DATA ANALYSIS AND MACHINE LEARNING WITH INTEL® ARCHITECTURE

One-Day Workshop

Andrey Vladimirov and Ryo Asai
Colfax International — *colfaxresearch.com*

February 2017

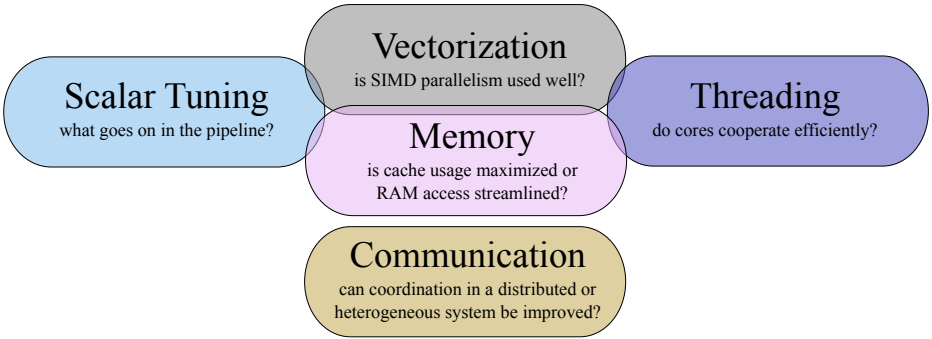
DISCLAIMER

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

§1. QUICK REVIEW OF YESTERDAY

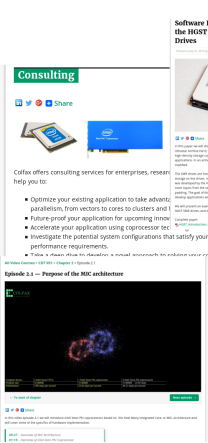
Code Modernization

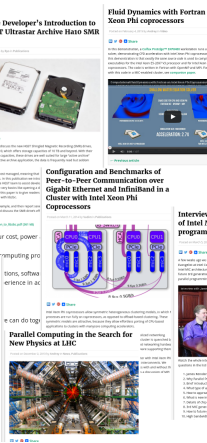
Optimizing software to better utilize features available in modern computer architectures.




THE "HOW" SERIES TRAINING
DEEP DIVE
WITH CODE MODERNIZATION EXPERTS
It's free
→ HOWSERIES.COM
*10x 2-hour sessions | 24-hour 2-weeks remote access to a system

COLFAX RESEARCH





http://colfaxresearch.com/

§2. ML IN THE CONTEXT OF HPC

THREE APPROACHES

Low Level Approach

Apply code modernization techniques to frameworks/applications.

- Colfax Research Website, HOW series, Intel Modern Code page etc.

High Level Approach

Use high level libraries that are pre-optimized for modern architectures.

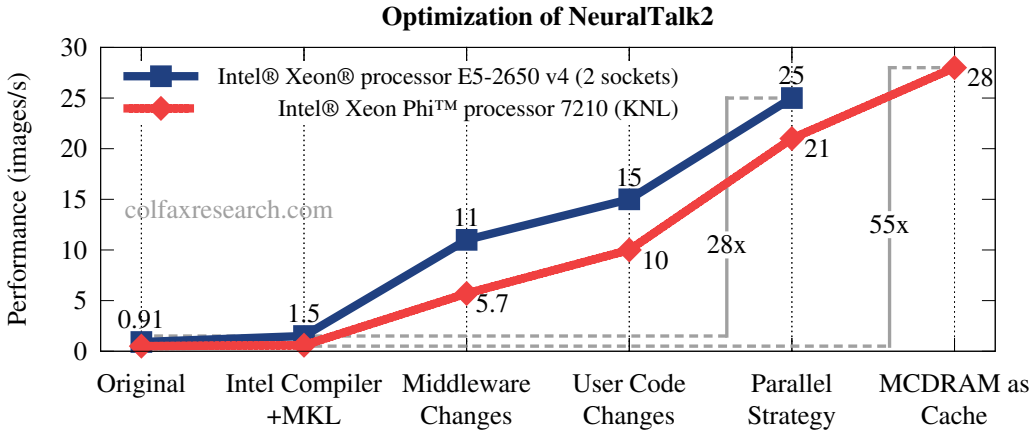
- IntelCaffe, TensorFlow, Scikit-learn etc.

