High Performance Computing
Advanced MPI implementations: Collectives

Jesper Larsson Träff
traff@par. ...
Institute of Computer Engineering, Parallel Computing, 191-4 Favoritenstrasse 16, 3. Stock
Implementing MPI collective operations (MPI 3.1, Chapter 5)

Ideally, MPI library

• implements best known/possible algorithms for given communication network

• gives smooth performance for each operation in problem size, data layout, number of processes, process mapping (given by communicator), ...

• has predictable performance (concrete performance model for concrete library?)

• has consistent performance between related operations
Questions (empirical and theoretical):

• How good are actual MPI libraries?

• What are realistic/possible expectations for a “high quality” MPI library?

• How to judge?
  
  Possible answer: Benchmark with (performance guideline) expectations

There is usually no single, “one size fits all” algorithm. For most collectives, MPI libraries use a mix of different algorithms (depending problem size, numbers of processes, placement in network, …)

But, there are some recurring, common ideas in all these algorithms and implementations
Modeling basic communication performance

1. Between (any) pairs of processors (MPI processes). Are all pairs equal (homogeneous/heterogeneous communication system)?

2. Between all/some pairs of processors (MPI processes). Contention effects in system/network? How to model?

Algorithms for collective operations built from basic, pairwise communication operations. Want model to estimate cost of each collective operation (MPI and in general)
Performance models, definitions, notations:

\begin{align*}
p &: \text{Number of (physical processors) } \approx \text{number of MPI processes in communicator} \\
m &: \text{Total size of message, total size of problem in collective}
\end{align*}

For hierarchical systems (like SMP clusters, more later):
\begin{align*}
N &: \text{Number of nodes} \\
n &: \text{Number of processes per node, } p = nN \text{ (regular cluster)} \\
n_i &: \text{Number of processes at node } N_i, p = \sum N_i n_i
\end{align*}

\textbf{Note}: Here, \( \log p \) denotes base 2 logarithm, \( \log_2 p \)
Linear transmission cost model: Two processes, nothing else

First approximation:
Model physical point-to-point communication time, time to transmit message between any two processors (processes)

\[ t(m) = \alpha + \beta m \]

\( \alpha \): start-up latency (unit: seconds)

\( \beta \): time per unit, inverse bandwidth (unit: seconds/Byte)

Is this so?
First approximation:
Model physical point-to-point communication time, time to transmit message between any two processors (processes)

\[ t(m) = \alpha + \beta m \]

Note:
• Models transfer time, both processors involved in the transfer (synchronous point-to-point)
• Assumes homogeneous network, same transfer time for any pair of processes

Second assumption not realistic: SMP clusters, specific, high-diameter mesh/torus networks, …
Processor pi sending m-unit message to pj
MPI_Send(&x, c, datatype, dest, tag, comm);

Software and algorithmic(*) latency:
• decode arguments (datatype, comm, ...)
• (optional) check arguments
• select algorithm/protocol, initialize
• MPIR_Send(&x, ...); // library internal ADI
• decide fabric, build envelope: communication context, source rank, size, mode info, ... (Note: no datatype)

Software α: 100-10000 instructions

Hardware latency: setup, initiate transfer

(*) can be a decisive factor for collective operations
Typical MPI software/algorithmic latency

- Different “protocols” depending on message size (short, eager, rendezvous)
- Software pipelining
- Handling of structured data (possibly intermediate packing into consecutive buffer)
- Ensuring message integrity (MPI reliability) and order
- Data structures for received and scheduled messages
Linear transmission cost model is sometimes called Hockney-model (misnomer; but common in MPI community)

Linear cost model justified?

Measure the time of transmission $m$ units of data (Bytes) between two processes

Repeat (until result is stable: how?):
1. Synchronize processes (with MPI_Barrier: Beware!)
2. Start time (with MPI_Wtime: Resolution?)
3. Perform communication between processes
4. Stop time
5. Optional: Synchronize
6. Time for operation is time of slowest process
   ($\text{MPI_Allreduce}(\text{MPI\_MAX})$)

Jesper Larsson Träff: mpicroscope: Towards an MPI Benchmark Tool for Performance Guideline Verification. EuroMPI 2012: 100-109
mpicroscope benchmark reports only the best seen completion time (inspired from mpptest), and uses this to determine the number of repetitions (repeat until best time has not changed for some window of iterations). Does not report all measurements (raw data), no statistical analysis.

Jesper Larsson Träff: mpicroscope: Towards an MPI Benchmark Tool for Performance Guideline Verification. EuroMPI 2012: 100-109
Communication patterns (over MPI_COMM_WORLD, or other communicator):

1. Ping:
   Process $i \to$ process $j$, $i$ even, $j=i+1$ (odd), $p$ even

2. PingPing:
   Process $i \leftrightarrow$ process $j$

3. PingPong:
   Process $i \to$ process $j$, process $j \to$ process $i$

**MPI operations:**

->: MPI_Send, MPI_Recv

<-: MPI_Sendrecv

Also interesting/possible:

- MPI_Isend/MPI_Irecv
- MPI_Put/MPI_Get
- ...

Performance differences?
Data:

- Which data sizes (MPI counts)? Beware of only choosing selectively, e.g., powers of 2 (bad experimental practice!)
- Structure of data? MPI_Datatypes?

Slurm, 2 nodes, 1 process/node

srun -N 2 - tasks-per-node=1 ./mpicro -range=0,2000 -lin=50 -tail=1000 -E PingEvenOdd PingPingEvenOdd PingPongEvenOdd -gnuplot -lin -split

50 equidistant message sizes

Experiments

1000 repetitions, always

One plot per experiment

2000 Bytes, as MPI_INT (default)
"Hydra" system at TU Wien

36 nodes with dual-socket Intel Xeon Gold 6130 at 2.1GHz, dual-rail Intel OmniPath network

MPI libraries:
• OpenMPI (version 3.1.3, 4.0.1), compiled with gcc 8.3.0
• mpich 3.3, compiled with gcc 8.3.0
• Intel MPI 2018

Compilation with -O3

Following measurements with OpenMPI

Reproducibility: State all experimental circumstances (context, environment)
Linear cost model on Hydra (mpicroscope benchmark)?

Ping pattern, m=2000Bytes

Not linear

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Ping pattern, \( m=20000\) Bytes

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter

Not linear
Ping pattern, \( m = 200000 \) Bytes

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter

\[ \alpha \approx 2.38 \mu S \]
\[ \beta \approx 7.88 \times 10^{-5} \mu S/\text{Byte} \]

\[ \alpha \approx 5.09 \mu S \]
\[ \beta \approx 1.23 \times 10^{-4} \mu S/\text{Byte} \]

(for \( m \geq 32000 \) Bytes)
Is the network bidirectional?  
Hypothesis: $t(\text{MPI\_Send}) = t(\text{MPI\_Sendrecv})$

PingPing pattern, $m=2000\text{Bytes}$

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Is the network bidirectional?
Hypothesis: \( t(\text{MPI\_Send}) = t(\text{MPI\_Sendrecv}) \)

PingPing pattern, \( m=20000\text{Bytes} \)

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Is the network bidirectional? Compare against Ping pattern

PingPing pattern, m=200000Bytes

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Hypothesis: $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Send})$

PingPong pattern, $m=2000\text{Bytes}$

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Hypothesis: $t(\text{MPI\_Send+MPI\_Recv}) \approx 2t(\text{MPI\_Send})$

PingPong pattern, $m=20000\text{Bytes}$

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Hypothesis: \( t(\text{MPI\_Send+MPI\_Recv}) \approx 2t(\text{MPI\_Send}) \)

Compare against Ping pattern

PingPong pattern, \( m=200000\text{Bytes} \)

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
All process pairs communicate (even-odd), `MPI_COMM_WORLD` vs. cyclic communicator (all messages between nodes)

**MPI_COMM_WORLD**

0 1 ... 

... 

