MPI and Distributed DL

Ulrich Finkler – Wei Zhang

CSCI-GA.3033-020 HPML

Performance Factors

Algorithms Performance

- **Communication patterns**
- **Performance Modeling**

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- Python, PyTorch Distributed DL

Libraries Performance

- CUDA, cuDNN, cuBLAS, **Communication Libraries (MPI, Gloo)**

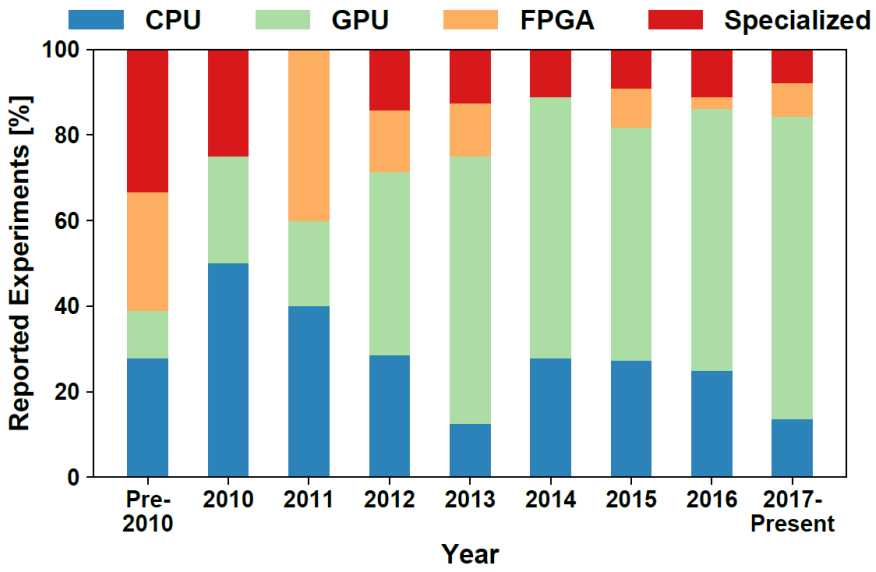
Hardware Performance

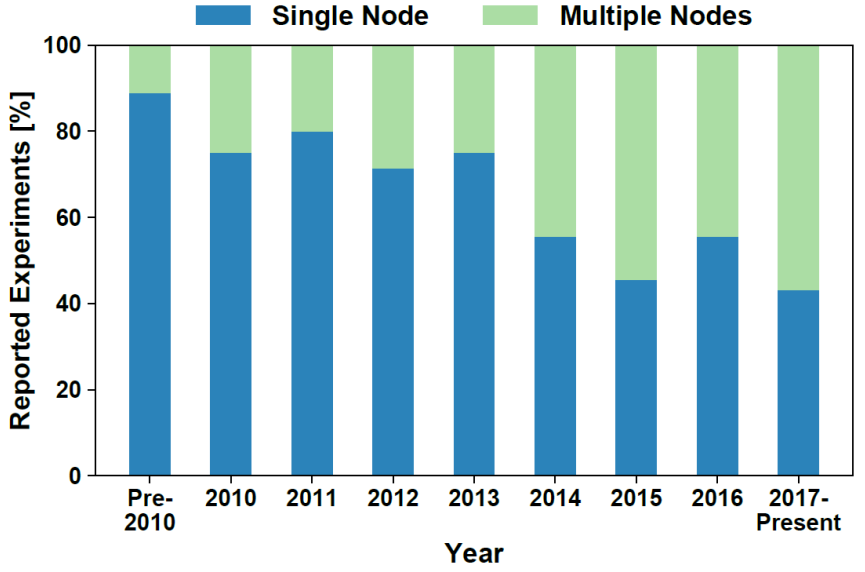
- CPU, DRAM, GPU, HBM, Tensor Units, Disk/Filesystem, **Network**

DL Hardware Trends

Trend: From Parallel to Distributed DL

- Deep Learning has always exploited **parallel architectures**
- During the years:
  - DL has been moving towards **distributed architectures**
  - **GPUs** have become the default compute architecture

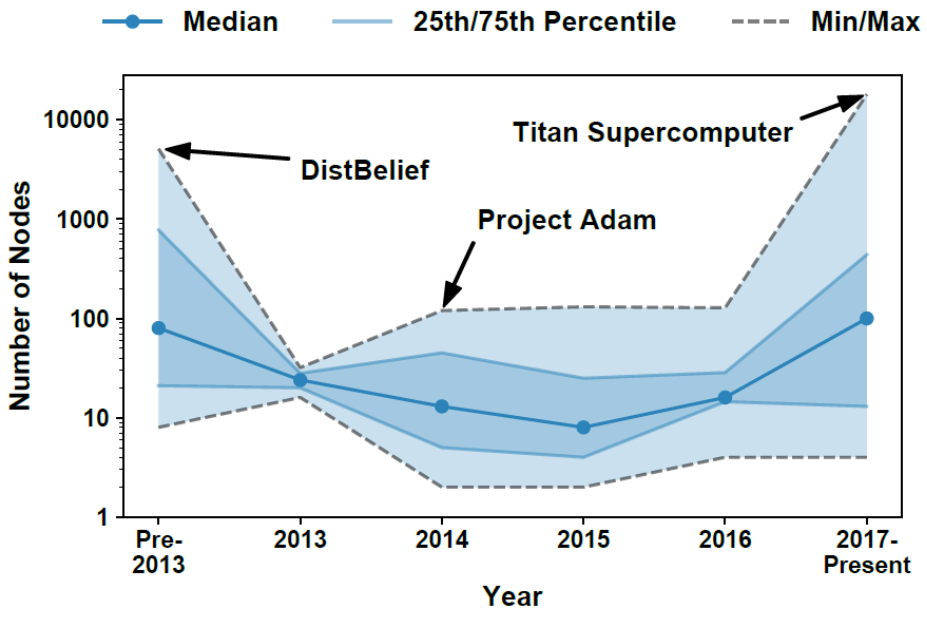




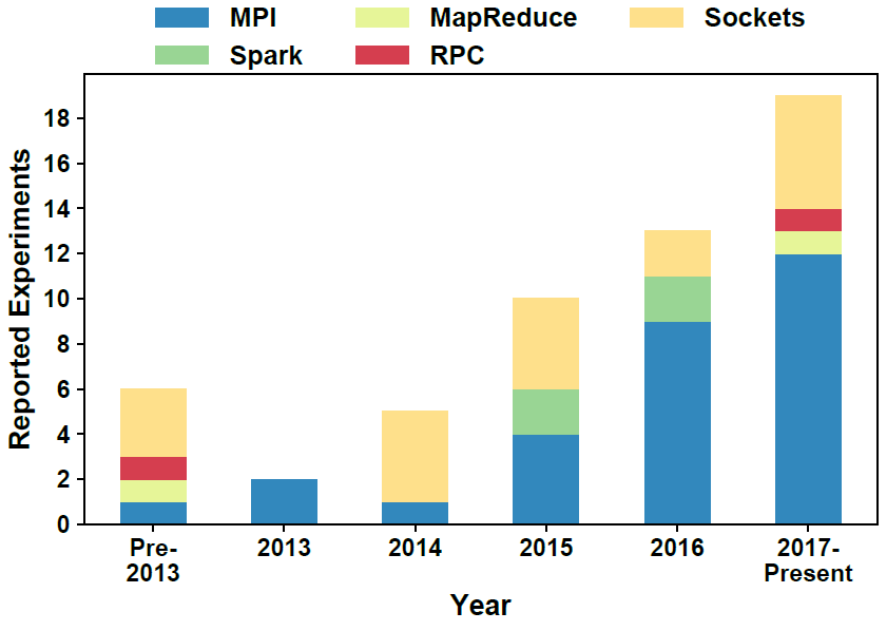
From:https://arxiv.org/abs/1802.09941

Trend: Growing scale for Distributed DL

- GPUs are the main focus from 2013
- Node count has been growing since 2015
- MPI adoption becomes larger



