Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Communication Reduction Techniques in Numerical Methods and Deep Neural Networks

PhD Candidate:
Sicong Zhuang

Thesis Director:
Dr. Marc Casas

Tutor:
Dr. Eduard Ayguadé

02/09/2019

Outline

- Background
- Communication reduction in Conjugate Gradient Method
- Communication reduction in DNN training on multi-GPU environments
- Communication reduction in model parallelism for DNN
- Conclusions and future works

Background

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Modern HPC Systems

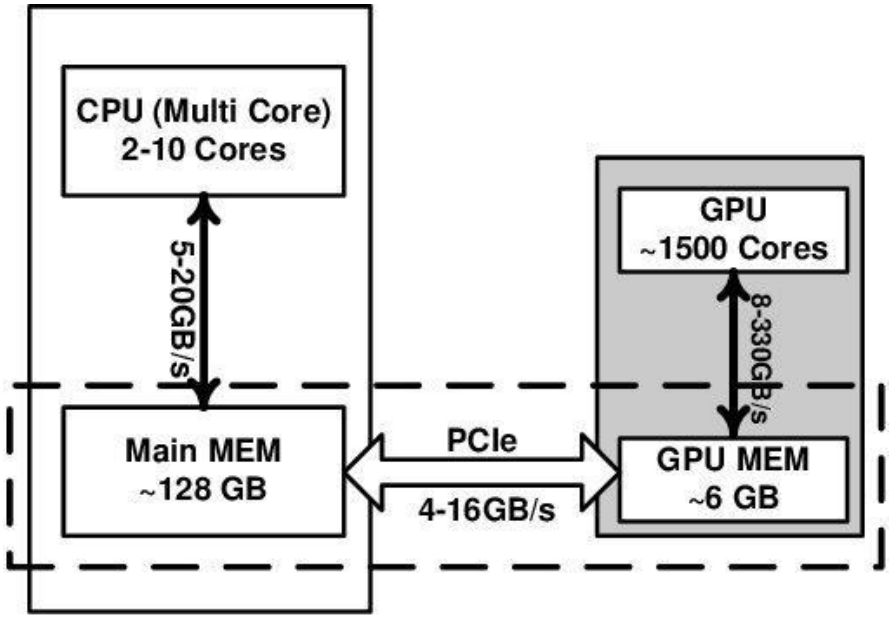
- Massively parallel system
  - Multi-core processor
  - Multi-node cluster
- Shared-memory architecture
  - All the computation units can access the entire memory region
  - Inherently limited by scale
  - Programming via OpenMP
- Distributed-memory architecture
  - Cross-region memory access via means of communication
  - The physical proximity of nodes determines the quality of communication
  - Imbalanced communication creates choke points thus forces some computation units into a stall
  - Programming via MPI

Heterogeneous Computing

- Various accelerators
  - GPU, FPGA, ASIC etc.
  - Massive parallelism, re-configurability and domain-specific efficiency
  - Connected through external buses: PCIe, NVlink etc.

Heterogeneous Computing

- Various accelerators
  - GPU, FPGA, ASIC etc.
  - Massive parallelism, re-configurability and domain-specific efficiency
  - Connected through external buses: PCIe, NVlink etc.
  - Need to offload necessary data from the host to the accelerators
  - Prominent among iterative algorithms



Outline

- Background
- Communication reduction in Conjugate Gradient Method
- Communication reduction in DNN training on multi-GPU environments
- Communication reduction in model parallelism for DNN
- Conclusions and future works

Communication Reduction in Conjugate Gradient Method

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conjugate Gradient Method (CG)

- Conjugate Gradient is a prevalent iterative solver for the numerical solution of system of equations
- Focus on the performance aspect
  - Increase parallelism
  - Reduce the number of synchronization
- Running time of an algorithm is a function of parameters involving
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # synchronization

Conjugate Gradient Method [1]

- Three synchronization points
  - Two separate dot-products (synchronization) per iteration
  - One synchronization between adjacent iterations
- No overlapping between synchronization and computation

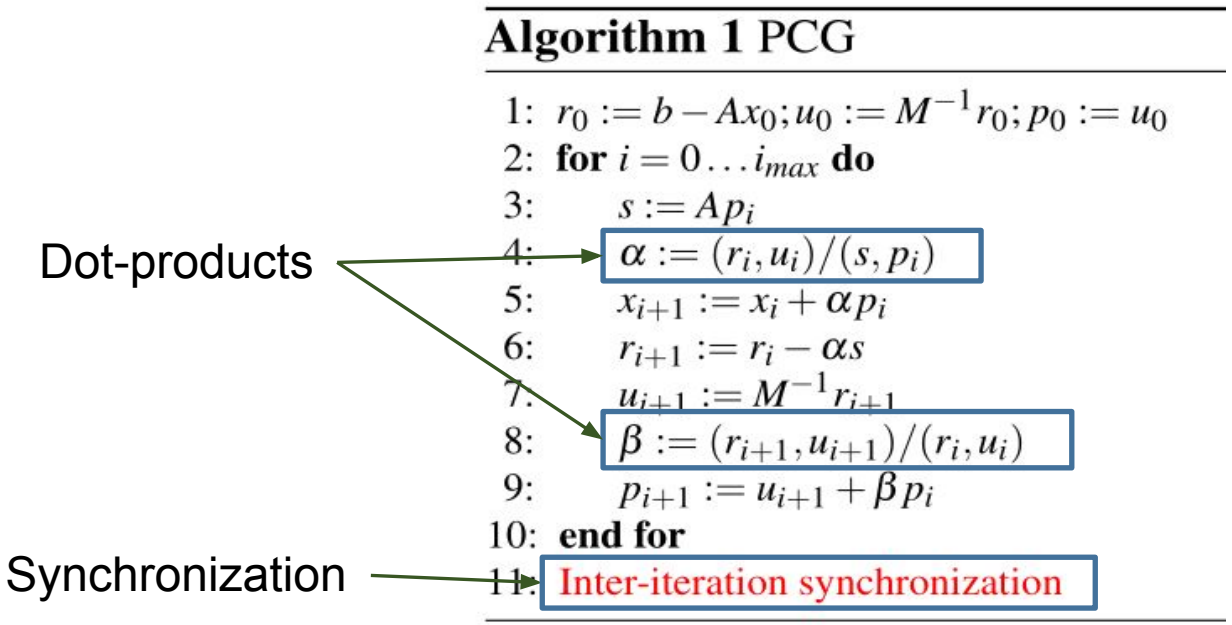
Algorithm 1 PCG

1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; p_0 := u_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: $\quad s := Ap_i$
4: $\quad \alpha := (r_i, u_i)/(s, p_i)$
5: $\quad x_{i+1} := x_i + \alpha p_i$
6: $\quad r_{i+1} := r_i - \alpha s$
7: $\quad u_{i+1} := M^{-1} r_{i+1}$
8: $\quad \beta := (r_{i+1}, u_{i+1})/(r_i, u_i)$
9: $\quad p_{i+1} := u_{i+1} + \beta p_i$
10: **end for**
11: Inter-iteration synchronization

[1] Hestenes, Magnus; Stiefel, E. (1952). "Methods of Conjugate Gradients for Solving Linear Systems" (PDF). Journal of Research of the National Bureau of Standards. 49 (6): 409–438.

Preconditioned CG (PCG)

- Three synchronization points
  - Two separate dot-products (synchronization) per iteration
  - One synchronization between adjacent iterations
- No overlapping between synchronization and computation



Pipelined CG [1]

- Packs the two PCG's reduction operations per iteration into a single double-reduction point
- Enables overlapping the reduction with expensive kernels
  - SpMV
  - Preconditioner
- Extra AXPYs

Algorithm 2 Pipelined CG

$r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
for $i = 0 \ldots i_{max}$ **do**
  $\gamma_i := (r_i, u_i)$
  $\delta_i := (w_i, u_i)$
  $m_i := M^{-1} w_i$
  $n_i := Am_i$
  if $i > 0$ **then**
    $\beta_i := \gamma_i / \gamma_{i-1}; \alpha_i := \gamma_i / (\delta_i - \beta_i \gamma_i / \alpha_{i-1})$
  else
    $\beta_i := 0; \alpha_i := \gamma_i / \delta_i$
  end if
  $z_i := n_i + \beta_i z_{i-1}$
  $q_i := m_i + \beta_i q_{i-1}$
  $s_i := w_i + \beta_i s_{i-1}$
  $p_i := u_i + \beta_i p_{i-1}$
  $x_{i+1} := x_i + \alpha_i p_i$
  $r_{i+1} := r_i - \alpha_i s_i$
  $u_{i+1} := u_i - \alpha_i q_i$
  $w_{i+1} := w_i - \alpha_i z_i$
end for
Inter-iteration synchronization

[1] P. Ghysels and W. Vanroose. 2014. Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. *Parallel Comput.* 40, 7 (July 2014), 224-238. DOI: http://dx.doi.org/10.1016/j.parco.2013.06.001

Pipelined CG

- Packs the two PCG's reduction operations per iteration into a single double-reduction point
- Enables overlapping the reduction with expensive kernels
  - SpMV
  - Preconditioner
- Extra AXPYs

Algorithm 2 Pipelined CG

1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: $\gamma_i := (r_i, u_i)$
4: $\delta_i := (w_i, u_i)$
5: $m_i := M^{-1} w_i$
6: $n_i := Am_i$
7: **if** $i > 0$ **then**
8: $\beta_i := \gamma_i / \gamma_{i-1}; \alpha_i := \gamma_i / (\delta_i - \beta_i \gamma_i / \alpha_{i-1})$
9: **else**
10: $\beta_i := 0; \alpha_i := \gamma_i / \delta_i$
11: **end if**
12: $z_i := n_i + \beta_i z_{i-1}$
13: $q_i := m_i + \beta_i q_{i-1}$
14: $s_i := w_i + \beta_i s_{i-1}$
15: $p_i := u_i + \beta_i p_{i-1}$
16: $x_{i+1} := x_i + \alpha_i p_i$
17: $r_{i+1} := r_i - \alpha_i s_i$
18: $u_{i+1} := u_i - \alpha_i q_i$
19: $w_{i+1} := w_i - \alpha_i z_i$
20: **end for**
21: Inter-iteration synchronization

Double-reduction point

Reduction, SpMv and Precond overlap

Extra operations

Iteration-Fusing Conjugate Gradient (IFCG)

- An evolution of Pipelined CG
- **Splits up some of the computation routines into smaller parallel tasks**
  - Relaxes data dependencies
  - Reduces idle time
- **Allows the overlap of computations belonging to adjacent iterations by removing the inter-iteration synchronization**
  - Reduction of synchronization costs
  - Convergence check once every several iterations
- Two algorithms with different aims
  - IFCG1 aims at hiding the cost of parallel reductions
  - IFCG2 aims at reducing idle time

IFCG1: hiding the cost of parallel reductions

- Blocks of operations are unrolled
- Inter-iteration synchronization is performed once every a certain number of iterations (FUSE parameter)

Algorithm 2 Pipelined CG

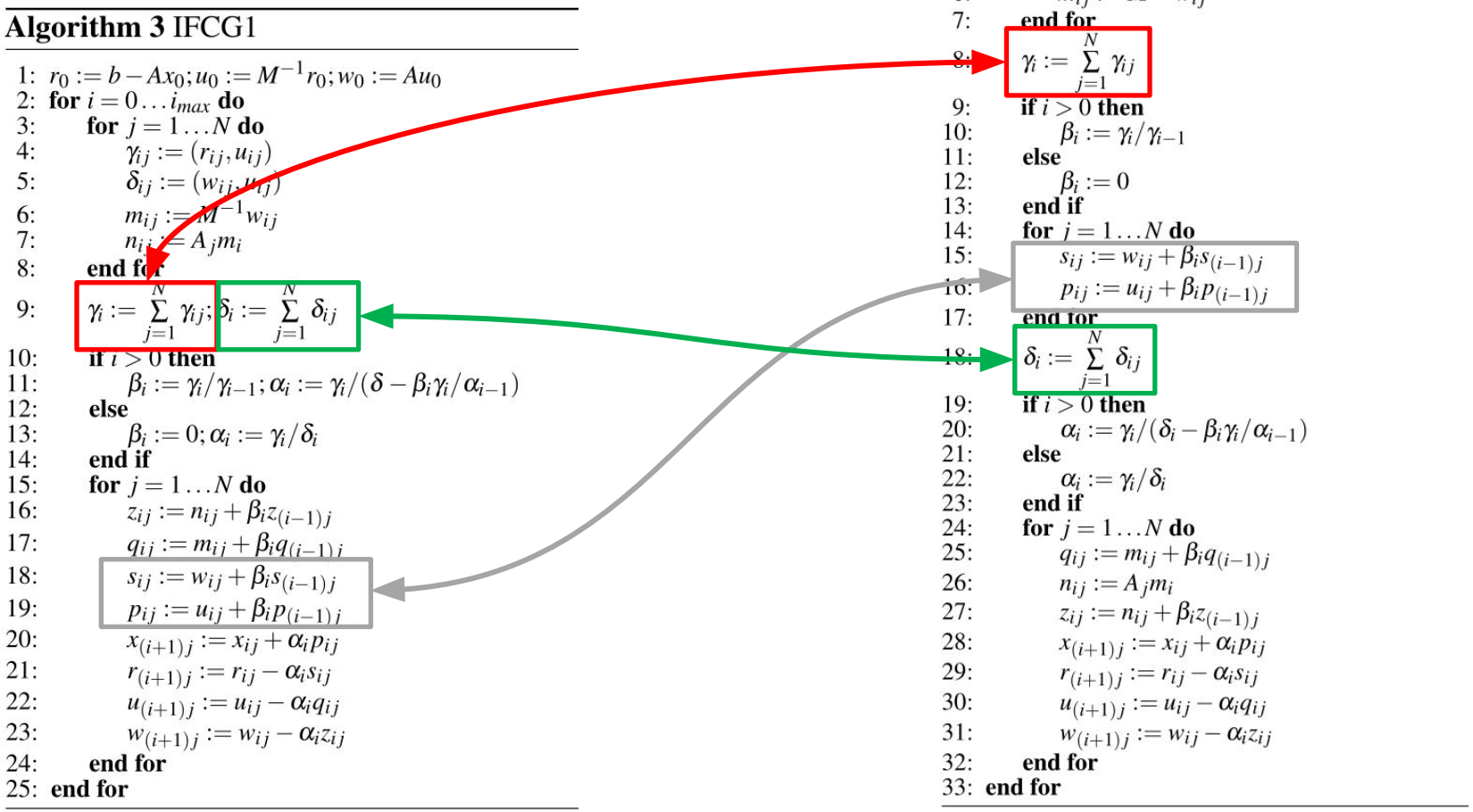
1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: $\gamma_i := (r_i, u_i)$
4: $\delta_i := (w_i, u_i)$
5: $m_i := M^{-1} w_i$
6: $n_i := Am_i$
7: **if** $i > 0$ **then**
8: $\beta_i := \gamma_i / \gamma_{i-1}; \alpha_i := \gamma_i / (\delta_i - \beta_i \gamma_i / \alpha_{i-1})$
9: **else**
10: $\beta_i := 0; \alpha_i := \gamma_i / \delta_i$
11: **end if**
12: $z_i := n_i + \beta_i z_{i-1}$
13: $q_i := m_i + \beta_i q_{i-1}$
14: $s_i := w_i + \beta_i s_{i-1}$
15: $p_i := u_i + \beta_i p_{i-1}$
16: $x_{i+1} := x_i + \alpha_i p_i$
17: $r_{i+1} := r_i - \alpha_i s_i$
18: $u_{i+1} := u_i - \alpha_i q_i$
19: $w_{i+1} := w_i - \alpha_i z_i$
20: **end for**
21: ~~Inter-iteration synchronization~~ Removed