**cyclic comm**

0 2 ... 

1 3 ... 

What is the difference between communication inside shared-memory node (intra) and between (inter)?
All process pairs communicate (even-odd), MPI_COMM_WORLD vs. cyclic communicator (all messages between nodes)

PingPing pattern, m=2000Bytes

MPI_COMM_WORLD, 2x32 processes  cyclic comm, 2x32 processes
PingPing pattern, m=20000Bytes

MPI_COMM_WORLD, 2x32 processes

cyclic comm, 2x32 processes
PingPing pattern, m=200000Bytes

MPI_COMM_WORLD, 2x32 processes

cyclic comm, 2x32 processes
All process pairs communicate (even-odd), MPI_COMM_WORLD vs. cyclic communicator (all messages between nodes)

Performance difference due to limited bandwidth out of compute nodes: All 32 MPI processes on compute node share bandwidth to network (see first lecture)

Note: Slowdown (much) less than a factor of $32/2$: Dual-rail (lane) network in “hydra” cluster

Communication network

One lane to one network

Communication network

Multiple lanes to (multiple) networks
All processes communicate (MPI_Isend, MPI_Irecv) with $k+K$ neighbors.

- $k$ neighbors on the same node
- $K$ neighbors on different nodes

$s$ processes per node

$kKsRing$ pattern

Can MPI_Cart_create, MPI_Dist_graph_create with reorder=1 make sense? Is it beneficial to favor intra-node communication?
**kKsRina pattern, \(m=200000\) Bytes**

- **kKsRings0**
- **kKsRings8**

**MPI_COMM_WORLD**, 36×32 processes, \(k=8, K=0\)

**MPI_COMM_WORLD**, 36×32 processes, \(k=0, K=8\)
All processes communicate (MPI_Isend, MPI_Irecv) with k+K neighbors.

Can MPI_Cart_create, MPI_Dist_graph_create with reorder=1 make sense? Is it beneficial to favor intra-node communication?

Performance difference due to all processes on node sharing the network bandwidth. Slower with K=8 than K=0, but not by a factor 8, rather a factor of 5-6: Dual lane network of “hydra” cluster.
**kKsRina pattern, m=200000Bytes**

**MPI_COMM_WORLD,**
36x32 processes, k=7, K=1

**MPI_COMM_WORLD,**
36x32 processes, k=6, K=2
**kKsRina pattern, \( m=200000 \text{Bytes} \)**

**MPI_COMM_WORLD**, 36x32 processes, \( k=5, K=3 \)  

**MPI_COMM_WORLD**, 36x32 processes, \( k=4, K=4 \)
**kKsRina pattern, m=200000Bytes**

- **kKsRings5**
  - Time (microseconds) vs Datasize (Bytes)
  - Processes: 36x32
  - Parameters: k=3, K=5

- **kKsRings6**
  - Time (microseconds) vs Datasize (Bytes)
  - Processes: 36x32
  - Parameters: k=2, K=6

---

**MPI_COMM_WORLD**

- 36x32 processes, k=3, K=5

- 36x32 processes, k=2, K=6
kKsRina pattern, m=200000Bytes

MPI_COMM_WORLD, 36x32 processes, k=1, K=7

MPI_COMM_WORLD, 36x32 processes, k=0, K=8
Some conclusions:

- Linearity only an approximation, valid at most in certain ranges of message sizes \( m \)
- Modern networks support bidirectional communication
- Raw bandwidth inside and across compute nodes in same ballpark, but cumulated node bandwidth limited (single-rail, multi-rail, number of NIC’s, …)
- Can make sense to have more communication inside compute node (intra) than between (inter)
- MPI communication (MPI_Send, MPI_Recv, …) is not strictly synchronous, the two processes are not both involved during entire transmission
Linear cost model on Jupiter (mpicroscope benchmark)?

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Linear cost model on Jupiter (mpicroscope benchmark)?

Two MPI processes on same SMP node: intra

Two MPI processes on different SMP nodes: inter
Linear cost model on Jupiter (mpicroscope benchmark)?

Larger messages: approx. linear(?)
Linear cost model on Jupiter (mpicroscope benchmark)?

Intra-node
\[ \alpha \approx 1\mu s \]
\[ \beta \approx 0.002\mu s/\text{Byte} \]

Inter-node
\[ \alpha \approx 2\mu s \]
\[ \beta \approx 0.002\mu s/\text{Byte} \]
Refinement:
For non-homogeneous systems, different processor pairs \((i,j)\)
may have different latencies and costs per unit

\[
t_{ij}(m) = \alpha_{ij} + \beta_{ij}m
\]

Cannot model congestion

Piece-wise linear model, short and long messages

\[
t(m) = \begin{cases} 
\alpha_1 + \beta_1 m, & \text{if } 0 \leq m < M_1 \\
\alpha_2 + \beta_2 m, & \text{if } M_1 \leq m < M_2 \\
\alpha_3 + \beta_3 m, & \text{if } M_2 \leq m \\
\end{cases}
\]

Etc.
Regular, hierarchical system can represent $a_{ij}, \beta_{ij}$ matrices more compactly, lookup via suitable tree structure

Regular processor-hierarchy:
Number of subnodes (with same number of nodes) at level $i$ is $n_i$. Communication between processors at level $i$ modeled linearly by $a_i, \beta_i$, system described by the sequence $(n_i, a_i, \beta_i)$, $i=0,...,k$. Total number of processors is $p=\prod n_i$. 
A more detailed model: LogP-family of models (LogGP)

Account for time processor is busy with communication, permit overlapping of communication/computation

L: Latency, time per unit for traveling through network

o: overhead, for processor to start/complete message transfer (both send and receive)

\( g \): gap, between injection (and ejection) of subsequent messages

G: Gap per byte, between injection of subsequent bytes for large messages

P: number of processors (homogeneous communication)
Processor i sending message to processor j

Sending small message at time t, receive at time t+o+L+o

Sender and receiver only involved for o seconds; overlap possible
LogP is not a (synchronous) transmission cost model

Additional network capacity constraint: at most \( \lceil L/g \rceil \) messages can be in transit (if more, sending process stalls)
Processor i sending message to processor j

Sending k small messages from time t, receive at \( t + o + (k-1)g + L + o \)

Next message can be sent after \( g \) time units (assume \( g \geq 0 \))
Processor \(i\) sending message to processor \(j\)

Sending large message at time \(t\), receive at time \(t + o + (m-1)G + L + o\)
Processor i sending message to processor j

Sending k large messages at time $t$, receive at time
$t + o + k(m-1)G + (k-1)g + L + o$
Processor i sending message to processor j

Compared to linear cost model (in which sender and receiver are occupied for the whole transfer, no overlap):

\[ a \approx 2o + L, \quad \beta \approx G \]
Starting point for LogGP, see:


Many variations, some (few) results (optimal tree shapes)


LogP-family vs. linear transmission cost model

- Transmission cost model symmetric, often leads to simple, balanced (in some sense, ..., see later) communication structures (trees), simple, closed form completion time expressions

- LogP-family asymmetric, sending process finishes earlier than receiving process, often leads to skewed structures (trees), often hard to find provably optimal structures, often no closed form completion time expressions
LogP-family vs. linear transmission cost model

• Transmission cost model: $\alpha$, $\beta$ parameters “easy” to measure

• LogP-family: Parameters very difficult to measure
Many, more recent papers in MPI community use some variations of the LogP model (see papers by Hoefler and others)

Recent overview on communication performance models:

Historical on LogP:


These papers (Culler, Patterson) were a major and final cause in terminating the PRAM as a respectable model for parallel computation.

“The LogP model eliminates a variety of loopholes that other models permit. For example, many PRAM algorithms are excessively fine grained, since there is no penalty for interprocessor communication. Although the EREW PRAM penalizes data access contention at the word level, it does not penalize contention at the module level.”