(a) Node Count



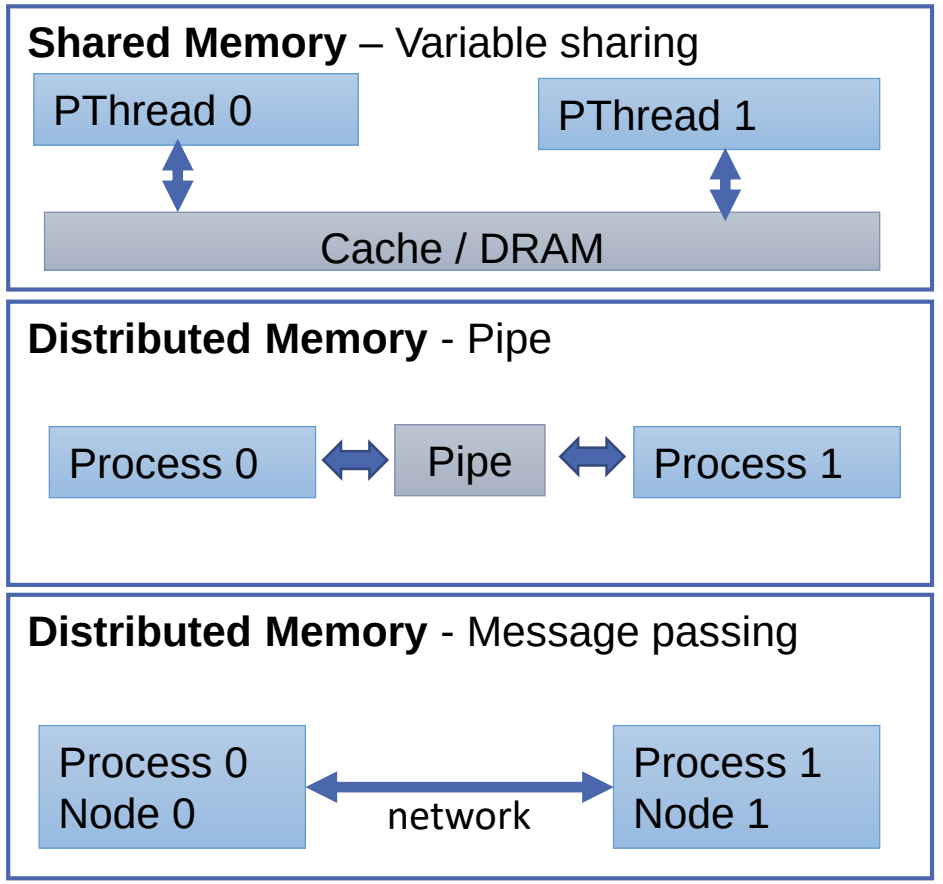
(b) Communication Layer

From:https://arxiv.org/abs/1802.09941

Message Passing Interface

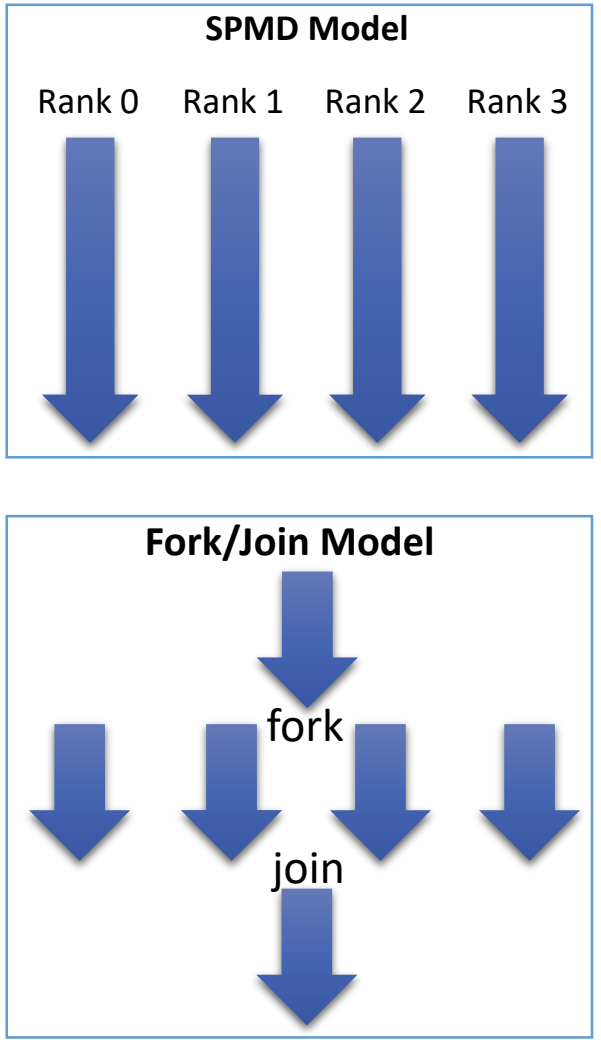
Distributed vs. Shared Memory Programming

- Shared Memory:
  - Access the same virtual address space
  - Pass data by **reference**
  - Example**:**
    - Share a variable with pthreads
- Distributed Memory:
  - Separate Virtual Address Spaces
    - Cannot access shared data structures
  - Pass data by **value** (copy)
  - Examples:
    - Send an element through a process pipe
    - Send message across network



SPMD vs Fork/Join

- **SPMD (Single Program Multiple Data)**: execution starts in parallel (different from fork-join style)
  - **MPI** implements SPMD
  - 1 program executed by multiple processes (MPI ranks)
  - Programmers need to think in parallel from the beginning of the execution
  - **Static parallelism**: number of processes does not change
- **Fork/Join**: execution starts serial new processes or threads are created at fork operation
  - pthread_create()/fork() /OpenMP are examples of fork/join
  - **Dynamic parallelism**: number of processes can change at each fork



TCP vs MPI

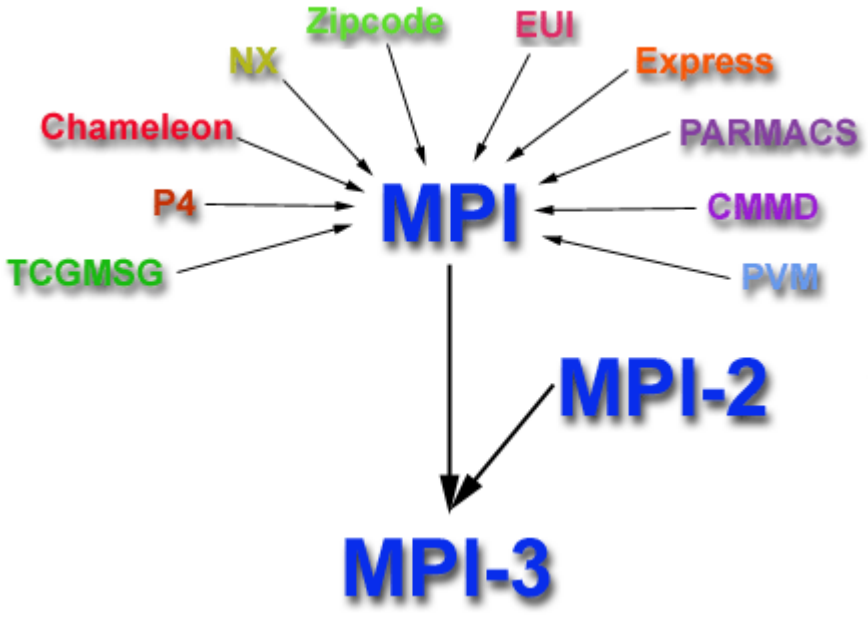
TCP stack	MPI stack
Connections based on IP addresses and ports	Based on **rank number**
Point-to-point communication	Point-to-point, **collectives, one-sided**
Stream-oriented	**Message** Oriented
Raw data (bytes/octets)	**Typed** messages
Network independent	Network independent
high latency	**low latency**

Message Passing Interface

- What is MPI?

  MPI is a **standard specification** of a message passing interface for large scale systems (Supercomputers)

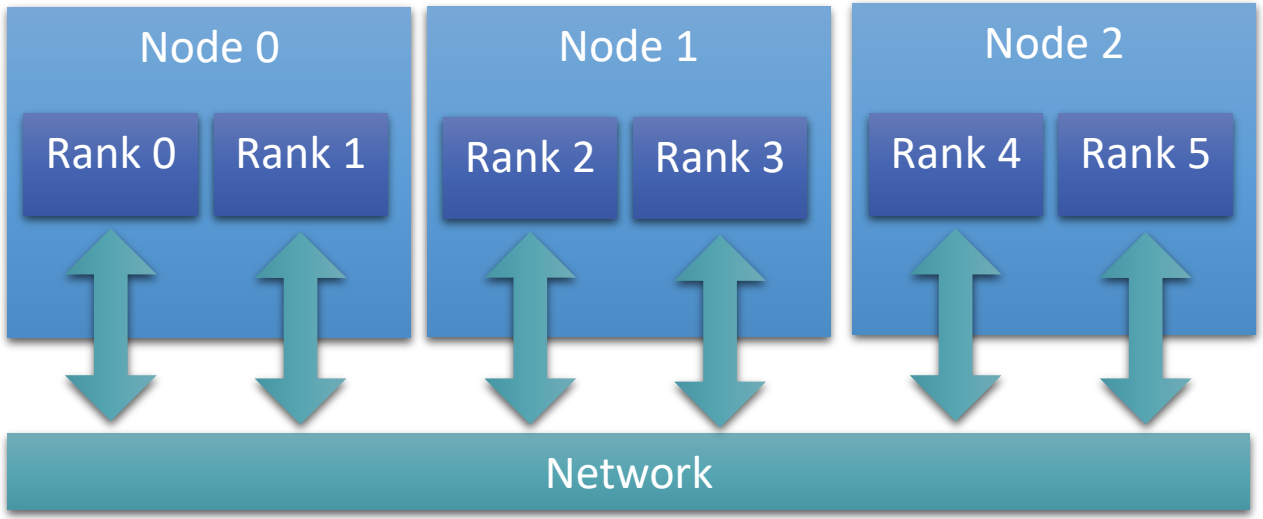
- Advantages
  - Standardization
  - Portability
  - Performance optimization
  - Functionality
- First version released: MPI-1.0 in 1994
- Today: MPI 3.1



From: https://computing.llnl.gov/tutorials/mpi/

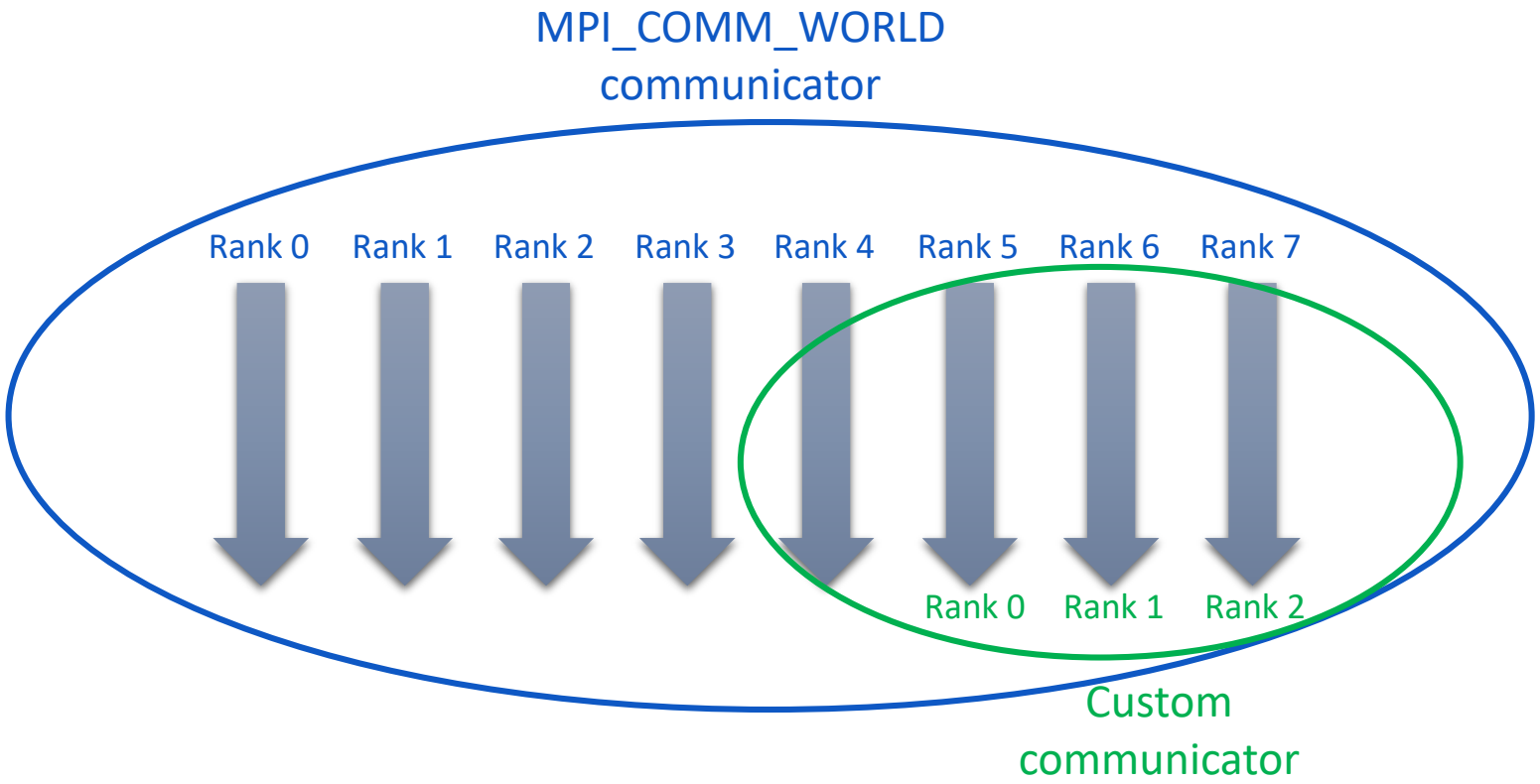
MPI – System components

- Node:
  - A single **host** on the network
- Rank:
  - A **process** executing the MPI program
- MPI is executed on multiple nodes (thousands easily)
- Each node can have many processes
  - Usually one per core



