

# High Performance Computing

## Advanced MPI implementations: Collectives

Jesper Larsson Träff

[traff@par. ...](mailto:traff@par. ....)

Institute of Computer Engineering, Parallel Computing, 191-4  
Treitlstrasse 3, 5. Stock (DG)

## 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:

$p$ : Number of (physical processors)  $\approx$  number of MPI processes in communicator

$m$ : Total size of message, total size of problem in collective

For hierarchical systems (like SMP clusters, **more later**):

$N$ : Number of nodes

$n$ : Number of processes per node,  $p = nN$  ( **regular cluster** )

$n_i$ : Number of processes at node  $N_i$ ,  $p = \sum N_i n_i$

**Note** : Here,  $\log p$  denotes base 2 logarithm,  $\log_2 p$

## Linear transmission cost model: Two processes, nothing else

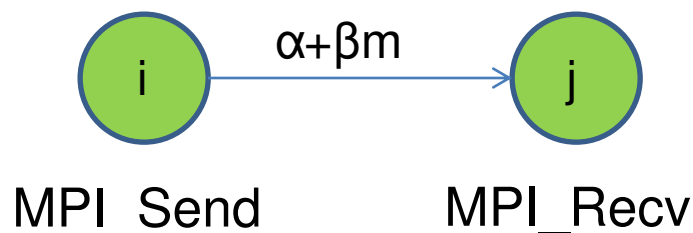
### First approximation :

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)

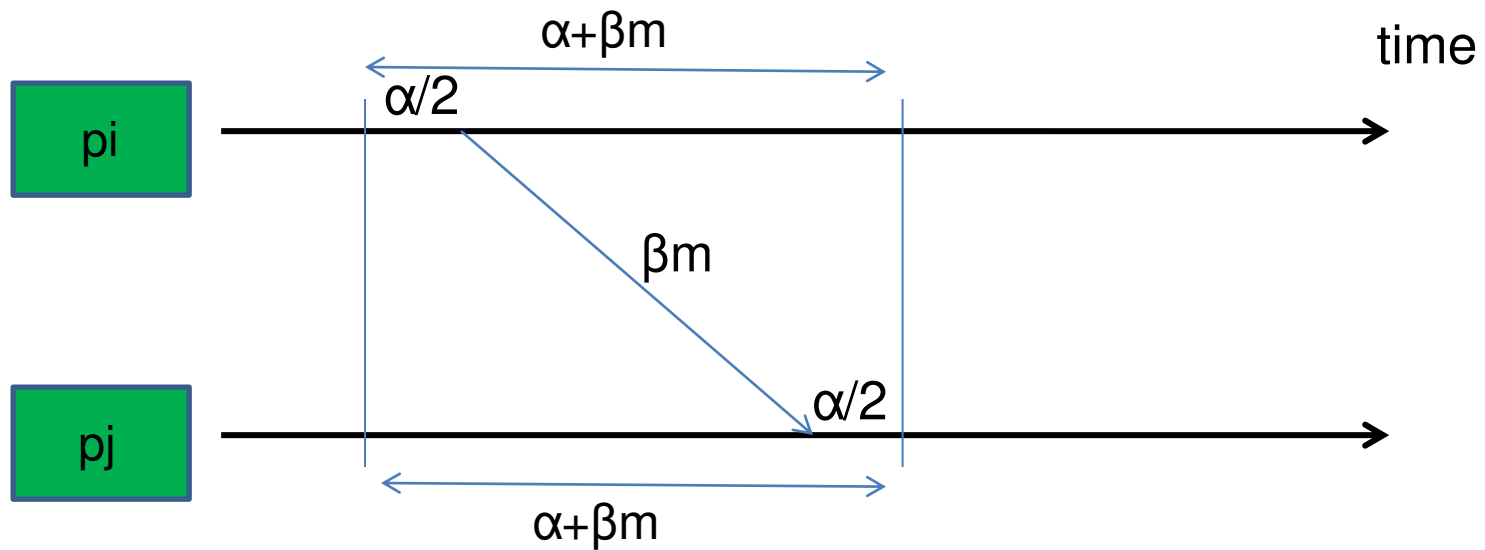
$$t(m) = \alpha + \beta m$$

### Note :

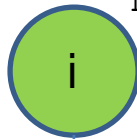
- Models **transfer time** , both processors involved in the transfer (synchronous point-to-point)
- Assumes **homogeneous network**, same transfer time for any pair of processes

Second assumption **not realistic** : SMP clusters, specific, high-diameter mesh/torus networks, ...

Processor  $p_i$  sending  $m$ -unit message to  $p_j$



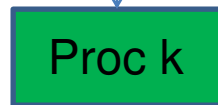
```
MPI_Send(&x, c, datatype, dest, tag, comm);
```



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  $\alpha$ : 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)

Roger W. Hockney: Parametrization of computer performance.

Parallel Computing 5(1-2): 97-103 (1987)

Roger W. Hockney: Performance parameters and benchmarking of supercomputers. Parallel Computing 17(10-11): 1111-1130 (1991)

Roger W. Hockney: The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. Parallel Computing 20(3): 389-398 (1994)

## 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  
(`MPI_Allreduce(MPI_MAX)`)

Jesper Larsson Träff: mpicroscope: Towards an MPI Benchmark Tool for Performance Guideline Verification. EuroMPI 2012: 100-109



## Experiment design, motivation

Experimental computer science is hypothesis driven: What are our expectations, how can these be corroborated or disproved?

Parallel computing:

- User “pays” for total time the system is used...
- We focus on total completion time, time from start of an algorithm, until the system is again free, that is, slowest process has finished
- Also for individual operations

Therefore: start all processes “at the same time”, wait for completion of slowest

Critique: Algorithms and (collective) operations are used in context, where processes are not starting at the same time. Other definitions of time/properties may be more relevant

## Experimental factors(I)

- Time is measured locally by the processes: Are the local clocks accurate? Synchronized? Drifting?
- Is the MPI\_Barrier operation synchronizing in time, do all processes “start at the same” time”?

Answers:

- Partly, no (or sometimes), yes (often)
- No (MPI standard has no performance model), nevertheless often good enough

Remedies:

Compensate, repeated measurements

Barrier algorithms design for temporal accuracy (“at the same time”)

`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

Rethink!

Jesper Larsson Träff: `mpicroscope`: Towards an MPI Benchmark Tool for Performance Guideline Verification. EuroMPI 2012: 100-109

William Gropp, Ewing L. Lusk: Reproducible Measurements of MPI Performance Characteristics. PVM/MPI 1999: 11-18

Communication patterns (over MPI\_COMM\_WORLD, or other communicator):

1. Ping:

Process  $i \rightarrow$  process  $j$ ,  $i$  even,  $j=i+1$  (odd),  $p$  even

2. PingPing:

Process  $i \leftrightarrow$  process  $j$

3. PingPong:

Process  $i \rightarrow$  process  $j$ , process  $j \rightarrow$  process  $i$

MPI operations:

$\rightarrow$ : MPI\_Send, MPI\_Recv

$\leftrightarrow$ : MPI\_Sendrecv

Also interesting/possible:

- MPI\_Isend/MPI\_Irecv
- MPI\_Put/MPI\_Get
- ...

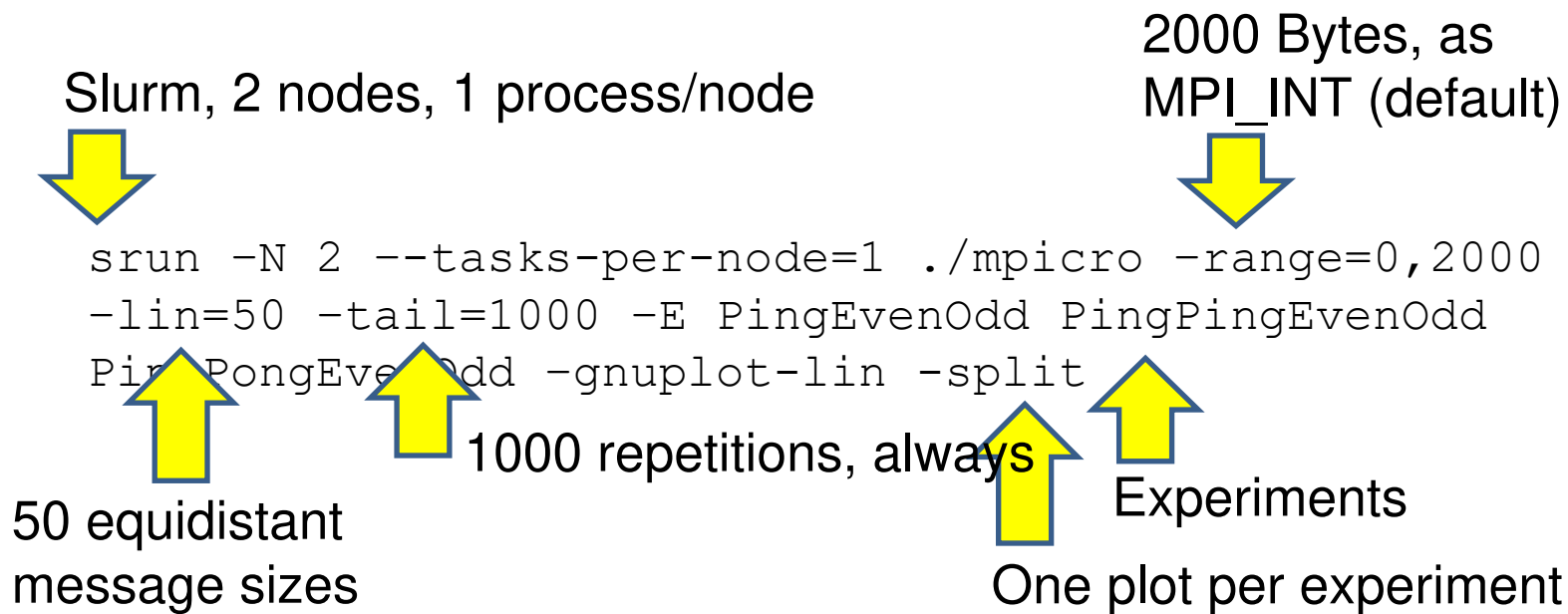
Performance differences?