Middle Ground Approach

Integrate pre-optimized kernels into frameworks/applications.

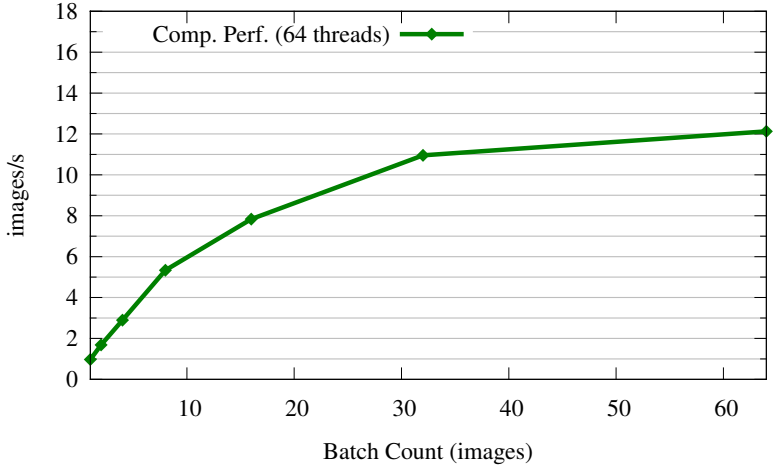
- Intel® MKL DNN primitives, Intel® DAAL, Intel® MKL DNN etc.

CASE STUDY: VGG-NET ON TORCH



Colfax Research Summary Paper

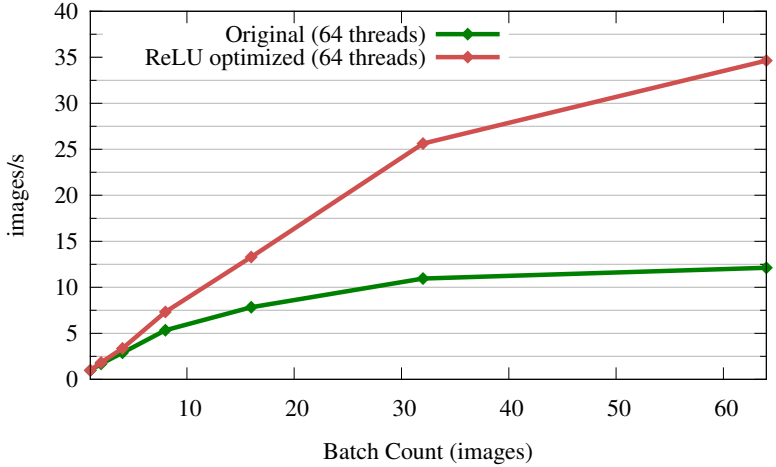
BASE TORCH PERFORMANCE



By Layer:

- ReLU: 66%
- Conv: 30%
- MaxPool: 3%
- Other: <1%

PERFORMANCE AFTER RELU OPTIMIZATION



RELU -> 160x boost

By Layer:

- ReLU: 1%
- Conv: 85%
- MaxPool: 11%
- Other: 3%

FALCON PAPER

https://colfaxresearch.com/falcon-library/

§3. INTEL-OPTIMIZED FRAMEWORKS

INTEL MKL: THE MAGIC PILL

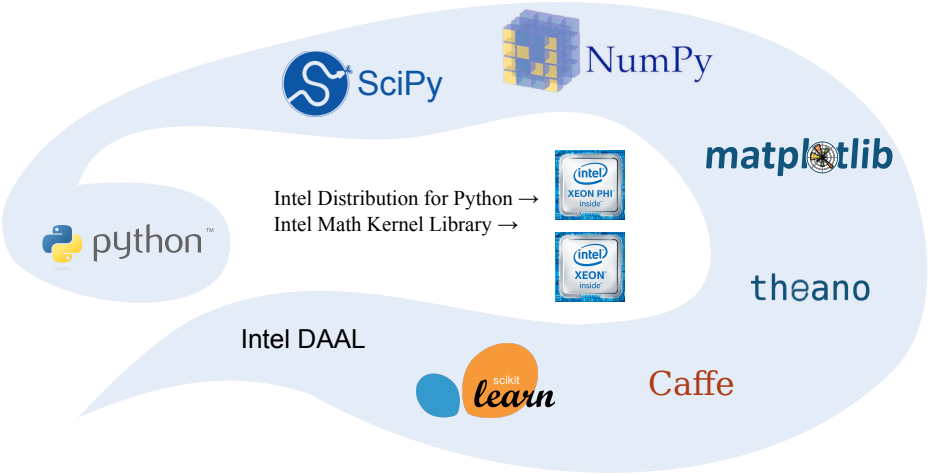
INTEL® MATH KERNEL LIBRARY

Intel® Math Kernel Library (MKL) — standard mathematical functions optimized for Intel architecture.

Dense Linear Algebra	BLAS + PBLAS LAPACK + ScaLAPACK Extended eigensolver
Fourier Transform	Multi-threaded Cluster mode 1D and multi-dimensional
Statistics and Probability	Random number generators Convolution and correlation Summary statistics
Sparse Linear Algebra	SpBLAS Iterative, direct solvers Preconditioners
Numerical Analysis	Partial differential equations Nonlinear optimization Data fitting. Vector math.
Machine Learning (New)	DNN Primitives: Convolution. Inner product. ReLU, LRNC, etc.

INTEL PYTHON

INTEL DISTRIBUTION FOR PYTHON



Portal: software.intel.com/intel-distribution-for-python. See also: CR paper.

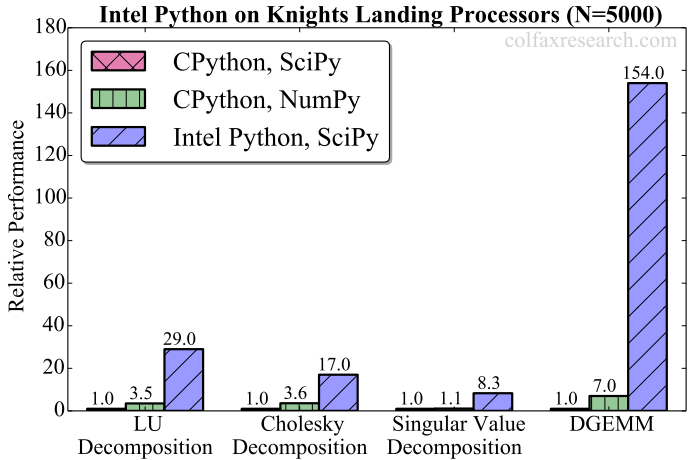
DGEMM WITH SCIPY

```
import numpy as np
from numpy.random import rand as rn
from scipy.linalg.blas import dgemm as dgemm

m=5000
n=5000
k=5000

// Using column-major format with order="F"
A = np.array(np.random.rand(m, n), dtype = np.double, order="F");
B = np.array(np.random.rand(n, k), dtype = np.double, order="F");

// Scipy matrix-matrix multiplication
C=dgemm(alpha=1.0, a=A, b=B, c=C, overwrite_c=1, trans_b=1)
```

DGEMM WITH NUMPY

```
import numpy as np
from numpy.random import rand as rn

m=5000
n=5000
k=5000

// Using row-major by default (you can also set order="C")
A = np.array(np.random.rand(m, n), dtype = np.double);
B = np.array(np.random.rand(n, k), dtype = np.double);

// Numpy matrix-matrix multiplication
C = np.dot(A, B)
```

INTEL PYTHON PERFORMANCE



Portal: software.intel.com/intel-distribution-for-python. See also: CR paper.