... “It has been suggested that the PRAM can serve as a good model for expressing the logical structure of parallel algorithms, and that implementation of these algorithms can be achieved by general-purpose simulations of the PRAM on distributed-memory machines [26]. However, these simulations require powerful interconnection networks, and, even then, may be unacceptably slow, especially when network bandwidth and processor overhead for sending and receiving messages are properly accounted.”
Network assumptions

A communication network graph $G=(V,E)$ describes the structure of the communication network ("topology")

Processors $i$ and $j$ with $(i,j)$ in $E$ are neighbors and can communicate directly with each other

Linear cost for neighbor communication, all pairs of neighbors have same cost: Network is homogeneous

A processor has $k$, $k \geq 1$, communication ports, and can at any instant be involved in at most $(2)^k$ communication operations (send and/or receive)

- $k=1$: Single-ported communication
Most (but not all: deBruijn, Kautz) networks $G=(V,E)$ are undirected, a communication edge $(i,j)$ allows communication from $i$ to $j$, and from $j$ to $i$

- Diameter of network $G=(V,E)$, $\text{diam}(G)$: Longest path between any two processors $i,j$ in $V$

- Degree of network $G=(V,E)$, $\Delta(G)$: Degree of $i$ with maximum number of (out-going and in-coming) edges

- Bisection width of network $G=(V,E)$, $\text{Bisec}(G)$: Number of edges in a smallest cut $(V_1,V_2)$ with $|V_1| \approx |V_2|$
Single-ported, fully bidirectional, send-receive communication:

Processor $i$ can simultaneously send to processor $j$ and receive from processor $k$, $k=j$ (telephone) or $k \neq j$.

For sparse networks (mesh/torus) $k$-ported, bidirectional (telephone) communication is often possible and assumed (e.g., $k=2d$ for $d$-dimensional torus).
k-ported vs. k-lane communication capabilities

k-ported assumption: A process can simultaneously communicate (e.g. MPI_Sendrecv) with k other processes

For clustered systems, communication system is shared between all processes on compute node

Is the k-ported model realistic?

Communication network

k lanes (rails) shared between processes
k-lane assumption:

Each of k processes on compute node can communicate simultaneously with a process on another node. Possibly each process can also do one (or more) communication operations with processes on the same node.

What can be done under this model? How do good algorithms look? How do they differ from traditional, k-ported algorithms?

Some communication networks (actual, or as design vehicles)

Linear processor array; diameter $p-1$

![Diagram of a linear processor array with nodes 0, 1, 2, and additional nodes]

Linear processor ring; diameter $p-1$, but now strongly connected

![Diagram of a linear processor ring with nodes connected in a ring]

© Jesper Larsson Träff
Fully connected network:
Complete graph, all \((u,v)\) in \(E\) for \(u \neq v\) in \(V\), each processor can communicate directly with any other processor; diameter 1, bisection width \(p^2/4\)
(rxcx...) \(d\)-dimensional mesh/torus: each processor has \(2d\) neighbors, diameter \((r-1)+(c-1)+\ldots\) (roughly, divide by 2 for torus)

Row-order (or some other) numbering

Torus wrap-around edges
Hypercube: \((\log_2 p)\)-dimensional torus, size 2 in each dimension

- **H0**: Processors 0x (binary)
- **H1**: Processors 0x (binary)
- **H2**: Processors 0x (binary) and 1x (binary)
- **H3**: Processors 0x (binary) and 1x (binary)

For \(x\) in \(H(i-1), i>1\)
Hypercube: \((\log_2 p)\)-dimensional torus, size 2 in each dimension

Diameter: \(\log p\)
Degree: \(\log p\), each processor has \(\log p\) neighbors
Naming: \(k\)'th neighbor of processor \(i\): flip \(k\)'th bit of \(i\)

Remark: This is a particular, often convenient naming of the processors in the hypercube
Communication algorithms in networks

Assumption: Independent pairs consisting of a sending and a receiving processor in a network can communicate independently, concurrently, in parallel with all other pairs.

In a k-ported, bidirectional communication system, each processor belongs to at most 2k pairs (as sending or receiving).
Synchronous, round-based algorithm communication complexity:

In each step of the given algorithm, there is a (maximal) set of such processor pairs, in each of which a message of size $m_i$ is transmitted. A step where all these processors communicate is called a **communication round**. The cost of a communication round is $\alpha + \beta \max_{0 \leq i < k} m_i$

Assume all possible processor pairs communicate in each round. The **communication complexity** of the algorithm is the sum of the round costs in the worst case.
Synchronous, round-based algorithm communication complexity:

Alternatively: Communication takes place in synchronized rounds, in each of which as many processor pairs as possible communicate. The complexity is the number (and cost) of such rounds.

Sometimes computation between rounds is not accounted for (too fast; not relevant; ...); unbounded computation assumed...

Algorithm design goals:
• Smallest possible number of communication rounds
• High network utilization
• Balanced communication rounds (all $m_i \approx m_j$)
Asynchronous communication complexity

Processors start at the same time. Communication between two processors can take place when both are ready. Complexity is cost of longest path to a processor finishing last.

Useful for algorithms for irregular collectives; optimization problem is scheduling problem in flavor. Rarely used.

Done after finish time of last incoming $i$, plus $\alpha + \beta m_i$. 

\[ \alpha + \beta m_i \]
For a synthesis with more networks:


- Shuffle-exchange
- deBruijn
- Benes
- Kautz
- Cube-connected cycles
- Tree
- Clos
- ...

© Jesper Larsson Träff
Abstract, network independent (“bridging”) round model: BSP

Parallel computation in synchronized rounds (“supersteps”), processors working on local data, followed by exchange of data (h-relation) and synchronization.

Claim: Any reasonable, real, parallel computer/network can realize (emulate) the BSP model

h-relation:
Data exchange operation in which each processor sends or receives at most h units of data (and at least one processor sends or receives h data units), data visible after synchronization
A BSP\((P,g,l,L)\) computer\((*)\) consists of \(P\) processors with local memory, a router (network+algorithm) that can route arbitrary \(h\)-relations with a cost of \(g\) per data unit, a synchronization algorithm with cost \(l(P)\), and a minimum superstep duration \(L\)

Cost of routing \(h\)-relation: \(gh+l(P)\)

\[(*)\]There are different variants of the BSP computer/model; also other related models (CGM). Note: Parameters have some similarity to LogP
Superstep of BSP algorithm: Local computation, h-relation, synchronization

h-relation (h=2)
BSP algorithm with $S$ supersteps:

Each superstep either computation step or communication step.

- Cost per superstep at least $L$
- Cost of computation step: $W = \max_{0 \leq i < p}(w_i, L)$
- Cost of communication step (h-relation): $H = \max(gh, L)$
- Cost of synchronization after superstep: $l(P)$

Total cost of algorithm: $Sl(P) + \sum S W_s + \sum S H_s$

Implementing BSP computer (library):
- Efficient h-relation (e.g., sparse, irregular MPI alltoall)
- Efficient synchronization

Designing good BSP algorithms:
- Small number of supersteps
- Small h-relations

Lower bounds on communication complexity

Broadcast operation (*MPI_Bcast*): One “root” process has data of size \( m \) to be communicated to all other processes

Assume (for proofs) that communication takes place in synchronized rounds

**Diameter lower bound:**
In a 1-ported network with diameter \( d \), broadcast takes at least \( d \) communication rounds, and time

\[
T_{\text{bcast}}(m) \geq \alpha d + \beta m
\]