Algorithm 3 IFCG1

1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: **for** $j = 1 \ldots N$ **do**
4: $\gamma_{ij} := (r_{ij}, u_{ij})$
5: $\delta_{ij} := (w_{ij}, u_{ij})$
6: $m_{ij} := M^{-1} w_{ij}$
7: $n_{ij} := A_j m_i$
8: **end for**
9: $\gamma_i := \sum_{j=1}^{N} \gamma_{ij}; \delta_i := \sum_{j=1}^{N} \delta_{ij}$
10: **if** $i > 0$ **then**
11: $\beta_i := \gamma_i / \gamma_{i-1}; \alpha_i := \gamma_i / (\delta - \beta_i \gamma_i / \alpha_{i-1})$
12: **else**
13: $\beta_i := 0; \alpha_i := \gamma_i / \delta_i$
14: **end if**
15: **for** $j = 1 \ldots N$ **do**
16: $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$
17: $q_{ij} := m_{ij} + \beta_i q_{(i-1)j}$
18: $s_{ij} := w_{ij} + \beta_i s_{(i-1)j}$
19: $p_{ij} := u_{ij} + \beta_i p_{(i-1)j}$
20: $x_{(i+1)j} := x_{ij} + \alpha_i p_{ij}$
21: $r_{(i+1)j} := r_{ij} - \alpha_i s_{ij}$
22: $u_{(i+1)j} := u_{ij} - \alpha_i q_{ij}$
23: $w_{(i+1)j} := w_{ij} - \alpha_i z_{ij}$
24: **end for**
25: **end for**

IFCG2: reducing idle time

- IFCG2 separates two co-efficients for an early launch of two AXPYs with fewer data dependencies



Algorithm 3 IFCG1

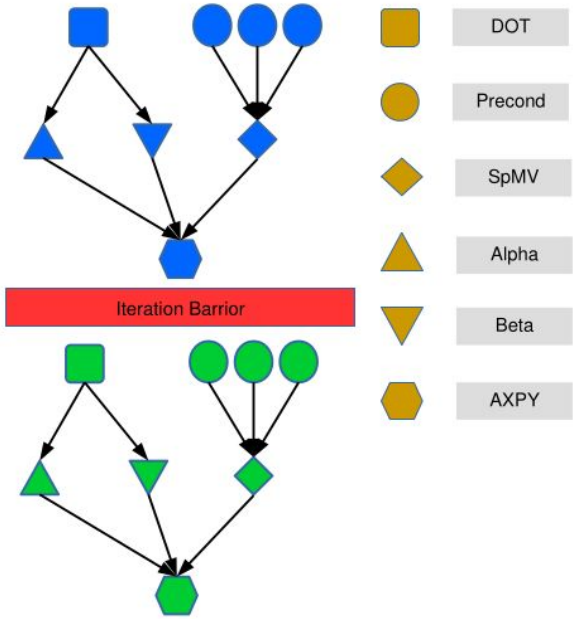
1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: **for** $j = 1 \ldots N$ **do**
4: $\gamma_{ij} := (r_{ij}, u_{ij})$
5: $\delta_{ij} := (w_{ij}, u_{ij})$
6: $m_{ij} := M^{-1} w_{ij}$
7: $n_{ij} := A_j m_i$
8: **end for**
9: $\gamma_i := \sum_{j=1}^{N} \gamma_{ij}; \delta_i := \sum_{j=1}^{N} \delta_{ij}$
10: **if** $i > 0$ **then**
11: $\beta_i := \gamma_i / \gamma_{i-1}; \alpha_i := \gamma_i / (\delta - \beta_i \gamma_i / \alpha_{i-1})$
12: **else**
13: $\beta_i := 0; \alpha_i := \gamma_i / \delta_i$
14: **end if**
15: **for** $j = 1 \ldots N$ **do**
16: $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$
17: $q_{ij} := m_{ij} + \beta_i q_{(i-1)j}$
18: $s_{ij} := w_{ij} + \beta_i s_{(i-1)j}$
19: $p_{ij} := u_{ij} + \beta_i p_{(i-1)j}$
20: $x_{(i+1)j} := x_{ij} + \alpha_i p_{ij}$
21: $r_{(i+1)j} := r_{ij} - \alpha_i s_{ij}$
22: $u_{(i+1)j} := u_{ij} - \alpha_i q_{ij}$
23: $w_{(i+1)j} := w_{ij} - \alpha_i z_{ij}$
24: **end for**
25: **end for**

Algorithm 4 IFCG2

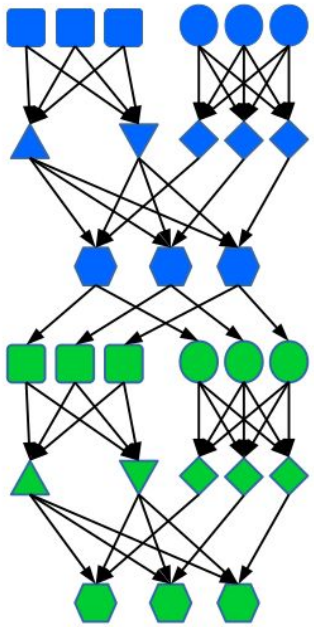
1: $r_0 := b - Ax_0; u_0 := M^{-1} r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: **for** $j = 1 \ldots N$ **do**
4: $\gamma_{ij} := (r_{ij}, u_{ij})$
5: $\delta_{ij} := (w_{ij}, u_{ij})$
6: $m_{ij} := M^{-1} w_{ij}$
7: **end for**
8: $\gamma_i := \sum_{j=1}^{N} \gamma_{ij}$
9: **if** $i > 0$ **then**
10: $\beta_i := \gamma_i / \gamma_{i-1}$
11: **else**
12: $\beta_i := 0$
13: **end if**
14: **for** $j = 1 \ldots N$ **do**
15: $s_{ij} := w_{ij} + \beta_i s_{(i-1)j}$
16: $p_{ij} := u_{ij} + \beta_i p_{(i-1)j}$
17: **end for**
18: $\delta_i := \sum_{j=1}^{N} \delta_{ij}$
19: **if** $i > 0$ **then**
20: $\alpha_i := \gamma_i / (\delta_i - \beta_i \gamma_i / \alpha_{i-1})$
21: **else**
22: $\alpha_i := \gamma_i / \delta_i$
23: **end if**
24: **for** $j = 1 \ldots N$ **do**
25: $q_{ij} := m_{ij} + \beta_i q_{(i-1)j}$
26: $n_{ij} := A_j m_i$
27: $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$
28: $x_{(i+1)j} := x_{ij} + \alpha_i p_{ij}$
29: $r_{(i+1)j} := r_{ij} - \alpha_i s_{ij}$
30: $u_{(i+1)j} := u_{ij} - \alpha_i q_{ij}$
31: $w_{(i+1)j} := w_{ij} - \alpha_i z_{ij}$
32: **end for**
33: **end for**

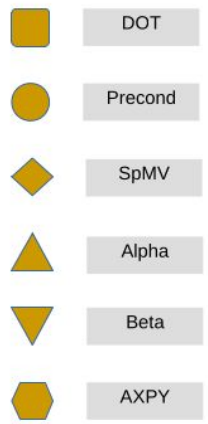
Increased Parallelism

- Task-Dependency Graphs (TDG)
- Example with two iterations and three tasks per kernel

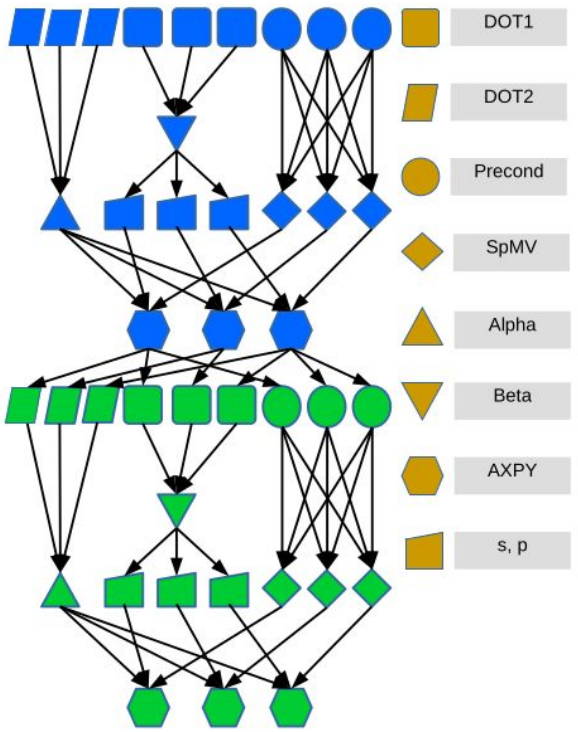


Pipelined CG





IFCG1



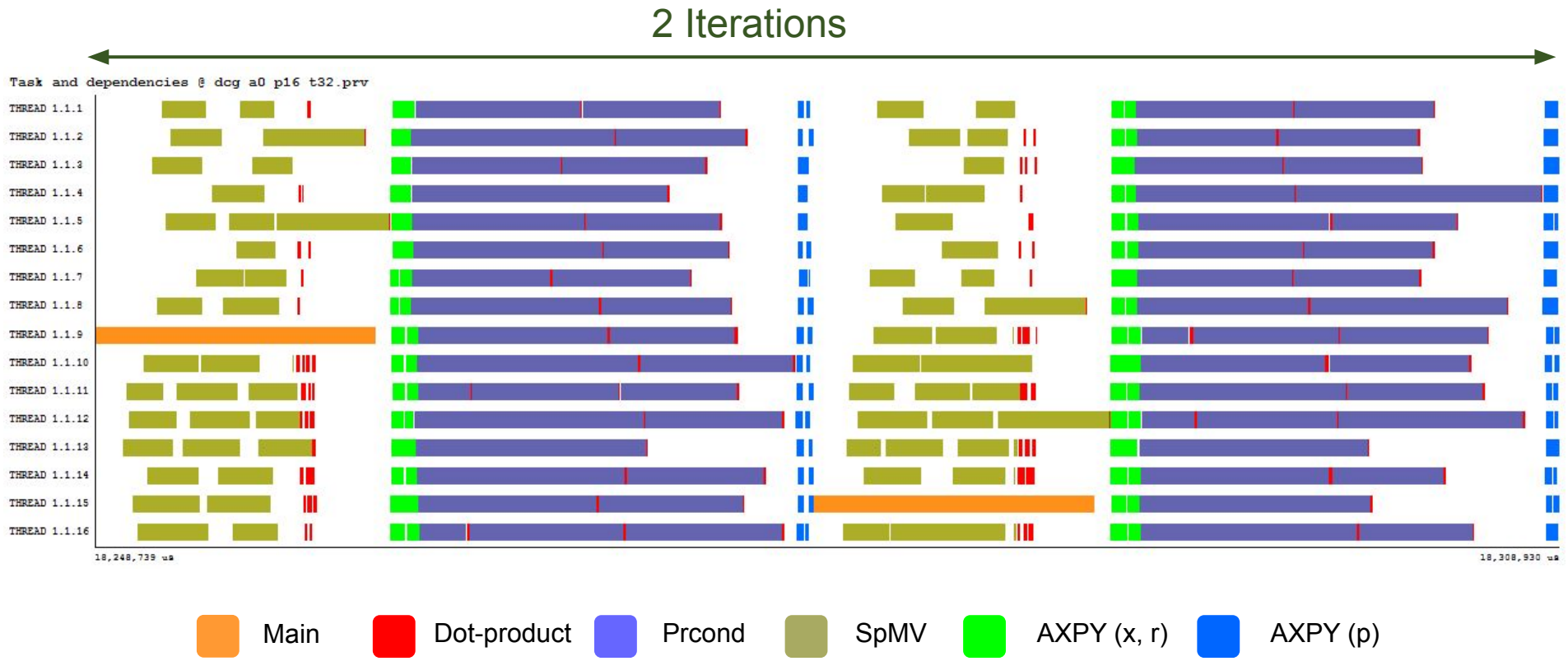
IFCG2

Experimental Setup

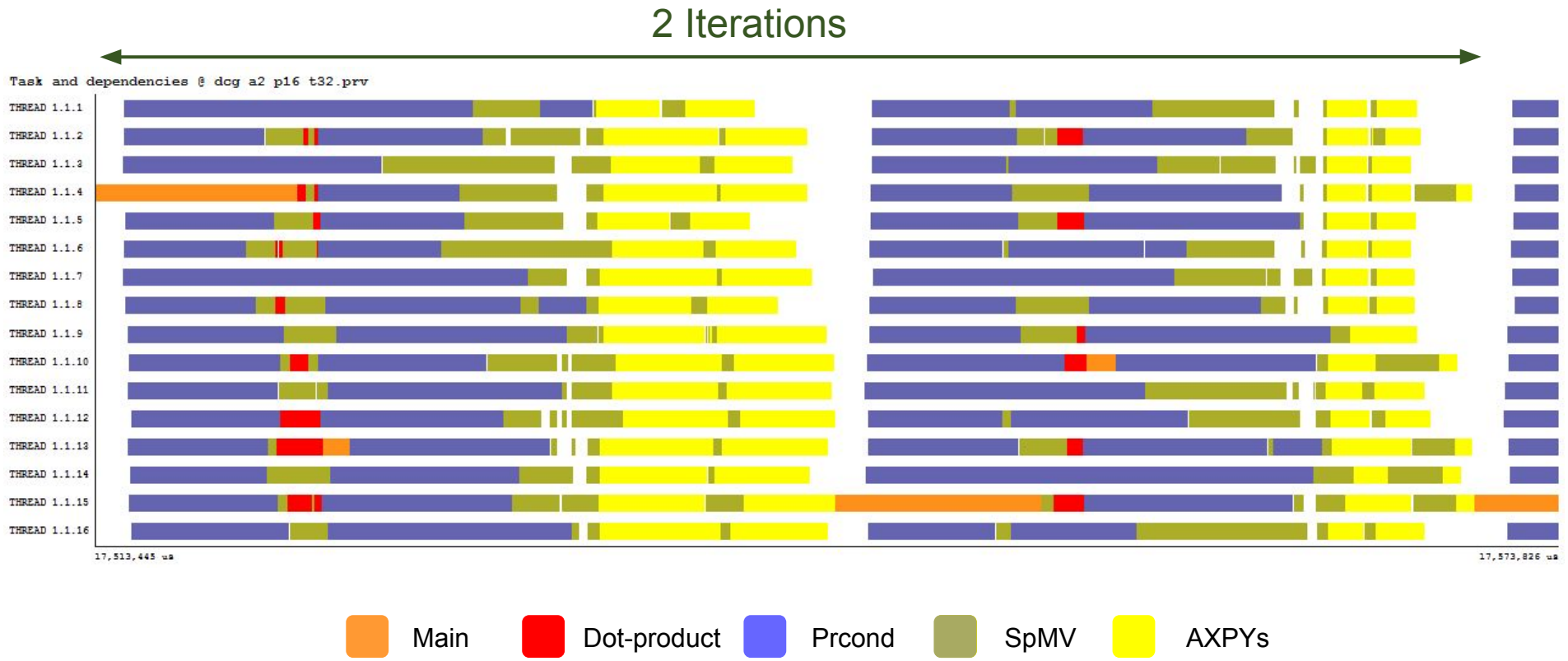
- OpenMP 4.0 programming model
- Node with
  - Two 8-core Intel Xeon E5-2670 processors
  - 20MB LLC
- Preconditioner: block-jacobi with incomplete cholesky factorization
- Seven sparse matrices from the SuiteSparse Matrix Collection

Name	Dimension	Nonzeros	Nonzeros%
G3_circuit	1585478	7660826	0.0003%
thermal2	1228045	8580313	0.0006%
ecology2	999999	4995991	0.0005%
af_shell8	504855	17579155	0.0068%
G2_circuit	150102	726674	0.003%
cfd2	123440	3085406	0.02%
consph	83334	6010480	0.087%