MPI Programmer View

- Nodes are transparent to programmer
- Only ranks matter
- Communicator:
  - A group of ranks that can communicate
- MPI_COMM_WORLD
  - Communicator that includes all the ranks



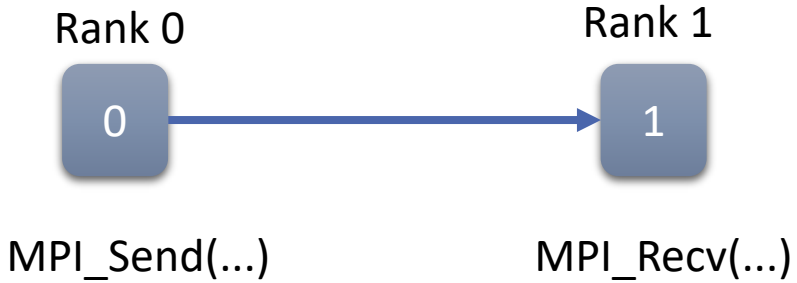
MPI Data Types

- MPI messages types and size is specified in advance
  - higher performance
  - less programming errors with types
- A message is always composed by one or more elements of MPI datatype size
- Definition of new types is possible
  - example: new data structures
- MPI_Byte and also dynamic sizing is also available for special cases

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Point-to-Point

- **MPI_Send**() and **MPI_Recv**() can happen in any order: MPI runtime will take care to deliver the message
- Destination is only based on the rank and communicator (0 or 1)
- Message is **typed**, we are not sending just “bytes”
- Can multiple int (float, double) etc.




```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status)

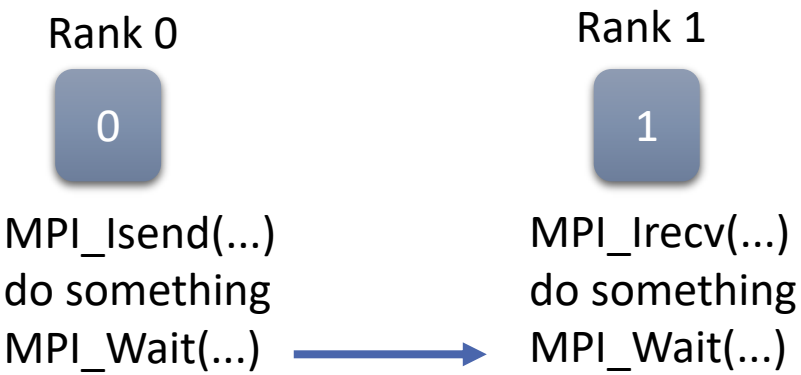
Point-to-Point – MPI_Send() Semantic

- When **MPI_Send**() returns:
  - We **can reuse the communication buffers**
  - **It does not imply** that the message arrived
  - **It does not imply** that the message left the local node
- How to know if a message actually arrived?
  - Need to send an MPI message back
  - Use **one-sided MPI primitives** to write directly to the receiver memory using RDMA (not part of the course)

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    number = 0 // reuse buffer
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
    number = 0; // reuse buffer
}
```

Non Blocking Point-to-Point

- **MPI_ISend**() and **MPI_IRecv**() can happen in any order: MPI runtime will take care to deliver the message
- Isend and Irecv are **non-blocking**: process can **continue execution** and then **wait** later

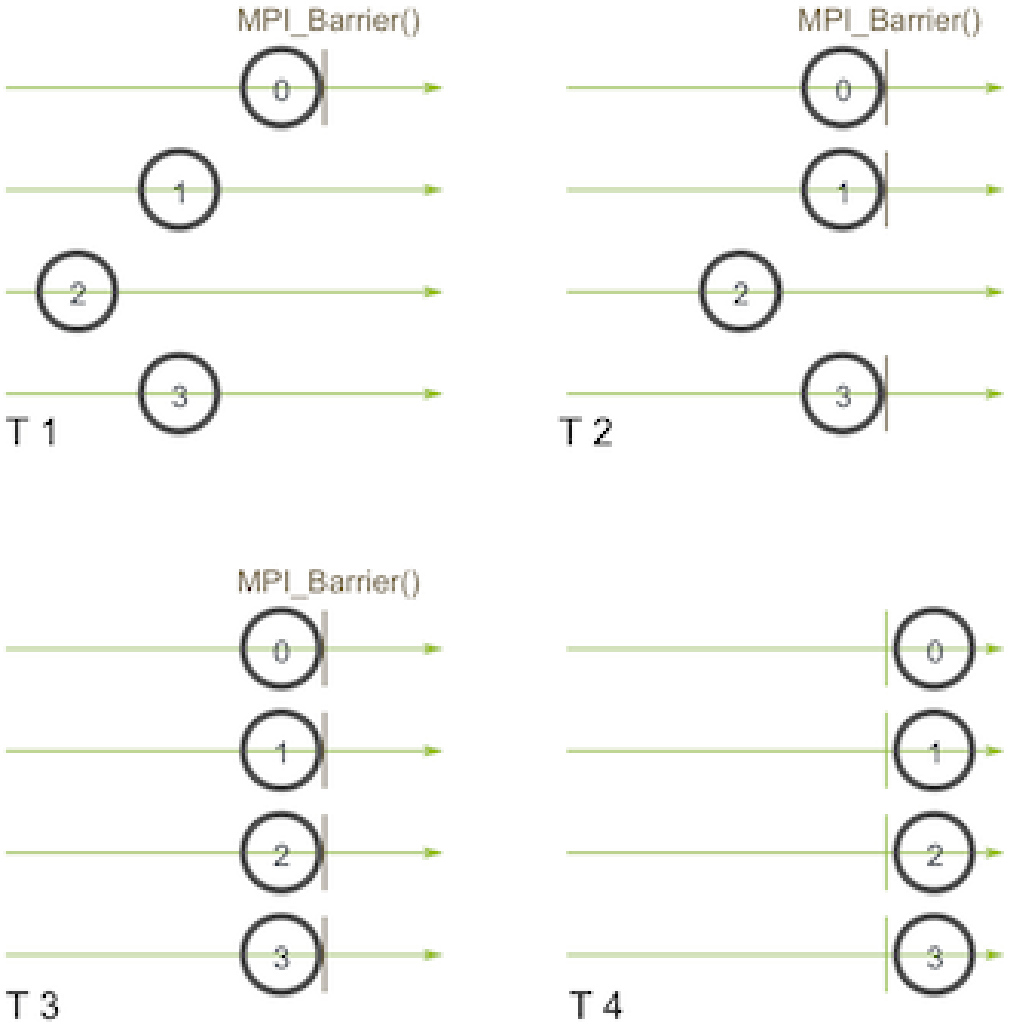



```
int myid, numprocs, left, right;
int buffer[10], buffer2[10];
MPI_Request request, request2;
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
right = (myid + 1) % numprocs;
left = myid - 1;
if (left < 0)
     left = numprocs - 1;
MPI_Irecv(buffer, 10, MPI_INT, left, 123, MPI_COMM_WORLD, &request);
MPI_Isend(buffer2, 10, MPI_INT, right, 123, MPI_COMM_WORLD, &request2);
MPI_Wait(&request, &status);
MPI_Wait(&request2, &status);
MPI_Finalize();
```

Barriers and Synchronization

- **MPI_Barrier()** is used to synchronize ranks
- It is a fundamental primitive for parallel programming
- Barrier implementation and **latency** are key to many algorithms performance

```
MPI_Barrier(MPI_Comm communicator)
```

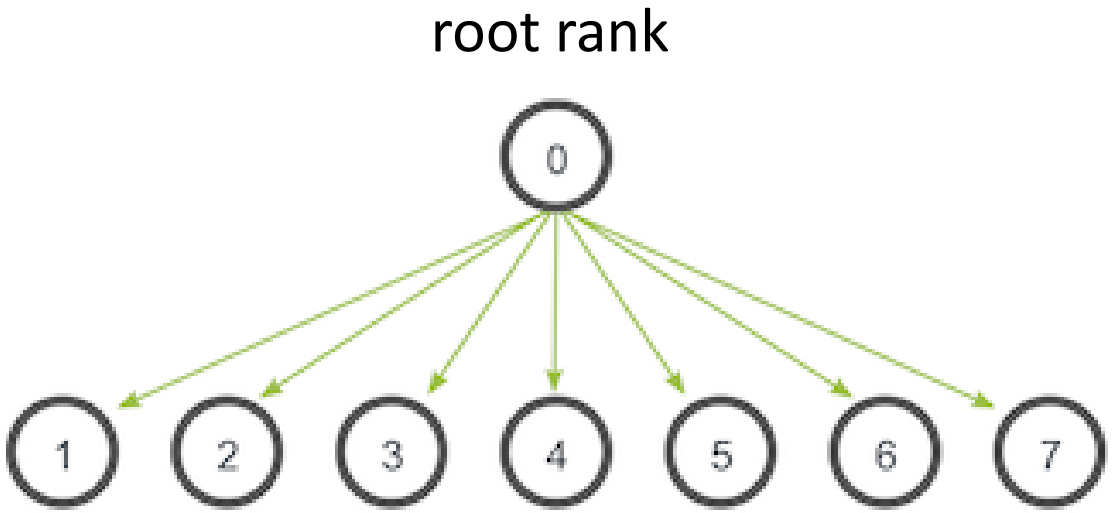


Collectives

- **Collectives**: communication patterns that involve data exchange across multiple ranks
- Type of collectives:
  - Reduce, AllReduce
  - Gather, AllGather
  - Scatter
  - AllToAll
  - Broadcast
- Collectives represent the biggest value of MPI implementations
- Some collectives can be **hardware accelerated (ex barriers, or reduce)**