**Proof:**
The processor at distance \( d \) from root must receive the data, distance can decrease by at most one in each round. All data must eventually be transferred from root
Fully connected lower bound:
In a fully connected, 1-ported network, broadcast takes at least $\text{ceil}(\log_2 p)$ communication rounds, and time

$$T_{\text{broadcast}}(m) \geq \alpha \cdot \text{ceil}(\log_2 p) + \beta m$$

Proof:
In round 0, only 1 root has data. The number of processors that have (some) data can at most double from one round to the next. By induction, in round $j$, $j=0, 1, \ldots$, the number of processors that have (some) data is at most $2^j$, therefore $\text{ceil}(\log_2 p)$ rounds are required for all processors to have the data.
Fully connected lower bound:
In a fully connected, k-ported network, broadcast takes at least
$\text{ceil}(\log_2 p)$ communication rounds, and time

$$T_{\text{cast}}(m) \geq \alpha \text{ceil}(\log_{k+1} p) + \beta m$$

Proof:
In round 0, only root has data. The number of processors that
have (some) data can at most grow by a factor of k from one
round to the next, by each processor sending on all of its k
ports. By induction, in round $j$, $j=0, 1, \ldots$, the number of
processors that have (some) data is at most $(k+1)^j-1 + k(k+1)^j-1 =
(k+1)^j$, therefore $\text{ceil}(\log_{k+1} p)$ rounds are required for all
processors to have the data.
Multiple message/pipelining lower bound:
The number of communication rounds required to broadcast $M$ blocks of data (in fully connected, 1-ported network) is at least

$$M-1 + \lceil \log_2 p \rceil$$

Proof:
The root can possibly send $M-1$ blocks in $M-1$ rounds; the last block requires at least $\lceil \log_2 p \rceil$ rounds
Observation:
Assume the m data are arbitrarily divisible into M blocks (MPI: could be difficult for structured data described by derived datatypes). The best possible broadcast (also: reduction) time in the linear cost model on fully connected network is

\[ T(m) = (\lceil \log p \rceil - 1)\alpha + 2\sqrt{\lceil \log p \rceil - 1}\alpha\beta m + \beta m \]

Proof: See pipeline lemma (but also next slide)
By lower bound, best time for $M$ blocks is

$$T(m,M) = (M-1+\log p)(\alpha+\beta m/M) = (\log p - 1)\alpha + M\alpha + (\log p - 1)\beta m/M + M\beta m/M = (\log p - 1)\alpha + M\alpha + (\log p - 1)\beta m/M + \beta m$$

Balancing $M\alpha$ and $(\log p-1)\beta m/M$ terms yields

- Best $M$: $\sqrt{[(\log p-1)\beta m/\alpha]}$
- Best blocksize $m/M$: $\sqrt{[\alpha m/(\log p-1)]}$

Question: Can the $(\log p)\alpha+o(m)+\beta m$ bound be achieved? ...

... follow the rest of the lecture!!
Basic lower bounds for collective operations in $\alpha,\beta$-model

$m$ data to be sent or received by any process

- Fully connected network, 1-ported: $\alpha \log_2 p + \beta m$
- Diameter $d$ network, 1-ported: $ad + \beta m$

All $m$ data have to be sent or received, diameter or doubling argument applies to at least some of the data

Can we match these simple lower bounds?
Bisection (band)width lower bound for alltoall communication

Regular alltoall problem: Each process has m individual data to be sent (and received from) any other process

Let G=(V,E) be a bidirectional communication network (with c(e) a capacity of each edge e in E) with (weighted) bisection width Bisec(G). Solving the regular alltoall problem requires at least $\beta |V/2|^2 / \text{Bisec}(G)$ time.

Proof:
Partition V into two subsets V1 and V2 of size $|V/2|$. The time to send and receive all data for all processors in either subset is at least $\beta |V/2|^2$ divided by the capacity of cut(V1,V2). This in particular holds for the cut in Bisec(G)
Before/after semantics of the (MPI) collectives

### Broadcast: before

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\times & & & \\
\end{array}
\]

### Broadcast: after

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\times & \times & \times & \times \\
\end{array}
\]

### Scatter: before

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\times0 & \times1 & \times2 & \times3 \\
\end{array}
\]

### Scatter: after

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\times0 & \times1 & \times2 & \times3 \\
\end{array}
\]

### Gather: before

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\times0 & \times1 & \times2 & \times3 \\
\end{array}
\]

### Gather: after

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\times0 & \times1 & \times2 & \times3 \\
\end{array}
\]
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>x0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td></td>
<td></td>
<td></td>
<td>x3</td>
</tr>
</tbody>
</table>

**Allgather: before**

<table>
<thead>
<tr>
<th></th>
<th>0x0</th>
<th>1x0</th>
<th>2x0</th>
<th>3x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x2</td>
<td>0x3</td>
<td></td>
</tr>
<tr>
<td>1x0</td>
<td>1x1</td>
<td>1x1</td>
<td>1x3</td>
<td></td>
</tr>
<tr>
<td>2x0</td>
<td>2x1</td>
<td>2x2</td>
<td>2x3</td>
<td></td>
</tr>
<tr>
<td>3x0</td>
<td>3x1</td>
<td>3x2</td>
<td>3x3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0x0</th>
<th>1x0</th>
<th>0x2</th>
<th>0x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x0</td>
<td>1x1</td>
<td>1x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x0</td>
<td>2x1</td>
<td>2x2</td>
<td>2x3</td>
<td></td>
</tr>
<tr>
<td>3x0</td>
<td>3x1</td>
<td>3x2</td>
<td>3x3</td>
<td></td>
</tr>
</tbody>
</table>

**Allgather: after**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>x0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td></td>
<td></td>
<td></td>
<td>x3</td>
</tr>
</tbody>
</table>

**Alltoall: before**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td></td>
<td></td>
<td></td>
<td>x3</td>
</tr>
</tbody>
</table>

**Alltoall: after**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x0</td>
<td>1x1</td>
<td>1x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x0</td>
<td>2x1</td>
<td>2x2</td>
<td>2x3</td>
<td></td>
</tr>
<tr>
<td>3x0</td>
<td>3x1</td>
<td>3x2</td>
<td>3x3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td><strong>Reduce: before</strong></td>
<td>(x_0)</td>
<td>(x_1)</td>
<td>(x_2)</td>
<td>(x_3)</td>
</tr>
<tr>
<td><strong>Allreduce: before</strong></td>
<td>(x_0)</td>
<td>(x_1)</td>
<td>(x_2)</td>
<td>(x_3)</td>
</tr>
<tr>
<td><strong>Reducescatter: before</strong></td>
<td>(x_0)</td>
<td>(x_0)</td>
<td>(x_0)</td>
<td>(x_0)</td>
</tr>
<tr>
<td><strong>Scan/exscan: before</strong></td>
<td>(y_0)</td>
<td>(y_1)</td>
<td>(y_2)</td>
<td>(y_3)</td>
</tr>
</tbody>
</table>

**Reduce: after**
\[
\sum x_i
\]

**Allreduce: after**
\[
\sum x_i \quad \sum x_i \quad \sum x_i \quad \sum x_i
\]

**Reducescatter: after**
\[
\sum x_0 \quad \sum x_1 \quad \sum x_2 \quad \sum x_3
\]

**Scan/exscan: after**
\[
y_0 \quad y_1 \quad y_2 \quad y_3
\]

**Scan**
\[
y_i = \sum_{0 \leq j \leq i} x_i
\]

**Exscan**
\[
y_i = \sum_{0 \leq j < i} x_i, \text{ no } y_0
\]
Observations

Gather and Scatter are “dual” operations

\[
\begin{array}{c}
\text{Scatter: before} \\
\begin{array}{c}
 x_0 \\
 x_1 \\
 x_2 \\
 x_3
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{Gather: before} \\
\begin{array}{c}
 x_0 \\
 x_1 \\
 x_2 \\
 x_3
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{Scatter: after} \\
\begin{array}{c}
 x_0 \\
 x_1 \\
 x_2 \\
 x_3
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{Gather: after} \\
\begin{array}{c}
 x_0 \\
 x_1 \\
 x_2 \\
 x_3
\end{array}
\end{array}
\]
Broadcast and Reduce are “dual” operations

**Broadcast: before**

\[ x \]

**Reduce: before**

\[ x_0 \times x_1 \times x_2 \times x_3 \]