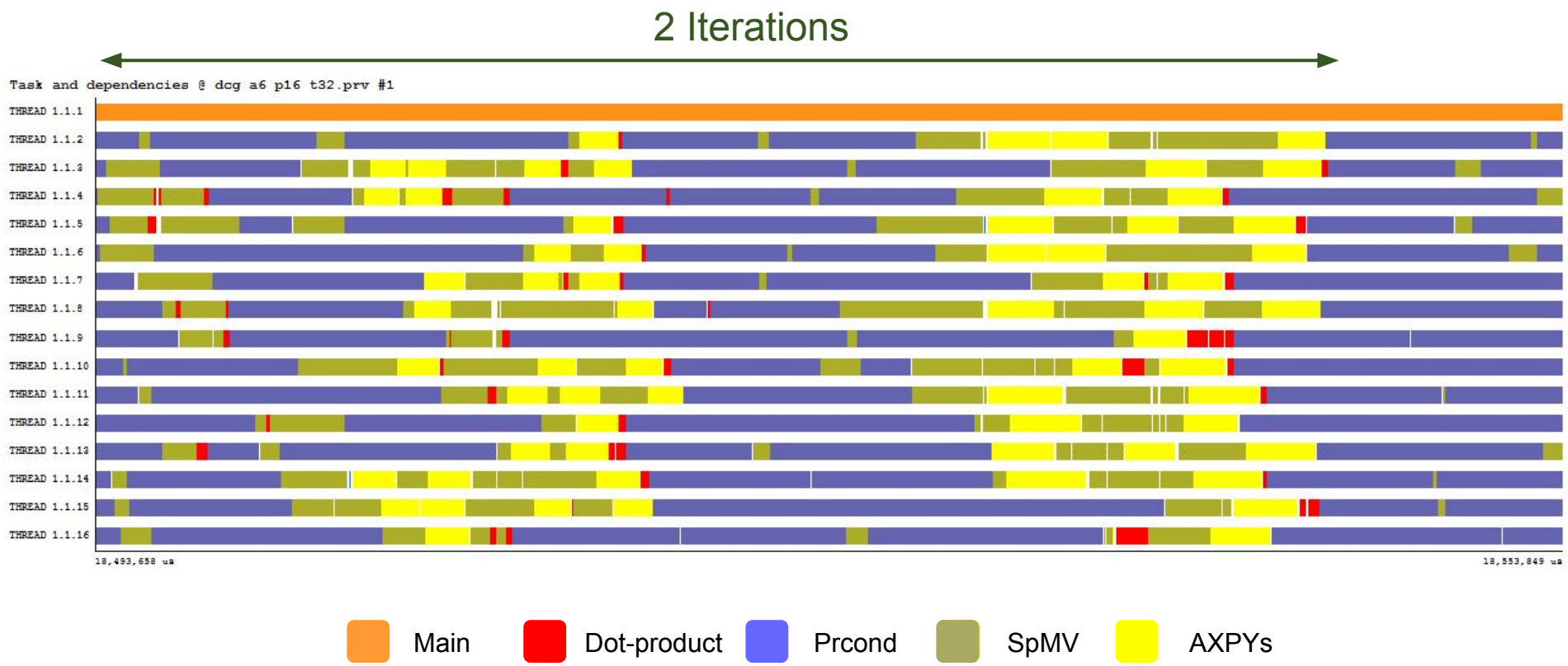
Parallel Execution Preconditioned CG (thermal2 matrix)



Parallel Execution Pipelined CG (thermal2 matrix)

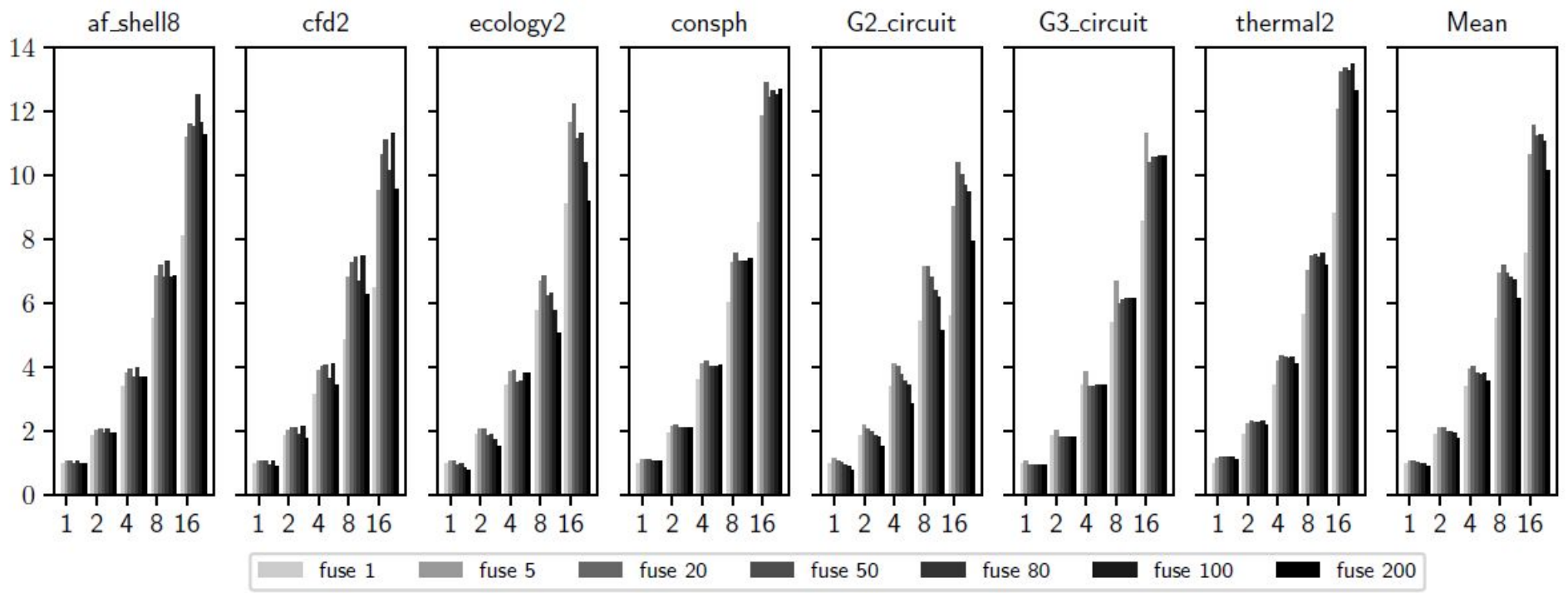


Parallel Execution IFCG1 (thermal2 matrix)



FUSE Parameter

- Number of iterations without convergence check
- In need of an optimal FUSE parameter
  - Too small a FUSE, we hinder overlaps across iterations
  - Too large a FUSE, we overshoot by many iterations



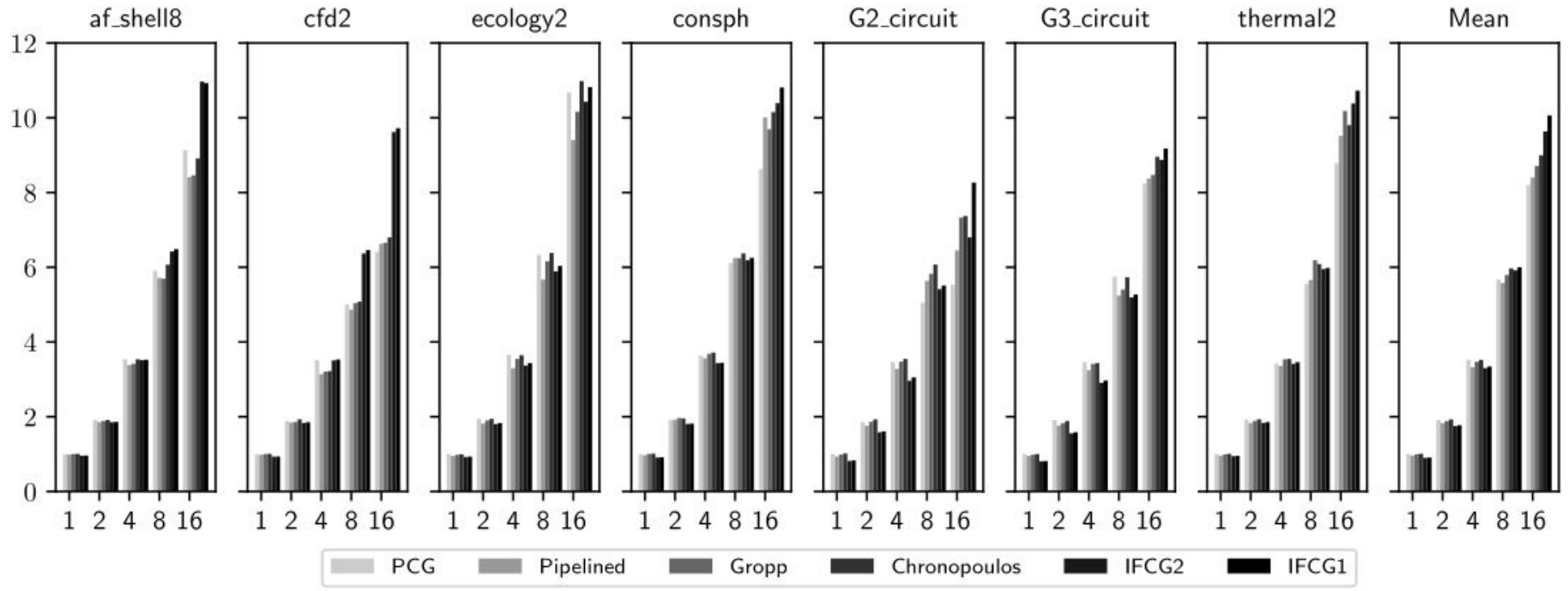
Evaluation

- Against Preconditioned CG and 3 state-of-the-art variants
  - Pipelined CG
  - Chronopoulos CG [1]
  - Gropp CG [2]

[1] A. T. Chronopoulos and C. W. Gear. 1989. *s*-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.* 25, 2 (February 1989), 153-168. DOI=http://dx.doi.org/10.1016/0377-0427(89)90045-9
[2] W. Gropp. Update on libraries for blue waters. http://jointlab-pc.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf, 2010

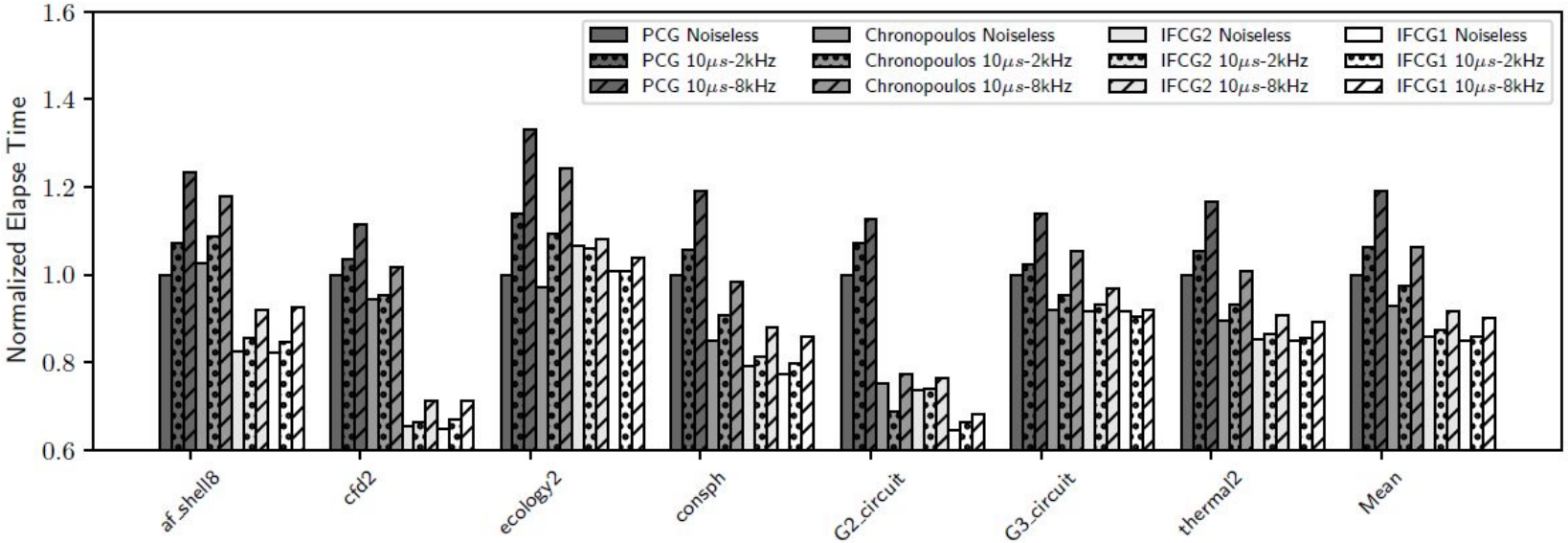
Evaluation

- Against Preconditioned CG and 3 state-of-the-art variants
- Average improvements of 11.8% and 7.1%
- Up to 42.9% and 41.5% (cfd2)



System Noise Tolerance

- IFCGs perform fewer reductions/synchronizations, which amplify jitter effects due to system noise
- Two realistic noise regimes
  - 10 µs – 2kHz (2% overhead), 10 µs – 8kHz (8% overhead)
- Inject uniformly distributed random noise to a certain amount of tasks and double their execution time
  - IFCG1 runs 18% faster than the state-of-the-art (Chronopoulos)



Barcelona Supercomputing Center
Centro Nacional de Superc

Conclusions and Future Work

- IFCG1 and IFCG2 increase performance of state-of-the-art methods by
  - Allows the **overlap of computations belonging to adjacent iterations** by removing the inter-iteration synchronization
  - Such overlap is increased by **splitting linear algebra kernels into subkernels**
- The IFCG algorithms are much more tolerant to system noise effects than the state-of-the-art due to their reduced number of synchronizations
- In the future, we will create
  - a distributed-memory implementation of IFCG by combining MPI and OmpSs/OpenMP (Universitat Jaume I is working on it)
  - smarter FUSE parameter tuning

Outline

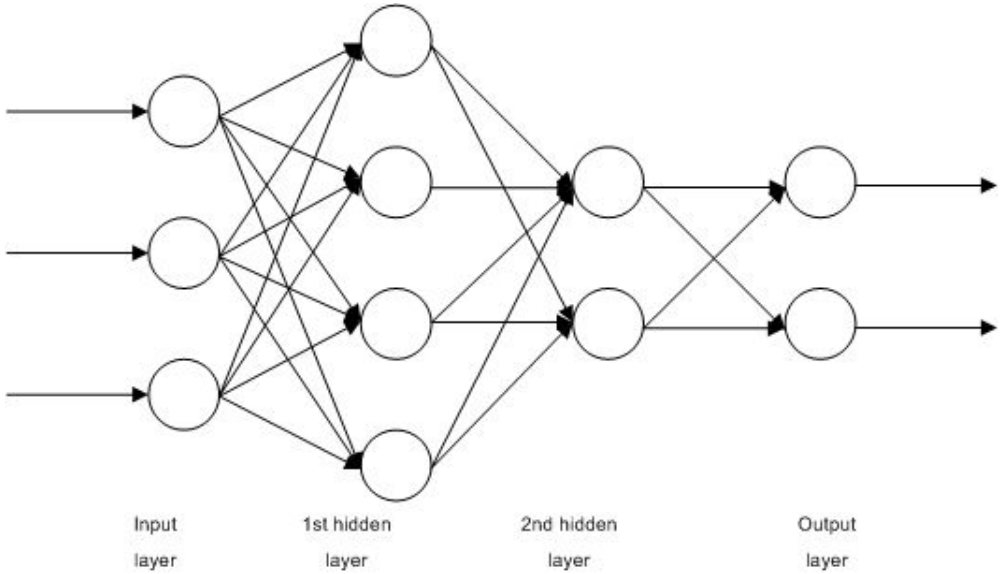
- Background
- Communication reduction in Conjugate Gradient Method
- Communication reduction in DNN training on multi-GPU environments
- Communication reduction in model parallelism for DNN
- Conclusions and future works

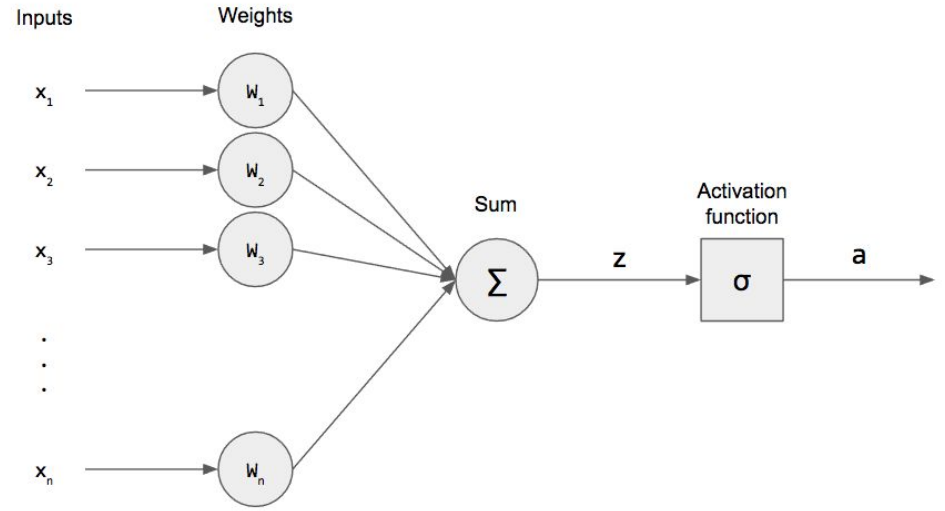
Communication Reduction in DNN Training on Multi-GPU Environments

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Deep Neural Network (DNN)

- Multi-layered MLP, CNN, RNN
- Provides pattern detection capabilities
- Mass adoption in computer vision [1], speech recognition [2], NLP [3] etc.