Broadcast

- Root rank (can be any rank) sends the **same single message** to multiple receivers
- Message can contain multiple elements of type DataType

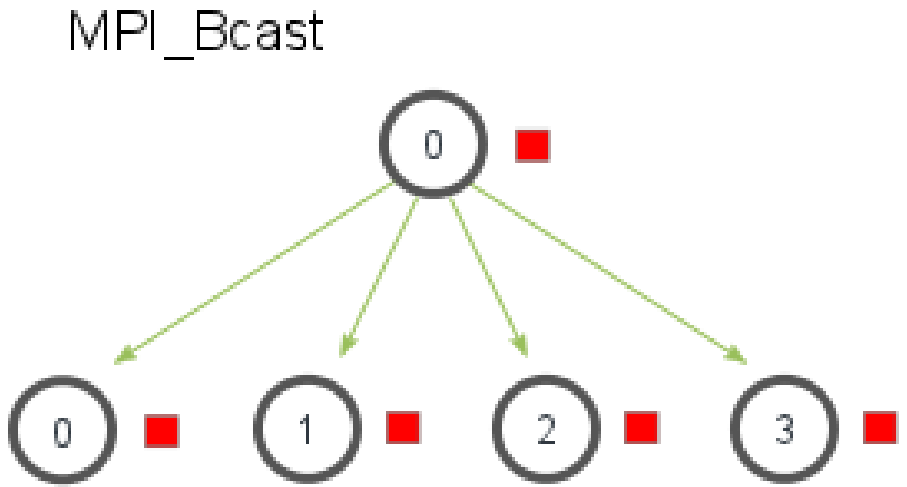


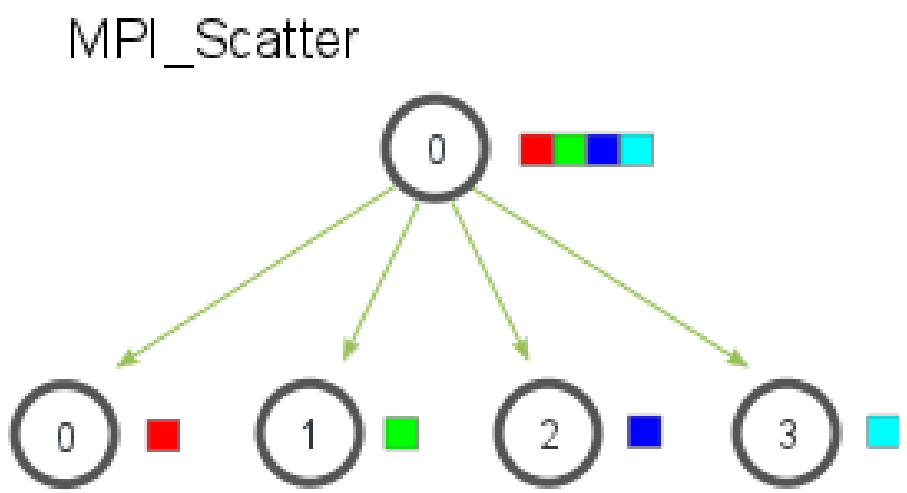

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

Scatter

- Root rank sends the **a different single message** to each receiver

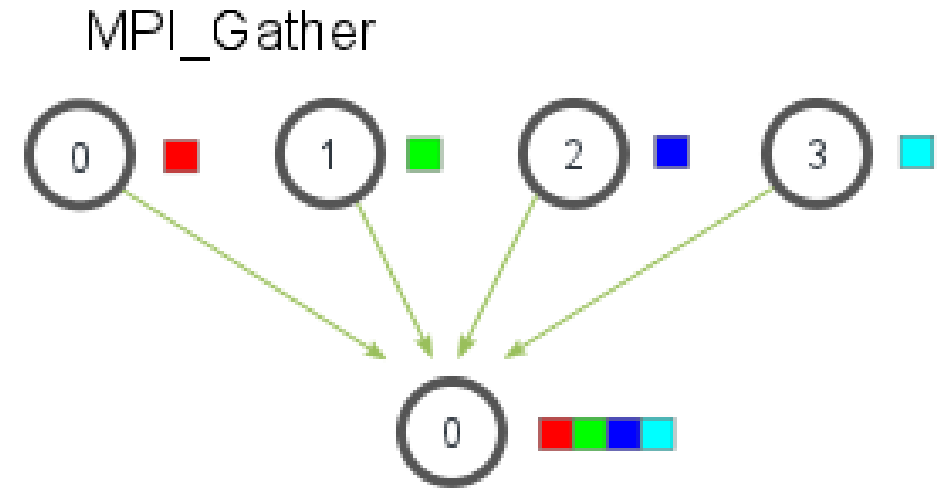
```
MPI_Scatter(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```





Gather

- Root rank gather **a different single message** from each receiver




```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

Average with Gather and Scatter

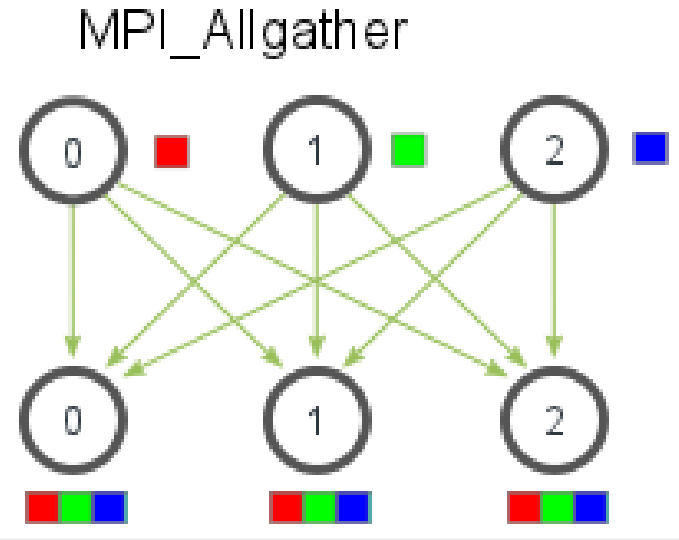
- Gather and Scatter example to compute the average of FLOAT numbers
- Ranks are in *world_rank*

```
if (world_rank == 0) {
  rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
          elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
  sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
          MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
  float avg = compute_avg(sub_avgs, world_size);
}
```

All Gather

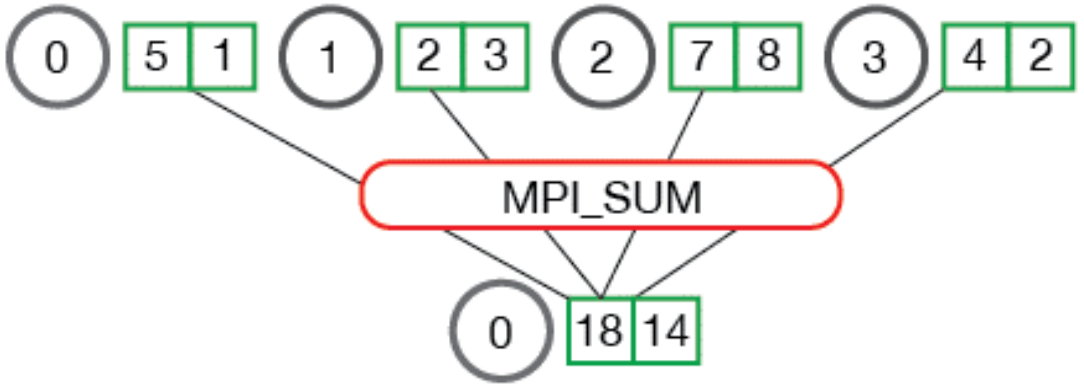
- All Gather is composed of one gather for each rank
- Useful when all ranks need the result of the gather

```
MPI_Allgather(
   void* send_data,
   int send_count,
   MPI_Datatype send_datatype,
   void* recv_data,
   int recv_count,
   MPI_Datatype recv_datatype,
   MPI_Comm communicator)
```



Reduce

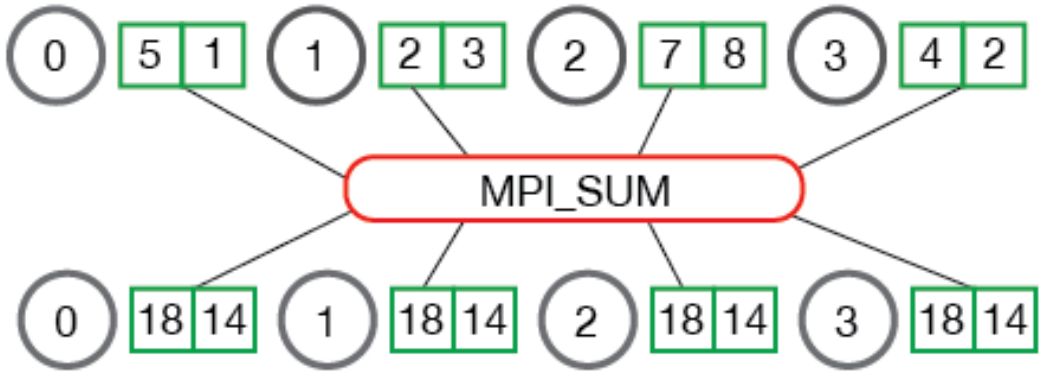
- Reduces a set of numbers into a smaller set of numbers via a function
- In the picture MPI_SUM is used as reduce function
- Reduce can involve multiple elements per rank




```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

All Reduce

- Reduces a set of numbers into a smaller set of numbers via a function
- **Each rank obtains the result**
- In the picture MPI_SUM is used as reduce function
- All Reduce can involve multiple elements per rank




```
MPI_Allreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```

Reduce – Available Functions

- **MPI_MAX** - Returns the maximum element
- **MPI_MIN** - Returns the minimum element
- **MPI_SUM** - Sums the elements
- **MPI_PROD** - Multiplies all elements
- **MPI_LAND** - Performs a logical and across the elements
- **MPI_LOR** - Performs a logical or across the elements
- **MPI_BAND** - Performs a bitwise and across the bits of the elements
- **MPI_BOR** - Performs a bitwise or across the bits of the elements
- **MPI_MAXLOC** - Returns the maximum value and the rank of the process that owns it
- **MPI_MINLOC** - Returns the minimum value and the rank of the process that owns it