MISC. SCIPY WORKLOADS

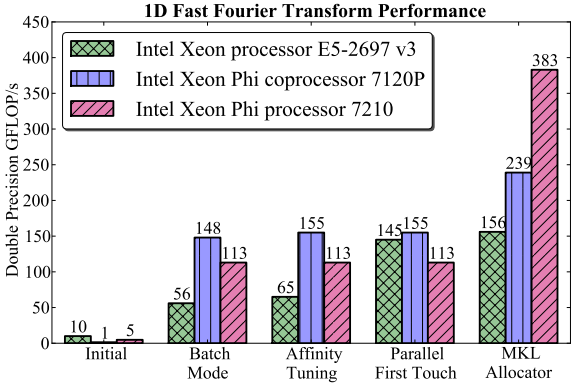
```
from scipy.linalg import lu as lu;
# ... #
A = np.array(np.random.rand(k, k), dtype = np.double, order='F')
P,L,U = lu(A, permute_l=False, overwrite_a=True, check_finite=False)
```

```
from scipy.linalg import cholesky as cholesky
# ... #
A = np.array(tempA, dtype=np.double, order='F')
L = cholesky(A, overwrite_a=True, check_finite=False)
```

```
from scipy.linalg import svd as svd
# ... #
A = np.array(np.random.rand(k, k), dtype = np.double, order='F');
U, s, V = svd(A)
```

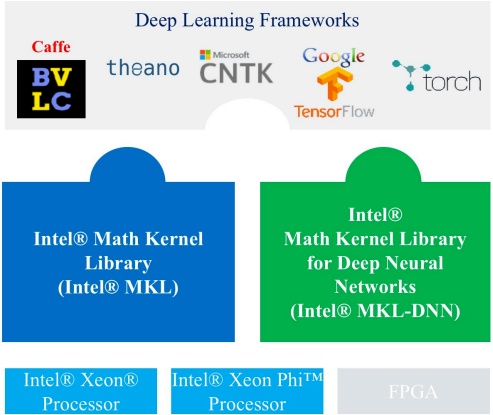
TUNING MAY BE NECESSARY

$2 \cdot 10^5$ FFTs of size 2048. See HOW “KNL” for details.



§4. MIDDLE GROUND: ML PERFORMANCE LIBRARIES

INTEL MKL AND INTEL MKL-DNN



Intel® MKL	Intel® MKL-DNN
DNN primitives + wide variety of other math functions	DNN primitives
C DNN APIs (C++ future)	C/C++ DNN APIs
Binary distribution	Open source DNN code[1]
Free community license. Premium support available as part of Intel® Parallel Studio XE	Apache 2.0 license
Broad usage DNN primitives; not specific to individual frameworks	Multiple variants of DNN primitives as required for framework integrations
Quarterly update releases	Rapid development ahead of Intel MKL releases

[1] GEMM building blocks are binary.

slide credit: Intel corp.

STAND-ALONE EXAMPLE: CONVOLUTION

```
// Creating MKL DNN primitive object
dnnPrimitive_t convFwd;
dnnConvolutionCreateForward_F32(&convFwd, NULL, dnnAlgorithmConvolutionDirect,
                                dim, input_dims, output_dims, filter_dims,
                                conv_strides, padding, dnnBorderZeros);

// Creating the needed data buffer
void* conv_res[dnnResourceNumber];
conv_res[dnnResourceSrc] = (void*) input;
conv_res[dnnResourceFilter] = (void*) filter;
conv_res[dnnResourceDst] = (void*) output;

// Execute the workload
dnnExecute_F32(pConvFwd, conv_res);
```