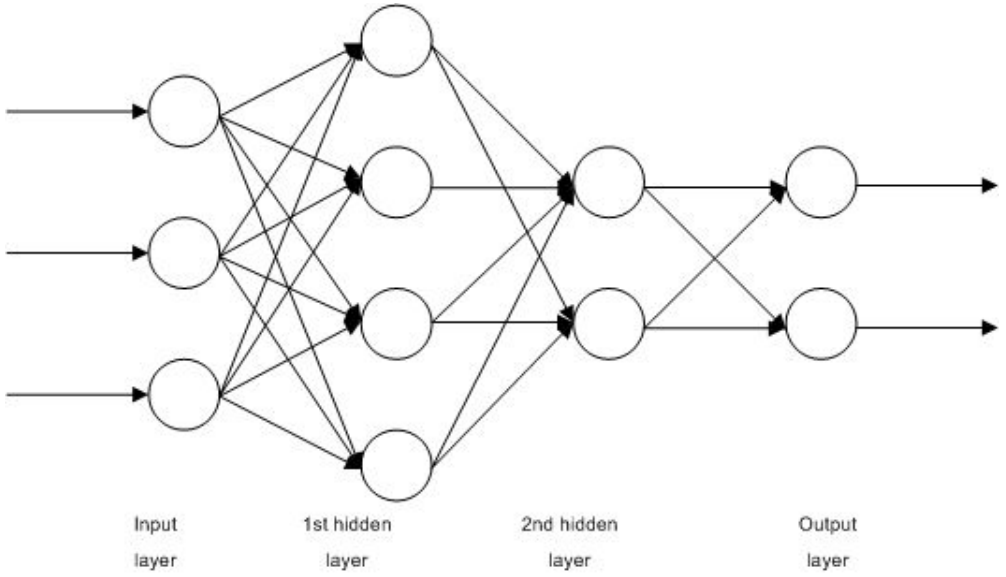




[1] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E. (2017-05-24). "ImageNet classification with deep convolutional neural networks" (PDF). Communications of the ACM. 60 (6): 84–90. doi:10.1145/3065386. ISSN 0001-0782.
[2] G. Hinton *et al*., "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," in *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82-97, Nov. 2012.
[3] Yonghui Wu et al, “Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. “ https://arxiv.org/abs/1609.08144 , 2016

Training DNN via Backpropagation [1]

- Forward stage
- Backward stage
- Update weights using Stochastic Gradient Descent (SGD) [2] and its variants
- Dispose of intrinsic data-level parallelism that is suitable for using accelerators



[1] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". *Nature*. **323** (6088): 533–536.
[2] Robbins, Herbert; Monro, Sutton. A Stochastic Approximation Method. Ann. Math. Statist. 22 (1951), no. 3, 400--407. doi:10.1214/aoms/1177729586. https://projecteuclid.org/euclid.aoms/1177729586

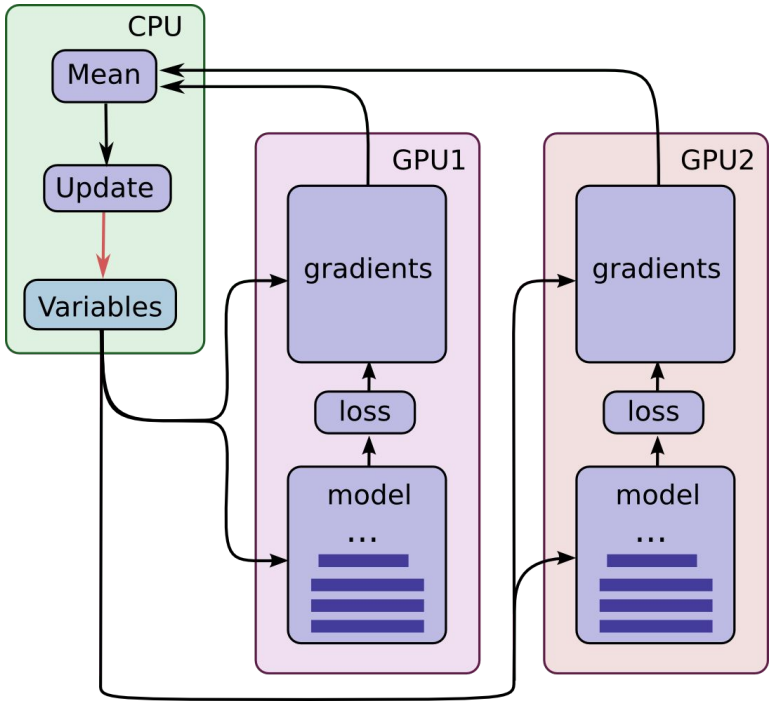
Reduced Precision Training

- Alternative data representations
  - Fixed-precision arithmetic [1]
  - Flexpoint [2]
  - BFloat16 [3]
- Mobile devices
  - Inference [4]
  - Distributed training [5]

[1] Hopkins, Michael et al. (2019). Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ODEs.
[2] Köster, Urs et al.. (2017). Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks.
[3] Tagliavini, Giuseppe et al. "A transprecision floating-point platform for ultra-low power computing". *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. arXiv:1711.10374. doi:10.23919/DATE.2018.8342167.
[4] Han, Song et al. (2016). EIE: Efficient Inference Engine on Compressed Deep Neural Network. ACM SIGARCH Computer Architecture News. 44. 10.1145/3007787.3001163.
[5] Lin, Yujun et al. (2017). Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training.

Multi-GPU Training

- Host CPU serves as a parameter server that stores and updates the parameters
- GPUs receive the parameters, compute gradients and send them back to CPU on a batch basis
- Frequent & large volume host-device data exchange inflicts substantial performance penalties



Reduce Communication During Training

- Exploit DNN's tolerance towards data representation formats less than 32-bit FP
- Compress and decompress weights respectively on CPU and GPUs
  - Approximate Data Transfer (ADT)
- The compression rate is dynamically adjusted during the runtime
  - Adaptive Weight Precision (AWP)

Adaptive Weight Precision (AWP)

- Increment of L2 norm of the weights correlates with the improvement of network accuracy

Adaptive Weight Precision (AWP)

- Increment of L2 norm of the weights correlates with the improvement of network accuracy
- Monitoring the change rate of the L2 norm of weights per layer

Algorithm 5 Adaptive Weight Precision (AWP) Algorithm

BitsPerLayer := $[B_0, B_1, \ldots, B_{NumLayers}]$ ▷ List storing the number of bits corresponding to the data representation of each layer
IntervalCounter := [0, 0, ..., 0] ▷ List storing the number of times the relative change rate fails to meet the threshold per layer
for batch := 0 ... NumBatches **do**
    Apply backpropagation to batch
    for layer := 0 ... NumLayers **do**
        $\delta := \frac{(|W_{batch,layer}| - |W_{batch-1,layer}|)}{|W_{batch-1,layer}|}$
        if $\delta < 1$ **then**
            $IntervalCounter_{layer}$ += 1
        end if
        if $IntervalCounter_{layer}$ == INTERVAL **then**
            $BitsPerLayer_{layer}$ += N
            $IntervalCounter_{layer}$:= 0
        end if
    end for
end for

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Adaptive Weight Precision (AWP)

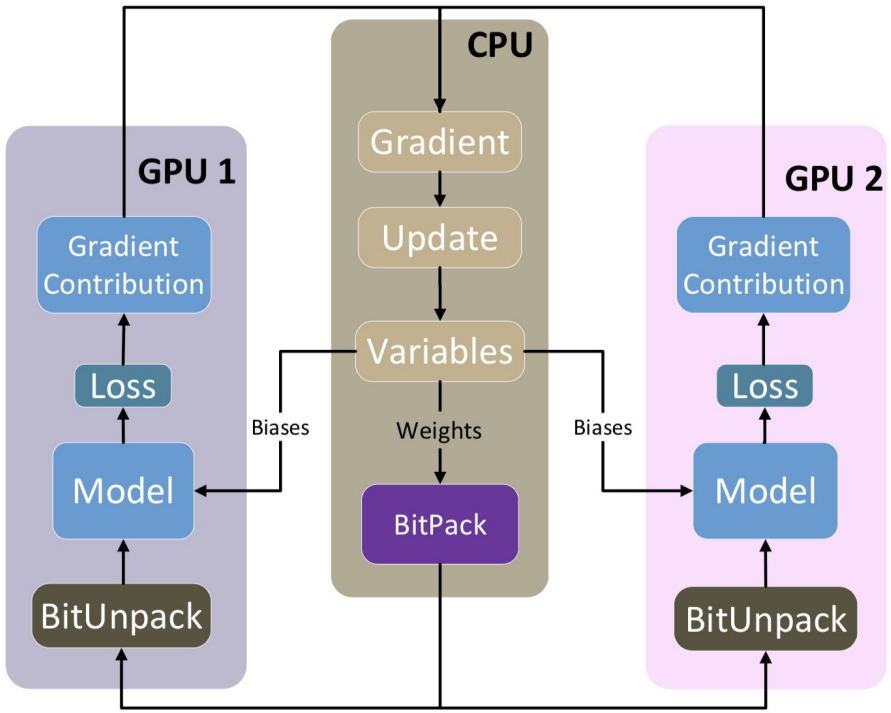
- Increment of L2 norm of the weights correlates with the improvement of network accuracy
- Monitoring the change rate of the L2 norm of weights per layer
- Increase by ***N*** bits if the change rate of L2 norm is less than ***T*** in a consecutive ***INTERVAL*** batches

Algorithm 5 Adaptive Weight Precision (AWP) Algorithm

1: BitsPerLayer := $[B_0, B_1, \ldots, B_{NumLayers}]$ ▷ List storing the number of bits corresponding to the data representation of each layer
2: IntervalCounter := [0, 0, ..., 0] ▷ List storing the number of times the relative change rate fails to meet the threshold per layer
3: **for** batch := 0 ... NumBatches **do**
4: Apply backpropagation to batch
5: **for** layer := 0 ... NumLayers **do**
6: $\delta := \frac{(|W_{batch,layer}| - |W_{batch-1,layer}|)}{|W_{batch-1,layer}|}$
7: **if** $\delta <$ T **then**
8: $\text{IntervalCounter}_{layer}$ += 1
9: **end if**
10: **if** $\text{IntervalCounter}_{layer}$ == INTERVAL **then**
11: $\text{BitsPerLayer}_{layer}$ += N
12: $\text{IntervalCounter}_{layer}$:= 0
13: **end if**
14: **end for**
15: **end for**

Approximate Data Transfer (ADT)

- **Bitpack**: compresses the weights discarding the least significant bits on CPU
- **Bitunpack**: converts the weights back to the IEEE-754 32-bit FP format on GPUs



Parallelizing Bitpack Using Thread-level Parallelism

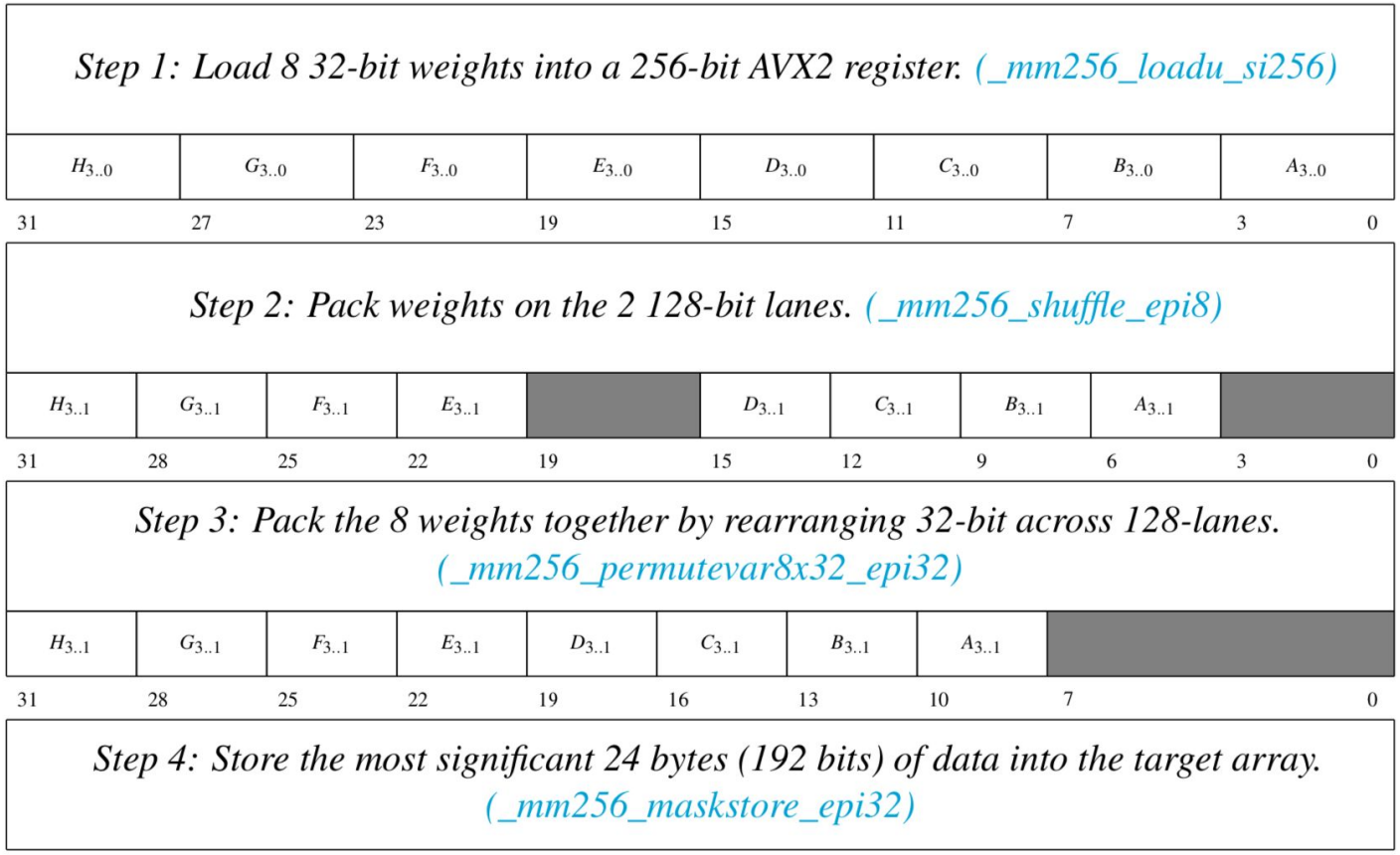
Algorithm 6 High Level Pseudo-code Version of Bitpack

```
1: W                                   ▷ Array of 32-bit Floating Point values containing weights
2: Pw                                  ▷ Array containing the reduced precision weights
3: RoundTo                             ▷ Number of bytes to keep per weight
4: POffset := 0                        ▷ Indicates the current size (in bytes) of Pw
5: for weight in W do
6:     Pw[POffset : POffset+RoundTo] := weight[0 : RoundTo] ▷ Copy most significant RoundTo
   bytes to Pw
7:     POffset := POffset + RoundTo
8: end for
```