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

2000 Bytes, as  
MPI\_INT (default)



```
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

1000 repetitions, always

Experiments

One plot per experiment

## “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



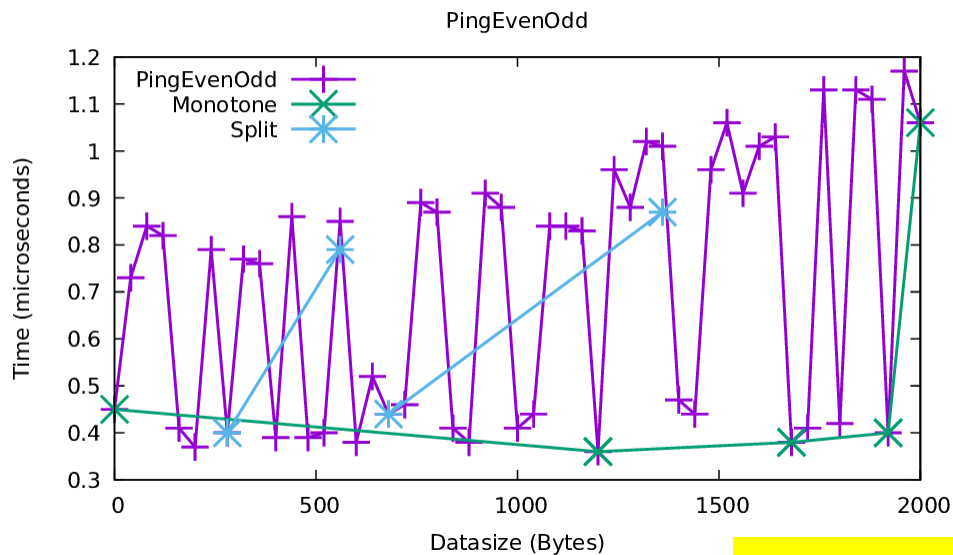
Following measurements  
with OpenMPI

Compilation with `-O3`

Reproducibility: State all experimental circumstances (context, environment)

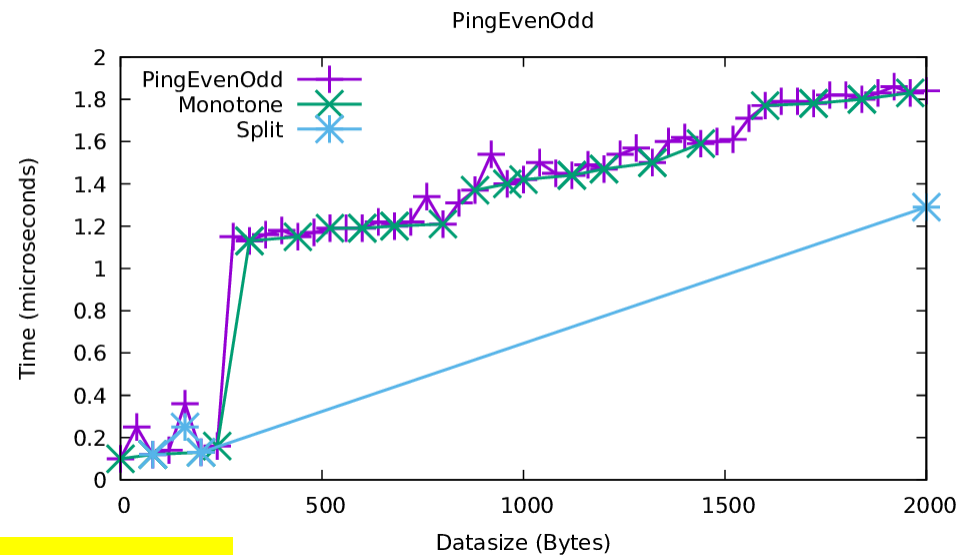
## Linear cost model on Hydra (mpicroscope benchmark)?

### Ping pattern, $m=2.000\text{Bytes}$



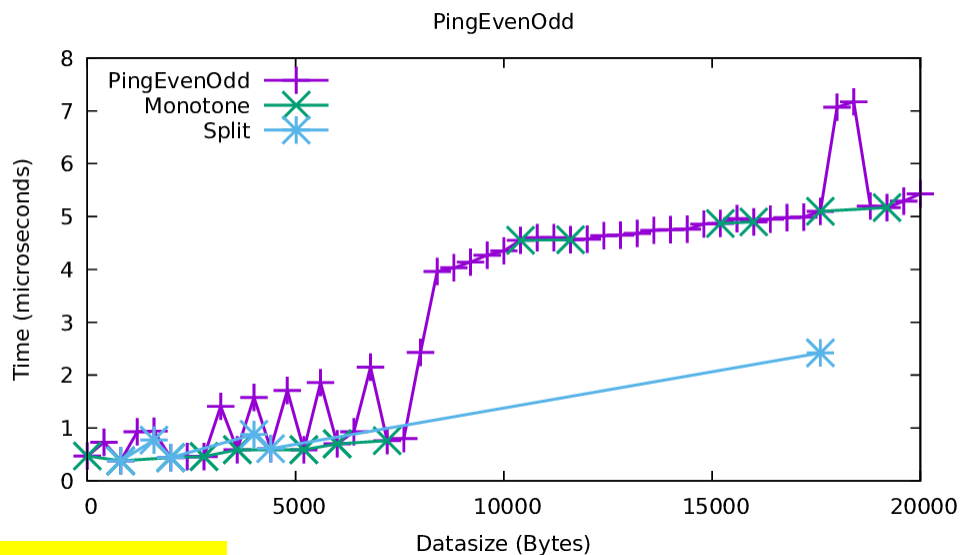
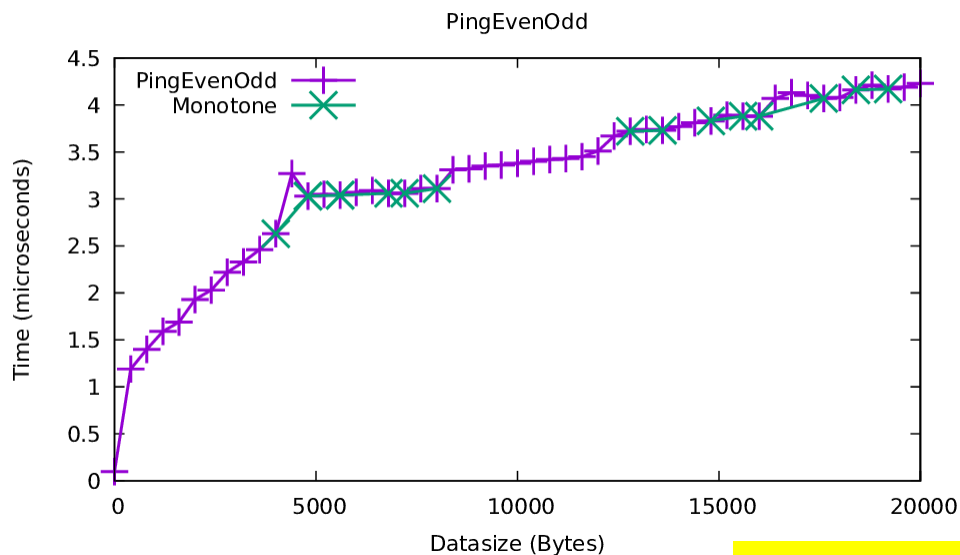
Not linear

Two MPI processes on same  
SMP node: intra



Two MPI processes on  
different SMP nodes: inter

## Ping pattern, m=20.000Bytes



Not linear

Two MPI processes on same  
SMP node: intra

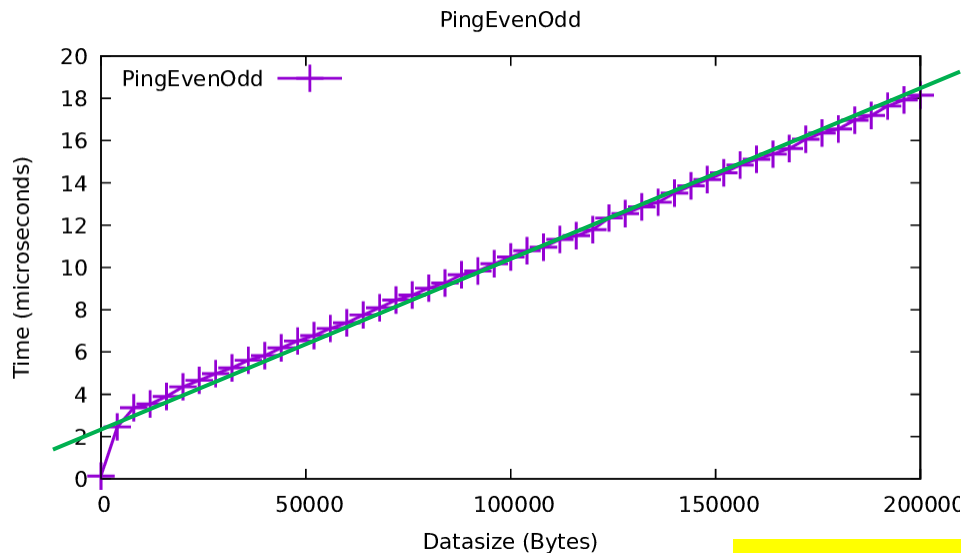
Two MPI processes on  
different SMP nodes: inter



$$\alpha \approx 2.38\mu\text{S}$$

$$\beta \approx 7.88 \cdot 10^{-5} \mu\text{S/Byte}$$

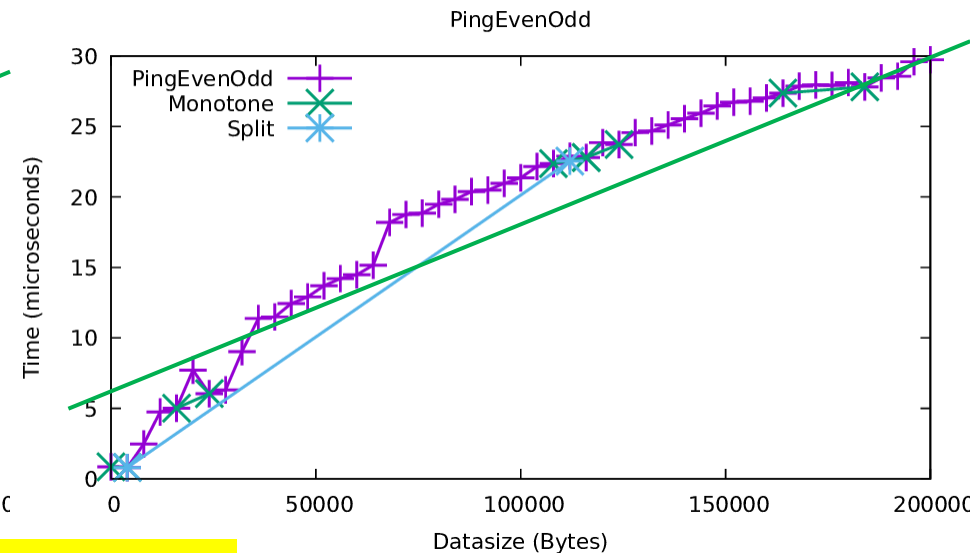
Ping pattern,  $m=200.000\text{Bytes}$



$$\alpha \approx 5.09\mu\text{S}$$

$$\beta \approx 1.23 \cdot 10^{-4} \mu\text{S/Byte}$$

(for  $m \geq 32000\text{Bytes}$ )

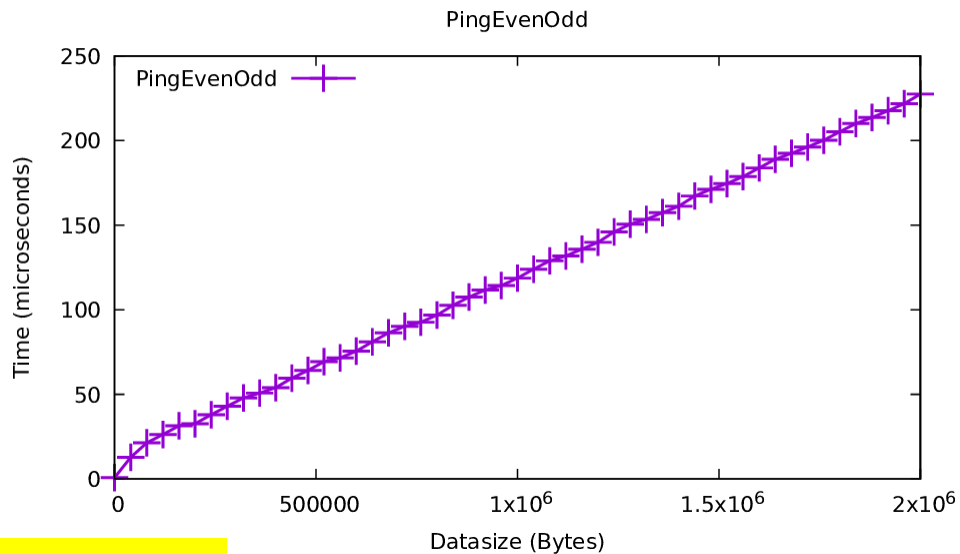
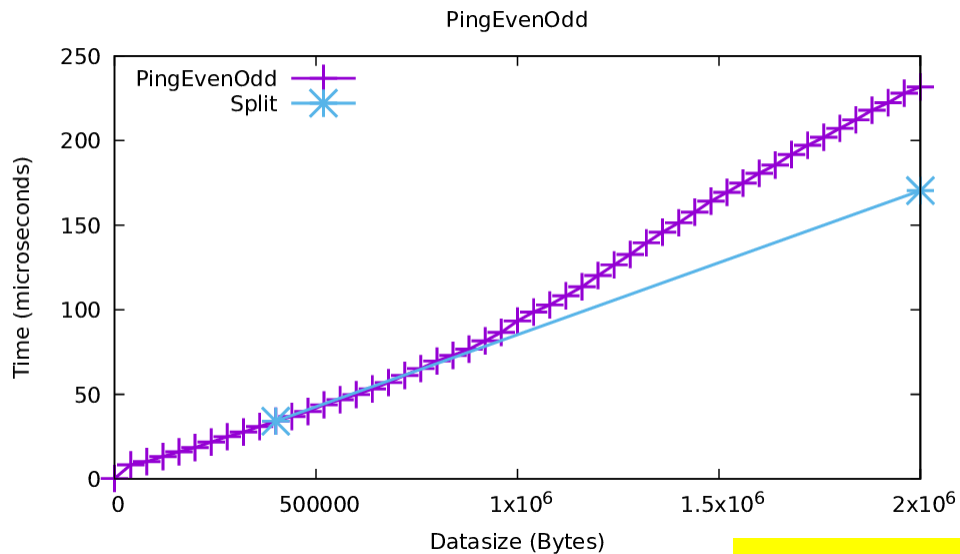


Linear?

Two MPI processes on same  
SMP node: intra

Two MPI processes on  
different SMP nodes: inter

## Ping pattern, $m=2000.000\text{Bytes}$



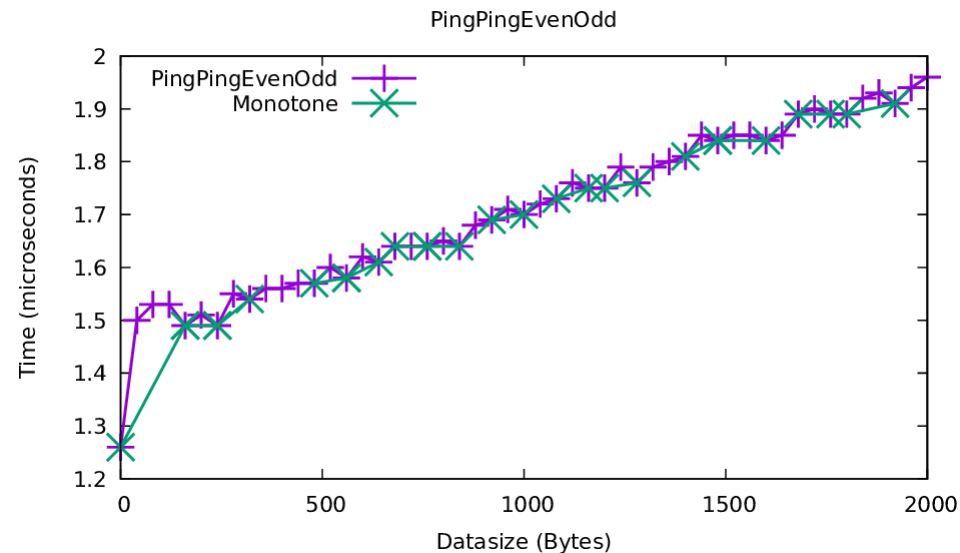
Linear?

Two MPI processes on same  
SMP node: intra

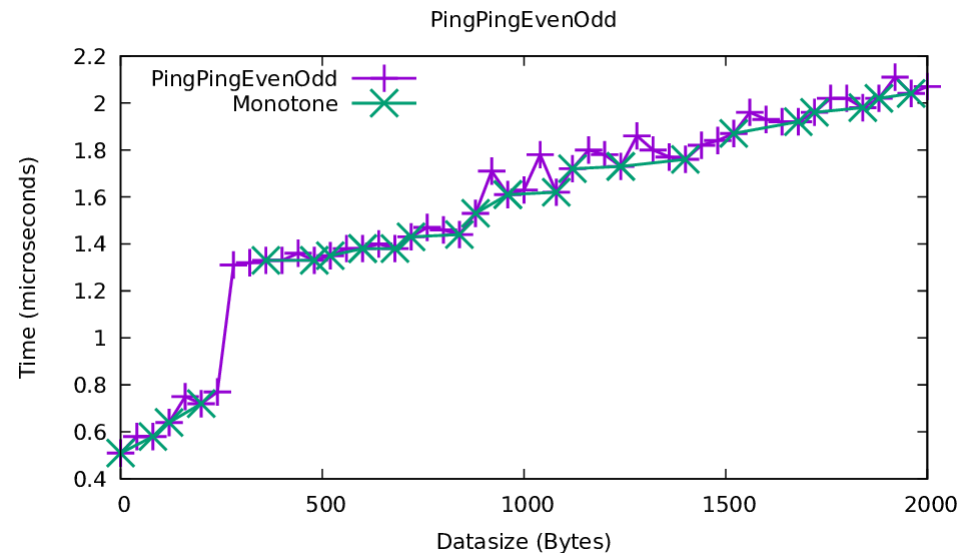
Two MPI processes on  
different SMP nodes: inter

Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

PingPing pattern,  $m=2.000\text{Bytes}$



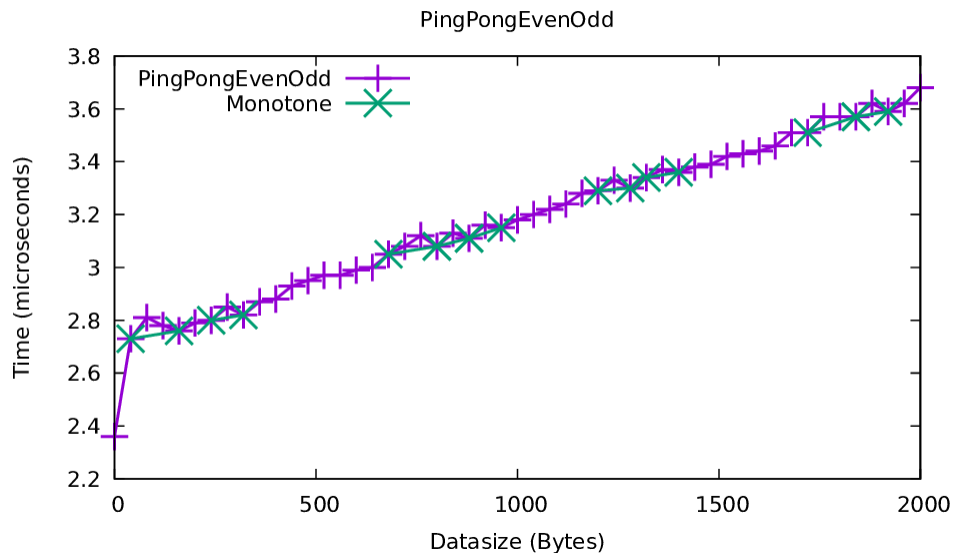
Two MPI processes on same  
SMP node: intra



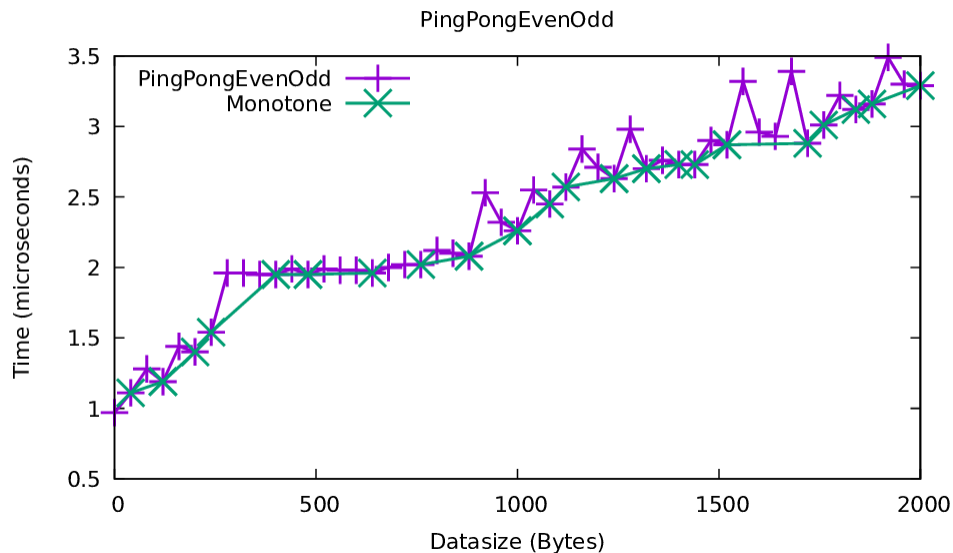
Two MPI processes on  
different SMP nodes: inter

Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

**PingPong** pattern,  $m=2.000\text{Bytes}$



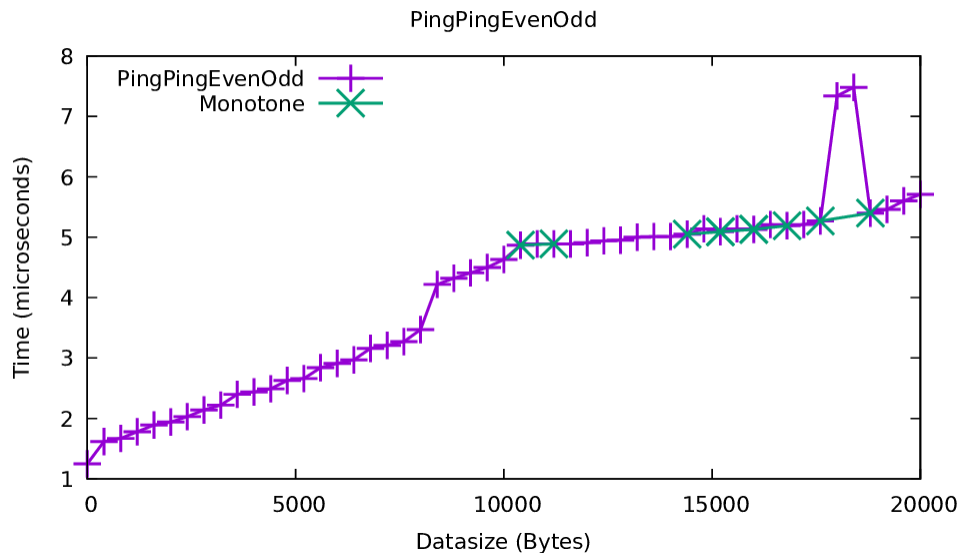
Two MPI processes on same  
SMP node: intra



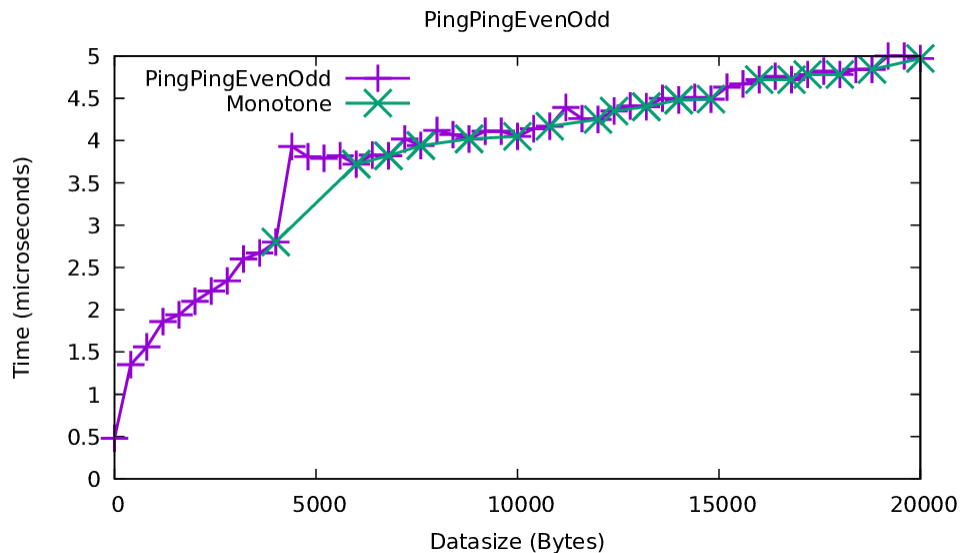
Two MPI processes on  
different SMP nodes: inter

Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

PingPing pattern,  $m=20.000\text{Bytes}$



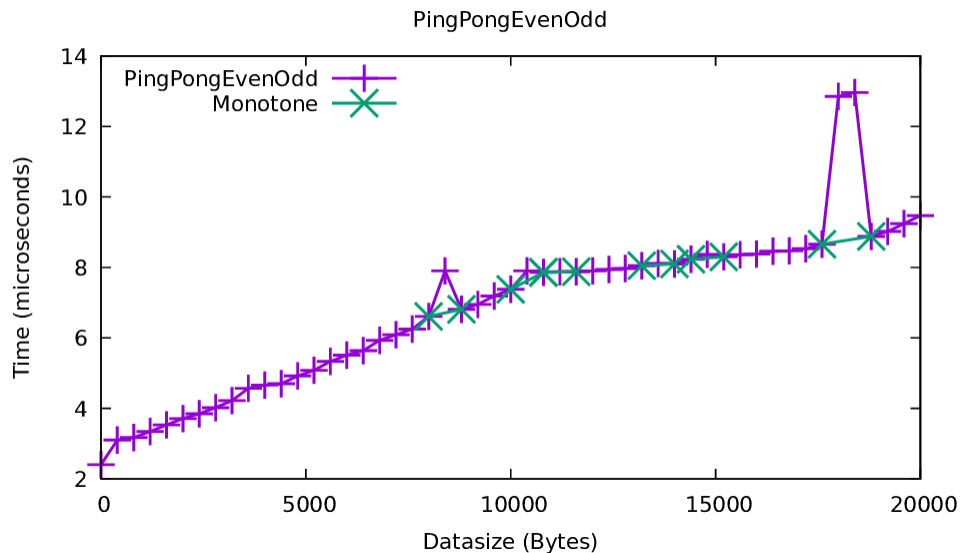
Two MPI processes on same  
SMP node: intra



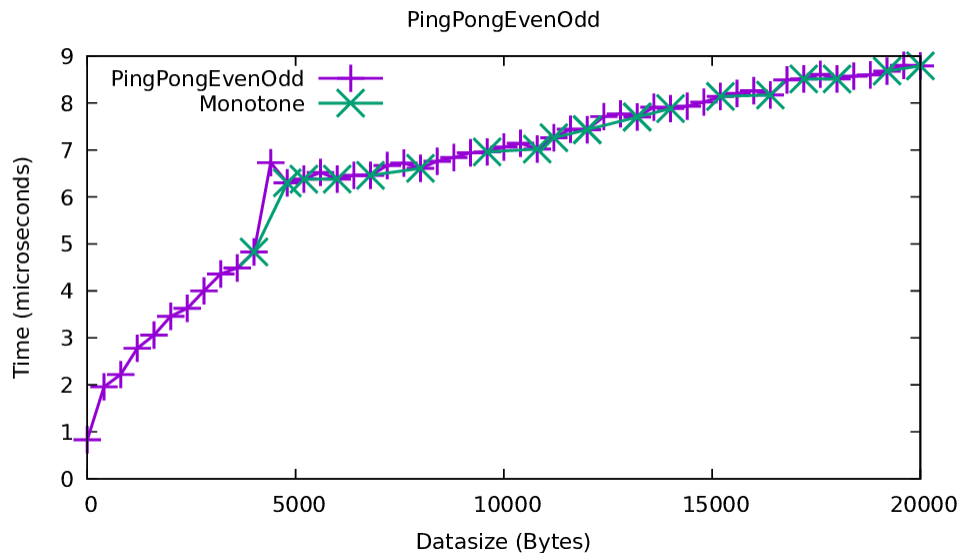
Two MPI processes on  
different SMP nodes: inter

Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

**PingPong** pattern,  $m=20.000\text{Bytes}$



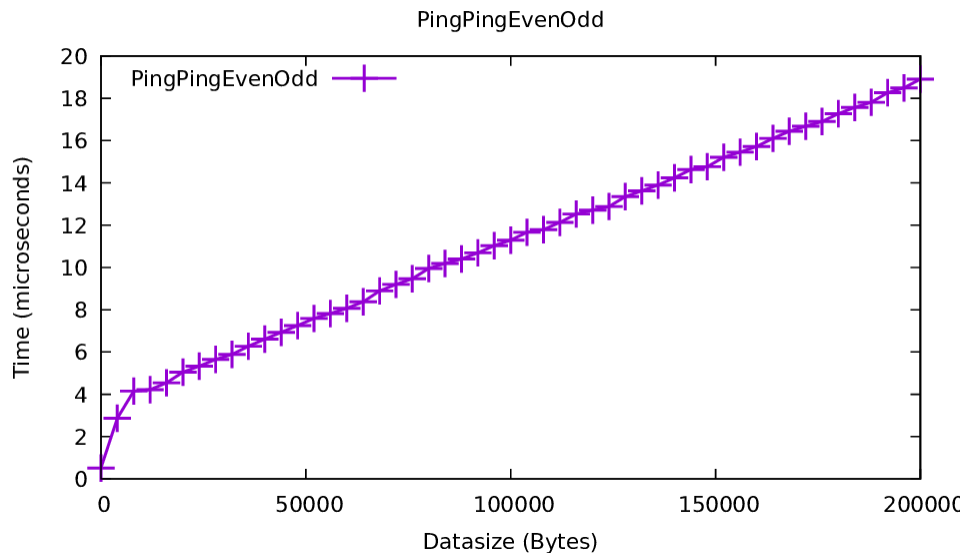
Two MPI processes on same  
SMP node: intra



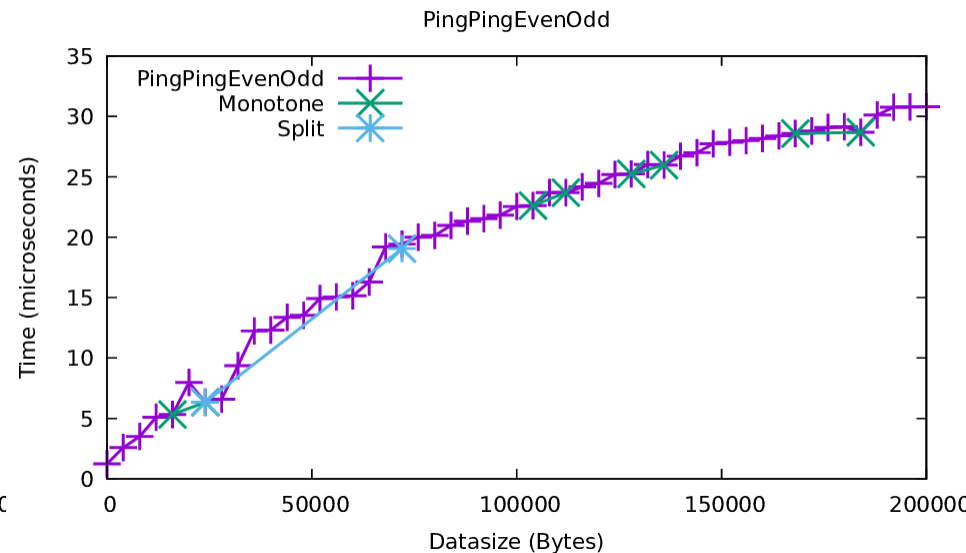
Two MPI processes on  
different SMP nodes: inter

Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

PingPing pattern,  $m=200.000\text{Bytes}$



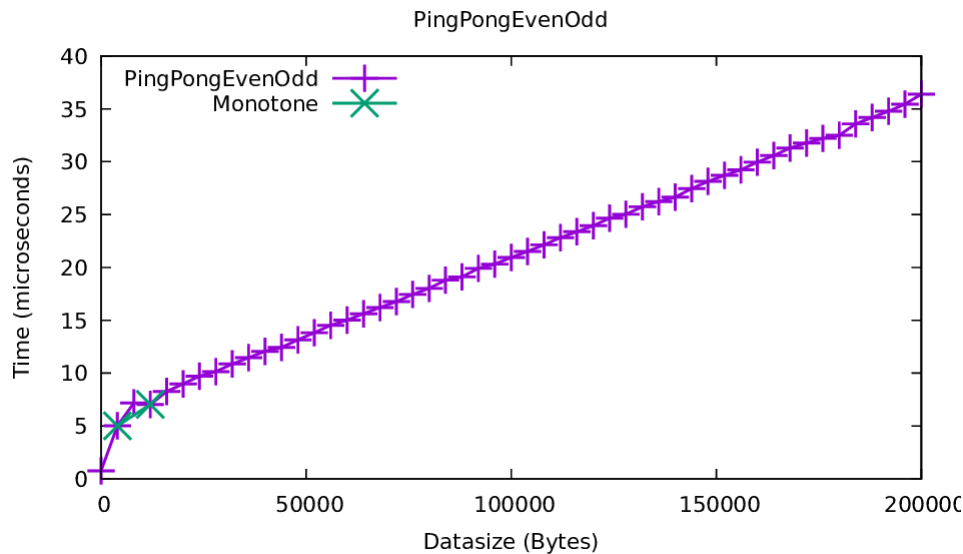
Two MPI processes on same  
SMP node: intra



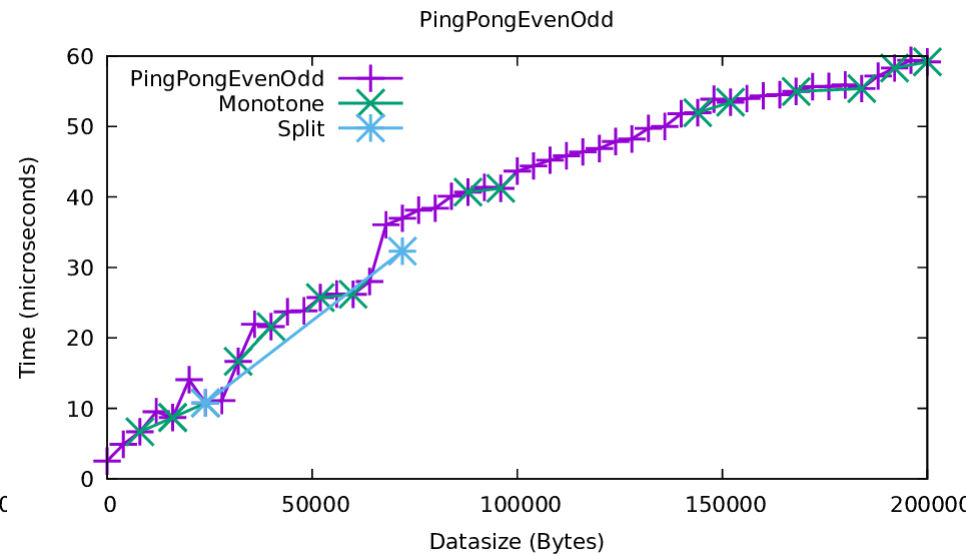
Two MPI processes on  
different SMP nodes: inter

Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

**PingPong** pattern,  $m=200.000\text{Bytes}$



Two MPI processes on same  
SMP node: intra

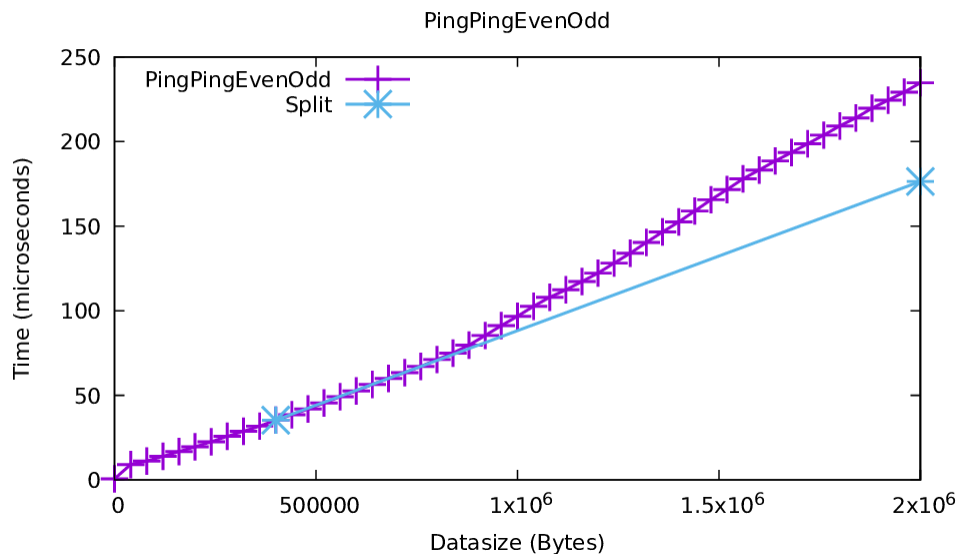


Two MPI processes on  
different SMP nodes: inter

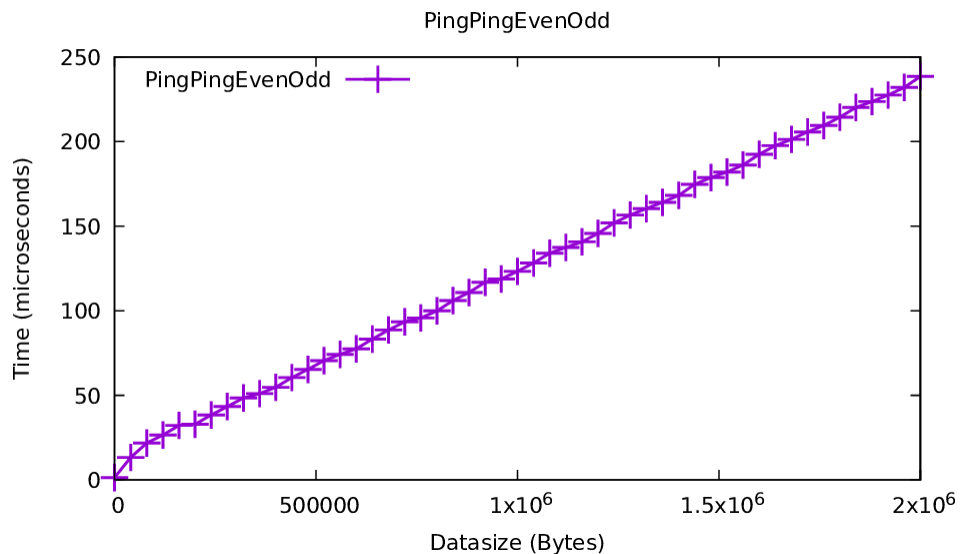


Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

PingPing pattern,  $m=2000.000\text{Bytes}$



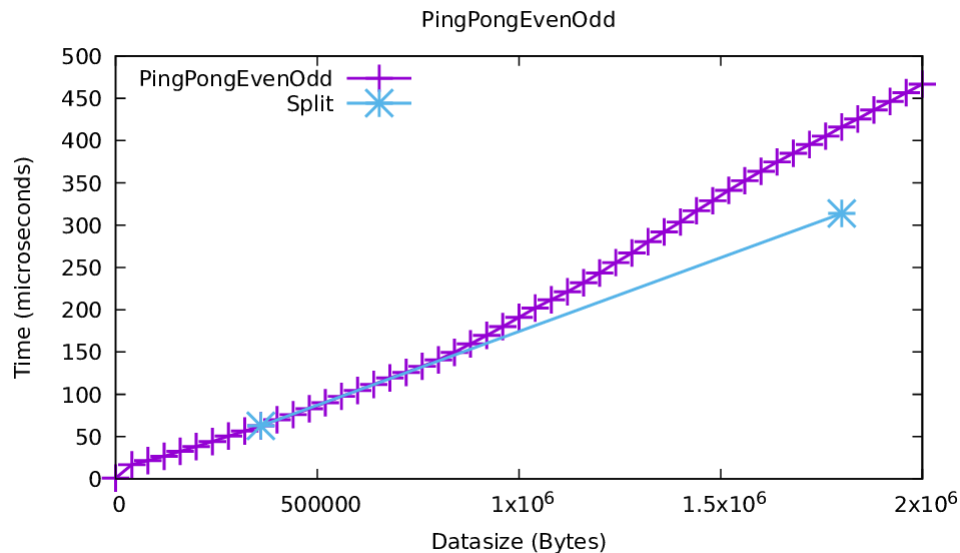
Two MPI processes on same  
SMP node: intra



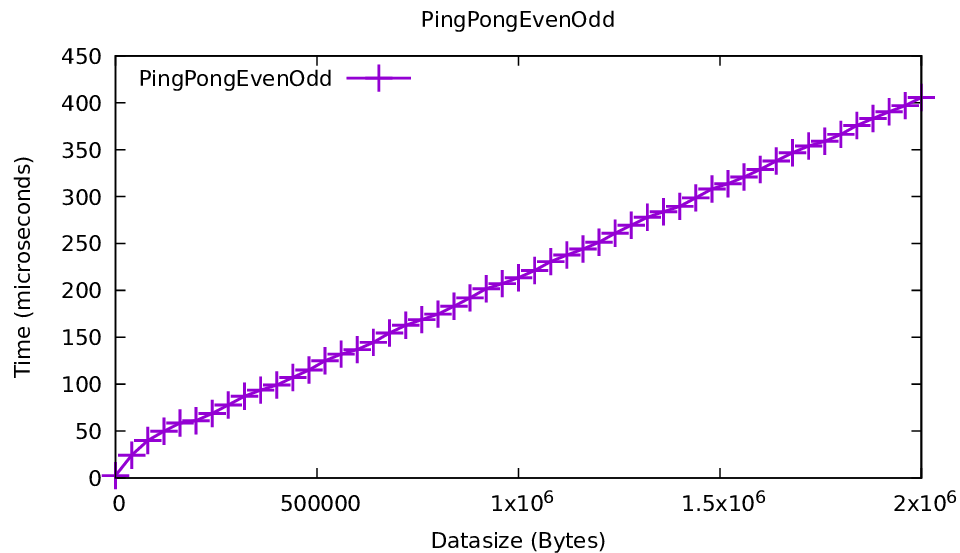
Two MPI processes on  
different SMP nodes: inter

Is the network bidirectional? Compare against PingPing pattern  
Hypothesis:  $t(\text{MPI\_Send} + \text{MPI\_Recv}) \approx 2t(\text{MPI\_Sendrecv})$

**PingPong** pattern,  $m=2000.000\text{Bytes}$



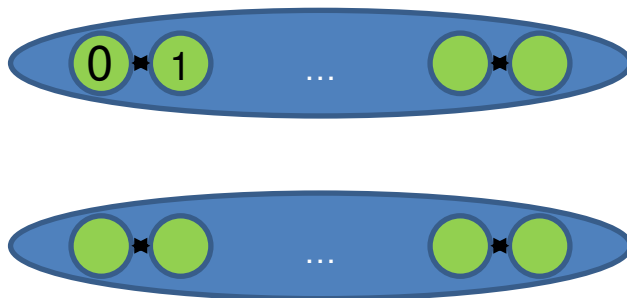
Two MPI processes on same  
SMP node: intra



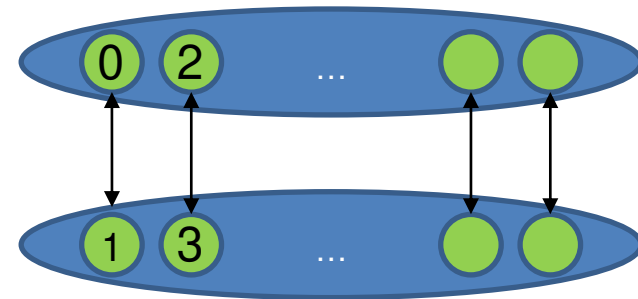
Two MPI processes on  
different SMP nodes: inter

All process pairs communicate (even-odd), MPI\_COMM\_WORLD  
vs. cyclic communicator (all messages between nodes)

MPI\_COMM\_WORLD



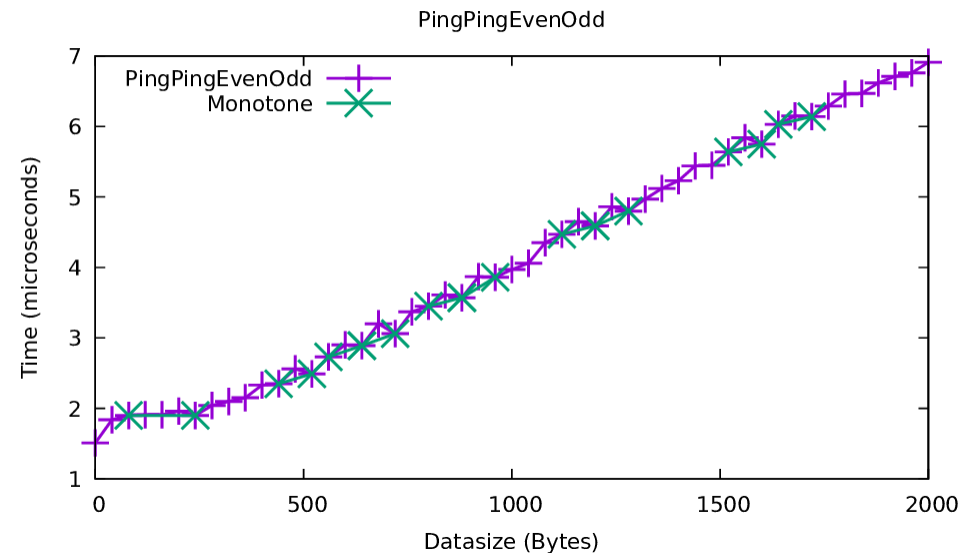
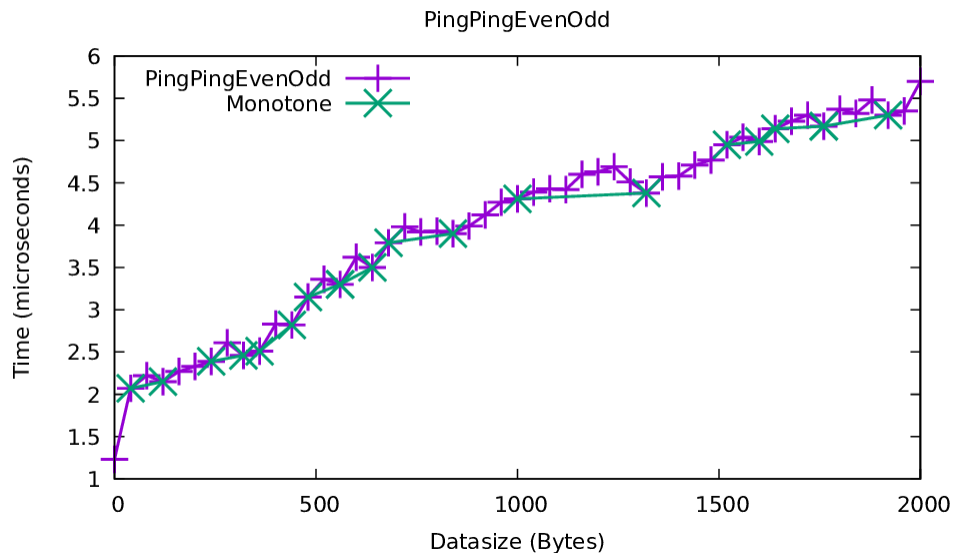
cyclic comm



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)

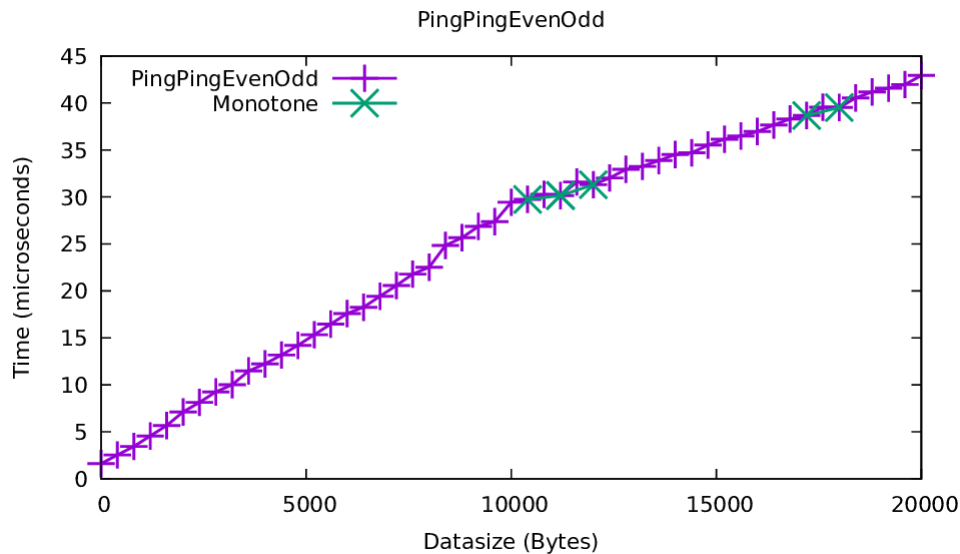
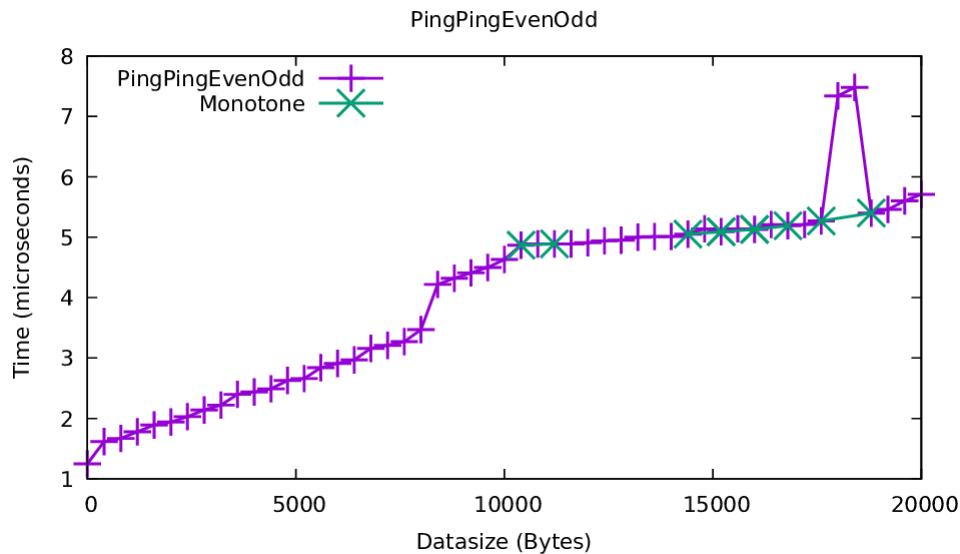
PingPing pattern,  $m=2.000\text{Bytes}$



MPI\_COMM\_WORLD, 2x32  
processes

cyclic comm, 2x32 processes

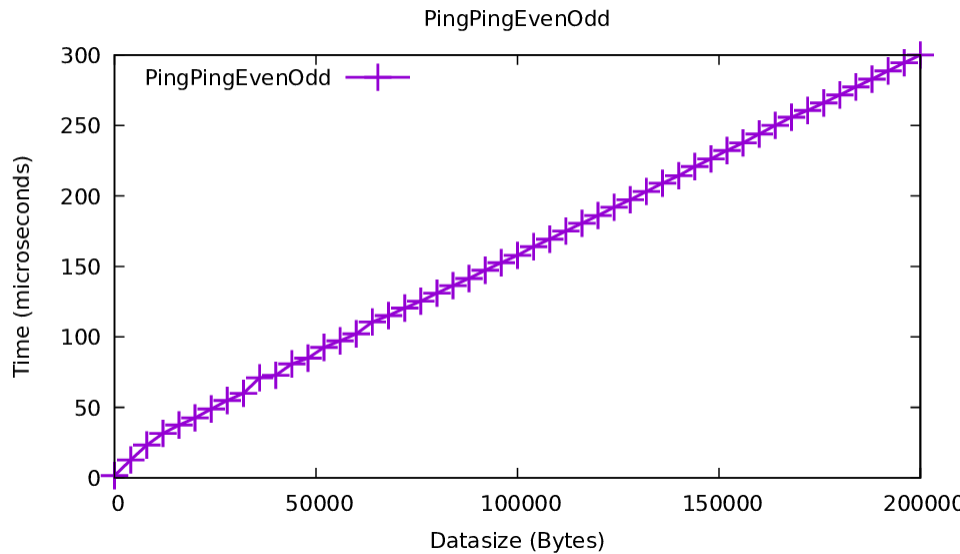
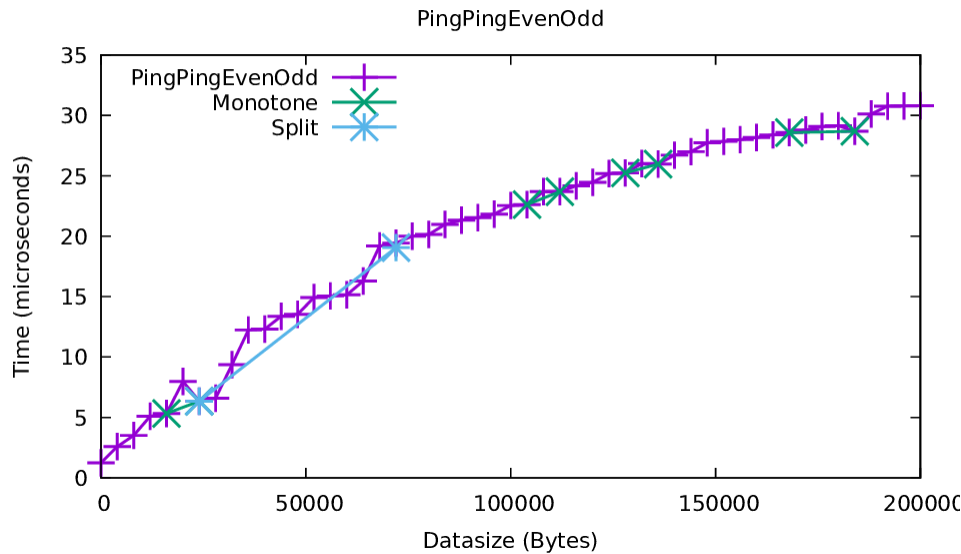
## PingPing pattern, m=20.000Bytes



MPI\_COMM\_WORLD, 2x32  
processes

cyclic comm, 2x32 processes

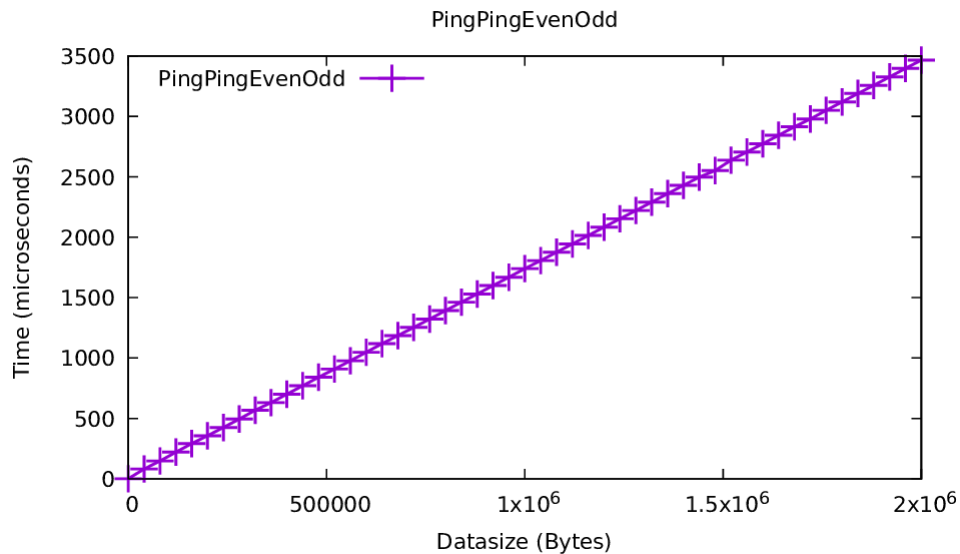
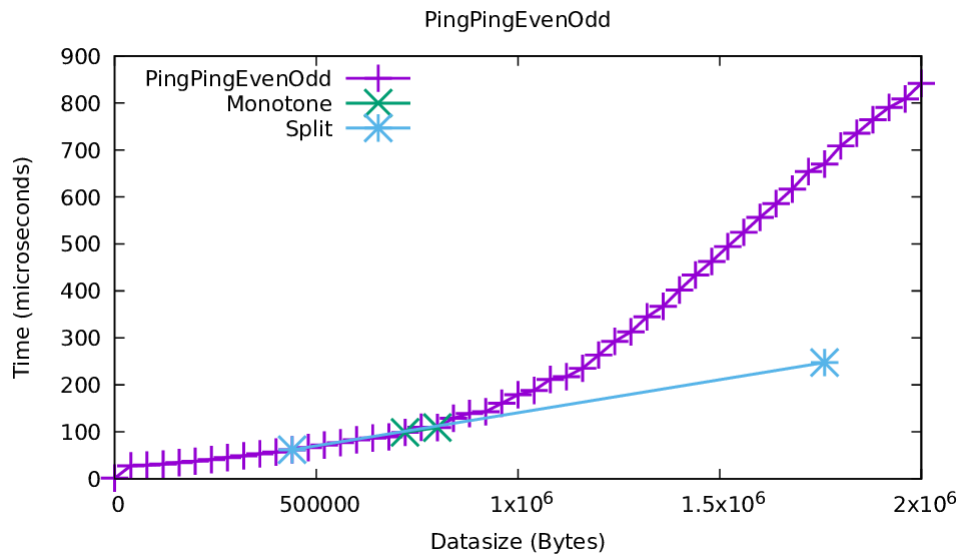
## PingPing pattern, m=200.000Bytes



MPI\_COMM\_WORLD, 2x32  
processes

cyclic comm, 2x32 processes

## PingPing pattern, $m=2000.000\text{Bytes}$



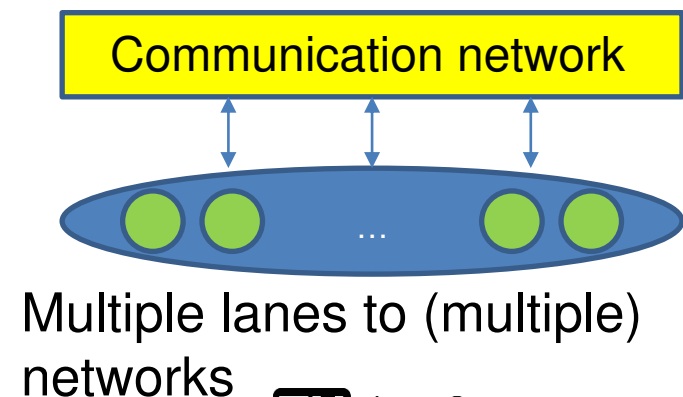
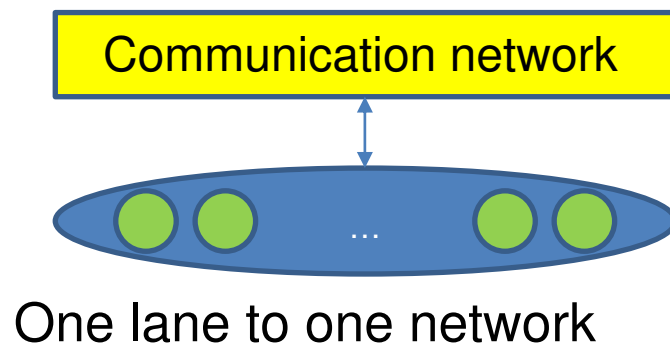
MPI\_COMM\_WORLD, 2x32  
processes

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** )

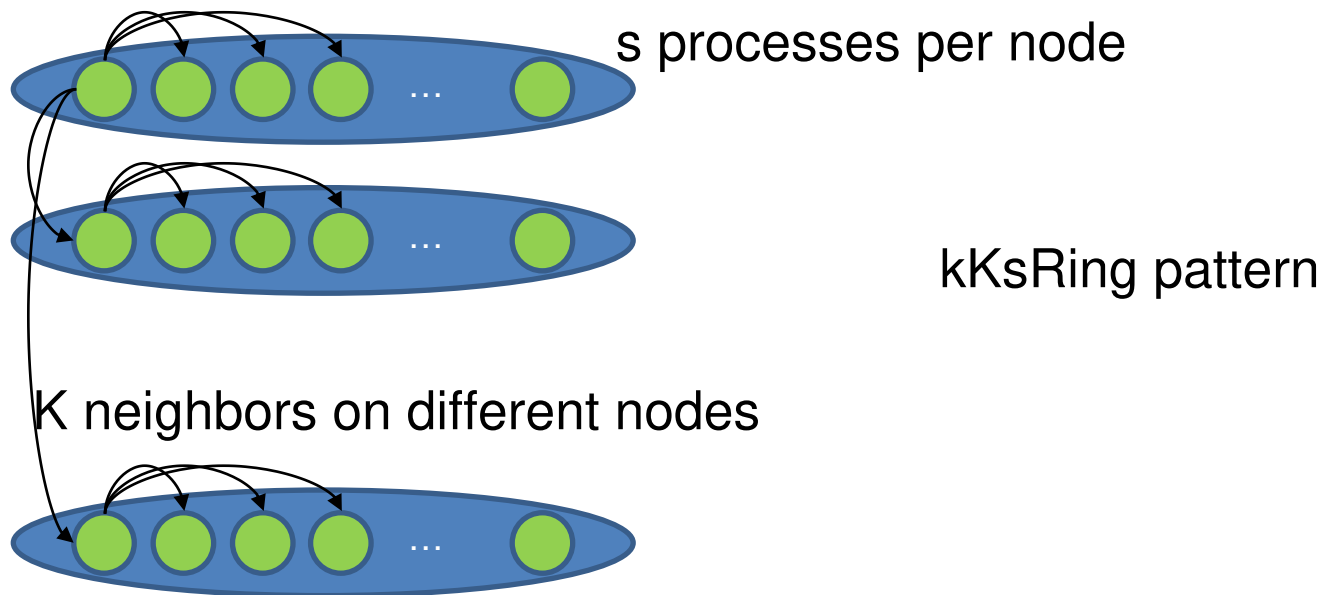
**Note** : Slowdown (much) less than a factor of 32/2: Dual-rail (lane) network in “hydra” cluster





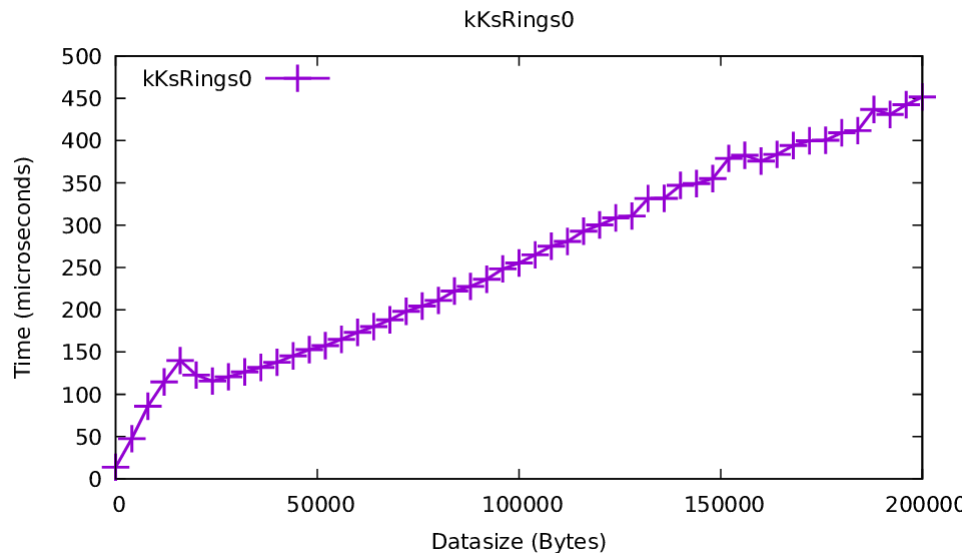
All processes communicate (MPI\_Isend, MPI\_Irecv) with  $k+K$  neighbors.

$k$  neighbors on same node

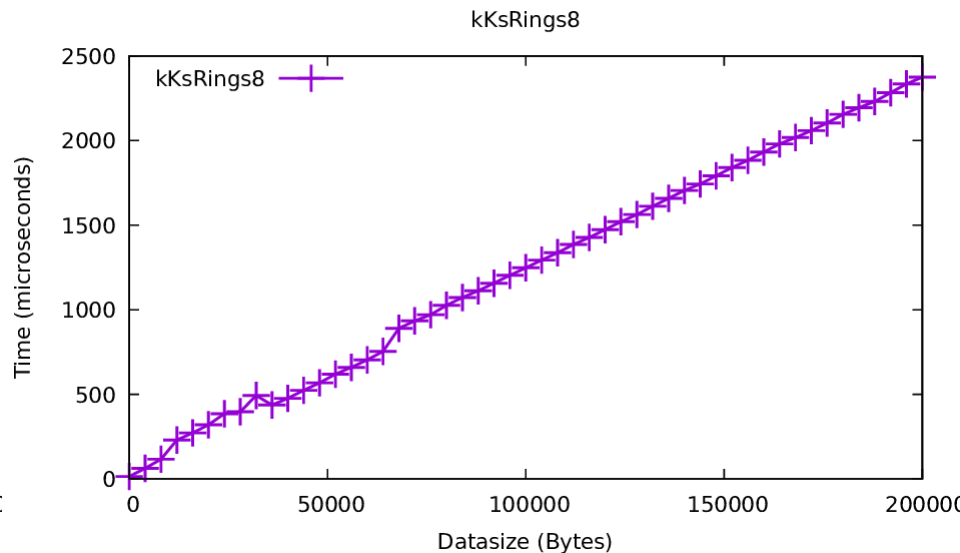


Can MPI\_Cart\_create, MPI\_Dist\_graph\_create with reorder=1 make sense? Is it beneficial to favor intra-node communication?

## kKsRing pattern, m=200.000Bytes

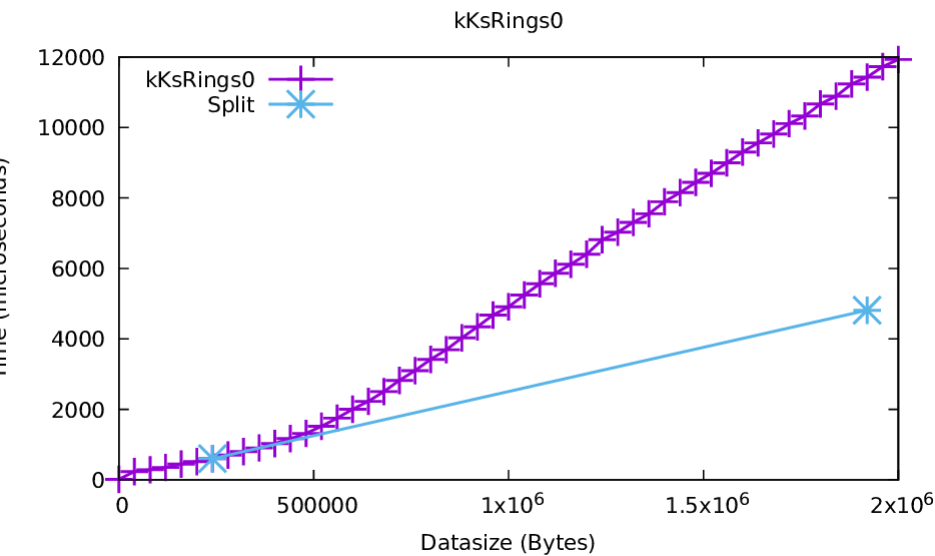


MPI\_COMM\_WORLD,  
36x32 processes, k=8, K=0

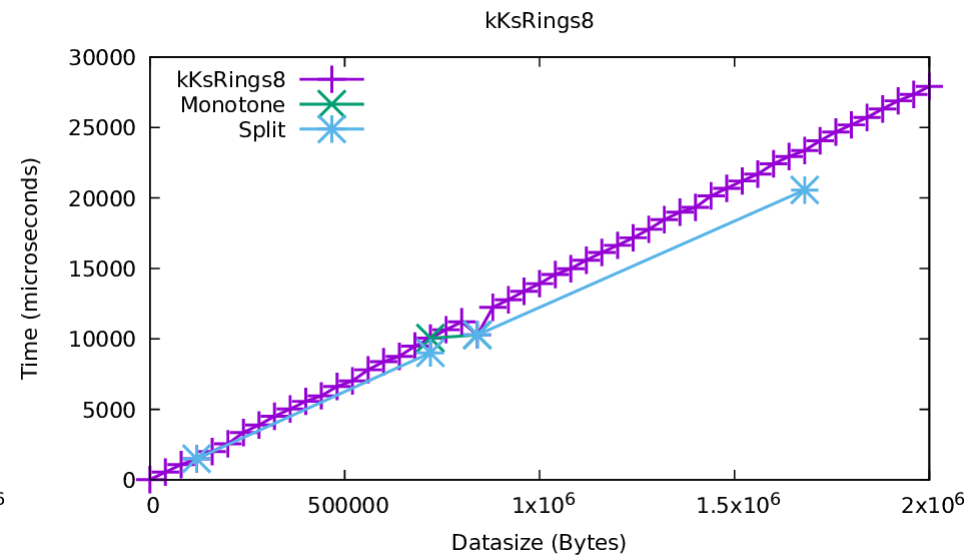


MPI\_COMM\_WORLD,  
36x32 processes, k=0, K=8

## kKsRing pattern, m=2000.000Bytes



MPI\_COMM\_WORLD,  
36x32 processes, k=8, K=0



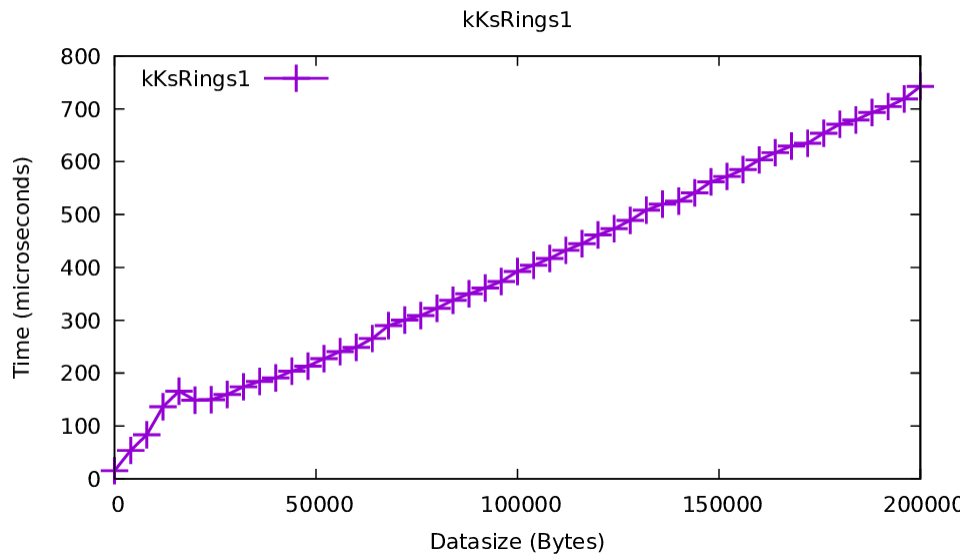
MPI\_COMM\_WORLD,  
36x32 processes, k=0, K=8

All processes communicate (MPI\_Isend, MPI\_Irecv) with  $k+K$  neighbors.

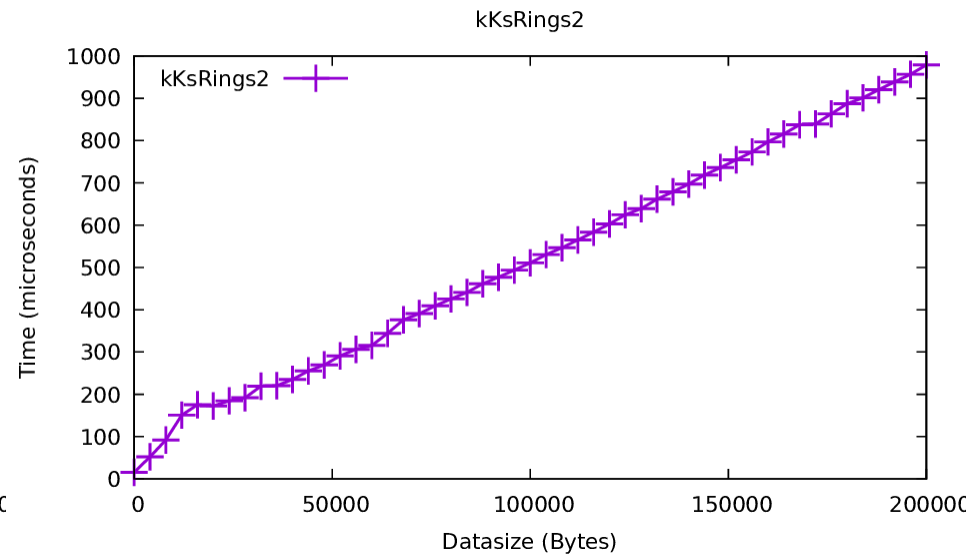
Can MPI\_Cart\_create, MPI\_Dist\_graph\_create with reorder=1 make sense? Is it beneficial to favor intra-node communication?

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

## kKsRing pattern, m=200.000Bytes

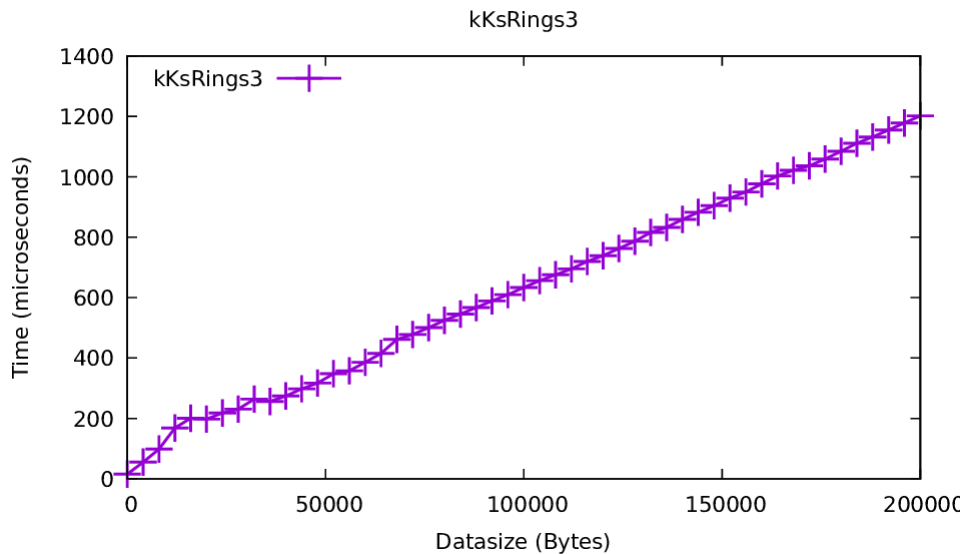


MPI\_COMM\_WORLD,  
36x32 processes, k=7, K=1

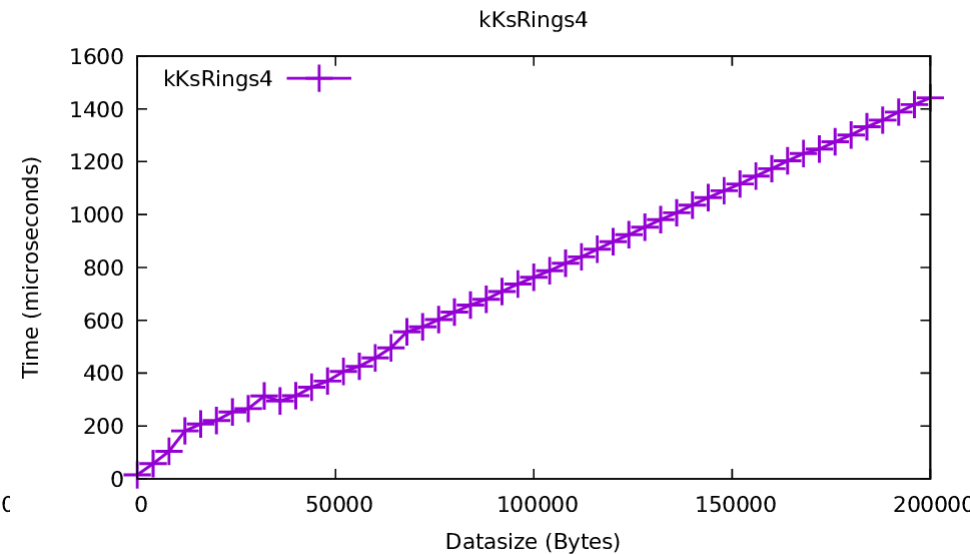


MPI\_COMM\_WORLD,  
36x32 processes, k=6, K=2

## kKsRing pattern, m=200.000Bytes

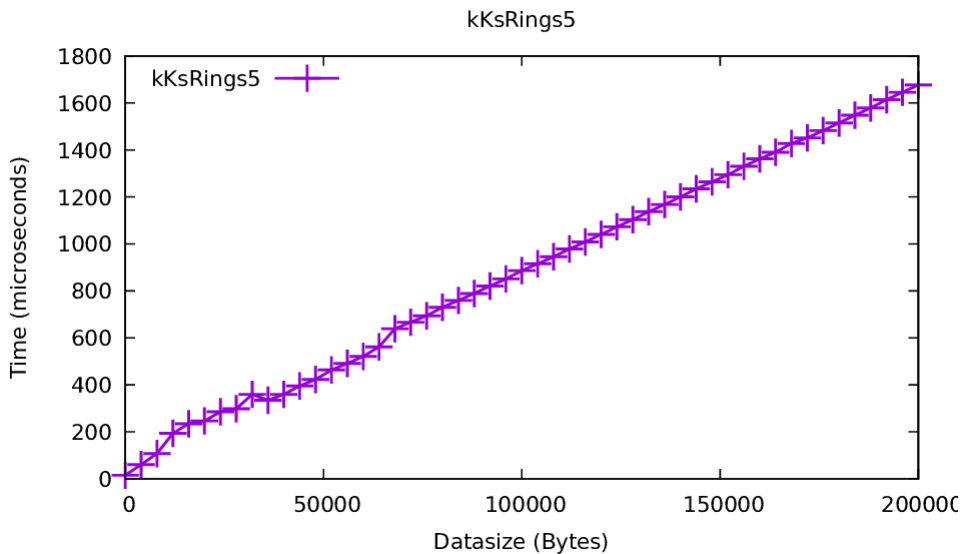


MPI\_COMM\_WORLD,  
36x32 processes, k=5, K=3

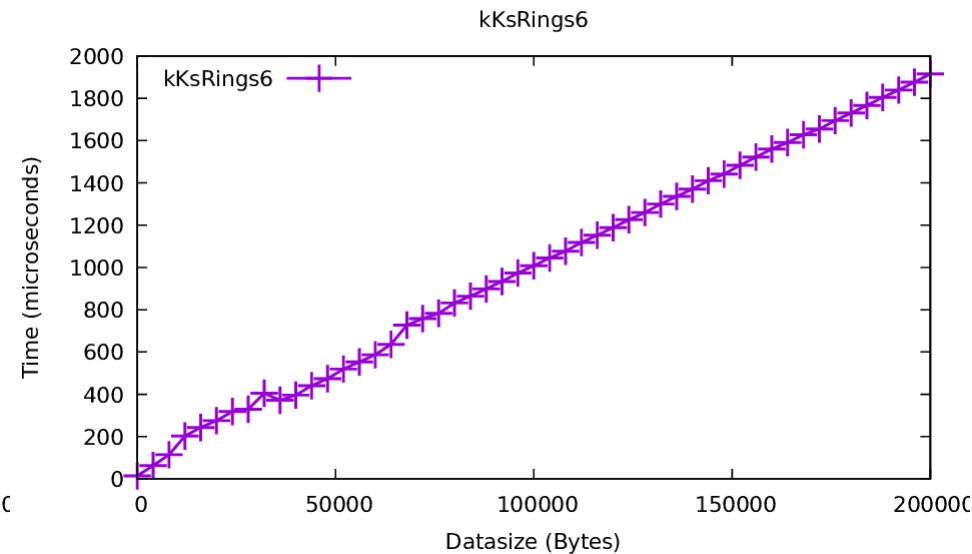


MPI\_COMM\_WORLD,  
36x32 processes, k=4, K=4

## kKsRing pattern, m=200.000Bytes

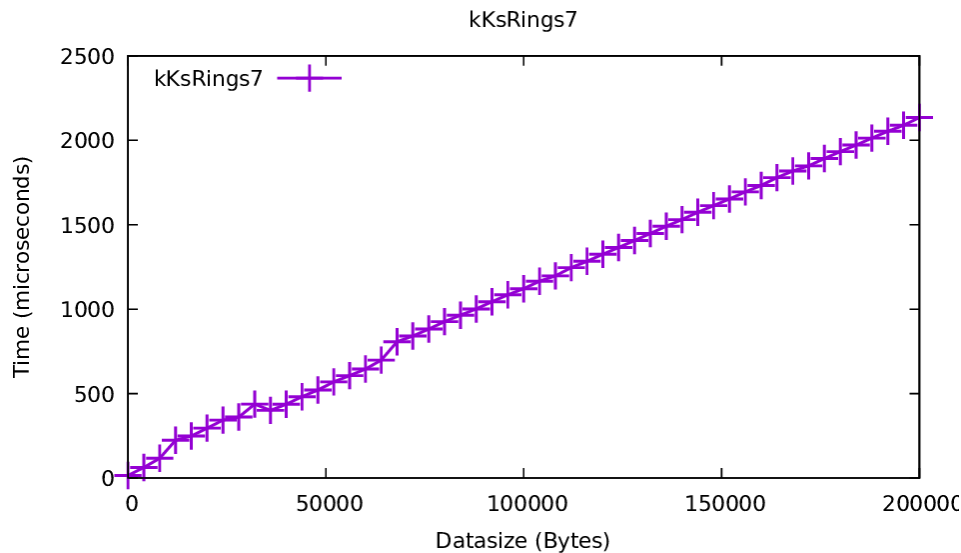


MPI\_COMM\_WORLD,  
36x32 processes, k=3, K=5

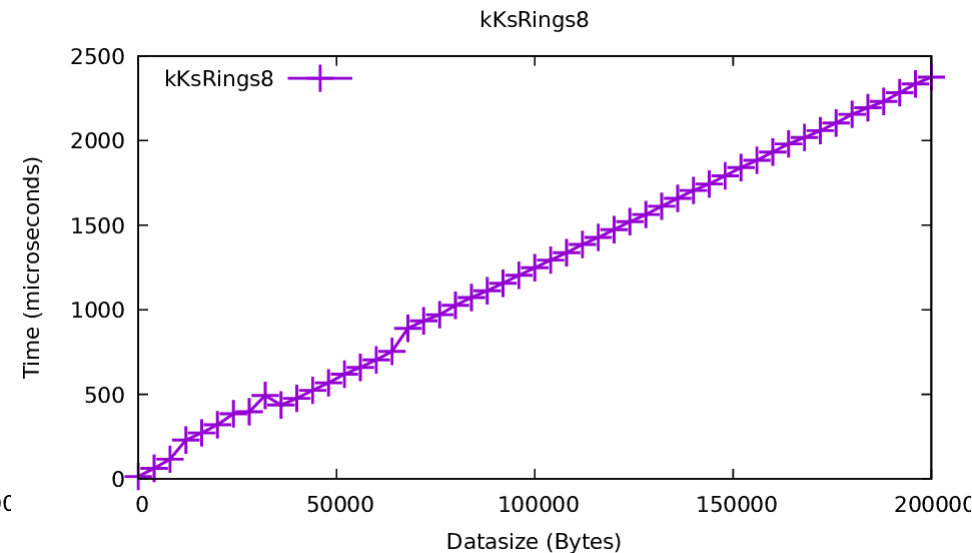


MPI\_COMM\_WORLD,  
36x32 processes, k=2, K=6

## kKsRing pattern, m=200.000Bytes



MPI\_COMM\_WORLD,  
36x32 processes, k=1, K=7



MPI\_COMM\_WORLD,  
36x32 processes, k=0, K=8



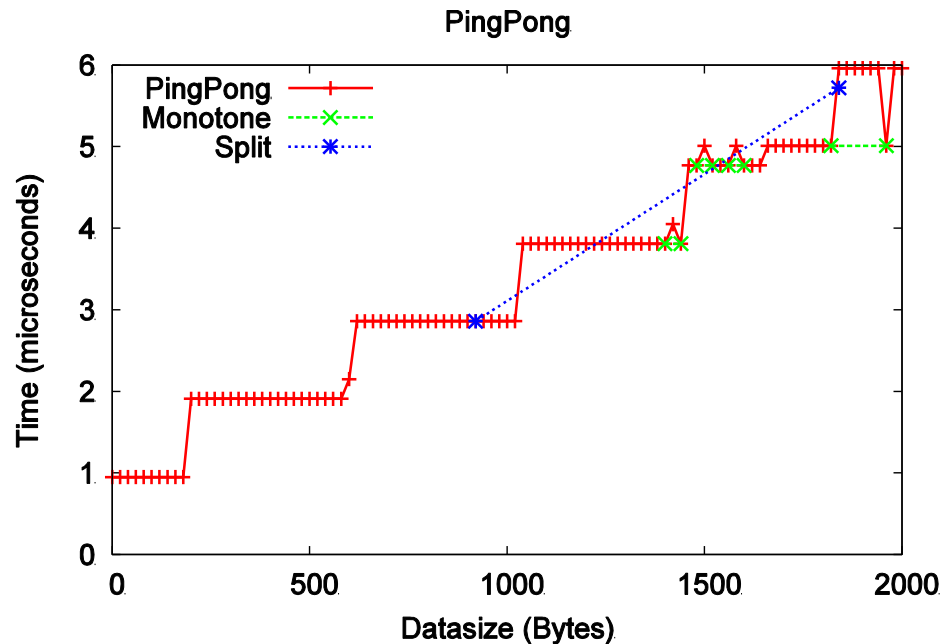
## Some conclusions:

- 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

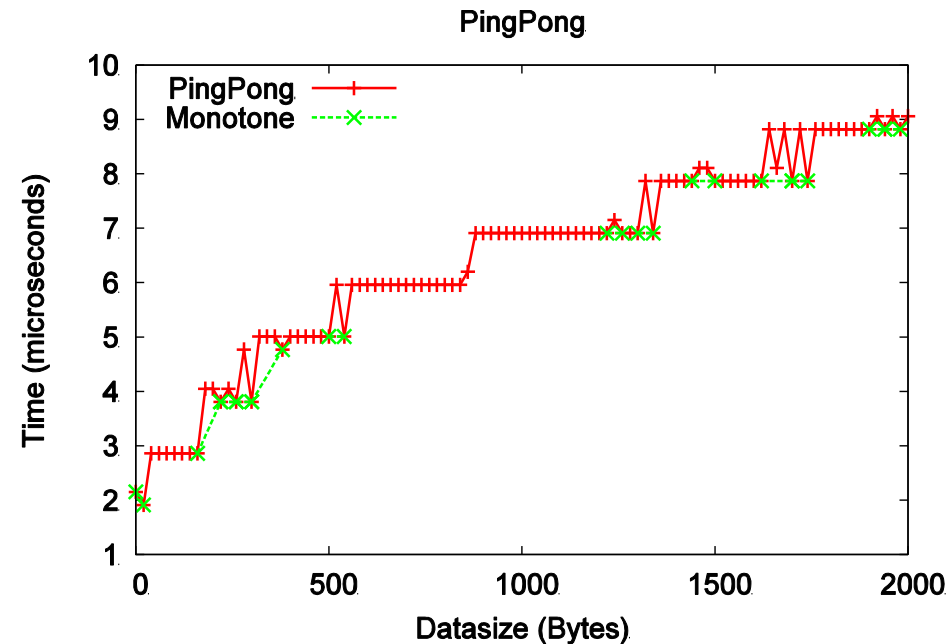
## Experimental factors(II)

- Process placement (across the nodes/parts of the system)
- Process pinning (on the node, disable process migration)

## Linear cost model on Jupiter (mpicroscope benchmark)?

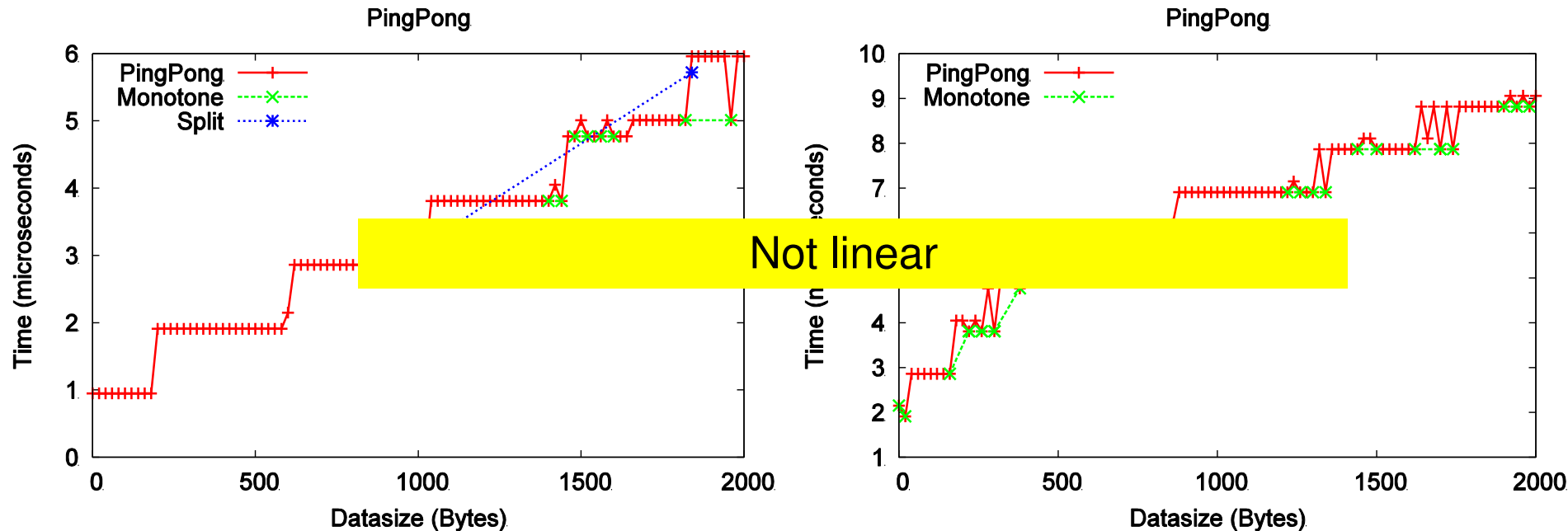


Two MPI processes on same  
SMP node: intra



Two MPI processes on  
different SMP nodes: inter

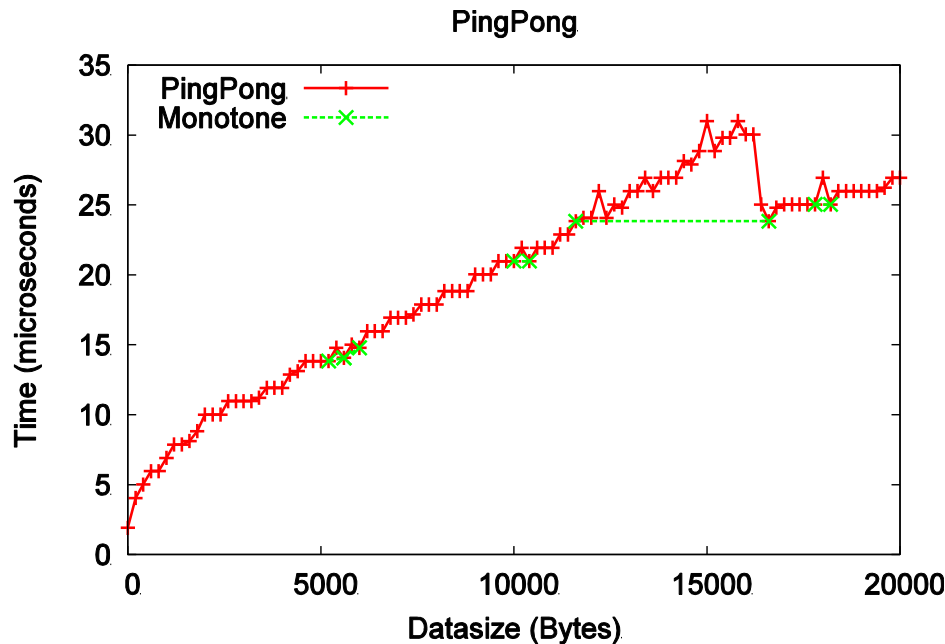
# Linear cost model on Jupiter (mpicroscope benchmark)?



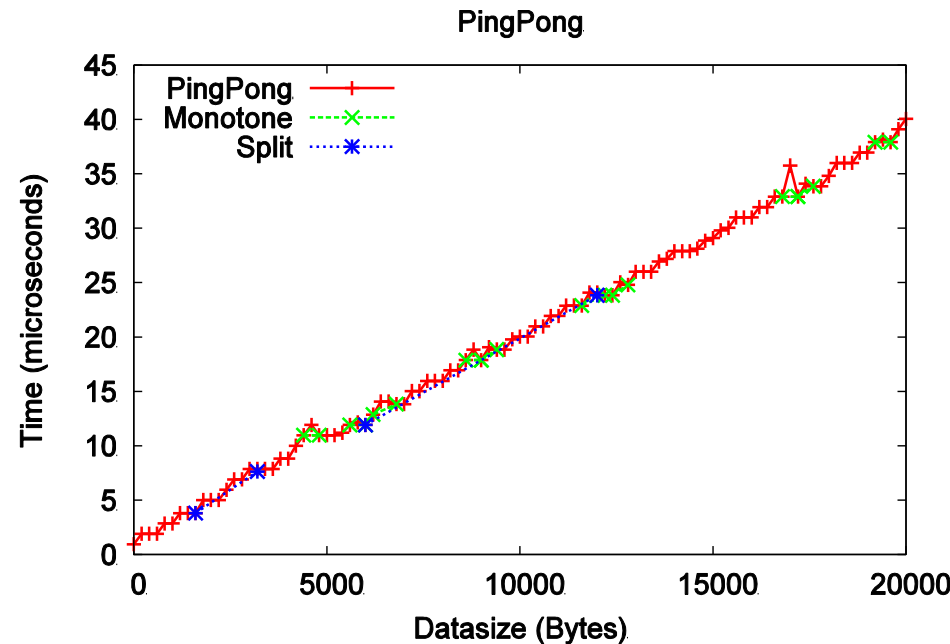
Two MPI processes on same  
SMP node: intra

Two MPI processes on  
different SMP nodes: inter

# Linear cost model on Jupiter (mpicroscope benchmark)?



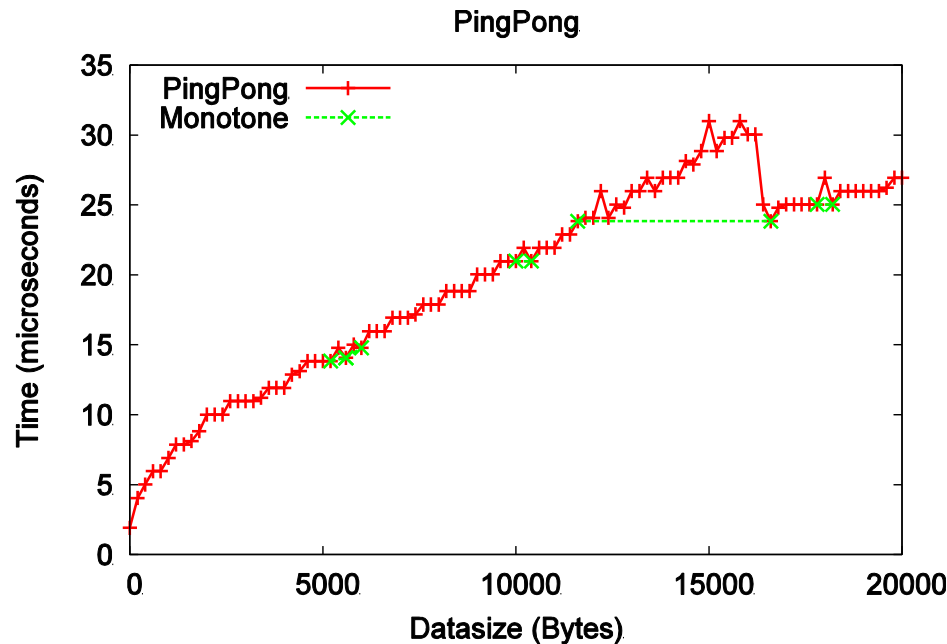
Intra-node



Inter-node

Larger messages: approx. linear(?)

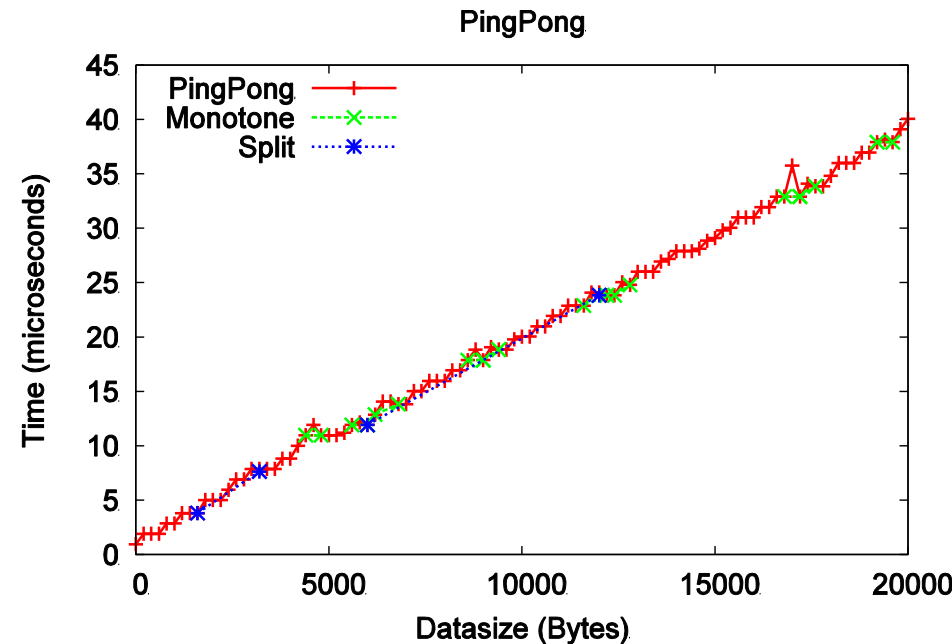
# Linear cost model on Jupiter (mpicroscope benchmark)?



Intra-node

$$\alpha \approx 1 \mu\text{s}$$

$$\beta \approx 0.002 \mu\text{s/Byte}$$



Inter-node

$$\alpha \approx 2 \mu\text{s}$$

$$\beta \approx 0.002 \mu\text{s/Byte}$$

## Experimental factors(III)

- The `mpirun` command (different runs behave differently)
- Compiler, compiler options
- Cache (warm cache vs. cold cache: a question of experimental design)

Good practice:

Average benchmark over several `mpirun`'s

Experimental factors (things that cannot be influenced, “noise”: repetition&statistics) vs. experimental design (design choices)

Sascha Hunold, Alexandra Carpen-Amarie: Reproducible MPI Benchmarking is Still Not as Easy as You Think. IEEE Trans. Parallel Distrib. Syst. 27(12): 3617-3630 (2016)

## Refinement of linear cost model :

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

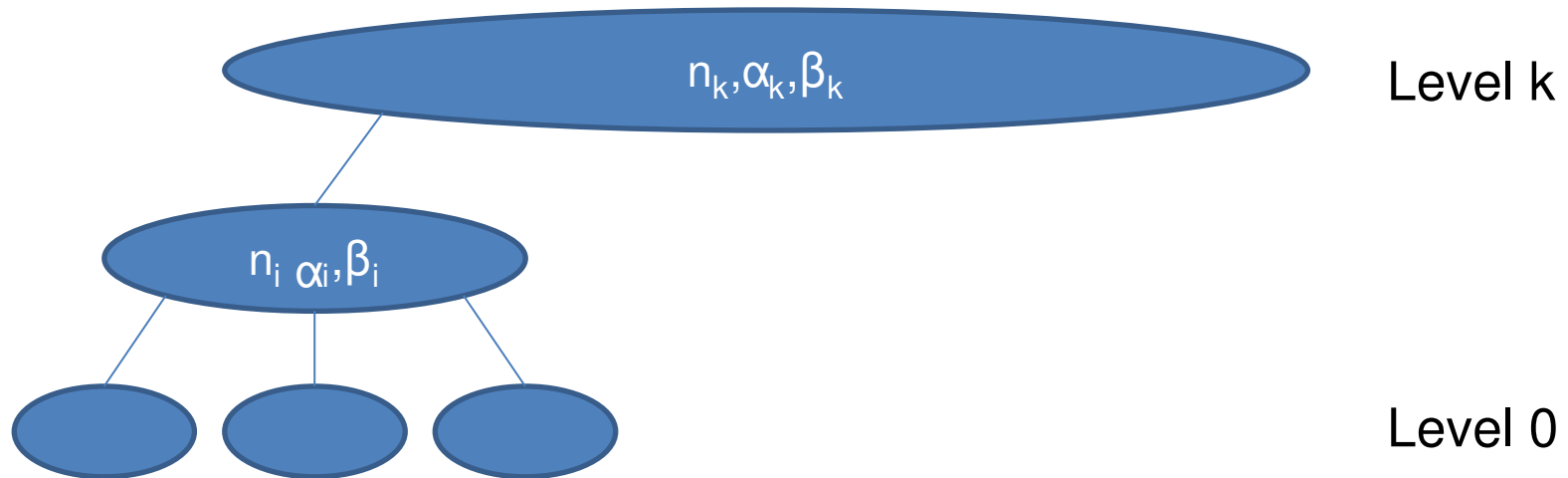
$$t(m) = \begin{cases} \alpha_1 + \beta_1 m, & \text{if } 0 \leq m < M_1 \\ \alpha_2 + \beta_2 m, & \text{if } M_1 \leq m < M_2 \\ \alpha_3 + \beta_3 m, & \text{if } M_2 \leq m \end{cases}$$

Etc.



## Non-homogeneous, hierarchical systems

Regular, hierarchical system can represent  $\alpha_{ij}, \beta_{ij}$  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  $\alpha_i, \beta_i$ , system described by the sequence  $(n_i, \alpha_i, \beta_i), i=0, \dots, k$ . 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

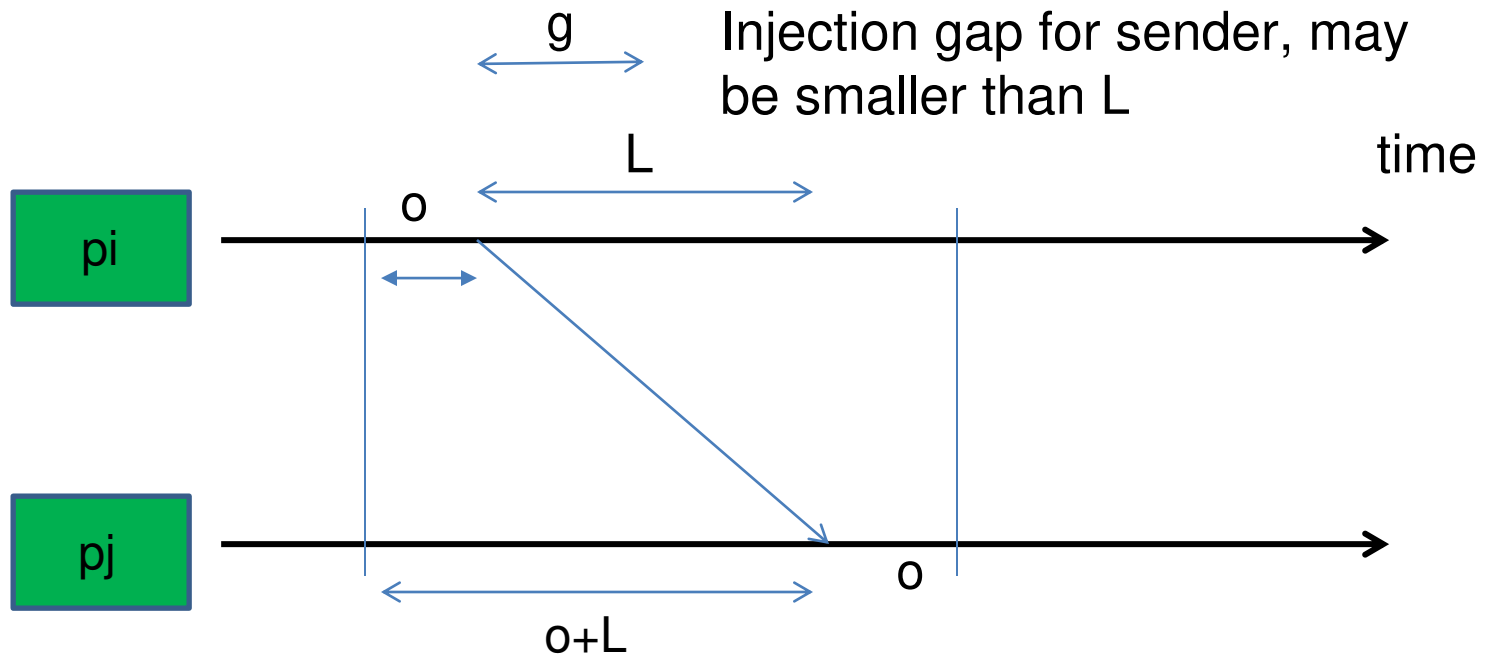
o: overhead, for processor to start/complete message transfer (both send and receive)

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)

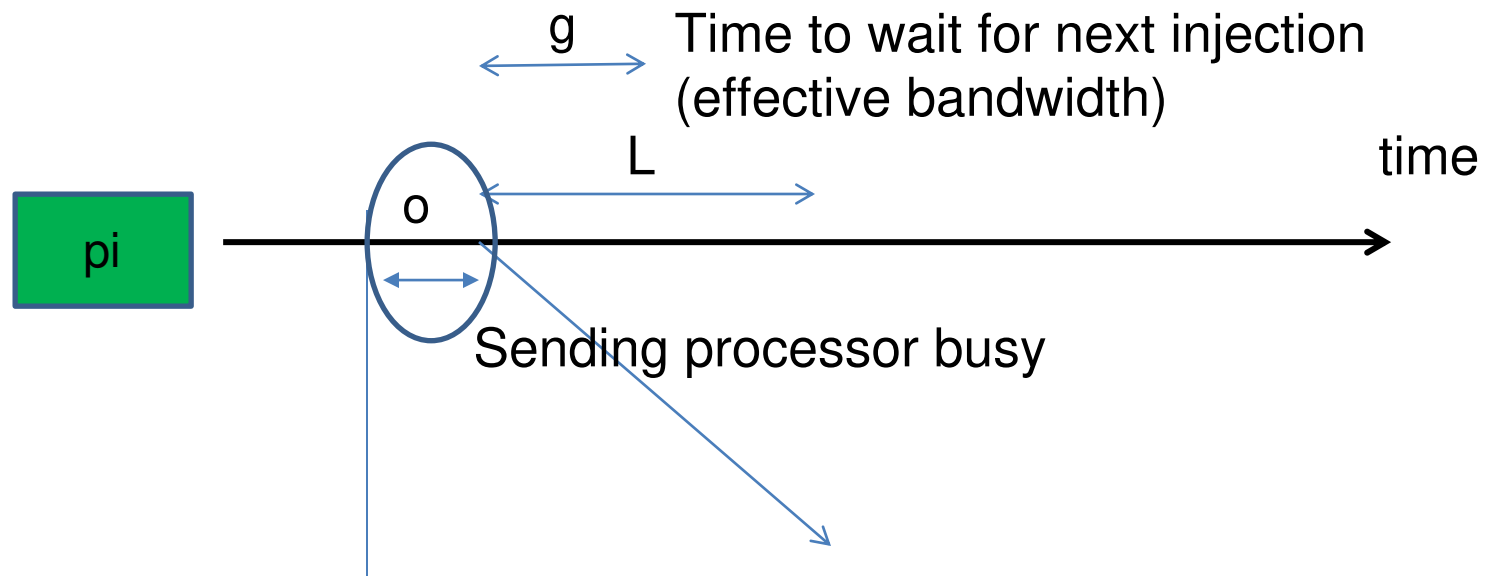
Processor i sending message to processor j



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

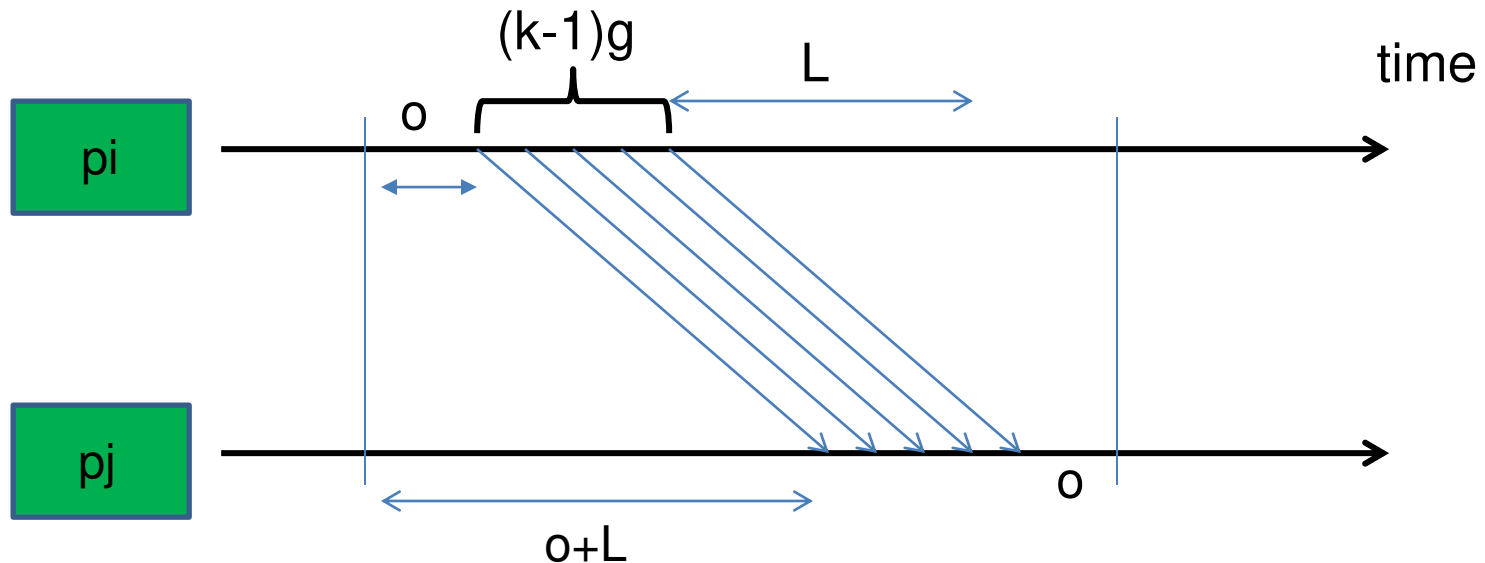
Sender and receiver only involved for  $o$  seconds; overlap possible

LogP is not a (synchronous) transmission cost model



Additional network capacity constraint: at most  $\text{ceil}(L/g)$  messages can be in transit (if more, sending process stalls)

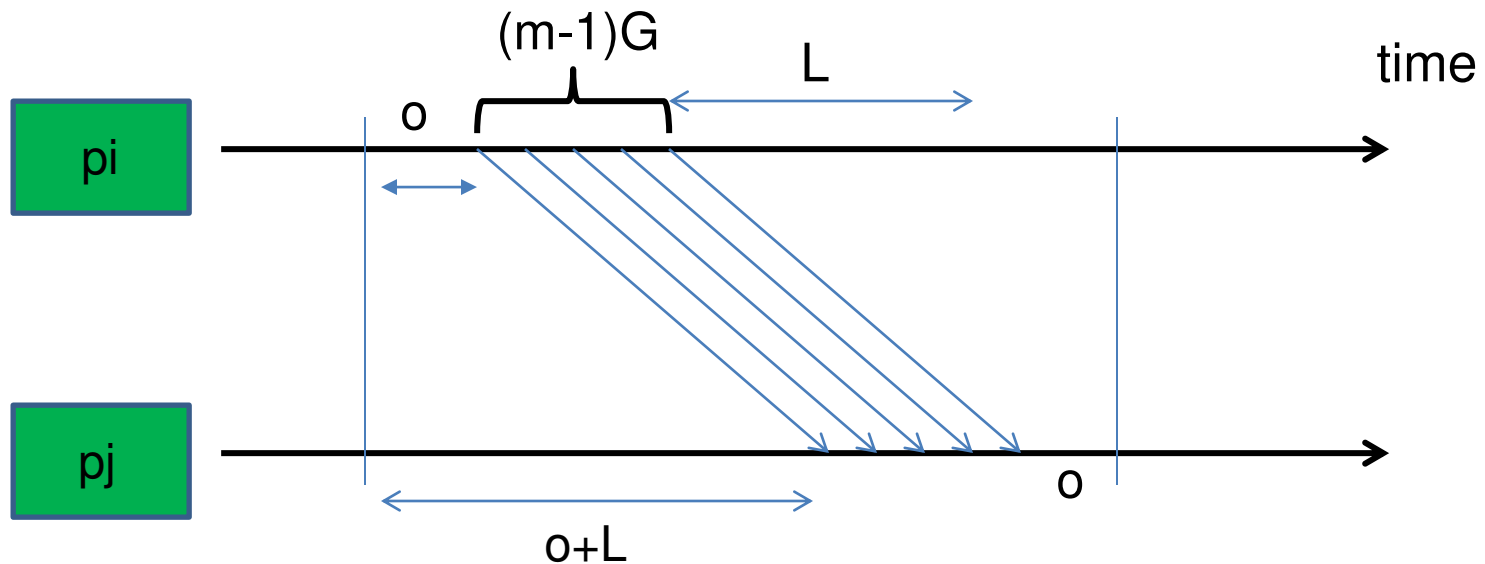
Processor i sending message to processor j



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

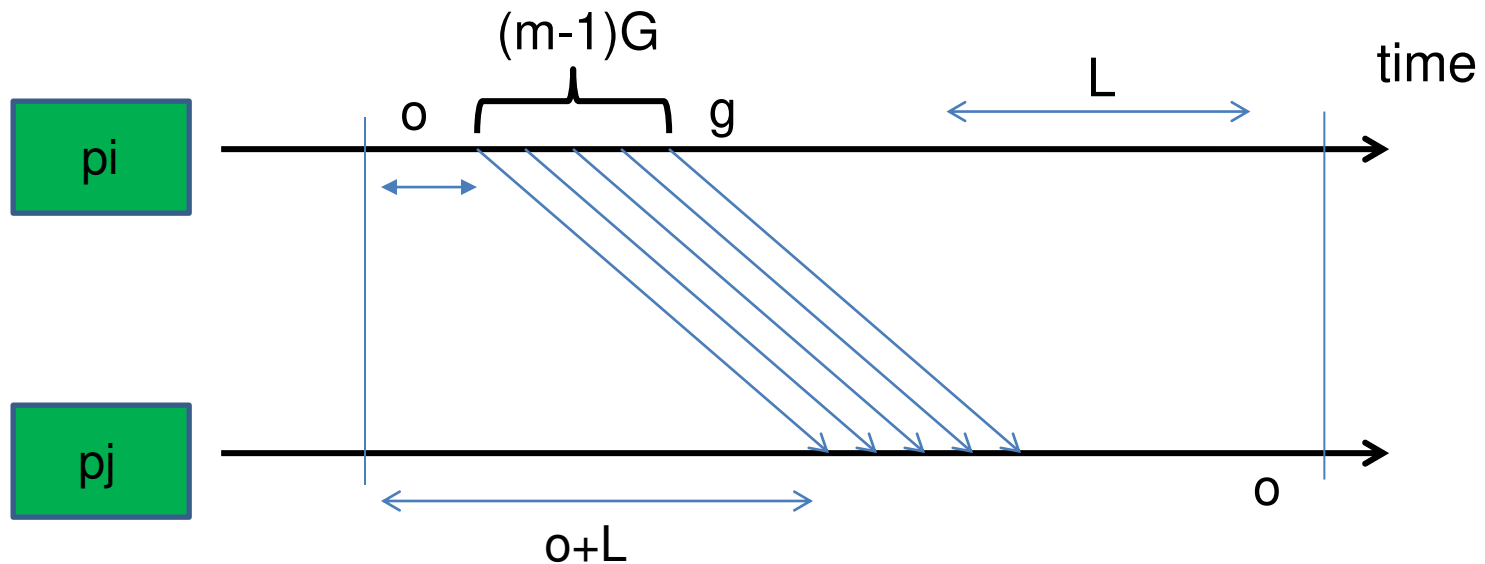
Next message can be sent after  $g$  time units (assume  $g \geq 0$ )

Processor i sending message to processor j



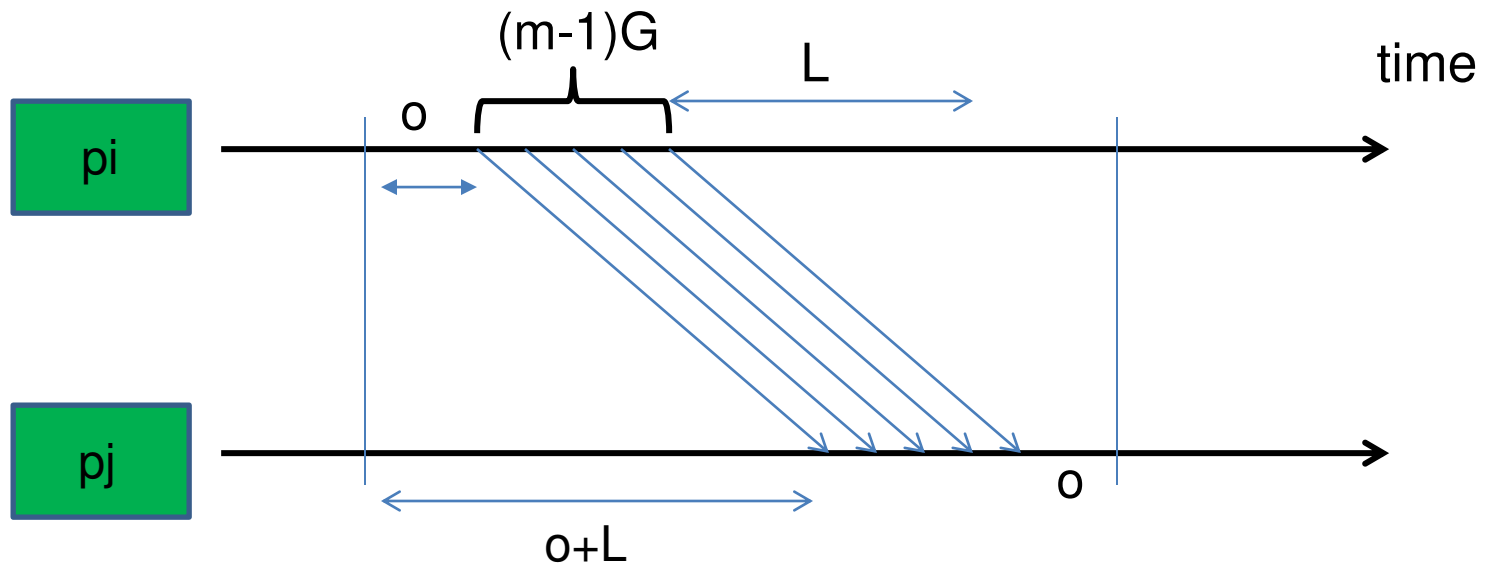
Sending large message at time  $t$ , receive at time  $t+o+(m-1)G+L+o$

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**):

$$\alpha \approx 2o+L, \beta \approx G$$



Starting point for LogGP, see:

D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauer, R. Subramonian, T. von Eicken: LogP: A Practical Model of Parallel Computation. Comm ACM 39(11): 78-85 (1996)  
Alexandrov, M. F. Ionescu, Klaus E. Schauer, C. J. Scheiman: LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. J. Parallel Distrib. Comput. 44(1): 71-79 (1997)

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

Eunice E. Santos: Optimal and Near-Optimal Algorithms for k-Item Broadcast. J. Parallel Distrib. Comput. 57(2): 121-139 (1999)  
Eunice E. Santos: Optimal and Efficient Algorithms for Summing and Prefix Summing on Parallel Machines. J. Parallel Distrib. Comput. 62(4): 517-543 (2002)

## 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:

Juan A. Rico-Gallego, Juan Carlos Díaz Martín, Ravi Reddy Manumachu, Alexey L. Lastovetsky: A Survey of Communication Performance Models for High-Performance Computing. ACM Comput. Surv. 51(6): 126:1-126:36 (2019)

## Historical on LogP:

D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauer, R. Subramonian, T. von Eicken: LogP: A Practical Model of Parallel Computation. Comm ACM 39(11): 78-85 (1996)

- David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauer, Eunice E. Santos, Ramesh Subramonian, Thorsten von Eicken: LogP: Towards a Realistic Model of Parallel Computation. PPOPP 1993: 1-12

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

Was this right? Productive?

“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”, “communication structure”)

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

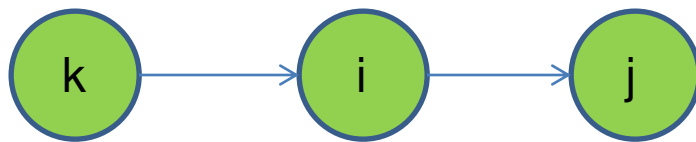
Most ( **but not all** : shuffle-exchange, deBruijn, Kautz) networks  $G=(V,E)$  are undirected, a communication edge  $(i,j)$  allows communication from  $i$  to  $j$ , and from  $j$  to  $i$ .

Network (graph) properties:

- 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  $|V_1| \approx |V_2|$

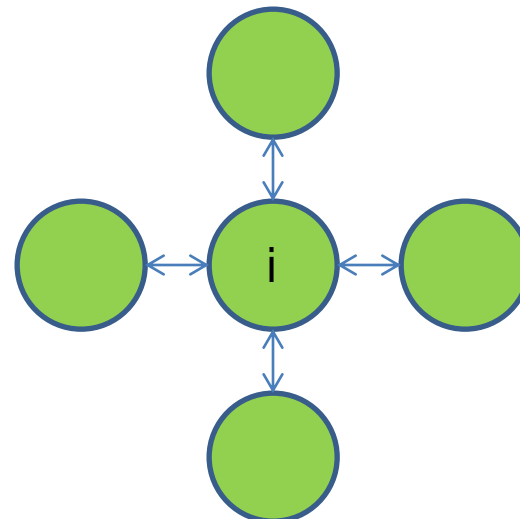


Single-ported, fully bidirectional, send-receive communication:

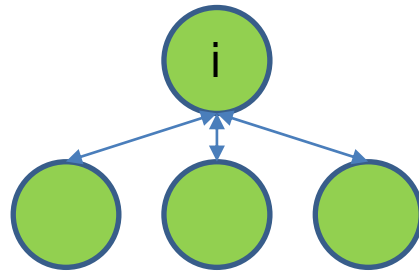


Processor i can simultaneously send to processor j and receive from processor k,  $k=j$  (telephone) or  $k \neq j$

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

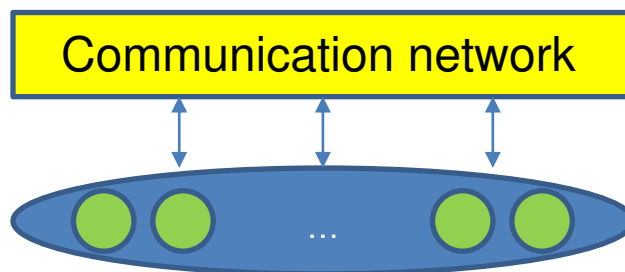


## k-ported vs. k-lane communication capabilities



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

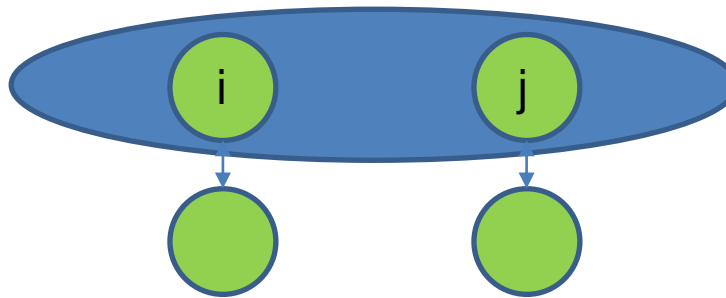


$k$  lanes (rails) shared between processes

Is the k-ported model realistic?

k-lane assumption:

Master theses?



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?

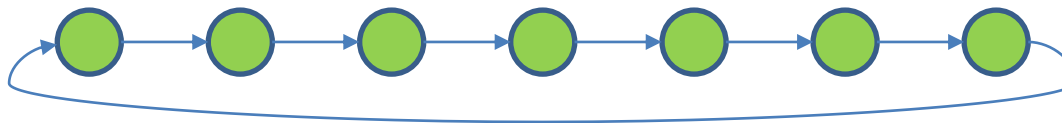
Jesper Larsson Träff:  $k$ -ported vs.  $k$ -lane Broadcast, Scatter, and Alltoall Algorithms. CoRR abs/2008.12144 (2020)

## Some communication networks (actual, or as design vehicles)

Linear processor array; diameter  $p-1$

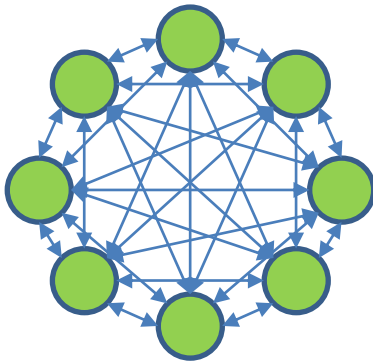


Linear processor ring; diameter  $p-1$ , but now strongly connected

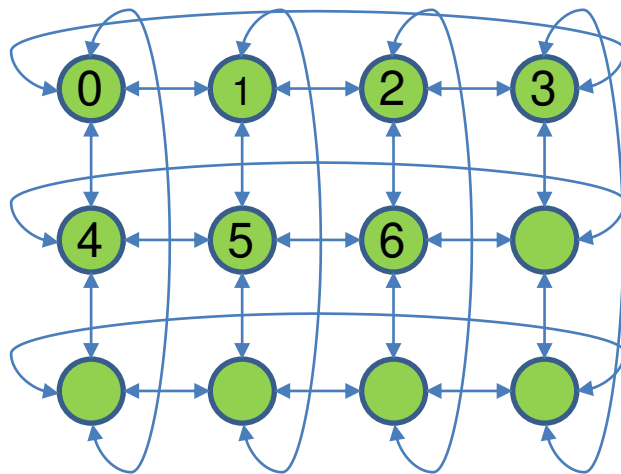


Fully connected network:

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  $p^2/4$



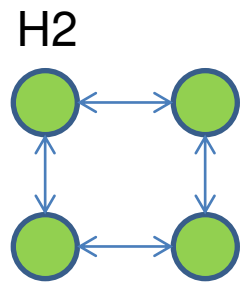
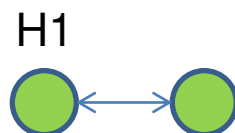
( $r \times c \times \dots$ ) d-dimensional mesh/torus: each processor has 2d neighbors, diameter  $(r-1)+(c-1)+\dots$  (roughly, divide by 2 for torus)



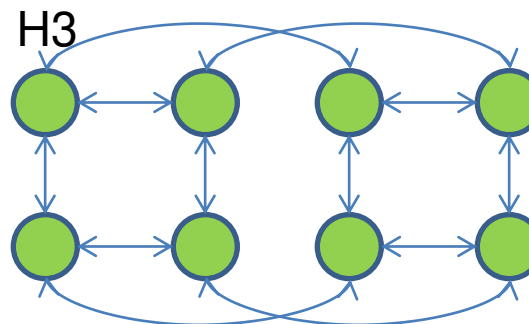
Row-order (or some other) numbering

Torus wrap-around edges

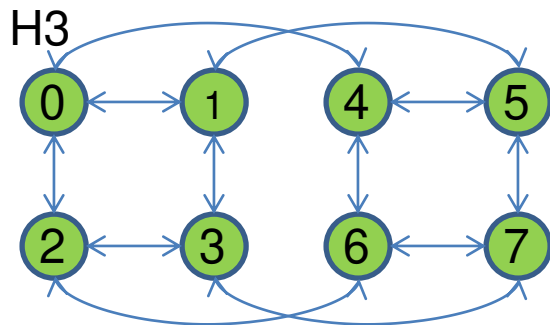
Hypercube:  $(\log_2 p)$ -dimensional torus, size 2 in each dimension



Processors  $0x$  (binary)  
and  $1x$  (binary)  
neighbors in  $H_i$  for  $x$  in  
 $H(i-1)$ ,  $i > 1$



Hypercube:  $(\log_2 p)$ -dimensional torus, size 2 in each dimension



Processors  $0x$  (binary) and  $1x$  (binary) neighbors in  $H_i$  for  $x$  in  $H(i-1)$ ,  $i > 1$

Diameter:  $\log p$

Degree:  $\log p$ , each processor has  $\log p$  neighbors

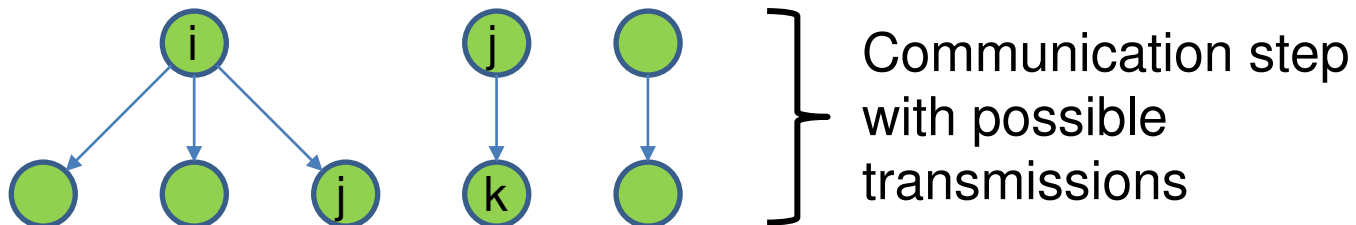
Naming:  $k$ 'th neighbor of processor  $i$ : flip  $k$ 'th bit of  $i$

**Remark:** This is a particular, often convenient naming of the processors in the hypercube



## Communication algorithms in networks

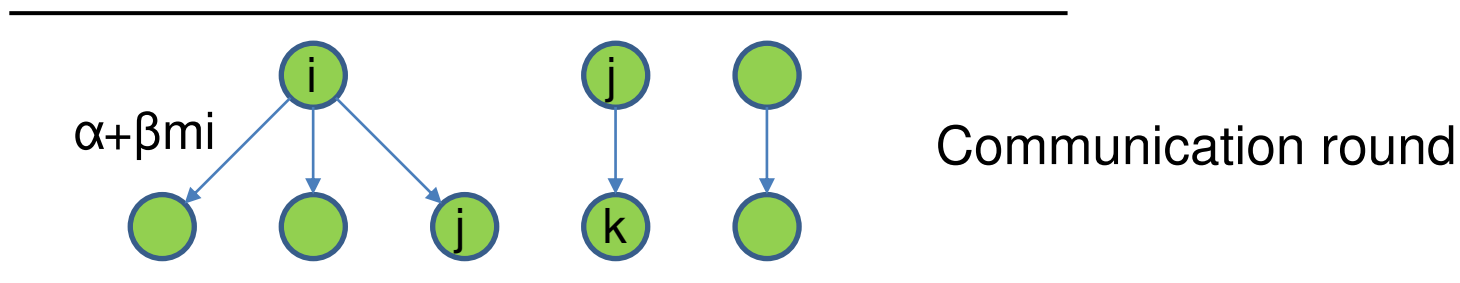
Assumption: Independent pairs consisting of a sending and a receiving processor in a network can communicate independently, concurrently, in parallel with all other pairs.



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

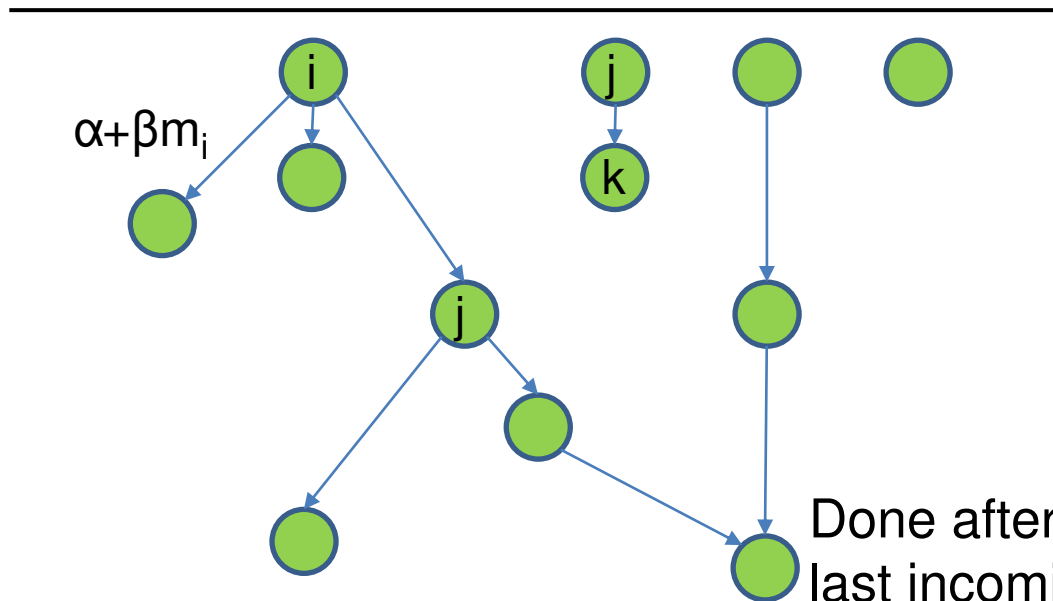
Sometimes computation between rounds is not accounted for (too fast; not relevant; ...); unbounded computation assumed...

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  $\alpha + \beta m_i$

For a synthesis with more networks:

Pierre Fraigniaud, Emmanuel Lazard: Methods and problems of communication in usual networks. Discrete Applied Mathematics 53(1-3): 79-133 (1994)

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

See also notes on  
“Algorithms for  
Collective Operations”

## 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:

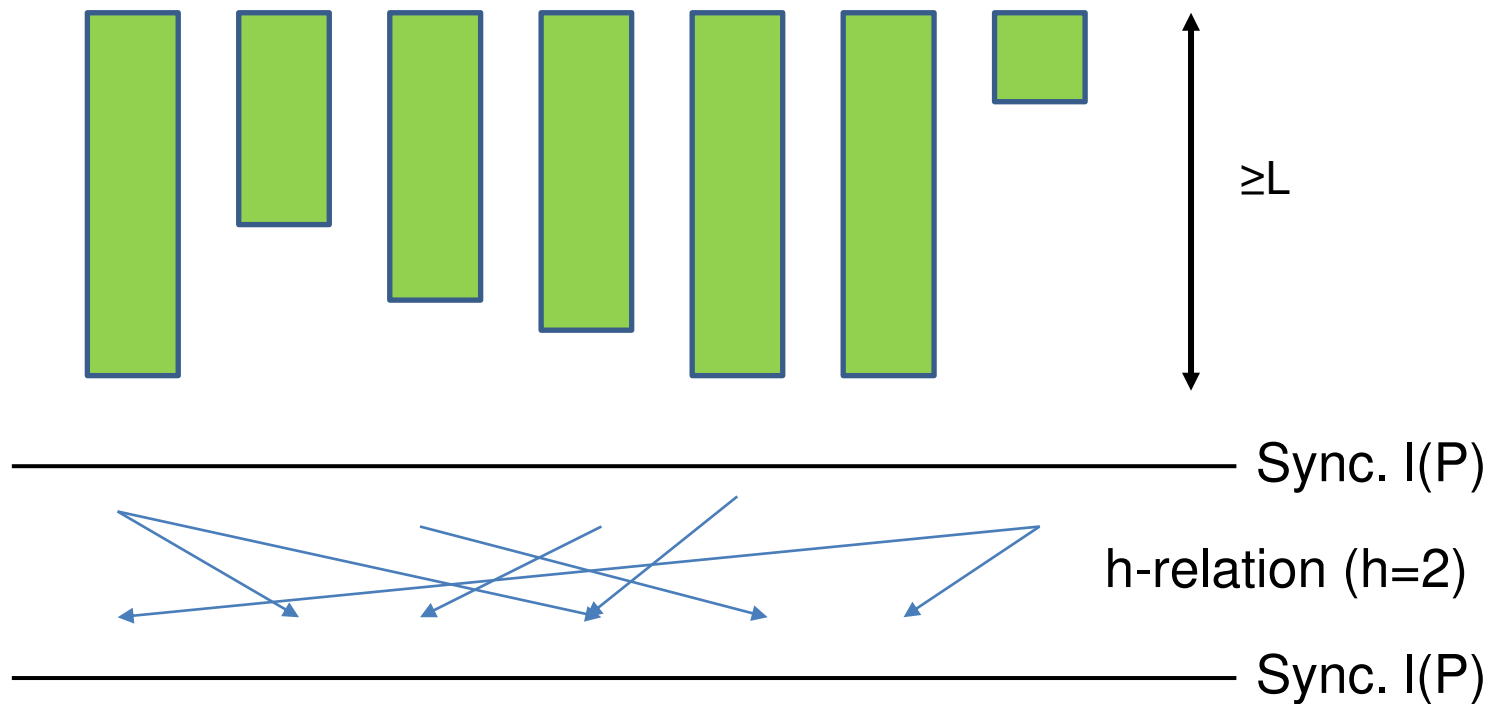
Data exchange operation in which each processor sends or receives at most  $h$  units of data (and at least one processor sends or receives  $h$  data units), data visible after synchronization

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





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:  $I(P)$

Total cost of algorithm:  $SI(P) + \sum_s W_s + \sum_s H_s$

Leslie G. Valiant: A Bridging Model for Parallel Computation.  
Commun. ACM 33(8): 103-111 (1990)

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

Rob. H. Bisseling: Parallel Scientific Computation. A structured approach using BSP and MPI. Oxford University Press, 2004 (reprint 2010)

## 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:

1-ported

In a fully connected, 1-ported network, broadcast takes at least  $\text{ceil}(\log_2 p)$  communication rounds, and time

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

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  $j$ ,  $j=0, 1, \dots$ , 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:

Easy generalization: k-ported

In a fully connected, k-ported network, broadcast takes at least  $\text{ceil}(\log_{k+1} p)$  communication rounds, and time

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

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,  $j=0, 1, \dots$ , 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

$$M-1 + \text{ceil}(\log_2 p)$$


Proof:

The root can possibly send  $M-1$  blocks in  $M-1$  rounds; the last block requires at least  $\text{ceil}(\log_2 p)$  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

$$T(m) = (\text{ceil}(\log p) - 1)\alpha + 2\sqrt{[\text{ceil}(\log p) - 1]\alpha\beta m} + \beta m$$

  
 $o(m)$  extra  
 latency

  
 “Full, optimal  
 bandwidth”

Proof: See pipeline lemma (but also next slide)

By lower bound, best time for  $M$  blocks is

$$\begin{aligned} T(m,M) &= (M-1+\log p)(\alpha+\beta m/M) = \\ &(\log p - 1)\alpha + M\alpha + (\log p - 1)\beta m/M + M\beta m/M = \\ &(\log p - 1)\alpha + M\alpha + (\log p - 1)\beta m/M + \beta m \end{aligned}$$

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

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

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:  $\alpha d + \beta m$

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  $\text{Bisec}(G)$ . Solving the regular alltoall problem requires at least  $\beta m |V/2|^2 / \text{Bisec}(G)$  time.

Proof:

Partition  $V$  into two subsets  $V_1$  and  $V_2$  of size  $|V/2|$ . The time to send and receive all data for all processors in either subset is at least  $\beta m |V/2| |V/2|$  divided by the capacity of  $\text{cut}(V_1, V_2)$ . This in particular holds for the cut in  $\text{Bisec}(G)$ .

## Before/after semantics of the (MPI) collectives

Input: Vector(s)  $x$  of elements,  $x_0, x_1, \dots$

Output: Vector(s) of elements,  $x_0, x_1, \dots$

Collectives:

Broadcast, Gather/Scatter, Allgather, Alltoall, Reduce,  
Allreduce, Reduce-scatter, Scan

**0                      1                      2                      3**

Broadcast: before

x

Scatter: before

x0  
x1  
x2  
x3

Gather: before

x0  
x1  
x2  
x3

**0                      1                      2                      3**

Broadcast: after

x                      x                      x                      x

Scatter: after

x0  
x1  
x2  
x3

Gather: after

x0  
x1  
x2  
x3

0	1	2	3
Allgather: before			
x0			
	x1		
		x2	
			x3

Alltoall: before			
0x0	1x0	2x0	3x0
0x1	1x1	2x1	3x1
0x2	1x2	2x2	3x1
0x3	1x3	2x3	3x3

0	1	2	3
Allgather: after			
x0	x0	x0	x0
x1	x1	x1	x1
x2	x2	x2	x2
x3	x3	x3	x3

Alltoall: after			
0x0	0x1	0x2	0x3
1x0	1x1	1x2	1x3
2x0	2x1	2x2	2x3
3x0	3x1	3x2	3x3

0	1	2	3
Reduce: before			
x0	x1	x2	x3
Allreduce: before			
x0	x1	x2	x3
Reducescatter: before			
x0	x0	x0	x0
x1	x1	x1	x1
x2	x2	x2	x2
x3	x3	x3	x3
Scan/exscan: before			
x0	x1	x2	x3

0	1	2	3
Reduce: after			
$\sum x_i$			
Allreduce: after			
$\sum x_i$	$\sum x_i$	$\sum x_i$	$\sum x_i$
Reducescatter: after			
$\sum x_0$			
	$\sum x_1$		
		$\sum x_2$	
			$\sum x_3$
Scan/exscan: after			
y0	y1	y2	y3

Scan:  $y_i = \sum_{(0 \leq j \leq i)} x_j$

Exscan:  $y_i = \sum_{(0 \leq j < i)} x_j$ , no  $y_0$

## Observations

Gather and Scatter are “dual” operations

Scatter: before

x0  
x1  
x2  
x3

Scatter: after

x0  
x1  
x2  
x3

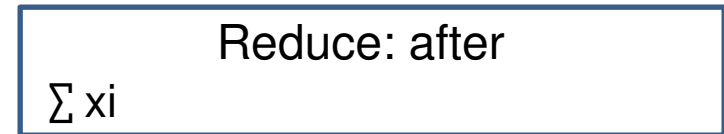
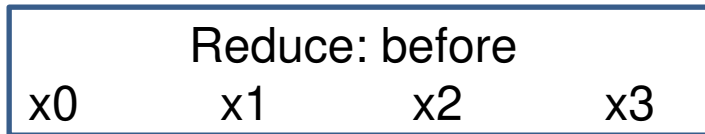
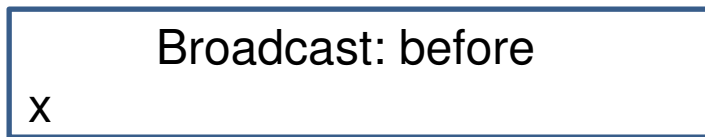
Gather: before

x0  
x1  
x2  
x3

Gather: after

x0  
x1  
x2  
x3

Broadcast and Reduce are “dual” operations





Alltoall  $\approx$  pxp matrix-transpose

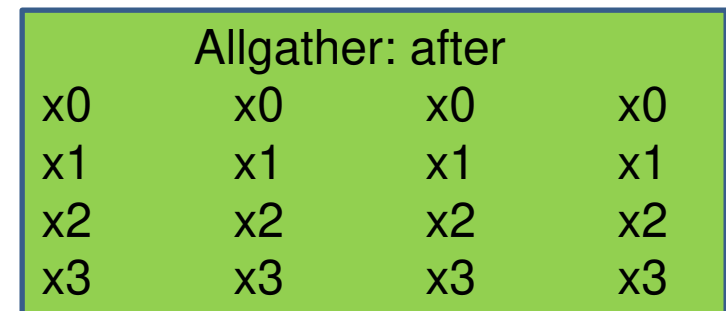
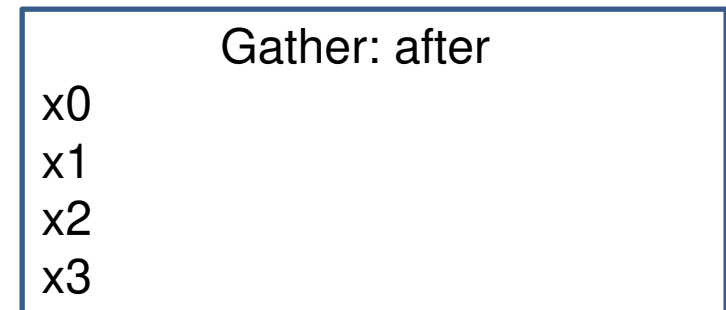
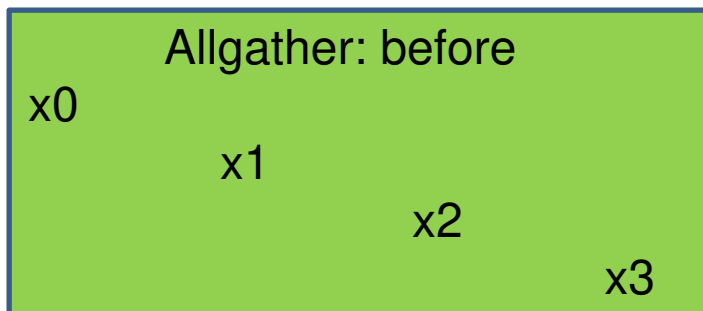
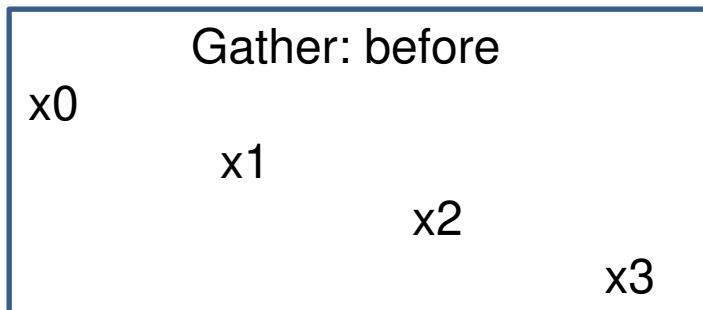
Alltoall: before

0x0	1x0	2x0	3x0
0x1	1x1	2x1	3x1
0x2	1x2	2x2	3x1
0x3	1x3	2x3	3x3

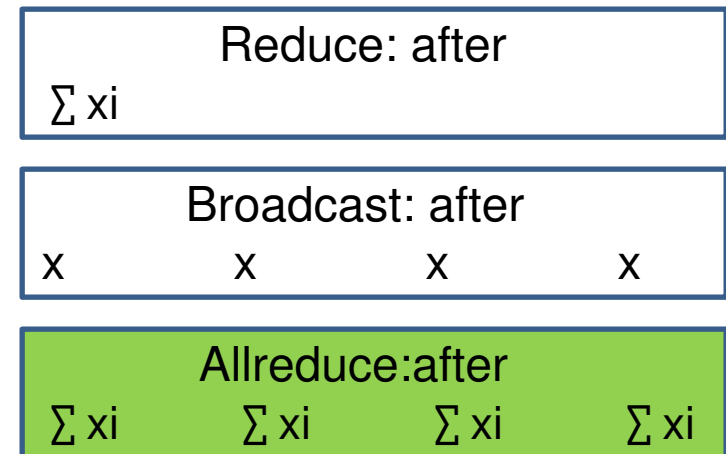
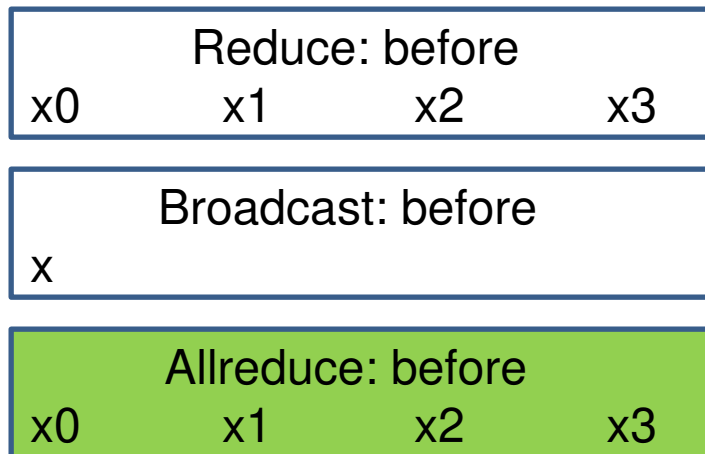
Alltoall: after

0x0	0x1	0x2	0x3
1x0	1x1	1x2	1x3
2x0	2x1	2x2	2x3
3x0	3x1	3x2	3x3

Allgather  $\approx$  Gather + Broadcast



Allreduce  $\approx$  Reduce + Broadcast



Reducescatter  $\approx$  Reduce + Scatter

Reduce: before			
x0	x1	x2	x3

Scatter: before			
x0			
x1			
x2			
x3			

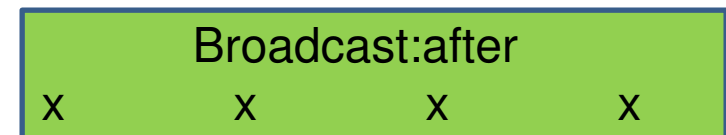
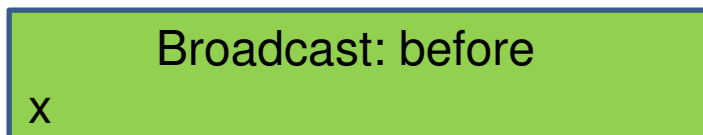
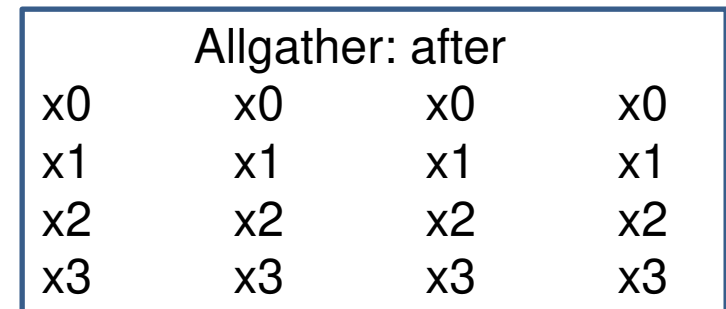
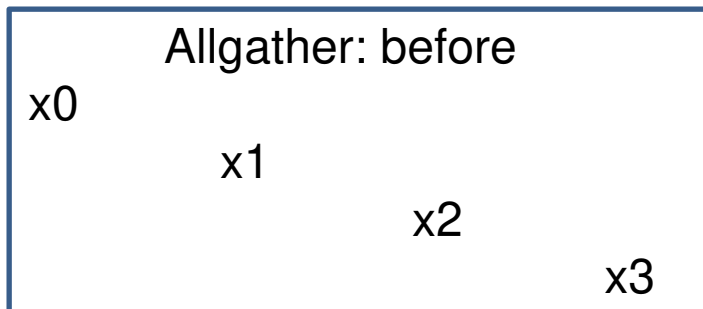
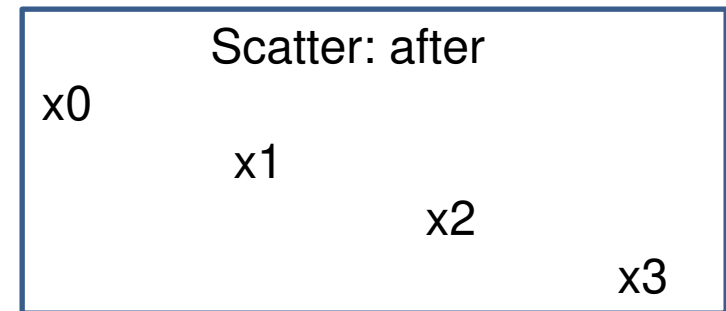
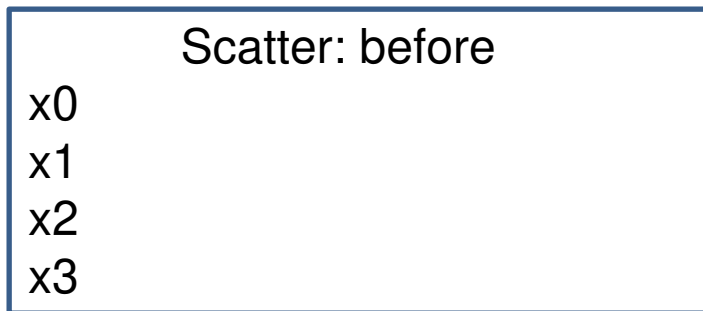
Reducescatter: before			
x0	x0	x0	x0
x1	x1	x1	x1
x2	x2	x2	x2
x3	x3	x3	x3

Reduce: after			
$\sum x_i$			

Scatter: after			
x0			
	x1		
		x2	
			x3

Reducescatter: after			
$\sum x_0$			
	$\sum x_1$		
		$\sum x_2$	
			$\sum x_3$

Broadcast  $\approx$  Scatter + Allgather



Allreduce  $\approx$  Reducescatter + Allgather

Reducescatter: before

x0	x0	x0	x0
x1	x1	x1	x1
x2	x2	x2	x2
x3	x3	x3	x3

Reducescatter: after

$\Sigma x0$			
	$\Sigma x1$		
		$\Sigma x2$	
			$\Sigma x3$

Allgather: before

x0			
	x1		
		x2	
			x3

Allgather: after

x0	x0	x0	x0
x1	x1	x1	x1
x2	x2	x2	x2
x3	x3	x3	x3

Allreduce: before

x0	x1	x2	x3
----	----	----	----

Allreduce: after

$\Sigma x_i$	$\Sigma x_i$	$\Sigma x_i$	$\Sigma x_i$
--------------	--------------	--------------	--------------

Reduce  $\approx$  Reducescatter + Gather

Reducescatter: before

x0	x0	x0	x0
x1	x1	x1	x1
x2	x2	x2	x2
x3	x3	x3	x3

Reducescatter: after

$\Sigma x0$			
	$\Sigma x1$		
		$\Sigma x2$	
			$\Sigma x3$

Gather: before

x0			
	x1		
		x2	
			x3

Gather: after

x0
x1
x2
x3

Reduce: before

x0	x1	x2	x3
----	----	----	----

Reduce:after

$\Sigma x_i$
--------------

Allgather  $\approx \parallel_{0 \leq i < p} \text{Broadcast}(i)$

Broadcast(0): before
x0

Broadcast(1): before
x1

Broadcast(2): before
x2

Allgather: before			
x0			
	x1		
		x2	
			x3

...

Broadcast(0): after			
x0	x0	x0	x0

Broadcast(1): after			
x1	x1	x1	x1

Broadcast(2): after			
x2	x2	x2	x2

Allgather: after			
x0	x0	x0	x0
x1	x1	x1	x1
x2	x2	x2	x2
x3	x3	x3	x3



Alltoall  $\approx \parallel_{0 \leq i < p} \text{Scatter}(i)$

Scatter(i): before

ix0  
ix1  
ix2  
ix3

Scatter(i): after

ix0      ix1      ix2      ix3

Alltoall: before

0x0	1x0	2x0	3x0
0x1	1x1	2x1	3x1
0x2	1x2	2x2	3x1
0x3	1x3	2x3	3x3

Alltoall: after

0x0	0x1	0x2	0x3
1x0	1x1	1x2	1x3
2x0	2x1	2x2	2x3
3x0	3x1	3x2	3x3

Alltoall  $\approx \parallel_{0 \leq i < p} \text{Gather}(i)$

Gather(i): before

0xi	1xi	2xi	3xi
-----	-----	-----	-----

Gather(i): after

0xi
1xi
2xi
3xi

Alltoall: before

0x0	1x0	2x0	3x0
0x1	1x1	2x1	3x1
0x2	1x2	2x2	3x1
0x3	1x3	2x3	3x3

Alltoall: after

0x0	0x1	0x2	0x3
1x0	1x1	1x2	1x3
2x0	2x1	2x2	2x3
3x0	3x1	3x2	3x3

## MPI collective interfaces (reminder): Regular

Bcast:

```
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

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

```
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:

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

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

## Reduce-scatter:

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

## Scan, exscan:

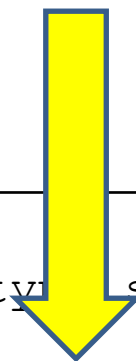
```
MPI_Scan(void *sendbuf, void *recvbuf,  
         int count, MPI_Datatype datatype,  
         MPI_Op op, MPI_Comm comm)
```

```
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:



```
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:

```
MPI_Scatterv(void *sendbuf,  
             int sendcounts[], int senddispls[],  
             MPI_Datatype sendtype,  
             void *recvbuf,  
             int recvcount, MPI_Datatype recvtype,  
             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:


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

## Alltoall:

```
MPI_Alltoallv(void *sendbuf,  
              int sendcounts[], int senddispls[],  
              MPI_Datatype sendtype,  
              void *recvbuf,  
              int recvcounts[], int recvdispls[],  
              MPI_Datatype recvtype,  
              MPI_Comm comm)
```

Alltoall:

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

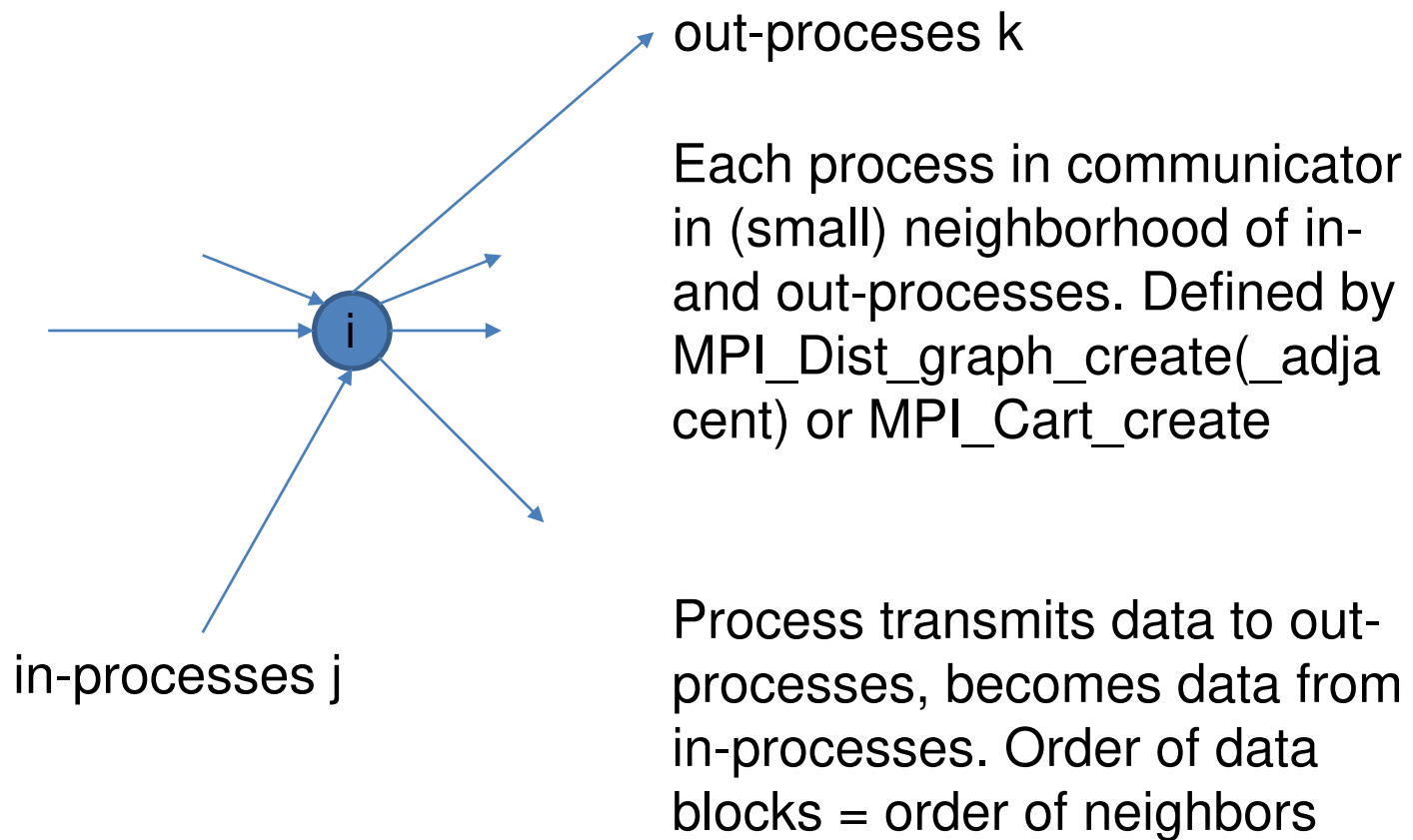


Note: Only collective where each block may have individual structure (signatures must match pairwise)

## Reduce-scatter:

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

## MPI sparse collective interfaces



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

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

Note: Same signature as standard collective counterparts



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

```
MPI_Neighbor_alltoallv(void *sendbuf,  
                      int sendcounts[], int senddispls[],  
                      MPI_Datatype sendtype,  
                      void *recvbuf,  
                      int recvcounts[], int recvdisepls[],  
                      MPI_Datatype recvtype,  
                      MPI_Comm comm)
```

```
MPI_Neighbor_alltoallw(void *sendbuf,  
                      int sendcounts[], int senddispls[],  
                      MPI_Datatype sendtypes[],  
                      void *recvbuf,  
                      int recvcounts[], int recvdisepls[],  
                      MPI_Datatype recvtypes[],  
                      MPI_Comm comm)
```

Torsten Hoefler, Rolf Rabenseifner, Hubert Ritzdorf, Bronis R. de Supinski, Rajeev Thakur, Jesper Larsson Träff: The scalable process topology interface of MPI 2.2. *Concurr. Comput. Pract. Exp.* 23(4): 293-310 (2011)

Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, Daniel J. Holmes: MPI collective communication through a single set of interfaces: A case for orthogonality. *Parallel Comput.* 107: 102826 (2021)

## 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 4.0 has persistent collectives. Lead to more blow-up!

## Questions?

- Why is there an MPI\_Allgather ( $\approx$  MPI\_Gather+MPI\_Bcast) in MPI?
- Why is there an MPI\_Allreduce ( $\approx$  MPI\_Reduce+MPI\_Bcast) in MPI?
- Why is there an MPI\_Reduce\_scatter ( $\approx$  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  $\text{MPI\_Allgather} \leq \text{MPI\_Gather} + \text{MPI\_Bcast}$ ; if not, implementation is bad (broken)

## MPI collectives and algorithm design

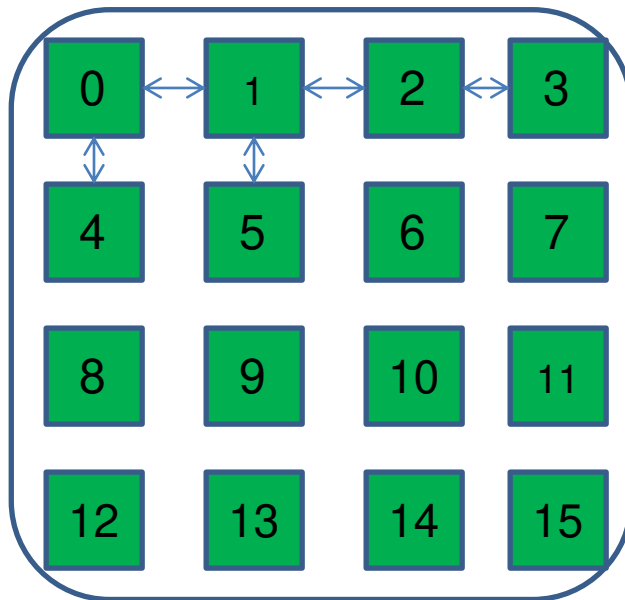
- Many specific requirements (arbitrary roots, datatypes, mapping in network, ...)
- Any MPI communicator allows (sendrecv) communication between any pair of processes: **Virtually fully connected**
- **Underlying (routing) system (software and hardware) should ensure good (homogeneous) performance between any process pair, under any communication pattern**

### Approach:

- Implement algorithms with send-recv 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

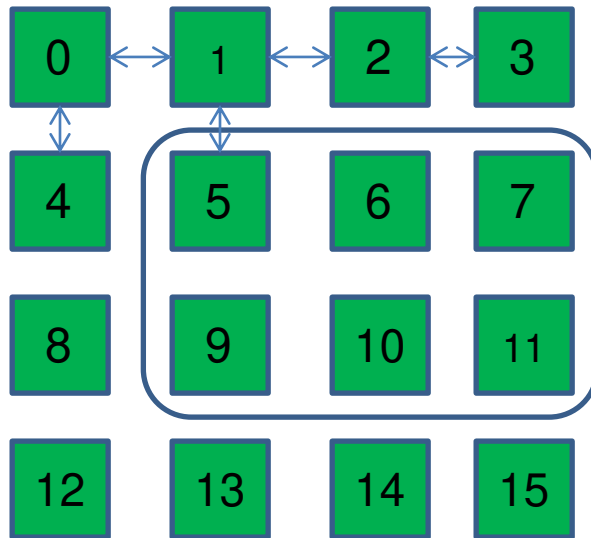


MPI comm (e.g.,  
MPI\_COMM\_WORLD): Must  
support all collectives

Torus algorithms

MPI is (too) powerful

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

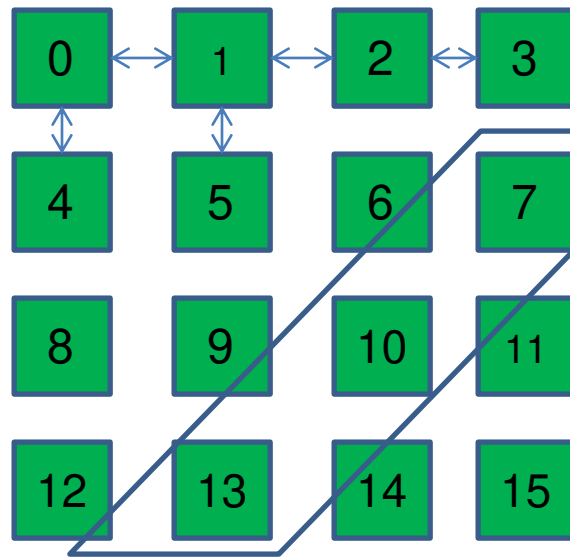


MPI comm (subcomm): Must support all collectives

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



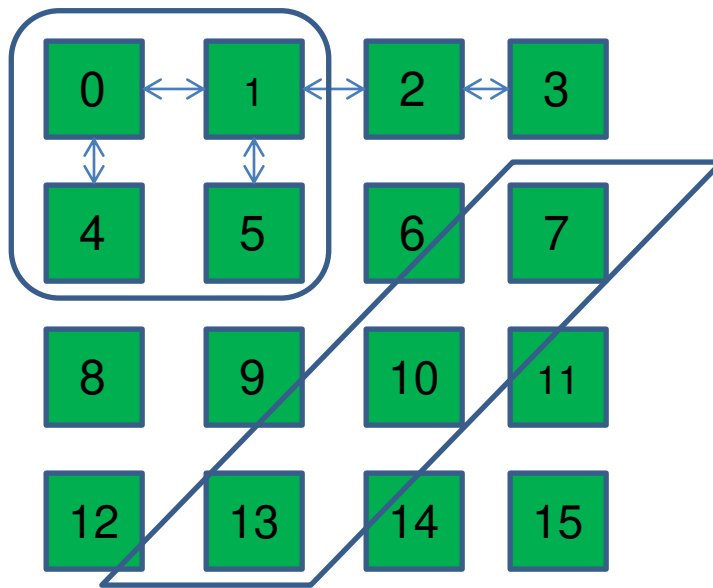
MPI comm (subcomm): Must support all collectives

Torus assumption does not hold



MPI is (too) powerful

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

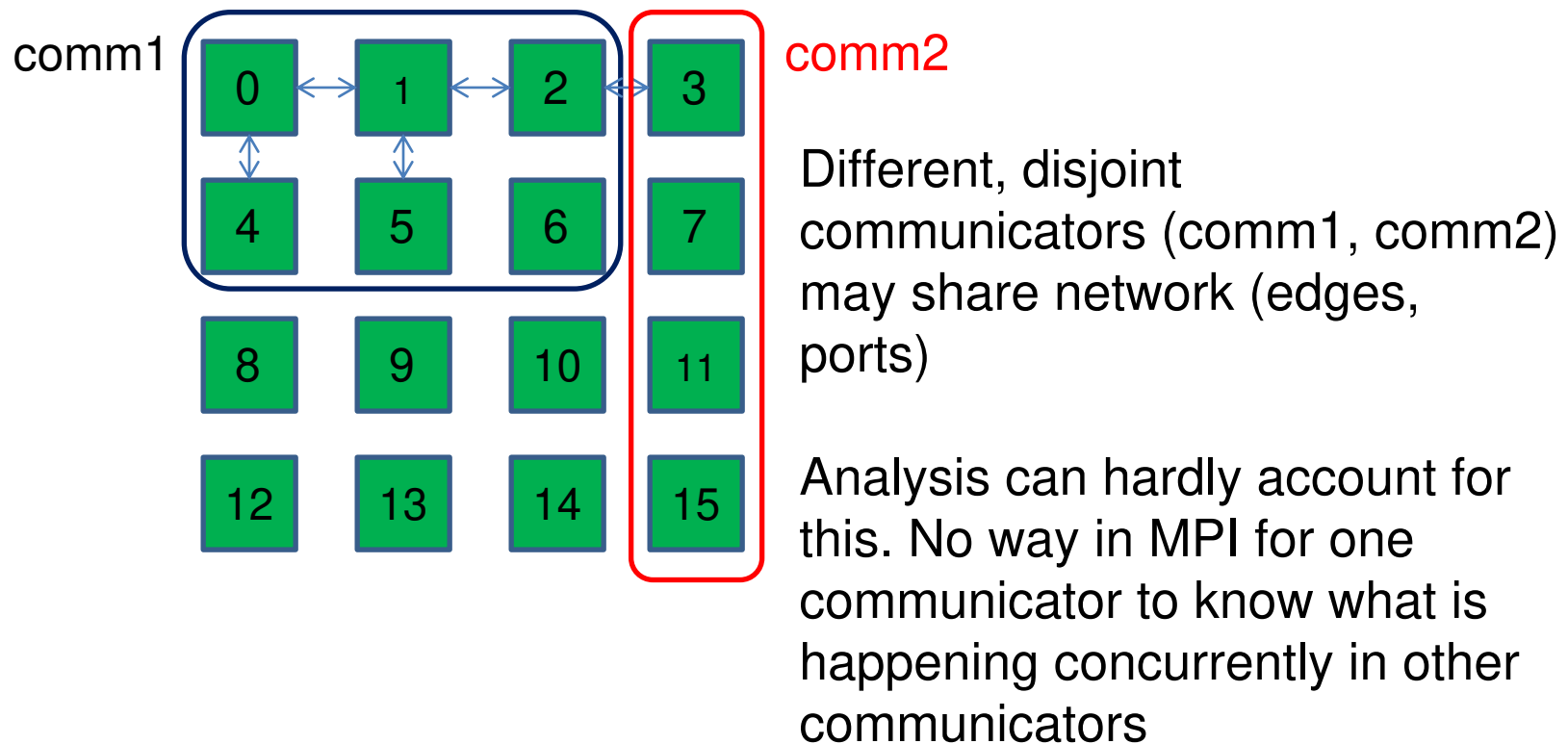


MPI comm (subcomm): Must support all collectives

Torus assumption does not hold

MPI is (too) powerful

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



## 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

# Application

start (synchronized)

MPI\_Bcast

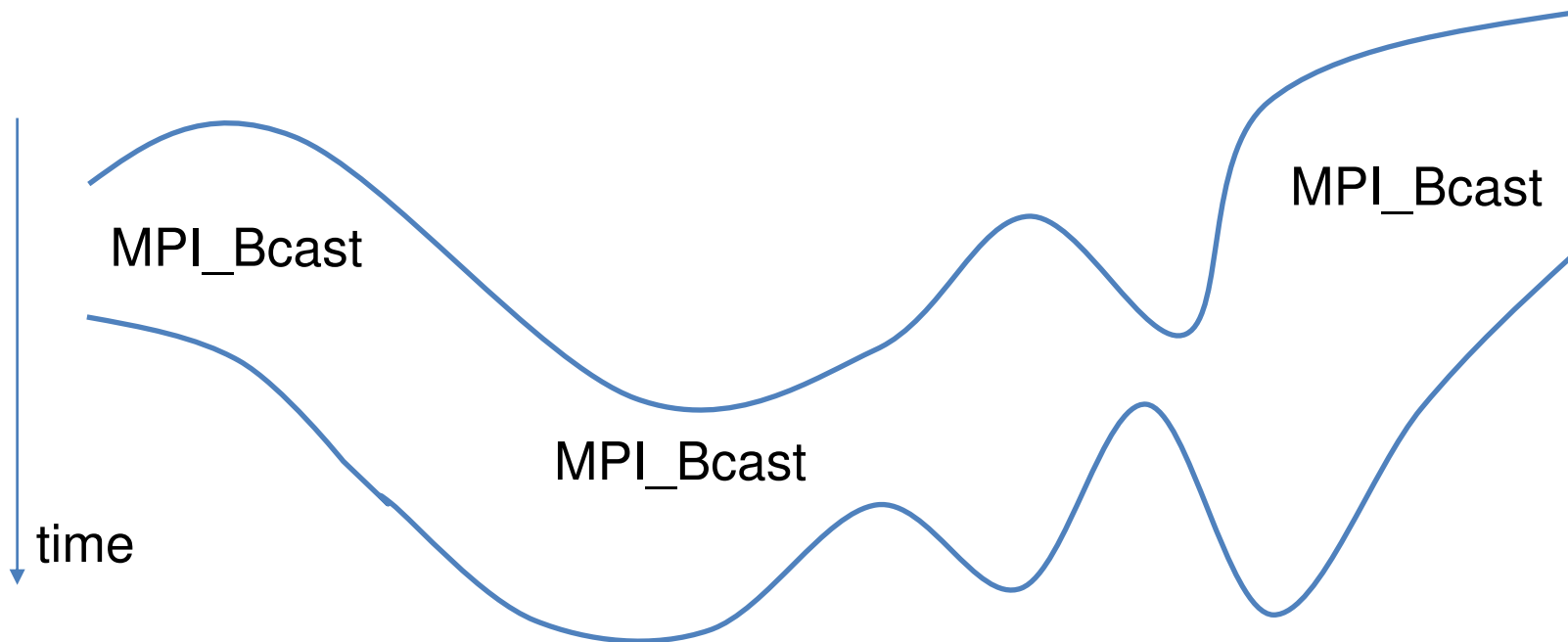
MPI\_Bcast

MPI\_Bcast

end (time is time of  
last process)

time

## Application



In actual execution of MPI application:

**No** requirement, no **guarantee** that all processes call collective operation at the same time. Often they do not! (trace application to find out)

## Questions :

How bad can the algorithms be under non-synchronized process arrival (and progress) patterns? How can algorithms be designed that are (provably) good under non-synchronized process arrival?

**Answer:** Not much known...

Not covered in these lectures

Research on sensitivity of MPI (collective) operations to

- Non-synchronized process arrival patterns
- “Noise” (OS)

still needed

Ahmad Faraj, Pitch Patarasuk, Xin Yuan: A Study of Process Arrival Patterns for MPI Collective Operations. International Journal of Parallel Programming 36(6): 543-570 (2008)

Petar Marendic, Jan Lemeire, Dean Vucinic, Peter Schelkens: A novel MPI reduction algorithm resilient to imbalances in process arrival times. J. Supercomput. 72(5): 1973-2013 (2016)

Fabrizio Petrini, Darren J. Kerbyson, Scott Pakin:  
The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8, 192 Processors of ASCI Q. SC 2003: 55

Torsten Hoefler, Timo Schneider, Andrew Lumsdaine: The Effect of Network Noise on Large-Scale Collective Communications. Parallel Processing Letters 19(4): 573-593 (2009)



## 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)

```
MPI_Bcast(...,comm);
```

Rank i:

```
MPI_Bcast(...,comm);
```

```
MPI_Bcast(...,comm);
```

... is legal

## 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)

```
MPI_Bcast(...,comm);  
MPI_Gather(comm);
```

Rank i:



```
MPI_Bcast(...,comm);  
MPI_Gather(comm);
```

```
MPI_Gather(comm);
```

```
MPI_Bcast(...,comm);
```

... is not

## 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

## Mixing MPI collectives with different completion semantics

## 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

E. Chan, M. Heimlich, A. Purkayastha, Robert A. van de Geijn:  
Collective communication: theory, practice, and experience.  
Concurrency and Computation: Practice and Experience 19(13):  
1749-1783 (2007)

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

**Geoffrey C. Fox** , Mark Johnson, Gregory Lyzenga,  
Steve Otto, John Salmon, and David Walker: Solving Problems on  
Concurrent Processors. Volume 1: General Techniques and Regular  
Problems. Prentice-Hall, 1988

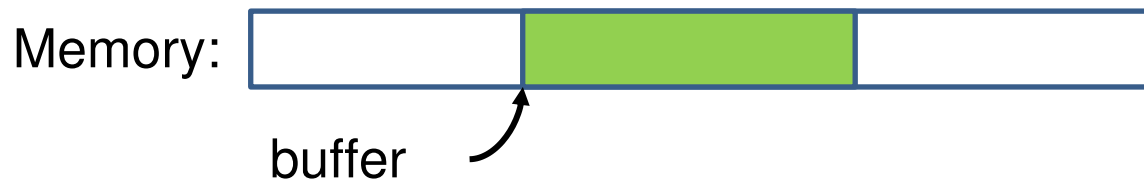
Before starting...

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

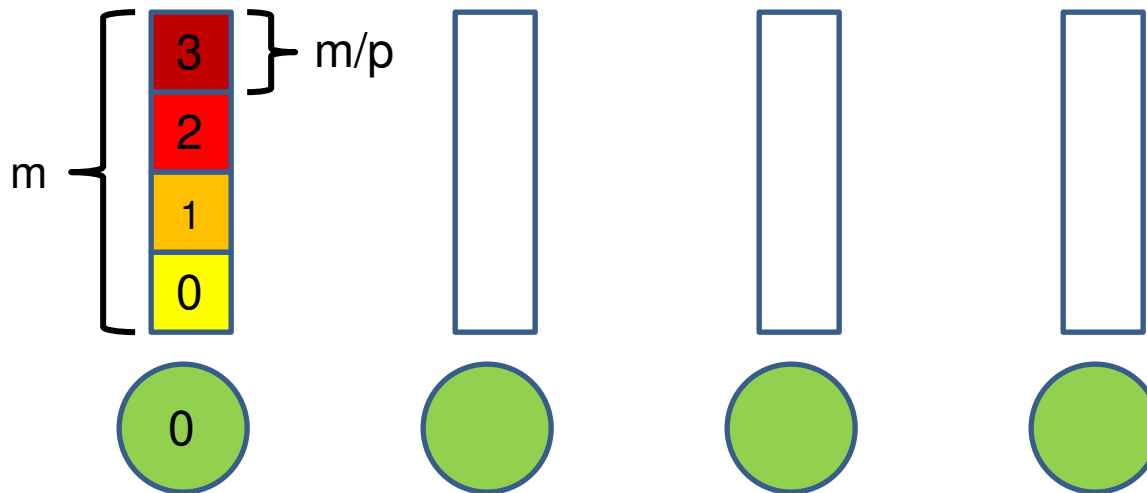
Get process rank  $i$ , and number of processes  $p$ . It always holds that  $0 \leq i < p$

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

Blocks for now are stored consecutively in some buffer



## 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);
```

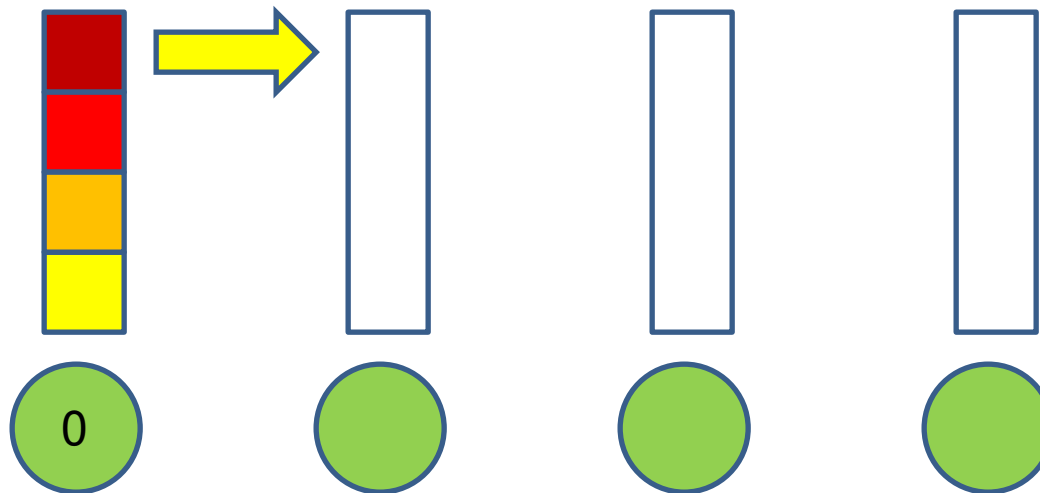


p consecutive blocks scattered from root

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

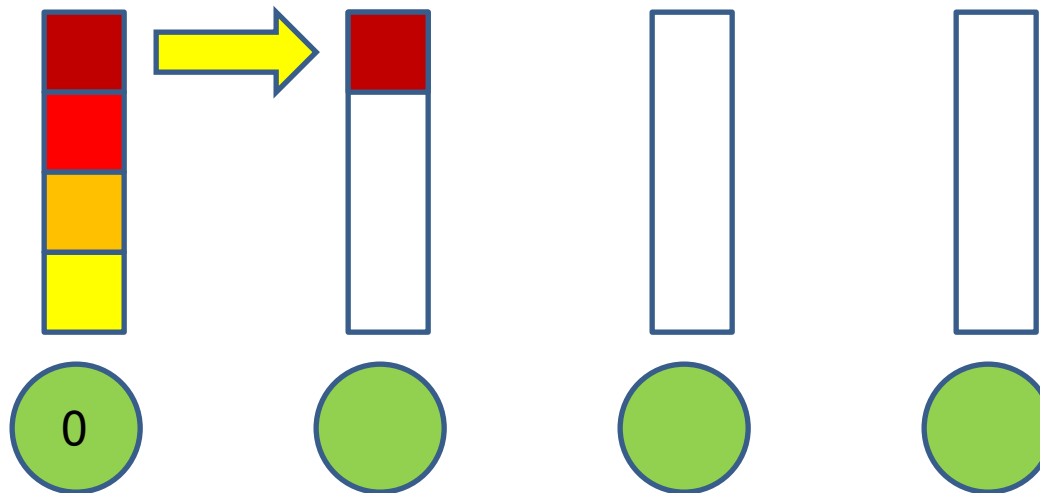


p triples: ( **recvbuf+recvdyspls[i]** ,recvcounts[i],recvtype)



Root (0):

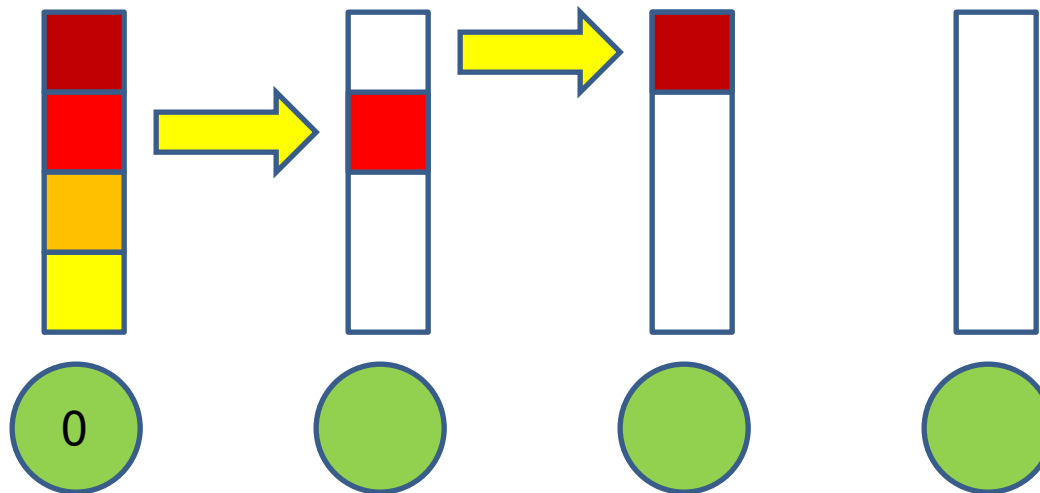
```
for (i=p-1; i>0; i--) {  
    MPI_Send(sendbuf[i], ..., root+1, comm);  
}
```



Non-root,  $0 < \text{rank} < \text{size}$ :

```
MPI_Recv(recvbuf, ...rank-1, ..., comm);
for (i=rank; i<p-1; i++) {
    MPI_Sendrecv_replace(recvbuf, ...,
                          rank-1, rank+1, ..., comm);
}
```

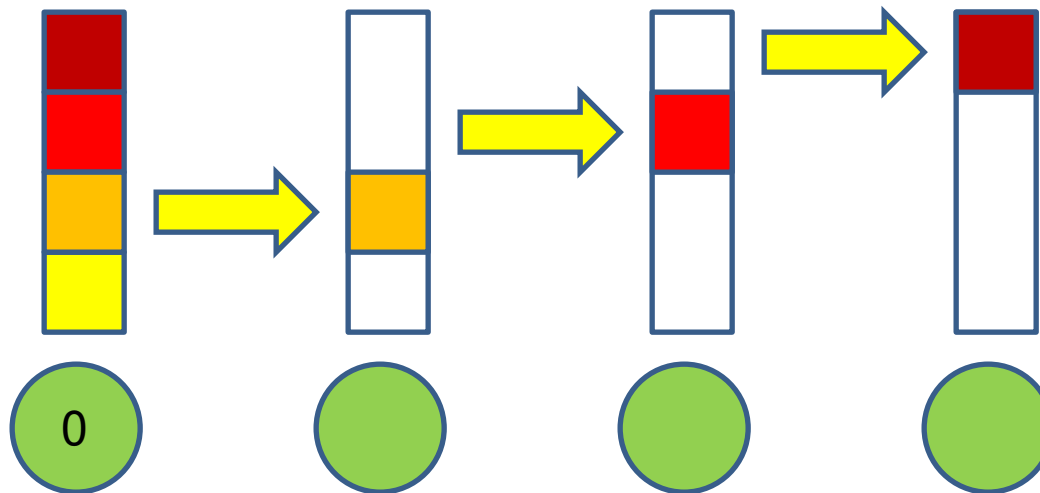
## 1-ported, bidirectional send-receive communication



Non-root,  $0 < \text{rank} < \text{size}$ :

```
MPI_Recv(recvbuf, ...rank-1, ..., comm);
for (i=rank; i<p-1; i++) {
    MPI_Sendrecv_replace(recvbuf, ...,
                          rank-1, rank+1, ..., comm);
}
```

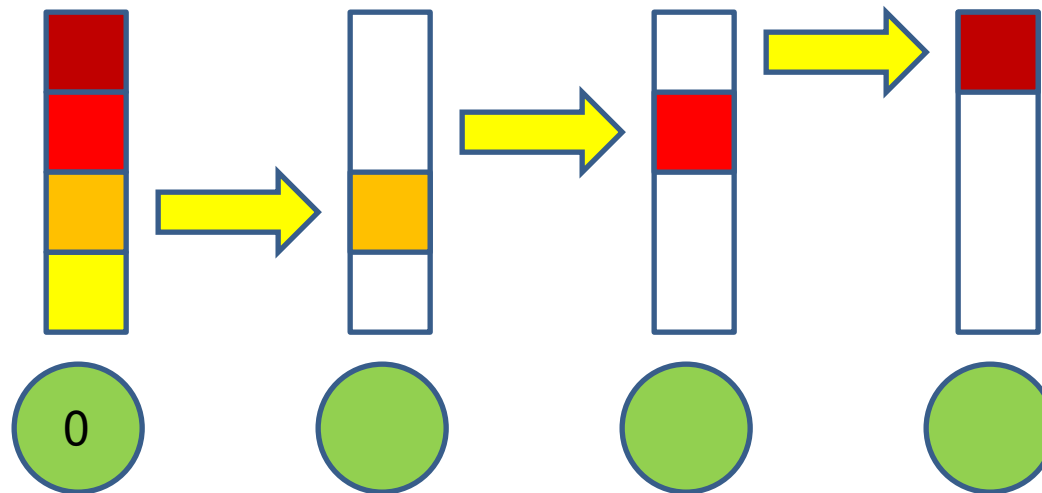
## 1-ported, bidirectional send-receive communication



Non-root,  $0 < \text{rank} < \text{size}$ :

```
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



$$T_{\text{scatter}}(m) = (p-1)(\alpha + \beta m/p) = (p-1)\alpha + (p-1)/p \beta m$$

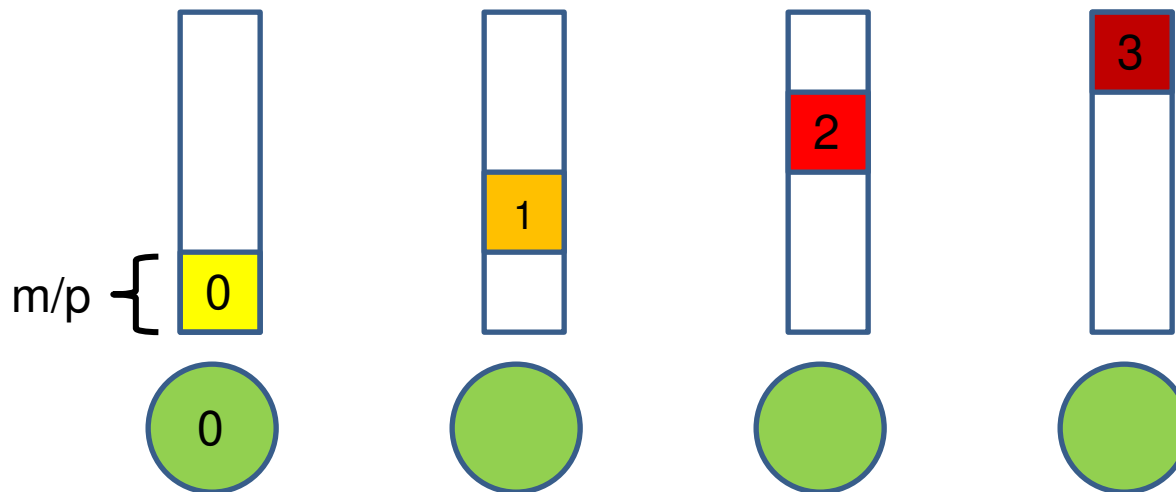
Non-optimal in  $\alpha$ -term

Optimal in  $\beta$ -term

in linear processor array with 1-ported, bidirectional,  
send-receive communication

©Jesper Larsson Träff

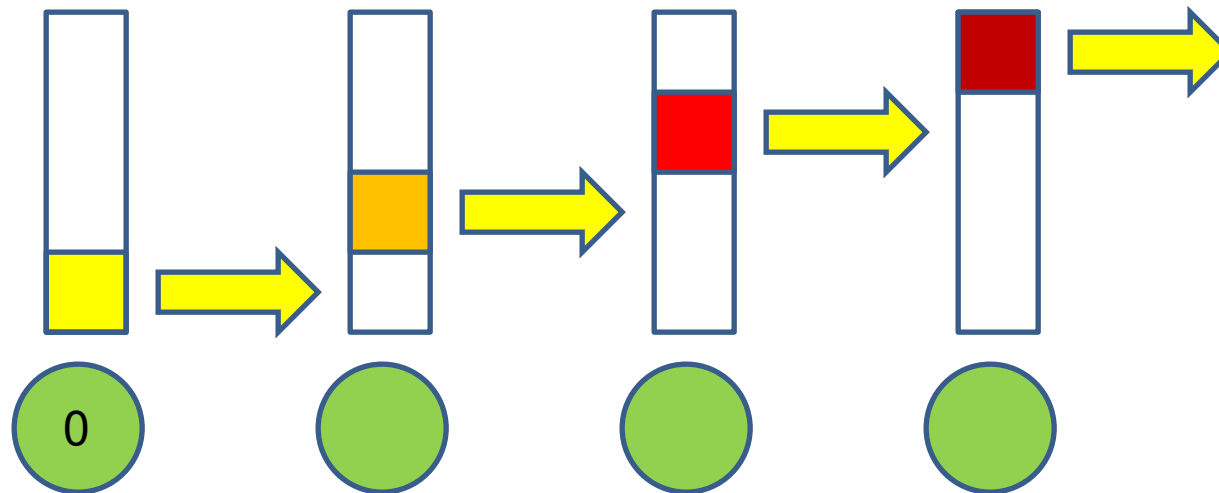
## Linear-ring allgather



**MPI:**  
All processes  
stores gathered  
blocks in rank  
order

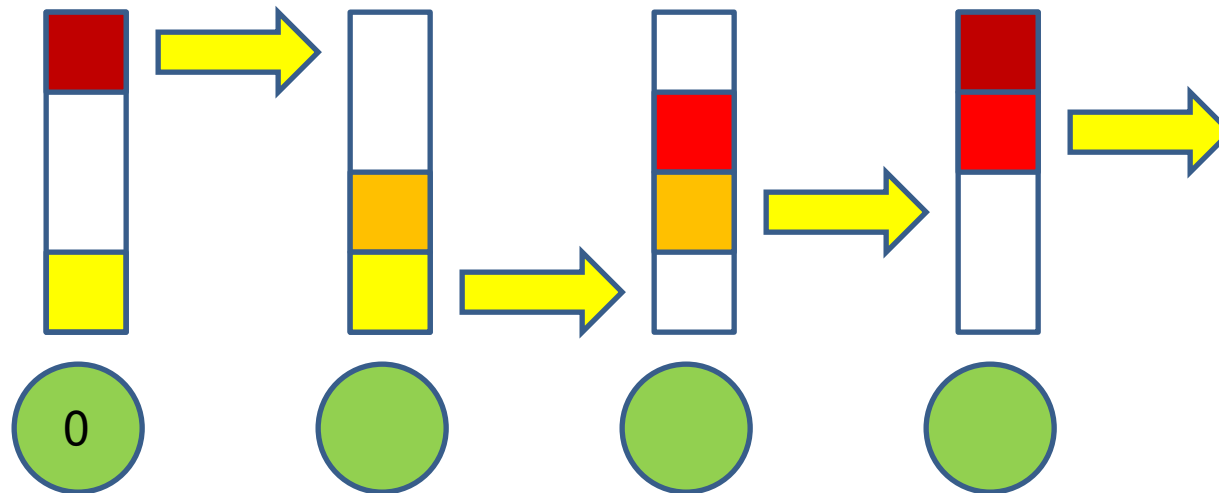
```
MPI_Allgather(sendbuf, scount, stype,  
              recvbuf, rcount, rtype, comm);
```

`scount/rcount`: number of elements in one block, each process  
contributes one block and gathers  $p-1$  blocks

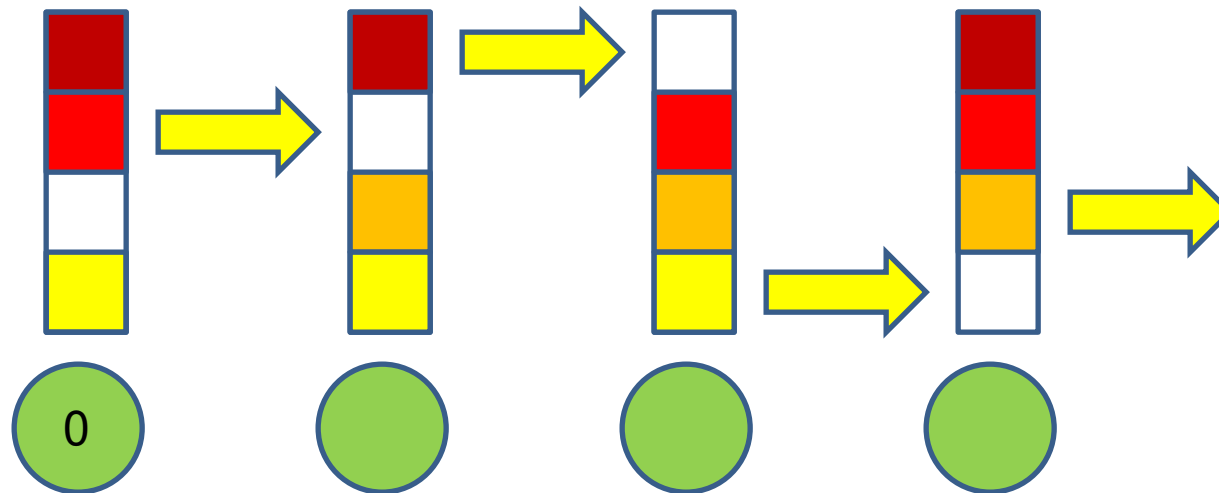


```
for (i=0; i<p-1; i++) {  
    MPI_Sendrecv(recvbuf[(rank-i+size)%size], ...,  
                 recvbuf[(rank-i-1+size)%size], ..., comm);  
}
```

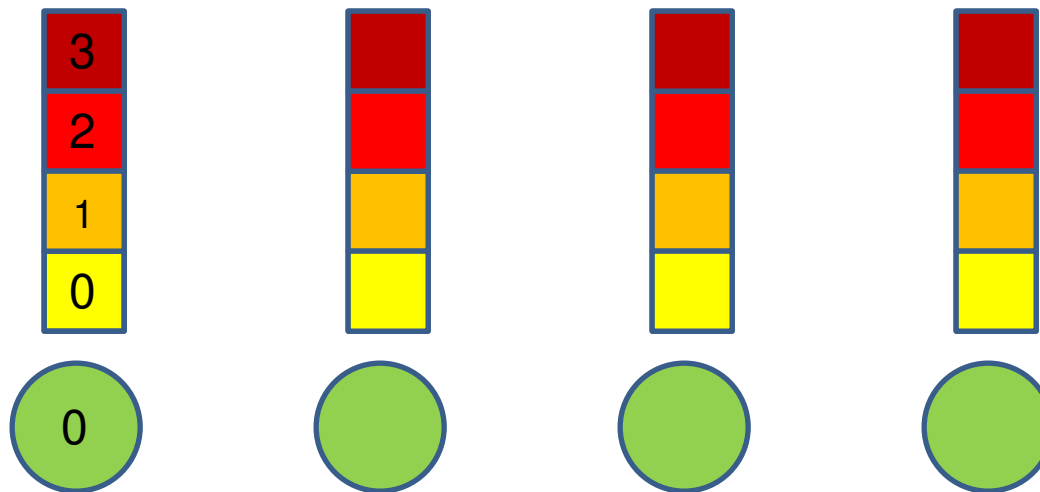




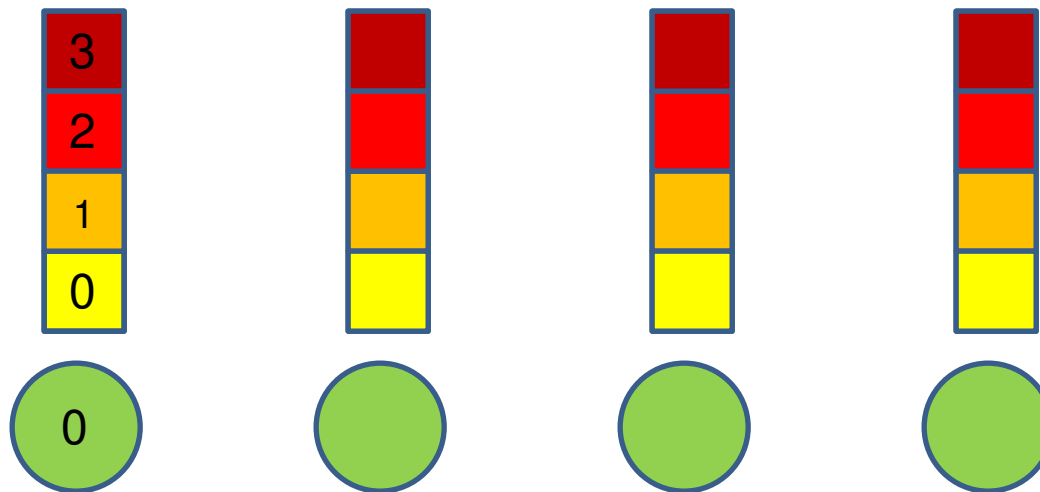
```
for (i=0; i<p-1; i++) {  
    MPI_Sendrecv(recvbuf[(rank-i+size)%size], ...,  
                 recvbuf[(rank-i-1+size)%size], ..., comm);  
}
```



```
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);
```

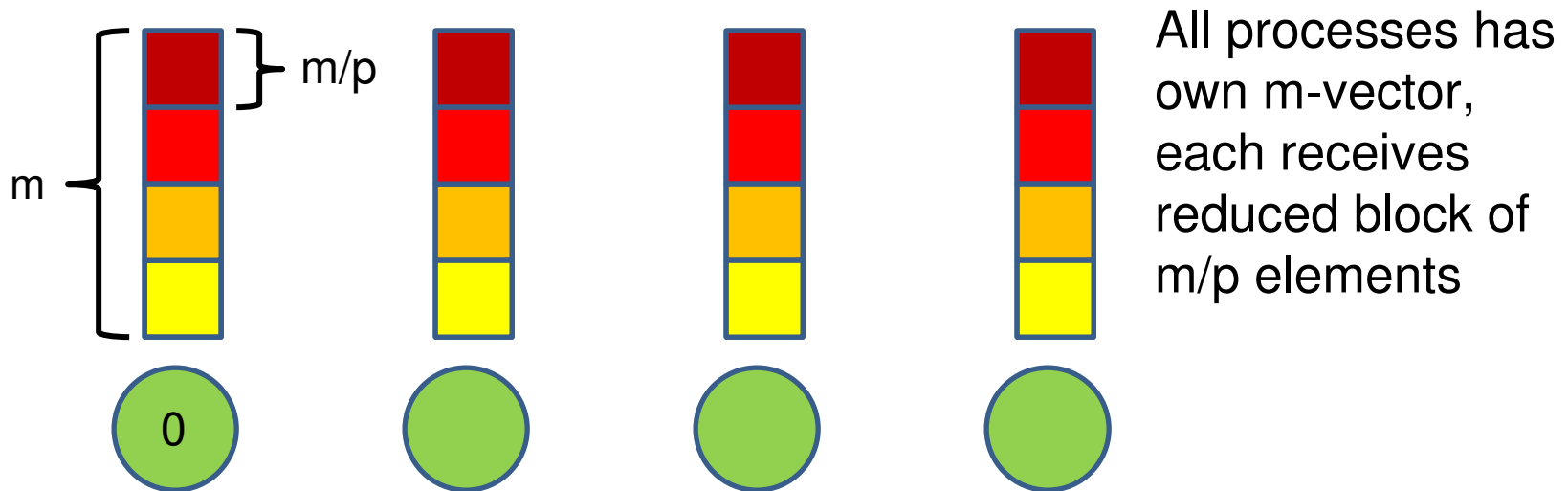
$$\begin{aligned} T_{\text{broadcast}}(m) &= T_{\text{scatter}}(m) + T_{\text{allgather}}(m) \\ &= 2(p-1)\alpha + 2(p-1)/p \beta m \end{aligned}$$

by Broadcast  $\approx$  Scatter + Allgather observation

Factor 2 from optimal in  $\beta$ -term

But major improvement over trivial algorithm that sends the  $m$  elements along the ring:  $T_{\text{broadcast}}(m) = (p-1)(\alpha + \beta m)$

## Linear-ring reduce-scatter



```
MPI_Reduce_scatter_block(sendbuf,  
                           resultbuf, count, datatype,  
                           op, comm) ;
```

## Aside: Requirements for MPI reduction collectives

- op one of MPI\_SUM (+), MPI\_PROD (\*), MPI\_BAND, MPI\_LAND, MPI\_MAX, MPI\_MIN, ...
- Special: 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

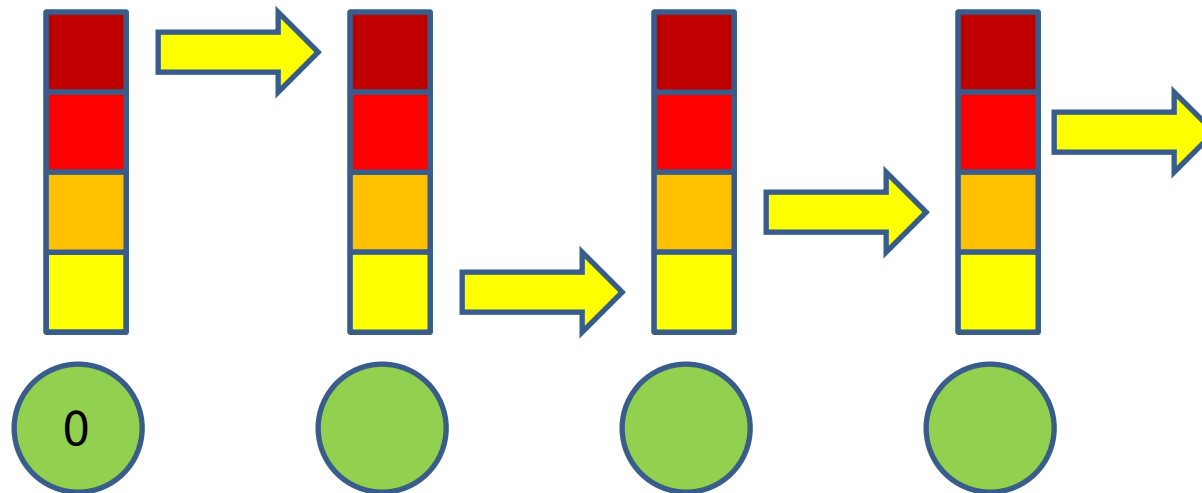
Let  $X_i$  be the vector contributed by MPI process  $i$