For more: Intel MKL documentation on DNN primitives

EXAMPLE WITH INTEL CAFFE

EXAMPLE INTEGRATION: INTEL CAFFE

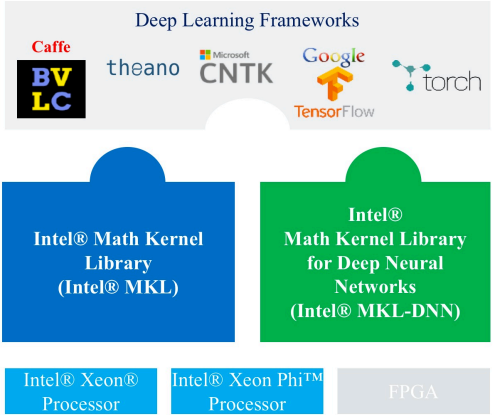
GitHub link: https://github.com/intel/caffe/

Example layer implementations: caffe/src/caffe/layers/mkl_*.cpp

```
// Grabbing parameters from Caffe Layers
PoolingParameter pool_param = this->layer_param_.pooling_param();
channels_ = bottom[0]->channels();
height_ = bottom[0]->height();
width_ = bottom[0]->width();
num_ = bottom[0]->num();
// ... //
kernel_h_ = pool_param.kernel_h();  kernel_w_ = pool_param.kernel_w();
// ..... //

// Creating the math kernel from these parameters
status = dnnPoolingCreateForward<Dtype>( /* ... */ );
```

§5. HIGH LEVEL: ML FRAMEWORKS

INTEL MKL AND INTEL MKL-DNN



Intel® MKL	Intel® MKL-DNN
DNN primitives + wide variety of other math functions	DNN primitives
C DNN APIs (C++ future)	C/C++ DNN APIs
Binary distribution	Open source DNN code[1]
Free community license. Premium support available as part of Intel® Parallel Studio XE	Apache 2.0 license
Broad usage DNN primitives; not specific to individual frameworks	Multiple variants of DNN primitives as required for framework integrations
Quarterly update releases	Rapid development ahead of Intel MKL releases

[1] GEMM building blocks are binary.

slide credit: Intel corp.

(CURRENTLY) AVAILABLE LIBRARIES

Machine Learning frameworks that have been optimized for Intel Architectures

- Intel Caffe - Python - Deep Learning
  GitHub
- Scikit-learn - Python - Classical ML
  Part of Intel Python
- Intel Theano/Keras - Python - Deep Learning
  Github Theano/Keras/
- Intel Torch - Lua - Classical ML, Deep learning
  GitHub
- Intel DAAL - Java/C++/Python - Classical ML, Deep Learning

... and more to come!

USING LIBRAIES ON THE CLUSTER

- Intel Caffe

```
~> /opt/intel/caffe/build/caffe
```

- Scikit-learn

```
import sklearn
```

- Theano/Keras

```
import theano
import keras
```

- Intel DAAL

```
1 #include "daal"
```

§6. DISTRIBUTED DEEP LEARNING

"FLOPS ARE CHEAP"?

Theoretical estimates, Intel® Xeon E5-2697 V3 processor

$$\text{Performance} = 28\ \text{cores} \times 2.7\ \text{GHz} \times (256/64)\ \text{vec.lanes} \times 2\ \text{FMA} \times 2\ \text{FPU} \approx 1.2\ \text{TFLOP/s}$$

$$\text{Required Data Rate} = 1.2\ \text{TFLOP/s} \times 8\ \text{bytes} \approx 10\ \text{TB/s}$$

