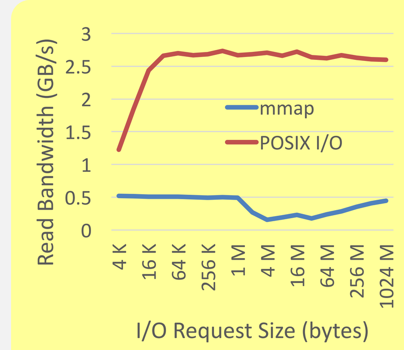


LMDB redundant data movement

Problem 3: `Mmap`'s Workflow Overheads

- Since `mmap` performs implicit I/O, the user has no control over when an I/O operation is issued.
- To showcase this overhead, we developed a microbenchmark to read a 256 GB file using a single reader on a single machine
  - `Mmap` benchmark uses `memcpy` on a mmap buffer
  - POSIX I/O benchmark uses pread
- `mmap`'s read bandwidth is approximately 2.5x lower than that of POSIX I/O

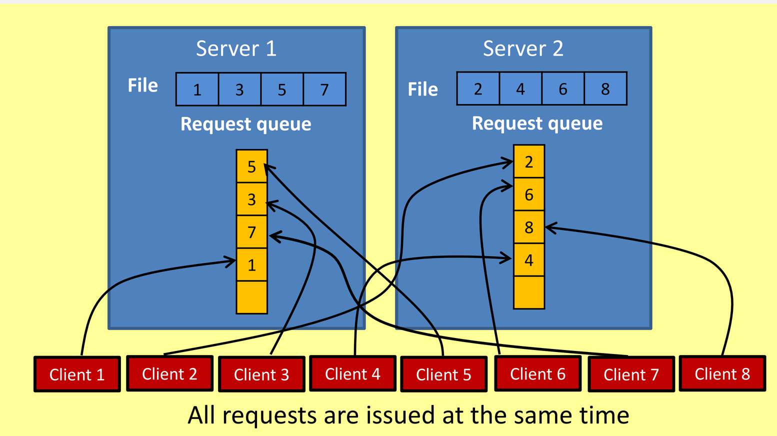
Problem 4: I/O Block Size Management



- As the number of processes increases, subbatch is smaller
- POSIX I/O benefits from larger block size, while `mmap` does not
- Migrating LMDB to use direct I/O and larger block size can give a significant performance improvement

Problem 5: I/O Randomization

- I/O requests are typically out of order in parallel I/O
  - A large number of processes need to divide a large file into smaller pieces and each process needs to access a part of it
  - Each process issues an I/O request at the same time
  - I/O requests do not arrive at the I/O server processes in any specific order as each process is independent
  - This causes the server processes to access the file in a nondeterministic fashion



All requests are issued at the same time

Our Solution: LMDBIO

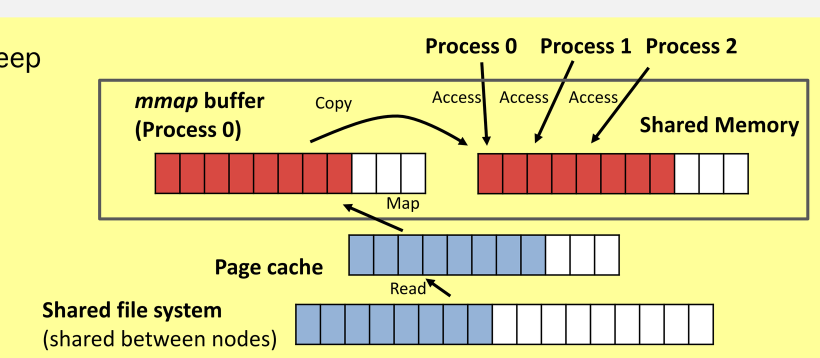
Summary of LMDBIO Optimizations

Library	Optimization	Reducing Interprocess Contention	Explicit I/O	Eliminating Sequential Seek	Managing I/O Size	Reducing I/O Randomization
LMDB	-					
LMDBIO	LMM	✔				
	LMM-DM	✔		(partial)		
	LMM-DIO	✔	✔			
	LMM-DIO-PROV	✔	✔	✔		
	LMM-DIO-PROV-COAL	✔	✔	✔	✔	
	LMM-DIO-PROV-COAL-STAG	✔	✔	✔	✔	✔

LMDBIO-LMM

Optimization: Take into account **data access pattern** of deep learning and **Linux's I/O scheduling** to reduce `mmap`'s contentions

- Localized `mmap`
  - Only one process does `mmap` on each node
  - Using MPI shared-memory (MPI-3) to share data
- Even LMDBIO has extra copy (from `mmap` to shared memory), Caffe still gains benefit from LMDBIO