StdDev with All Reduce

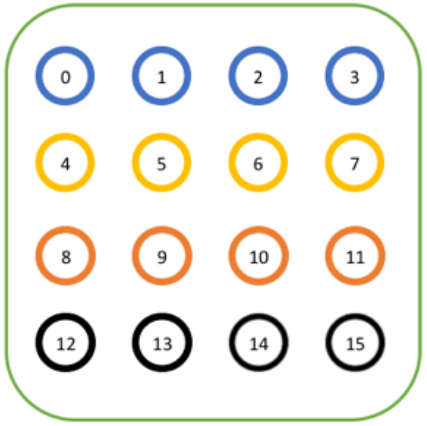
- Using all reduce to compute the standard deviation of a set of numbers
- Each rank obtains the result

```
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Reduce all of the local sums into the global sum in order to
// calculate the mean
float global_sum;
MPI_Allreduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM,
              MPI_COMM_WORLD);
float mean = global_sum / (num_elements_per_proc * world_size);

// Compute the local sum of the squared differences from the mean
float local_sq_diff = 0;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sq_diff += (rand_nums[i] - mean) * (rand_nums[i] - mean);
}

// Reduce the global sum of the squared differences to the root
// process and print off the answer
float global_sq_diff;
MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// The standard deviation is the square root of the mean of the
// squared differences.
if (world_rank == 0) {
  float stddev = sqrt(global_sq_diff /
                      (num_elements_per_proc * world_size));
  printf("Mean - %f, Standard deviation = %f\n", mean, stddev);
}
```

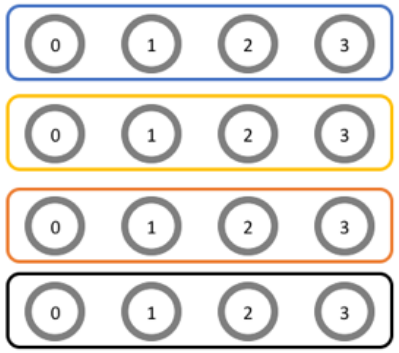
Multiple Communicators

- MPI_COMM_WORLD can be split in multiple communicators to allow communication in sub-groups of processes
- Rank numbers and communication are always associated with a communicator
- Example shows how to split communicators with *MPI_Comm_Split()*