$$\text{OPA Max Bandwidth} = 12.5\ \text{GB/s} \approx 0.01\ \text{TB/s}$$

$$\text{Ratio} = 10/0.01 \approx 1000\ \text{(FLOPs)/(Memory Transferred)}$$

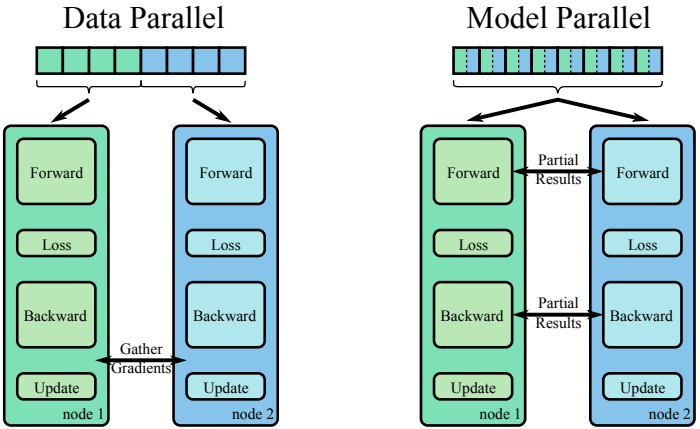
To put it short...

Difficulty of Distributed Computation

In the time it takes to transfer one data element, processors can do thousands of operation on one data element.

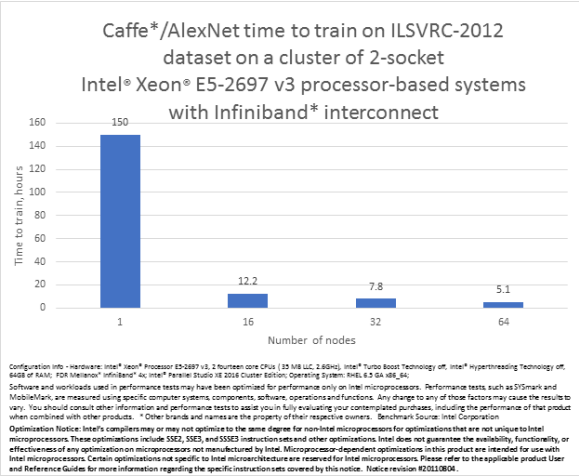
DISTRIBUTED COMPUTATION FOR NEURAL NETWORKS

Because of issue with large mini-batches, it is not easy to scale DNN.



Intel Machine Learning Scaling Library

CAFFE SCALING



Source: Intel® Corporation. (Caffe* Training on Multi-node Distributed-memory Systems Based on Intel® Xeon® Processor E5 Family)

MACHINE LEARNING FRAMEWORK: INTEL® DAAL

ALGORITHMS IN DAAL

Analysis

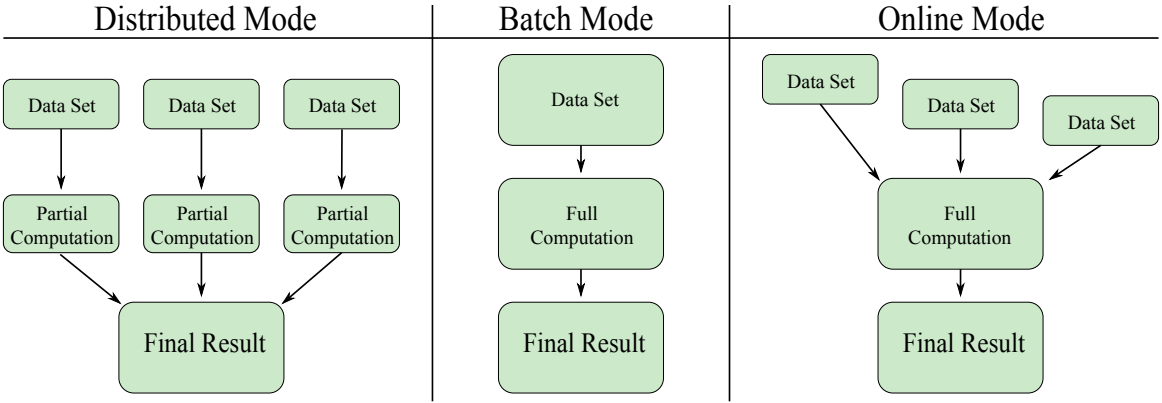
- Low Order Moments
- Quantile
- Correlation and Variance
- Cosine Distance Matrix
- Correlation Distance Matrix
- K-Means Clustering
- Principal Component Analysis
- Cholesky Decomposition
- Singular Value Decomposition
- QR Decomposition
- Expectation-Maximization
- Multivariate Outlier Detection
- Univariate Outlier Detection
- Association Rules
- Kernel Functions
- Quality Metrics

Training & prediction

- Regression
  - Linear/Ridge Regresion
- Clasification
  - Naive Bayes Classifier
  - Boosting
  - SVM
  - Neural Networks
  - Multi-Class Classifier

Portal: DAAL page. See also: intro article, CR papers.