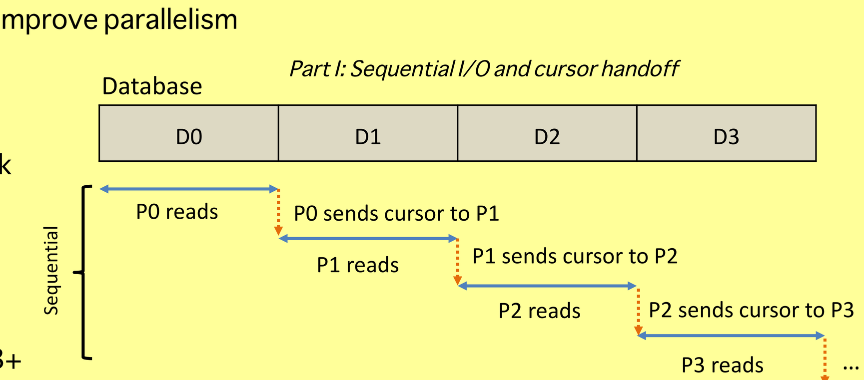
Sarunya Pumma, Min Si, Wu-chun Feng and Pavan Balaji. *Towards Scalable Deep Learning via I/O Analysis and Optimization.* IEEE International Conference on High Performance Computing and Communications (HPCC). Dec. 18-20, 2017, Bangkok, Thailand.

LMDBIO-LMM-DM

Optimization: **coordinate between reader processes** to improve parallelism

Part I: Serializing I/O

- Serialize data reading and coordinate between processes
- Each process reads its data and sends the higher rank process the location to start fetching its data from
- This allows NO extra data reading: number of bytes read is EXACT… but I/O is done sequentially



Portable Cursor Representation

- LMDB calls the position indicator for a record within B+ tree a "cursor"
  - Not a simple offset from the start of file
  - It contains the complete path of the record's parent branch nodes (multiple pointers), a pointer to the page header, and access flags
- It is not trivial to port pointers across processes as virtual address spaces are different

Portable Cursor Representation (cont.)

- Our solution: **symmetric address space**
  - Every process memory-maps the database file to the same memory location
  - Allowing the pointers within the B+ tree to be portable across processes

Sarunya Pumma, Min Si, Wu-chun Feng and Pavan Balaji. *Parallel I/O Optimizations for Scalable Deep Learning.* IEEE International Conference on Parallel and Distributed Systems (ICPADS). Dec. 15-17, 2017, Shenzhen, China.

Our Solution: LMDBIO (cont.)

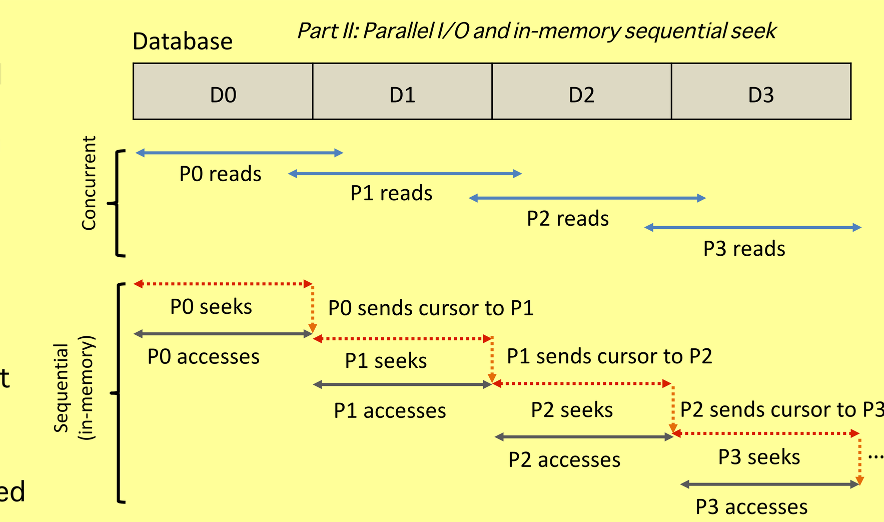
LMDBIO-LMM-DM (cont.)

Part II: Speculative Parallel I/O

- Each process estimates pages that it will need and speculatively fetches pages to memory in parallel
- Then each process sequentially seeks the location for another processes and sends the cursor to the next higher rank process
  - The expectation is that the seek can be done entirely in memory
- Once the sequential seek is done, each reader can perform actual data access
- This adds a small amount of extra data reading, but allows parallel I/O

Estimation of Speculative I/O

- The estimation of number of pages to fetch is based on the first record's data size
  - I.e., CIFAR10-Large record's size is 3 KB, which is ~1 page. To read *n* records, it needs to fetch *n* pages
- The estimation of the read offset is performed in the same fashion
- Estimation of the "approximate" start and end location for each process is important
  - If the estimate is completely wrong, we will end up reading up to 2x the dataset size (still better than the LMDB)

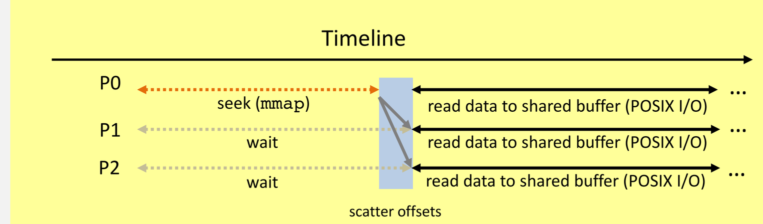


Estimation of Speculative I/O (cont.)