MPI_COMM_WORLD communicator



Multiple communicators




```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank,
&row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
         world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

DL Performance Modeling

Performance Modeling for distributed systems

- Performance Modeling on current Large Scale Distributed Systems is extremely complex:
  - Network performance
    - Network Topologies
    - Network Traffic
  - Node performance
    - CPU, Multicores, Hardware Threading
    - GPU, Blocks, Shared memory
    - Memories:
      - Caches, DRAM, HBM

Performance Modeling with Cost Models

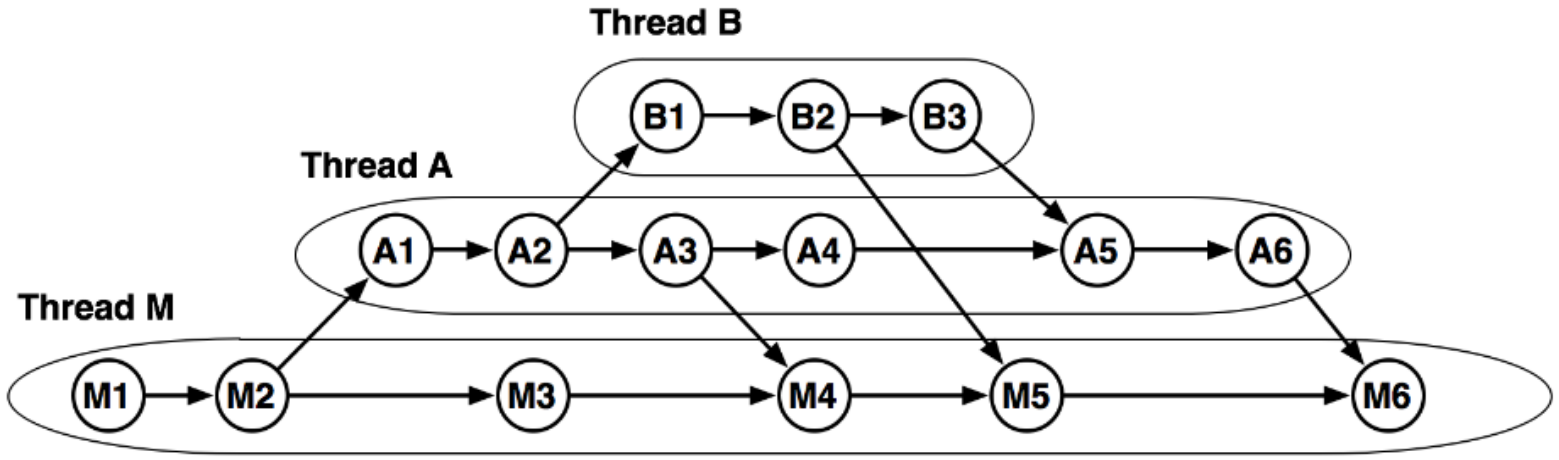
- Simplify reality with very strong assumption
- Make possible reasoning about algorithms and underlying systems
- Need to be validated with measurement and experimentation
- No substitute for actual implementation and real experiments

Work and Depth Cost Model

- Cost Model for **Parallel Algorithms** (ignores communication cost) executing on multiple processors
- **Assumptions:**
  - Communication time among processors ignored
  - Memory accesses ignored
  - Processors activity is clock-synchronized
  - Computations steps are constant size

Work and Depth Cost Model

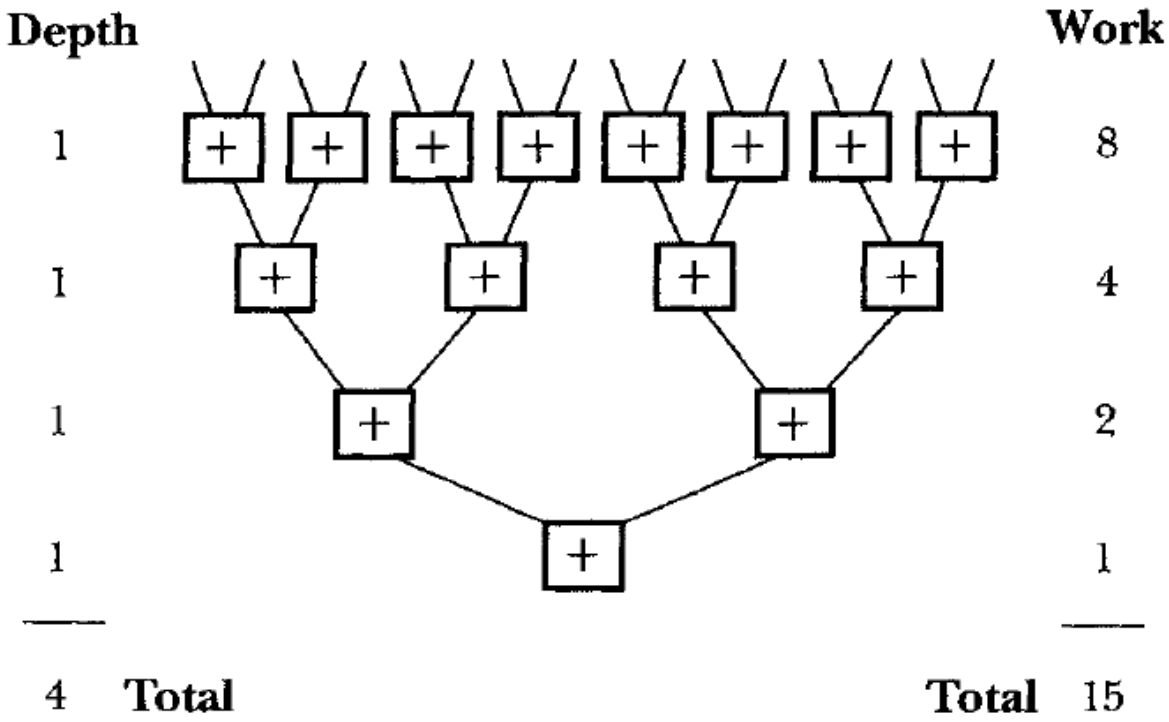
- The **execution of a parallel algorithm** can be represented by a directed acyclic graph - **DAG**
- Directed edges represent **data dependencies**
- Vertices represent a **computation**



from: http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html#_dag_representation

Work and Depth Cost Model (1)

- Work ***w*** : the total number of operations executed by a **computation**
- Depth ***d*** (also called **span**): the longest chain of sequential dependencies
- Example:
  - Reduction with SUM function



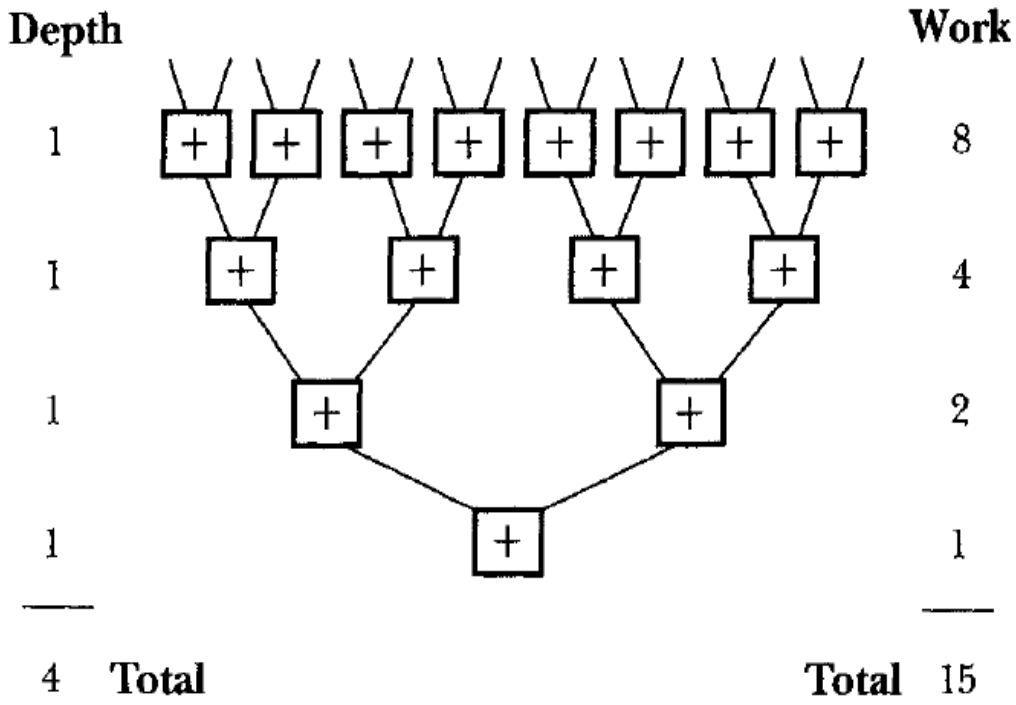
From:https://www.cs.cmu.edu/afs/cs/Web/People/blelloch/papers/B85.pdf

Work and Depth Model (2)

- We assume **1 time unit per compute operation**
- T_1: time units to **compute** work on a single processor
- T_∞: time units to **compute** work on a infinite number of processors

- Relationship to work and depth:

$$T_1 = W$$
$$T_\infty = D$$

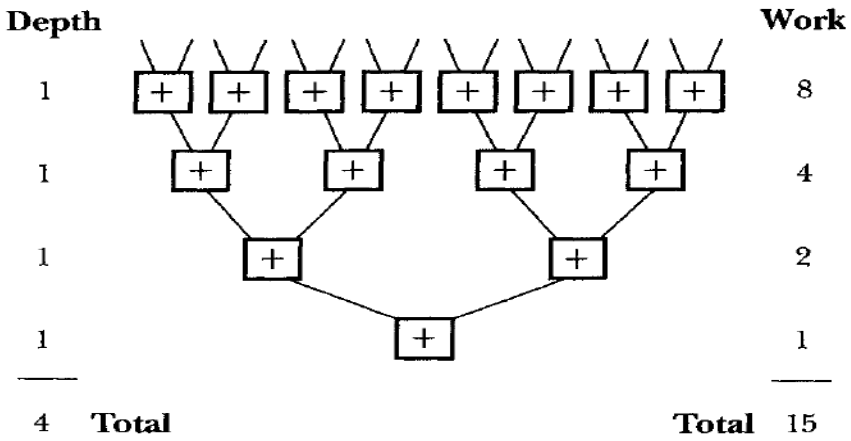


From:https://www.cs.cmu.edu/afs/cs/Web/People/blelloch/papers/B85.pdf

Work and Depth model bounds

- T_p : **execution Time** on *p* processes
  - Lower bound: max{W/p, D}
  - Upper bound: *O*(W/p + D)
  - where $\frac{W}{p} \geq 1$

$$\max\left\{\frac{W}{p}, D\right\} \leq \boldsymbol{T_p} \leq O\left(\frac{W}{p} + D\right)$$



From:https://www.cs.cmu.edu/afs/cs/Web/People/blelloch/papers/B85.pdf

Analysis of All-Reduce with Work and Depth (1)

- All-Reduce with n operations using operator $\oplus$:
  - $Y = X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5 \oplus X_6 \oplus X_7 \oplus X_n$

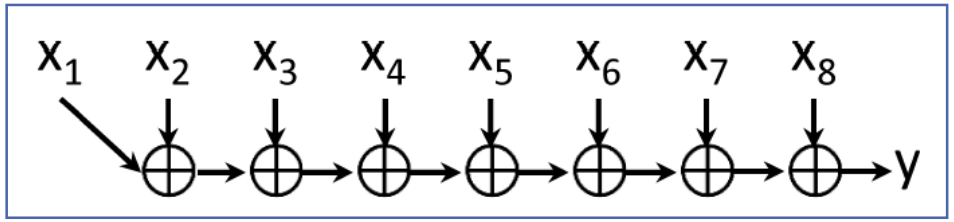
- If $\oplus$ is **non-associative** (ex. subtraction or division):
  - $W = n - 1 = 7$
  - $D = n - 1 = 7$
  - **Average Parallelism**: $W/D = 1$
  - $\boldsymbol{T_p}$ **bounds**:

$$\max\left\{\frac{W}{p}, D\right\} \leq \boldsymbol{T_p} \leq O\left(\frac{W}{p} + D\right)$$

$$\max\left\{\frac{n-1}{p}, n-1\right\} \leq \boldsymbol{T_p} \leq O\left(\frac{n-1}{p} + n - 1\right)$$

$$\mathrm{n} - 1 \leq \boldsymbol{T_p} \leq O\left((n-1)\left(1 - \frac{1}{p}\right)\right)$$

$$\mathrm{n} - 1 \leq \boldsymbol{T_p} \leq O(n-1)$$



From:https://arxiv.org/abs/1802.09941

Analysis of All-Reduce with Work and Depth (3)