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

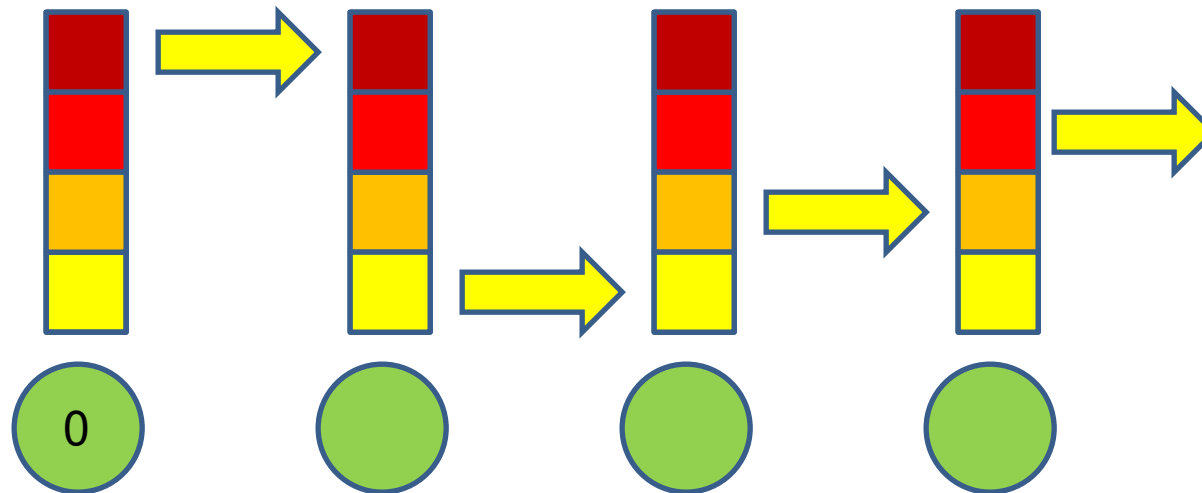
$$((X_0+X_1)+(X_2+X_3))+((X_4+X_5)+(X_6+X_7))$$

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
}
```

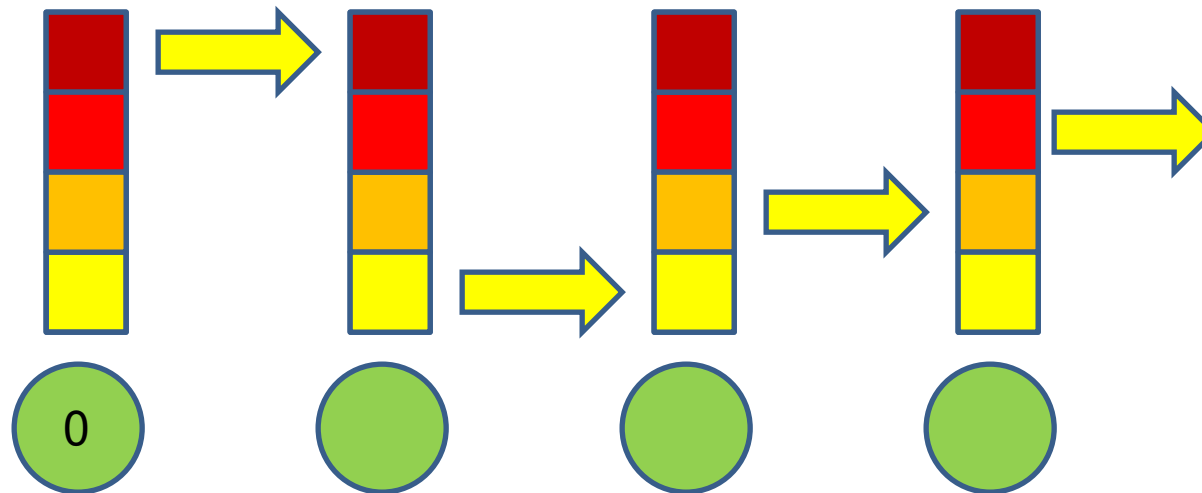


```
recvbuf[ri] = tmp OP recvbuf[ri]; // do MPI op
```

Not originally in MPI

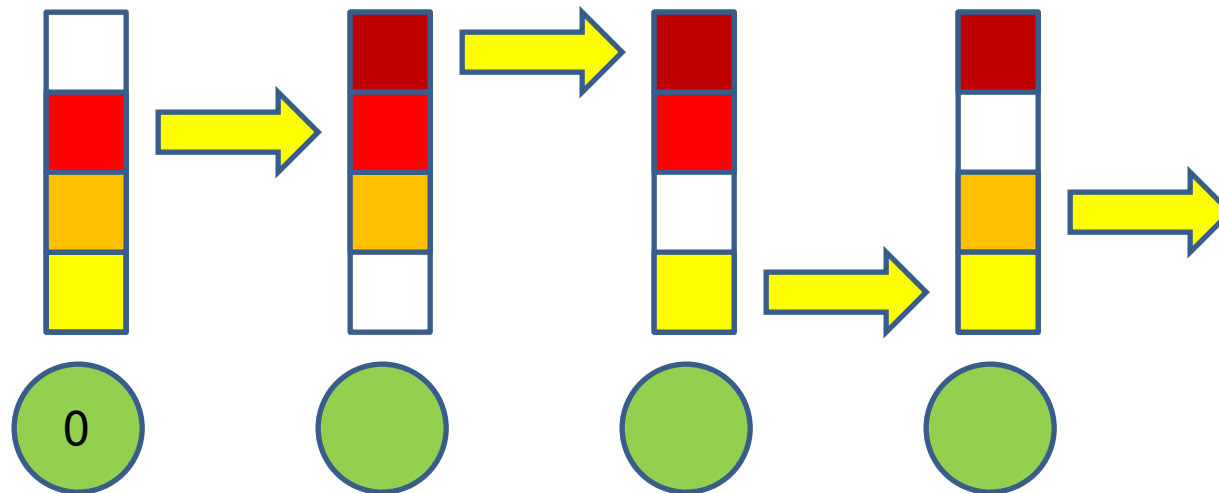
Need for (MPI 2.2 function):

```
MPI_Reduce_local(in, inout, count, datatype, op);
```



```
recvbuf[ri] = tmp OP recvbuf[ri]; // do MPI op
```

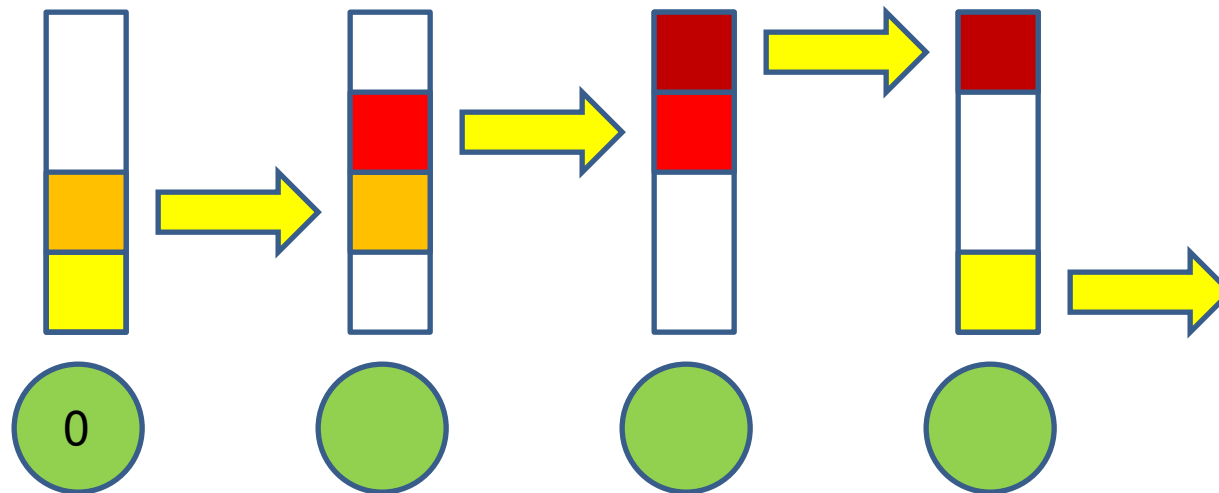
**Note :** Technically, it was not possible to implement MPI\_Reduce\_scatter algorithms on top of MPI before MPI 2.2



```

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
}

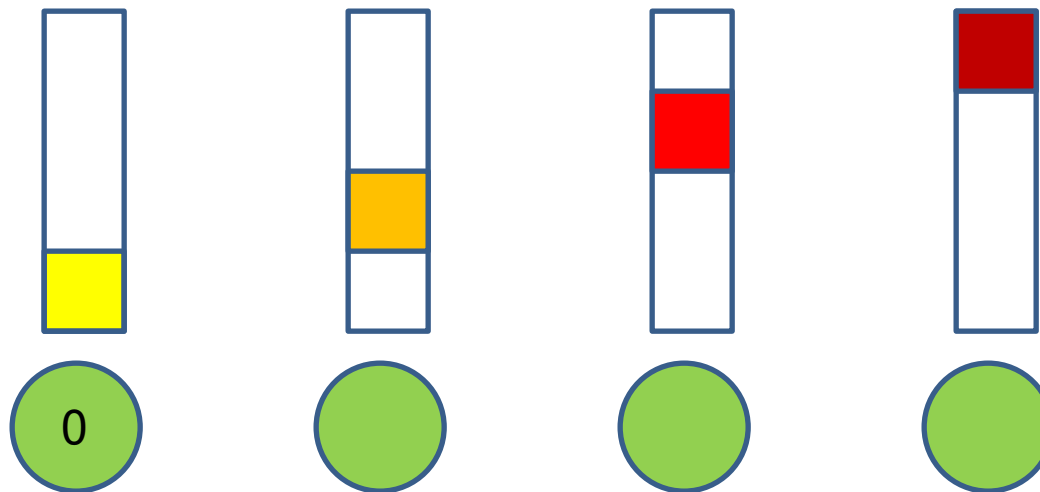
```



```

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
}

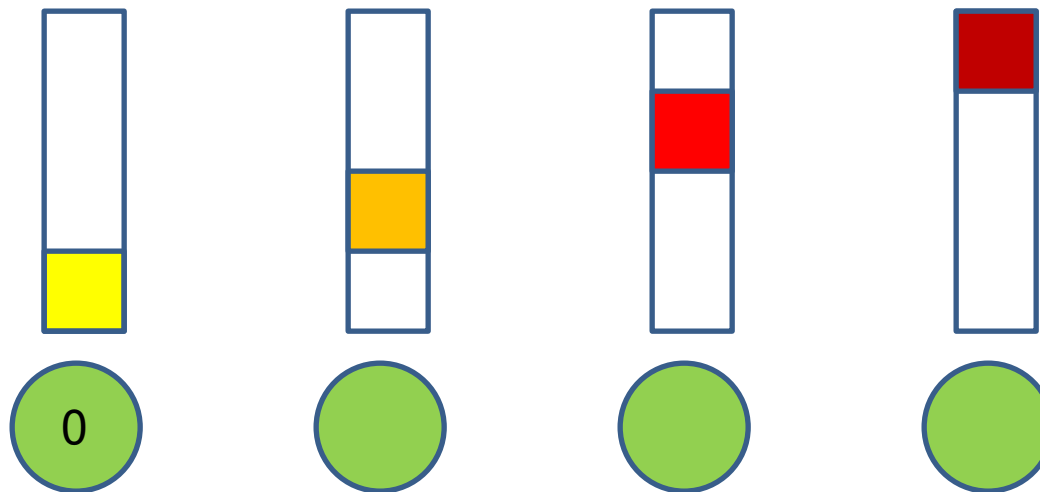
```



```

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
}

```

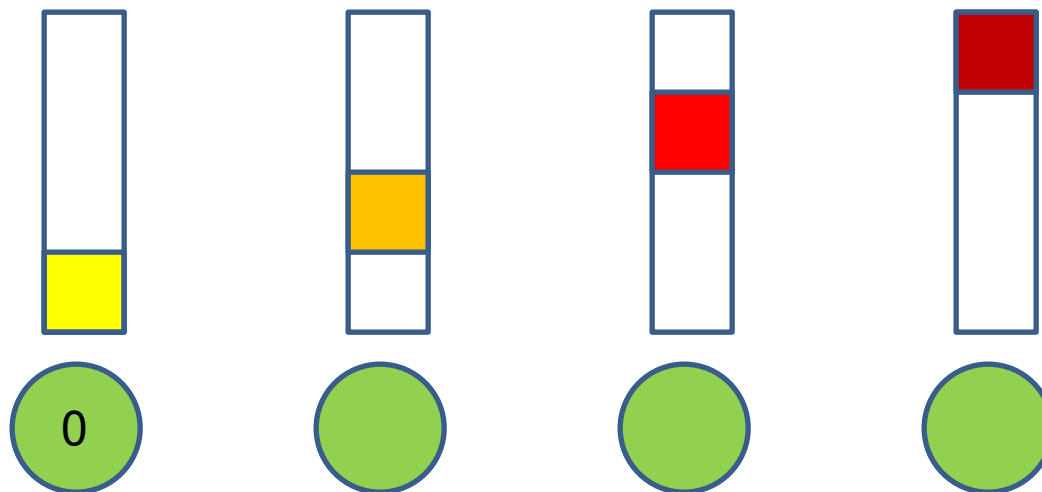


$p-1$  rounds  $i=1,2,\dots,p-1$ ; after round  $i$ , process rank has computed  $\sum_{\text{rank}-i \leq j \leq \text{rank}} x[\text{rank}-1-i]$

**But:** Exploits commutativity of +

Not suited for all MPI op's/datatypes

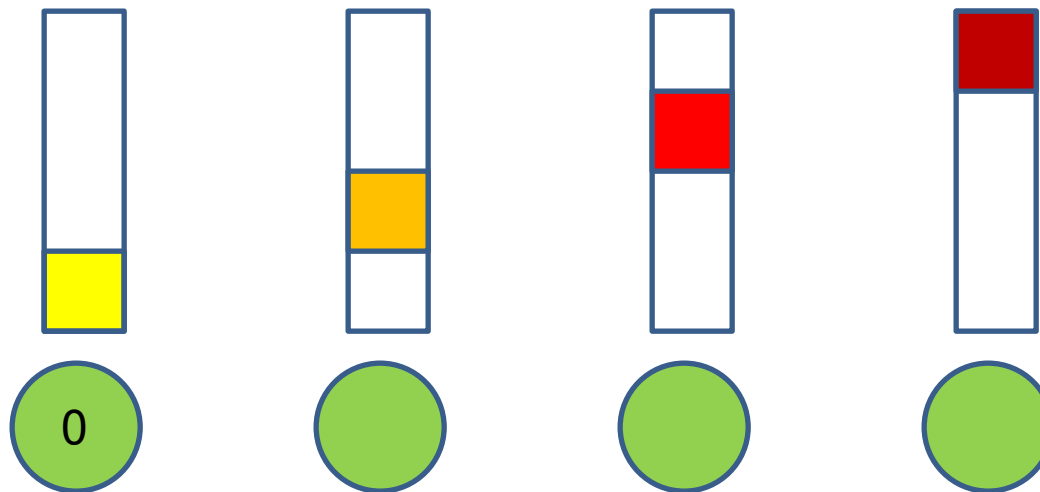
MPI (user-defined) operations **may not** be commutative



$p-1$  rounds  $i=1,2,\dots,p-1$ ; after round  $i$ , process rank has computed  $\sum_{\text{rank}-i \leq j \leq \text{rank}} x[\text{rank}-1-i]$

**Recall:** MPI assumes that all operations for MPI\_Reduce etc. are associative; **but floating point operations are not associative**, e.g.,  $(\text{large} + \text{small}) + \text{small} \neq \text{large} + (\text{small} + \text{small})$





$$\begin{aligned} \text{TreduceScatter}(m) &= (p-1)(\alpha + \beta m/p) \\ &= (p-1)\alpha + (p-1)/p \beta m \end{aligned}$$

Ignoring time to  
perform  $p-1$   $m/p$   
block reductions

Non-optimal in  $\alpha$ -term

Optimal in  $\beta$ -term

Combining reduce-scatter with allgather/gather immediately gives

```
MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,
               comm) ;
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,
            comm) ;
```

$$\begin{array}{l} \text{Tallreduce}(m) = 2(p-1)\alpha + 2(p-1)/p \beta m \\ \text{Treduce}(m) = 2(p-1)\alpha + 2(p-1)/p \beta m \end{array} \quad \left. \vphantom{\begin{array}{l} \text{Tallreduce}(m) \\ \text{Treduce}(m) \end{array}} \right\} \begin{array}{l} \text{Note: Same complexity} \\ \text{under model} \\ \text{assumptions. Is that} \\ \text{really so (benchmark!)} \end{array}$$

Factor 2 from optimal in  $\beta$ -term

But major improvement over trivial algorithms that reduce the  $m$  elements along the ring:  $\text{Treduce}(m) = \text{Tallreduce}(m) = 2(p-1)(\alpha + \beta m)$

## The power of pipelining

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

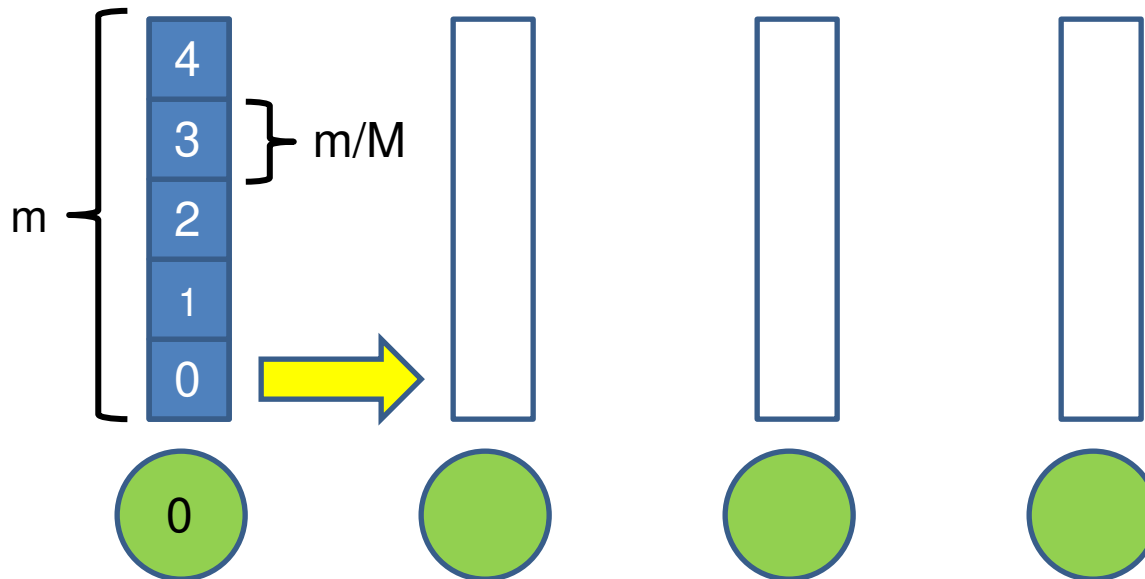
Assume  $m$  large,  $m > p$

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

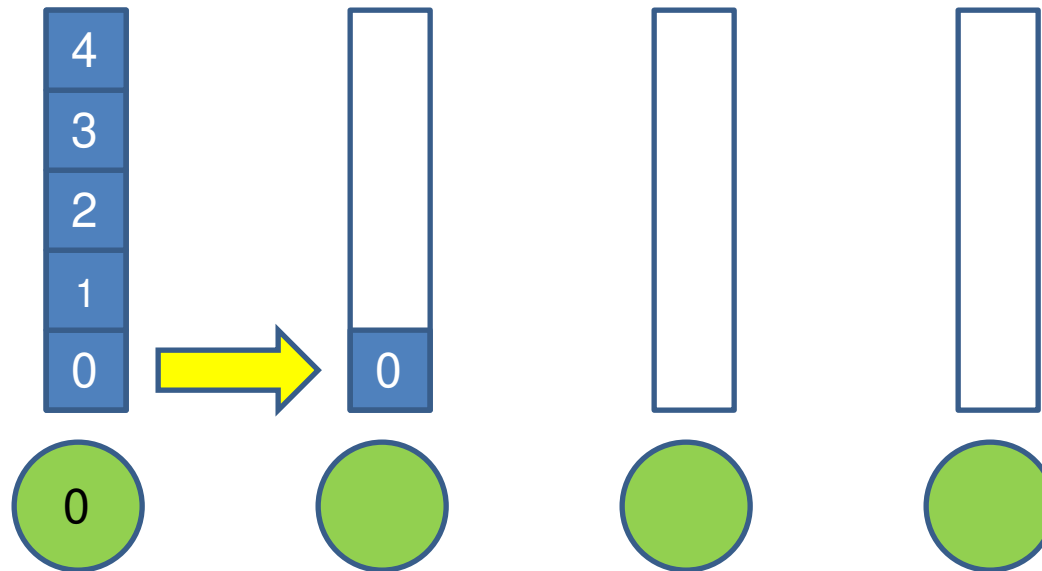
**MPI technicality** : If `datatype` (structure of data element in `buffer` of `count` elements) describes a large element, dividing into blocks of  $m/M$  units requires special capabilities from MPI library implementation. **Currently insufficient MPI functionality**

## Pipelined broadcast



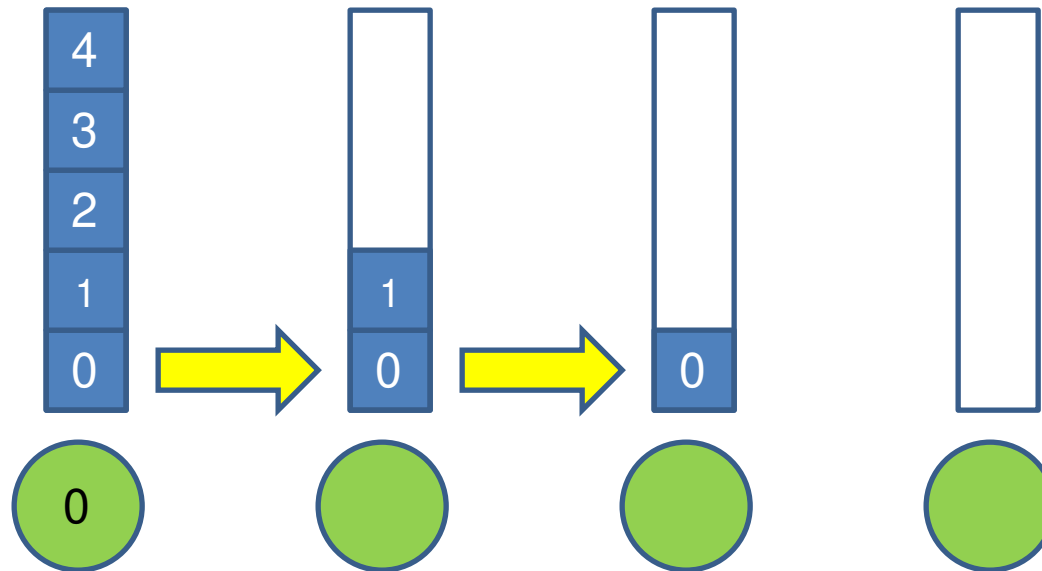
Root (0):

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



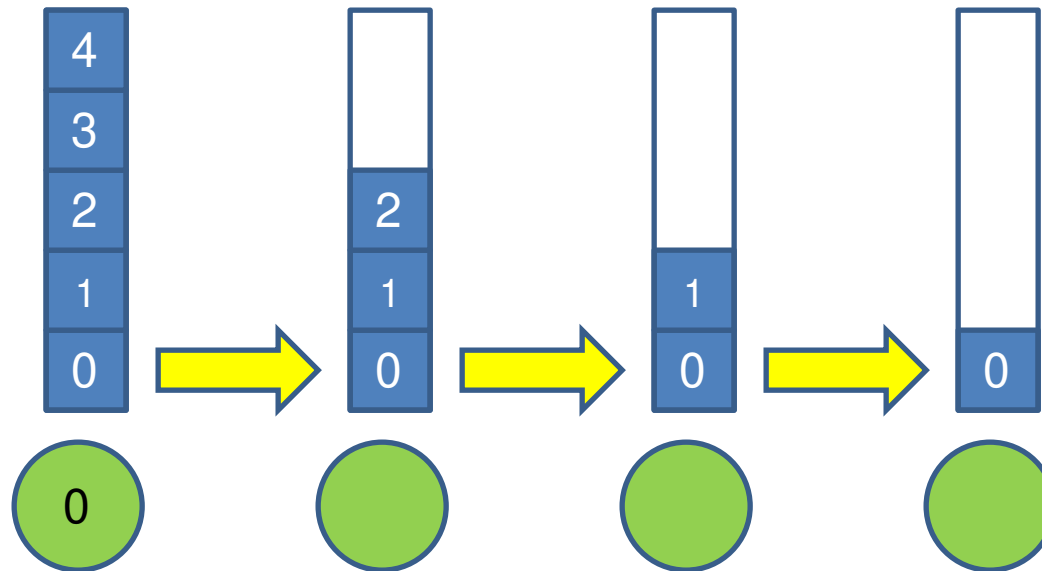
Non-root, rank < size-1:

```
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);
```



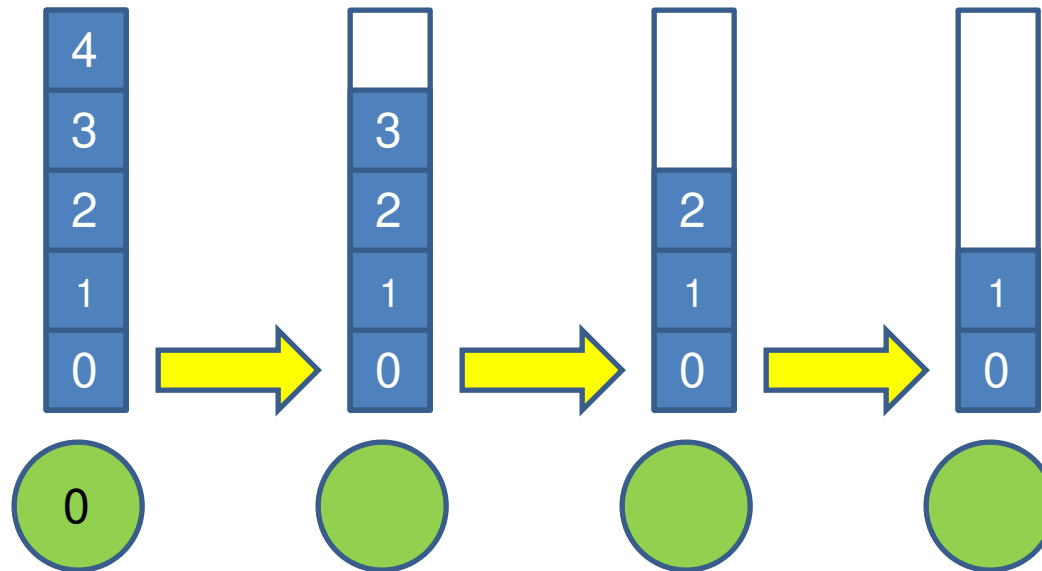
Non-root, rank < size-1:

```
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,  $\text{rank} < \text{size} - 1$ :

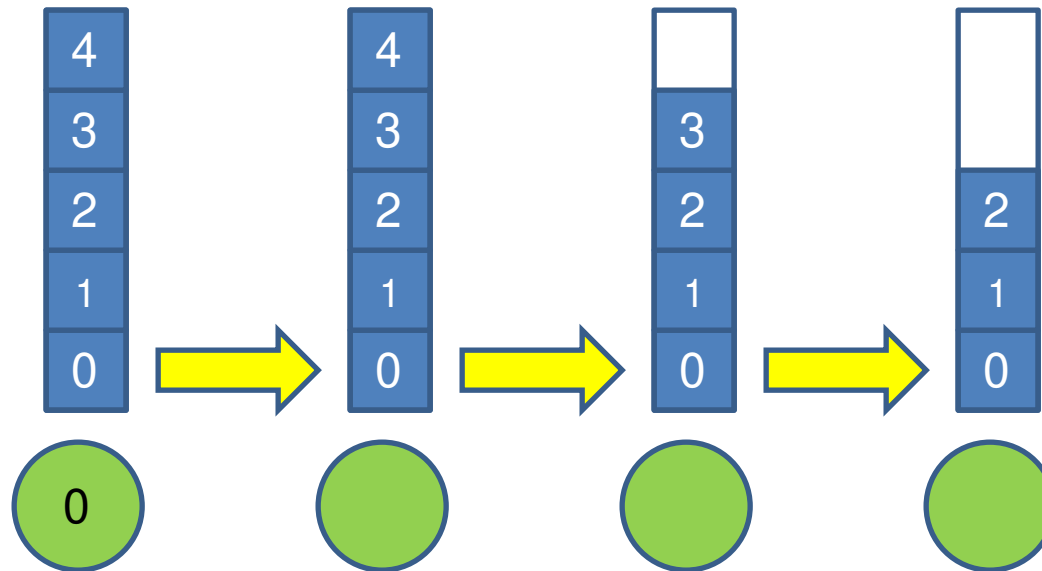
```
MPI_Recv(buffer[0], ..., comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1], ..., buffer[b], ..., comm);
}
MPI_Send(buffer[M-1], ..., comm);
```



Non-root, rank < size-1:

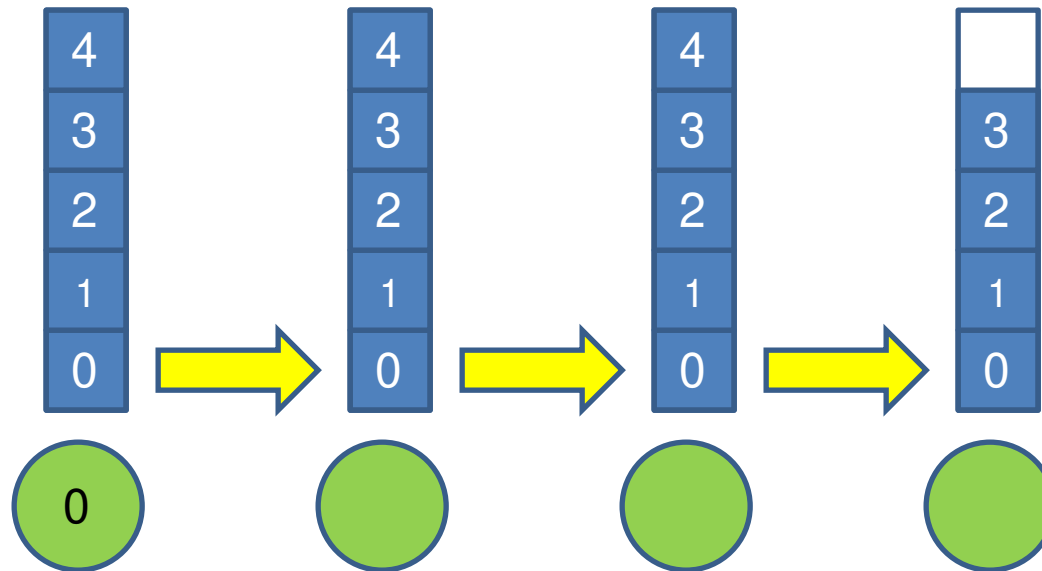
```
MPI_Recv(buffer[0], ..., comm);
for (b=1; j<M; j++) {
    MPI_Sendrecv(buffer[b-1], ..., buffer[b], ..., comm);
}
MPI_Send(buffer[M-1], ..., comm);
```





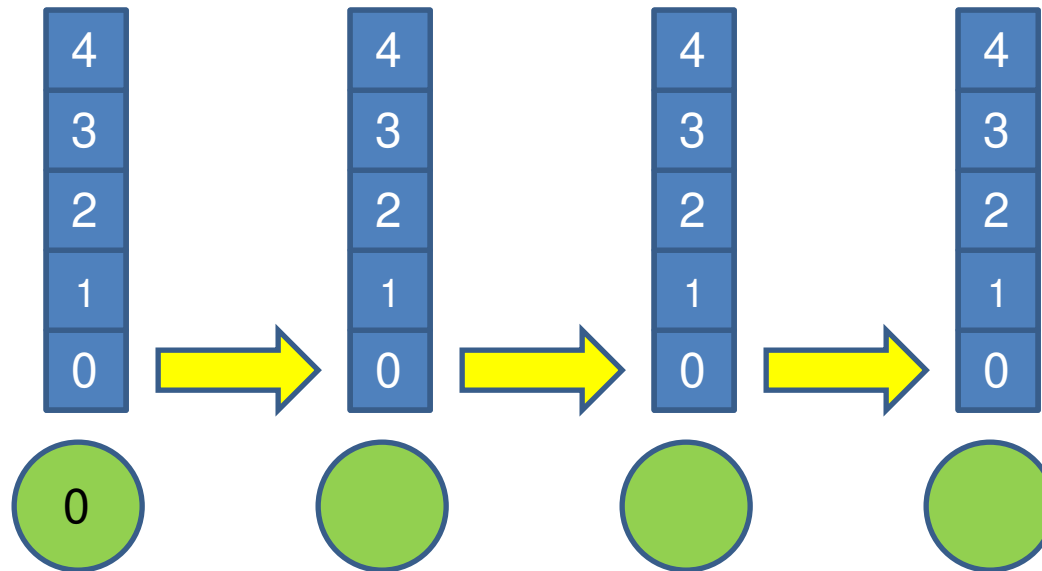
Non-root, rank < size-1:

```
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);
```



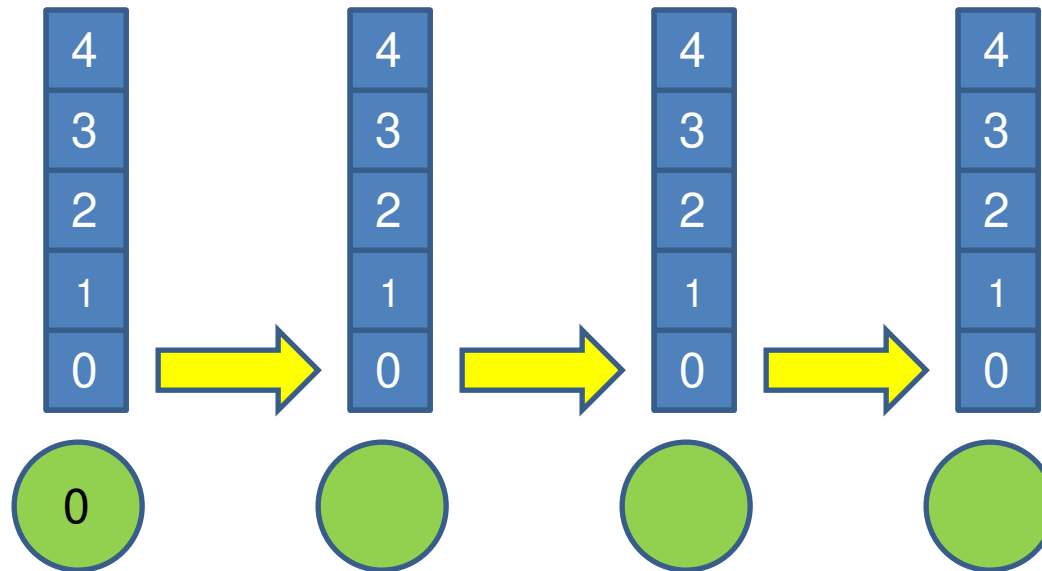
Non-root, rank < size-1:

```
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);
```



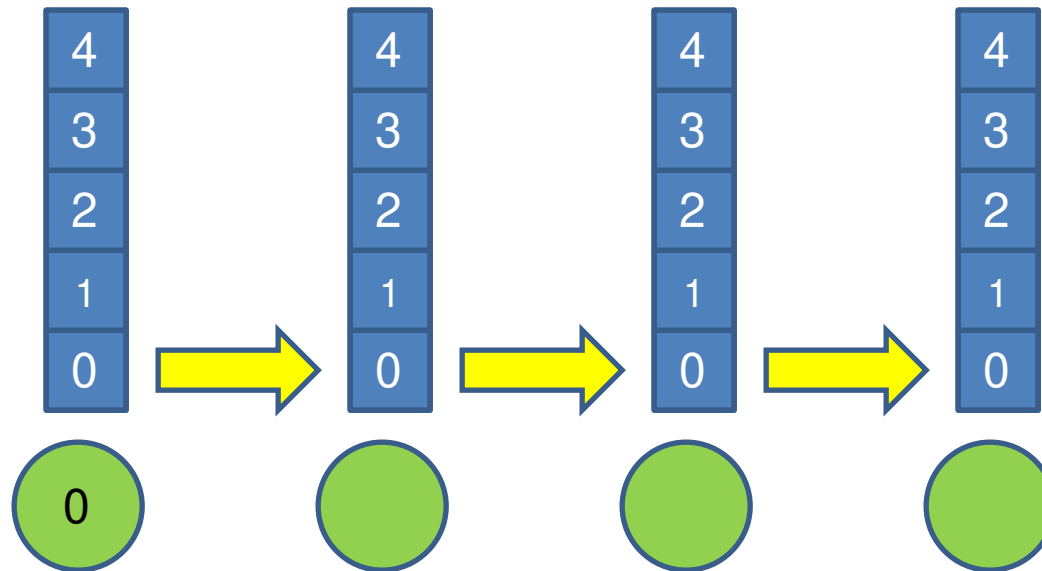
Non-root, rank < size-1:

```
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:

```
for (b=0; j<M; j++) {  
    MPI_Recv(buffer[b], ..., comm);  
}
```



- 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:

$$T_{\text{broadcast}}(m) = (p-1+M-1)(\alpha+\beta m/M)$$

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  $\text{root} \neq 0$  and  $\text{root} \neq p-1$

### 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

$$(k-s)\alpha + 2\sqrt{[s(k-s)\alpha\beta m]} + s\beta m$$

Proof:

Pipelining with  $M$  blocks takes

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

Corollary:

Best possible time for linear pipeline broadcast is

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

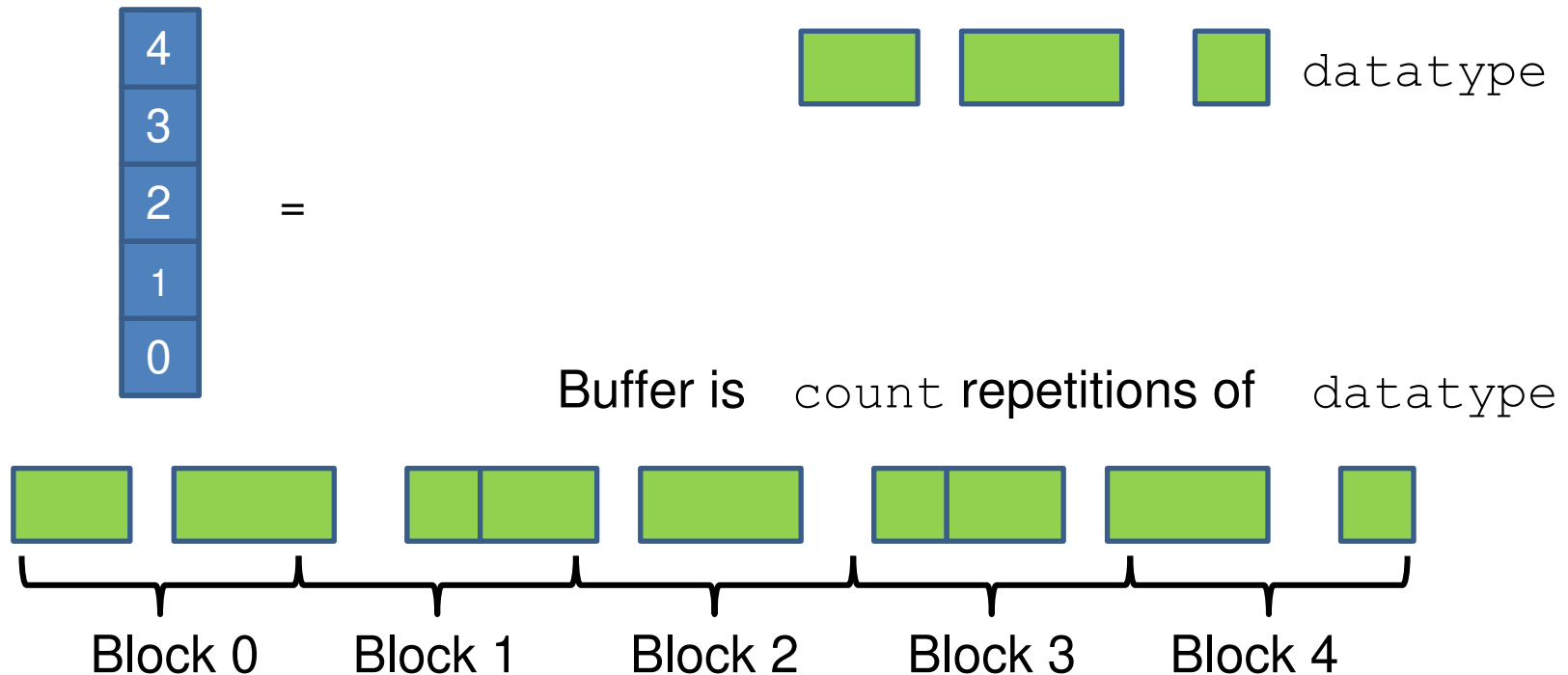
since  $k=p-1$  and  $s=1$

Optimal in  $\beta$ -term

**Practical relevance** : Extremely simple, good when  $m \gg p$



## MPI difficulty with pipelined algorithms: Structured buffers



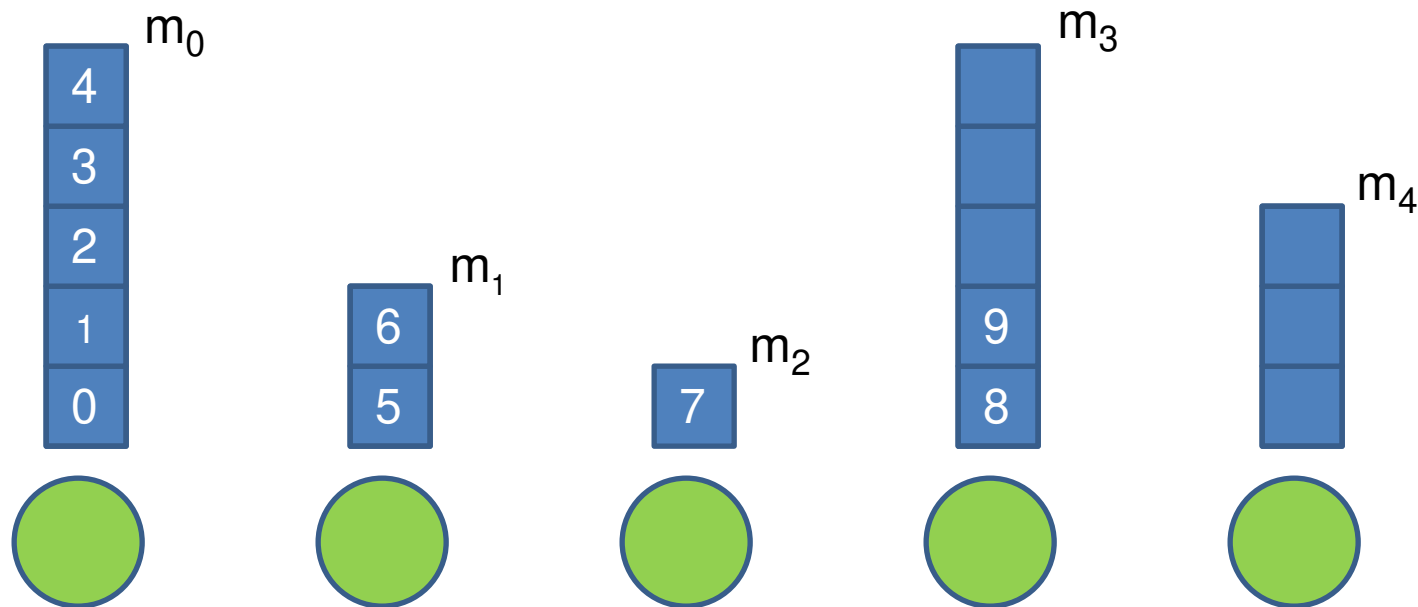
MPI library needs internal functionality to access parts of structured buffers. MPI specification does **not expose any** such functionality

## Pipelined irregular allgather: First result for irreg. collective

```
MPI_Allgatherv(sendbuf, sendcount, sendtype,  
               recvbuf,  
               recvcounts[], recvdispl[], recvtype,  
               comm) ;
```

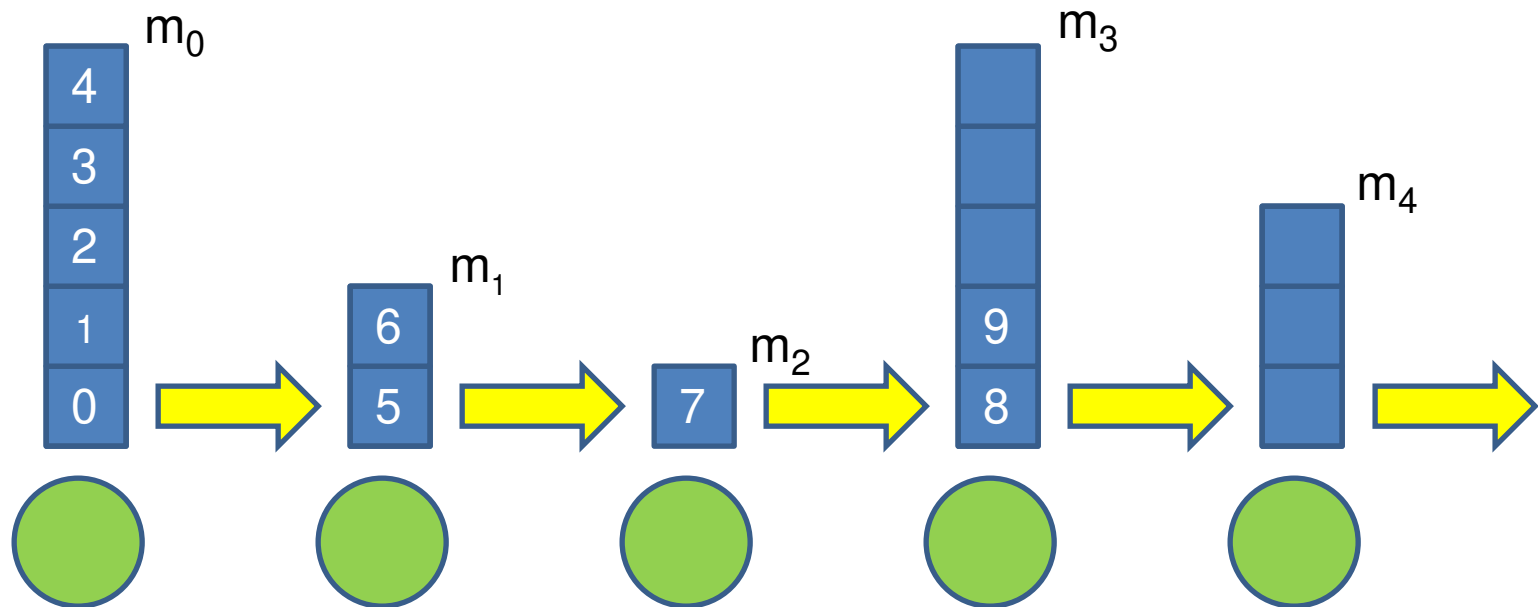
Total data  $m = \sum \text{sendcount} * \text{size}(\text{sendtype})$ , each process can have a different amount of data (sendcount)

Choose blocksize  $b$ , send and receive in  $M$  rounds until all blocks received by all processes. What is  $M$ ?

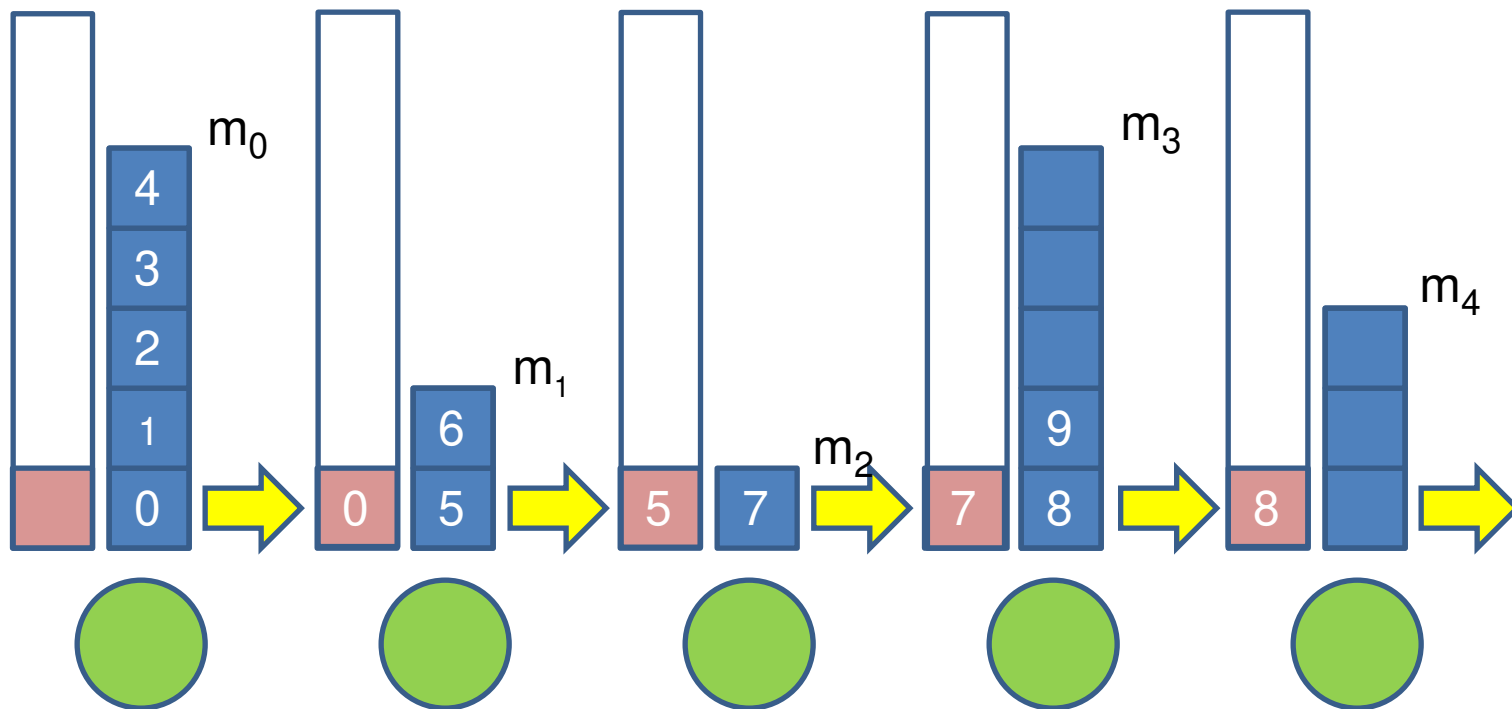


Organize processes in a ring, and pipeline. Observation from:

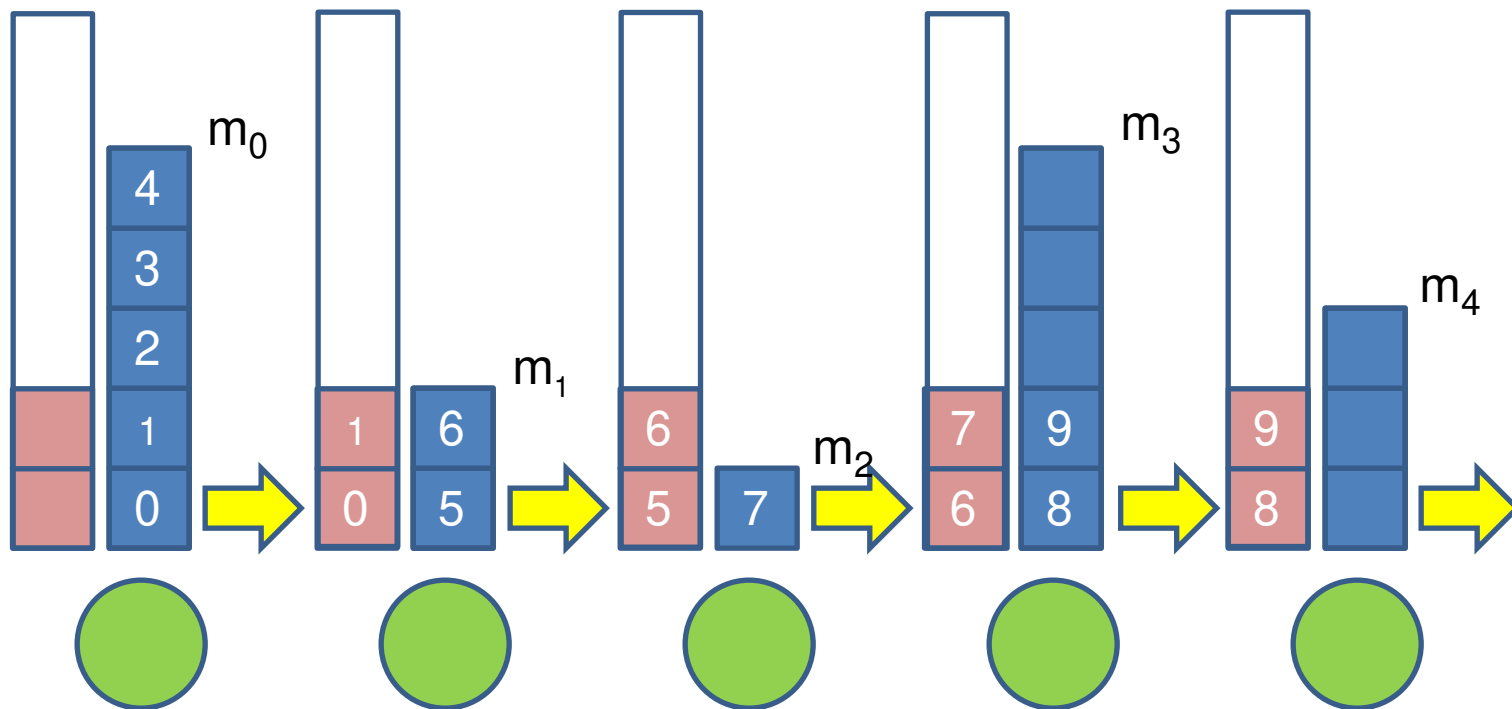
Jesper Larsson Träff, Andreas Ripke, Christian Siebert, Pavan Balaji, Rajeev Thakur, William Gropp: A Pipelined Algorithm for Large, Irregular All-Gather Problems. Int. J. High Perform. Comput. Appl. 24(1): 58-68 (2010)



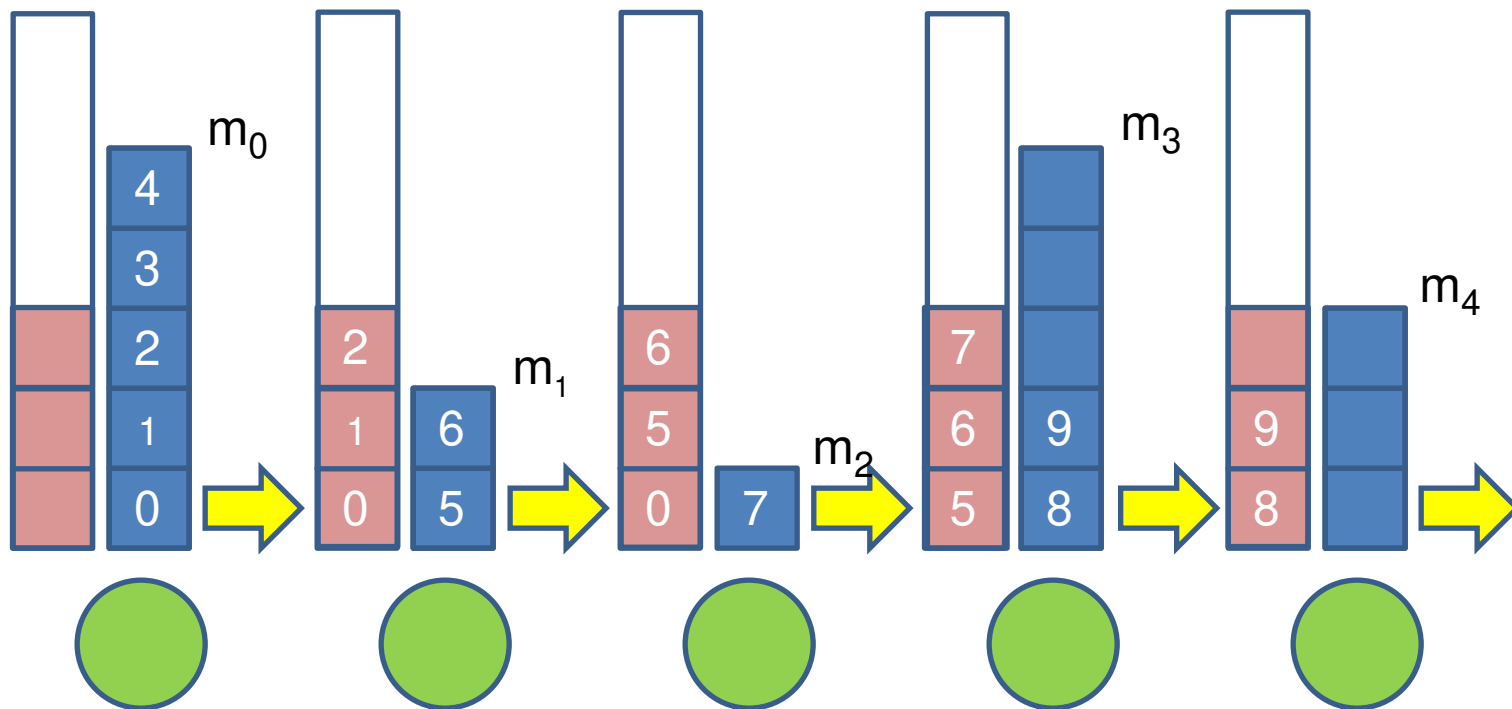
Organize processes in a ring, and pipeline.



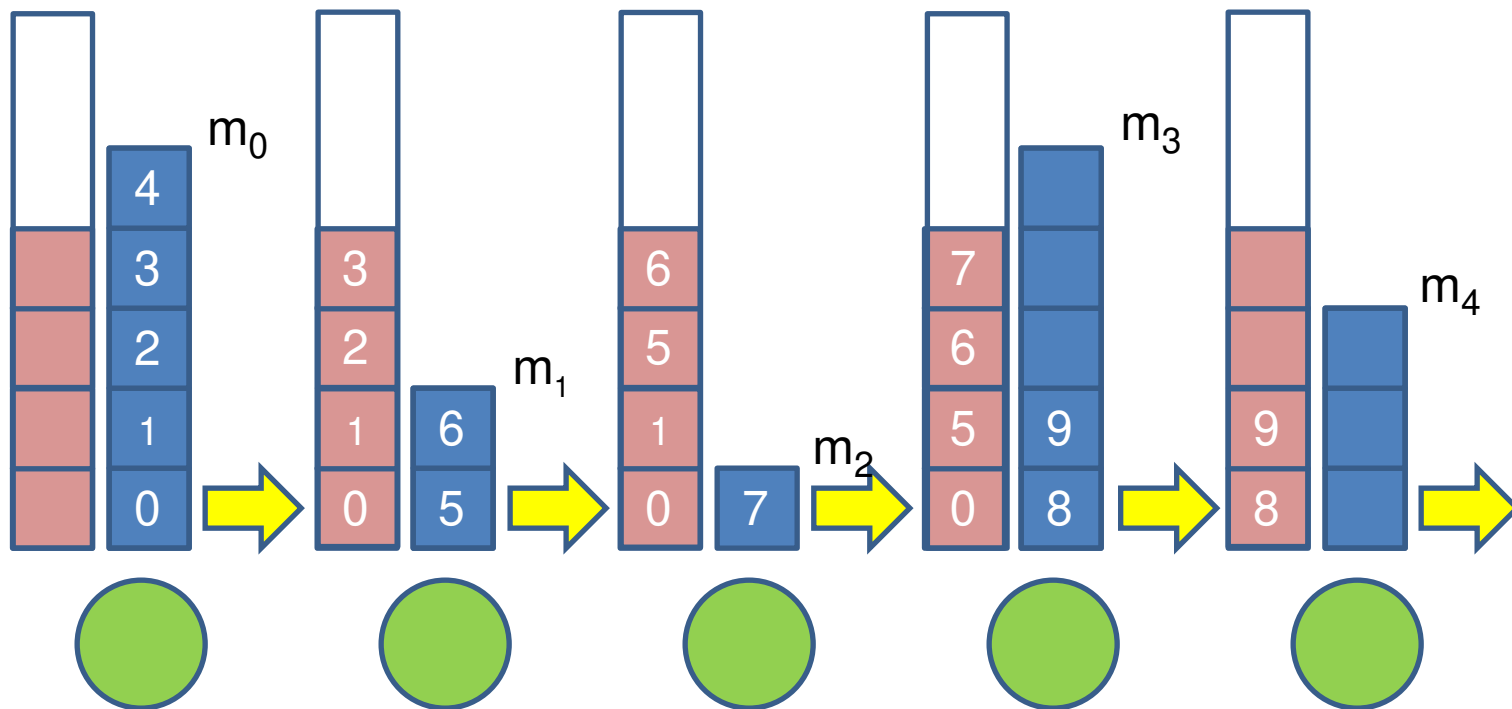
Organize processes in a ring, and pipeline.



Organize processes in a ring, and pipeline.

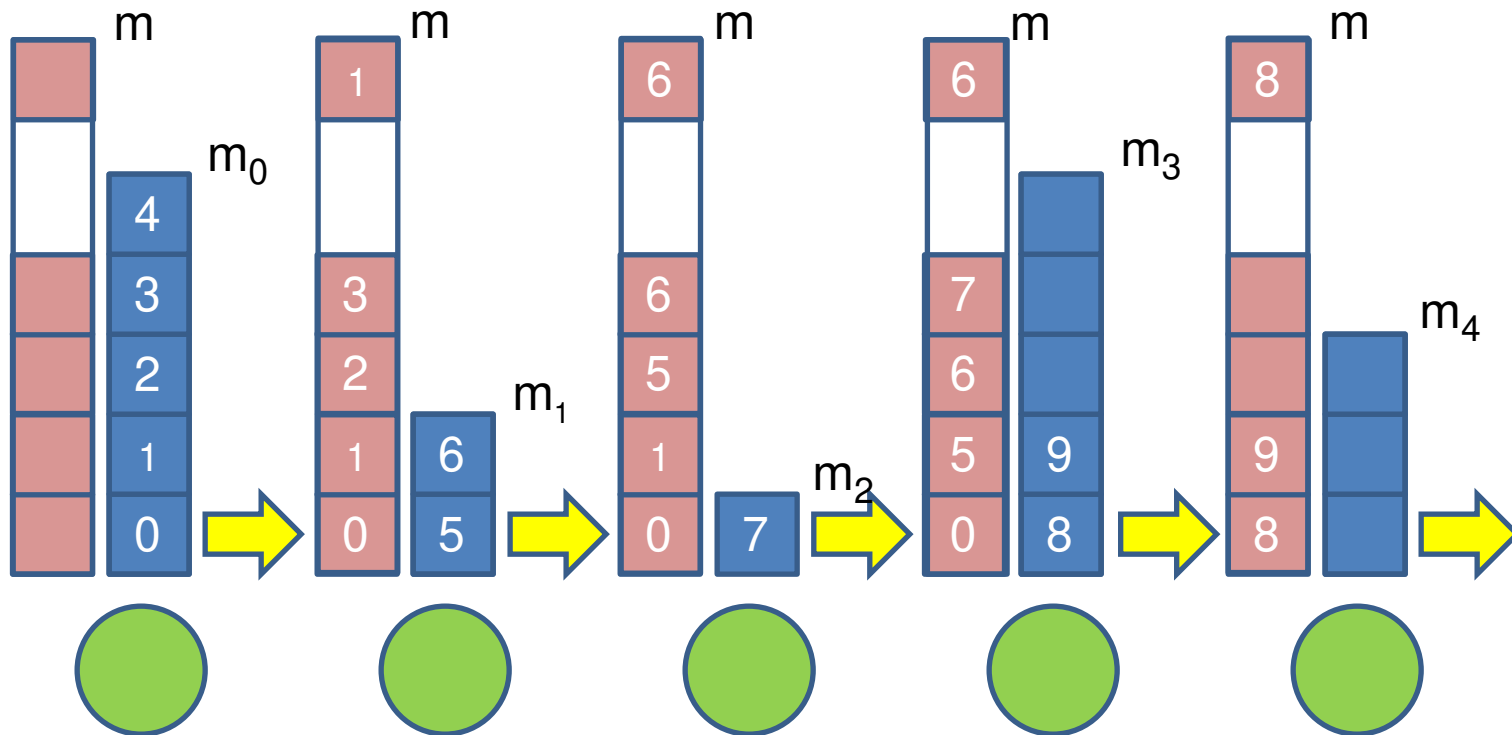


Organize processes in a ring, and pipeline.

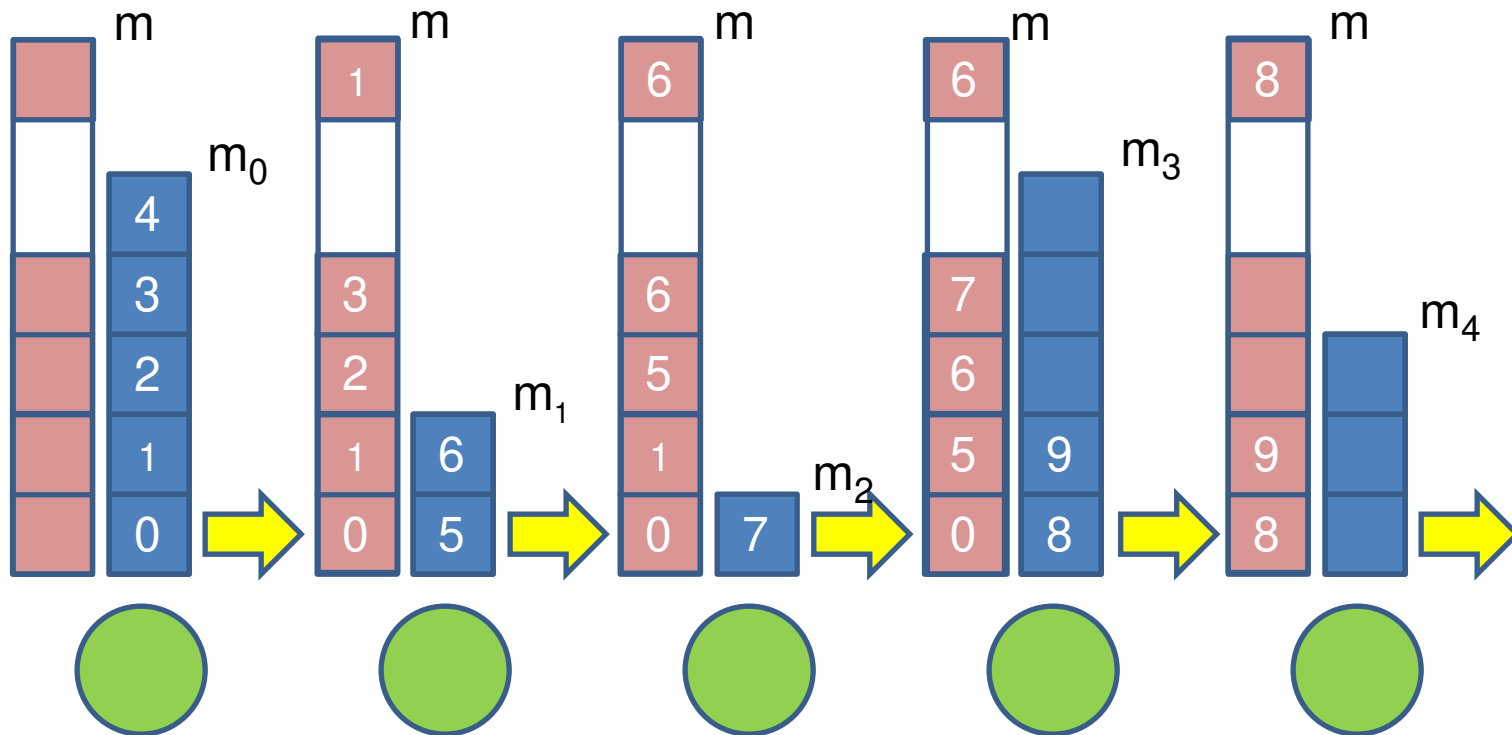


Organize processes in a ring, and pipeline.





Organize processes in a ring, and pipeline.



Organize processes in a ring, and pipeline. Number of rounds is  $M = \sum \text{ceiling}(m_i/b)$ , where  $m_i$  is the amount of data in the block of processor  $i$ . Effective latency  $k$  is  $p-1$  (why?). Each process sends and receives a new block every  $s=1$  rounds. Pipelining lemma applies with  $m = Mb$ . **Exercise: Fill in details, implement**

## The power of trees: Binomial tree gather

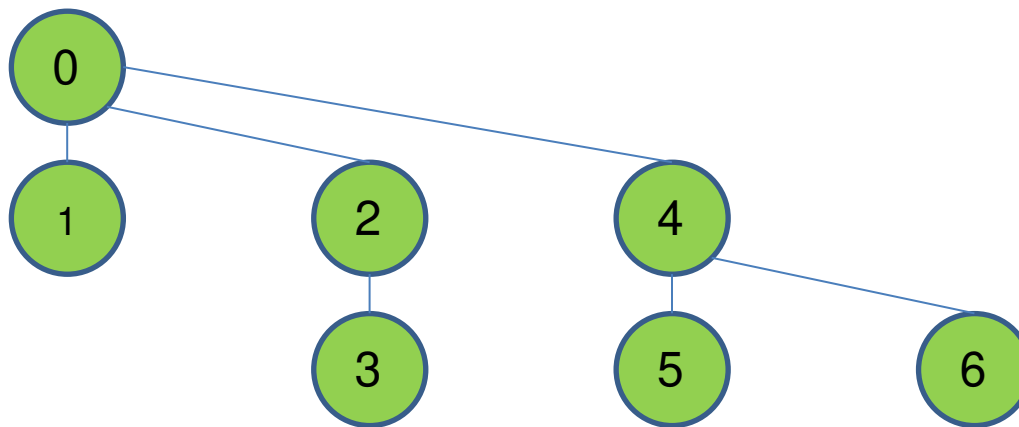
Assume (some convenient) tree can be embedded in communication network: Each tree edge is mapped to a network edge. Many processors in tree receive blocks from subtrees in parallel

For simplicity, continue to assume  $\text{root}=0$

Wlog: Case  $\text{root} \neq 0$  can be handled by “shifting towards 0” (set  $\text{rank}' = (\text{rank} - \text{root} + \text{size}) \% \text{size}$ ) for broadcast, gather/scatter. For reduction to root, same works for commutative MPI operations, otherwise, modifications are necessary

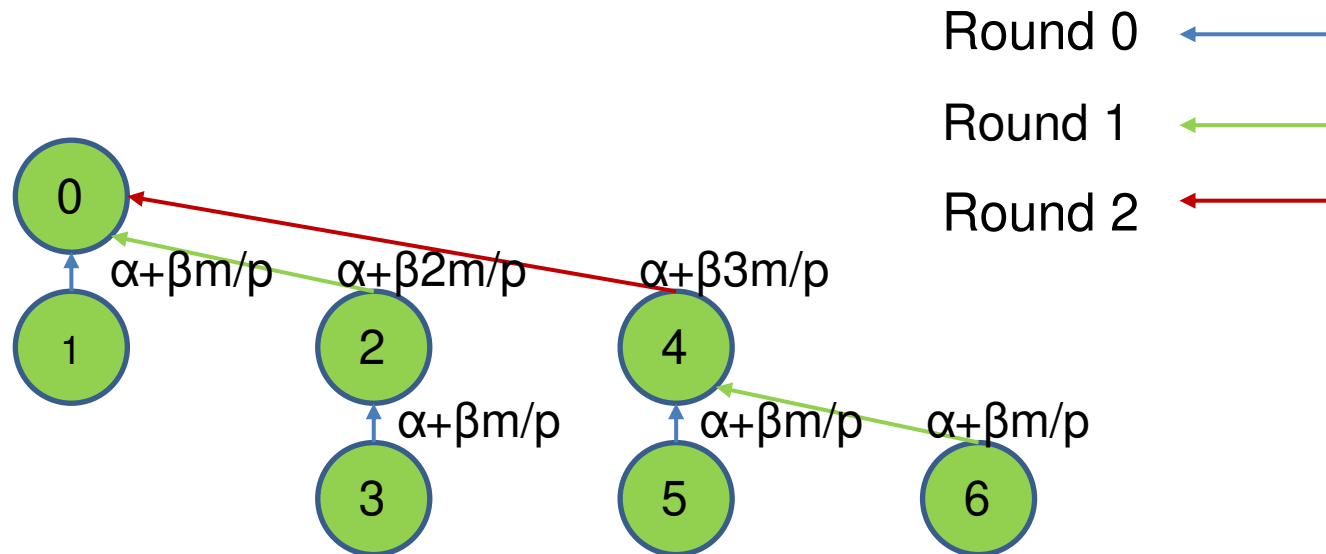
**Note** : Algorithms exploits 1-ported communication, but can be (optimally) generalized to k-ported communication

Binomial tree for gathering to root=0

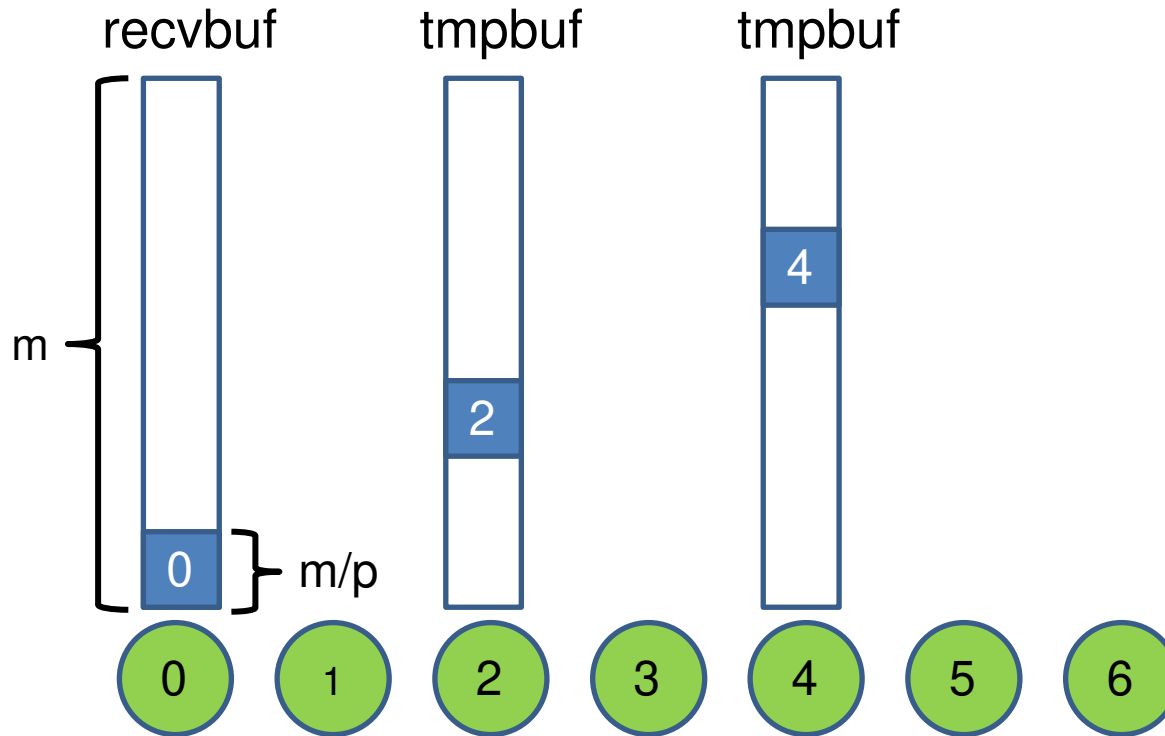


Processors numbered by pre-order traversal

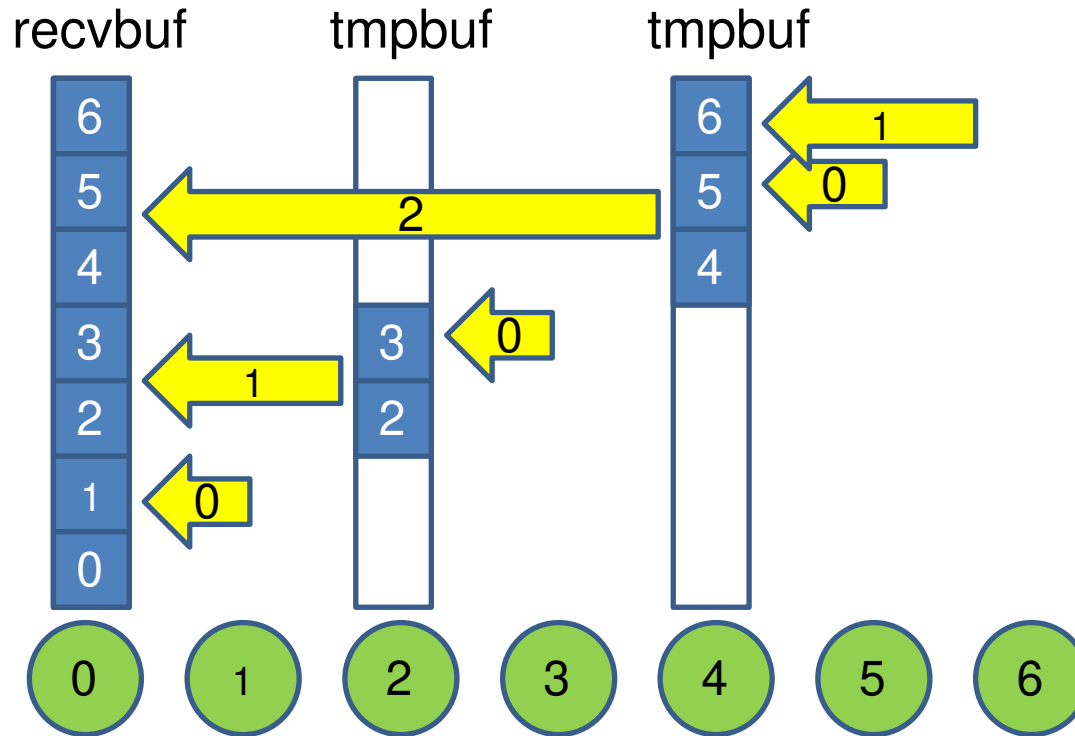
## Binomial tree gather in rounds



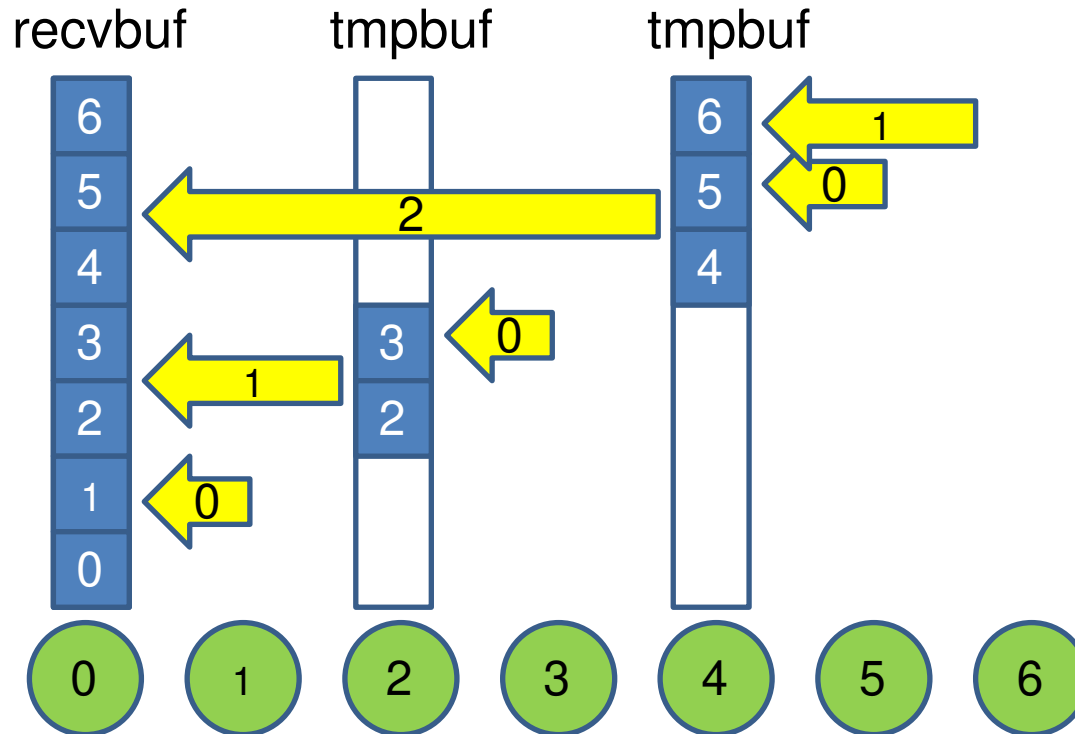
- Three (3) communication rounds ( $= \text{ceil}(\log 7)$ )
- Each child can be found in  $O(1)$  steps, followed by parent in  $O(1)$  steps
- At most  $2^i m/p$  data per round,  $i=0,1,\dots,\text{ceil}(\log p)-1$
- Total time in linear cost transmission model  $3\alpha + \beta 6m/p$



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

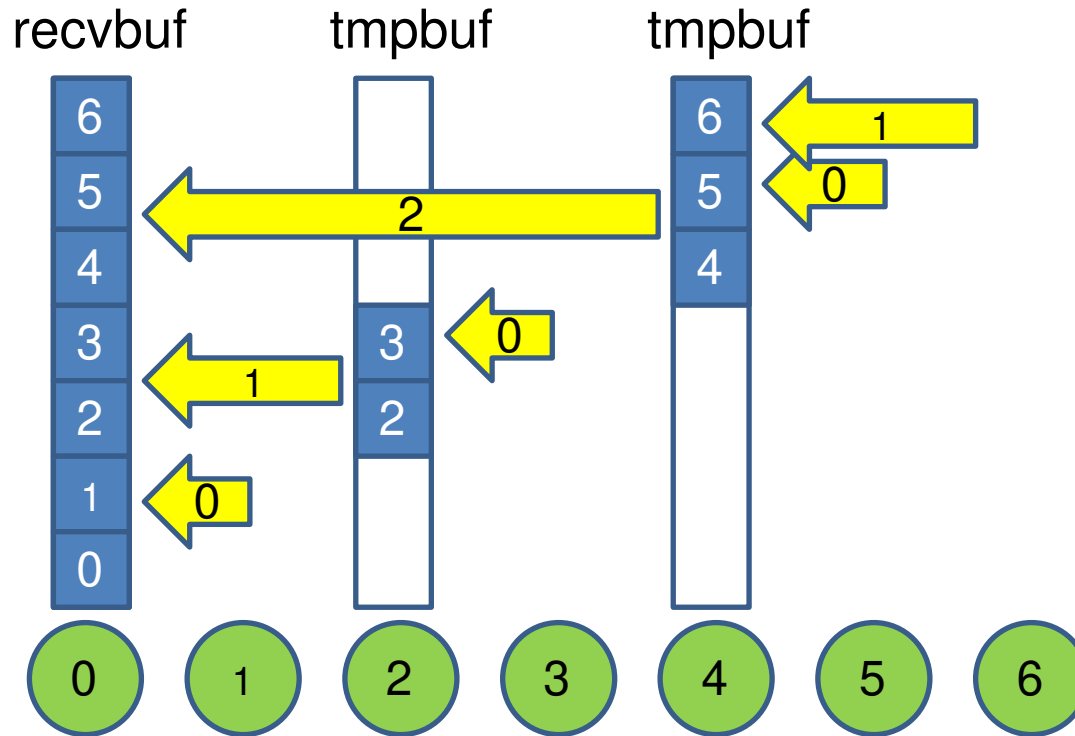