**Broadcast: after**

\[ x \times x \times x \times x \]

**Reduce: after**

\[ \sum x_i \]
Alltoall $\approx$ pxp matrix-transpose

<table>
<thead>
<tr>
<th>Alltoall: before</th>
<th>Alltoall: after</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0 1x0 2x0 3x0</td>
<td>0x0 0x1 0x2 0x3</td>
</tr>
<tr>
<td>0x1 1x1 2x1 3x1</td>
<td>1x0 1x1 1x2 1x3</td>
</tr>
<tr>
<td>0x2 1x2 2x2 3x1</td>
<td>2x0 2x1 2x2 2x3</td>
</tr>
<tr>
<td>0x3 1x3 2x3 3x3</td>
<td>3x0 3x1 3x2 3x3</td>
</tr>
</tbody>
</table>
Allgather ≈ Gather + Broadcast

Gather: before
\[
\begin{array}{c}
\times 0 \\
\times 1 \\
\times 2 \\
\times 3 \\
\end{array}
\]

Gather: after
\[
\begin{array}{c}
\times 0 \\
\times 1 \\
\times 2 \\
\times 3 \\
\end{array}
\]

Broadcast: before
\[
\begin{array}{c}
\times \\
\end{array}
\]

Broadcast: after
\[
\begin{array}{c}
\times \\
\times \\
\times \\
\times \\
\times \\
\end{array}
\]

Allgather: before
\[
\begin{array}{c}
\times 0 \\
\times 1 \\
\times 2 \\
\times 3 \\
\end{array}
\]

Allgather: after
\[
\begin{array}{c}
\times 0 \\
\times 1 \\
\times 2 \\
\times 3 \\
\end{array}
\]

\[
\begin{array}{c}
\times 0 \\
\times 1 \\
\times 1 \\
\times 2 \\
\times 2 \\
\times 3 \\
\times 3 \\
\end{array}
\]
Allreduce $\approx$ Reduce + Broadcast

Reduce: before
\[
\begin{array}{cccc}
  x_0 & x_1 & x_2 & x_3 \\
\end{array}
\]

Broadcast: before
\[
\begin{array}{cccc}
  x \\
\end{array}
\]

Allreduce: before
\[
\begin{array}{cccc}
  x_0 & x_1 & x_2 & x_3 \\
\end{array}
\]

Reduce: after
\[
\begin{array}{cccc}
  \sum x_i \\
\end{array}
\]

Broadcast: after
\[
\begin{array}{cccc}
  x & x & x & x \\
\end{array}
\]

Allreduce: after
\[
\begin{array}{cccc}
  \sum x_i & \sum x_i & \sum x_i & \sum x_i \\
\end{array}
\]
Reducescatter \approx \text{Reduce} + \text{Scatter}

<table>
<thead>
<tr>
<th>Reduce: before</th>
<th>Reduce: after</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)  (x_1)  (x_2)  (x_3)</td>
<td>(\sum x_i)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scatter: before</th>
<th>Scatter: after</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)  (x_1)  (x_2)  (x_3)</td>
<td>(x_0)  (x_1)  (x_2)  (x_3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reducescatter: before</th>
<th>Reducescatter: after</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)  (x_0)  (x_0)  (x_0)</td>
<td>(\sum x_0)  (\sum x_1)  (\sum x_2)  (\sum x_3)</td>
</tr>
</tbody>
</table>

© Jesper Larsson Träff
Broadcast $\approx$ Scatter + Allgather

**Scatter: before**

- $x_0$
- $x_1$  
- $x_2$
- $x_3$

**Allgather: before**

- $x_0$
- $x_1$
- $x_2$
- $x_3$

**Broadcast: before**

- $x$

**Scatter: after**

- $x_0$
- $x_1$  

**Allgather: after**

- $x_0$ $x_0$ $x_0$ $x_0$
- $x_1$ $x_1$ $x_1$ $x_1$
- $x_2$ $x_2$ $x_2$ $x_2$
- $x_3$ $x_3$ $x_3$ $x_3$

**Broadcast: after**

- $x$ $x$ $x$ $x$ $x$
Allreduce \approx Reducescatter + Allgather

**Reducescatter: before**
\[
\begin{array}{cccc}
  x_0 & x_0 & x_0 & x_0 \\
  x_1 & x_1 & x_1 & x_1 \\
  x_2 & x_2 & x_2 & x_2 \\
  x_3 & x_3 & x_3 & x_3 \\
\end{array}
\]

**Reducescatter: after**
\[
\begin{array}{cccc}
  \Sigma x_0 & & & \\
  \Sigma x_1 & & & \\
  \Sigma x_2 & & & \\
  \Sigma x_3 & & & \\
\end{array}
\]

**Allgather: before**
\[
\begin{array}{c}
  x_0 \\
  x_1 \\
  x_2 \\
  x_3 \\
\end{array}
\]

**Allgather: after**
\[
\begin{array}{cccc}
  x_0 & x_0 & x_0 & x_0 \\
  x_1 & x_1 & x_1 & x_1 \\
  x_2 & x_2 & x_2 & x_2 \\
  x_3 & x_3 & x_3 & x_3 \\
\end{array}
\]

**Allreduce: before**
\[
\begin{array}{cccc}
  x_0 & x_1 & x_2 & x_3 \\
\end{array}
\]

**Allreduce: after**
\[
\begin{array}{cccc}
  \Sigma x_i & \Sigma x_i & \Sigma x_i & \Sigma x_i \\
\end{array}
\]
Reduce $\approx$ Reducescatter + Gather

<table>
<thead>
<tr>
<th>Reducescatter: before</th>
<th>Reducescatter: after</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0  x0  x0  x0</td>
<td>$\Sigma x0$</td>
</tr>
<tr>
<td>x1  x1  x1  x1</td>
<td>$\Sigma x1$</td>
</tr>
<tr>
<td>x2  x2  x2  x2</td>
<td>$\Sigma x2$</td>
</tr>
<tr>
<td>x3  x3  x3  x3</td>
<td>$\Sigma x3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gather: before</th>
<th>Gather: after</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td></td>
<td>x2</td>
</tr>
<tr>
<td></td>
<td>x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reduce: before</th>
<th>Reduce:after</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0  x1  x2  x3</td>
<td>$\Sigma x_i$</td>
</tr>
</tbody>
</table>
Allgather $\approx \|_{0 \leq i < p} \text{Broadcast}(i)$

Broadcast(0): before
\[
\begin{array}{c}
\bullet \ x_0 \\
\end{array}
\]
Broadcast(0): after
\[
\begin{array}{cccc}
\bullet \ x_0 & \bullet \ x_0 & \bullet \ x_0 & \bullet \ x_0 \\
\end{array}
\]

Broadcast(1): before
\[
\begin{array}{c}
\bullet \ x_1 \\
\end{array}
\]
Broadcast(1): after
\[
\begin{array}{cccc}
\bullet \ x_1 & \bullet \ x_1 & \bullet \ x_2 & \bullet \ x_1 \\
\end{array}
\]

Broadcast(2): before
\[
\begin{array}{c}
\bullet \ x_2 \\
\end{array}
\]
Broadcast(2): after
\[
\begin{array}{cccc}
\bullet \ x_2 & \bullet \ x_2 & \bullet \ x_2 & \bullet \ x_2 \\
\end{array}
\]