- All-Reduce with n operations using operator $\oplus$:
  - $Y = X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5 \oplus X_6 \oplus X_7 \oplus X_n$

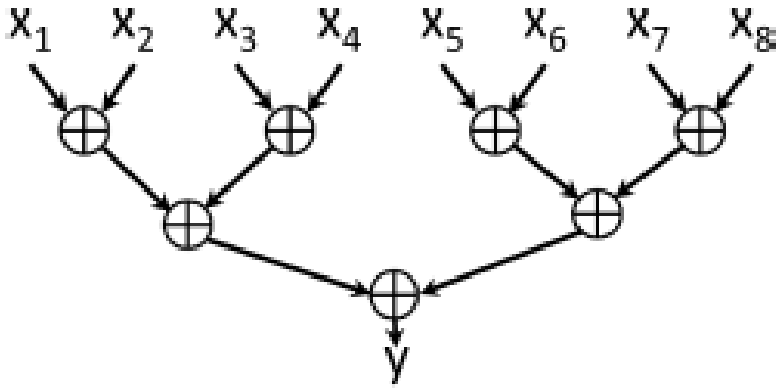
- If $\oplus$ is **associative** (ex. addition or multiplication):
  - $W = n - 1 = 7$
  - $D = log_2(n) = 2.81$
  - **Average Parallelism**: $W/D = 2.49$
  - $\boldsymbol{T_p}$ **bounds**:

$$\max\left\{\frac{W}{p}, D\right\} \leq \boldsymbol{T_p} \leq O\left(\frac{W}{p} + D\right)$$

$$\max\left\{\frac{n\ -1}{p}, log_2(n)\right\} \leq \boldsymbol{T_p} \leq O\left(\frac{n-1}{p} + D\right)$$

$$log_2(n) \leq \boldsymbol{T_p} \leq O\left(\frac{n-1}{p} + log_2(n)\right) \text{ assuming } \mathrm{p} = n\ - 1$$

$$log_2(n) \leq \boldsymbol{T_p} \leq O(log_2(n))$$



From:https://arxiv.org/abs/1802.09941

Alpha Beta Cost Model

- Network communication cost model based on **latency** and **bandwidth** only
- **Assumptions:**
  - Network costs are constant during the time
  - Latency and Bandwidth are constant across different nodes/topologies
  - No CPU costs involved in communication

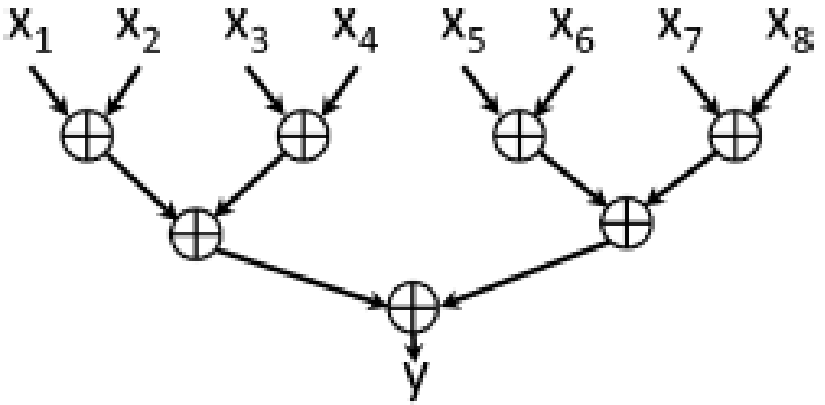
Alpha Beta Cost Model

- Parameters:
  - ***P***: number of processors on the network (nodes)
  - $\boldsymbol{\alpha}$: fixed communication latency – (time to send zero-length msg)
  - $\boldsymbol{\beta}$: time to send one byte
  - $\boldsymbol{\gamma}$: size in bytes of message

- Time to transmit a message:
  - $\mathrm{T} = \alpha + \beta \times \gamma$

Alpha Beta Cost Model

- Reduction analysis with Alpha-beta cost model:
  - E.g. all nodes on same switch
  - E.g. 1 Message with all elements in array (γ =N)
  - Phase (2->1, 4->3,…), phase (3->1, 7->5,…)…
  - Reduction-Time estimates:
    - $\boldsymbol{T_r} \geq \alpha \, log_2(P)$, latency time
    - $\boldsymbol{T_r} \geq log_2(P) \, \gamma \;\; \beta$
  - Pipelining => more messages per node
    but communication overlap.



From: https://arxiv.org/abs/1802.09941

LogP Cost Model

- Cost model that includes **Network Communication cost**
- See: https://dl.acm.org/citation.cfm?doid=155332.155333
- **Assumptions:**
  - one processor per node (no intra-node communication)
  - message size is small
  - communication time (network time + cpu time) dominates execution
  - computation time usually assumed 1 cycle
  - memory access ignored

LogP Cost Model

- Parameters:
  - ***P***: number of processors on the network (nodes)
  - ***L***: ***communication latency*** to send one small message of one word over the network
  - ***o***: ***cpu time*** spent to receive or send a message
  - ***g***: ***minimum time gap*** between two network messages
    - **1/g** = injection rate
    - **word-size/g** = bandwidth

LogP Cost Model (2)

- Example: Broadcast analysis with LogP model
- Optimal tree: **unbalanced**

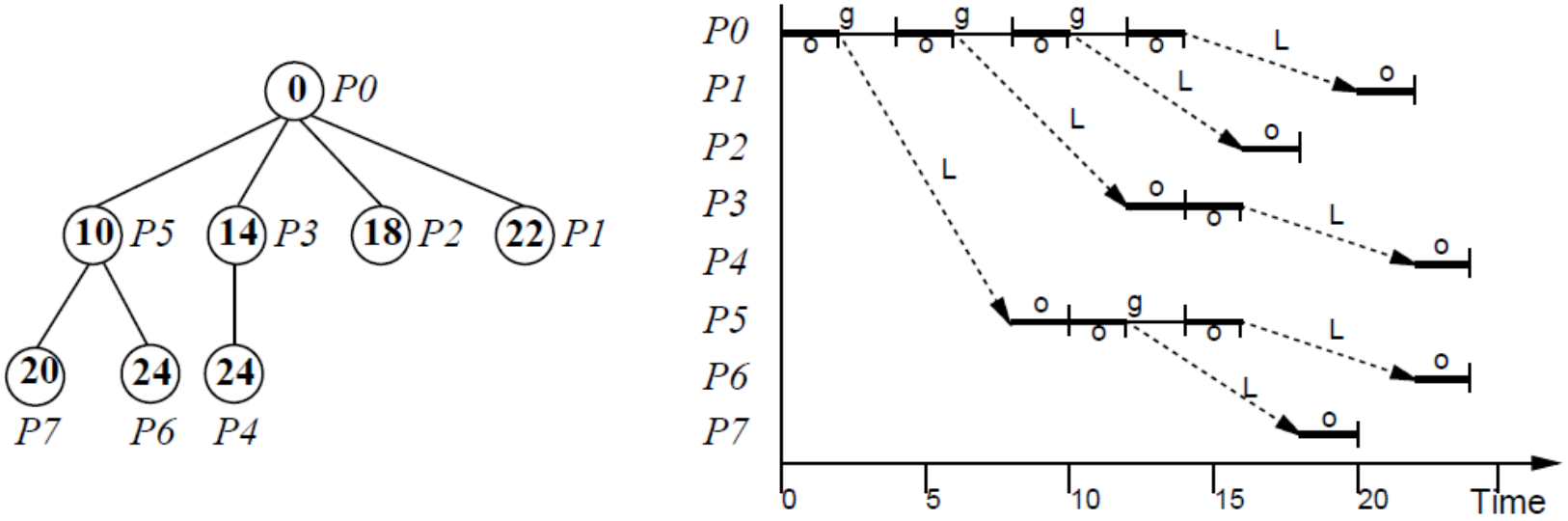


Figure 3: *Optimal broadcast tree for $P = 8, L = 6, g = 4, o = 2$ (left) and the activity of each processor over time (right). The number shown for each node is the time at which it has received the datum and can begin sending it on. The last value is received at time 24.*

From: https://dl.acm.org/citation.cfm?doid=155332.155333

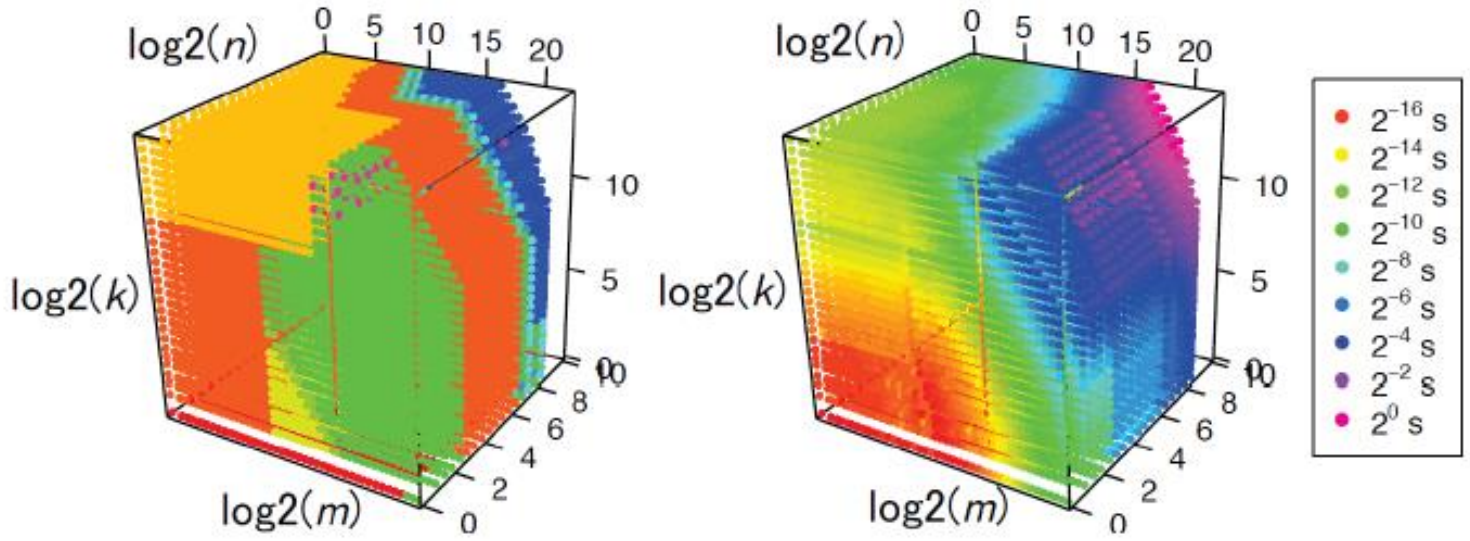
DDL Performance Modeling

- DDL are composed of many parts
- For simplicity we can divide DDL performance modeling in
  - Layers Computation Modeling
    - Fully connected Layers (linear + activation)
    - Convolution Layers
    - Pooling
    - RNN
  - Communication Modeling