```
MPI_Gather(sendbuf, scount, stype,
           recvbuf, rcount, rtype, root, comm);
```



- $\text{ceil}(\log p)$  communication rounds
- Root process 0 active in each communication round
- Non-root processes send only once: in round  $k$ , where  $k$  ( $=0, 1, \dots, \log p - 1$ ) is the first set bit in process rank
- Amount of data gathered doubles in each round

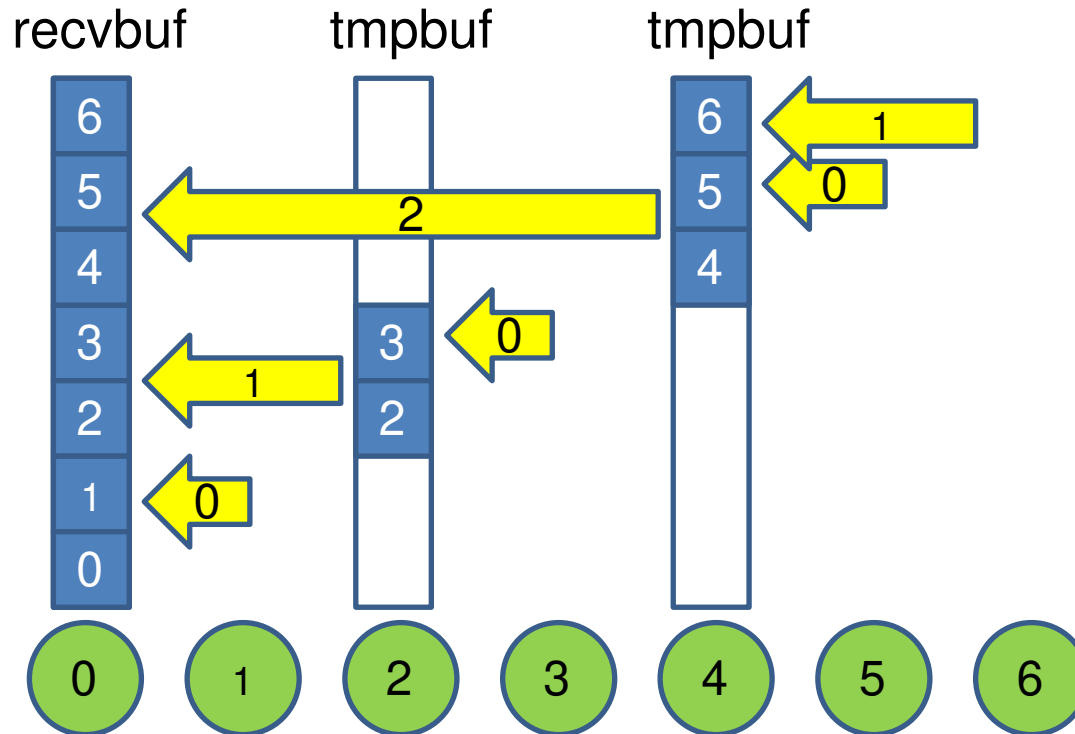




```

d=1;
while ((rank&d)<size) {
    if (rank+d<size) MPI_Recv(tmpbuf,...,rank+d,...,comm);
    d <<= 1;
}
if (rank!=root) MPI_Send(tmpbuf,...,rank-d,...,comm);

```



$$T_{\text{gather}}(m) = \text{ceil}(\log p)\alpha + \beta m(p-1)/p$$

Optimal in  $\alpha$ -term

Optimal in  $\beta$ -term

**Note :** In each round, a non-idle processor either only sends or receives. Each processor can determine in  $O(1)$  steps what to do

## Binomial tree reduction

```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm);
```

All processes contribute a (typed) vector in sendbuf. Root process computes result in recvbuf (significant at root only)

```
if (rank==root) result = recvbuf;  
else result = malloc(count*...); // datatype  
memcpy(result,sendbuf,...); // need typed memcpy
```

```

if (rank==root) result = recvbuf;
else result = malloc(count*...); // datatype
memcpy(result,sendbuf,...); // need typed memcpy

```

There is no typed, local memcpy function in MPI. But library might implement typed copy efficiently?:

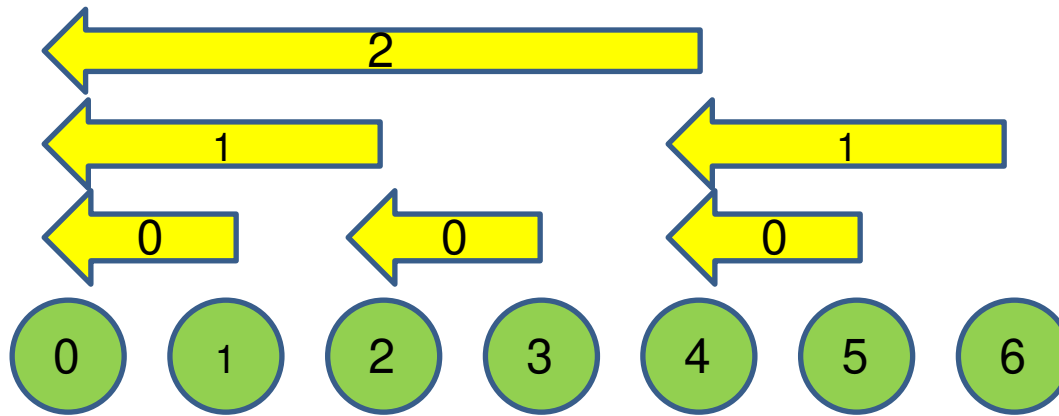
```

if (rank==root) result = recvbuf;
else result = malloc(count*...); // datatype
MPI_Sendrecv(sendbuf,count,datatype,TAG,0,
              result,count,datatype,TAG,0,
              MPI_COMM_SELF,MPI_STATUS_IGNORE);

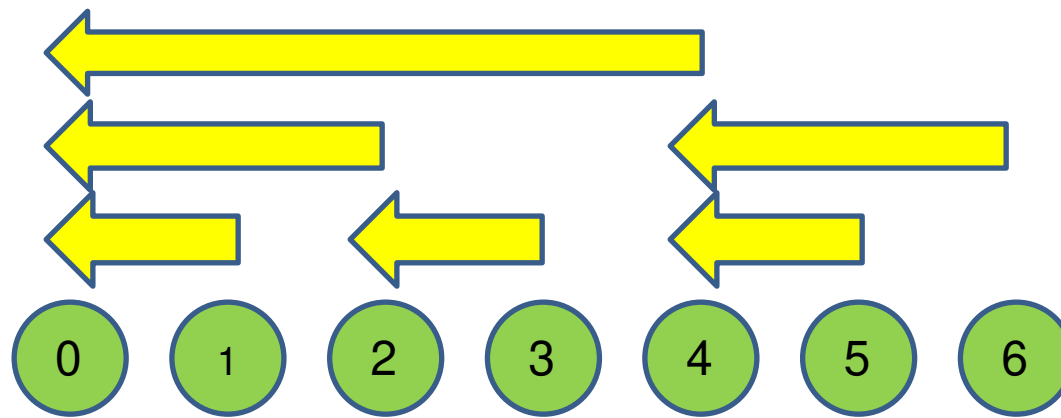
```



Special, singleton communicator, process  
has rank 0 in MPI\_COMM\_SELF



```
d=1;
while ((rank&d)≠d&&d<size) {
    if (rank+d<size) {
        MPI_Recv(tmpbuf,...,rank+d,...,comm);
        MPI_Reduce_local(tmpbuf,result,...);
    }
    d <<= 1;
}
if (rank!=root) MPI_Send(result,...,rank-d,...,comm);
```



Ignoring time  
for  $\text{ceil}(\log p)$   
m-element  
reductions

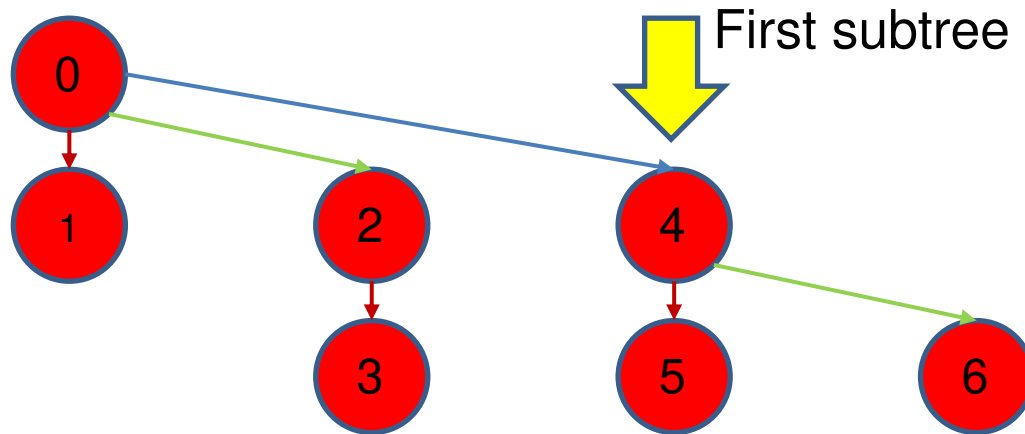
$$T_{\text{reduce}}(m) = \text{ceil}(\log p)(\alpha + \beta m) = \text{ceil}(\log p)\alpha + \text{ceil}(\log p)\beta m$$

Optimal in  $\alpha$ -term

Non-optimal in  $\beta$ -term

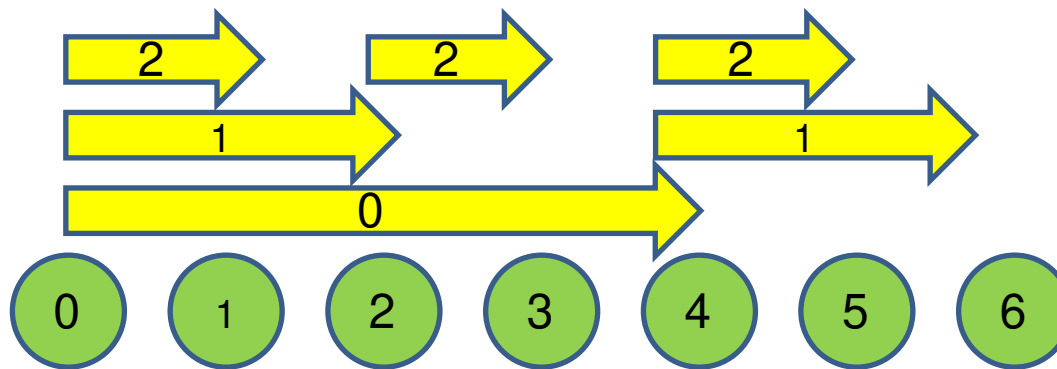
**No** algorithmic latency: Each process can start receiving immediately, that is after  $O(1)$  operations (same for gather)

## Binomial tree broadcast



Pre-order traversal numbering unsuited for broadcast:  
**algorithmic latency** . Processor 0 can start sending to processor  $2^i$  after  $i$  steps, where  $2^i < p$ , undesirable broadcast latency

## Binomial tree broadcast with harmful algorithmic latency



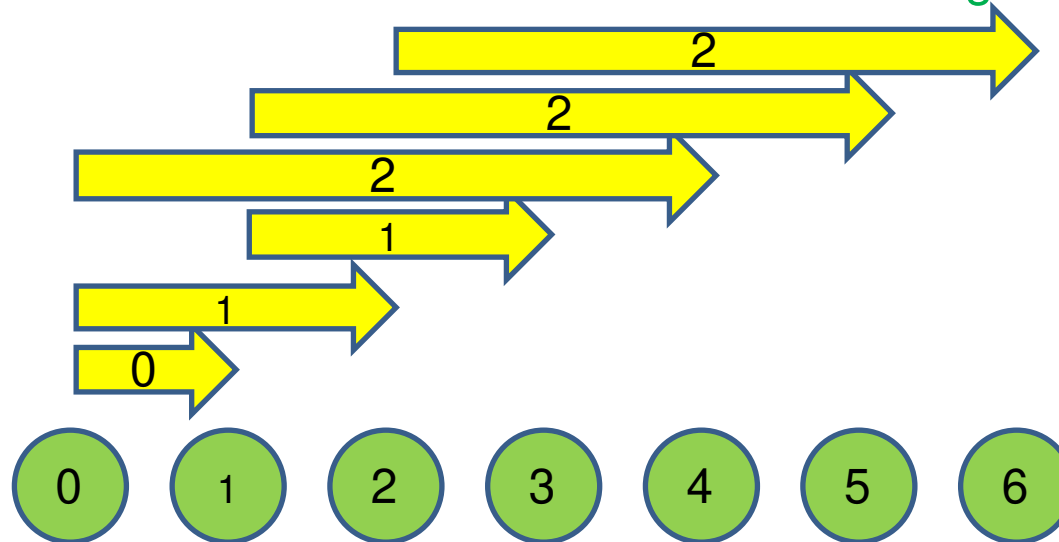
```

d = 1;
while ((rank&d)!=d&& d<size) d <<= 1;
if (rank!=root) MPI_Recv(buffer,...,rank-d,...,comm);
while (d>1) {
    d >>= 1;
    if (rank+d<size) MPI_Send(buffer,...,rank+d,...,comm);
}

```



# Binomial tree broadcast without harmful algorithmic latency

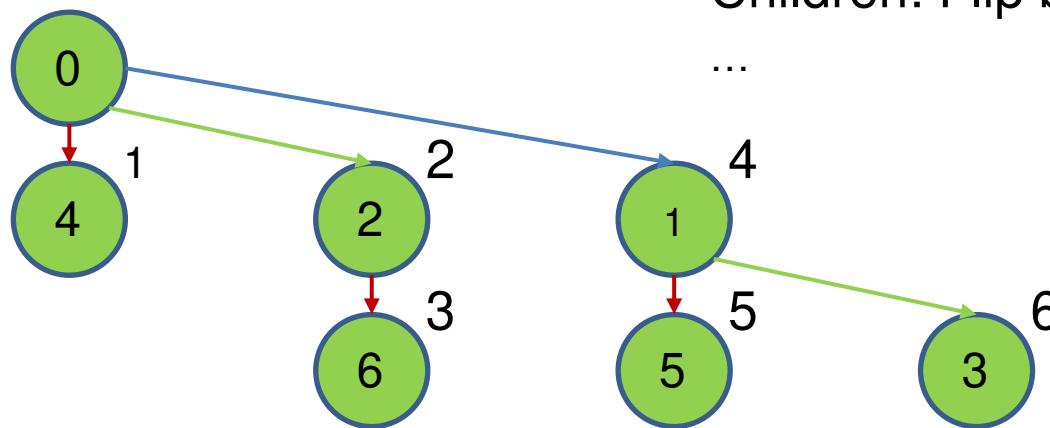


```

d = 1;
while (rank >= d) { dd = d; d <<= 1; }
if (rank != root) MPI_Recv(buffer, ..., rank-dd, ..., comm);
while (rank + d < size) {
    MPI_Send(buffer, ..., rank+d, ..., comm);
    d <<= 1;
}

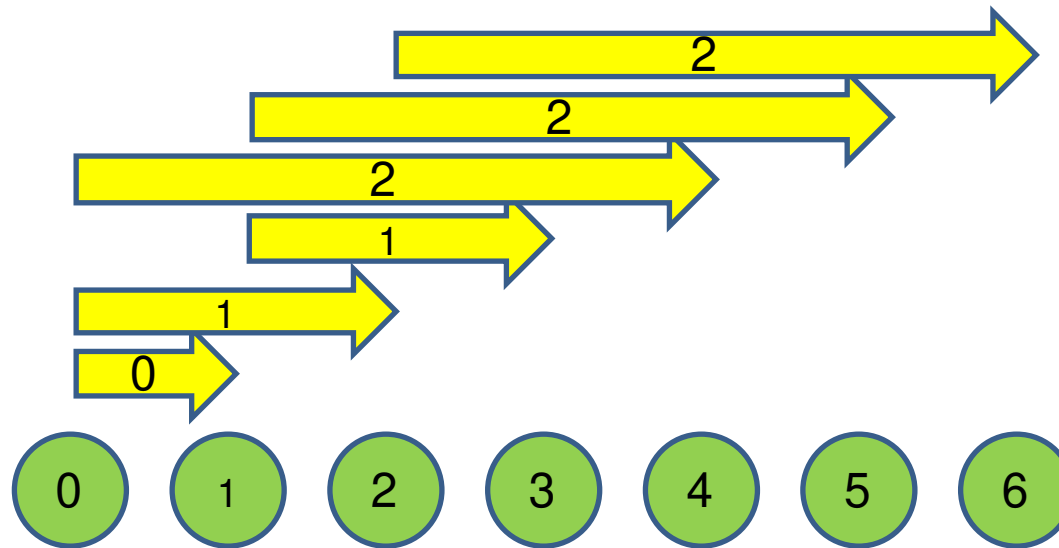
```

- Find first set bit  $k$  (lsb)
- Parent: Flip bit  $k$  (subtract  $2^k$ )
- Children: Flip bits  $k+1, k+2,$   
...



Better suited for broadcast: Bit-reversed pre-order traversal numbers. Parent needed in round  $k$  can be found  $O(k)$  steps

**First HPC lecture** : Parent can be found in  $O(\log k)$  steps, or  $O(1)$  steps with hardware support

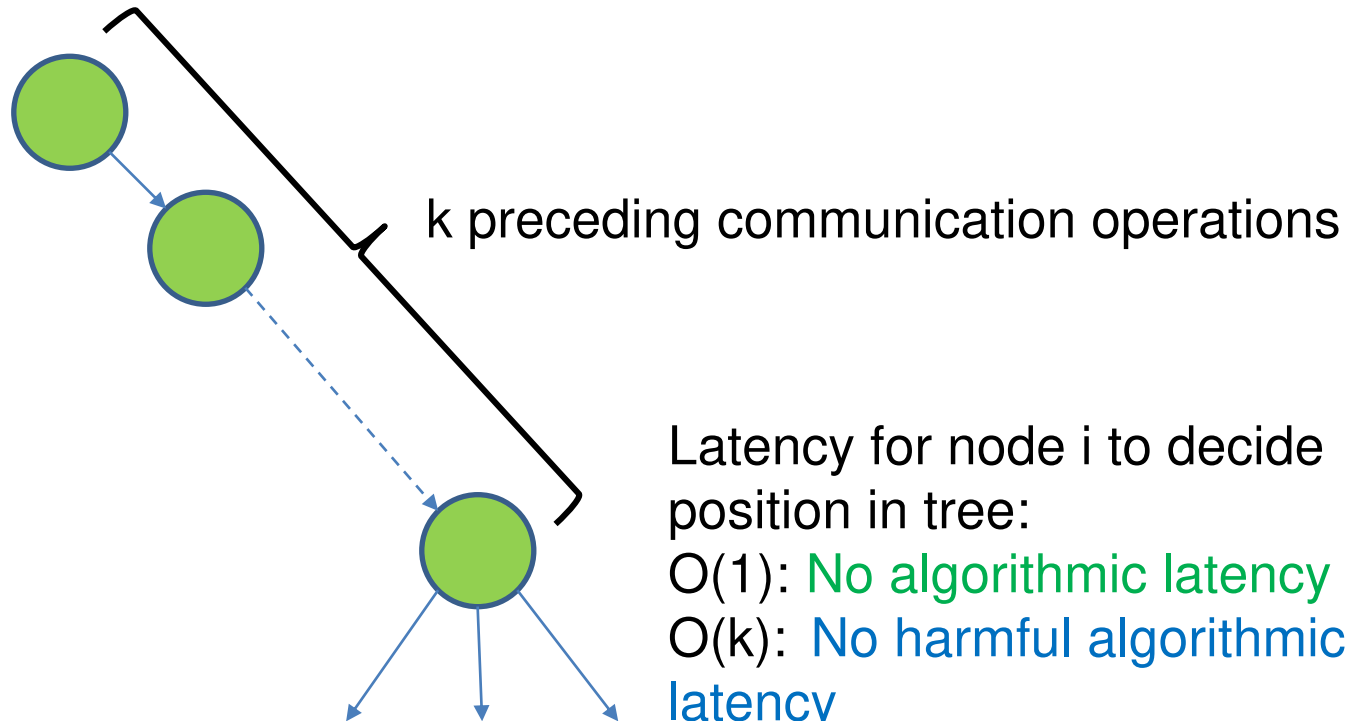


$$T_{\text{broadcast}}(m) = \text{ceil}(\log p)(\alpha + \beta m) = \text{ceil}(\log p)\alpha + \text{ceil}(\log p)\beta m$$

Optimal in  $\alpha$ -term, not in  $\beta$ -term

**No** harmful algorithmic latency: Process to start in round  $k$  executes while loop of  $k$  iterations

## Algorithmic latency in tree algorithms



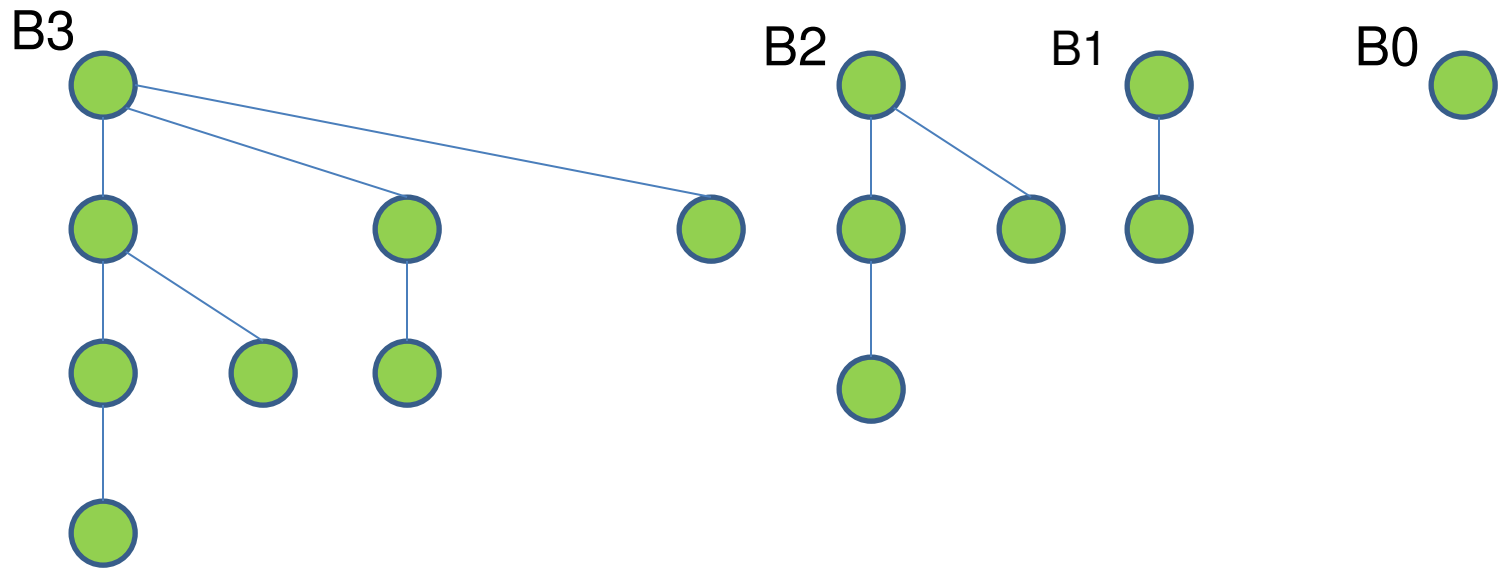
Latency for node  $i$  to decide position in tree:

$O(1)$ : No algorithmic latency

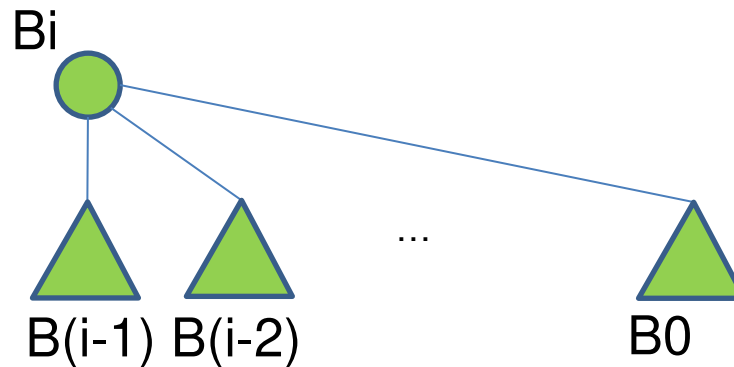
$O(k)$ : No harmful algorithmic latency

$\omega(k)$ : Possibly harmful algorithmic latency

## Binomial tree: Structure



## Binomial tree: Structure



### Properties:

- $B_i$  has  $2^i$  nodes,  $i \geq 0$
- $B_i$  has  $i+1$  levels,  $0, \dots, i$
- $B_i$  has  $\text{choose}(i, k) = i! / (i-k)!k!$  nodes at level  $k$

Home exercise : Prove by induction

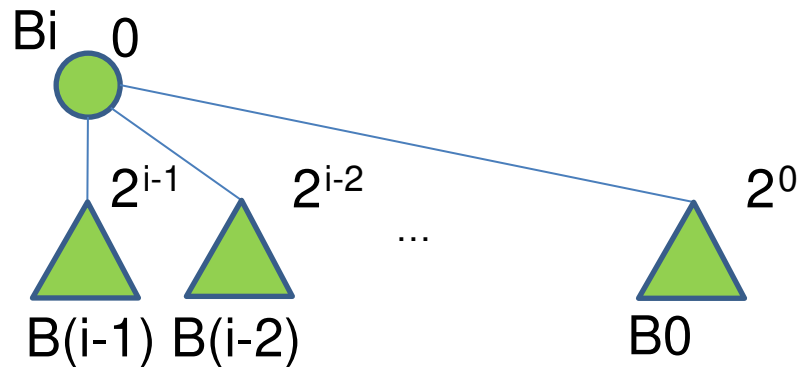
## Standard choose(i,k) exercises

Prove that

- $\text{choose}(i-1,k) + \text{choose}(i-1,k-1) = \text{choose}(i,k)$
- $\sum_{0 \leq k \leq i} \text{choose}(i,k) = 2^i$

Argue either a) combinatorially, b) by definition and induction.  
Do not use the binomial theorem

## Binomial tree: Naming (ranks)

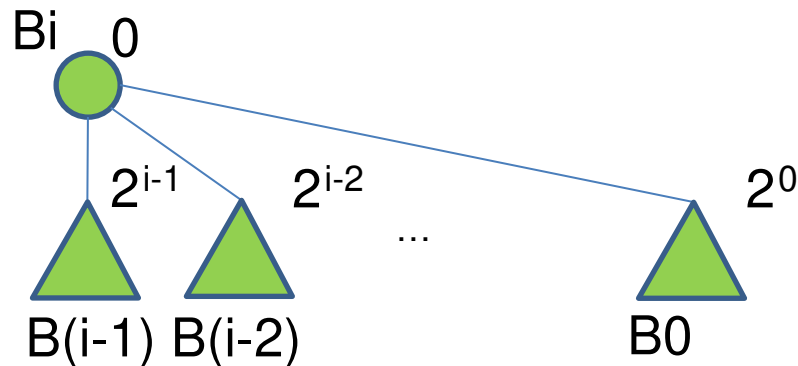


Recursively:

- Rank in  $B_0$  is 0
- For  $k=1, \dots, i$ , add  $2^{i-k}$  to ranks in subtree  $B(i-k)$



## Binomial tree: Naming (ranks)

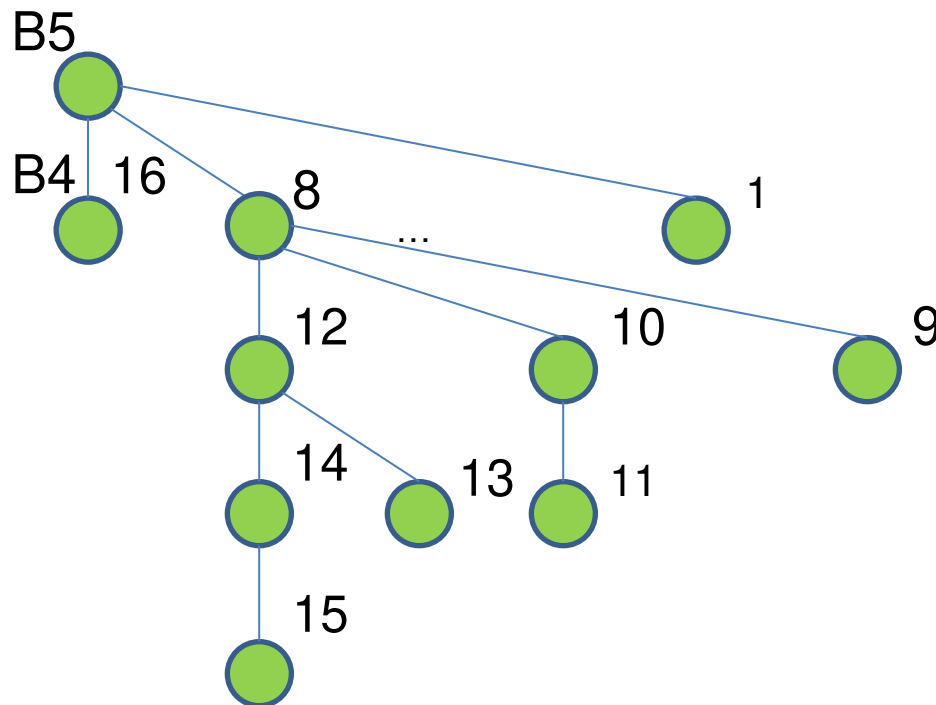


In binary (root=0):

Let rank  $i = Y10\dots00_2$  for some binary prefix  $Y$ ,  $k$  be position of first 1 (from least significant bit,  $k \geq 1$ )

- $i$ 's  $k-1$  children  $Y10\dots01$ ,  $Y10\dots10$ , ...,  $Y11\dots00$
- $i$ 's parent  $Y00\dots00$
- all  $i$ 's descendants  $Y1X$  for  $k-1$  bit suffix  $X$

## Binomial tree: Naming (ranks)

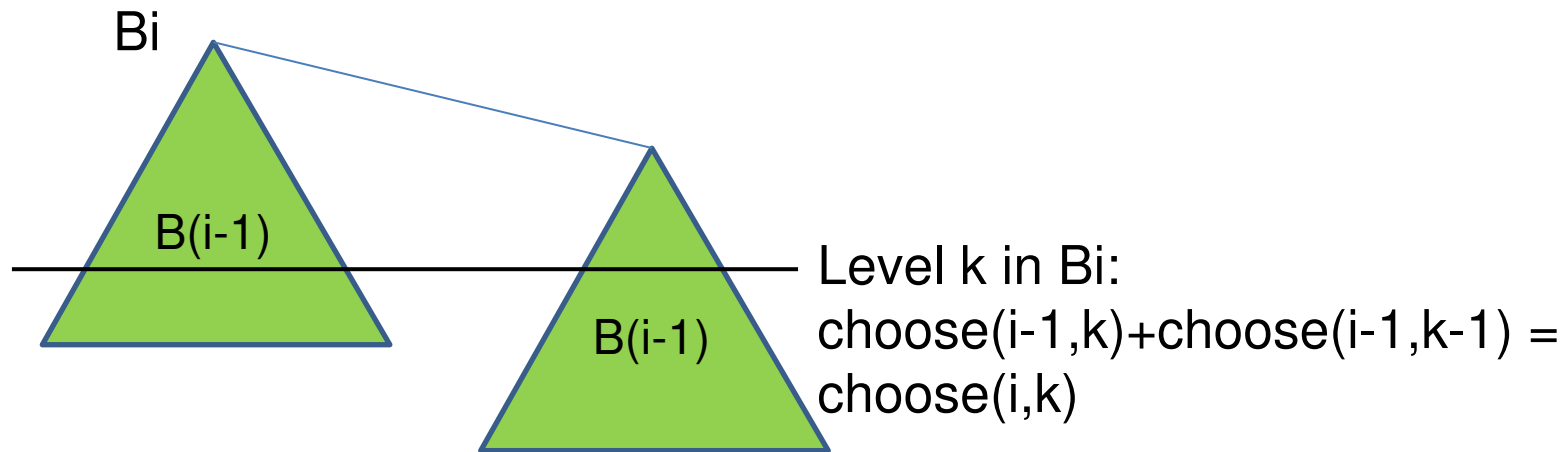


This naming corresponds to a [pre-order traversal](#) numbering of the binomial tree.

Other namings: post-order numbering, level numbering (BFS), ...

For any  $i$ : Find parent in  $O(k)$  steps (or even  $O(\log k)$  steps,  $O(1)$  if  $\text{lsb}(x)$  in hardware), find each child in  $O(1)$  steps

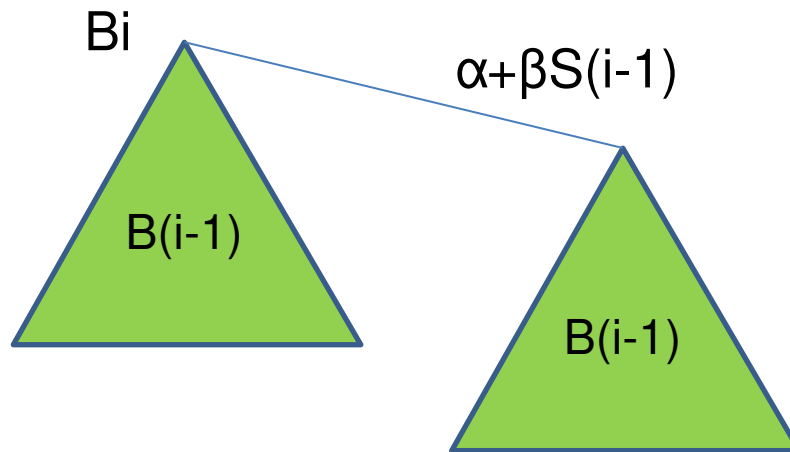
## Binomial tree: Alternative definition (and proof of 3<sup>rd</sup> property)



### Properties:

- $B_i$  has  $2^i$  nodes,  $i \geq 0$
- $B_i$  has  $i+1$  levels,  $0, \dots, i$
- $B_i$  has  $\text{choose}(i,k) = i! / ((i-k)!k!)$  nodes at level  $k$

## Binomial tree gather/broadcast/reduction



$T(i)$ : Processing time  
 $S(i)$ : Data volume

Optimality in linear cost model:

Since the processing time of the two subtrees is the same,  
 transfer between root and subtree root incurs no delay

## Structural properties of trees and collectives

Completion times for Broadcast, Reduction, Gather and Scatter with binomial trees can be expressed as recurrence relations

... and solved by standard techniques

## Gather:

- $T(i) = T(i-1) + \alpha + \beta S(i-1)$
- $T(1) = 0$
- $S(1) = m/p$
- $S(i) = S(i-1) + S(i-1)$

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

## Scatter:

- $T(i) = \alpha + \beta S(i-1) + T(i-1)$

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

## Broadcast:

- $T(i) = \alpha + \beta S(i-1) + T(i-1)$
- $S(i) = m$

$$T(i) = (\log p)(\alpha + \beta m)$$

## Reduction:

- $T(i) = T(i-1) + \alpha + \beta S(i-1) + \gamma S(i-1)$

$$T(i) = (\log p)(\alpha + \beta m + \gamma m)$$

Very different recurrences in LogGP model

## Broadcast/scatter completion times in $\text{Log}(G)P$ , $0 \leq g$

Broadcast, single item ( $\text{Log}P$ ):

- $T(P) = \min_{1 \leq i < P} \max[o + T(P-i) + (g-o), o + L + o + T(i)]$
- $T(1) = 0$

Linear algorithm,  
star-tree

Scatter, k item ( $\text{Log}P$ ):

$$T^k(P) = \min_{1 \leq i < P} \max[o + (ki-1)g + T^k(P-i) + (g-o), o + L + (ki-1)g + o + T^k(i)]$$

$$T^k(1) = 0$$

Best possible:  $T^k(P) = o + ((P-1)k-1)g + L + o$



Scatter, single item ( $\text{Log}GP$ ):

- $T(P) = \min_{1 \leq i < P} \max[o + T(P-i) + (g-o), o + L + (i-1)G + o + T(i)]$
- $T(1) = 0$

Scatter, k item ( $\text{Log}GP$ ) assuming tree communication

- $T^k(P) = \min_{1 \leq i < P} \max[o + (ik-1)G + T^k(P-i) + (g-o), o + L + (ik-1)G + o + T^k(i)]$
- $T^k(1) = 0$

Broadcast, single item (LogP):

- $T(P) = \min_{1 \leq i < P} \max[o + T(P-i) + (g-o), o + L + o + T(i)]$
- $T(1) = 0$



Root in set of  $P-i$  processors sends item to root in other set of  $i$  processors with overhead time  $o$ . Root is ready for next message after  $g-o$  time units (next message can be injected after the gap  $g$ , but the overhead can be overlapped)



Root in set of  $i$  processors receives item after  $o + L + o$  time units and can start broadcast over the  $i$  processors



## Master thesis?

For  $\text{Log}(G)P$ , best possible completion times depends on the values of  $L$ ,  $o$ ,  $g$ ,  $G$ ,  $P$ . Closed form expression (and best tree) can sometimes be found.

**Claim: Best solution can be found by dynamic programming**

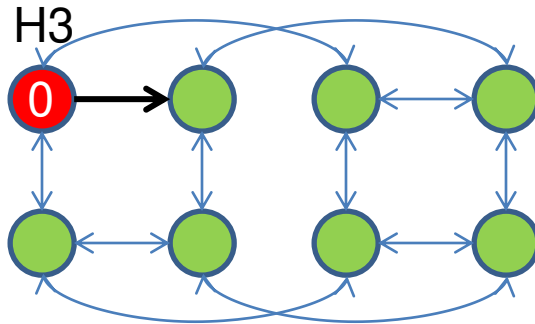
Richard M. Karp, Abhijit Sahay, Eunice E. Santos, Klaus E. Schauer: Optimal Broadcast and Summation in the  $\text{Log}P$  Model. SPAA 1993: 142-153

Eunice E. Santos: Optimal and Near-Optimal Algorithms for  $k$ -Item Broadcast. J. Parallel Distrib. Comput. 57(2): 121-139 (1999)

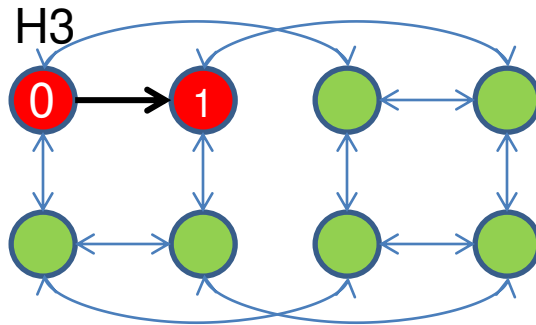
Albert D. Alexandrov, Mihai F. Ionescu, Klaus E. Schauer, Chris J. Scheiman:  $\text{LogGP}$ : Incorporating Long Messages into the  $\text{Log}P$  Model for Parallel Computation. J. Parallel Distrib. Comput. 44(1): 71-79 (1997)

Eunice E. Santos: Optimal and Efficient Algorithms for Summing and Prefix Summing on Parallel Machines. J. Parallel Distrib. Comput. 62(4): 517-543 (2002)

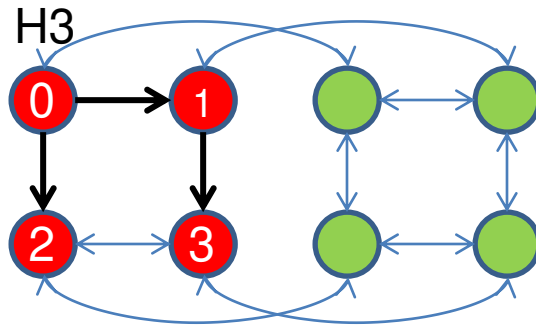
## Binomial tree broadcast in hypercube



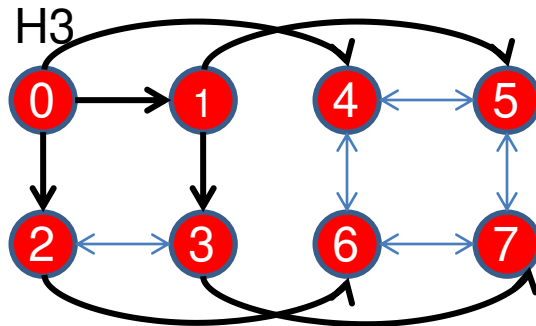
Round 0:  
If processor has data,  
flip bit 0  
Send block



Round 1:  
If processor has data,  
flip bit 1  
Send block



Round 2:  
If processor has data,  
flip bit 2  
Send block



Round  $i$ :  
 If processor has data,  
 flip bit  $i$   
 Send block

### Theorem:

Binomial tree  $B_i$  can be optimally embedded into hypercube  $H_i$ ,  
 each edge in  $B_i$  corresponds to an edge in  $H_i$ :

- Dilation 1: every edge in  $B_i$  is mapped to an edge in  $H_i$  (not a path)
- Congestion 1: There is at most one edge in  $B_i$  mapping to an edge in  $H_i$

Does not contradict NP-completeness of  
 general embedding problem

## Embedding, terminology

Let  $\Gamma: G \rightarrow H$  be an embedding (injective mapping) of a guest graph  $G=(V,E)$  into a host graph  $H=(V',E')$  with a path function  $R(u,v)$  that maps  $e=(u,v)$  to a path from  $\Gamma(u)$  to  $\Gamma(v)$  in  $H$ .

- The congestion of  $\Gamma$  is the maximum number over all  $e'$  in  $V$  of edges  $e=(u,v)$  in  $G$  such that there is a path  $R(u,v)$  in  $H$  from  $\Gamma(u)$  to  $\Gamma(v)$  that includes  $e'$  for some  $e'$  in  $H$
- The dilation of  $\Gamma$  is the longest path in  $H$  from  $\Gamma(u)$  to  $\Gamma(v)$  over all edges  $e=(u,v)$  in  $G$
- The expansion is the ratio  $|V'|/|V|$

Loosely:

- Congestion: How many parallel communication operations need the same edge? Can cause slowdown proportional to congestion
- Dilation: What extra distance must a message travel? Increases latency (by length of path), can decrease bandwidth

Optimal embedding: Congestion 1, dilation 1

Paul D. Manuel, Indra Rajasingh, R. Sundara Rajan, N. Parthiban, T. M. Rajalaxmi: A Tight Bound for Congestion of an Embedding. CALDAM 2015: 229-237

Optimizing congestion, dilation, expansion are different problems

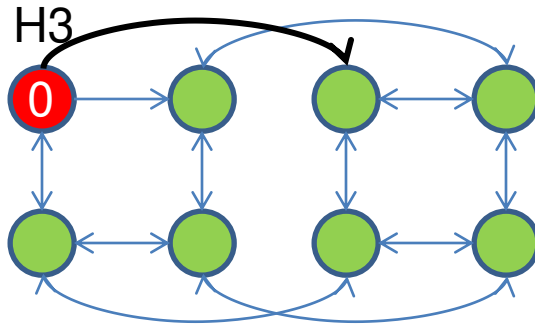
## Network design vehicle approach

Many, many results on embedding different, fixed networks into other networks (e.g., binomial tree into hypercube, trees into meshes, ...); research in 80ties, 90ties

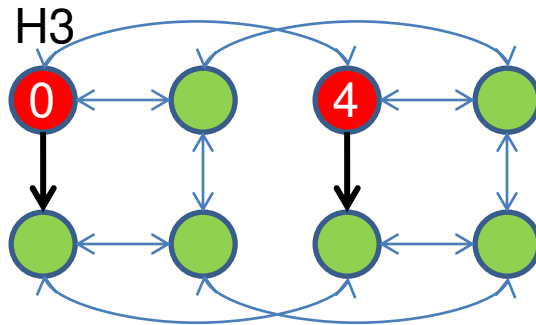
1. Chose convenient network for design of algorithm
2. Prove properties
3. Embed into actual system network
4. Use embedding results to prove further properties



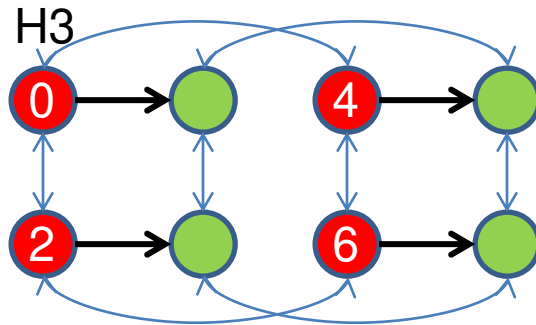
## Binomial tree scatter in hypercube



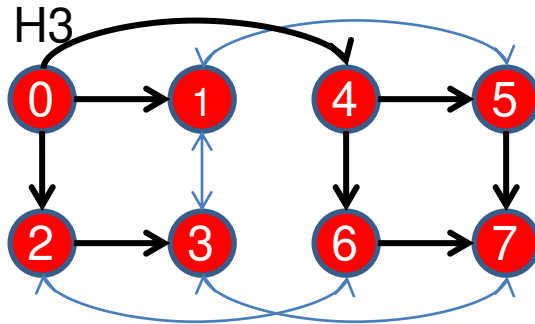
Round 0:  
If processor has  
data, flip bit d-1  
Send  $p/2$  blocks



Round 1:  
If processor has  
data, flip bit d-2  
Send  $p/4$  blocks



Round 2:  
 If processor has  
 data, flip bit d-3  
 Send  $p/8$  blocks

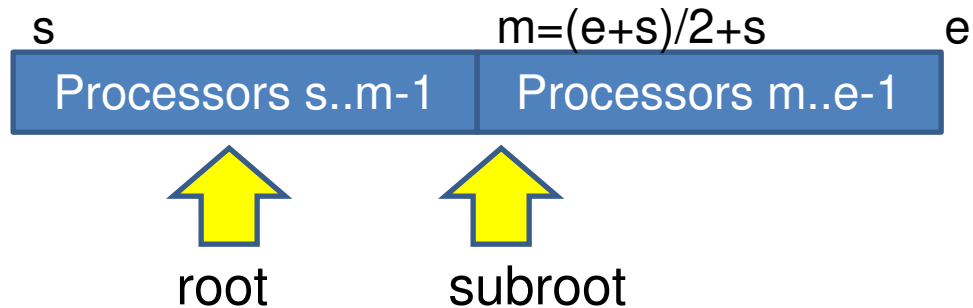


**Result:** Different embedding of binomial tree into hypercube.

**Question :**

What if  $p$  is not a power of two? Possible to achieve  $\text{ceil}(\log_2 p)$  rounds?

## Binomial tree-like, divide-and-conquer scatter (≠power-of-two)



**Idea** : Divide processors into two parts, root sends blocks for other half to subroot; recurse

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

sendbuf only significant on root; data blocks in rank order

```

Scatter(void *sendbuf, ..., void *recvbuf, ...,
        int s, int e, int root, MPI_Comm comm)
{
    // base case: only one process
    if (s+1==e) {
        memcpy(recvbuf, sendbuf, ...); // should be typed
        // can (should) be avoided for non-roots
        return;
    }

    // ...
}

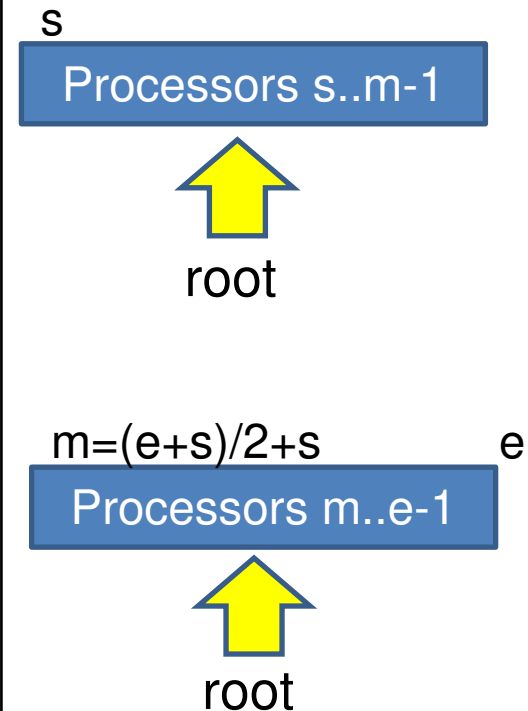
```

MPI\_Scatter calls recursive Scatter with s=0 and e=p,  
sendbuf=void\*, unless rank==root

```

n = (e-s)/2; m = s+n;
if (root<m) {
  subroot = m; blocks = e-s-n;
  if (rank<m) {
    if (rank==root) sendbuf += m;
    e = m; newroot = root;
  } else {
    s = m; newroot = subroot;
  }
} else {
  subroot = s; blocks = n;
  if (rank>=m) {
    s = m; newroot = root;
  } else {
    e = m; newroot = subroot;
  }
}

```



```

// root sends data to subroot
if (rank==root) {
    MPI_Send(sendbuf,blocks,...,subroot,...,comm);
} else if (rank==subroot) {
    sendbuf = (void*)malloc(blocks*...);
    MPI_Recv(sendbuf,blocks,...,root,...,comm);
}

// recurse
Scatter(sendbuf,...,recvbuf,...,newroot,s,e,comm);
if (rank!=root&&sendbuf!=NULL) free(sendbuf);
return;

```

- Algorithmic latency is constant, root/subroot can start immediately
- Contiguous blocks from sendbuf
- Temporary buffers for subroots



$$T_{\text{scatter}}(m) = \text{ceil}(\log p)\alpha + (p-1)/p \beta m$$

Optimal in both  $\alpha$  and  $\beta$  terms

**Note :**

Same scheme for reduction/gather will have  $O(\log p)$  harmful algorithmic latency

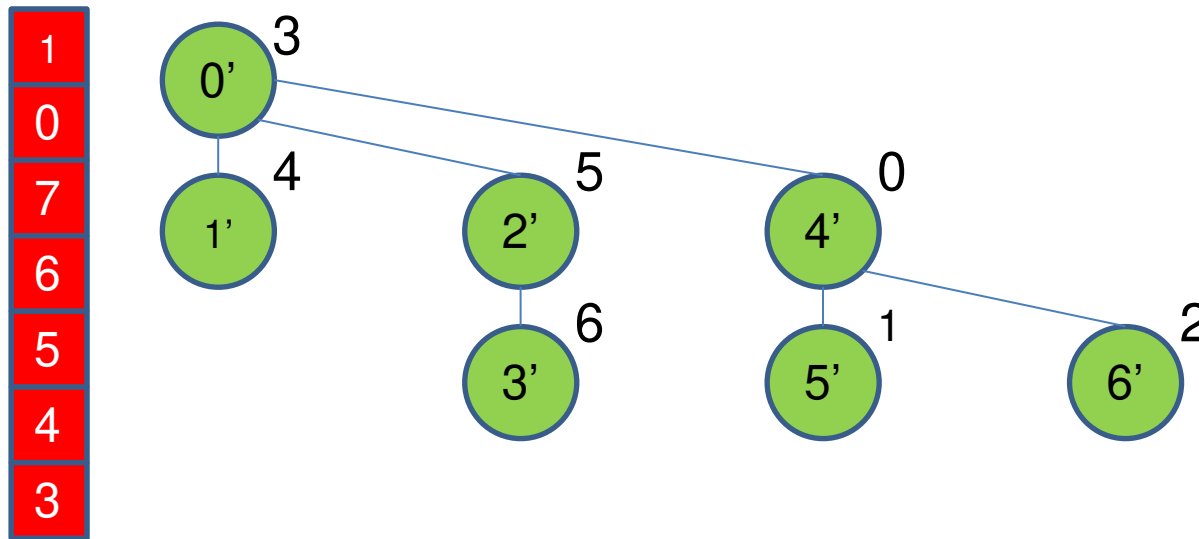
**Remark:**

Van de Geijn and others call these algorithms “MST (Minimum Spanning Tree)”... **Misnomer:**

- Minimum wrt. to what?
- Any algorithm for broadcast etc. must use one (or more) spanning trees, since all processors must be reached

## Binomial gather/scatter with root≠0

Standard solution: Use virtual rank' = (rank-root+size)%size



**Drawback:** Result not in rank order (shifted), requires local reordering at root, further algorithmic latency

Divide-and-conquer gather does not have this drawback, received buffers always in rank order, still  $\text{ceil}(\log p)$  rounds

## Final remark: binomial and k-nomial trees

The binomial-tree algorithms and variants naturally extend to systems with k-ported communication.

The latency term becomes  $\text{ceil}(\log_{k+1} p)$

## Irregular collectives: Gather and scatter

- Irregular (v-) collectives algorithmically much more difficult than regular collectives seen so far.
- Not that many good results
- MPI libraries most often have **trivial implementations** : Same algorithm as corresponding, regular collective

What to do?

“Well-behaved”, irregular collectives: Gather, scatter, allgather, reduce\_scatter

Jesper Larsson Träff: Practical, linear-time, fully distributed algorithms for irregular gather and scatter. EuroMPI/USA 2017: 1:1-1:10

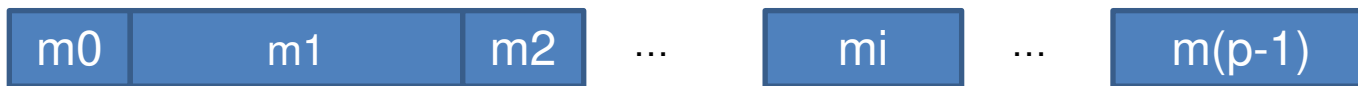
Jesper Larsson Träff, Andreas Ripke, Christian Siebert, Pavan Balaji, Rajeev Thakur, William Gropp: A Pipelined Algorithm for Large, Irregular All-Gather Problems. IJHPCA 24(1): 58-68 (2010)

J. L. Träff: An Improved Algorithm for (Non-commutative) Reduce-Scatter with an Application. PVM/MPI 2005: 129-137

And others...

## Linear time irregular gather

Let  $m_0, m_1, m_2, \dots, m_i, \dots, m_{p-1}$  be the (sizes of the) blocks contributed by the  $p$  processors. Assume blocks are to be gathered in rank order at some processor  $r$ ,  $0 \leq r < p$