Allgather: before
\[
\begin{array}{cccc}
\bullet \ x_0 & \bullet \ x_1 & \bullet \ x_2 & \bullet \ x_3 \\
\end{array}
\]
Allgather: after
\[
\begin{array}{cccc}
\bullet \ x_0 & \bullet \ x_0 & \bullet \ x_0 & \bullet \ x_0 \\
\bullet \ x_1 & \bullet \ x_1 & \bullet \ x_1 & \bullet \ x_1 \\
\bullet \ x_2 & \bullet \ x_2 & \bullet \ x_2 & \bullet \ x_2 \\
\bullet \ x_3 & \bullet \ x_3 & \bullet \ x_3 & \bullet \ x_3 \\
\end{array}
\]
Alltoall \approx \big|_{0 \leq i < p} \text{Scatter}(i)

\begin{align*}
\text{Scatter}(i): \text{before} & \\
& \begin{array}{cccc}
ix0 \\
ix1 \\
ix2 \\
ix3 \\
\end{array} \\
\text{Scatter}(i): \text{after} & \\
& \begin{array}{cccc}
ix0 & ix1 & ix2 & ix3 \\
\end{array}
\end{align*}

\begin{align*}
\text{Alltoall: before} & \\
& \begin{array}{cccc}
0x0 & 1x0 & 2x0 & 3x0 \\
0x1 & 1x1 & 2x1 & 3x1 \\
0x2 & 1x2 & 2x2 & 3x1 \\
0x3 & 1x3 & 2x3 & 3x3 \\
\end{array} \\
\text{Alltoall: after} & \\
& \begin{array}{cccc}
0x0 & 0x1 & 0x2 & 0x3 \\
1x0 & 1x1 & 1x2 & 1x3 \\
2x0 & 2x1 & 2x2 & 2x3 \\
3x0 & 3x1 & 3x2 & 3x3 \\
\end{array}
\end{align*}
MPI collective interfaces (reminder): Regular

**Bcast:**

```c
MPI_Bcast(void *buffer,
          int count, MPI_Datatype datatype,
          int root, MPI_Comm comm)
```

Triple (buffer, count, datatype) describes block of data in broadcast: where, how much, which structure?

Recall MPI rule: Count and datatype may be different on different processes, but the signature of the block must match.
Gather, scatter:

Triples \((sendbuf, sendcount, sendtype)\) and \((recvbuf, recvcount, recvtype)\) describe blocks sent and received. Signatures must match

```c
MPI_Gather(void *sendbuf,
           int sendcount, MPI_Datatype sendtype,
           void *recvbuf,
           int recvcount, MPI_Datatype recvtype,
           int root, MPI_Comm comm)
```

```c
MPI_Scatter(void *sendbuf,
            int sendcount, MPI_Datatype sendtype,
            void *recvbuf,
            int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)
```
Allgather, alltoall:

Check: Sometimes MPI_IN_PLACE can be used to avoid communication from a process to itself

```
MPI_Allgather(void *sendbuf,
              int sendcount, MPI_Datatype sendtype,
              void *recvbuf,
              int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)
```

```
MPI_Alltoall(void *sendbuf,
              int sendcount, MPI_Datatype sendtype,
              void *recvbuf,
              int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)
```
Reduce, allreduce:

\[
\text{MPI\_Reduce} (\text{void} * \text{sendbuf}, \text{void} * \text{recvbuf}, \\
\text{int} \text{count}, \text{MPI\_Datatype} \text{ datatype}, \\
\text{MPI\_Op} \text{ op}, \text{int} \text{root}, \text{MPI\_Comm} \text{ comm})
\]

\[
\text{MPI\_Allreduce} (\text{void} * \text{sendbuf}, \text{void} * \text{recvbuf}, \\
\text{int} \text{count}, \text{MPI\_Datatype} \text{ datatype}, \\
\text{MPI\_Op} \text{ op}, \text{MPI\_Comm} \text{ comm})
\]
Reduce-scatter:

```c
MPI_Reduce_scatter_block(void *sendbuf,
                          void *recvbuf,
                          int count,
                          MPI_Datatype datatype,
                          MPI_Op op, MPI_Comm comm)
```
Scan, exscan:

```c
MPI_Scan(void *sendbuf, void *recvbuf,
        int count, MPI_Datatype datatype,
        MPI_Op op, MPI_Comm comm)
```

```c
MPI_Exscan(void *sendbuf, void *recvbuf,
           int count, MPI_Datatype datatype,
           MPI_Op op, MPI_Comm comm)
```
MPI collective interfaces (reminder): Irregular (v-vector)

Gather:

```c
MPI_Gatherv(void *sendbuf,
            int sendcount, MPI_Datatype sendtype,
            void *recvbuf,
            int recvcounts[], int recvdispls[],
            MPI_Datatype recvtype,
            int root, MPI_Comm comm)
```

4-tuples (recvbuf, recvcounts[i], recvdispls[i], recvtype) describe blocks to be received. Signature of tuple i must match triple sendcount, sendtype of process i

Recall: User responsibility, violation can lead to disaster
Scatter:

\[
\text{MPI\_Scatterv(} \text{void *sendbuf,}
\text{int sendcounts[], int senddispls[],}
\text{MPI\_Datatype sendtype,}
\text{void *recvbuf,}
\text{int recvcount, MPI\_Datatype recvtype,}
\text{int root, MPI\_Comm comm)}
\]

4-tuples \((sendbuf, sendcounts[i], senddispls[i], sendtype)\) describe blocks to be sent. Signature of tuple \(i\) must match \(recvcount\), \(recvtype\) of process \(i\)

Recall: User responsibility, violation can lead to disaster
Allgather:

```c
MPI_Allgatherv(void *sendbuf, int sendcount, 
    MPI_Datatype sendtype, 
    void *recvbuf, 
    int recvcounts[], int recvdispls[], 
    MPI_Datatype recvtype, 
    MPI_Comm comm)
```
Alltoall:

\[ \text{MPI\textunderscore Alltoallv} \text{(void *sendbuf,} \]
\[ \text{int sendcounts[]}, \text{int senddispls[]}, \]
\[ \text{MPI\textunderscore Datatype sendtype,} \]
\[ \text{void *recvbuf,} \]
\[ \text{int recvcounts[]}, \text{int recvdispls[]}, \]
\[ \text{MPI\textunderscore Datatype recvtype,} \]
\[ \text{MPI\textunderscore Comm comm)} \]
Alltoall:

```c
MPI_Alltoallw(void *sendbuf,
               int sendcounts[], int senddispls[],
               MPI_Datatype sendtypes[],
               void *recvbuf,
               int recvcounts[], int recvdispls[],
               MPI_Datatype recvtypes[],
               MPI_Comm comm)
```

Note: Only collective where each block may have individual structure (signatures must match pairwise)
Reduce-scatter:

```
MPI_Reduce_scatter(void *sendbuf, void *recvbuf,
                     int count[],
                     MPI_Datatype datatype,
                     MPI_Op op, MPI_Comm comm)
```
MPI collective interfaces (reminder)

All collectives shown so far are blocking (in the MPI sense)

Since MPI 3.1, non-blocking versions of all collectives

Specification blow-up!

Since MPI 3.1, special, so-called neighborhood collectives for sparse alltoall and allgather type operations, with the same interface signatures(!)

Specification blow-up!

MPI 4 may have persistent collectives. More blow-up!
Questions?

- Why is there an `MPI_Allgather` (∼ `MPI_Gather+MPI_Bcast`) in MPI?
- Why is there an `MPI_Allreduce` (∼ `MPI_Reduce+MPI_Bcast`) in MPI?
- Why is there an `MPI_Reduce_scatter` (∼ `MPI_Reduce+MPI_Scatter`) in MPI?

Answers:
- Convenience, specialized operation possibly more handy for application context
- Better algorithms possible (this lecture)

`MPI library implementer should ensure MPI_Allgather ≤ MPI_Gather+MPI_Bcast; if not, implementation is bad (broken)`
MPI collectives and algorithm design

- Many specific requirements (arbitrary roots, datatypes, mapping in network, ...)