Algorithm 7 Bitpack with OpenMP

```
1: W                                   ▷ Array of 32-bit Floating Point values containing weights
2: Pw                                  ▷ Array containing the reduced precision weights
3: RoundTo                             ▷ Number of bytes to keep per weight
4: NumThreads                          ▷ Number of OpenMP threads
5: #pragma omp parallel for
6: for weight in W do
7:     POffset := Corresponding position in Pw
8:     Pw[POffset : POffset+RoundTo] := weight[0 : RoundTo]   ▷ Copy the most significant
   RoundTo bytes to Pw
9: end for
```

Parallelizing Bitpack Using Vectorization



Experimental Setup

- ImageNet ILSVRC-2012 200-class
- 2 HPC clusters
  - 16-core Intel Xeon Haswell + Nvidia K80 x 2
  - 40-core IBM POWER9 + Nvidia V100 x 4
- 4 data representations: 8-, 16-, 24-, 32-bit
- Sample elapsed time and validation error every 4000 batches

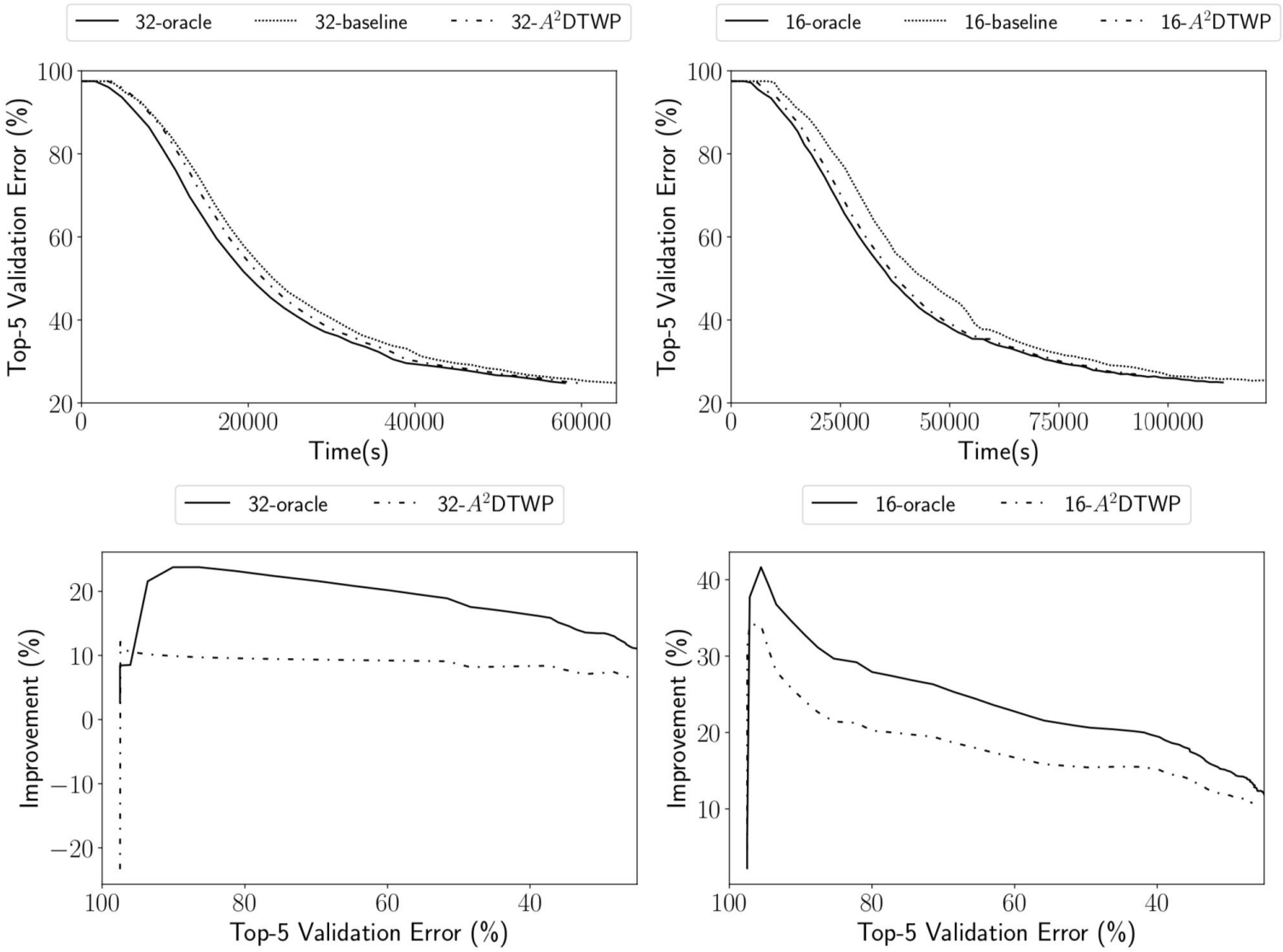
Alexnet	VGG	Resnet-34
input(224x224 RGB image)		
conv11-64	conv3-64	conv7-64
maxpool		
conv5-192	conv3-128	conv3-64 conv3-64 x3
maxpool		
conv3-384	conv3-256 conv3-256	conv3-128 conv3-128 x4
maxpool		
conv3-384	conv3-512 conv3-512	conv3-256 conv3-256 x6
maxpool		
conv3-256	conv3-512 conv3-512	conv3-512 conv3-512 x3
maxpool		avgpool
FC-4096		
FC-4096 FC-4096	FC-4096	
FC-200		
softmax		

Experimental Setup

- Compare 3 variants of one network
  - ***Baseline:*** always using 32-bit FP
  - ***Oracle:*** using the best data representation
  - ***A^2DTWP:*** combining ADT and AWP to dynamically adjust the data representation of each layer

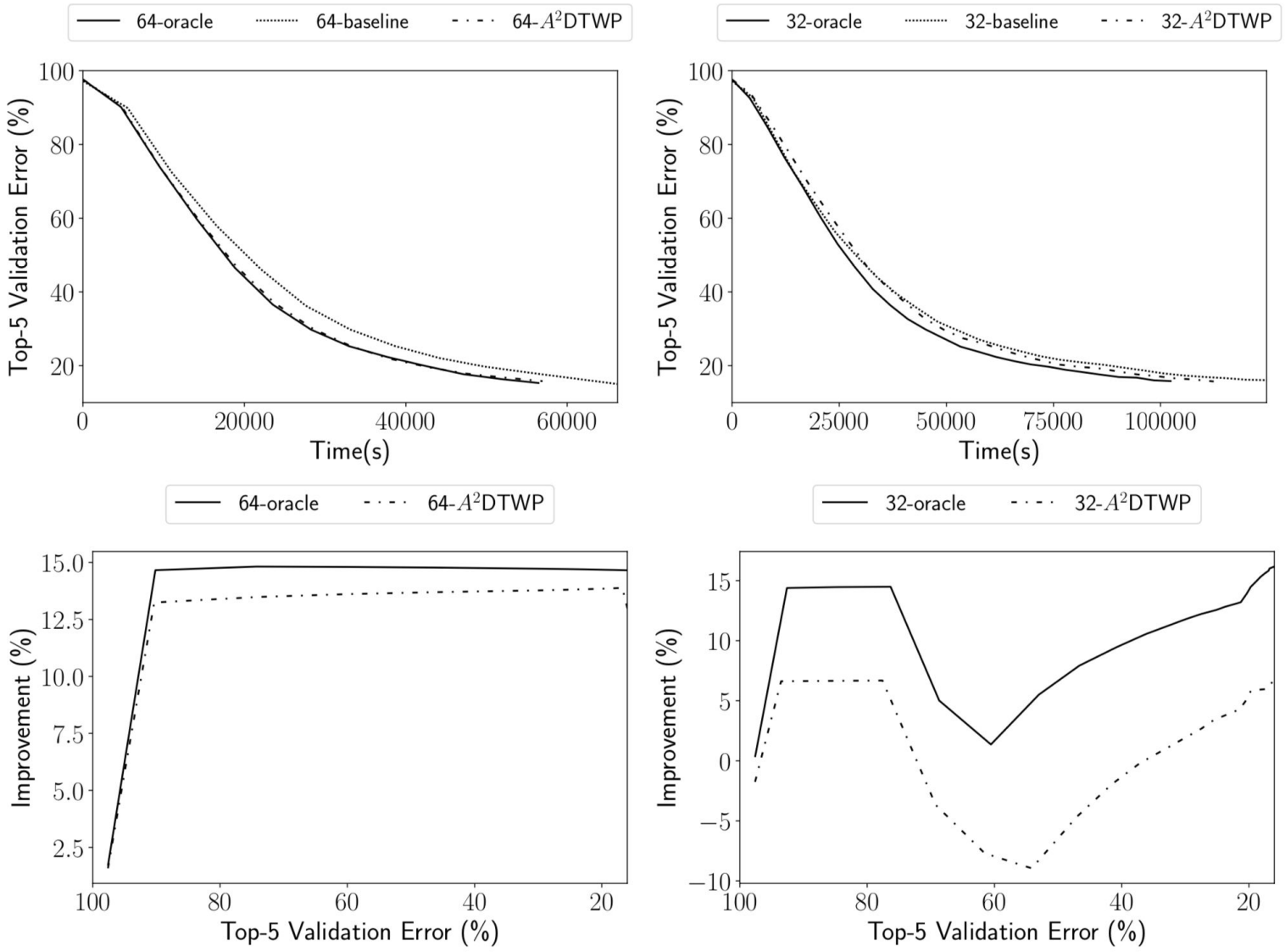
Evaluation on Alexnet

- Intel Xeon Haswell
- 6.51% and 10.75% faster than *baseline* when reaching 25% top-5 validation error



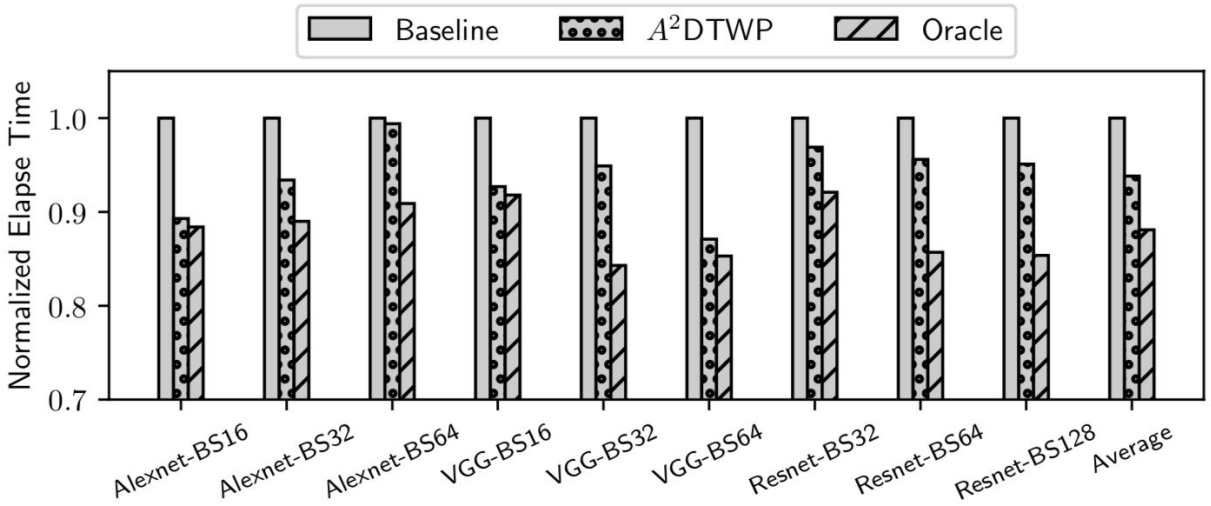
Evaluation on VGG

- Intel Xeon Haswell
- 12.88% and 5.02% faster than *baseline* when reaching 15% top-5 validation error



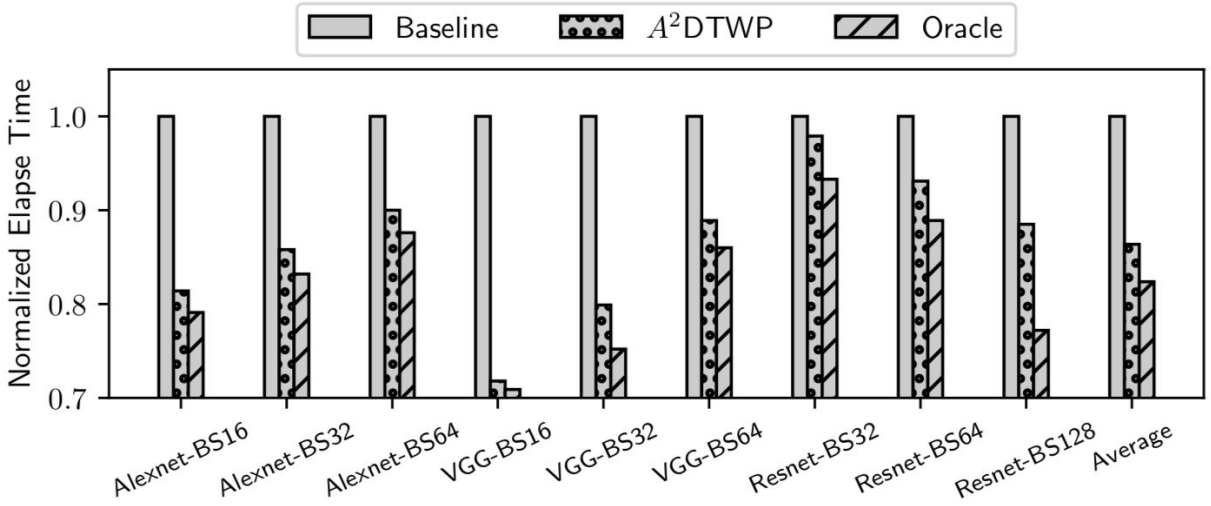
Overall Improvements

Intel Xeon Haswell



Avg: *6.18%*

IBM POWER9



Avg: *11.91%*

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusion

- Reducing host-device data transfer
  - ADT compresses and decompresses weights
    - Truncating the length of mantissa
    - Exploit thread- and SIMD-level parallelism
  - AWP guides the compression rate of the ADT procedure in runtime according to L2 norm change rate
- Retaining the training accuracy
- Generalize the approach to other hardware accelerators

Outline

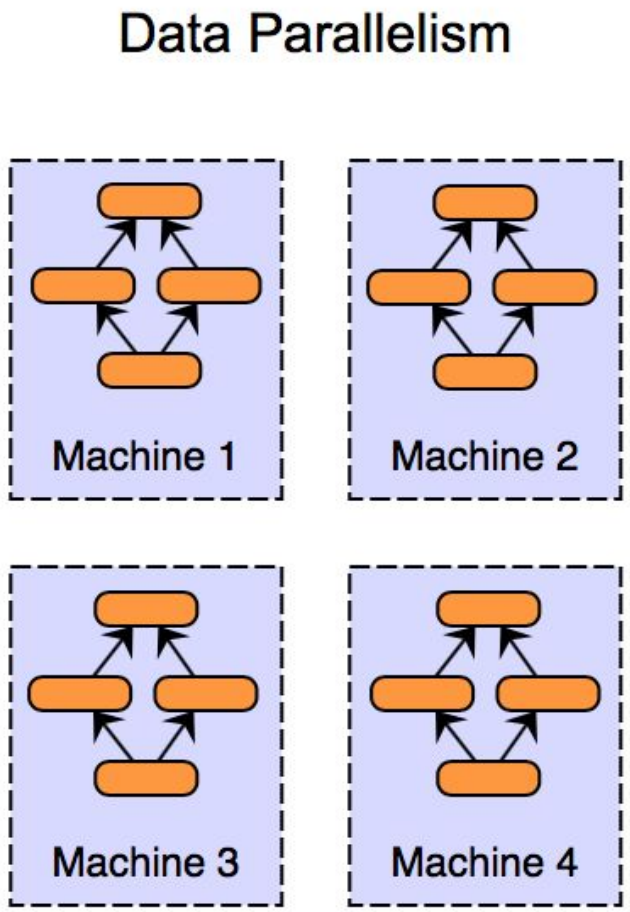
- Background
- Communication reduction in Conjugate Gradient Method
- Communication reduction in DNN training on multi-GPU environments
- Communication reduction in model parallelism for DNN
- Conclusions and future works