    - All-Reduce
    - Parameter server

Layer Computation Performance Modeling

- Layers Performance modeling is hard even with basic work models:
  - Work/Depth
- Example: **cublasSgem performance** for various sizes

Kernel choice



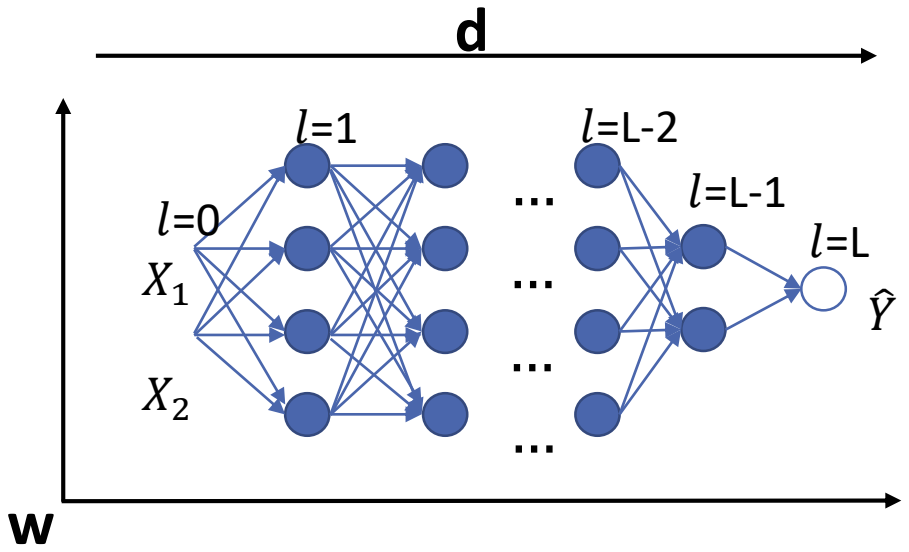
time

Fig. 17. Performance of cublasSgemm on an NVIDIA Tesla K80 for various matrix sizes (adapted from [Oyama et al. 2016])

From:https://arxiv.org/abs/1802.09941

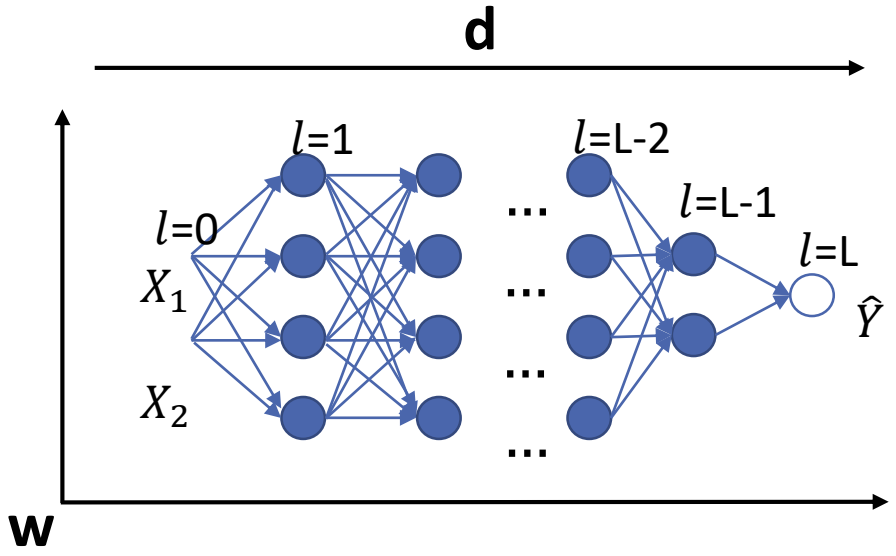
Layer Computation Performance Modeling

- Work and depth application to DL:
  - Work **w** is **proportional** to the size and depth of layers (tensors)
    - Proportional to **samples/batch**
  - Depth **d** is **proportional** to the depth of the network



Layer Computation Performance Modeling (d)

- Average parallelism: **w/d**
  - More features => higher **w** => higher **average parallelism**
  - More layers => higher **w** but also **d =>** same **average parallelism**
  - More layers but same work => higher **w** but also **d =>** less **average parallelism**
  - **Large batchsize => expensive layers fully use GPU!**



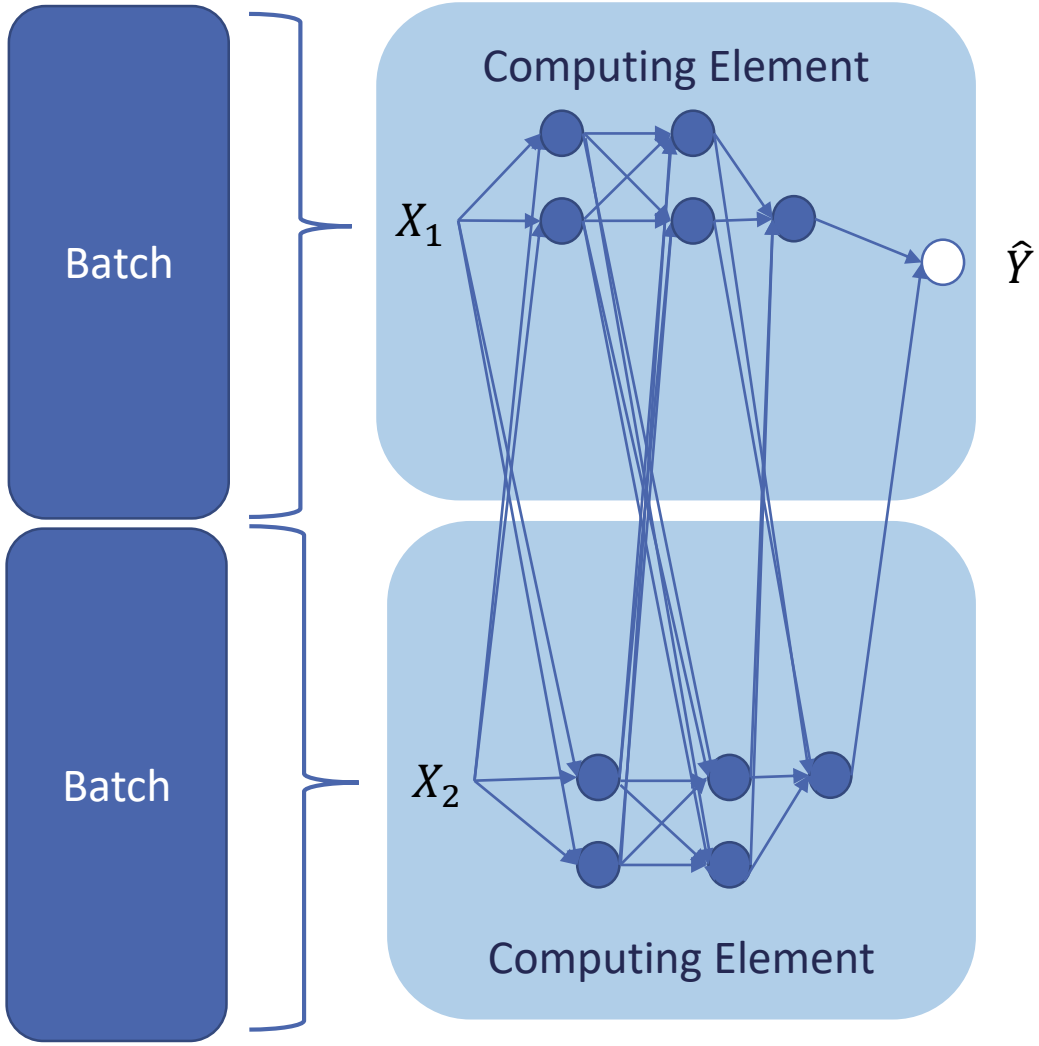
DL Parallelism Approaches

- **Model Parallelism**
  - split model across multiple compute engines
- **Data Parallelism**
  - split data across multiple compute engines
- **Pipelining**
  - use layers as pipeline to keep all compute engines busy
- **Hybrid Parallelism**
  - use multiple approaches together

ML Model Parallelism

- **Model Parallelism:**
  1. Divide the model in parts
  2. Each computing element (CPU core or GPU thread) receives its part of the model
  3. Send same batch to the computing elements
  4. Model merges at the end

- **Characteristics:**
  - **Latency of communication between the computing elements (fine-gran communication) degrades performance**
  - Harder to obtain performance if computing elements are far apart
  - Used among GPU threads or CPU cores/threads

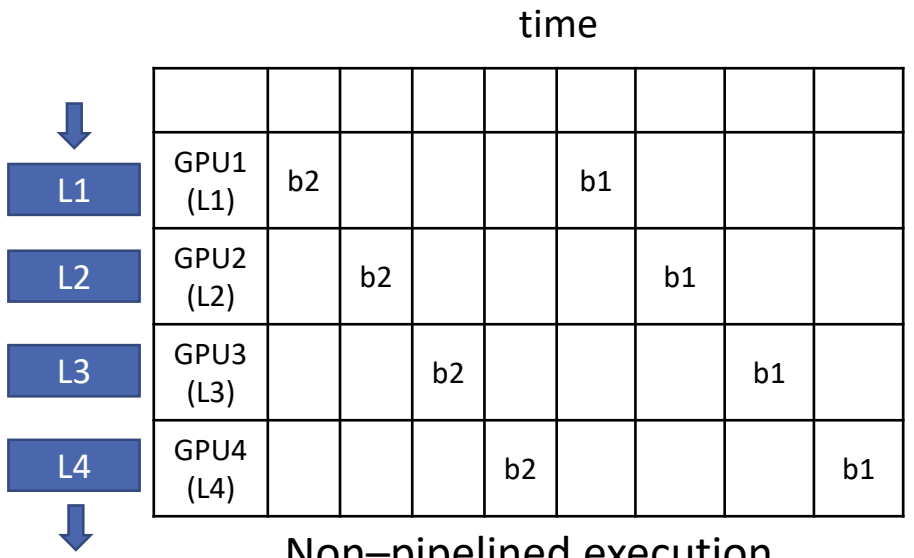


DL Pipelining

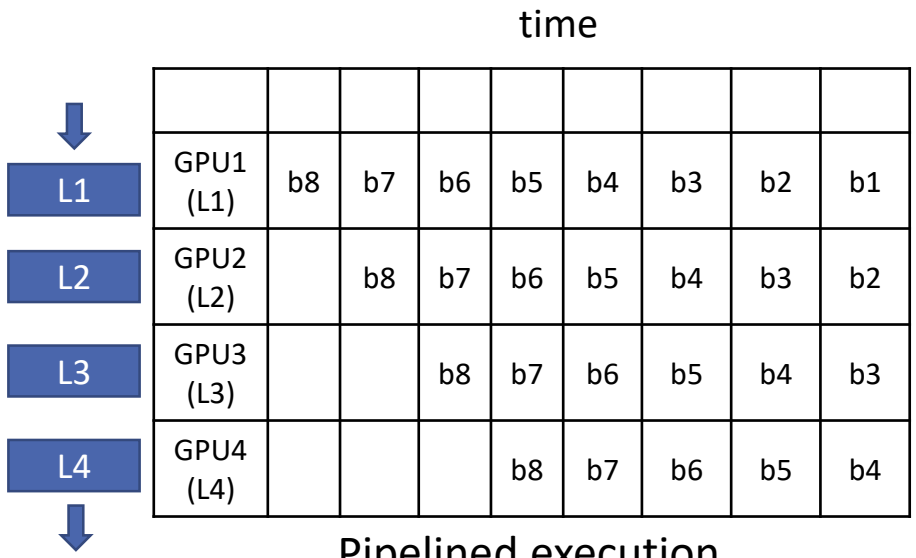
- Pipelining approach:
  - Split layers among compute engines
  - Each minibatch *b (or sample s)* goes from one compute engine to the next one: *no need to wait for next one to exit the pipeline*
- Is a form of **Model Parallelism**
- Pipelining performance
  - Ideal pipelining speedup*:*

$$S = \frac{time\ without\ pipeline}{number\ of\ pipeline\ stages}$$

  - Speedup is higher for deeper networks
  - Ideal pipelining never reached because of "bubbles" that cause idle CPUs
  - SGD pipeline bubble:
    - Before weights update, all batches need to have completed forward (otherwise accept **staleness**)



Non–pipelined execution

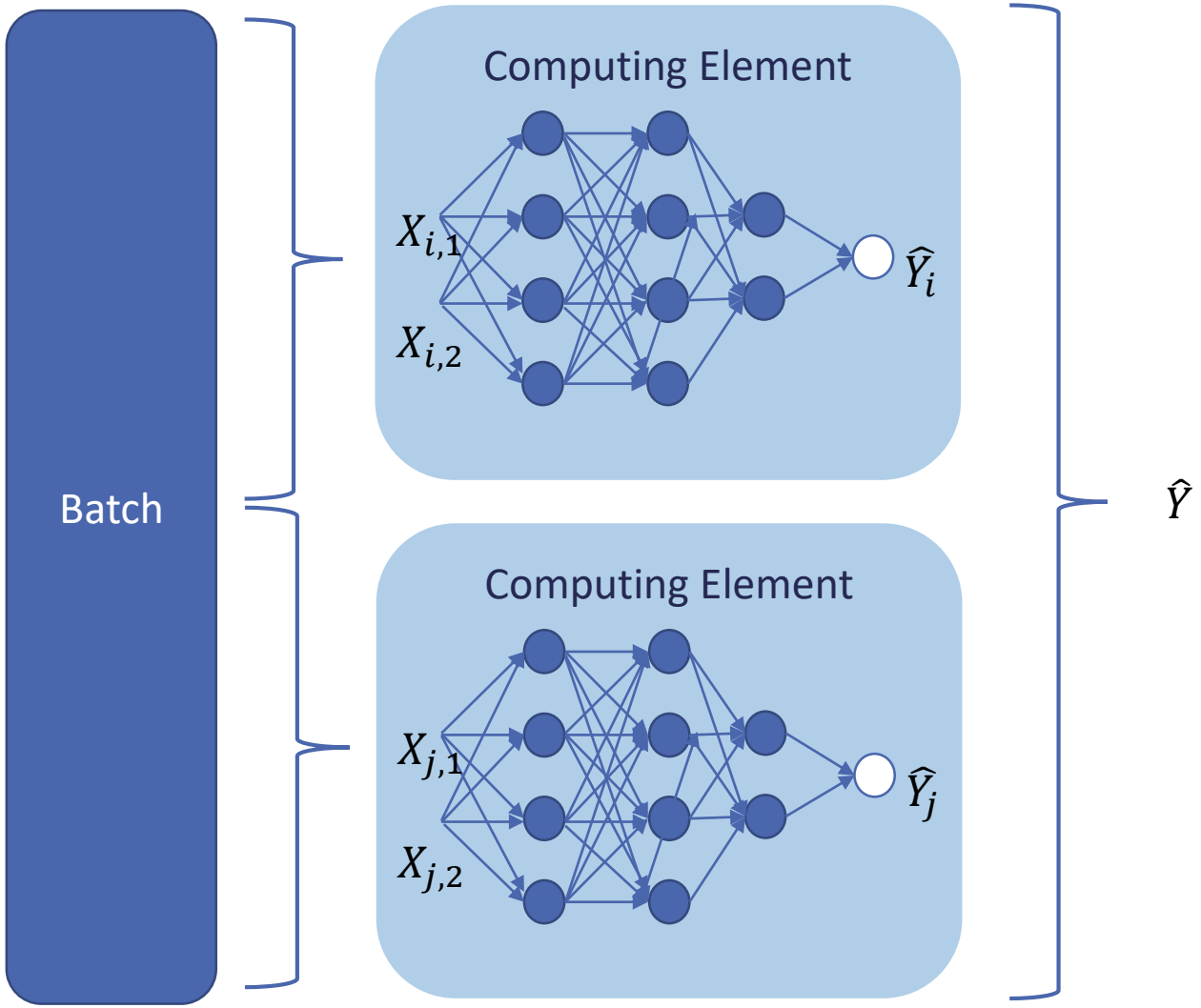


Pipelined execution

ML Data Parallelism

- **Data Parallelism:**
  1. replicate model on each computing element (CPU core, GPU thread, or Cluster node)
  2. Each computing element gets a portion of the batch
  3. Each computing element trains/infers in parallel on the model
  4. Gather the results at the end.

- **Characteristics:**
  - Used among GPUs devices or among cluster nodes
  - PyTorch: DataParallel already implemented
  - Can be also implemented using different batches instead of parts of one batch (typically among nodes in a cluster)



Lesson Key Points

- DL Hardware Trends
  - GPU and Communication trends
- MPI
  - Interface
  - Communication primitives
  - API Semantic
- Performance Models
  - Work Depth model
  - Alpha Beta model
  - LogP model