ALGORITHMS IN DAAL



Portal: DAAL page. See also: intro article, CR papers.

COMMUNICATION FRAMEWORK: MPI

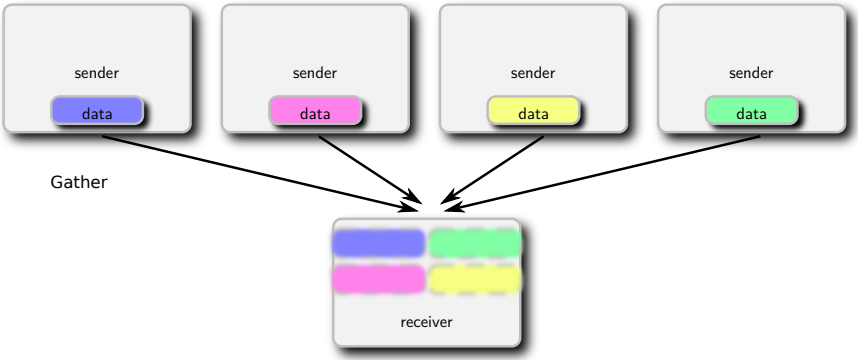
STRUCTURE OF MPI APPLICATIONS: HELLO WORLD

```
#include "mpi.h"
#include <cstdio>
int main (int argc, char *argv[]) {
  MPI_Init (&argc, &argv);  // Initialize MPI envirnmnt
  int rank, size, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Comm_rank (MPI_COMM_WORLD, &rank); // ID of current process
  MPI_Get_processor_name (name, &namelen); // Hostname of node
  MPI_Comm_size (MPI_COMM_WORLD, &size); // Number of processes
  printf ("Hello World from rank %d running on %s!\n", rank, name);
  if (rank == 0) printf("MPI World size = %d processes\n", size);
  MPI_Finalize (); // Terminate MPI environment
}
```

MPICH site contains a list of MPI 3.2 routines

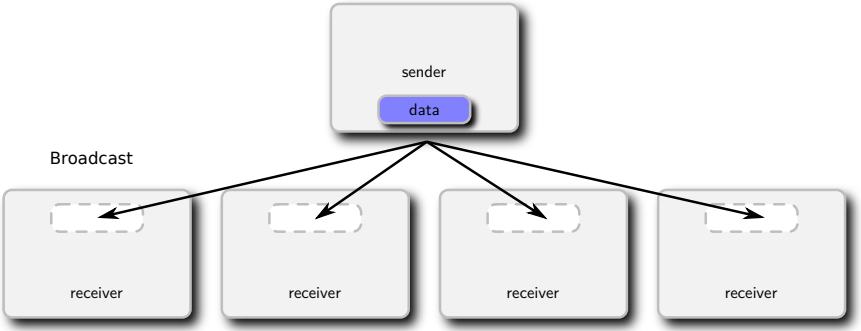
COLLECTIVE COMMUNICATION: GATHER

```
1 int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
2     void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



COLLECTIVE COMMUNICATION: BROADCAST

```
1 int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype,
2     int root, MPI_Comm comm );
```



IMPLEMENTATION

EXAMPLE DISTRIBUTED IMAGE PROCESSING: DAAL

- Algorithm <step1Local> is responsible for the forward/backward propagation.