Communication Reduction in DNN Model Parallelism

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Parallelism Paradigms in DNN

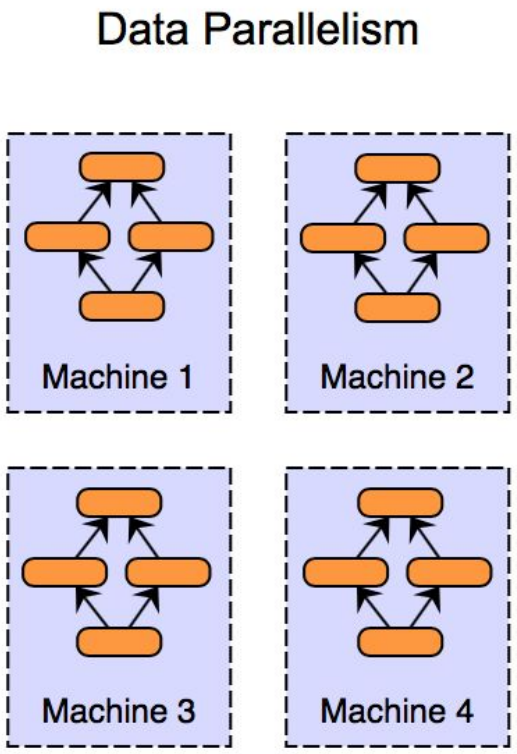
- Data parallelism [1]
  - Each computation unit retains a full replica of the network
  - Input data are split and distributed to each computation unit

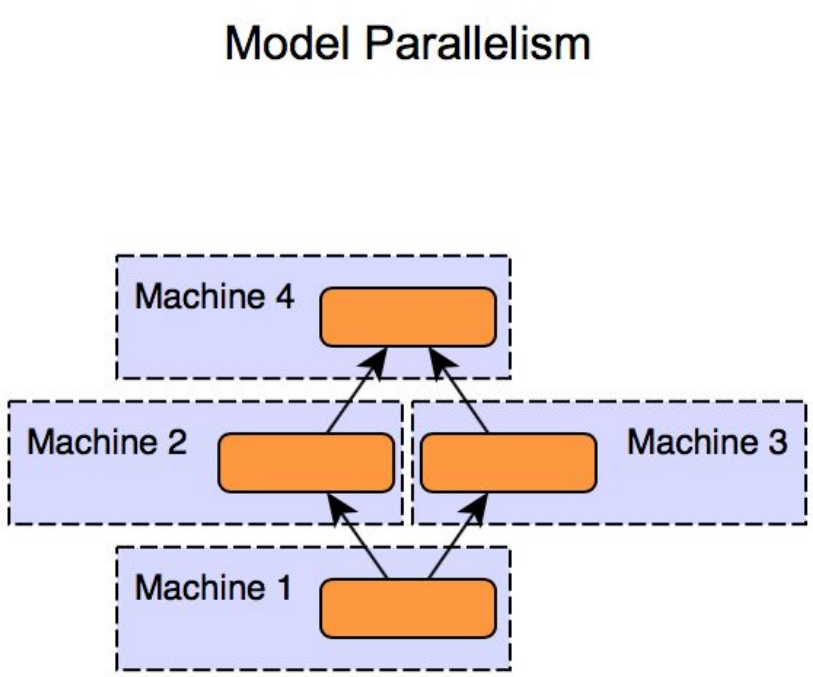


[1] Mu Li, et al. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (OSDI'14). USENIX Association, Berkeley, CA, USA, 583-598.

Parallelism Paradigms in DNN

- Data parallelism [1]
  - Each computation unit retains a full replica of the network
  - Input data are split and distributed to each computation unit
- Model parallelism [2]
  - The network itself is scattered into each computation unit
  - Data propagate through each computation unit as it goes through the entire network

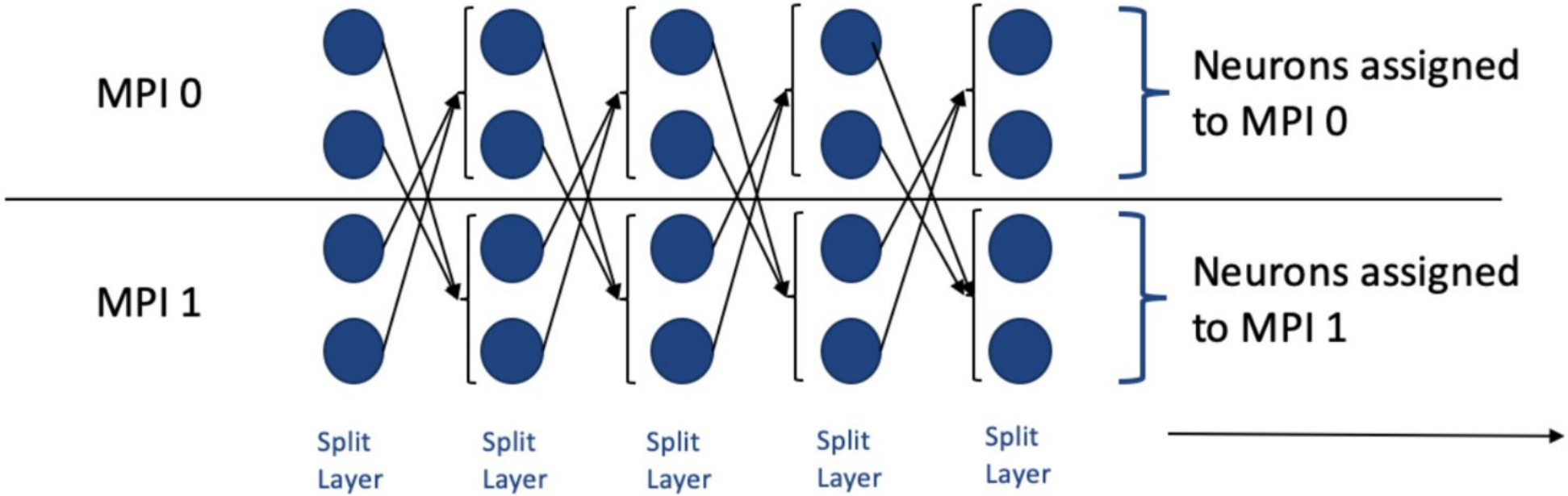




[1] Mu Li, et al. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (OSDI'14). USENIX Association, Berkeley, CA, USA, 583-598.
[2] Ben-Nun, Tal & Hoefler, Torsten. (2018). Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis.

State-of-the-Art Approach To Model Parallelism

- Feed-forward neural network
- Distributed-memory system using MPI



State-of-the-Art Approach To Model Parallelism

Algorithm 10 Sequential DNN

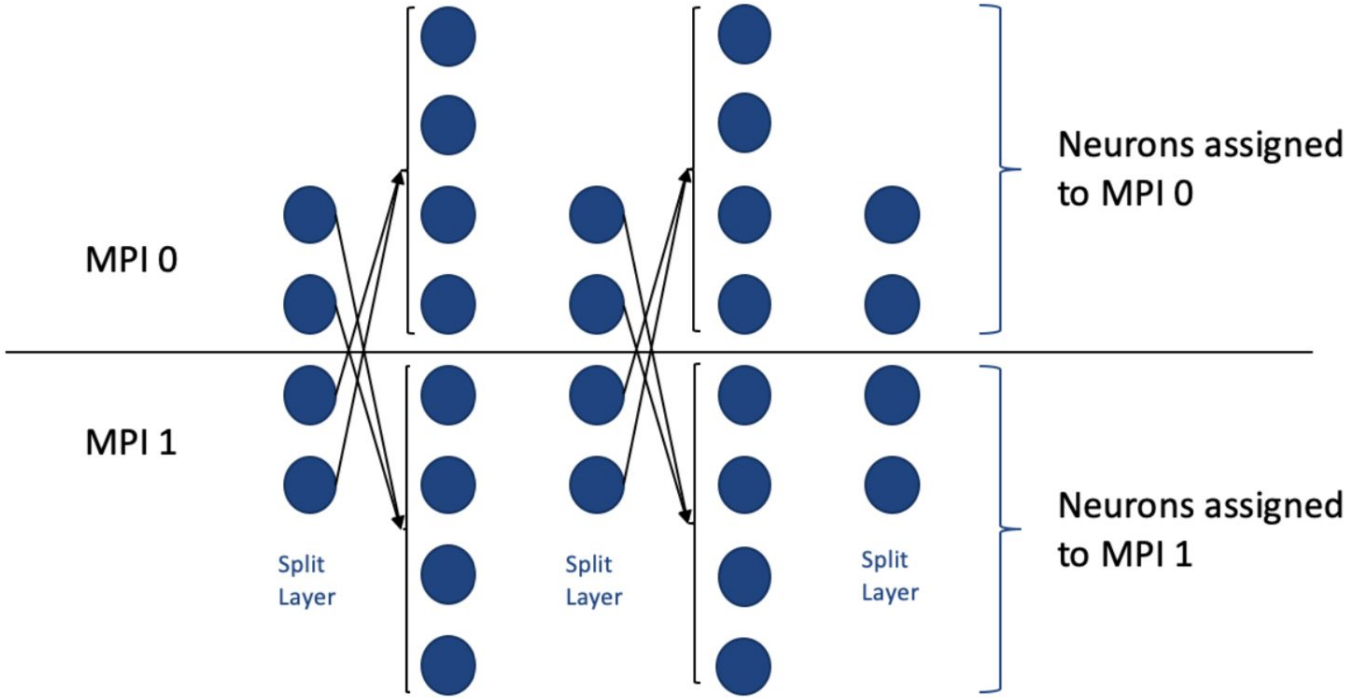
1: **for** $l = 1 \ldots L$ **do**
2: $\quad \boldsymbol{Y}_l[bs, N_l] = \boldsymbol{O}_{l-1}[bs, N_{l-1}] * (\boldsymbol{W}_l)^T[N_{l-1}, N_l]$
3: $\quad \boldsymbol{O}_l[bs, N_l] = \phi(\boldsymbol{Y}[bs, N_l])$
4: **end for**
5: **for** $l = L \ldots 1$ **do**
6: $\quad \nabla \boldsymbol{W}_l[bs, N_l] = \nabla \boldsymbol{W}_{l+1}[bs, N_{l+1}] * \boldsymbol{W}_{l+1}[N_{l+1}, N_l]$
7: $\quad \nabla \boldsymbol{W}_l[bs, N_l] = \nabla \boldsymbol{W}_l[bs, N_l] * (\partial \boldsymbol{O}_l[bs, N_l] / \partial \boldsymbol{Y}_l[bs, N_l])$
8: **end for**
9: Update parameters

Algorithm 11 State-of-the-art approach to model parallelism of DNN

1: **for all** $p \in MPI\_Processes$ **do**
2: $\quad$ **for** $l = 1 \ldots L$ **do**
3: $\quad\quad$ MPI_Allgather on $\boldsymbol{O}_{l-1}[bs, N_{l-1} * size]$ from all local $\boldsymbol{O}_{l-1}[bs, N_{l-1}]$
4: $\quad\quad \boldsymbol{Y}_l[bs, N_l] = \boldsymbol{O}_{l-1}[bs, N_{l-1} * size] * (\boldsymbol{W}_l)^T[N_{l-1} * size, N_l]$
5: $\quad\quad \boldsymbol{O}_l[bs, N_l] = \phi(\boldsymbol{Y}[bs, N_l])$
6: $\quad$ **end for**
7: **end for**
8: **for all** $p \in MPI\_Processes$ **do**
9: $\quad$ **for** $l = L \ldots 1$ **do**
10: $\quad\quad \nabla \boldsymbol{W}_l[bs, N_l * size] = \nabla \boldsymbol{W}_{l+1}[bs, N_{l+1}] * \boldsymbol{W}_{l+1}[N_{l+1}, N_l * size]$
11: $\quad\quad$ MPI_Allreduce_sum on $\nabla \boldsymbol{W}_l[bs, N_l * size]$
12: $\quad\quad$ Extract $\nabla \boldsymbol{W}_{l,p}[bs, N_l]$ from $\nabla \boldsymbol{W}_l[bs, N_l * size]$ according to *rank*
13: $\quad\quad \nabla \boldsymbol{W}_{l,p}[bs, N_l] = \nabla \boldsymbol{W}_{l,p}[bs, N_l] * (\partial \boldsymbol{O}_{l,p}[bs, N_l] / \partial \boldsymbol{Y}_{l,p}[bs, N_l])$
14: $\quad$ **end for**
15: **end for**
16: **for all** $p \in MPI\_Processes$ **do**
17: $\quad$ Update parameters
18: **end for**

The Alternate Split (Altsplit) Approach

- Alternate Split (Altsplit)
  - Split every other layer
  - Replicate the rest of the layers
- Cut 50% of the communication
- Add 25% extra computation



The Alternate Split (Altsplit) Approach

Algorithm 11 State-of-the-art approach to model parallelism of DNN

1: **for all** $p \in MPI\_Processes$ **do**
2: **for** $l = 1 \ldots L$ **do**
3: MPI_Allgather on $\boldsymbol{O}_{l-1}[bs, N_{l-1} * size]$ from all local $\boldsymbol{O}_{l-1}[bs, N_{l-1}]$
4: $\boldsymbol{Y}_l[bs, N_l] = \boldsymbol{O}_{l-1}[bs, N_{l-1} * size] * (\boldsymbol{W}_l)^T[N_{l-1} * size, N_l]$
5: $\boldsymbol{O}_l[bs, N_l] = \phi(\boldsymbol{Y}[bs, N_l])$
6: **end for**
7: **end for**
8: **for all** $p \in MPI\_Processes$ **do**
9: **for** $l = L \ldots 1$ **do**
10: $\nabla \boldsymbol{W}_l[bs, N_l * size] = \nabla \boldsymbol{W}_{l+1}[bs, N_{l+1}] * \boldsymbol{W}_{l+1}[N_{l+1}, N_l * size]$
11: MPI_Allreduce_sum on $\nabla \boldsymbol{W}_l[bs, N_l * size]$
12: Extract $\nabla \boldsymbol{W}_{l,p}[bs, N_l]$ from $\nabla \boldsymbol{W}_l[bs, N_l * size]$ according to *rank*
13: $\nabla \boldsymbol{W}_{l,p}[bs, N_l] = \nabla \boldsymbol{W}_{l,p}[bs, N_l] * (\partial \boldsymbol{O}_{l,p}[bs, N_l] / \partial \boldsymbol{Y}_{l,p}[bs, N_l])$
14: **end for**
15: **end for**
16: **for all** $p \in MPI\_Processes$ **do**
17: Update parameters
18: **end for**

Algorithm 12 *Altsplit* approach to model parallelism of DNN

1: **for all** $p \in MPI\_Processes$ **do**
2: **for** $l = 1 \ldots L$ **do**
3: $\boldsymbol{Y}_l[bs, N_l] = \boldsymbol{O}_{l-1}[bs, N_{l-1}] * (\boldsymbol{W}_l)^T[N_{l-1}, N_l]$
4: **if** $l - 1 == SPLIT$ **then**
5: MPI_Allreduce_sum on $\boldsymbol{Y}_{l-1}[bs, N_l]$
6: **end if**
7: $\boldsymbol{O}_l[bs, N_l] = \phi(\boldsymbol{Y}[bs, N_l])$
8: **end for**
9: **end for**
10: **for all** $p \in MPI\_Processes$ **do**
11: **for** $l = L \ldots 1$ **do**
12: $\nabla \boldsymbol{W}_l[bs, N_l] = \nabla \boldsymbol{W}_{l+1}[bs, N_{l+1}] * \boldsymbol{W}_{l+1}[N_{l+1}, N_l]$
13: **if** $l + 1 == SPLIT$ **then**
14: MPI_Allreduce_sum on $\nabla \boldsymbol{W}_l[bs, N_l]$
15: **end if**
16: $\nabla \boldsymbol{W}_{l,p}[bs, N_l] = \nabla \boldsymbol{W}_{l,p}[bs, N_l] * (\partial \boldsymbol{O}_{l,p}[bs, N_l] / \partial \boldsymbol{Y}_{l,p}[bs, N_l])$
17: **end for**
18: **end for**
19: **for all** $p \in MPI\_Processes$ **do**
20: Update parameters
21: **end for**