- Any MPI communicator allows (sendrecv) communication between any pair of processes: Virtually fully connected

- Underlying (routing) system (software and hardware) should ensure good (homogeneous) performance between any process pair, under any communication pattern

Approach:
- Implement algorithms with send-rerecv operations, assume fully connected network, use virtual network structure as design vehicle, use actual network for analysis and refinement
MPI is (too) powerful

Even if underlying network (hardware) is known, network specific algorithms may still not be useful

MPI comm (e.g., MPI_COMM_WORLD): Must support all collectives

Torus algorithms
MPI is (too) powerful

Even if underlying network (hardware) is known, network specific algorithms may still not be useful

MPI comm (subcomm): Must support all collectives
Torus/mesh algorithms (possibly different, virtual processor numbering)
MPI is (too) powerful

Even if underlying network (hardware) is known, network specific algorithms may still not be useful

MPI comm (subcomm): Must support all collectives

Torus assumption does not hold
MPI is (too) powerful

Even if underlying network (hardware) is known, network specific algorithms may still not be useful

MPI comm (subcomm): Must support all collectives

Torus assumption does not hold
MPI is (too) powerful

Even if underlying network (hardware) is known, network specific algorithms may still not be useful

Different, disjoint communicators (comm1, comm2) may share network (edges, ports)

Analysis can hardly account for this. No way in MPI for one communicator to know what is happening concurrently in other communicators
MPI (send-receive) communication guarantees:

- Any correct communication algorithm, designed under any network assumption (structure, ports) will work when implemented in MPI (for any communicator)

- Performance depends on how communicator mapping and traffic fits the assumptions of the algorithm

Exploit
- Fully bidirectional, send-receive communication: 
  `MPI_Sendrecv()`
- Multi-ported communication: `MPI_Isend()`/`MPI_Irecv` + `MPI_Waitall()`
MPI collective interfaces (in applications) and algorithms

Algorithm analysis assume that all processors participate at the same time: Communication rounds and cost determined by communication cost model

MPI (applications):
No guarantee, no requirement that processes call collective operation at the same time
Question: How bad can the algorithms be under non-synchronized process arrival (and progress) patterns?

Answer: Not much known...

Research on sensitivity of MPI (collective) operations to
• Non-synchronized process arrival patterns
• “Noise” (OS)
still needed


MPI collective correctness requirement

If some process in `comm` calls collective operation MPI\_<coll>, then eventually all other processes in `comm` must call MPI\_<coll> (with consistent arguments), and no process must call any other collective on `comm` before MPI\_<coll> (assumption: all collective calls before MPI\_<coll> have been completed)

```
Rank i:

MPI_Bcast(...,comm);

MPI_Bcast(...,comm);

MPI_Bcast(...,comm);

... is legal
```
MPI collective correctness requirement

If some process in comm calls collective operation MPI_\(<\text{coll}\)>,
then eventually all other processes in comm must call MPI_\(<\text{coll}\> (with consistent arguments), and no process must call any other
collective on comm before MPI_\(<\text{coll}\> (assumption: all collective
calls before MPI_\(<\text{coll}\> have been completed)

\[
\begin{align*}
\text{Rank i:} & \quad \text{MPI\_Bcast}(\ldots,\text{comm}); \\
& \quad \text{MPI\_Gather(\text{comm});}
\end{align*}
\]

\[
\begin{align*}
\text{... is not} & \quad \text{MPI\_Gather(\text{comm});} \\
& \quad \text{MPI\_Bcast}(\ldots,\text{comm});
\end{align*}
\]
**MPI collective correctness requirement**

If some process in comm calls collective operation MPI_<coll>, then eventually all other processes in comm must call MPI_<coll> (with consistent arguments), and no process must call any other collective on comm before MPI_<coll> (assumption: all collective calls before MPI_<coll> have been completed)

**Rule:**

MPI processes must call collectives on each communicator in the same sequence
The “Van de Geijn” implementations

• Linear-array algorithms for large problems
• Binomial tree ("Minimum Spanning Tree") for small problems
• Heavy use of Broadcast = Scatter + Allgather observation

• Assumes homogeneous, fully-connected network, linear transmission cost model
• 1-ported communication

• Tree algorithms almost always generalize to k-ported communication, number of rounds decrease from \( \log_2 p \) to \( \log_{k+1} p \)

Ignores many MPI specific problems: Buffer placement, datatypes, non-commutativity, …
Mostly concerned with collectives relevant for linear algebra: No scan/exscan, alltoall


See also this interesting, silently highly influential (on MPI and other things), but no longer very well known book

Before starting...

```c
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
```

Get process rank \( i \), and number of processes \( p \). It always holds that \( 0 \leq i < p \)

\( m \) denotes the total problem size (eg., in Bytes)

Blocks for now are stored consecutively in some buffer

![Memory Diagram]
Linear-array scatter

MPI_Scatter(sendbuf, scount, stype, recvbuf, rcount, rtype, root, comm);

scount/rcount: number of elements in one block (m/p), root scatters p-1 blocks to other processes, copies own block
Recall: MPI collectives convention

(buffer-address, count, datatype) triple in MPI collectives specifications always describes one block:

```
MPI_Bcast(buffer, count, datatype, root, comm);
```

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);
```

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, recvdispls, recvtype, root, comm);
```

\( p \) consecutive blocks scattered from root

\( p \) triples: \((\text{recvbuf} + \text{recvdispls}[i], \text{recvcounts}[i], \text{recvtype})\)
for (i=p-1; i>0; i--) {
    MPI_Send(sendbuf[i], ...., root+1, comm);
}

Root (0):
MPI_Recv(recvbuf,...rank-1,...,comm);
for (i=rank; i<p-1; i++) {
    MPI_Sendrecv_replace(recvbuf,...,
    rank-1,rank+1,...,comm);
}
1-ported, bidirectional send-receive communication

Non-root, 0<rank<size:

```c
MPI_Recv(recvbuf,...rank-1,...,comm);
for (i=rank; i<p-1; i++) {
    MPI_Sendrecv_replace(recvbuf,...,rank-1,rank+1,...,comm);
}
```
1-ported, bidirectional send-receive communication

Non-root, 0<rank<size:

\[
\text{MPIRecv}(\text{recvbuf}, \ldots \text{rank-1}, \ldots, \text{comm});
\]

\[
\text{for} \ (i=\text{rank}; \ i<\text{p-1}; \ i++) \ \{
\text{MPISendrecvreplace}(\text{recvbuf}, \ldots, \\
\text{rank-1}, \text{rank+1}, \ldots, \text{comm});
\}
\]
Tscatter(m) = (p-1)(α+βm/p) = (p-1)α + (p-1)/p βm

Optimal in β-term

in linear processor array with 1-ported, bidirectional, send-receive communication
MPI: All processes stores gathered blocks in rank order

\[\text{MPI\_Allgather}(\text{sendbuf, scount, stype, recvbuf, rcount, rtype, comm});\]

scount/rcount: number of elements in one block, each process contributes one block and gathers p-1 blocks
for (i=0; i<p-1; i++) {
    MPI_Sendrecv(recvbuf[(rank-i+size)%size], ..., 
                 recvbuf[(rank-i-1+size)%size], ..., comm);
}
for (i=0; i<p-1; i++) {
    MPI_Sendrecv(recvbuf[(rank-i+size)%size],...,
                recvbuf[(rank-i-1+size)%size],...,comm);
}
for (i=0; i<p-1; i++) {
    MPI_Sendrecv(recvbuf[(rank-i+size)%size], …,
                 recvbuf[(rank-i-1+size)%size], …, comm);
}
for (i=0; i<p-1; i++) {
    MPI_Sendrecv(recvbuf[(rank-i+size)%size], ...

    recvbuf[(rank-i-1+size)%size], ..., comm);
}
\[ \text{Tallgather}(m) = (p-1)(\alpha + \beta m/p) = (p-1)\alpha + (p-1)/p \beta m \]

Non-optimal in \( \alpha \)-term
Optimal in \( \beta \)-term
MPI_Bcast(buffer, count, datatype, root, comm);

\[ T_{\text{broadcast}}(m) = T_{\text{scatter}}(m) + T_{\text{allgather}}(m) = 2(p-1)\alpha + \frac{2(p-1)}{p} \beta m \]

by Broadcast ≈ Scatter + Allgather observation

Factor 2 from optimal in \( \beta \)-term
Linear-ring reduce-scatter

All processes have own m-vector, each receives reduced block of m/p elements

MPI_Reduce_scatter_block(sendbuf, resultbuf, count, datatype, op, comm);
Aside: Requirements for MPI reduction collectives