```
training::Distributed<step1Local> local_net;      // local net algorithm
local_net.compute();                              // forward/backward
part_res = local_net.getPartialResult();          // getting partial result
local_net.input.get(training::inputModel)
                      ->setWeightsAndBiases(wb); // Update the weights/bias
```

- Algorithm <step2Master> is responsible for accumulating the gradient.

```
training::Distributed<step2Master> master_net;    // master net algorithm
master_net.input.add(training::partialResults,    // Add partial result
                                    0, part_res);
master_net.compute();                             // Accumulate gradients
wbModel = master_net.getPartialResult()           // Get Current Model
              ->get(training::resultFromMaster)
                        ->get(training::model);
wb = wbModel->getWeightsAndBiases();              // Extract weights/bias
```

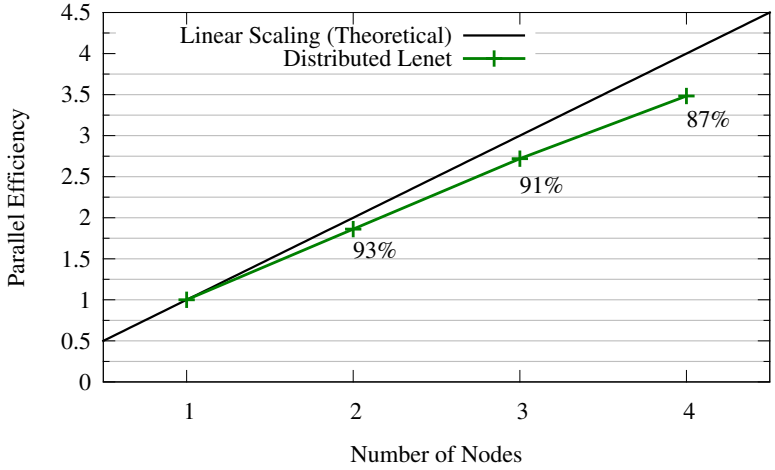
EXAMPLE DISTRIBUTED IMAGE PROCESSING (PART 1)

```
// Computation part of the node with the master net
// Local forward and backward propagation
local_net.compute();
part_res[master_node_id] = local_net.getPartialResult();

// ... Code to store the result into a buffer (char *) ... //

// Send the result to the master node
MPI_Gather(....);

// ... Code to reconstruct the partial result from the buffer... //

// accumulate the partial result from nodes
for(int i = 0; i < num_nodes; i++)
  master_net.input.add(training::partialResults, node, part_res[i]);
master_net.compute();
```

EXAMPLE DISTRIBUTED IMAGE PROCESSING (PART 2)

```
// ... Continuing on the master compute ... //

// Extract the weight/bias from the master net
training::ModelPtr wbModel = master_net.getPartialResult()
                              ->get(training::resultFromMaster)
                              ->get(training::model);
NumericTablePtr wb = wbModel->getWeightsAndBiases();

// ... Code to store weights/bias into a buffer (char*) ... //

// Broadcast the weights/bias to all nodes //
MPI_Bcast(.....);

// ... Code to reconstruct the weights/bias from buffer ... //

// Update the weights on local node
local_net.input.get(training::inputModel)->setWeightsAndBiases(wb);
```

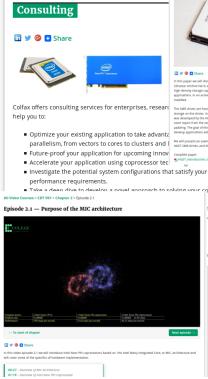
PARALLEL EFFICIENCY

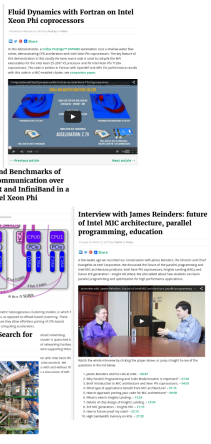


Further performance optimizations and model parallelism are coming soon...

§7. FINAL WORDS

COLFAX RESEARCH





http://colfaxresearch.com/

Thank you for your Attention!