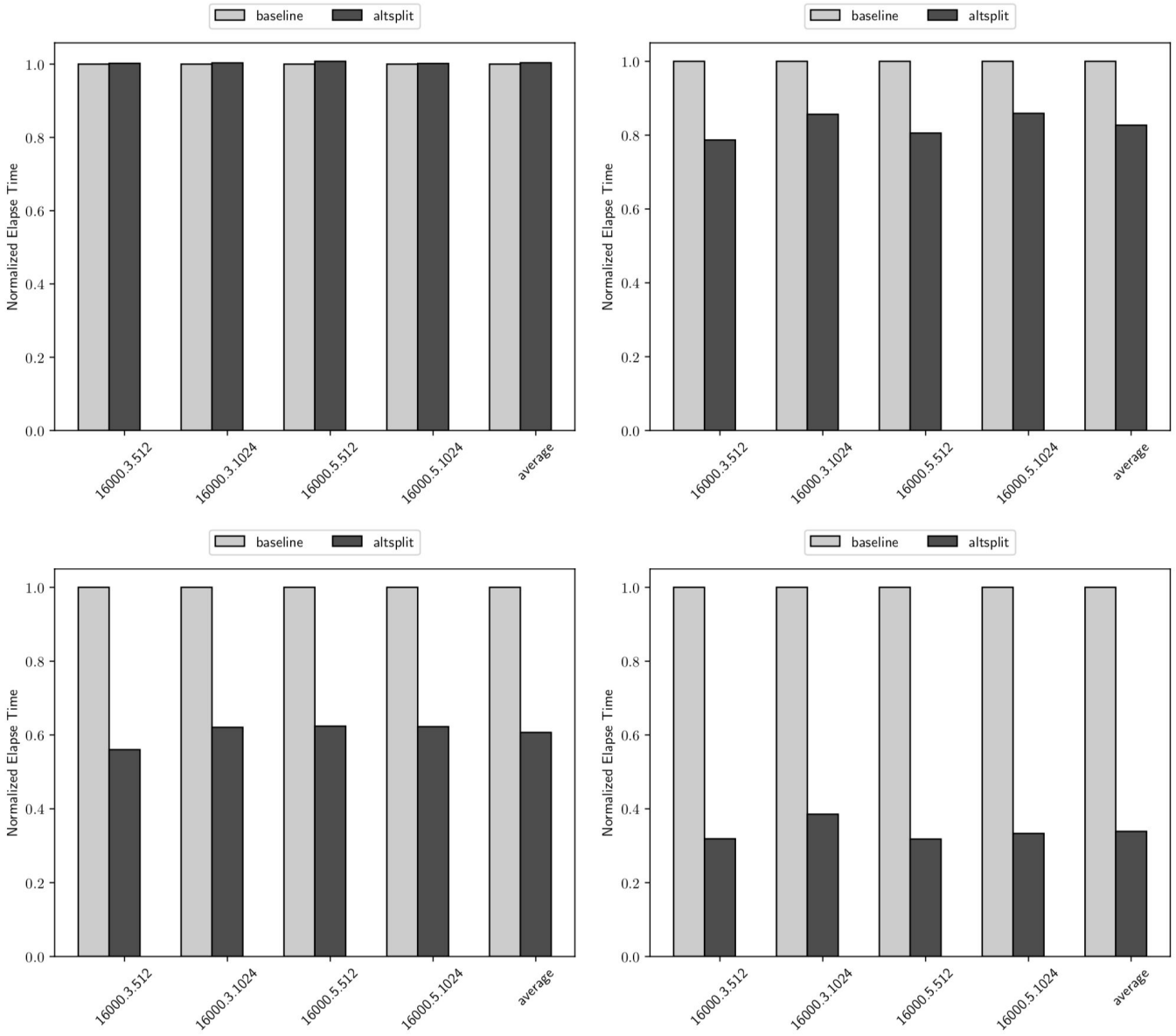
Experimental Setup

- KANN [1]
  - A C/C++ deep learning framework
  - Computation graph based
  - Insert MPI calls to relevant computation nodes
- OpenMPI
- 2 HPC clusters
  - 3456 nodes each contains 48-core Intel Xeon Haswell
  - 52 nodes each contains 40-core IBM POWER9

[1] KANN: A lightweight C library for artificial neural networks. https://github.com/attractivechaos/kann, 2019

Scalability of Backpropagation

- 4 configurations of MLP networks
  - 16k neuron-per-layer, 3-layer, batch size of 512
  - 16k neuron-per-layer, 3-layer, batch size of 1024
  - 16k neuron-per-layer, 5-layer, batch size of 512
  - 16k neuron-per-layer, 5-layer, batch size of 1024
- Dataset: CIFAR-10
- Measure elapsed time of 50 batches
- 40 cores per node
- 80, 160, 320, 640 MPI processes



Network Versatility

Parameters			Machine	
Neurons	Layers	BS	**x86**	**POWER9**
16k	3	512	68.12%	58.65%
		1024	61.47%	56.12%
	5	512	68.21%	58.43%
		1024	66.68%	55.47%
Average			66.12%	57.16%
32k	3	512	45.28%	32.75%
		1024	37.69%	31.58%
	5	512	43.87%	34.37%
		1024	37.58%	31.91%
Average			41.10%	32.65%

Performance improvements over the *baseline* on 640 MPI processes

Conclusions

- More efficient model parallelism approach to DNN
  - Trade communication with local floating-point computation
  - All-to-all communication occur at every other layers instead of a lock-step fashion
- Achieves significant speedup over baseline regardless underlying CPU architecture
- Consistent among multiple configurations of networks
- Further explore the number of splits before a replication takes place

Outline

- Background
- Communication reduction in Conjugate Gradient Method
- Communication reduction in DNN training on multi-GPU environments
- Communication reduction in model parallelism for DNN
- Conclusions and future works

Conclusions and Future Works

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusions

- Techniques to alleviate the bottlenecks introduced by communication in modern HPC systems
- No one-size-fit-all approach
  - Exploiting the resilience towards accumulation of rounding errors and loss of precision
  - Trading with a decreased computational precision with a marginal deterioration of accuracy
  - Trading at the cost of additional computation
- Need deep understanding into the synergy of communication with other aspects of an algorithm
  - Generalization

Publications

Publications related with the thesis

- Sicong Zhuang and Marc Casas. Iteration-Fusing conjugate gradient. In *Proceedings of the international Conference on Supercomputing,* ICS'17, pages 21:1-21:10, New York, NY, USA, 2017, ACM
- Sicong Zhuang, Cristiano Malossi and Marc Casas. Reducing Data Motion to Accelerate the Training of Deep Neural Networks (will be submitted to IPDPS'20)
- Sicong Zhuang, Panagiotis Hadjidoukas, Cristiano Malossi and Marc Casas. Altsplit: Communication Reduction In DNN Model Parallelism (future submission)

Other publications

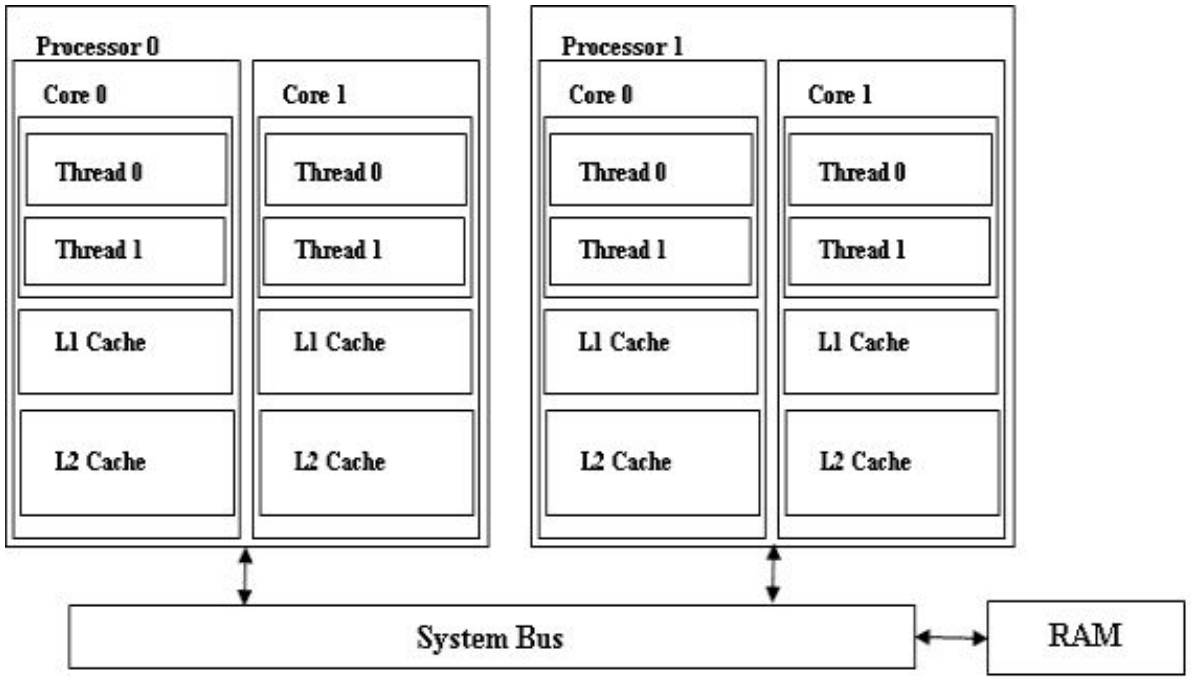
- Ilia Pietri, Sicong Zhuang, Marc Casas, Miquel Moretó and Rizos Sakellariou. Evaluating scientific workflow execution on an asymmetric multicore processor. In *Euro-Par 2017: Parallel Processing Workshops,* pages 439-451, Cham, 2018. Springer International Publishing

Thank you

sicong.zhuang@bsc.es

Modern HPC Systems

- Massively parallel system (multi-threaded + multi-core + multi-node)



Parallel Computing

- Embarrassingly parallel problems
  - Each chunk is inherently independent
  - Solved in a divide-and-conquer fashion
- Other parallel problems
  - Have non-parallelizable regions
  - Create interleaving parallel-sequential-parallel computation pattern
  - Require communication/synchronization
  - Commonplace among numerical linear algebra and other computational sciences

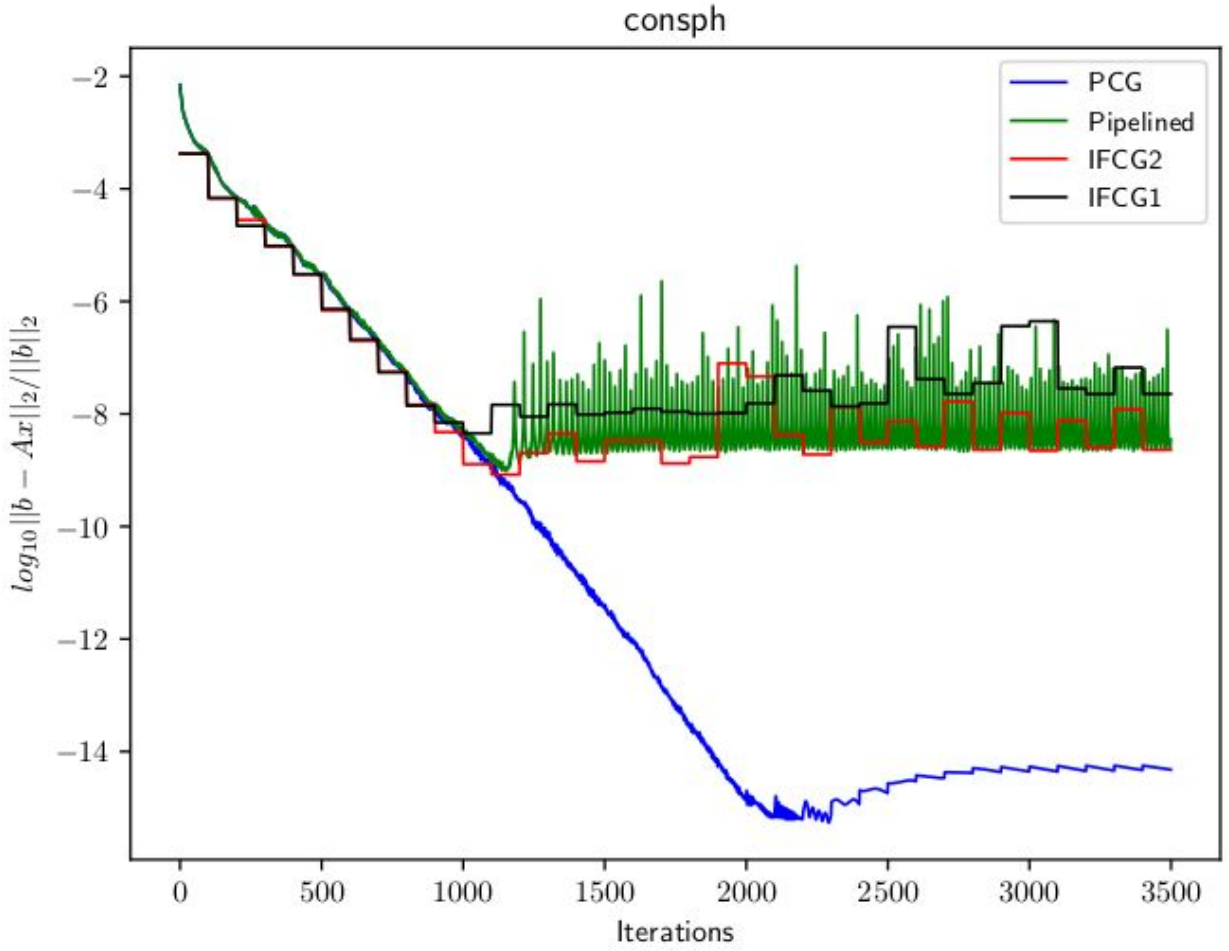
A Task-Based Implementation

- The task parallelism paradigm
  - Running many different tasks simultaneously on different data
  - In C/C++, using #pragma annotations provided by OpenMP to specify tasks, their data dependencies, etc.
  - Data dependencies are managed by the runtime by maintaining and looking up its Task Dependence Graph (TDG)

- Pipelined CG, IFCG1 and IFCG2 can easily adopt this paradigm:
  - Neat data dependencies among operations
  - Each operation can trivially be split into smaller dependence-free tasks

Numerical Stability

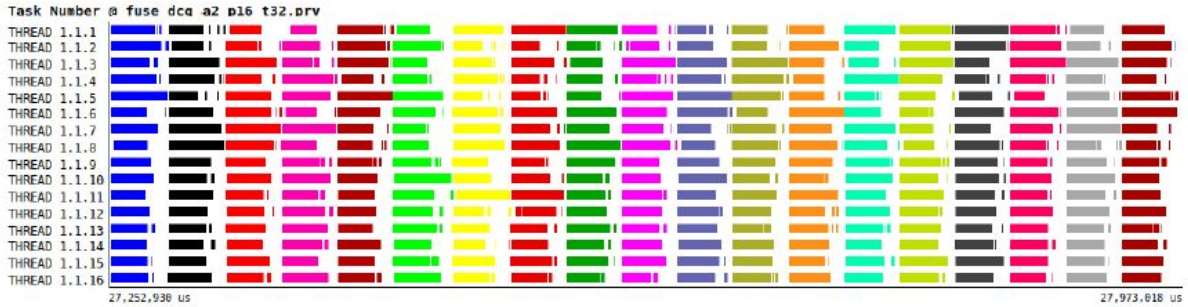
- Reduced precision compared with PCG
- On par with Pipelined CG