- op one of \texttt{MPI\_SUM (+), MPI\_PROD (*)}, \texttt{MPI\_BAND}, \texttt{MPI\_LAND}, \texttt{MPI\_MAX}, \texttt{MPI\_MIN}, ...
- Special: \texttt{MPI\_MINLOC, MPI\_MAXLOC}
- Work on vectors of specific basetypes
- User defined operations on any type
- All operations assumed to be associative (Note: Floating point addition etc. is not)
- Built-in operations also commutative

- Reduction in (canonical) rank order
- Result should be same irrespective of process placement (communicator)
- Preferably same order for all vector elements
- sendbuf must not be destroyed (cannot be used as temporary buffer)

“High quality” requirements
Let $X_i$ be the vector contributed by MPI process $i$

Order and bracketing: Chosen bracketing should be same for all vector elements (careful with pipelined or blocked, shared-memory implementations), e.g.,

$$((X_0+X_1)+(X_2+X_3))+((X_4+X_5)+(X_6+X_7))$$

And same, for any communicator of same size (careful with mix of algorithms for hierarchical systems)
for (i=1; i<p; i++) {
    si = (rank-i+size)%size;  
    ri = (rank-i-1+size)%size
    MPI_Sendrecv(recvbuf[si],...,tmp,...,comm);
    recvbuf[ri] = tmp OP recvbuf[ri]; // do MPI op
}
Need for (MPI 2.2 function):

\[
\text{MPI\_Reduce\_local}(\text{in}, \text{inout}, \text{count}, \text{datatype}, \text{op});
\]

\[
\text{recvbuf}[\text{ri}] = \text{tmp} \text{ OP} \text{ recvbuf}[\text{ri}]; \quad \text{// do MPI op}
\]

Not originally in MPI
recvbuf[ri] = tmp OP recvbuf[ri]; // do MPI op

Note: Technically, it was not possible to implement MPI_Reduce_scatter algorithms on top of MPI before MPI 2.2
for (i=1; i<p; i++) {
    si = (rank-i+size)%size; ri = (rank-i-1+size)%size
    MPI_Sendrecv(recvbuf[si], ..., tmp, ..., comm);
    recvbuf[ri] = tmp OP recvbuf[ri]; // do MPI op
}
for (i=1; i<p; i++) {
    si = (rank-i+size)%size;
    ri = (rank-i-1+size)%size;
    MPI_Sendrecv(recvbuf[si], ..., tmp, ..., comm);
    recvbuf[ri] = tmp OP recvbuf[ri]; // do MPI op
}
for (i=1; i<p; i++) {
    si = (rank-i+size)%size; ri = (rank-i-1+size)%size
    MPI_Sendrecv(recvbuf[si],…,tmp,…,comm);
    recvbuf[ri] = tmp OP recvbuf[ri]; // do MPI op
}
\[ \sum_{\text{rank}-i \leq j \leq \text{rank}} x[\text{rank}-1-i] \]

**But:** Exploits commutativity of +

Not suited for all MPI op's/datatypes

**MPI (user-defined) operations may not be commutative**
Recall: MPI assumes that all operations for MPI_Reduce etc. are associative; but floating point operations are not associative, e.g., \((\text{large}+\text{small})+\text{small} \neq \text{large}+(\text{small}+\text{small})\)
Treducescatter (m) = (p-1)(α+βm/p) = (p-1)α + (p-1)/p βm

Non-optimal in α-term

Optimal in β-term

Ignoring time to perform p-1 m/p block reductions
Combining reduce-scatter with allgather/gather immediately gives:

\[
\text{Tallreduce}(m) = 2(p-1)\alpha + 2(p-1)/p \beta m
\]

\[
\text{Treduce}(m) = 2(p-1)\alpha + 2(p-1)/p \beta m
\]

Factor 2 from optimal in \(\beta\)-term
The power of pipelining

```
MPI_Bcast(buffer, count, datatype, root, comm);
```

Assume $m$ large, $m > p$

Assume $m$ can be arbitrarily divided into smaller, roughly equal sized blocks

Chose $M$, number of rounds, send blocks of size $m/M$ one after the other

**MPI technicality:** If `datatype` (structure of data element in buffer of count elements) describes a large element, dividing into blocks of $m/M$ units require special capabilities from MPI library implementation. *Currently insufficient MPI functionality*
Pipelined broadcast

for (b=0; b<M; b++) {
    MPI_Send(buffer[b],...,rank+1,...,comm);
}

Root (0):
Non-root, rank<size-1>:

```c
MPI_Recv(buffer[0], ..., comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1], ..., buffer[b], ..., comm);
}
MPI_Send(buffer[M-1], ..., comm);
```
Non-root, rank < size - 1:

```c
MPI_Recv(buffer[0], ..., comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1], ..., buffer[b], ..., comm);
}
MPI_Send(buffer[M-1], ..., comm);
```
Non-root, rank<size-1>:

```c
MPI_Recv(buffer[0],...,comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1],...,buffer[b],...,comm);
}
MPI_Send(buffer[M-1],...,comm);
```
Non-root, rank < size - 1:

```c
MPI_Recv(buffer[0], ..., comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1], ..., buffer[b], ..., comm);
}
MPI_Send(buffer[M-1], ..., comm);
```
Non-root, rank<size-1>:

```c
MPI_Recv(buffer[0], ..., comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1], ..., buffer[b], ..., comm);
}
MPI_Send(buffer[M-1], ..., comm);
```
Non-root, rank<size-1:  

```c
MPI_Recv(buffer[0],...,comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1],...,buffer[b],...,comm);
}
MPI_Send(buffer[M-1],...,comm);
```
Non-root, rank<size-1>:

```c
MPI_Recv(buffer[0],...,comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1],...,buffer[b],...,comm);
}
MPI_Send(buffer[M-1],...,comm);
```
```c
for (b=0; j<M; j++) {
    MPI_Recv(buffer[b], ..., comm);
}
Non-root, rank=size-1:
```
• Last processor receives first block after p-1 rounds
• Last processor completes after p-1+M-1 rounds
• Processor i receives first block after i rounds, and a new block in every round
• Root completes after M rounds
Observation:

\[ T_{broadcast}(m) = (p-1+M-1)(a+\beta m/M) \]

A best possible number of blocks, and a best possible block size can easily be found: Pipelining lemma

Similar linear pipelined algorithms for reduction, scan/exscan

MPI_Reduce needs special care for root≠0 and root≠p-1
Pipelining lemma:
With latency of $k$ rounds to deliver the first block, and a new block every $s$ rounds, the best possible time (in linear cost model) for a pipelined algorithm that divides $m$ into blocks is

$$\text{Proof:}$$

Pipelining with $M$ blocks takes

$$(k+s(M-1))(a+\beta m/M) = (k-s)a + sM\alpha + (k-s)\beta m/M + s\beta m$$

rounds. Balancing the $sM\alpha$ and $(k-s)\beta m/M$ terms gives best $M$ of

$$\sqrt{[(k-s)\beta m/s\alpha]}$$

Substitution yields the claim

$$k-s)a + 2\sqrt{s(k-s)a\beta m} + s\beta m$$
**Corollary:**
Best possible time for linear pipeline broadcast is

\[(p-2)\alpha + 2\sqrt{(p-2)\alpha \beta m} + \beta m\]

since \(k=p-1\) and \(s=1\)

**Practical relevance:** Extremely simple, good when \(m\gg p\)
MPI difficulty with pipelined algorithms: Structured buffers

Buffer is \textit{count} repetitions of \textit{datatype}

MPI library needs internal functionality to access parts of structured buffers. MPI specification does not expose any such functionality.