- We use a history-based training for our estimation
  - We correct our estimate in each iteration depending on the actual data read in all of the previous iterations
  - The general ideal of out correction is that we attempt to expand the speculative boundaries to reduce the number of missed pages
  - Initial iterations might be slightly inaccurate, but we converge fairly quickly (1-2 iterations)

LMDBIO-LMM-DIO

Optimization: Replace `mmap` with POSIX I/O



- To use direct I/O, we need to know the position of each data record
  - The root process gets offsets of all data samples by seeking the database using `mmap`
  - Sequential seek is unavoidable because the offsets are not deterministic
- Other reader processes receive their offsets from root and perform data reading using POSIX I/O
- Readers share data using MPI shared buffer as same as LMM

LMDBIO-LMM-DIO-PROV

Optimization: Utilize provenance information to entirely replace `mmap` with POSIX I/O

- Making a case for storing data provenance information for deep learning (how the data was created)
- LMDB's database layout can be deterministic only if the information of how it is created is provided
- We can compute exactly where the data pages are located
- Sequential seek can be completely eliminated
- All I/O operations can be done via direct I/O (`mmap` is completely removed)

Important Notes

- Provenance information is not stored in the original LMDB format
  - This is an extension that we are proposing
- We use a separate auxiliary file to store this information
  - This file can be created while the database is being generated or later using a one-time read of the database
  - It is much smaller than the dataset itself (a few hundred bytes)

LMDBIO-LMM-DIO-PROV-COAL

Optimization: Coalesce multiple batches of data to be read at once to allow direct I/O to benefit from large I/O size

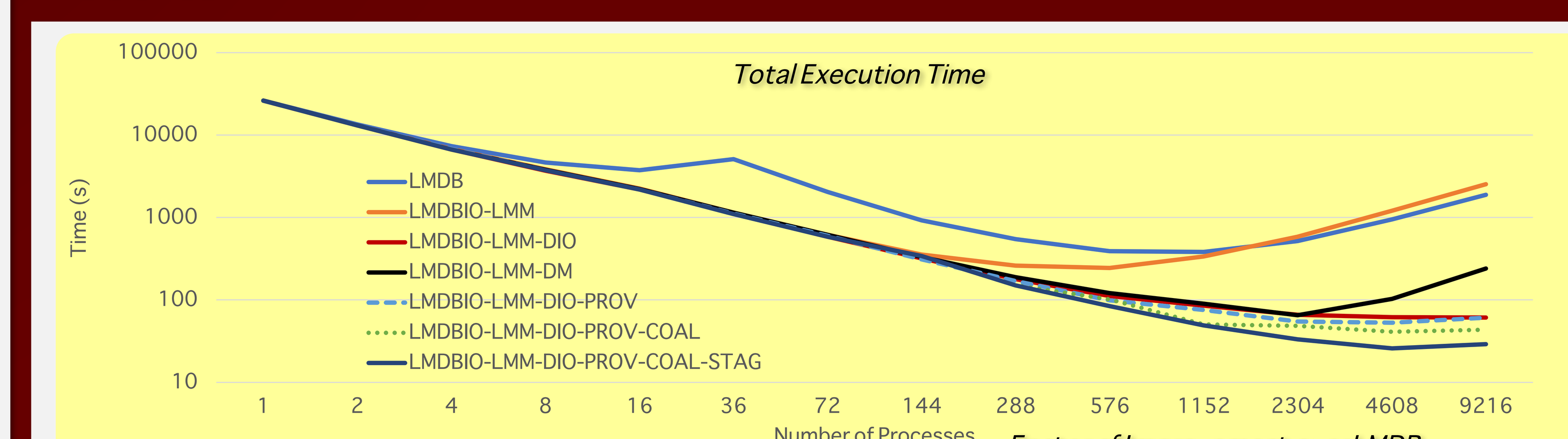
- We read a larger chunk of data to enlarge I/O time to eliminate the skew in I/O
  - A constant amount of memory is kept aside for data reading
- We read multiple batches of data at once

LMDBIO-LMM-DIO-PROV-COAL-STAG

Optimization: Adopt I/O staggering to reduce I/O randomization

- I/O staggering technique orders the requests
  - Readers are divided into multiple groups with the same number of members
  - Only one group can perform data reading at a time
- MPI_Send and MPI_Recv are used in the implementation

Results



Platform: Argonne's Blues
- InfiniBand Qlogic QDR
- 110 TB GPFS storage
- Each node
  - 2 Sandy Bridge 2.6 GHz Pentium Xeon (16 cores, hyperthreading disabled)
  - 64 GB memory
  - 15 GB RAM disk

Dataset: CIFAR10-Large
Network: AlexNet
MPI: MVAPICH-2.2