```
MPI_Gatherv(sendbuf, scount, stype,
             recvbuf, rcounts, rdispl, rtype, root,
             comm) ;
```

MPI specifics:

- Only root knows all  $m_i$ , in `rcounts` vector (of size  $p$ )
- Root provides a displacement `rdispl[i]` for each  $m_i$

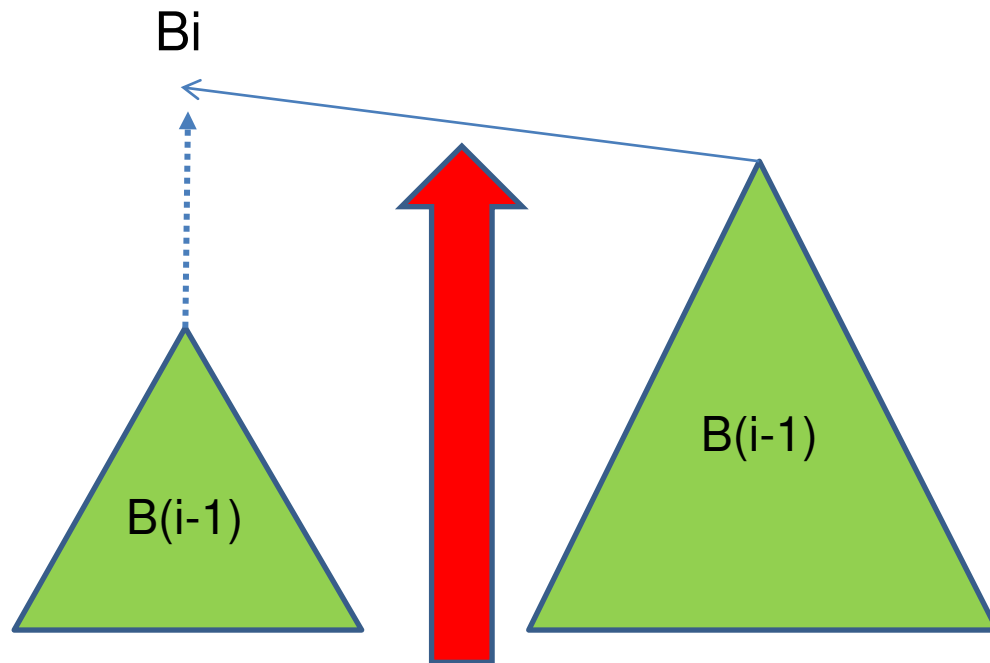
Algorithms (for regular collectives) so far have (implicitly) used a round-based, synchronous communication cost model

- Some pairs of processes active in each round
- Communication costs per round determined by largest  $m$  message transmitted
- Linear transmission cost  $t(m) = \alpha + \beta m$

The following algorithm for irregular gather/scatter (MPI\_Gatherv/MPI\_Scatterv) only assumes that processes start at the same time.

- Each message can be transmitted as soon as both sending and receiving process are ready

## Problem with static (rank-structured) binomial trees



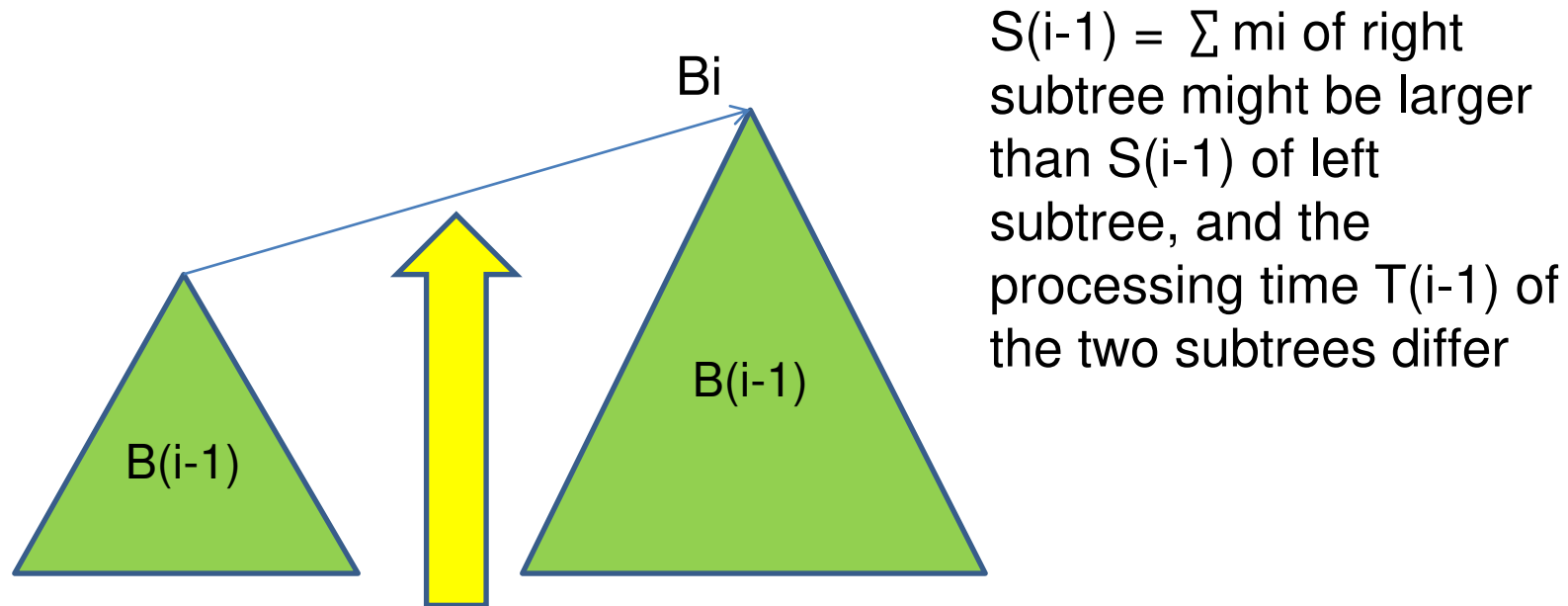
$S(i-1) = \sum m_i$  of right subtree might be larger than  $S(i-1)$  of left subtree, and the processing time  $T(i-1)$  of the two subtrees differ

**Delay** at root  $B_i$  (root of smaller, left tree) waiting for right tree to complete gather

Gather **delay**  $\delta = T_{\text{right}} - T_{\text{left}}$



## Problem with static (rank-structured) binomial trees



$S(i-1) = \sum m_i$  of right subtree might be larger than  $S(i-1)$  of left subtree, and the processing time  $T(i-1)$  of the two subtrees differ

No delay at root of  $B_i$  (root of larger, right tree), can start transmission as soon as right tree is completed

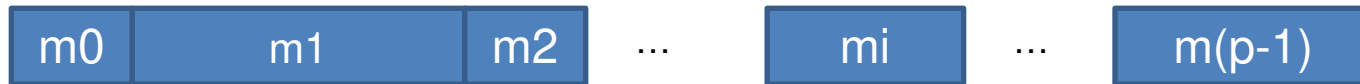
**Idea** : Avoid gather delays by letting faster constructed trees send to slower trees

**Extended hypercube** : Processors are organized in a hypercube  $H$ , and one extra non-hypercube edge between two processors is allowed.

An extended,  $d$ -dimensional hypercube  $H$  consists of two  $d-1$  dimensional, extended subcubes  $H'$  and  $H''$ , such that processors in  $H$  form a consecutive range of the processors in  $H'$  followed by the processors in  $H''$

Remark: The algorithm works also for incomplete hypercubes, where  $p$  is not a power of 2

Let  $m_0, m_1, m_2, \dots, m_i, \dots, m_{p-1}$  be the (sizes of the) blocks contributed by the  $p$  processors. Assume blocks are to be gathered in rank order at some processor  $r$ ,  $0 \leq r < p$



Lemma: There exist an extended hypercube algorithm that gathers the  $p$  blocks in rank order at some ( **unspecified** ) root  $r$  in

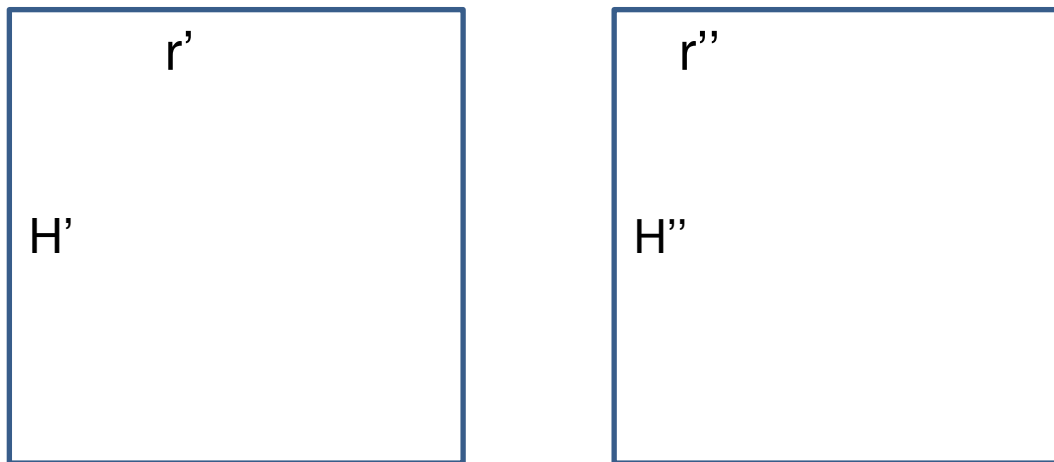
$$\text{ceil}(\log p)\alpha + \beta \sum_{i \neq r} m_i$$

time units

Proof: Induction on the dimension of the hypercube

Base case,  $d=0$ : Process  $i$  has its block  $m_i$ , nothing to gather

Induction step, assume claim holds for hypercubes of dimension  $d-1$ : Hypercube  $H$  of dimension  $d$  is formed by gathering data from one  $d-1$  dimensional cube  $H'$  to the root of the other  $d-1$  dimensional cube  $H''$ . Let  $r'$  and  $r''$  be the roots of  $H'$  and  $H''$ .



Assume that  $\sum_{i \in H', i \neq r'} m_i \leq \sum_{i \in H'', i \neq r''} m_i$  (with ties broken arbitrarily, but consistently).

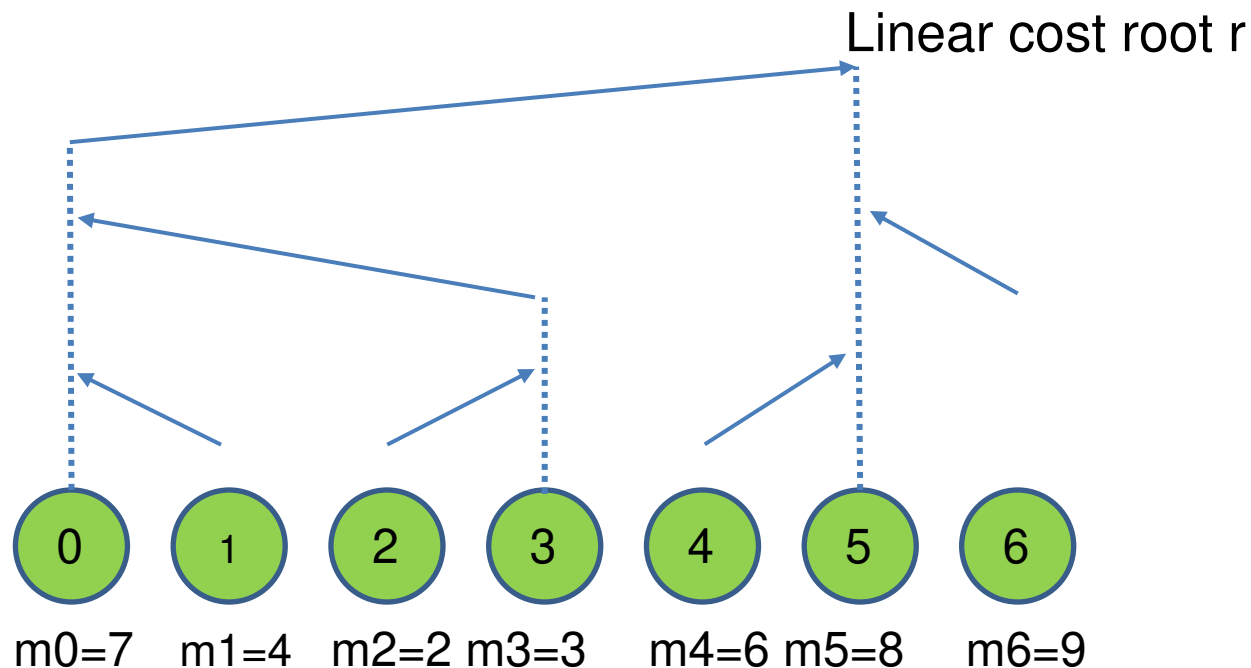
Send the data gathered at  $r'$  to  $r''$ .

Sending the data gathered at  $r'$  takes  $\alpha + \beta \sum_{i \in H'} m_i$  time units.

By the induction hypothesis, gathering the data at the other root  $r''$  has taken  $(d-1)\alpha + \beta \sum_{i \in H'', i \neq r''} m_i$  time units (which is not smaller than time of gathering in  $H'$ : **no gather delay**), for a total of  $d\alpha + \beta \sum_{i \in H, i \neq r''} m_i$  time units. The root in  $H$  becomes  $r''$ , and the claim follows.

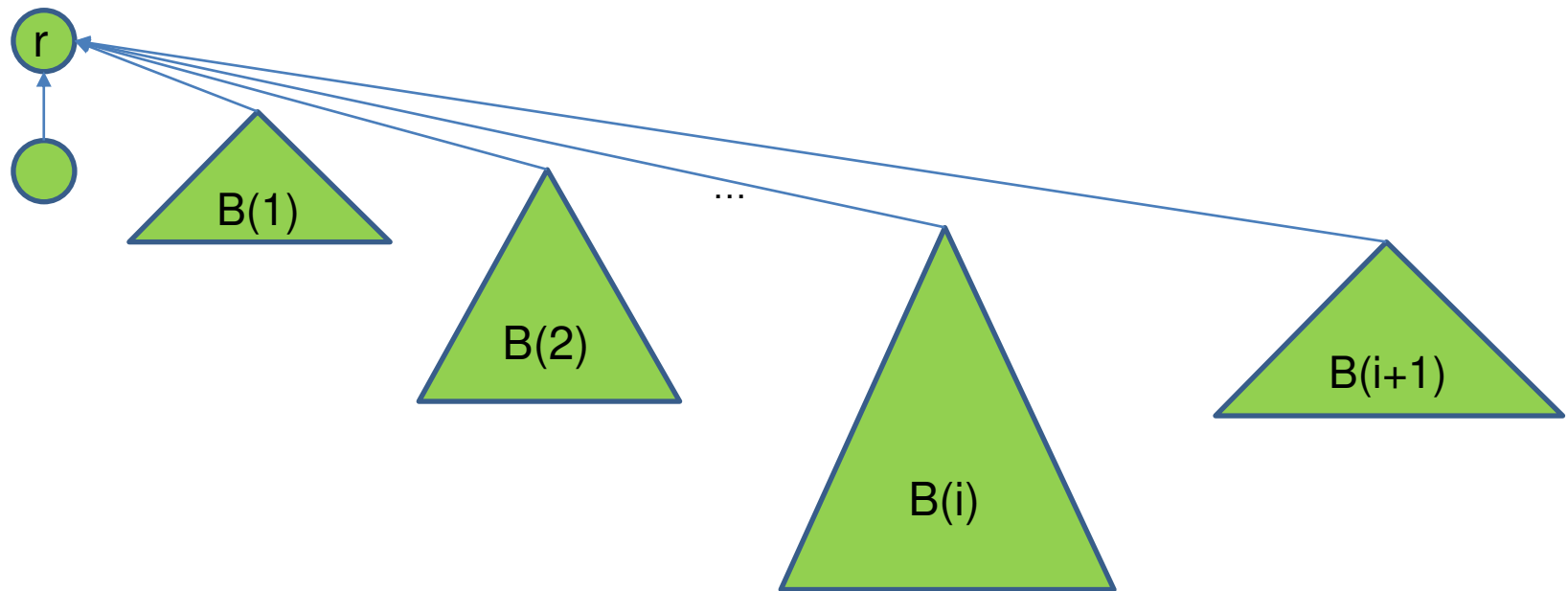
In  $H$ , there is communication along  $(r', r'')$  which may not be a hypercube edge, but is allowed in the extended hypercube

Data can be gathered in order: Assume  $H'$  contains the processors  $0, \dots, 2^{d-1}-1$ , and  $H''$  the processes  $2^{d-1}, \dots, 2^d-1$ . The buffer in  $H$  is allocated such that the  $H'$  blocks comes before the  $H''$  blocks

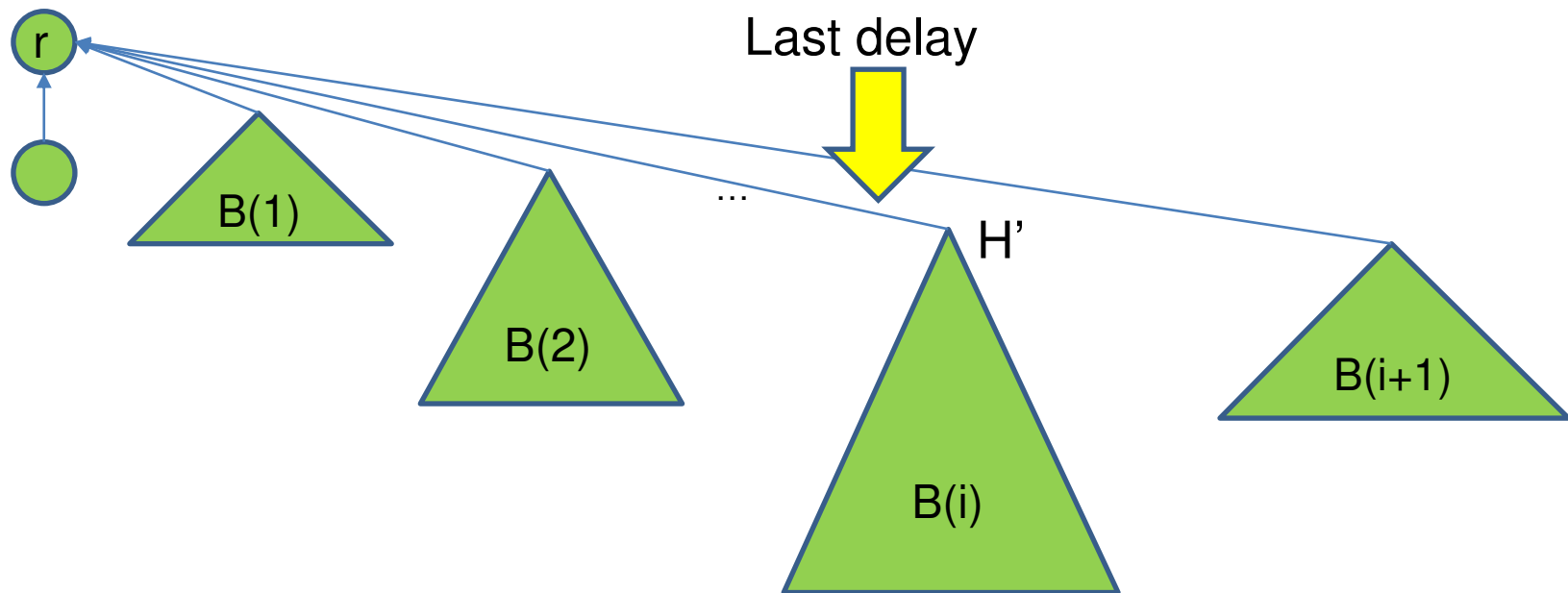


Proposition: There exist a hypercube algorithm that gathers the  $p$  blocks in rank order at a given root  $r$  in linear time at most  $\text{ceil}(\log p)\alpha + \beta \sum_{i \neq r} m_i + \beta \sum_{i \text{ in } H', i \neq r'} m_i$  time units for some subcube  $H'$

Proof: Break all ties in favor of given root  $r$ . Processor  $r$  receives from a sequence of optimal, binomial trees  $B(i)$ ,  $i=0,1,\dots,\text{ceil}(\log p)-1$



Let  $H'$  be the last tree causing a delay. Construction cost of  $H'$  is therefore larger than the cost of receiving from all previous subcubes (including delays). The cost of receiving from all  $\text{ceil}(\log p)$  subcubes is  $\text{ceil}(\log p)\alpha + \beta \sum_{i \neq r} m_i$ . The delay is at most  $i\alpha + \beta \sum_{i \text{ in } H', i \neq r} m_i - (i\alpha + \beta \sum_{i < H', i \neq r} m_i) \leq \beta \sum_{i \text{ in } H', i \neq r} m_i$

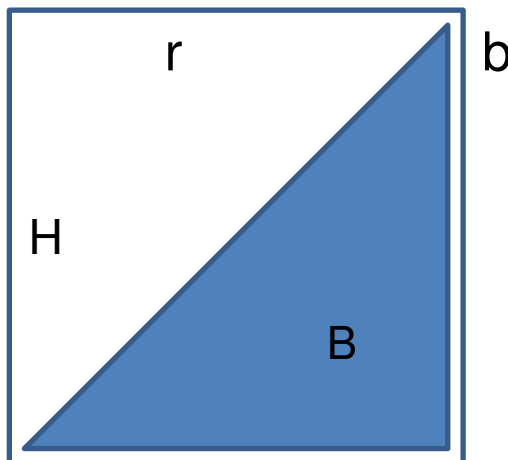




Question: How does  $r'$  know  $r''$  and vice versa?

Algorithm:

Maintain a fixed binomial tree  $B$  with root  $b$  in each hypercube  $H$  with root  $r$ . Maintain the invariant: The fixed root  $b$  knows  $r$ ,  $r$  knows  $b$ , and both  $r$  and  $b$  knows  $\sum_{i \neq r} m_i$  and  $\sum m_i$

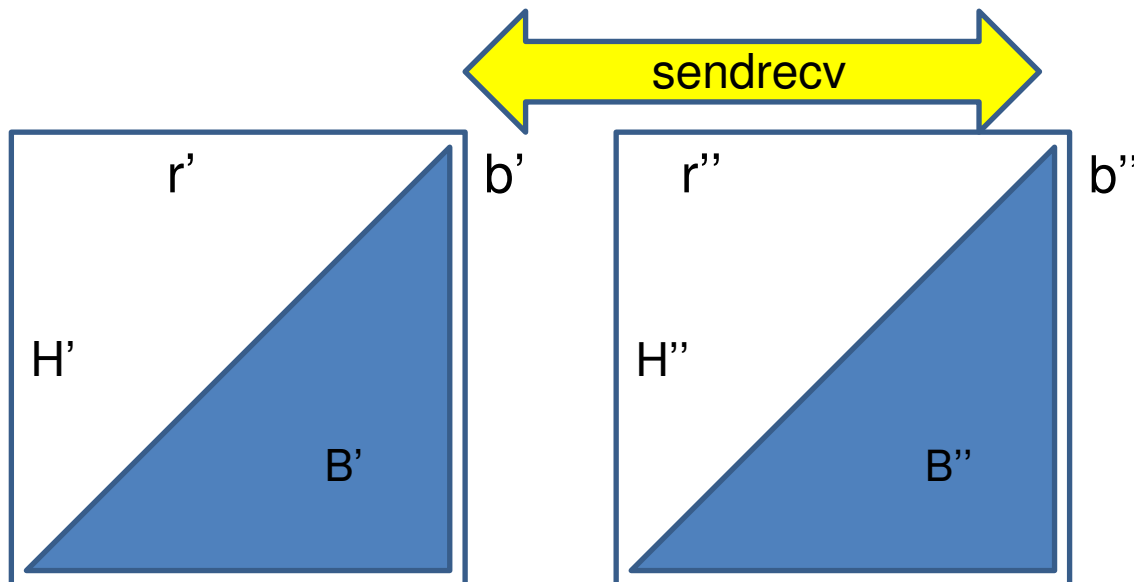


Invariant holds for 0-dimensional hypercube with  $r=b$

Algorithm:

Maintain a fixed binomial tree  $B$  with root  $b$  in each hypercube  $H$  with root  $r$ . Maintain the invariant: The fixed root  $b$  knows  $r$ ,  $r$  knows  $b$ , and both  $r$  and  $b$  knows  $\sum_{i \neq r} m_i$  and  $\sum m_i$

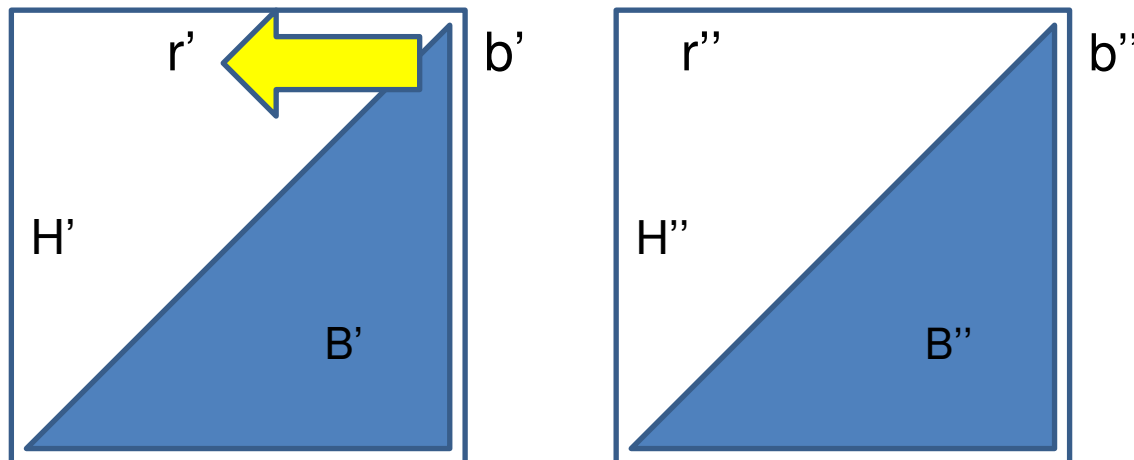
Step 1:  $b'$  and  $b''$  exchange  $(r', \sum_{i \neq r'} m_i, \sum m_i)$  and  $(r'', \sum_{i \neq r''} m_i, \sum m_i)$



Algorithm:

Maintain a fixed binomial tree  $B$  with root  $b$  in each hypercube  $H$  with root  $r$ . Maintain the invariant: The fixed root  $b$  knows  $r$ ,  $r$  knows  $b$ , and both  $r$  and  $b$  know  $\sum_{i \neq r} m_i$  and  $\sum m_i$

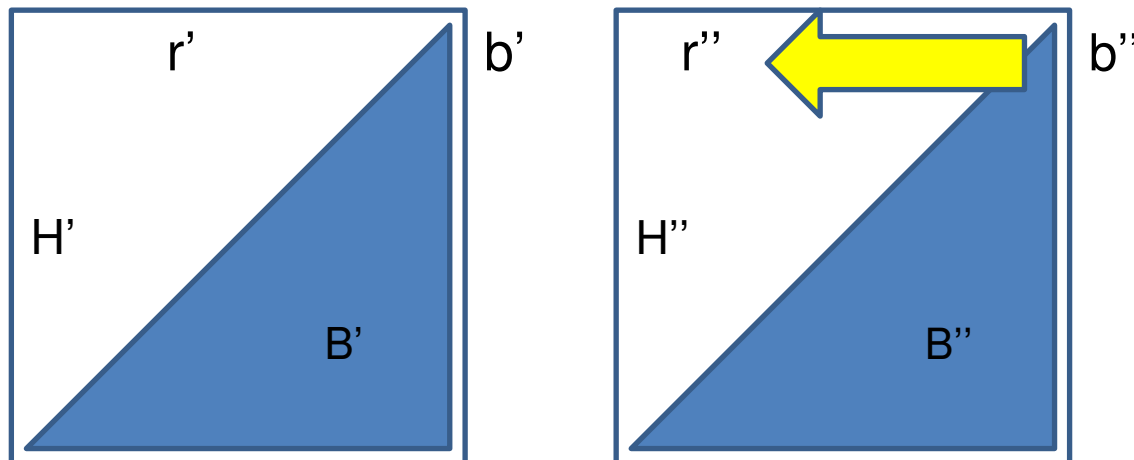
Step 2': Fixed root  $b'$  sends  $(r'', \sum_{i \neq r''} m_i, \sum m_i)$  to  $r'$  (if  $b' \neq r'$ )



Algorithm:

Maintain a fixed binomial tree  $B$  with root  $b$  in each hypercube  $H$  with root  $r$ . Maintain the invariant: The fixed root  $b$  knows  $r$ ,  $r$  knows  $b$ , and both  $r$  and  $b$  know  $\sum_{i \neq r} m_i$  and  $\sum m_i$

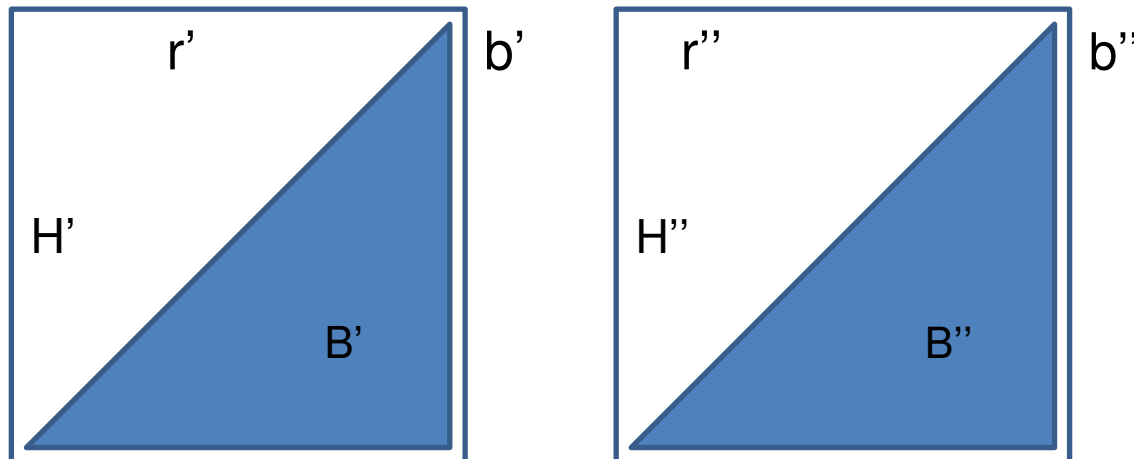
Step 2'': Fixed root  $b''$  sends  $(r', \sum_{i \neq r'} m_i, \sum m_i)$  to  $r''$  (if  $b'' \neq r''$ )



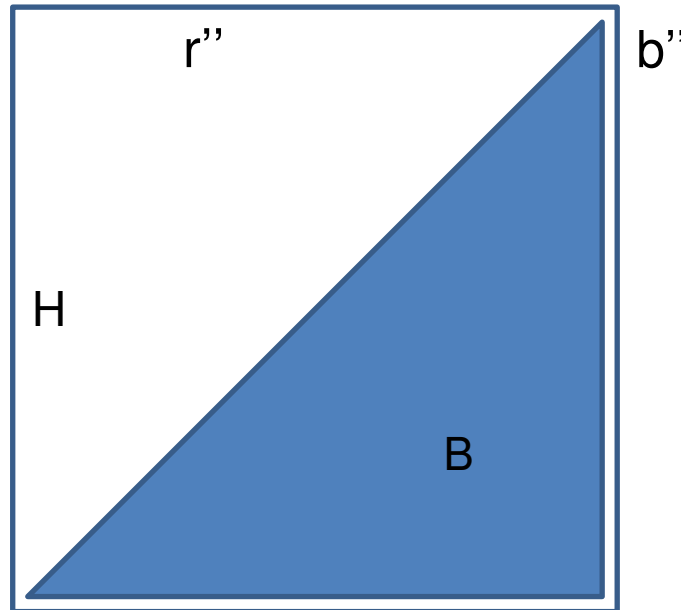
Algorithm:

Maintain a fixed binomial tree  $B$  with root  $b$  in each hypercube  $H$  with root  $r$ . Maintain the invariant: The fixed root  $b$  knows  $r$ ,  $r$  knows  $b$ , and both  $r$  and  $b$  knows  $\sum_{i \neq r} m_i$  and  $\sum m_i$

Step 3: Both hypercube roots  $r'$  and  $r''$  compare  $\sum_{i \neq r'} m_i$  and  $\sum_{i \neq r''} m_i$ , and decide which will be the new root in  $H$ . The new fixed root  $b''$  likewise decides



By the three steps, the invariant is maintained for  $H$  formed from  $H'$  and  $H''$ . The construction takes  $\text{ceil}(\log p)$  communication rounds, each consisting of 2 steps



Theorem: The irregular gather problem with block sizes  $m_0, m_1, \dots, m_{p-1}$ , and given root  $r$  can be solved in at most  $3\text{ceil}(\log p)\alpha + \beta \sum_{i \neq r} m_i + \beta \sum_{i \in H', i \neq r'} m_i$  time units

Jesper Larsson Träff: Practical, distributed, low overhead algorithms for irregular gather and scatter collectives. *Parallel Computing* 75: 100-117 (2018)

## Is this better than existing MPI\_Gather(v) implementations

Intel MPI 2017 on VSC-3 (Vienna Scientific Cluster),  
[www.vsc.ac.at](http://www.vsc.ac.at),  $p=500 \times 16=8000$  MPI processes, dual-rail  
 InfiniBand interconnect. For MPI\_Gatherv, Intel MPI uses a  
 binomial tree (can be configured), for MPI\_Scatterv a star (root  
 send to all)

Comparison against

- native MPI\_Gather ( **expectation** : for regular problems,  
 $\text{MPI\_Gather}(m) \leq \text{MPI\_Gatherv}(m)$ )
- native MPI\_Gatherv
- padded MPI\_Gather ( **expectation** :  $\text{Gatherv}(m) \leq$   
 $\text{MPI\_Allreduce}(1) + \text{MPI\_Gather}(m')$  for padded  $m'$  with  $m' \geq m$ )



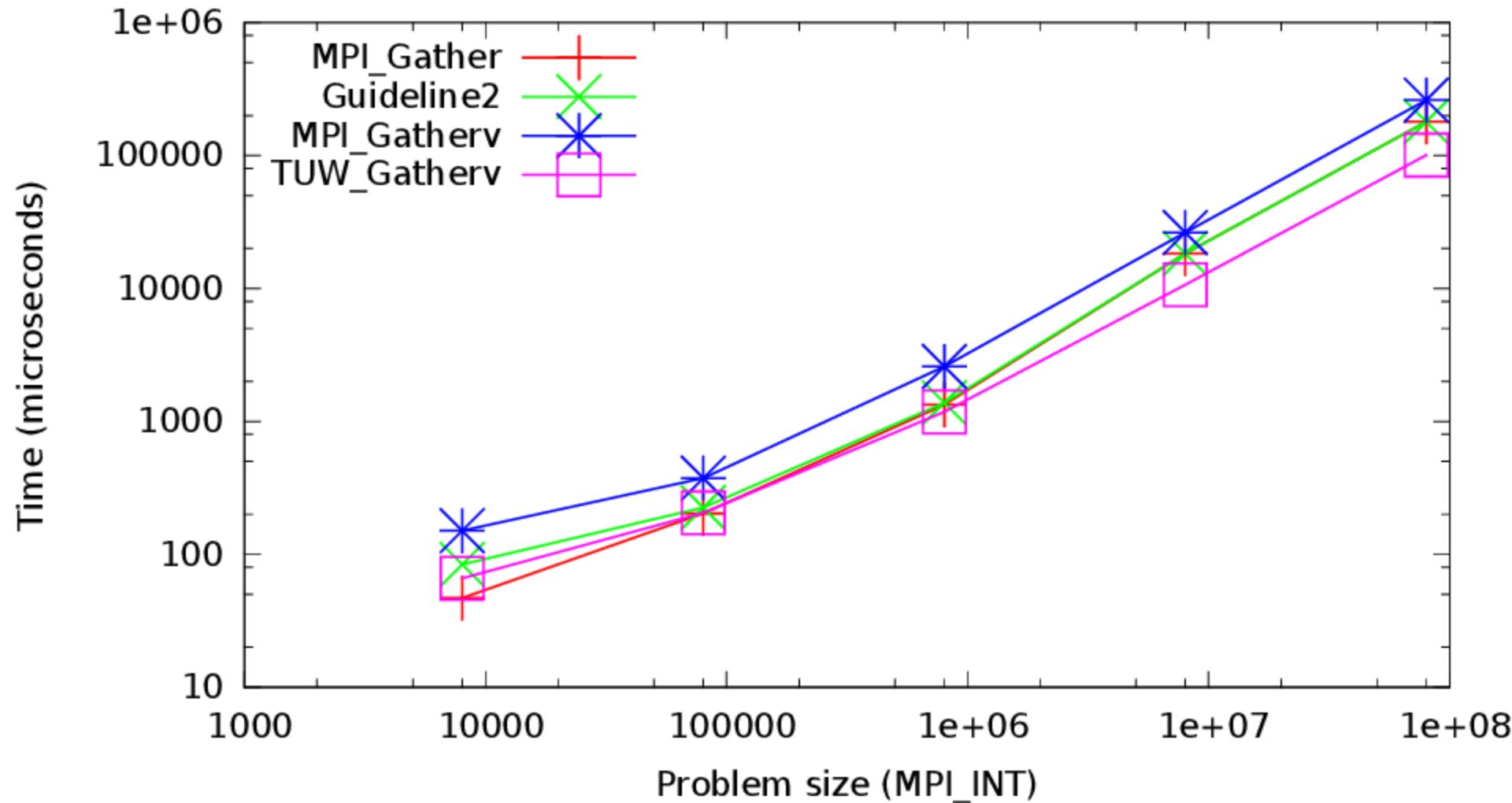
## Different problem types

- Same (all blocks same size)
- Random
- Bucket (fixed part plus random part)
- Increasing (block sizes increase linearly with rank)
- Decreasing (block sizes decrease)
- Spikes (some randomly chosen large blocks)
- Alternating (large/small blocks)
- Two blocks (first and large blocks small, all other unit size)

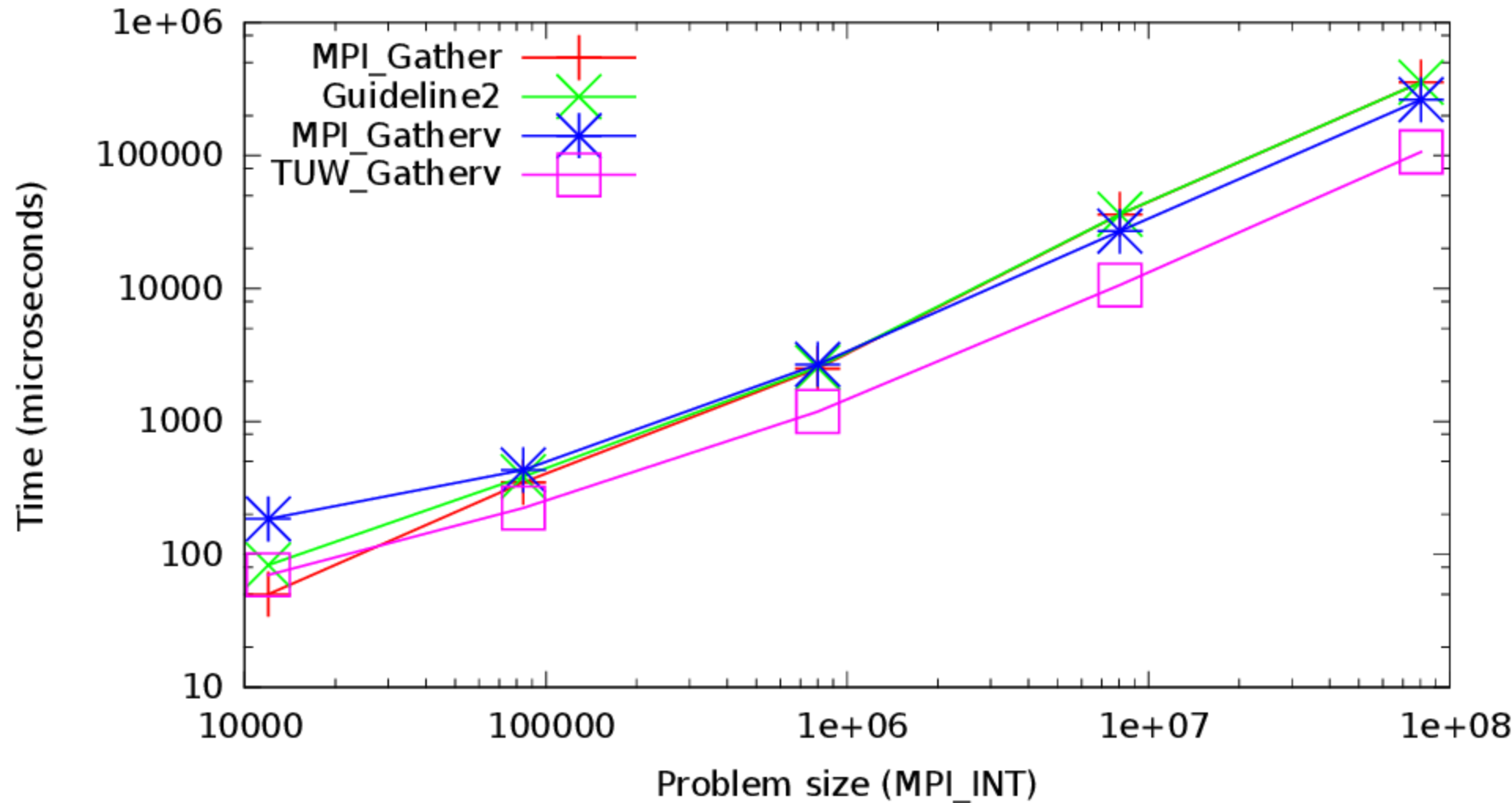
with increasing, total problem size  $m = \sum_{0 \leq i < p} m_i$

Findings say: **Yes, much better** (the block size aware irregular gather algorithm is implemented as TUW\_Gatherv)

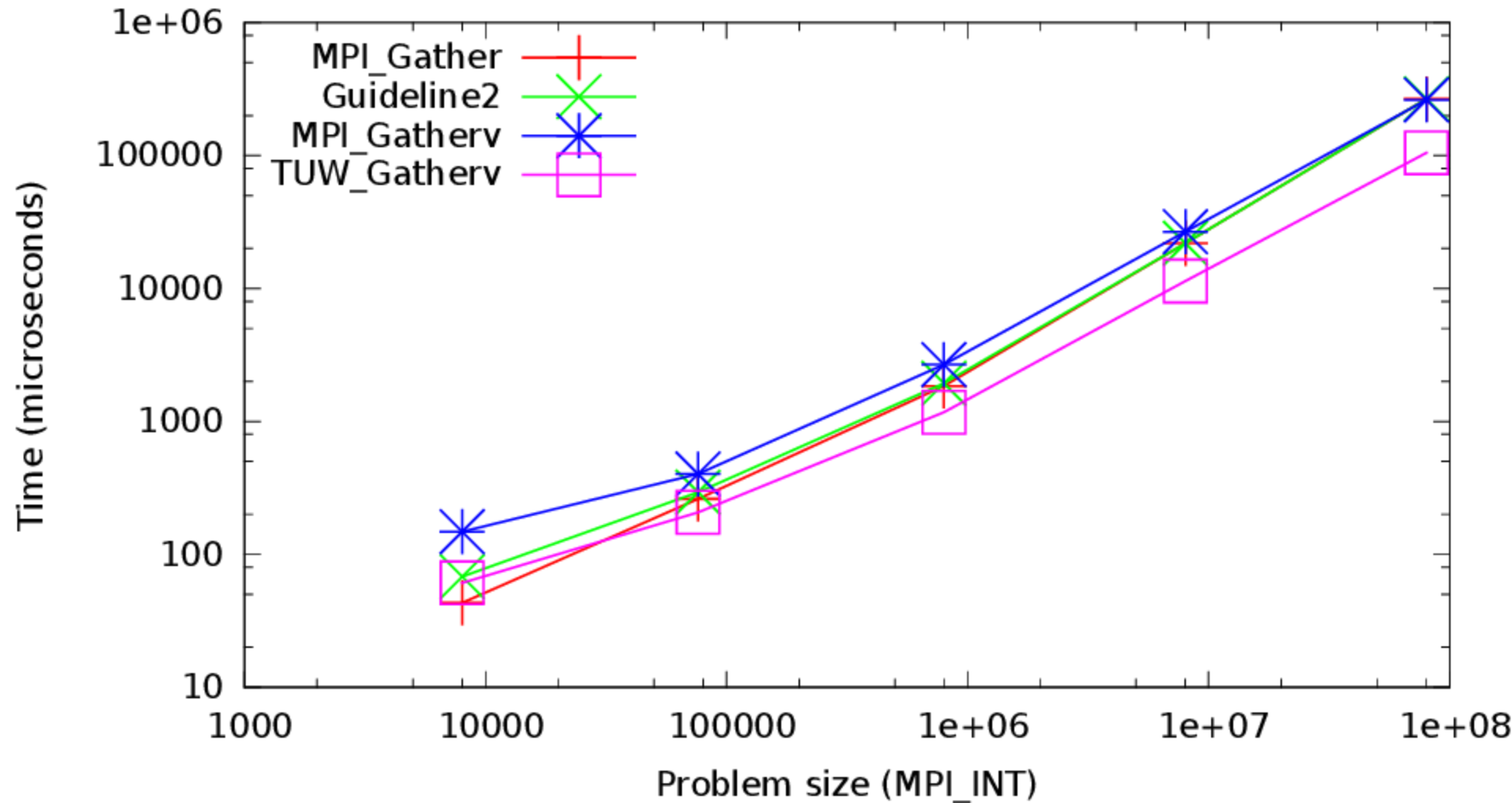
## Intel MPI, Same blocks



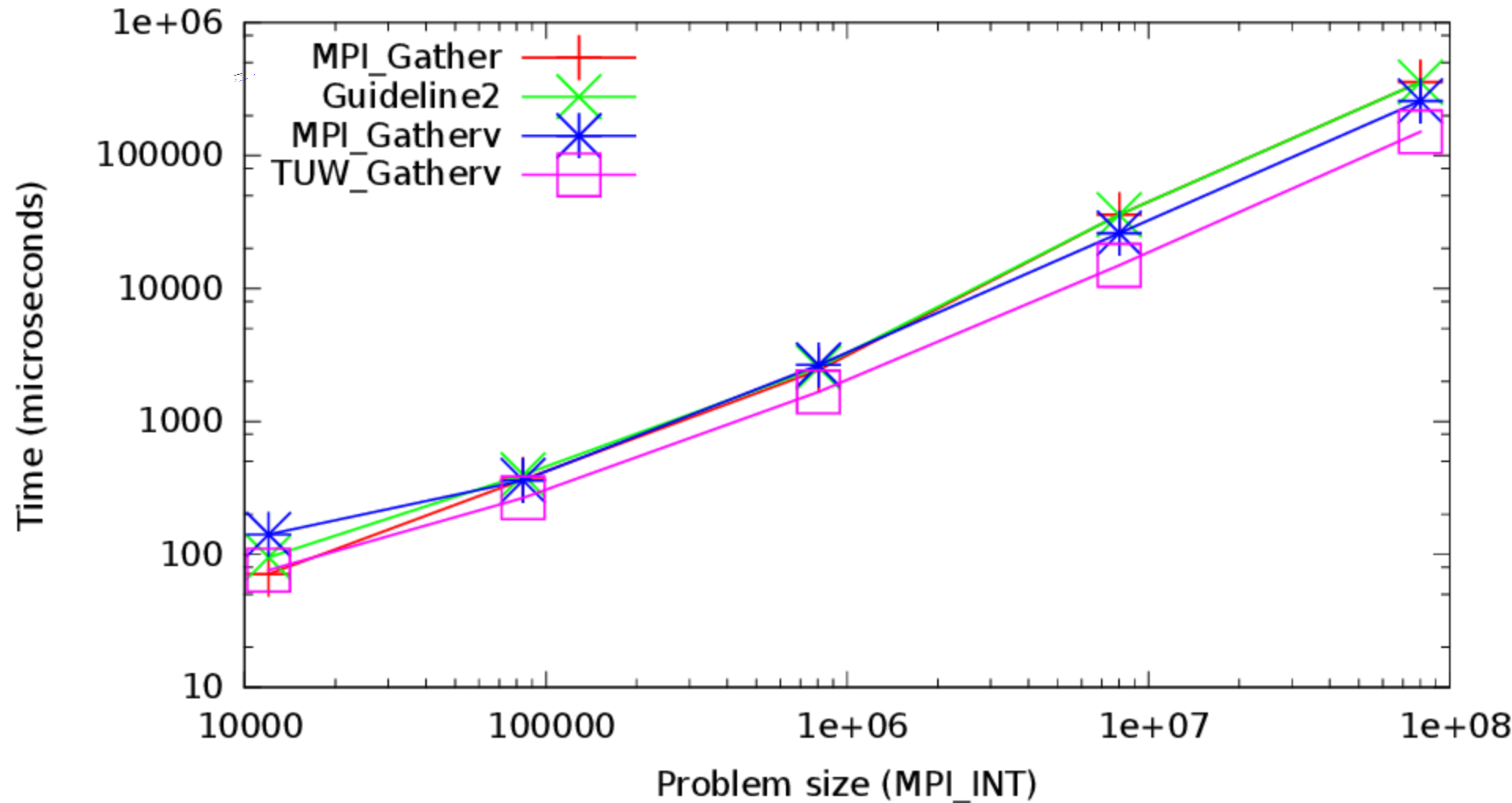
## Intel MPI, Random blocks



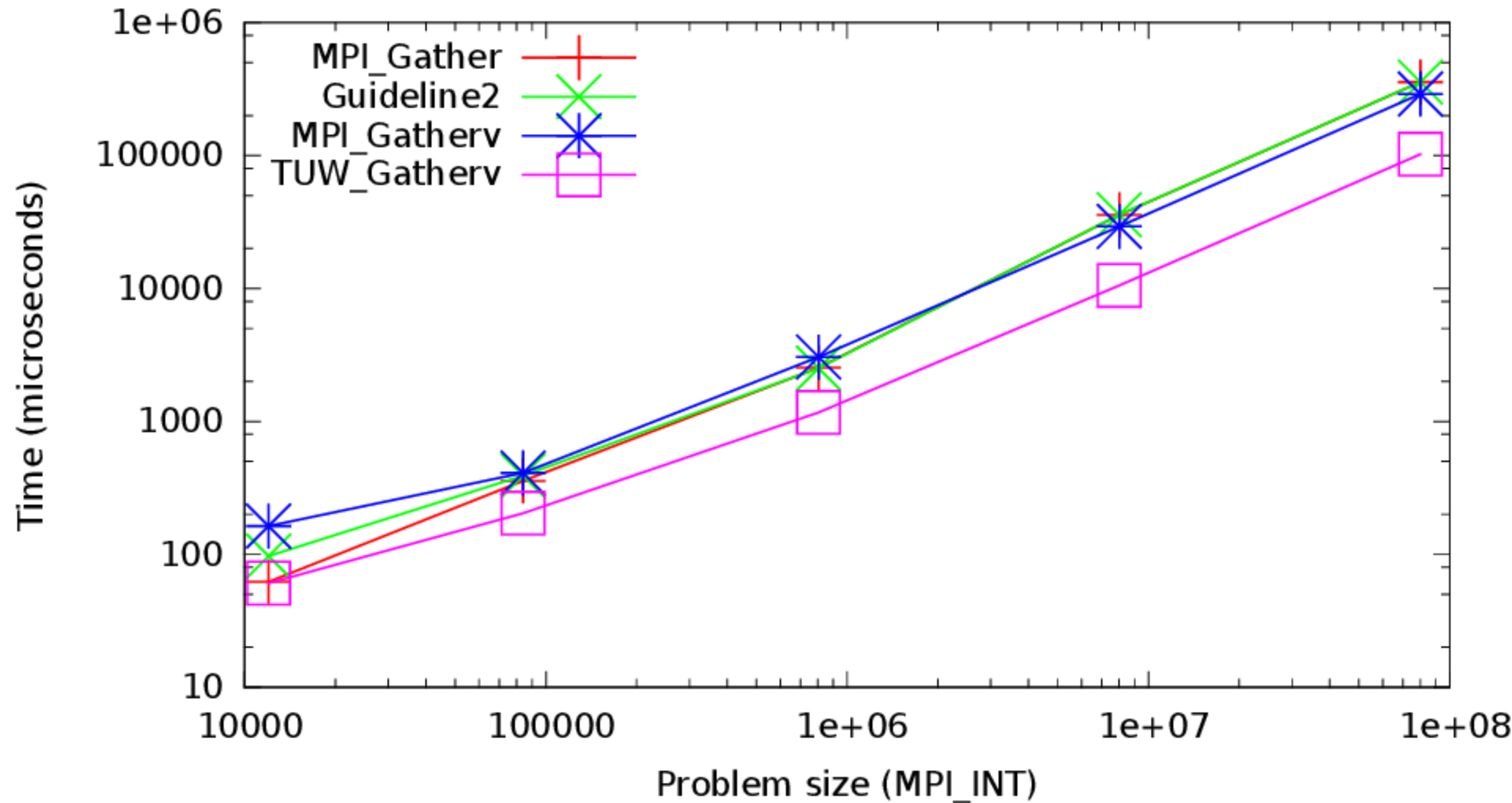
## Intel MPI, Random buckets



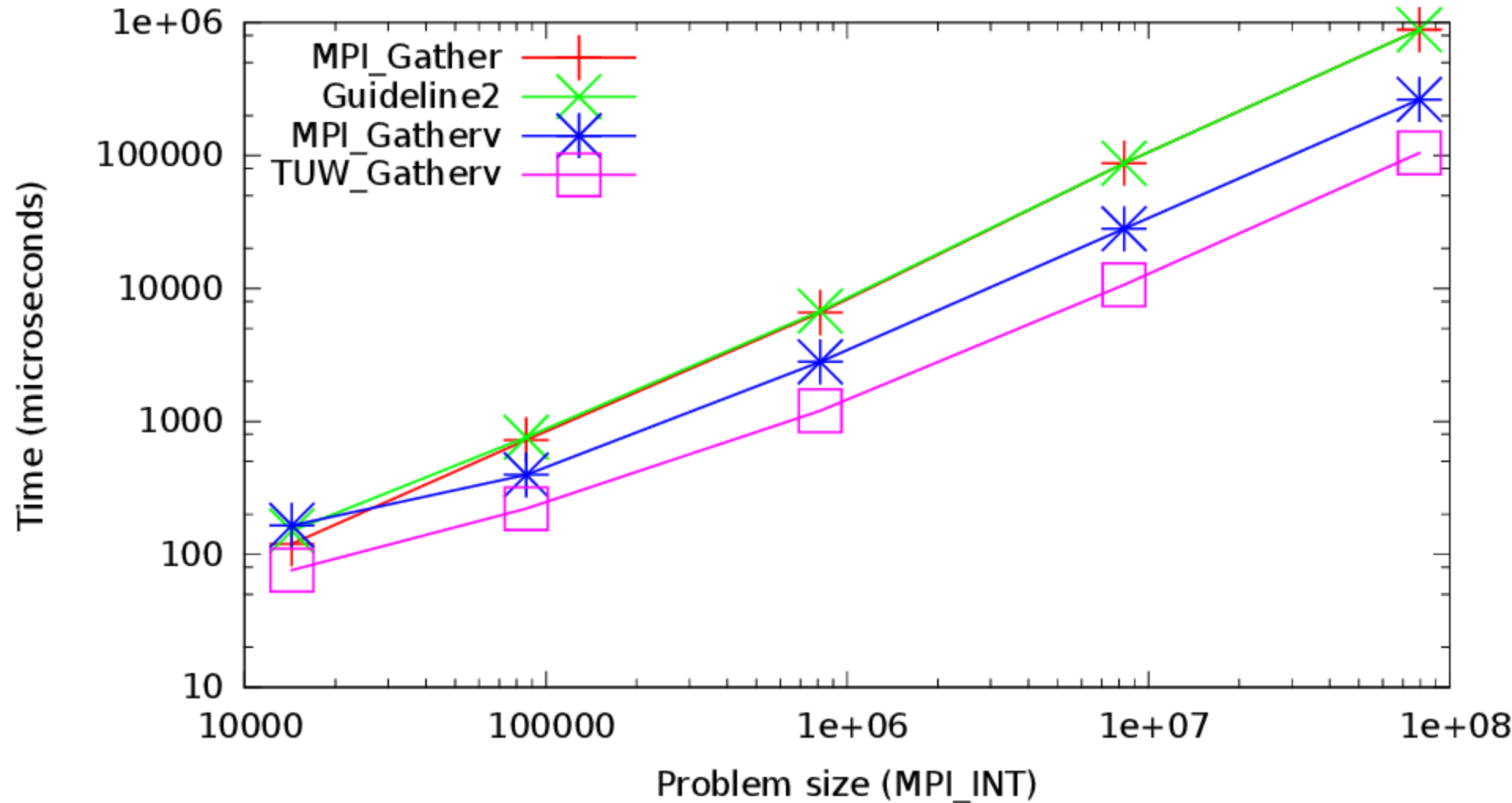
## Intel MPI, Increasing blocks



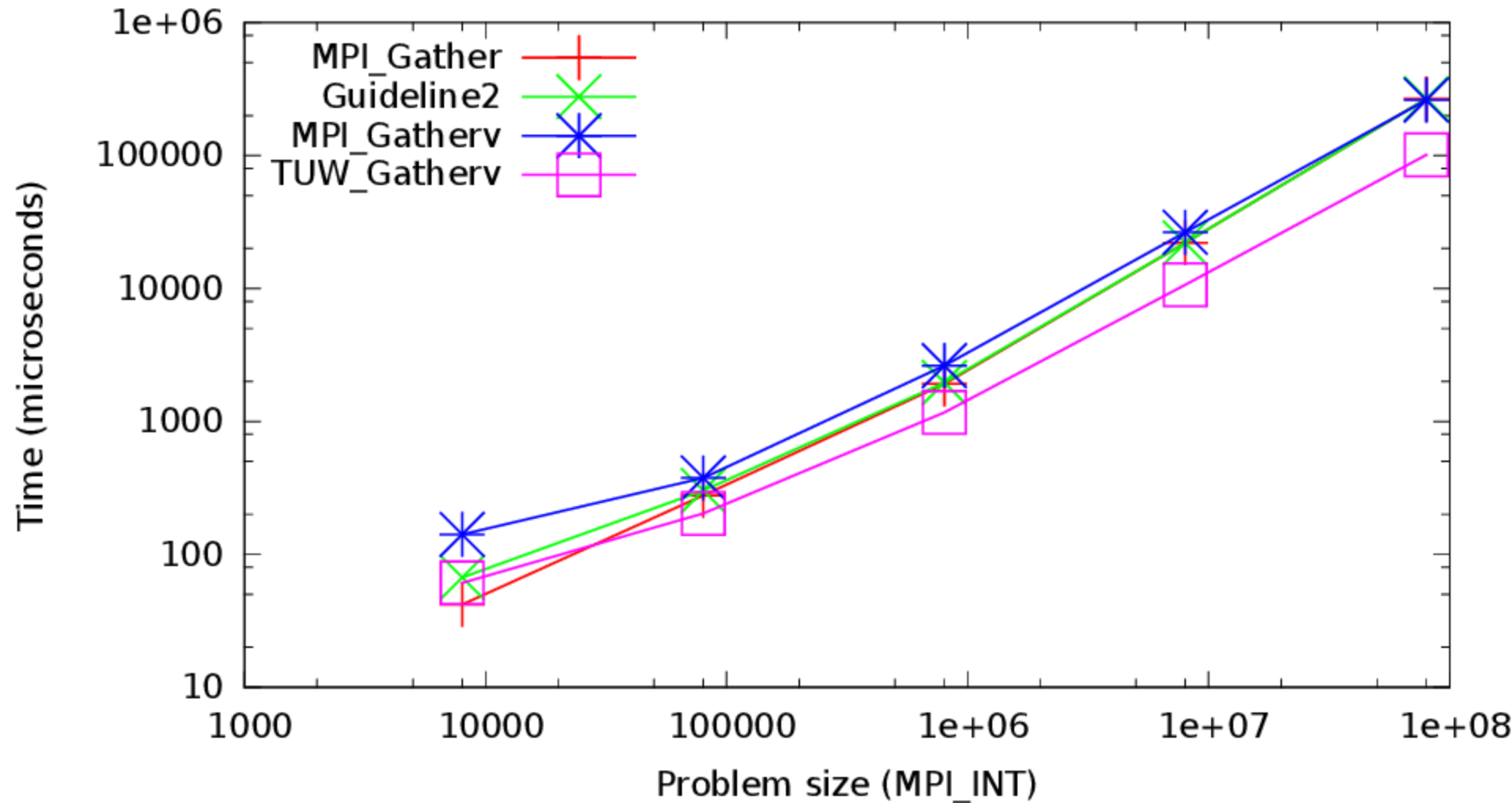
## Intel MPI, Decreasing blocks



## Intel MPI, Spiked blocks

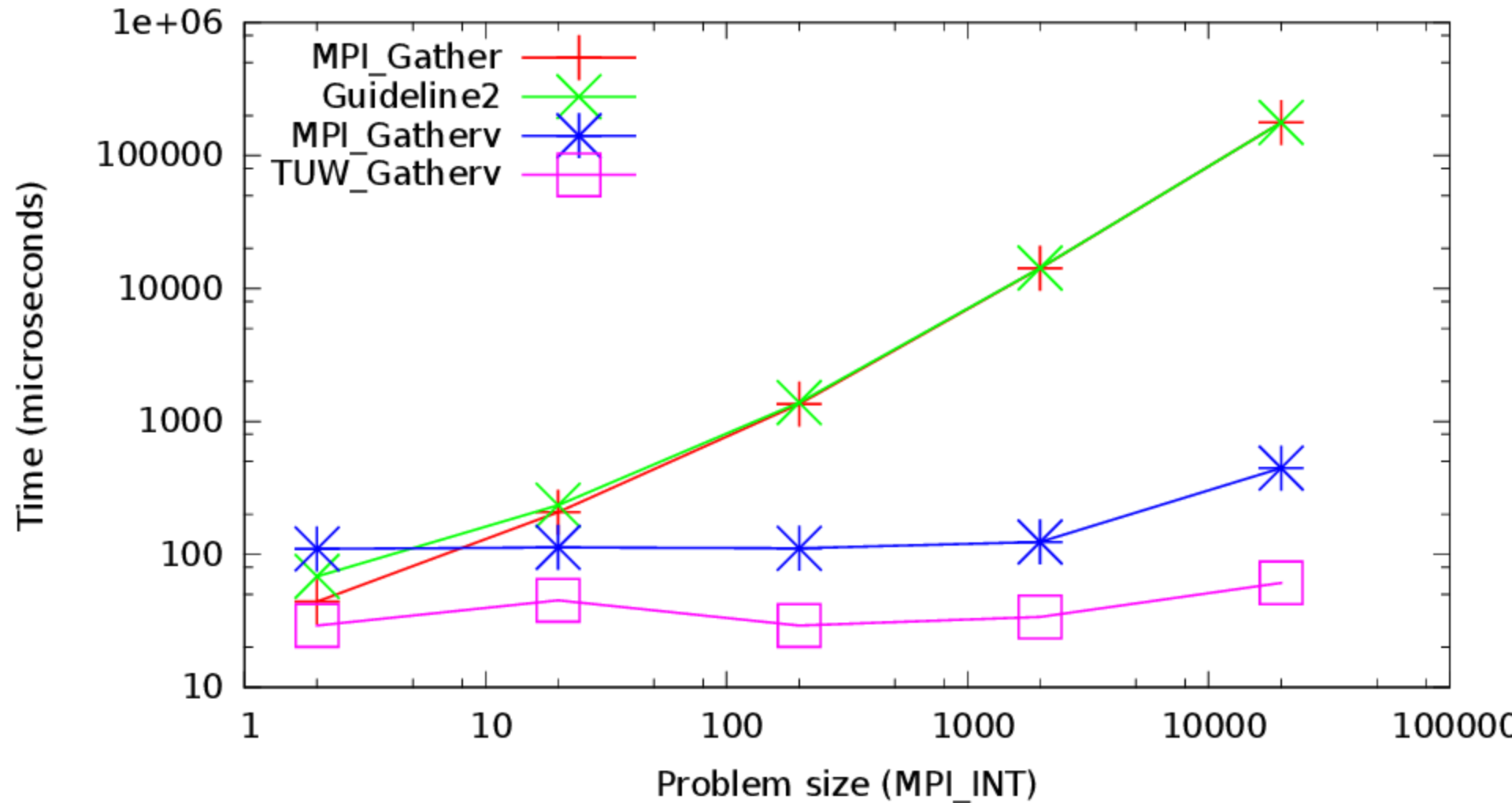


## Intel MPI, Alternating blocks

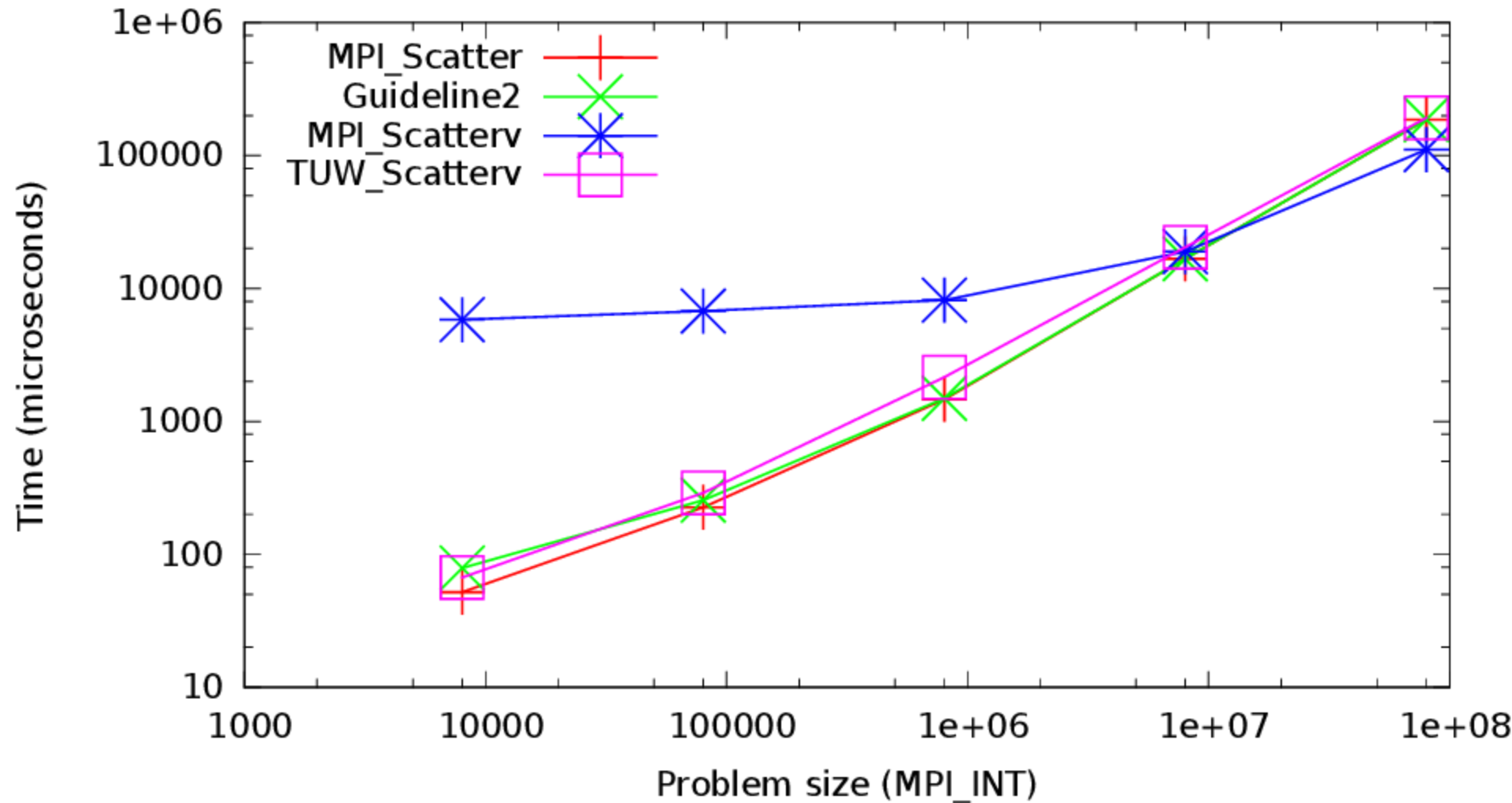




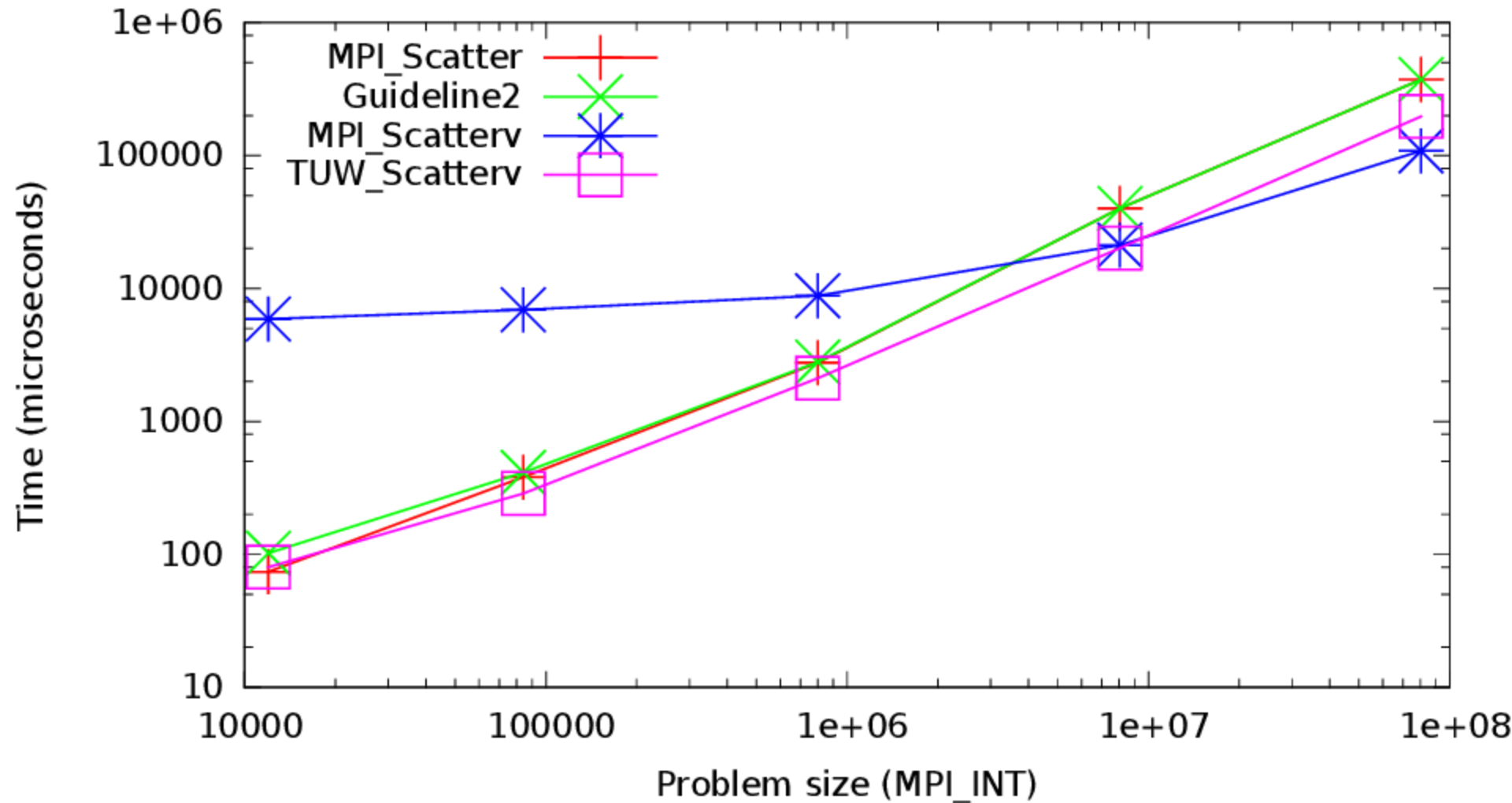
## Intel MPI, Two blocks



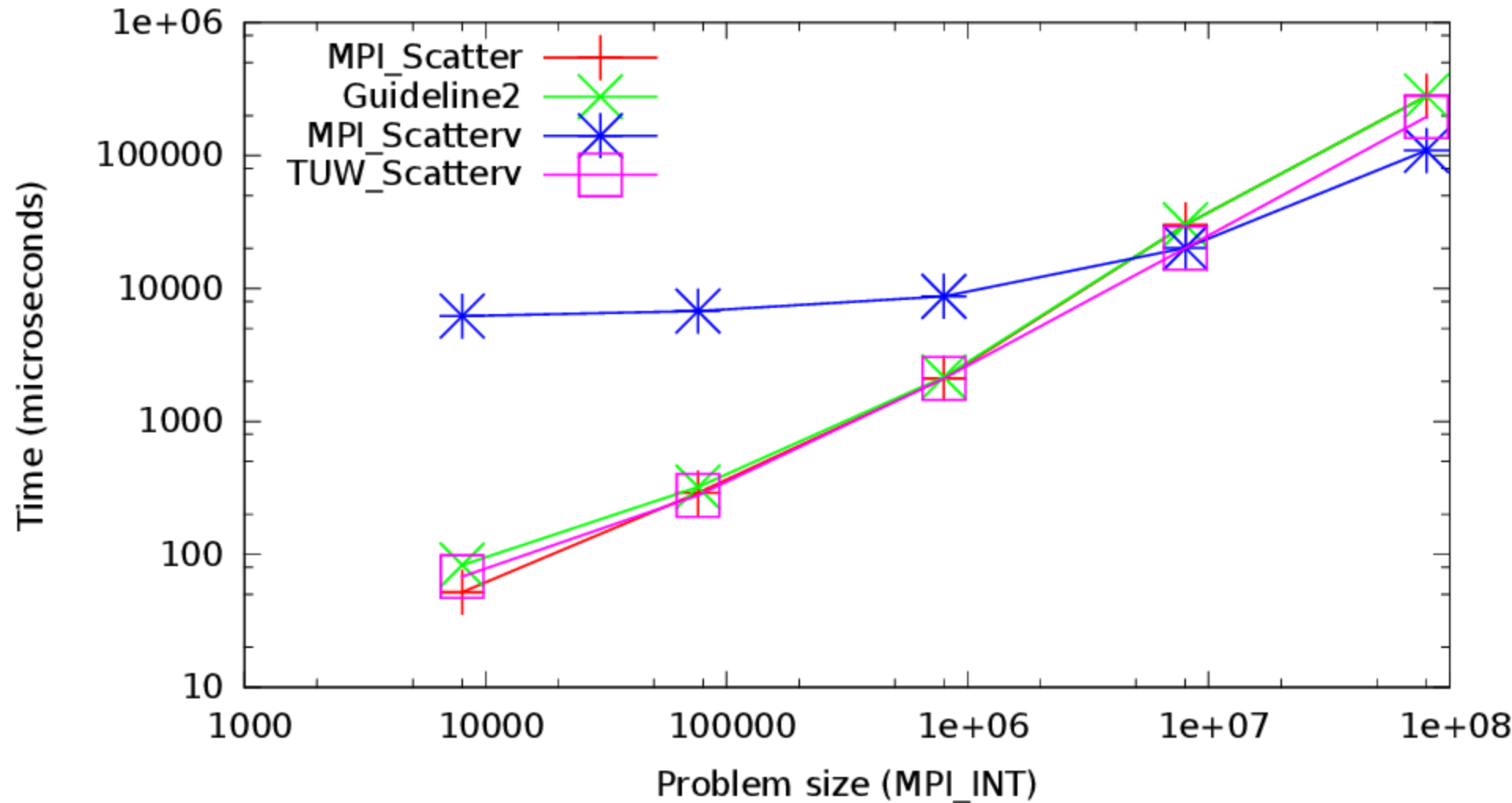
## Intel MPI, Same blocks



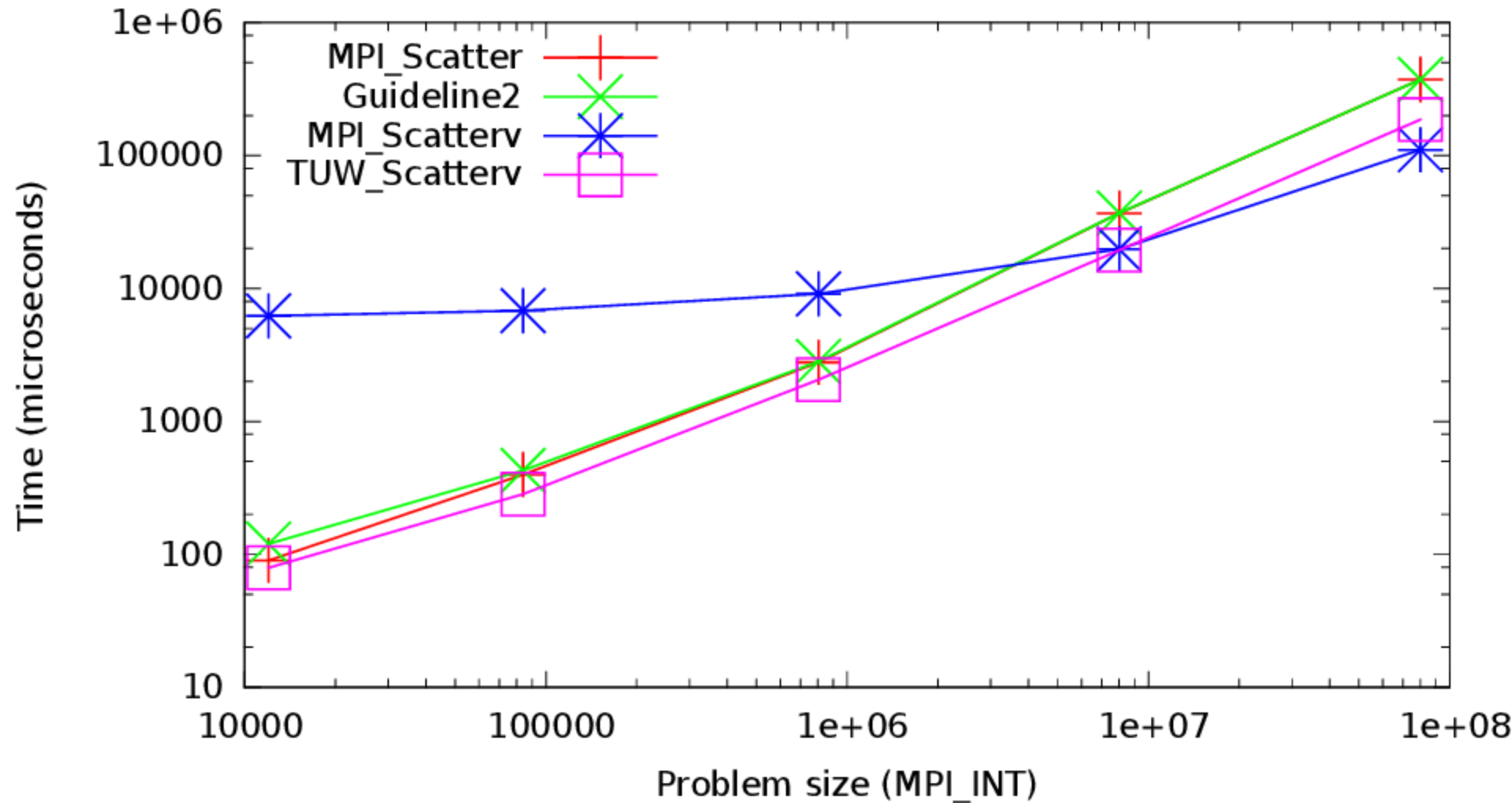
## Intel MPI, Random blocks

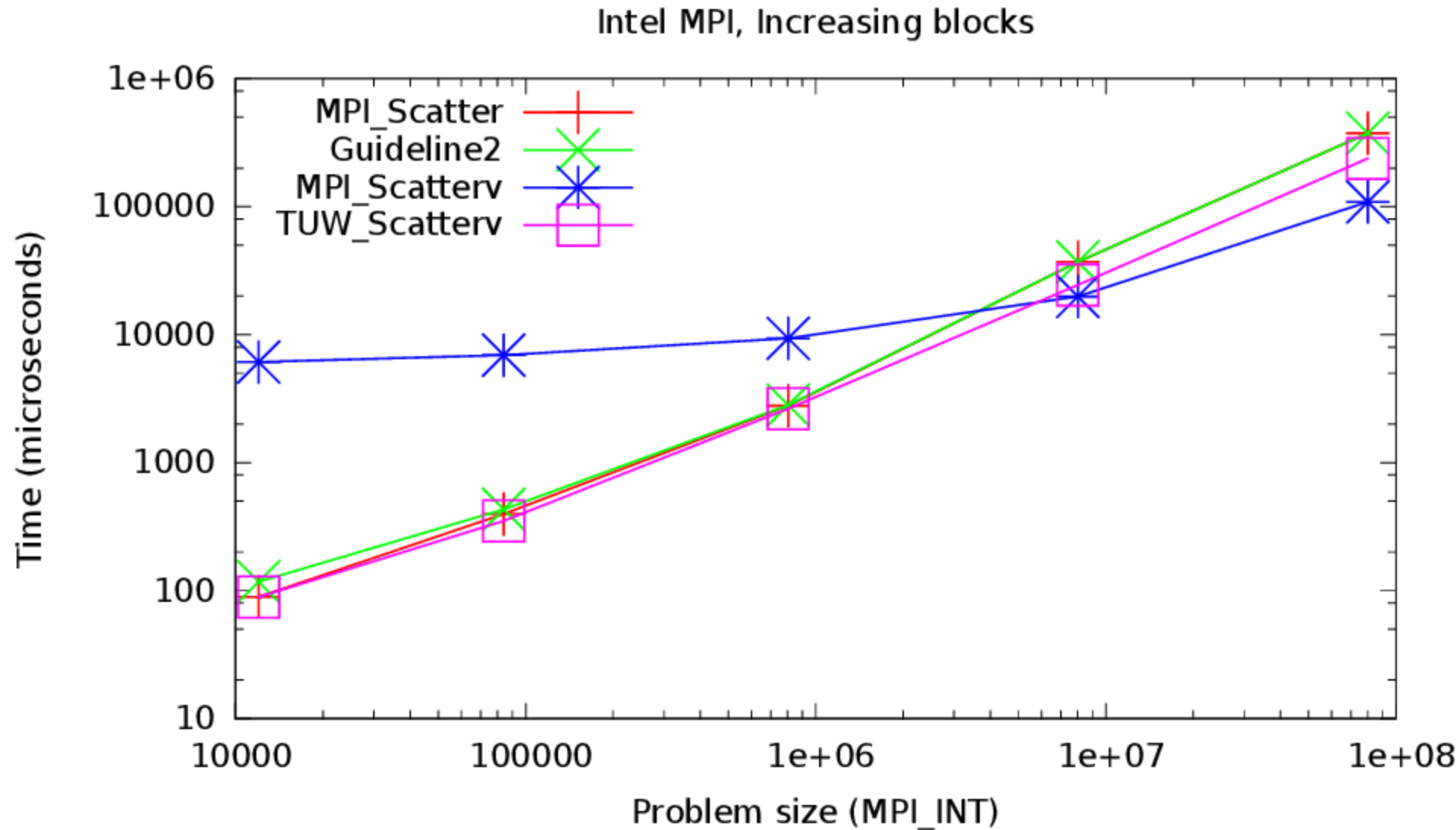


## Intel MPI, Random buckets

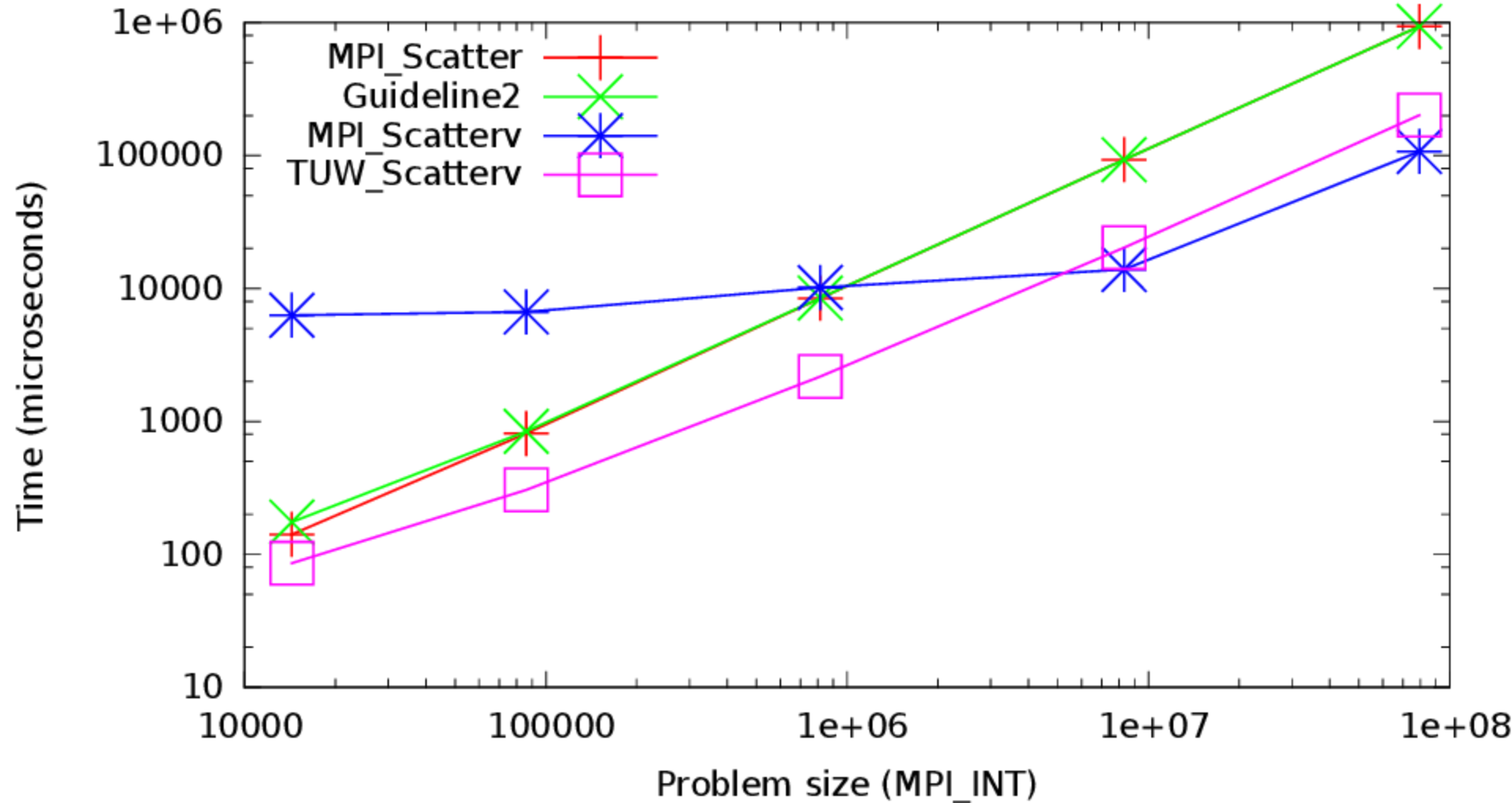


## Intel MPI, Decreasing blocks

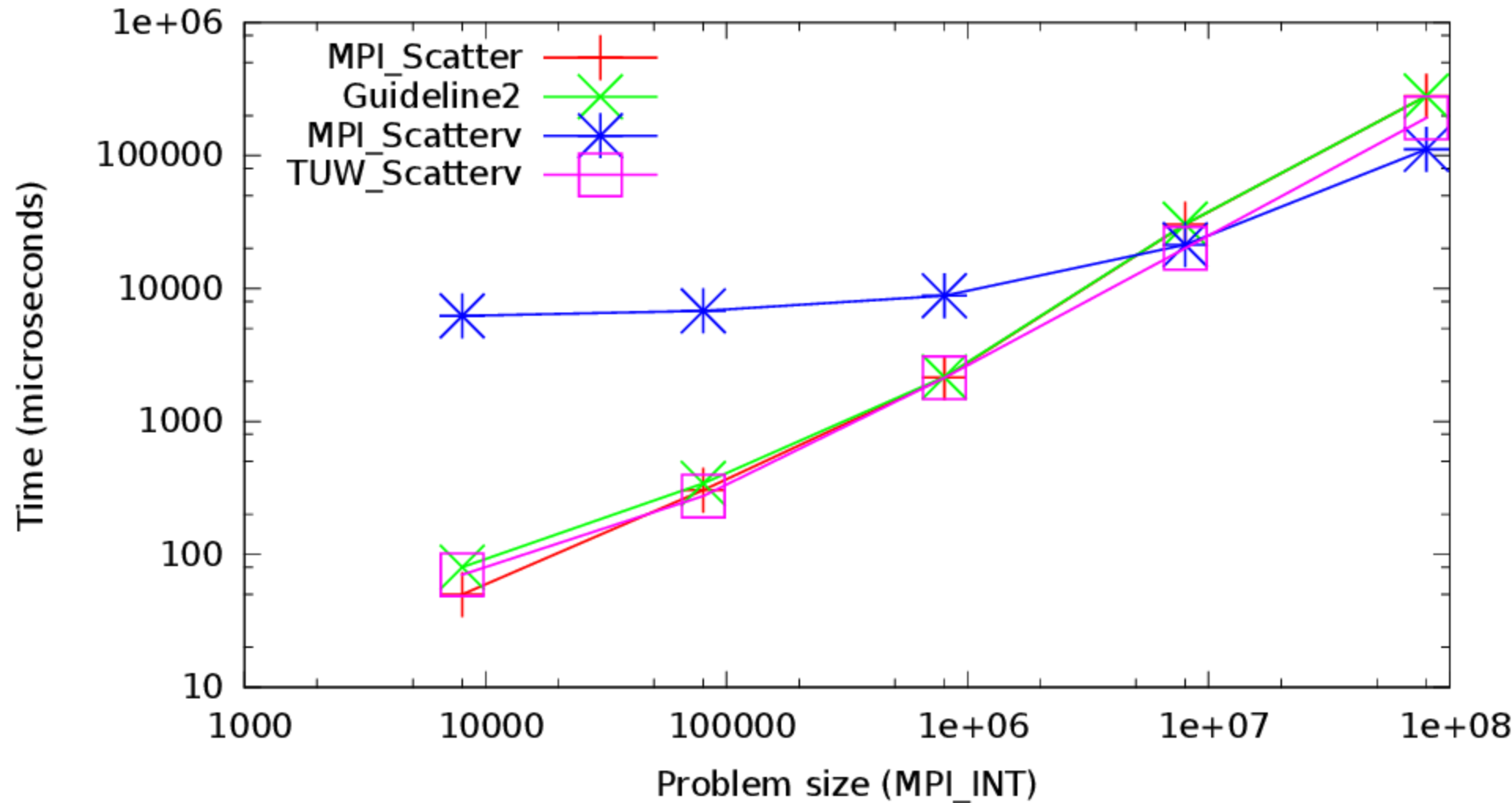




## Intel MPI, Spiked blocks

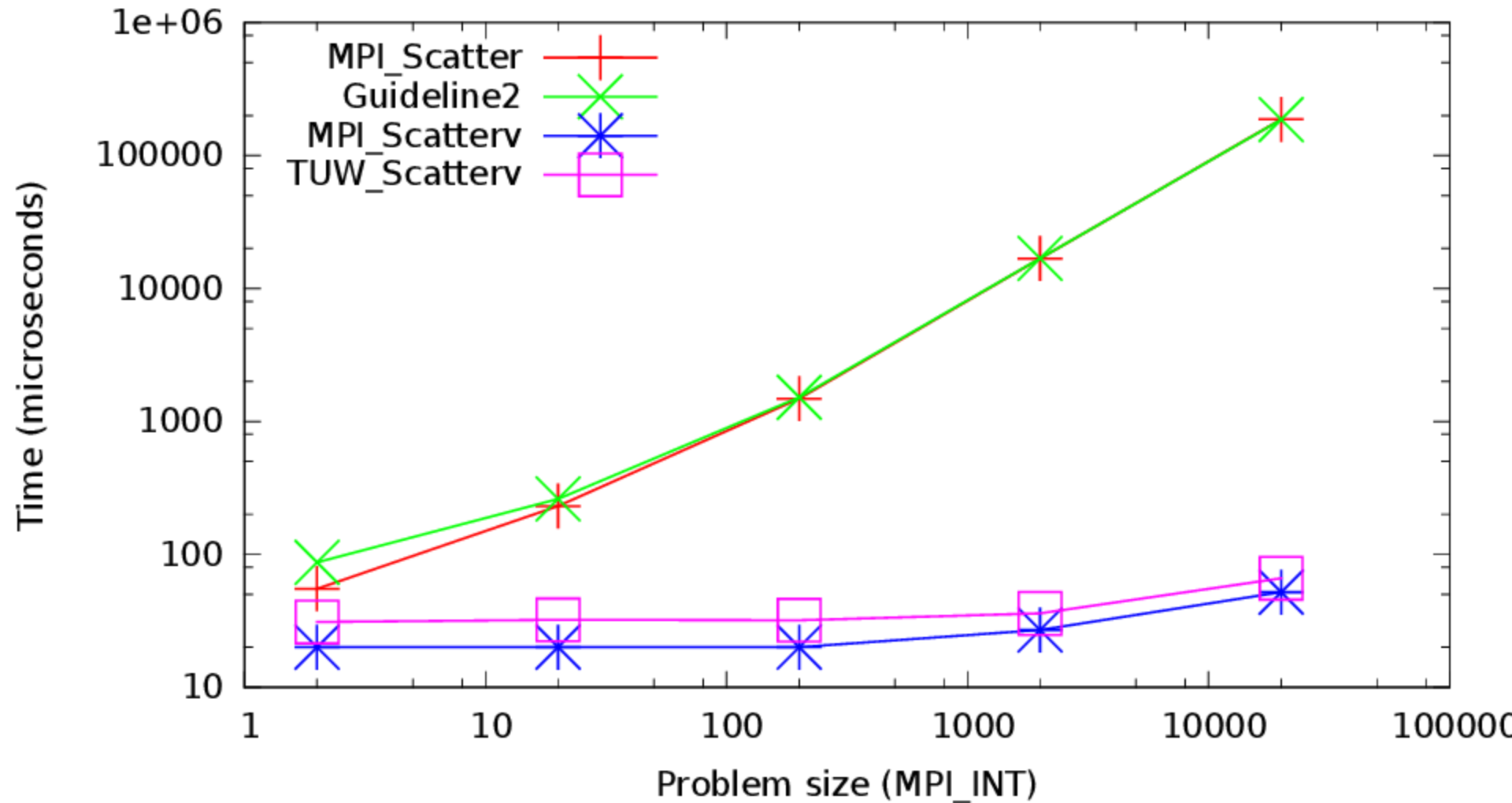


## Intel MPI, Alternating blocks





## Intel MPI, Two blocks



Theorem: The irregular gather problem with block sizes  $m_0, m_1, \dots, m_{p-1}$ , and given root  $r$  can be solved in at most  $3\text{ceil}(\log p)\alpha + \beta \sum_{i \neq r} m_i + \beta \sum_{i \in H', i \neq r'} m_i$  time units

**Question** : What is the minimum possible delay?

**Claim 1**: An **ordered** gather/scatter tree with minimum possible delay can be constructed in polynomial time

**Claim 2**: If the order restriction is dropped, finding a minimum delay tree is NP-hard

Jesper Larsson Träff: Practical, distributed, low overhead algorithms for irregular gather and scatter collectives. *Parallel Comput.* 75: 100-117 (2018)

Claim 1: Indeed, by offline dynamic programming in  $O(n^3)$  operations for homogeneous transmission costs  $t(m) = \alpha + \beta m$ , and  $O(n^4)$  operations for non-homogeneous costs  $t(m) = \alpha_{ij} + \beta_{ij} m$

Claim 2: By reduction from PARTITION

Ad claim 1: Such algorithms are not useful in practice. Is it possible to find better, low complexity, online algorithms than the adaptive binomial tree?

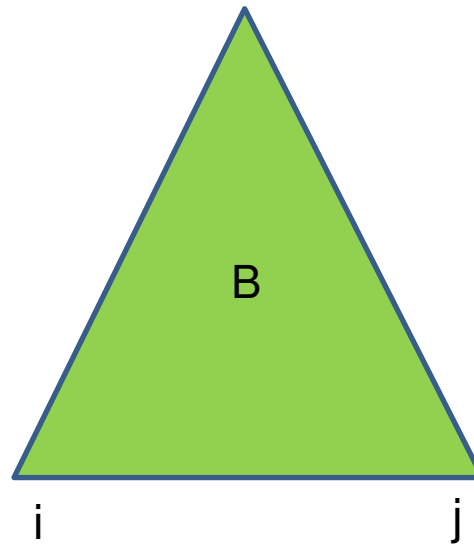
Master's thesis?

Jesper Larsson Träff: On Optimal Trees for Irregular Gather and Scatter Collectives. IEEE Trans. Parallel Distrib. Syst. 30(9): 2060-2074 (2019)

Claim 1, idea:

Ordered gather/scatter tree:

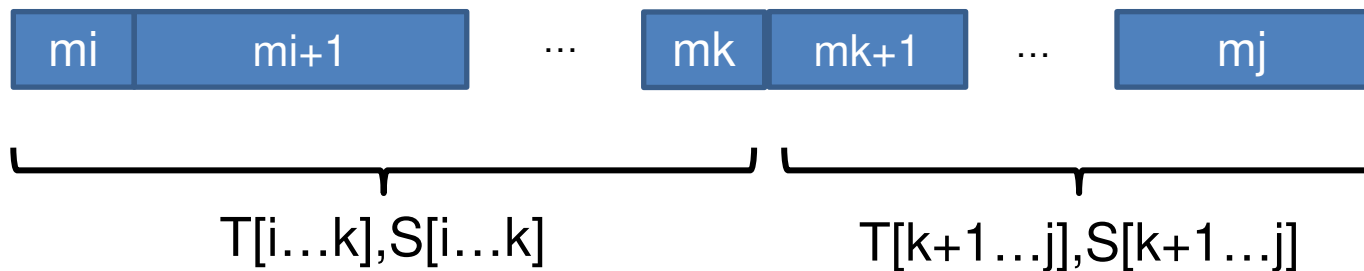
Each subtree is over a consecutive range of processes



Gather: Root in B gathers blocks from processes  $i, i+1, \dots, j$

Claim 1, idea: Let  $T[i \dots j]$  be the best possible gather time for some ordered tree over  $[i \dots j]$ , and  $S[i \dots j] = \sum_{i \leq k \leq j} m_k$

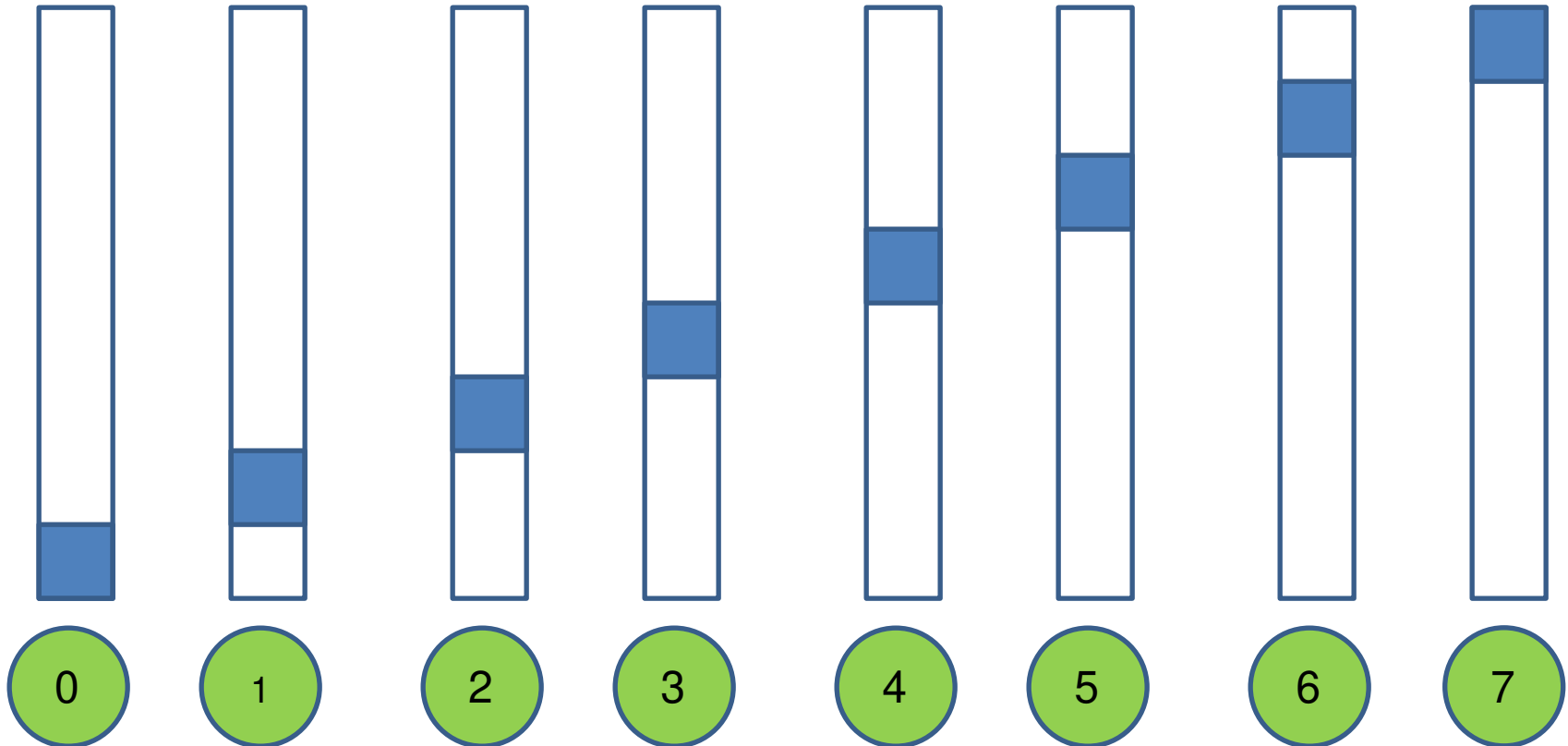
Best ordered tree over  $[i \dots j]$  from best subtrees  $[i \dots k]$ ,  $[k+1 \dots j]$  for some  $k$ ; optimal substructure obviously holds

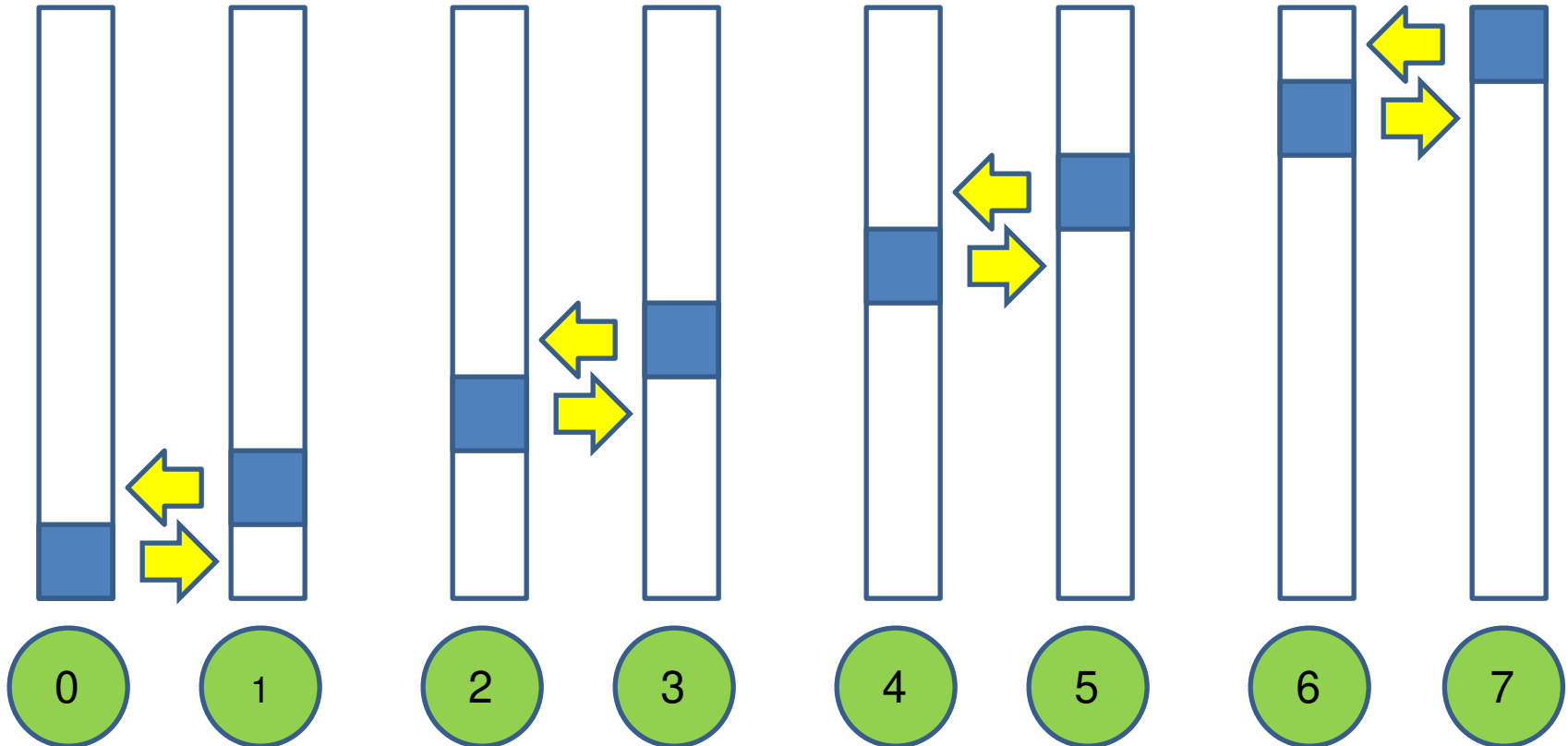


This gives the following dynamic programming equations

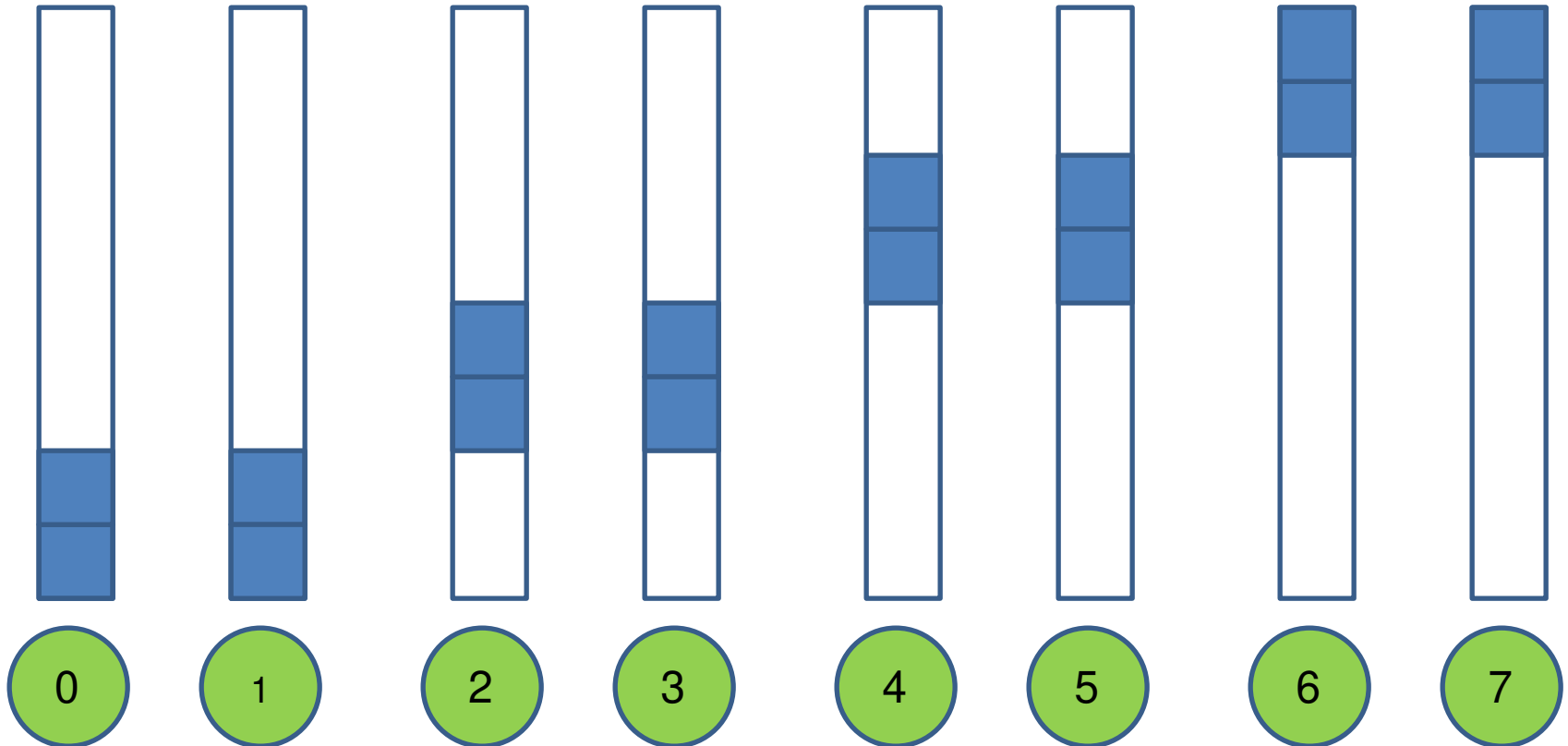
- $T[i] = 0$ ,  $S[i] = m_i$  for all  $i$
- $T[i \dots j] = \min_{i \leq k \leq j-1} \max(T[i \dots k], T[k+1 \dots j]) + \alpha + \beta \min(S[i \dots k], S[k+1 \dots j])$

## The power of the hypercube/butterfly: Allgather

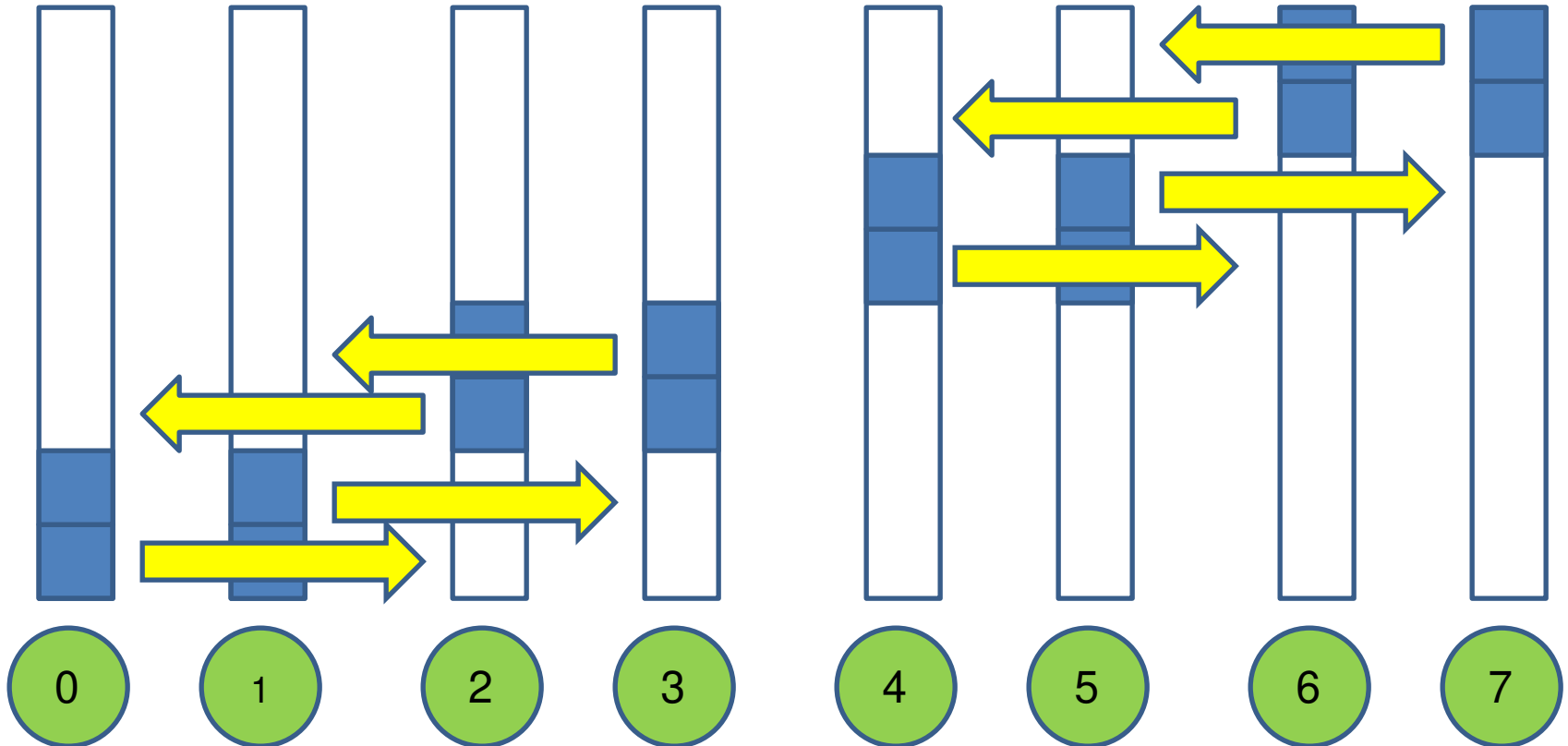




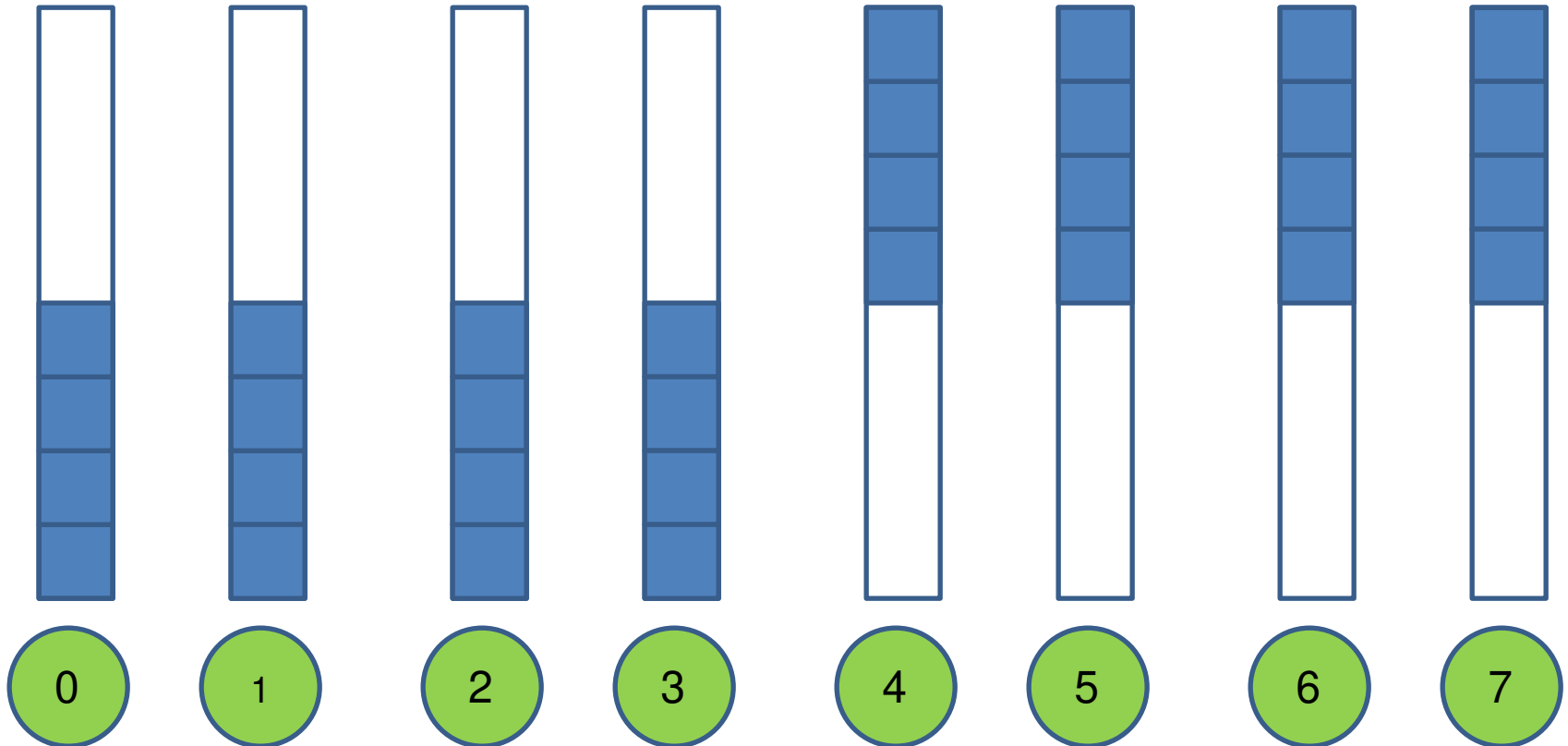
Round  $k$  partner: rank XOR  $2^k$  (flip  $k$ 'th bit), exchange distance  $2^k$

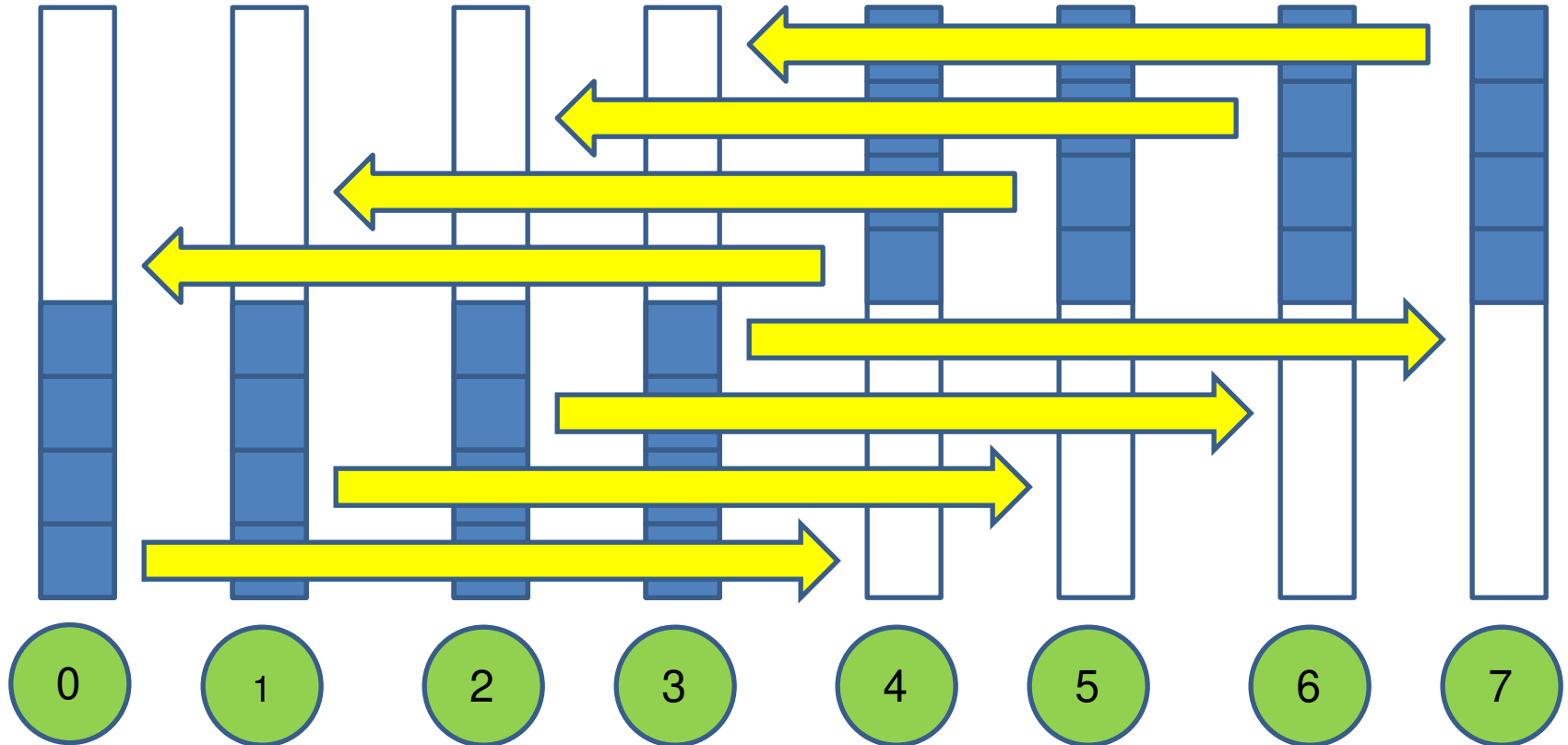




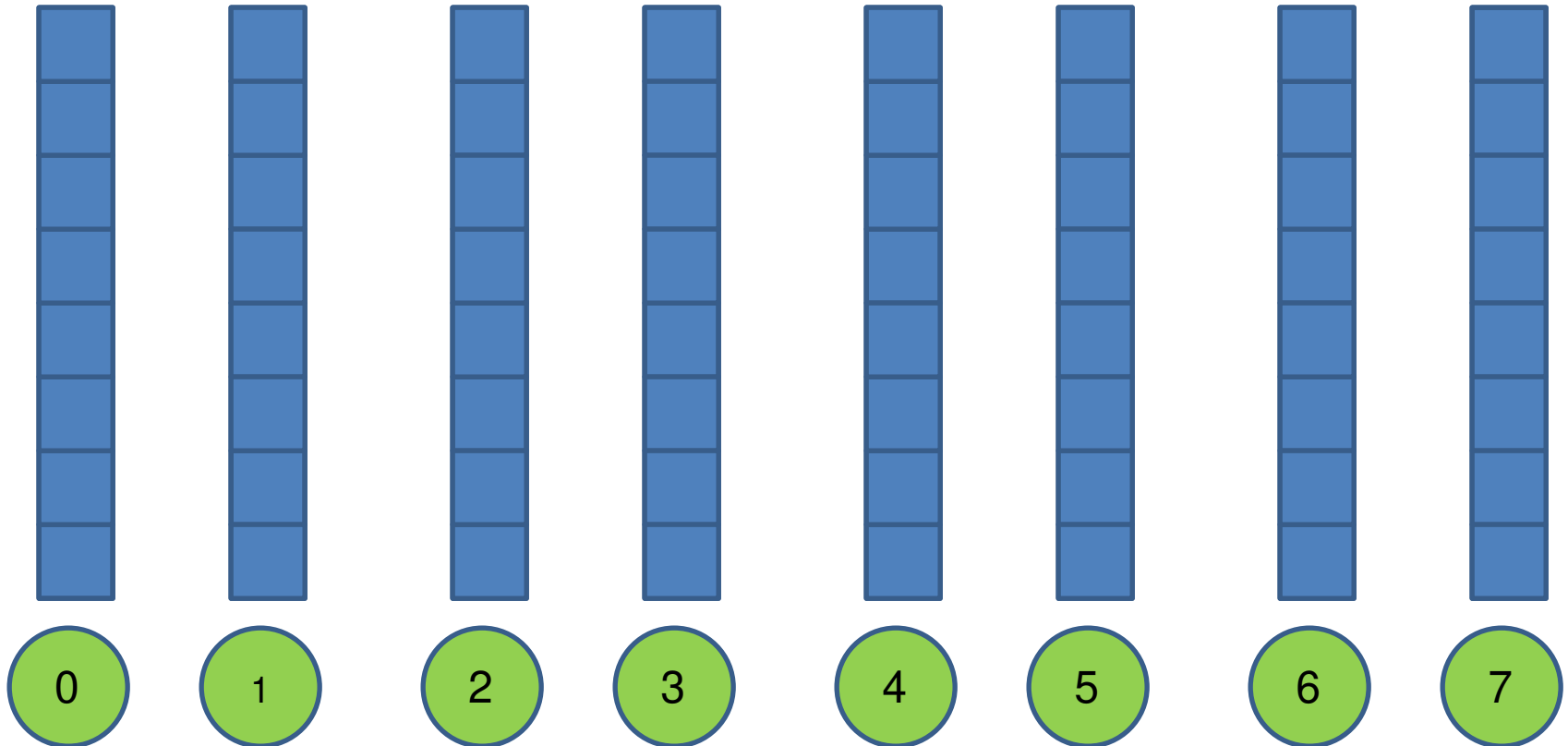


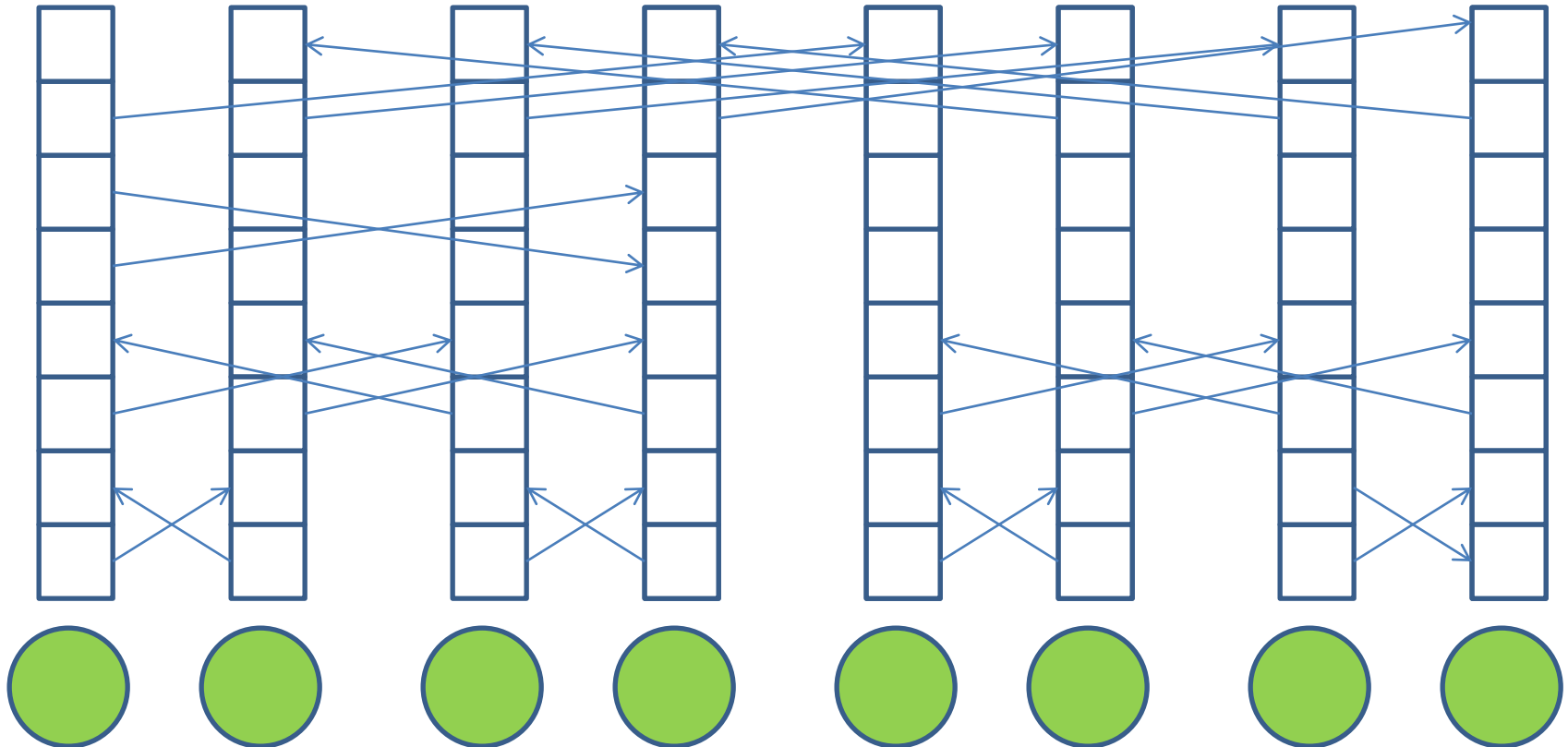
Round  $k$  partner: rank  $\text{rank} \oplus 2^k$  (flip  $k$ 'th bit), exchange distance  $2^k$





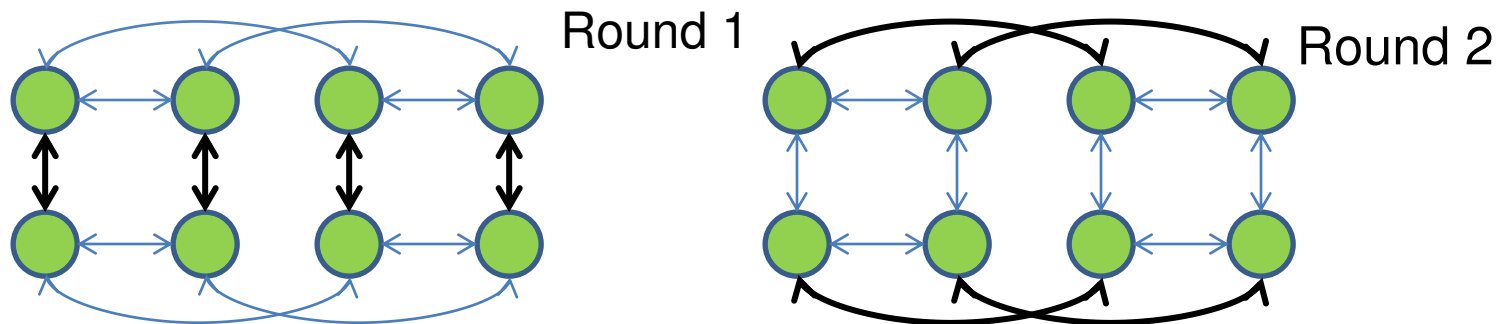
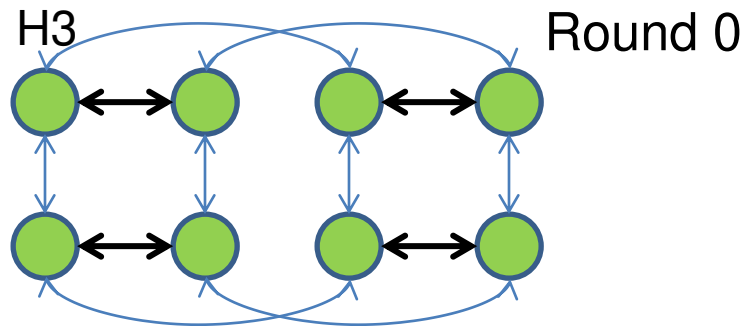
Round  $k$  partner: rank  $\text{XOR } 2^k$  (flip  $k$ 'th bit), exchange distance  $2^k$





Butterfly/FFT communication pattern: Telephone bidirectional

Hypercube embedding of butterfly:



## The power of the hypercube/butterfly: Allgather

$$\text{Tallgather}(m) = (\log p)\alpha + (p-1)/p \beta m$$

Optimal in both terms

since the amount of data exchanged in round  $k$  is  
 $m/2^{\log p - k}$ ,  $k=0, 1, \dots, \log p - 1$

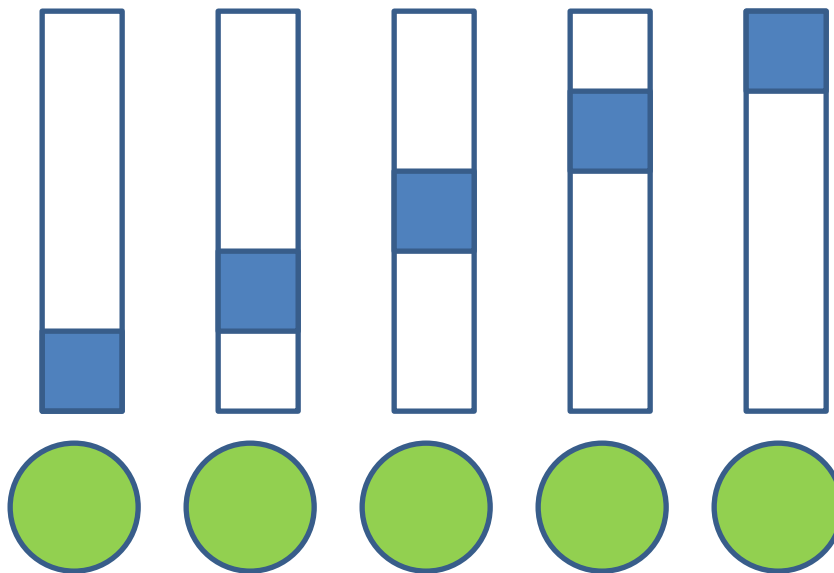
### Drawback:

Butterfly (hypercube) algorithms **do not extend nicely** to the case where  $p$  is not a power of 2

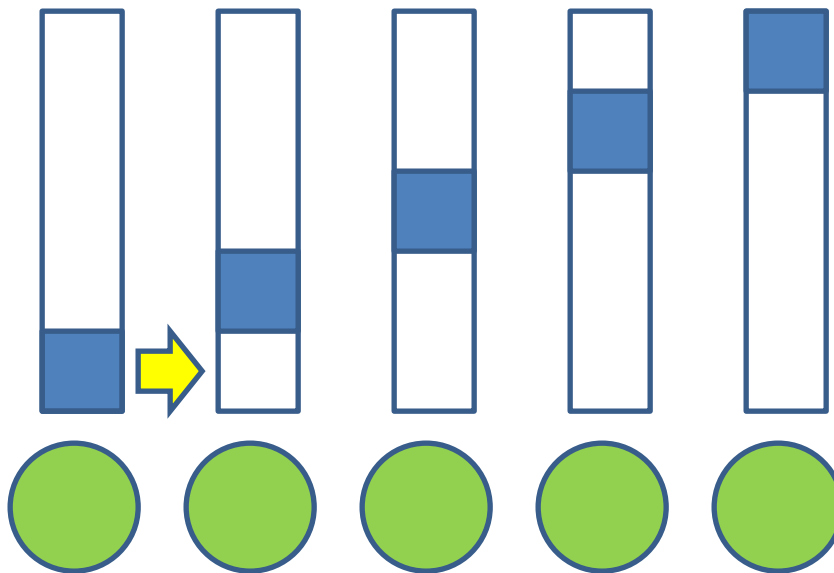
R. Rabenseifner, J. L. Träff: More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems. PVM/MPI 2004: 36-46  
 J. L. Träff: An Improved Algorithm for (Non-commutative) Reduce-Scatter with an Application. PVM/MPI 2005: 129-137

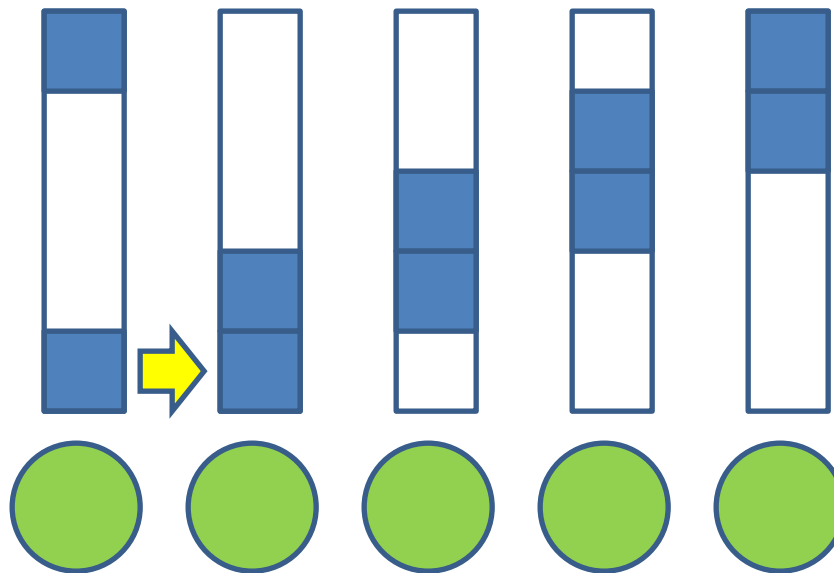
## Fully-connected network: Allgather

Exploit fully bidirectional communication to accumulate and disseminate blocks. Another viewpoint: Multiple, edge disjoint binomial trees.

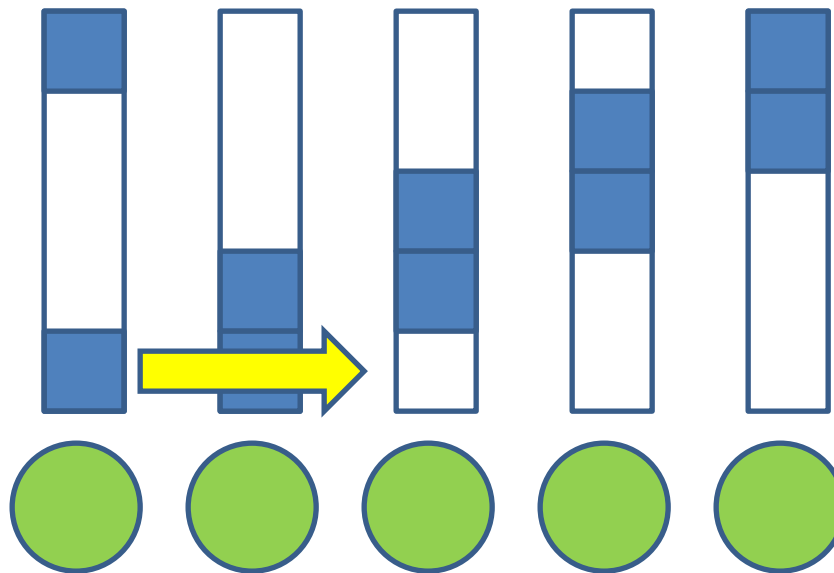




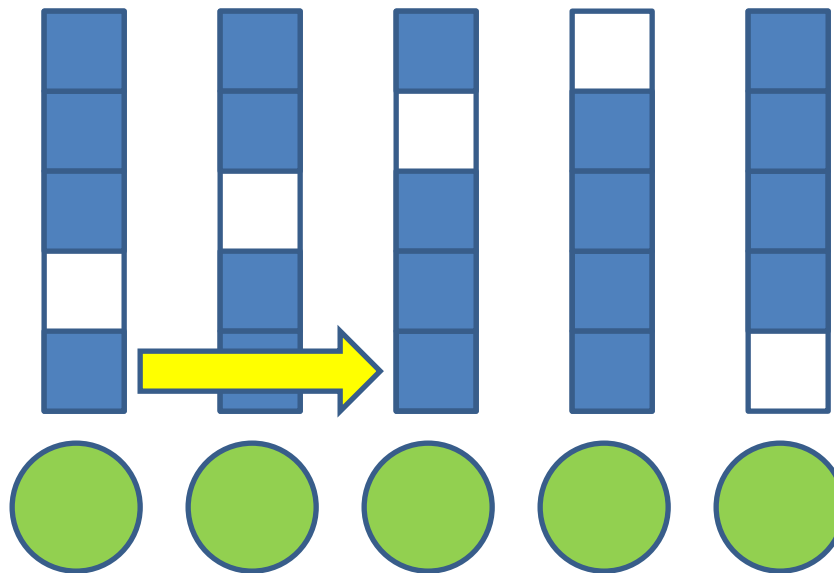




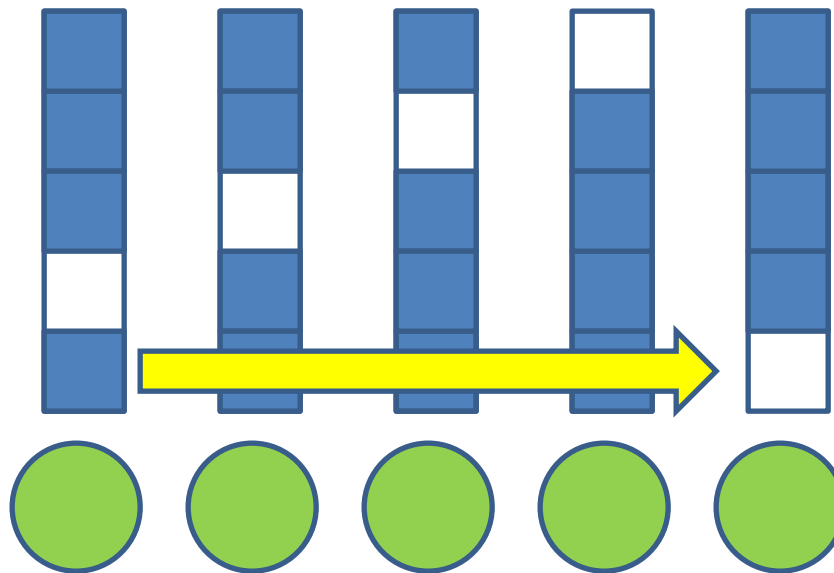
Round  $k$ :  
 Each process  
 sends/receives  
 block of size  $2^k$   
 to/from processor  
 $(i+2^k, i-2^k) \bmod p$



Round  $k$ :  
 Each process  
 sends/receives  
 block of size  $2^k$   
 to/from processor  
 $(i+2^k, i-2^k) \bmod p$

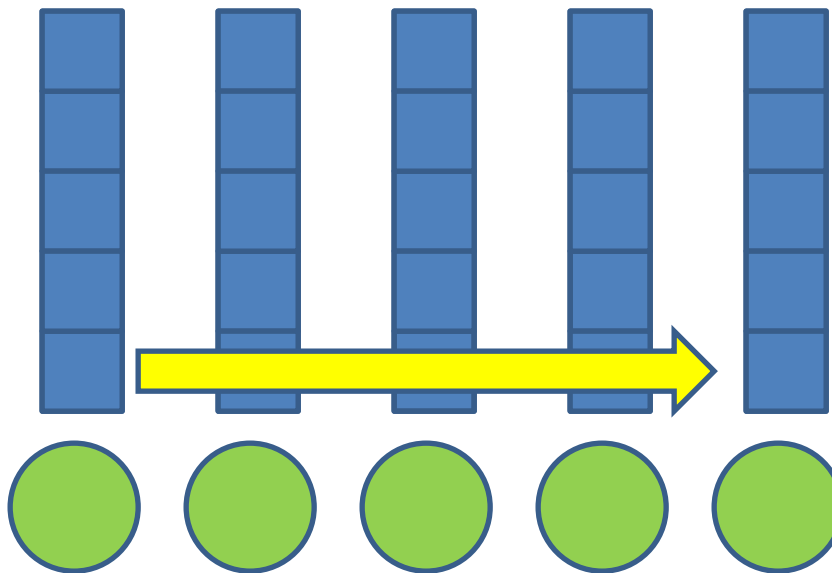


Round  $k$ :  
 Each process  
 sends/receives  
 block of size  $2^k$   
 to/from processor  
 $(i+2^k, i-2^k) \bmod p$



Round  $k$ :  
 Each process  
 sends/receives  
 block of size  $2^k$   
 to/from processor  
 $(i+2^k, i-2^k) \bmod p$

Except for last  
 round: block of size  
 $p-2^k$



Round  $k$ :  
 Each process  
 sends/receives  
 block of size  $2^k$   
 to/from processor  
 $(i+2^k, i-2^k) \bmod p$

**Except for last  
 round:** block of size  
 $p-2^k$

## “Bruck”/“Dissemination” Allgather algorithm (sketch):