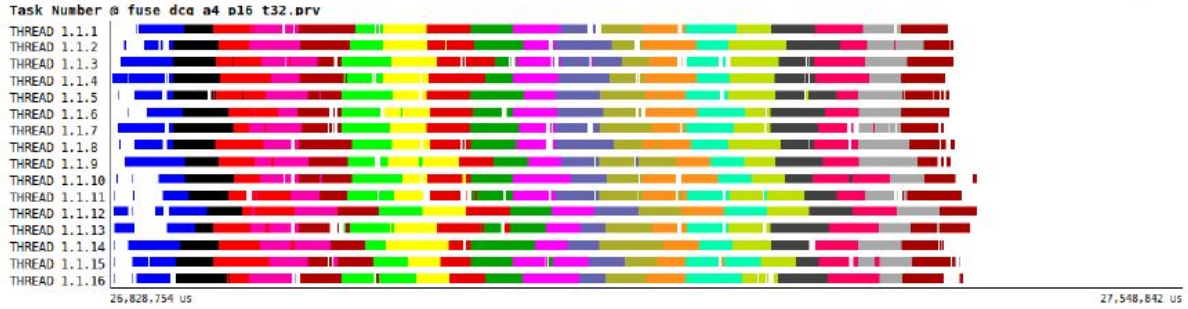
Evaluation

- 16-core run with 19-iterations (af_shell8)
- Iterations are marked by distinct colors
- Blank area indicates idle time and system software activities

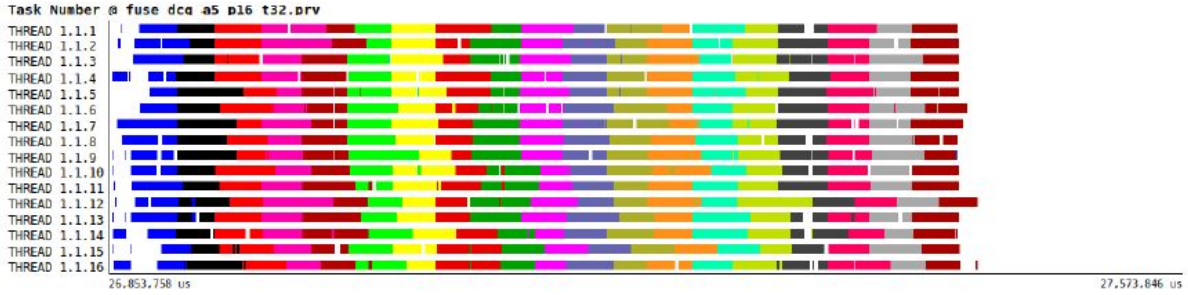
Pipelined CG



IFCG1

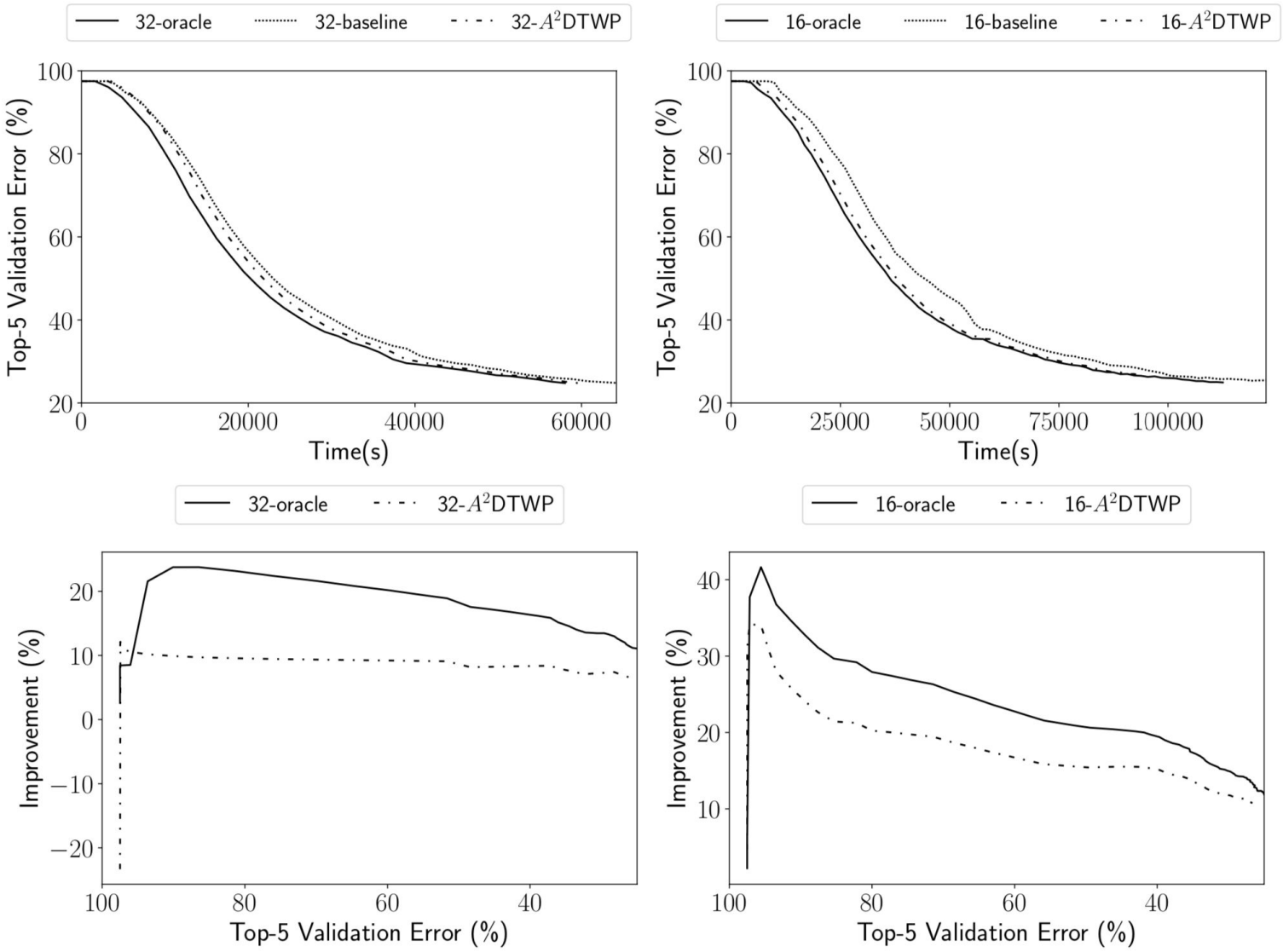


IFCG2



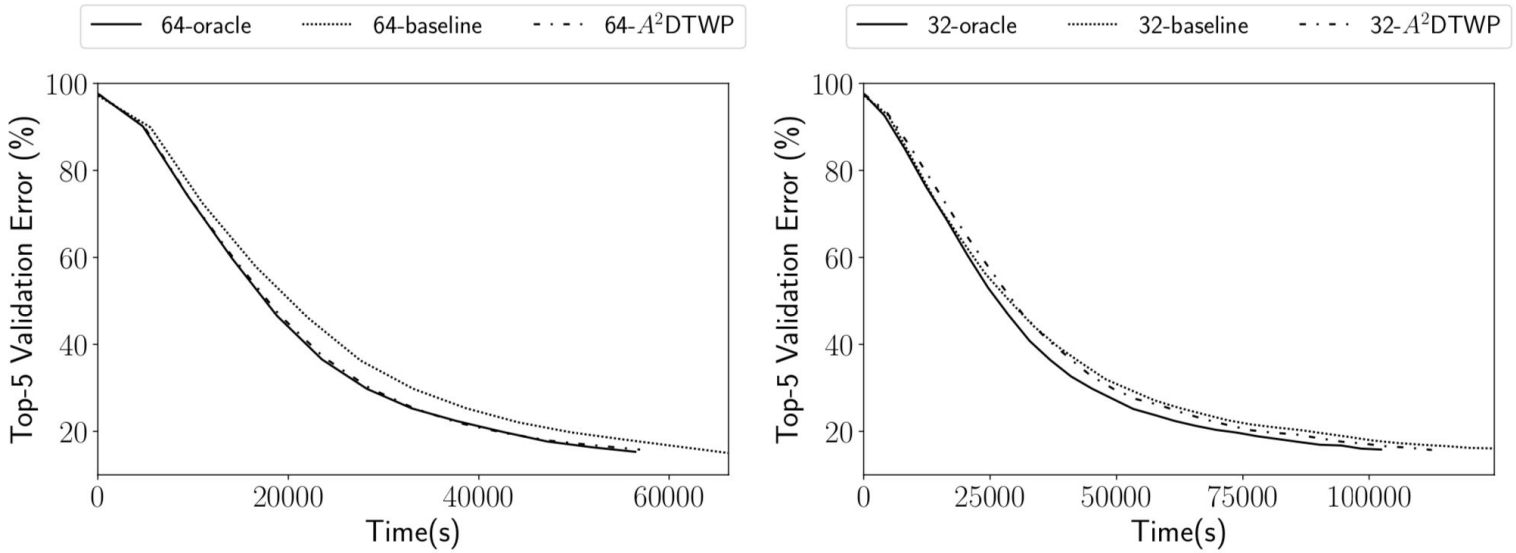
Evaluation on Alexnet

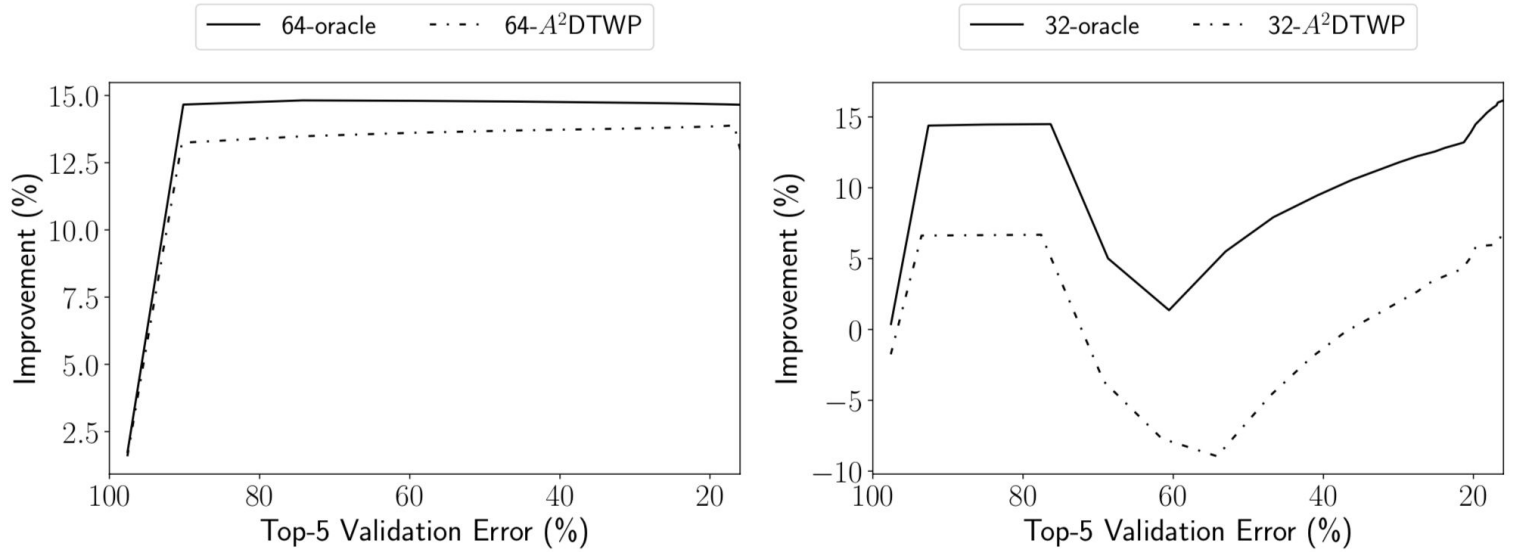
- BS: 32, 16
- T = -0.05
- INTERVAL = 4000



Evaluation on VGG

- BS: 32, 64
- T = -0.002
- INTERVAL = 4000

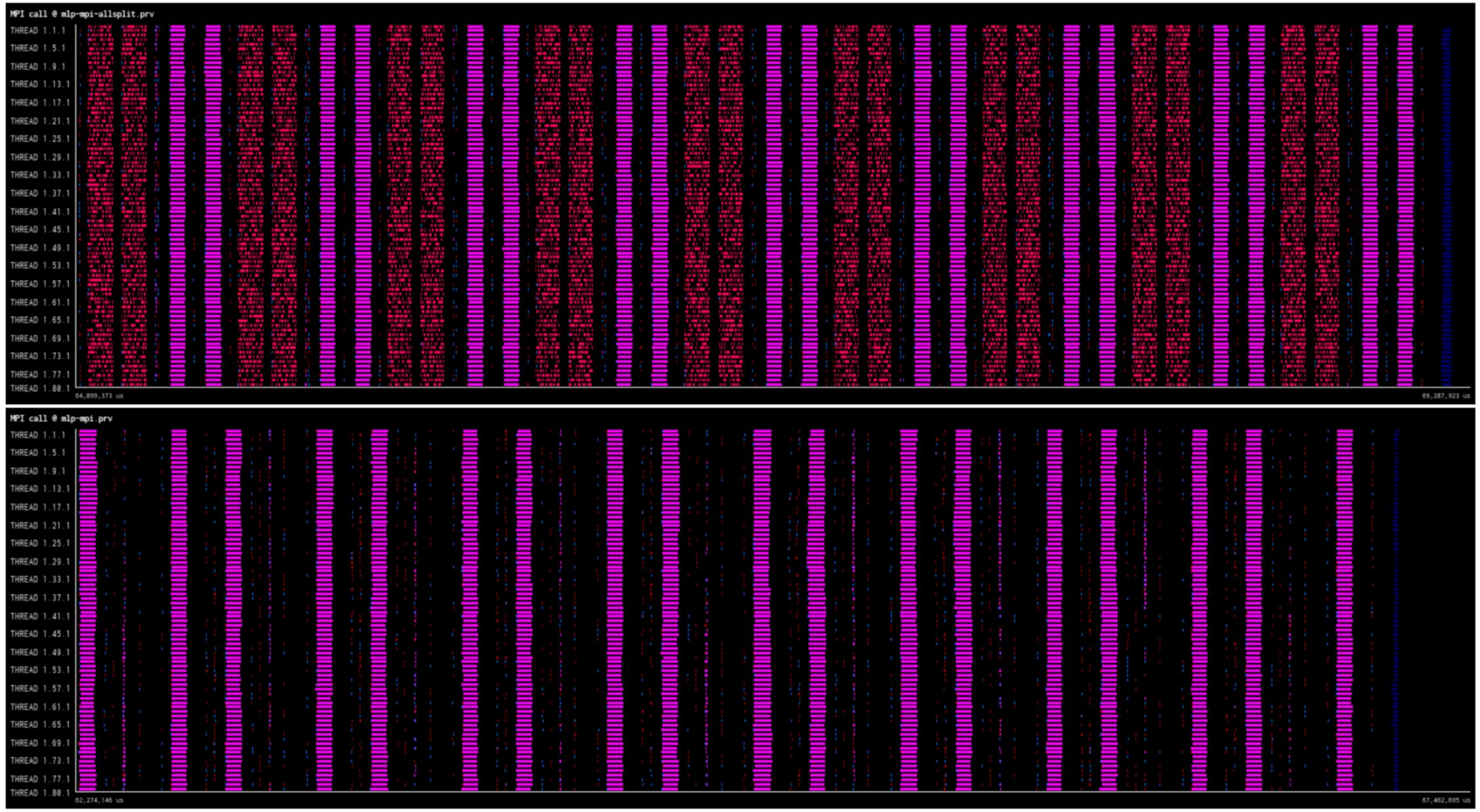




Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Traces

- 80 MPI Threads
- 9-batche run
- MPI_Allreduce in pink
- MPI_Allgather in red



Top: *baseline*, Bottom: *altsplit*

Barcelona Supercomputing Center
Centro Nacional de Supercomputación