```

d = 1
floor(log p) rounds:
    Receive block[rank-2*d+1:rank-d] from rank-d
    Send block[rank-d+1:rank] to rank+d
    d = d*2
if d < ceil(log p)
    Receive block[rank-d-r+1:rank-d] from rank-d
    Send block[rank-r+1:rank] to rank+d
where r = p-d
/* all rank+/-x operations are mod p */

```

Tallgather(m) =  $\text{ceil}(\log p)\alpha + (p-1)/p m\beta$

Optimal in both terms

No algorithmic latency

Fundamental paper to study



“Bruck”/“Dissemination” Allgather algorithm (sketch):

```

d = 1
floor(log p) rounds:
    Receive block[rank-2*d+1:rank-d] from rank-d
    Send block[rank-d+1:rank] to rank+d
    d = d*2
if d < ceil(log p)
    Receive block[rank-d-r+1:rank-d] from rank-d
    Send block[rank-r+1:rank] to rank+d
where r = p-d
/* all rank+/-x operations are mod p */
  
```

Tallgather(m) =  $\text{ceil}(\log p)\alpha + (p-1)/p m\beta$

J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, D. Weathersby:  
Efficient Algorithms for All-to-All Communications in Multiport  
Message-Passing Systems. IEEE Trans. Parallel Distrib. Syst.  
8(11): 1143-1156 (1997)



## “Bruck”/“Dissemination” Allgather algorithm (sketch):

```

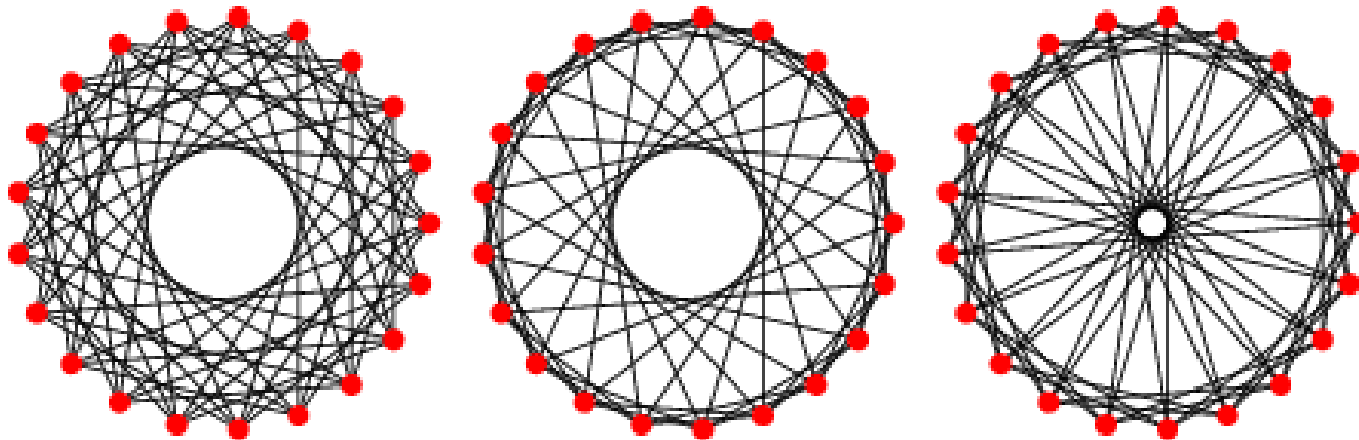
d = 1
floor(log p) rounds:
    Receive block[rank-2*d+1:rank-d] from rank-d
    Send block[rank-d+1:rank] to rank+d
    d = d*2
if d < ceil(log p)
    Receive block[rank-d-r+1:rank-d] from rank-d
    Send block[rank-r+1:rank] to rank+d
where r = p-d
/* all rank+/-x operations are mod p */

```

**For MPI** : Blocks are received in order, e.g. [4,5,6,0,1,2,3] for rank 3, so internal buffering and copying may be necessary in some steps(\*); MPI standard prescribes [0,1,2,3,4,5,6] order for all ranks

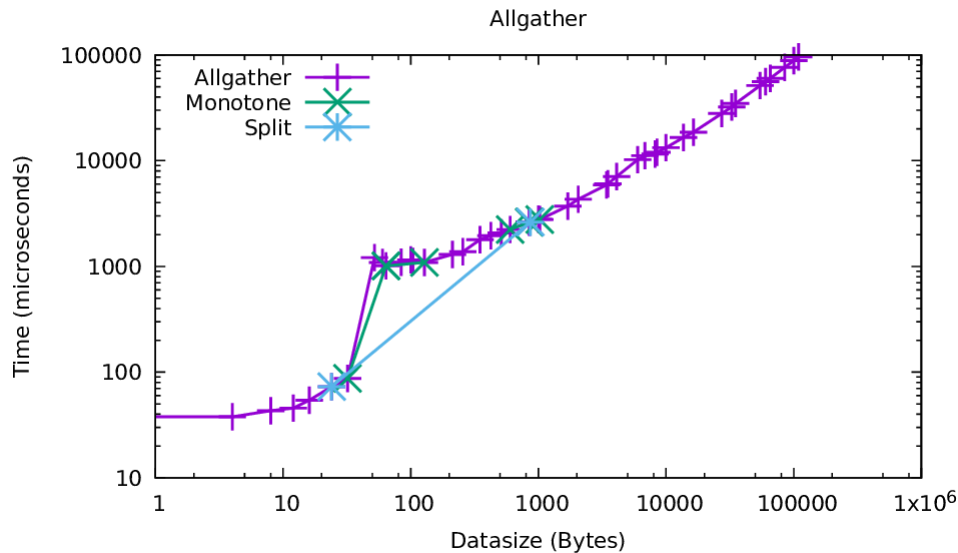
(\*) or use MPI derived datatypes

Communication pattern in Dissemination allgather is a **circulant graph**: processor  $i$  communicates with processors  $(i+d) \bmod p$  and  $(i-d) \bmod p$ , for  $d=1, 2, 4, 2^{\lceil \log p \rceil - 1}$

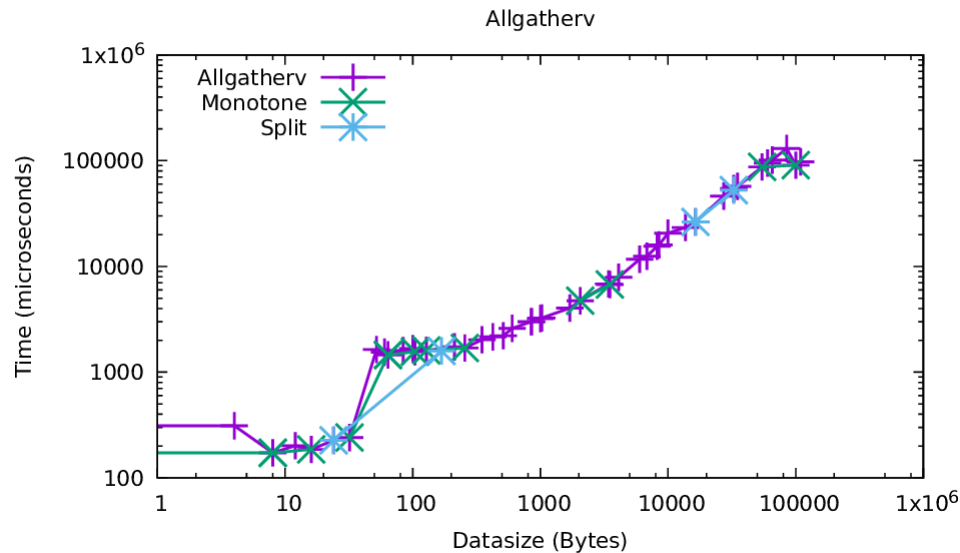


From Wolfram MathWorld,  
[www.mathworld.wolfram.com/CirculantGraph.html](http://www.mathworld.wolfram.com/CirculantGraph.html)

MPI\_Allgather vs. MPI\_Allgatherv (OpenMPI 3.1.3). Is log p round algorithm used?

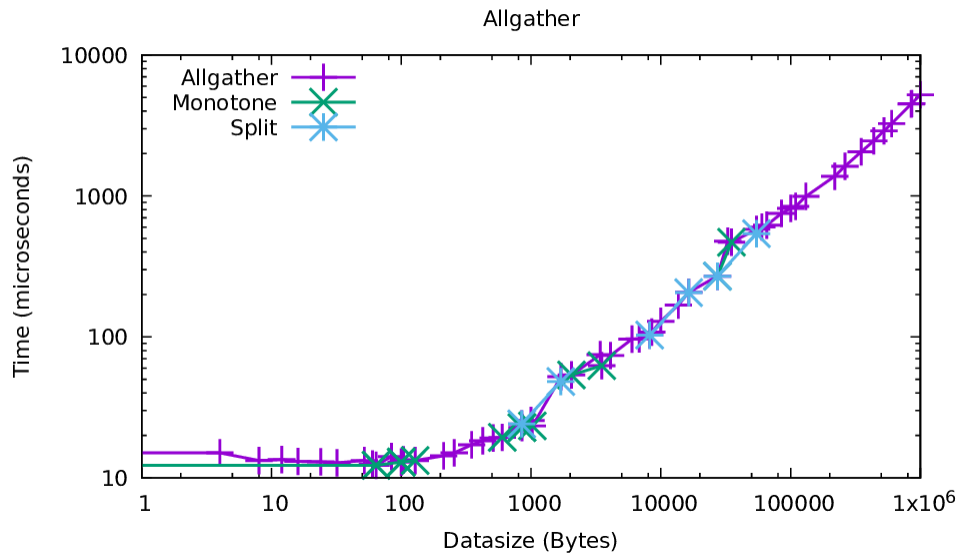


MPI\_COMM\_WORLD,  
36x32 processes

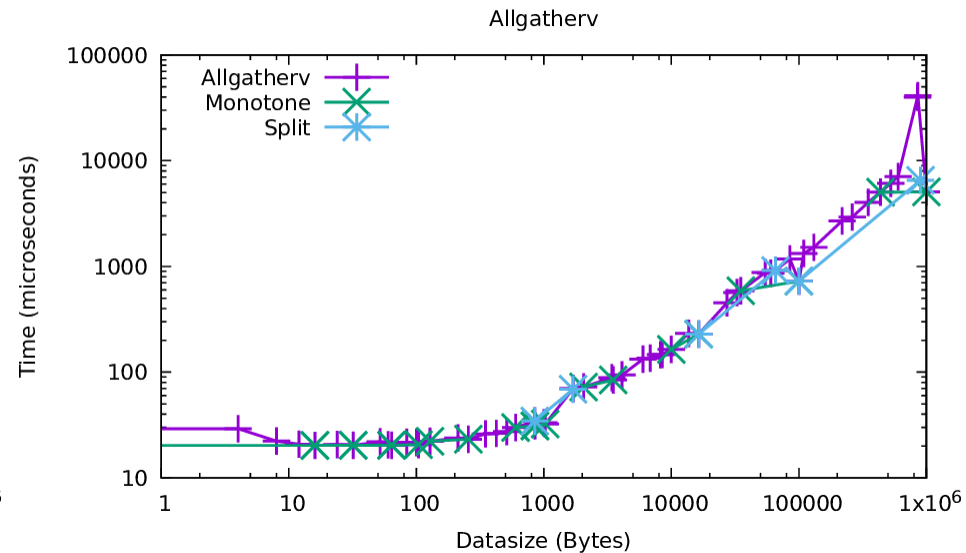


MPI\_COMM\_WORLD,  
36x32 processes

# MPI\_Allgather vs. MPI\_Allgatherv (OpenMPI 3.1.3). Is log p round algorithm used?

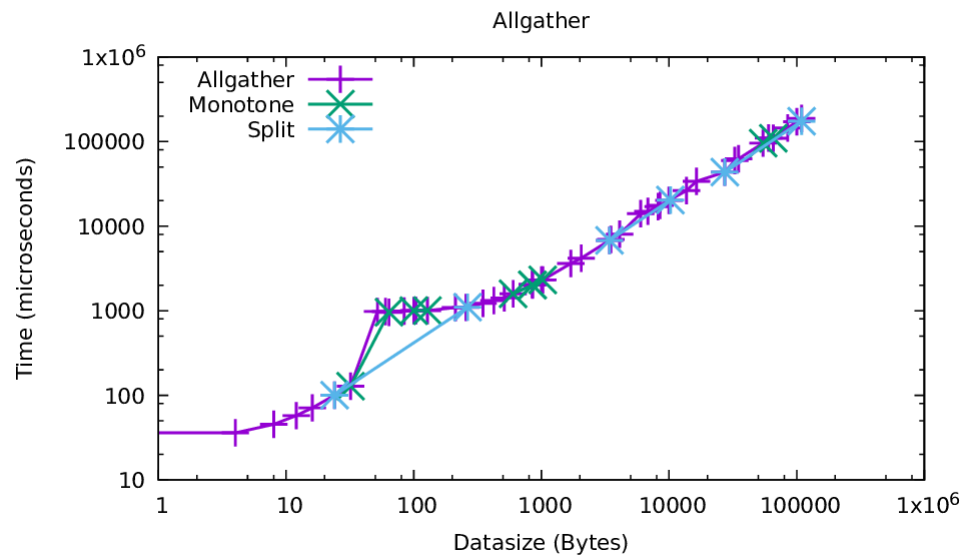


MPI\_COMM\_WORLD, 36x1  
processes

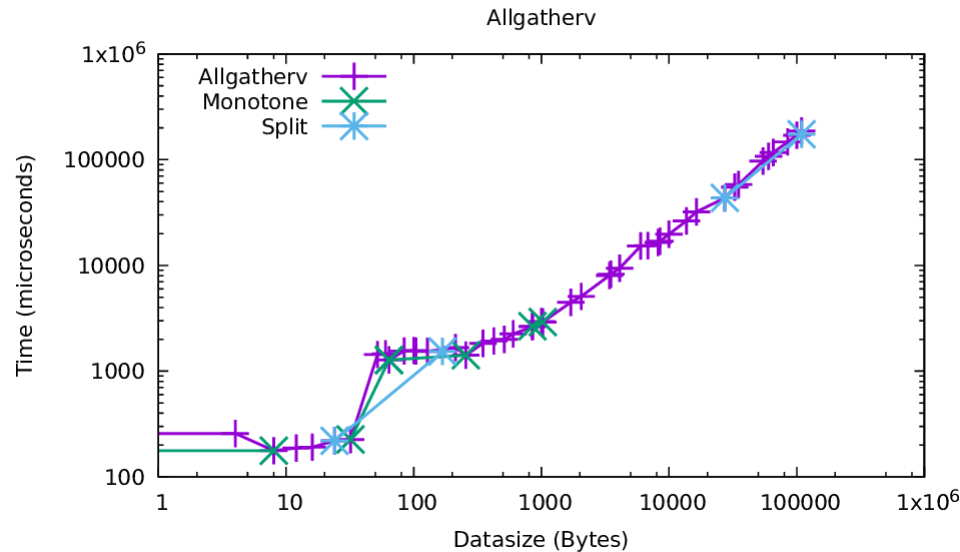


MPI\_COMM\_WORLD, 36x1  
processes

# MPI\_Allgather vs. MPI\_Allgatherv (OpenMPI 3.1.3). Is log p round algorithm used?



cyclic comm, 36x32  
processes



cyclic comm, 36x32  
processes

## Allgather pattern application: Non-MPI collective operation

Problem:

Each process has a block of  $n$  elements in order ( $\leq$ ). Collect at each process a large block consisting of all elements in order ( $\leq$ )

Solution: Use the circulant graph communication pattern, in each round merge received block with block at process. Complexity is  $\text{ceil}(\log p)$  communication rounds,  $O(pn)$  operations (send-receive and merge)

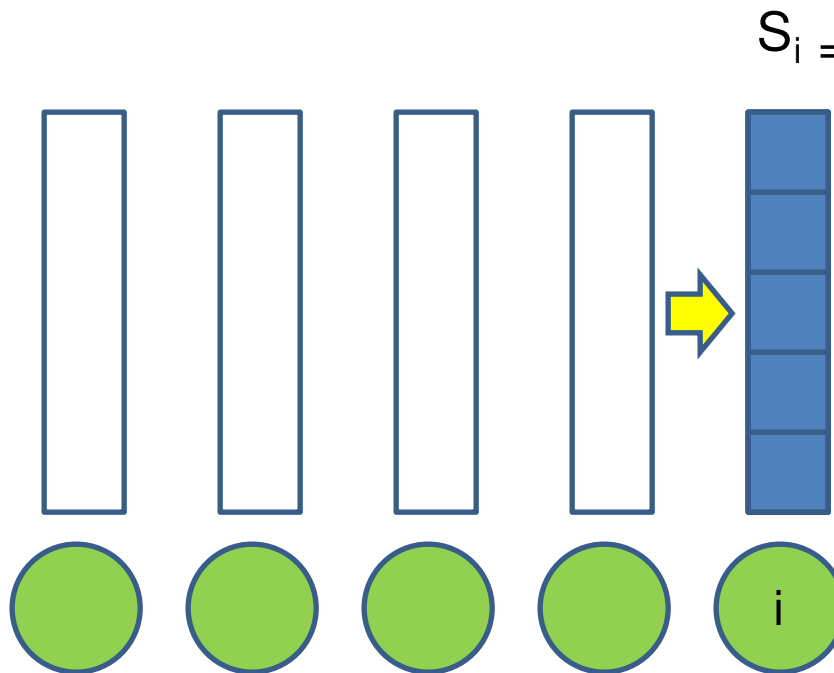
Can also be used for duplicate elimination, key-value pairs, etc. (e.g., maintaining tentative distances in a distributed implementation of Dijkstra's algorithm)

Is this correct?

Be very careful not to loose elements, see next algorithm

## Beyond the butterfly: Allreduce in fully-connected networks

Will the allgather circulant graph scheme work for (commutative) reduction operations (Allreduce)?



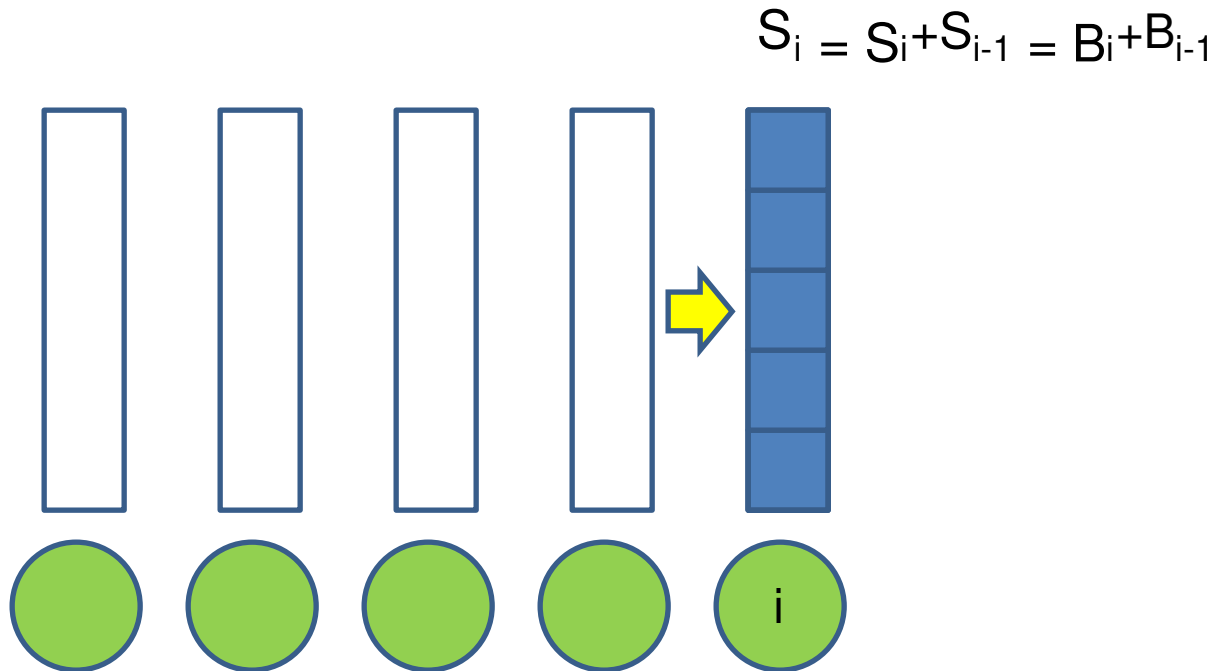
Idea:

Maintain partial sum  $S_i = \sum_{k=j}^i B_k$  where  $B_k$  is the input vector of process  $k$

Circulant graph: All processor indices are mod  $p$

Invariant:

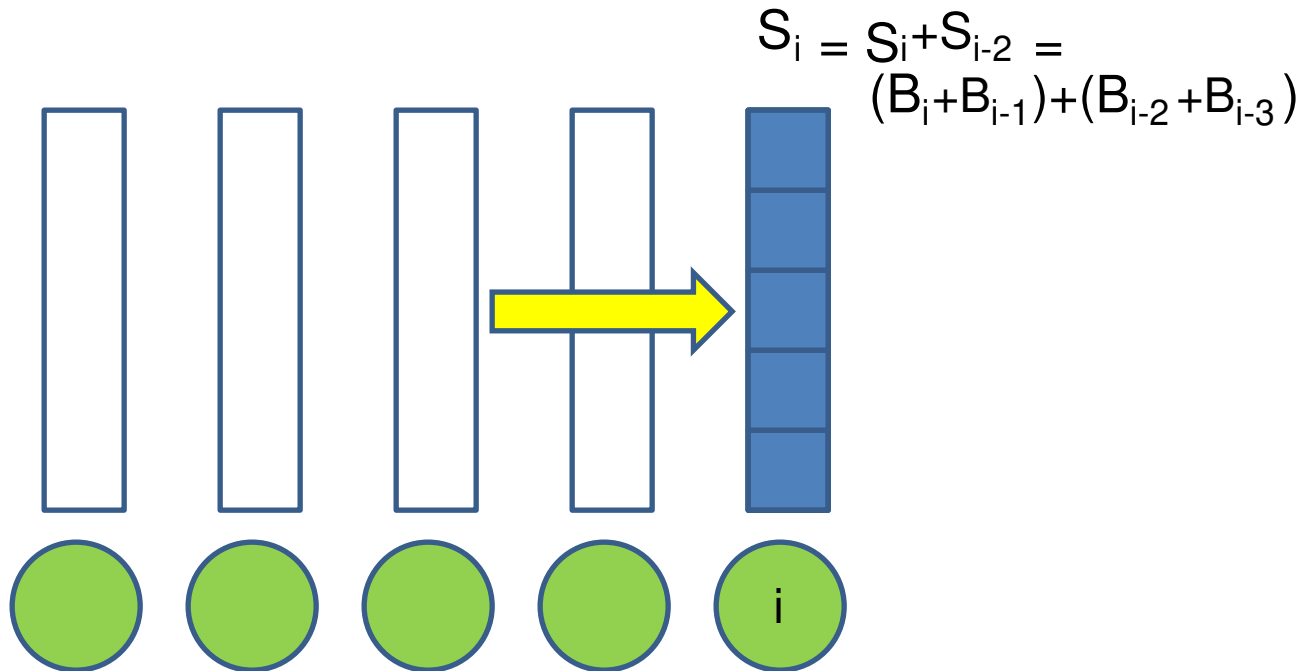
Partial sum  $S_i = \sum_{k=j}^i B_k$  where  $B_k$  is the input vector of process  $k$  and  $j = i - 2^r$  after round  $r = 0, 1, \dots$





Invariant:

Partial sum  $S_i = \sum_{k=j}^i B_k$  where  $B_k$  is the input vector of process  $k$  and  $j = i - 2^r$  after round  $r = 0, 1, \dots$



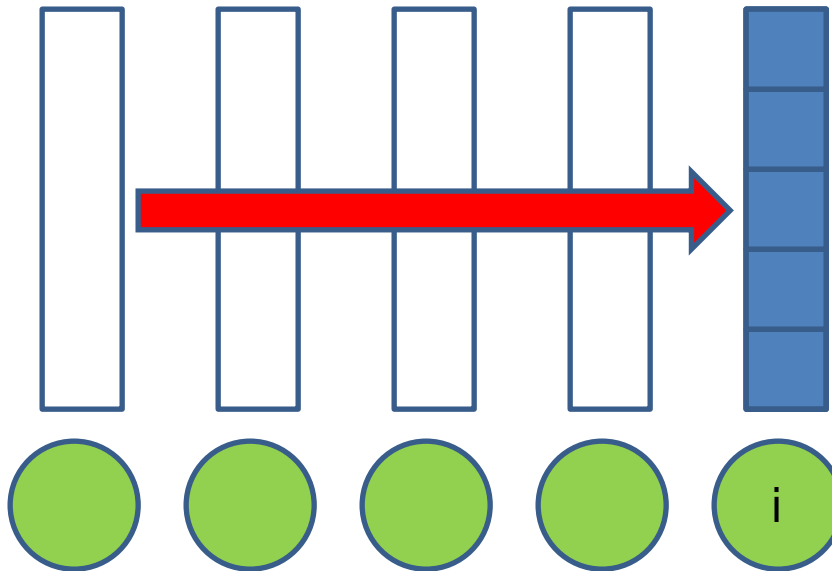
Invariant:

Partial sum  $S_i = \sum_{k=j}^i B_k$  where  $B_k$  is the input vector of process  $k$  and  $j = i - 2^r$  after round  $r = 0, 1, \dots$

Circulant graph scheme works only when  $p$  is a power of 2

Or idempotent operators, e.g., MAX

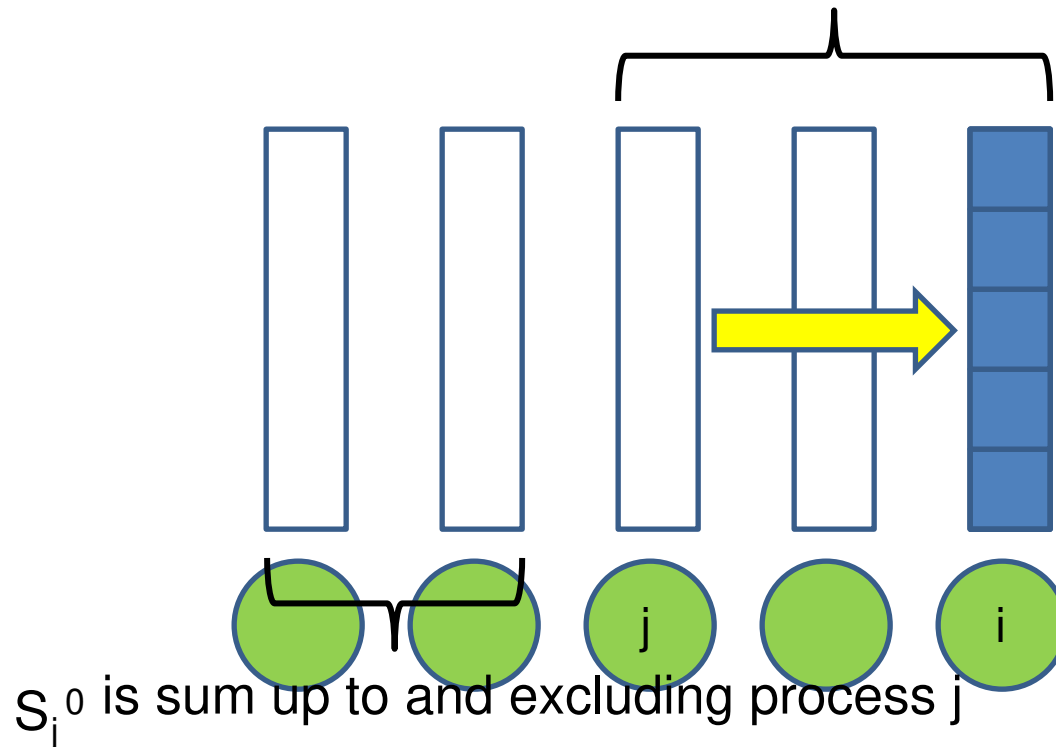
$$S_i = S_i + S_{i-4} = (B_i + B_{i-1} + B_{i-2} + B_{i-3}) + S_{i-4}$$



Different invariant, different communication pattern: Each process  $i$  maintains two sums,  $S_i^1 = \sum_{k=j}^i B_k$  and  $S_i^0 = \sum_{k=j}^{i-1} B_k$  for  $j \leq i$

May work?

$S_i^1$  is sum up to and including process  $i$



When process  $j$  receives from process  $j$ , it must decide whether  $B_j$  has already been added to  $S_i^1$  and  $S_i^0$

Which pattern, which  $j$ ?

Key insight



Write  $p-1$  in binary as  $(b_0 b_1 \dots b_j \dots b_{k-1})_2$  with bits  $b_j$  (0 or 1),  $b_0$  being the most significant bit of  $p-1$ . **Note** that  $b_0=1$ , always

Define  $n_{-1} = 0$ , and  $n_j = 2n_{j-1} + b_j$  for  $j=0, \dots, k-1$ .

It follows that  $n_j = \sum_{i=0}^j b_i 2^{j-i}$  (by induction,  $n_{j+1} = 2(\sum_{i=0}^j b_i 2^{j-i}) + b_{j+1} = \sum_{i=0}^{j+1} b_i 2^{j+1-i}$ ). By the observation that  $b_0=1$ , thus  $n_0=1$ . Also, for  $j=k-1$ ,  $n_j=p-1$

Each process  $i$  maintains the following invariants on  $S1$  and  $S0$ :

- $S1 = \sum_{l=i-n_j}^i B_l$ , the sum of  $n_j+1$  previous elements **including** the process itself,
- $S0 = \sum_{l=i-n_j}^{i-1} B_l$ , the sum of  $n_j$  previous elements **excluding** the process itself,

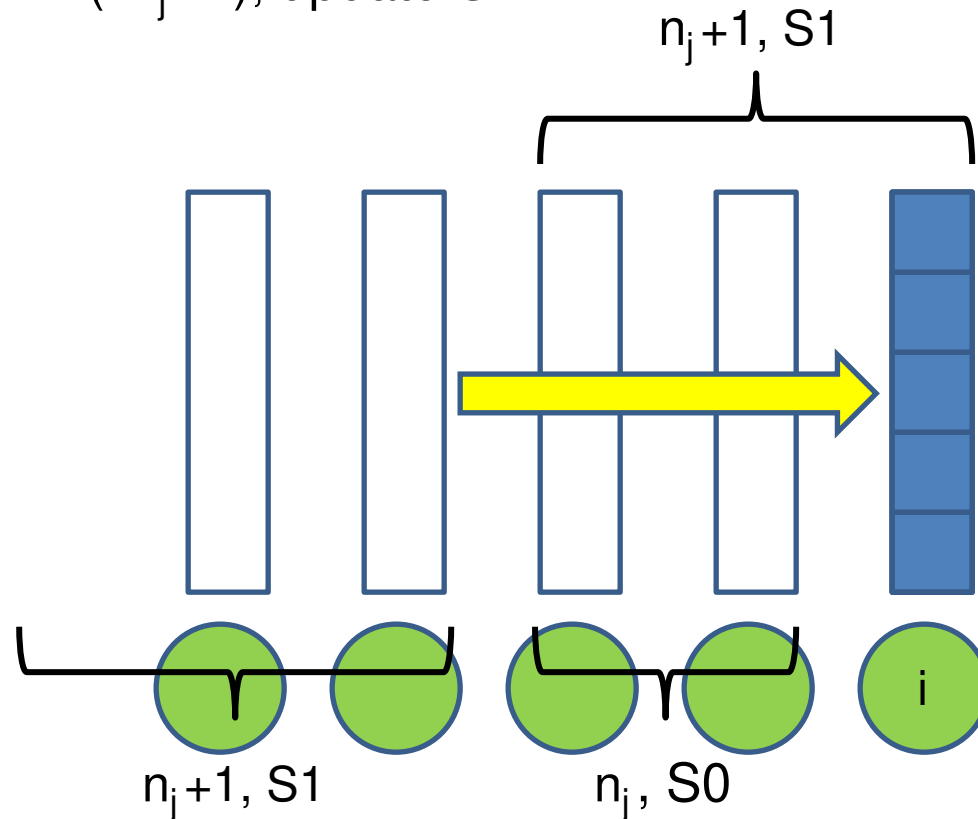
Each process  $i$  maintains the following invariants on  $S1$  and  $S0$ :

- $S1 = \sum_{l=i-n_j}^i B_l$ , the sum of  $n_j + 1$  previous elements **including** the process itself,
- $S0 = \sum_{l=i-n_j}^{i-1} B_l$ , the sum of  $n_j$  previous elements **excluding** the process itself,

When  $j=k-1$ ,  $S1$  is the desired result for each process (since  $n_j = n_{k-1} = p-1$ )

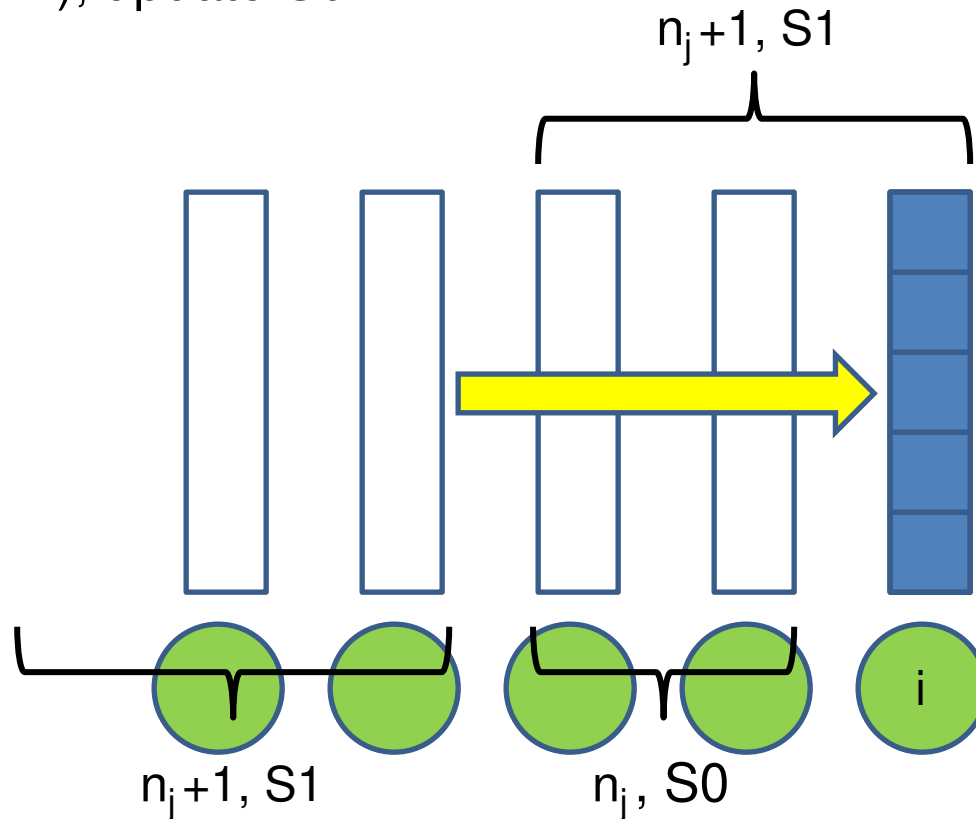
Maintaining the invariants,  $n_{j+1} = 2n_j + b_{j+1}$

Case analysis,  $b_{j+1}=1$ :  $n_{j+1}+1 = (2n_j+1)+1 = (n_j+1)+(n_j+1)$ , receive S1 from  $i-(n_j+1)$ , update S1



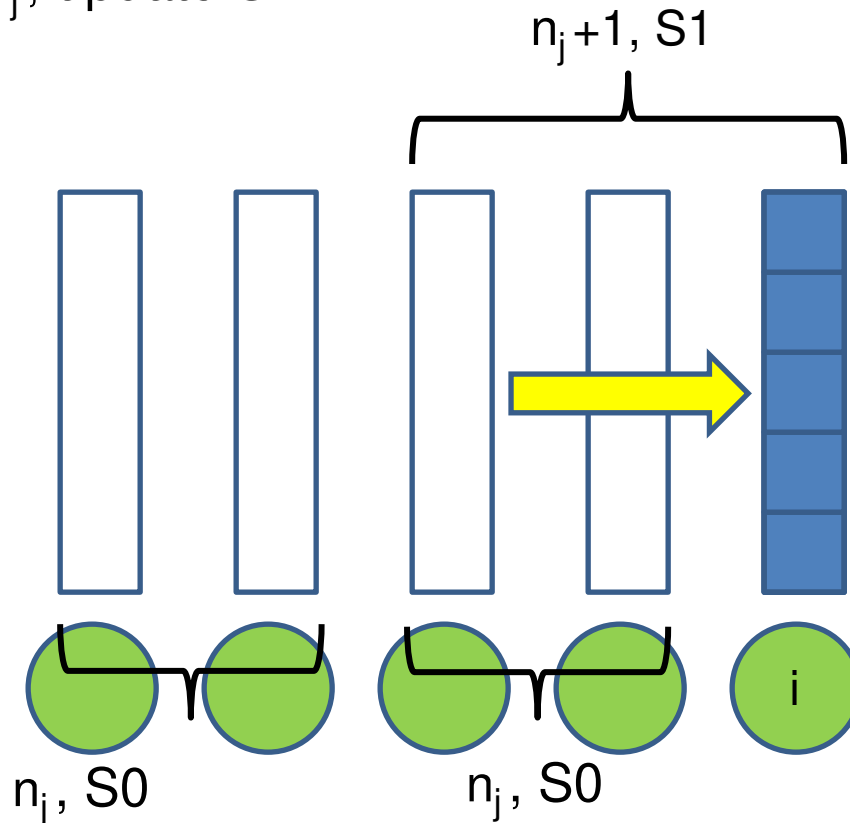
Maintaining the invariants,  $n_{j+1} = 2n_j + b_{j+1}$

Case analysis,  $b_{j+1} = 1$ :  $n_{j+1} = (2n_j + 1) = n_j + (n_j + 1)$ , receive S1 from  $i - (n_j + 1)$ , update S0



Maintaining the invariants,  $n_{j+1} = 2n_j + b_{j+1}$

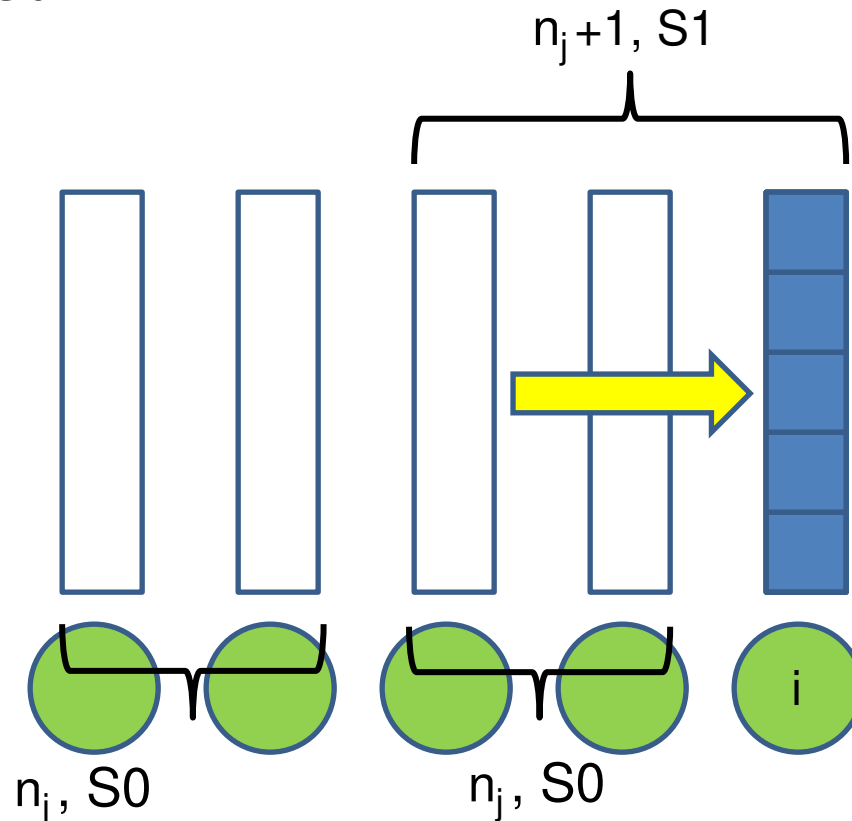
Case analysis,  $b_{j+1}=0$ :  $n_{j+1}+1 = (2n_j+0)+1 = (n_j+1)+n_j$ , receive S0 from  $i-n_j$ , update S1





Maintaining the invariants,  $n_{j+1} = 2n_j + b_{j+1}$

Case analysis,  $b_{j+1} = 0$ :  $n_{j+1} = (2n_j + 0) = n_j + n_j$ , receive S0 from  $i - n_j$ ,  
update S0



## Case analysis summary:

In round  $j$ :

- Receive  $S$  and update both  $S1 = S1 + S$  and  $S0 = S0 + S$
- If  $b_j = 1$ , receive  $S1$  from  $i - (n_j + 1)$  and send  $S1$  to  $i + (n_j + 1)$
- If  $b_j = 0$ , receive  $S0$  from  $i - n_j$  and send  $S0$  to  $i + n_j$

$$\text{Tallreduce}(m) = (\log p)(\alpha + \beta m)$$

Optimal in  $\alpha$  term

```

Algorithm ( $b_j$  bits of  $p-1$ ):
S1 = B /* input vector */
S0 = 0 /* neutral element under  $\Sigma$  */
n = 0
for j=0,1,...,k-1: /* round */
    if  $b_j=1$ 
        Send S1 to process rank+n+1
        Receive S from rank-(n+1)
    else
        Send S0 to process rank+n
        Receive S from rank-n
    S1 = S1+S
    S0 = S0+S
    n = 2n+b_j
  
```

But: For commutative operations only (+-like). Why?

All rank calculations mod  $p$

Algorithmic latency : find msb

**Remark:** A neutral element for  $\Sigma$  is actually **not needed**, S0 can be initialized after first round (recall  $b_0=1$ ).

Examples:

Communication neighbor



$p$	$p-1$	$(p-1)_2$	$n_{ij} (i=0,1,2,3)$	$n_{ij}+b_{ij} (i=0,1,2,3)$
5	4	$(100)_2$	0,1,2	1,1,2
6	5	$(101)_2$	0,1,2	1,1,3
7	6	$(110)_2$	0,1,3	1,2,3
8	7	$(111)_2$	0,1,3	1,2,4
9	8	$(1000)_2$	0,1,2,4	1,1,2,4
10	9	$(1001)_2$	0,1,2,4	1,1,2,5
11	10	$(1010)_2$	0,1,2,5	1,1,3,5
12	11	$(1011)_2$	0,1,2,5	1,1,3,6
13	12	$(1100)_2$	0,1,3,6	1,2,3,6

Same  
neighbor  
twice!

## Notes:

- Two papers with same idea, at the same time, from two different IBM labs (T. J. Watson/Almaden)!?
- Second paper generalizes to k-ported systems, and have other extended results

Can be used for the merge problem

Can the idea be made to work for non-commutative functions  $\Sigma$  ?

Amotz Bar-Noy, Shlomo Kipnis, Baruch Schieber: An Optimal Algorithm for computing Census Functions in Message-Passing Systems. Parallel Processing Letters 3: 19-23 (1993)

Jehoshua Bruck, Ching-Tien Ho: Efficient Global Combine Operations in Multi-Port Message-Passing Systems. Parallel Processing Letters 3: 335-346 (1993)

The same result, derived in a different way, also in

Jesper Larsson Träff, Sascha Hunold, Ioannis Vardas, Nikolaus Manes Funk: Uniform Algorithms for Reduce-scatter and (most) other Collectives for MPI. CLUSTER 2023: 284-294

The communication pattern and algorithms is extended to reduce-scatter (reduce, allgather, gather/scatter)

## Scan/Exscan in fully-connected networks

Hillis-Steele ( **reminder** from Bachelor lecture, sketch):

Processor  $i$ , round  $k$ ,  $k=0, 1, \dots, \text{ceil}(\log p)-1$ :

1. Receive partial result from processor  $i-2^k$  (if  $i-2^k \geq 0$ )
2. Send own partial result to processor  $i+2^k$  (if  $i+2^k < p$ )
3. Compute partial result for next round by summing own and received partial result

Partial result:  $\sum_{\max(0, i-2^k) < j \leq i} x_j$

$T_{\text{scan}}(m) = \text{ceil}(\log p)(\alpha + \beta m)$

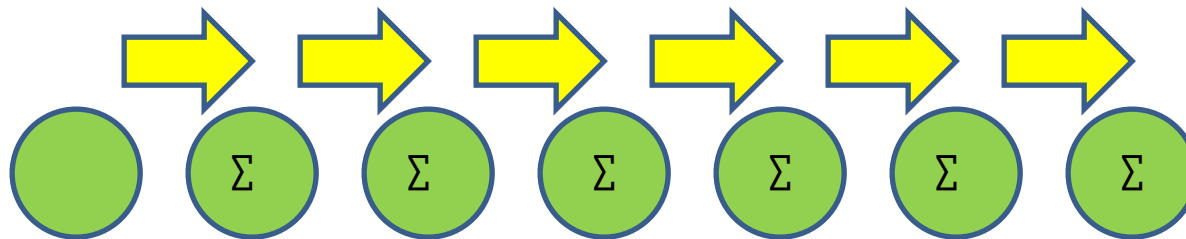
Not optimal in  $\beta$ -term

**Note :**

Attribution not correct,  
algorithm was known before  
Hillis and Steele

Round 0:

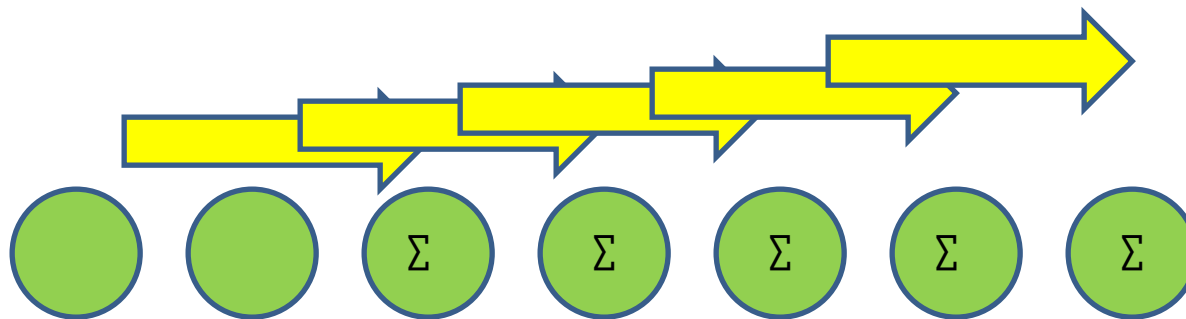
1. Sendrecv(rank-1,rank+1)
2. Add received partial result to own vector





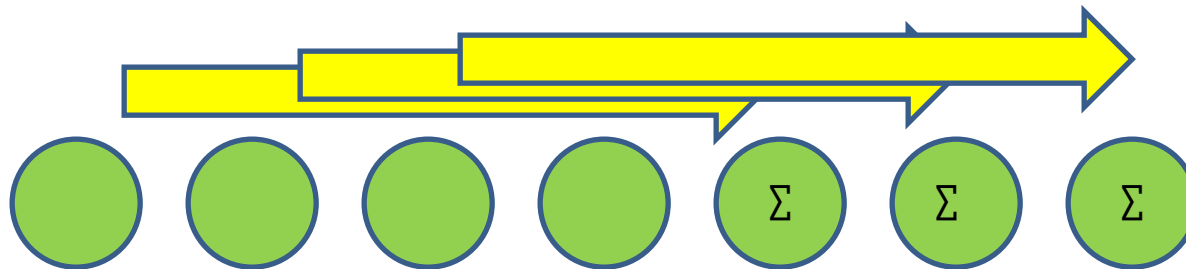
Round 1:

1. Sendrecv(rank-2,rank+2)
2. Add received partial result to own vector



Round 2:

1. Sendrecv(rank-4,rank+4)
2. Add received partial result to own vector



W. Daniel Hillis, Guy L. Steele Jr.: Data Parallel Algorithms.  
Commun. ACM 29(12): 1170-1183 (1986)

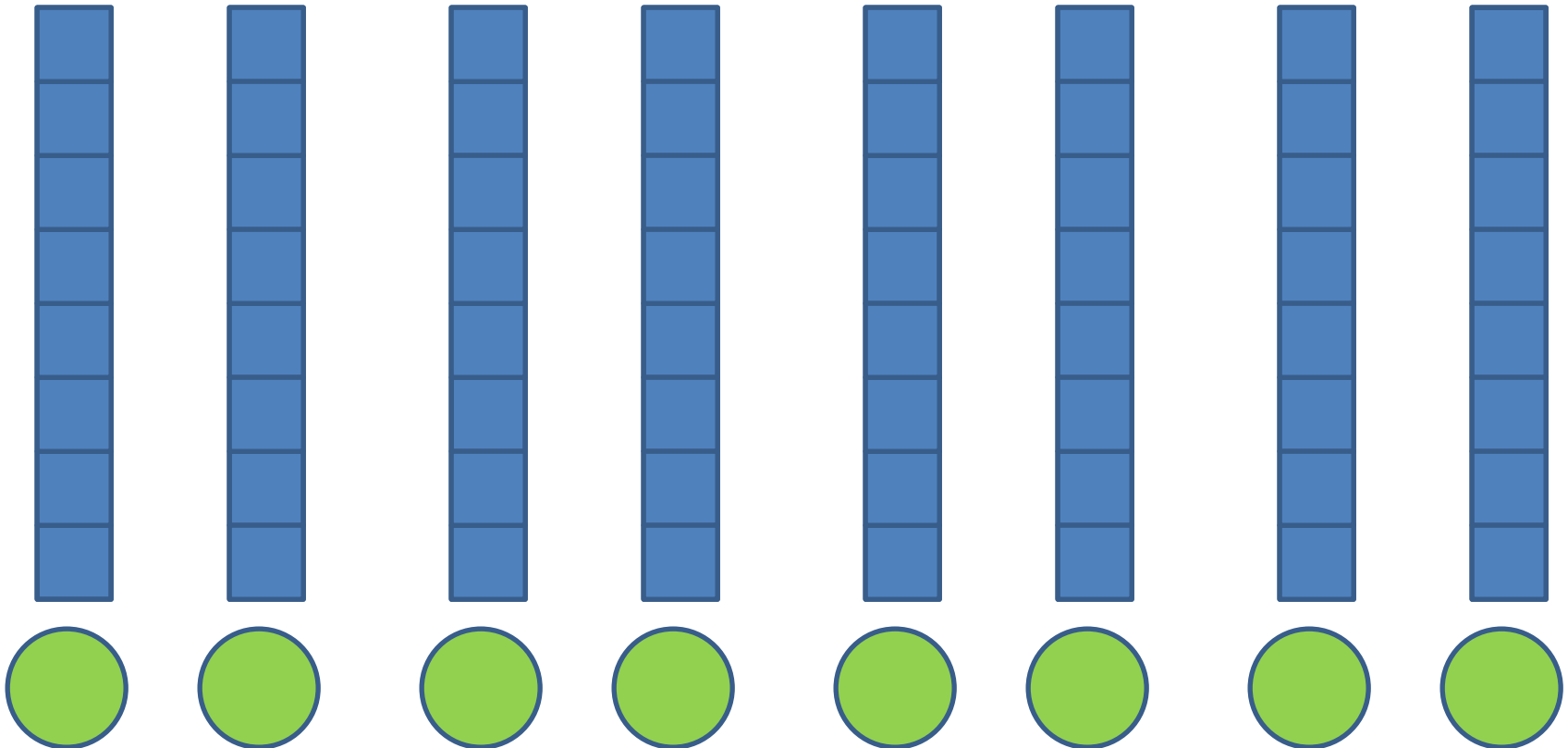
## The power of the hypercube/butterfly: Allreduce, ReduceScatter

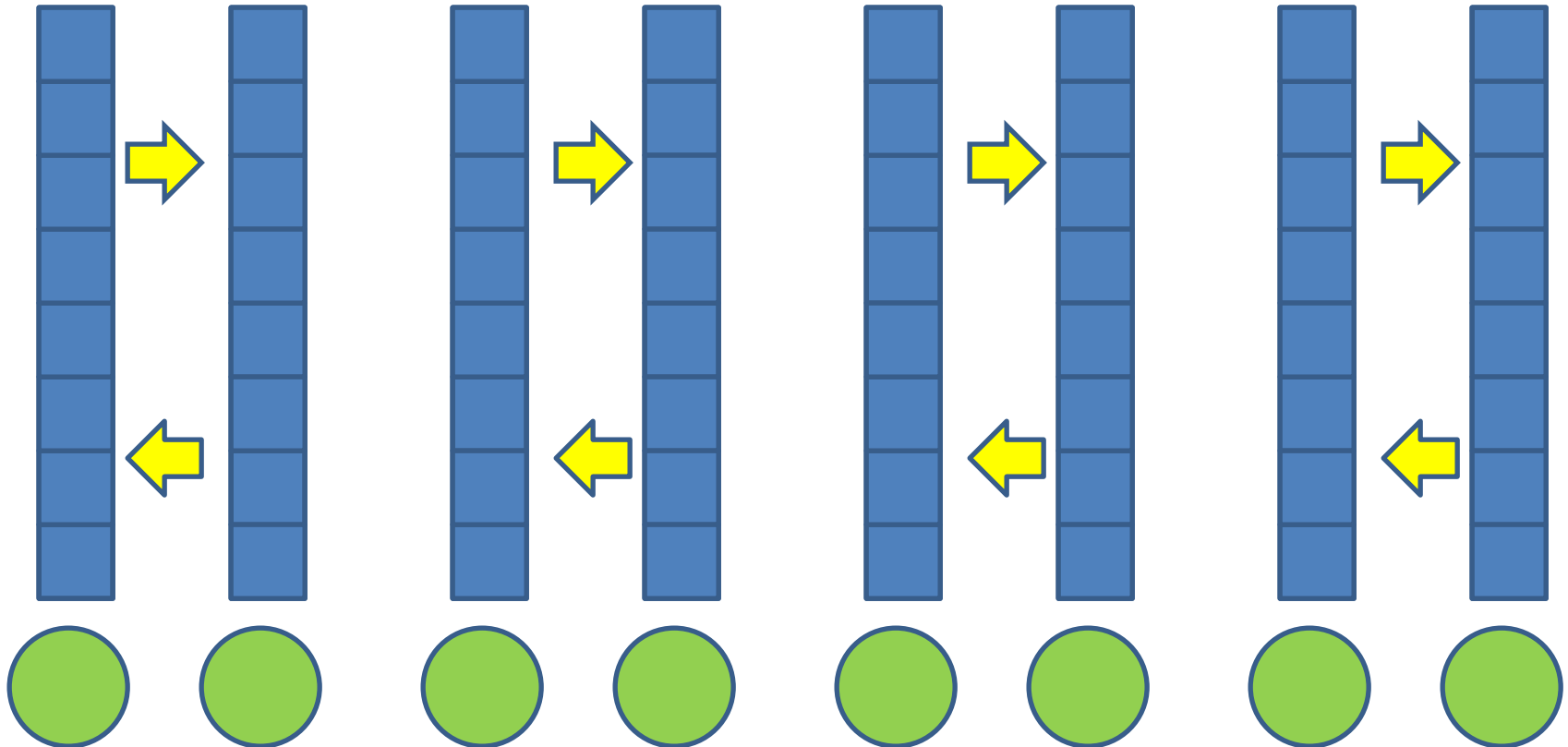
Building on the allreduce = reducescatter+allgather observation, the butterfly can be used for a good (best possible?) Allreduce algorithm

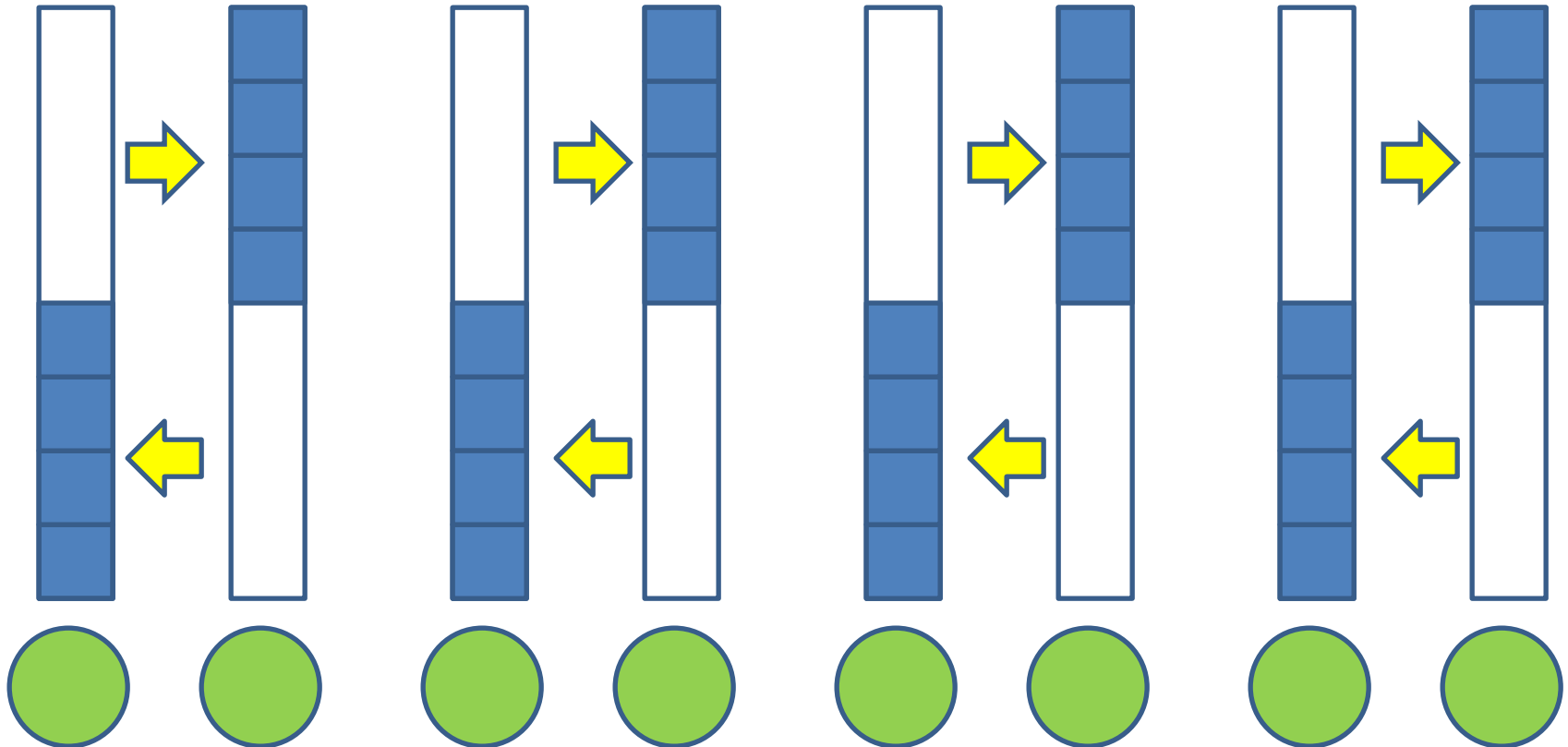
### Note :

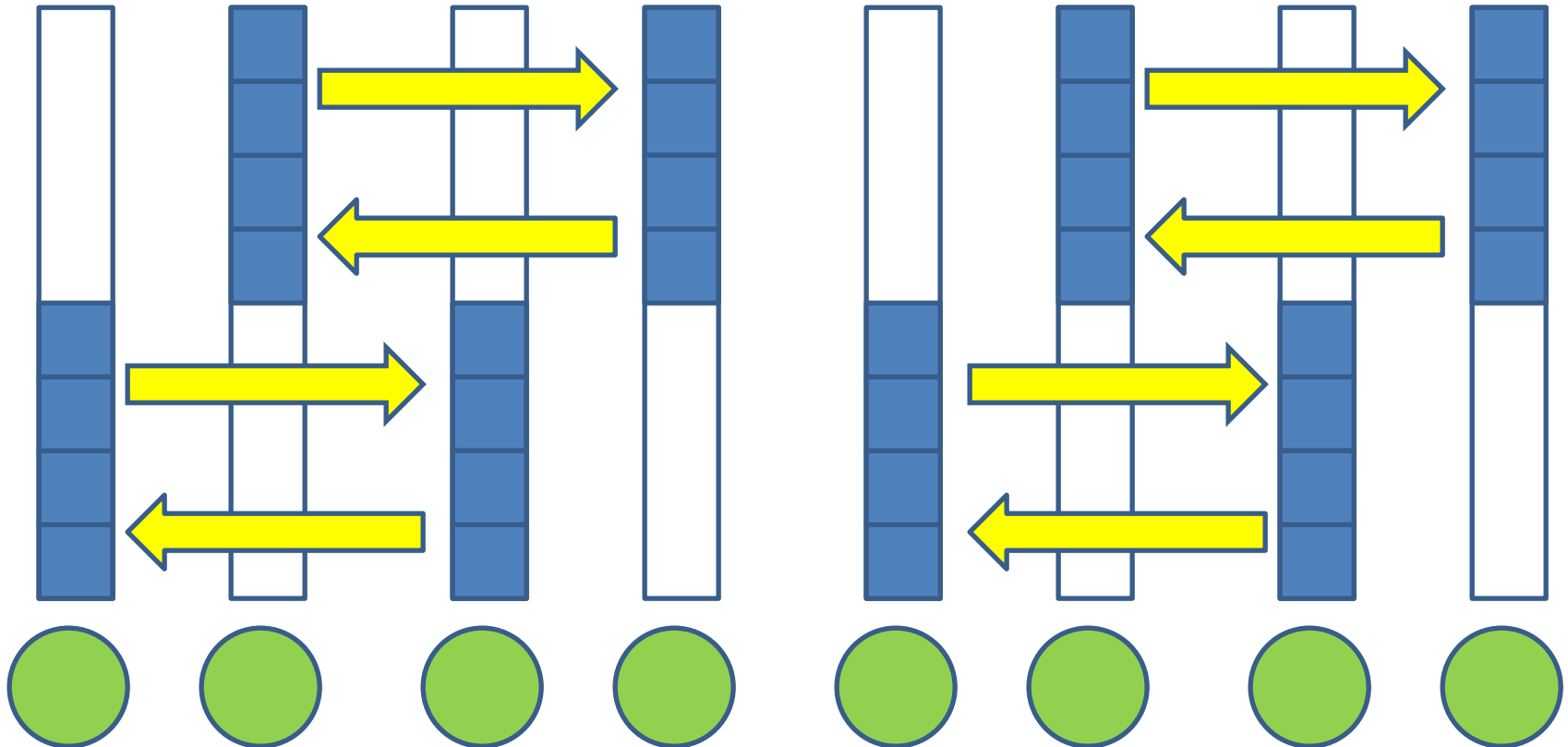
In MPI community the algorithm is sometimes called **Rabenseifner's algorithm** , although the algorithm has been known for longer in the parallel/distributed computing field (incorrect attribution), see for instance

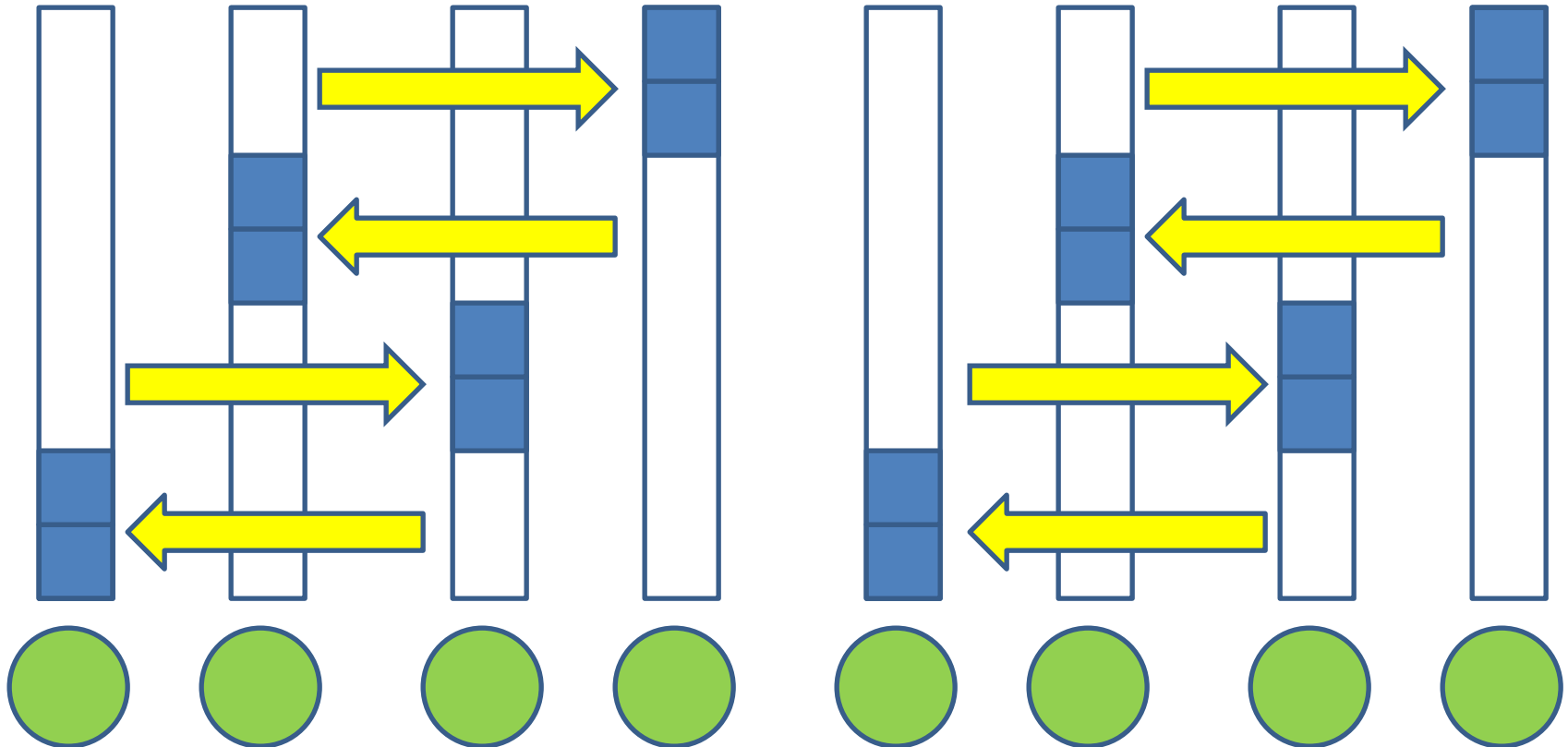
Robert A. van de Geijn: On Global Combine Operations. J. Parallel Distrib. Comput. 22(2): 324-328 (1994)



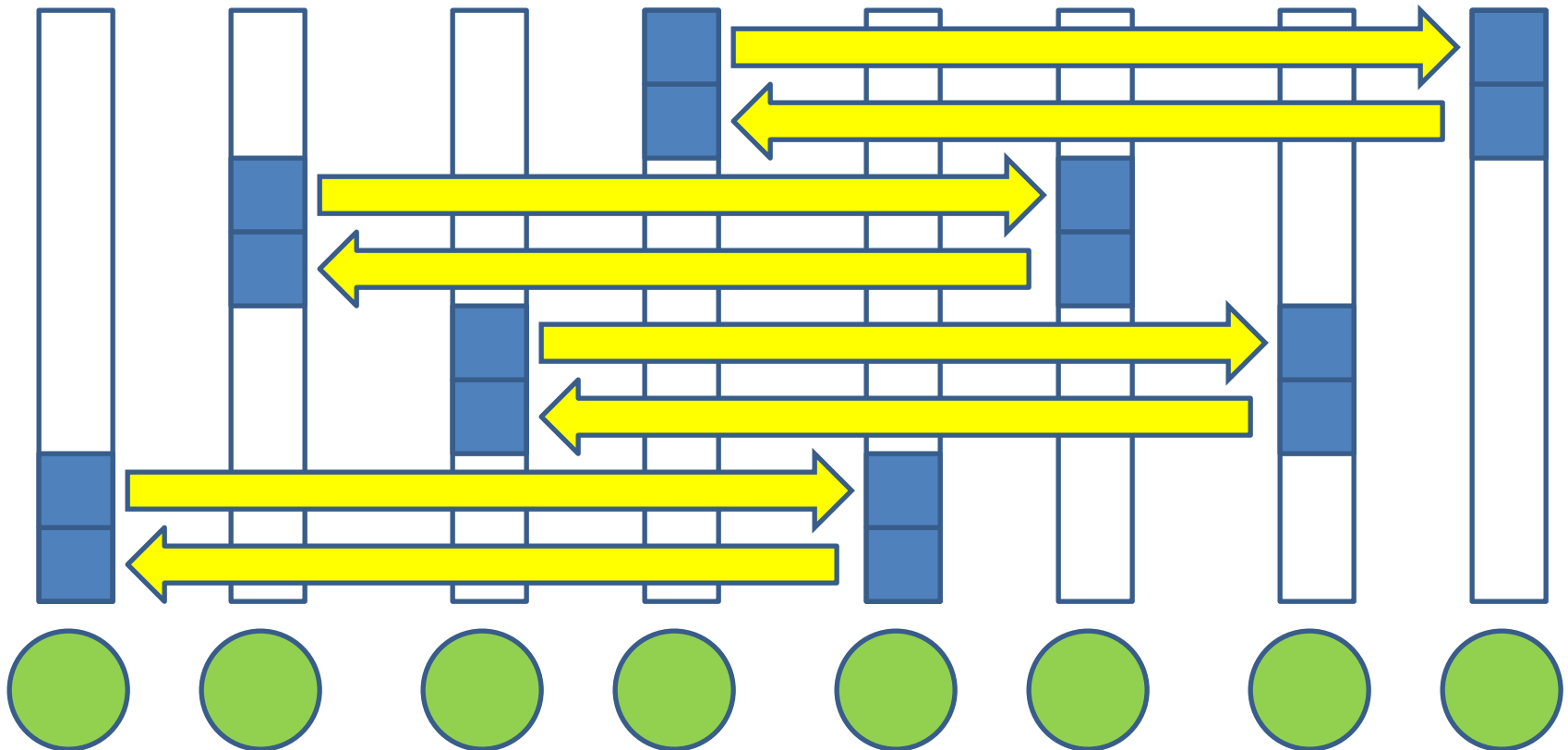


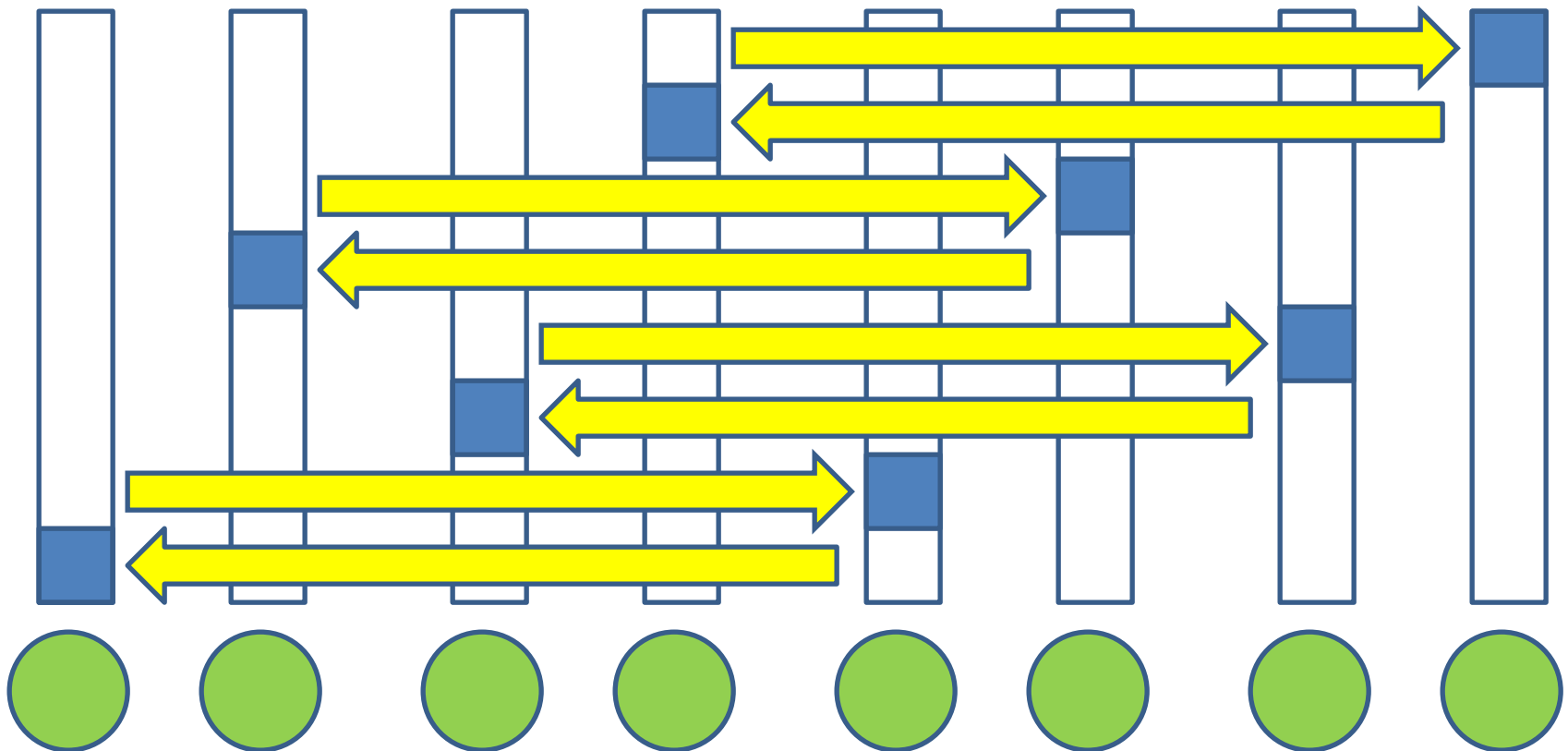




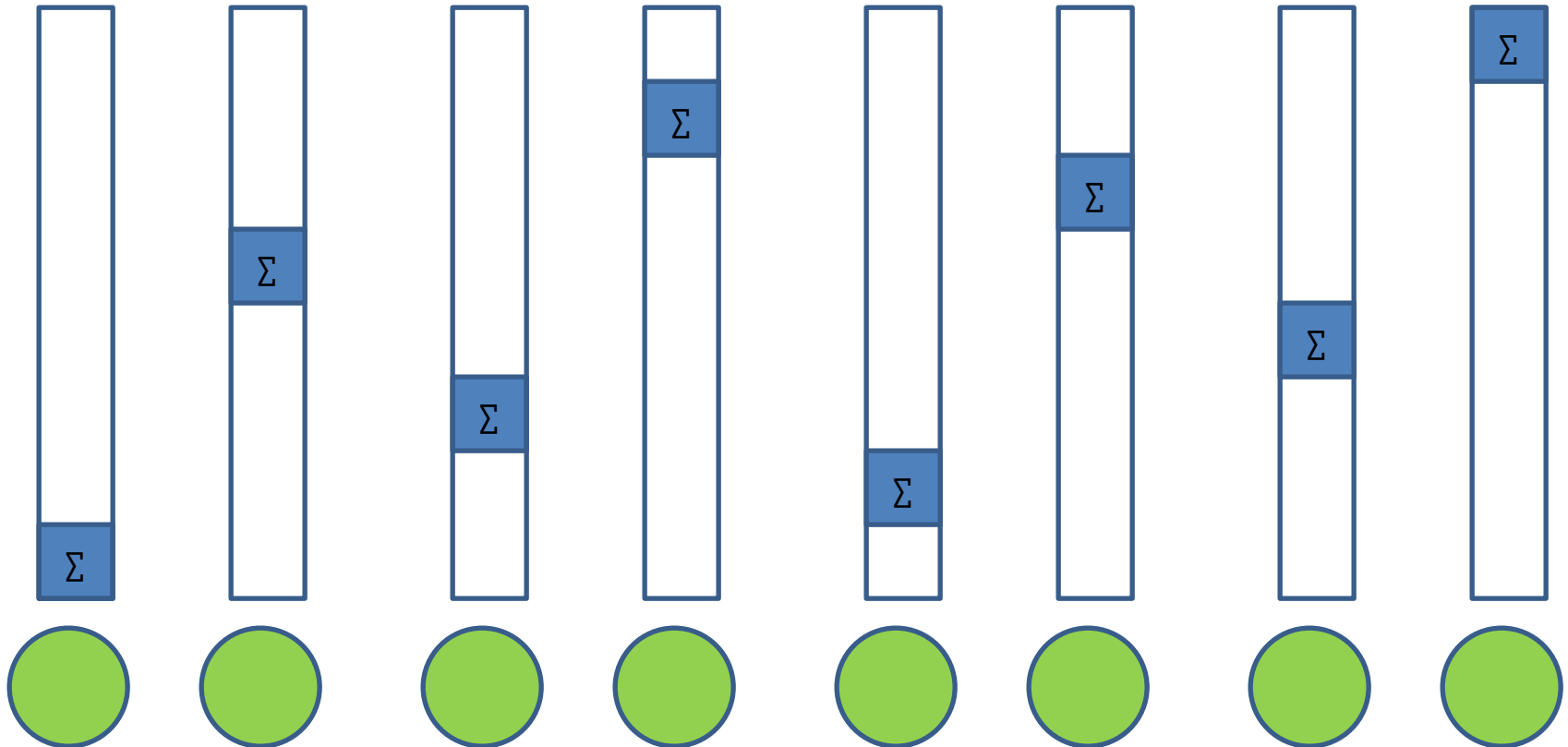




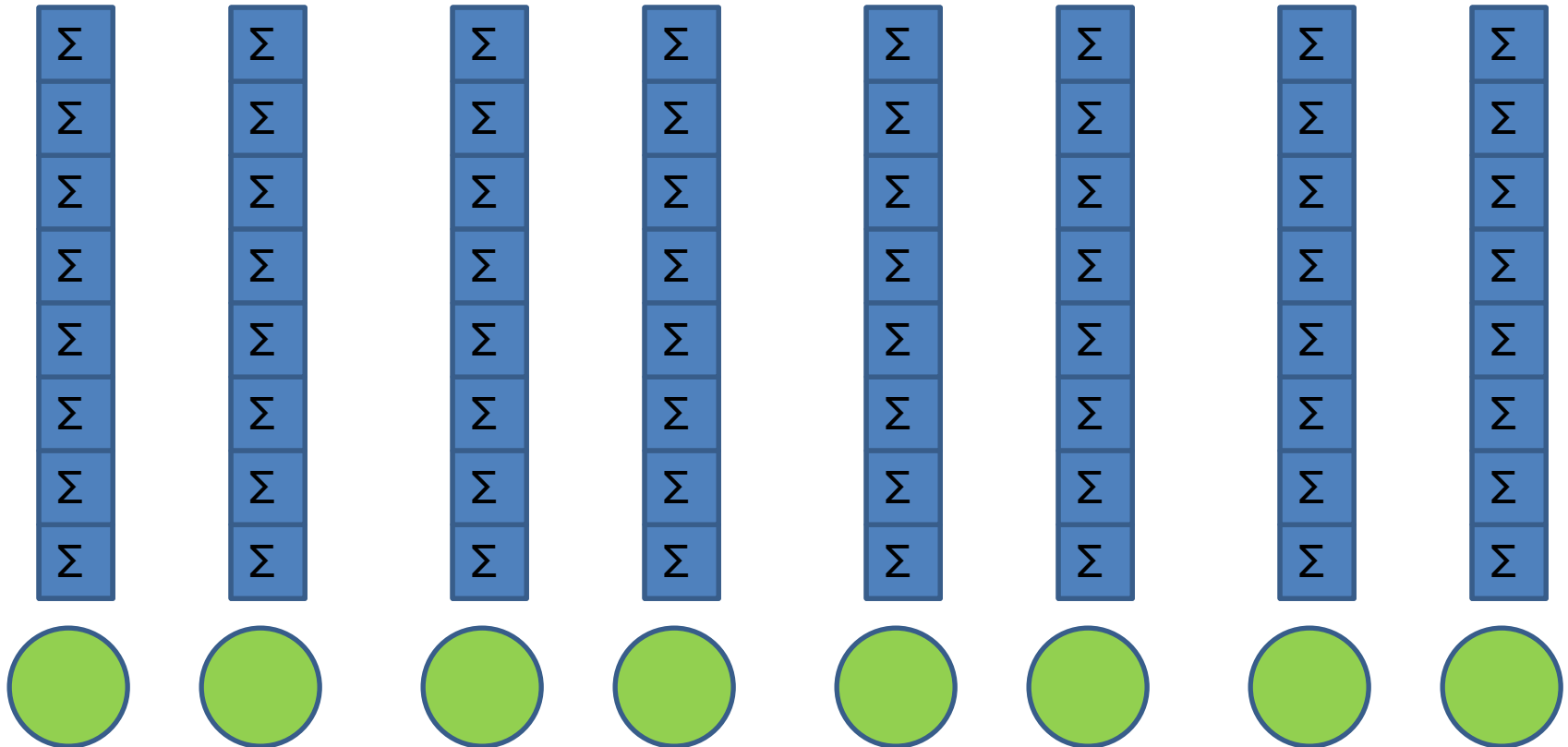




$$T_{\text{reduce}}(\text{scatter}')(\text{m}) = (\log p)\alpha + (p-1)/p \beta m$$



Reverse the process to allgather the result:



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

Reducescatter part, **MPI note** :

The blocks are not scattered as prescribed by MPI, i'th block at i'th process

Simple trick: Reorder blocks (FFT permutation) before starting, at the cost of an  $O(m)$  **algorithmic latency**

J. L. Träff: An Improved Algorithm for (Non-commutative) Reduce-Scatter with an Application. PVM/MPI 2005: 129-137

**Again:** Butterfly algorithm does not extend nicely to case where  $p$  is not a power of two. A better than trivial algorithm in:

R. Rabenseifner, J. L. Träff: More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems. PVM/MPI 2004: 36-46

## The power of fixed-degree trees

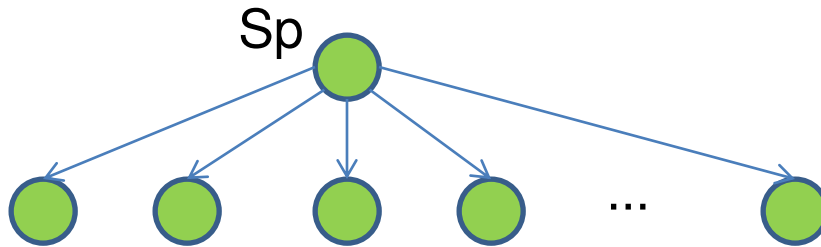
Binomial trees **cannot be pipelined** : For same block size  $m/M$ , different nodes in tree would have different amount of work per round

Fixed-degree trees, e.g., linear pipeline, binary trees, admit pipelining

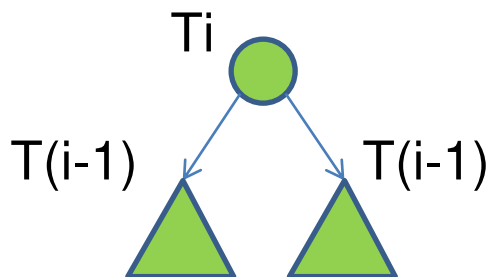
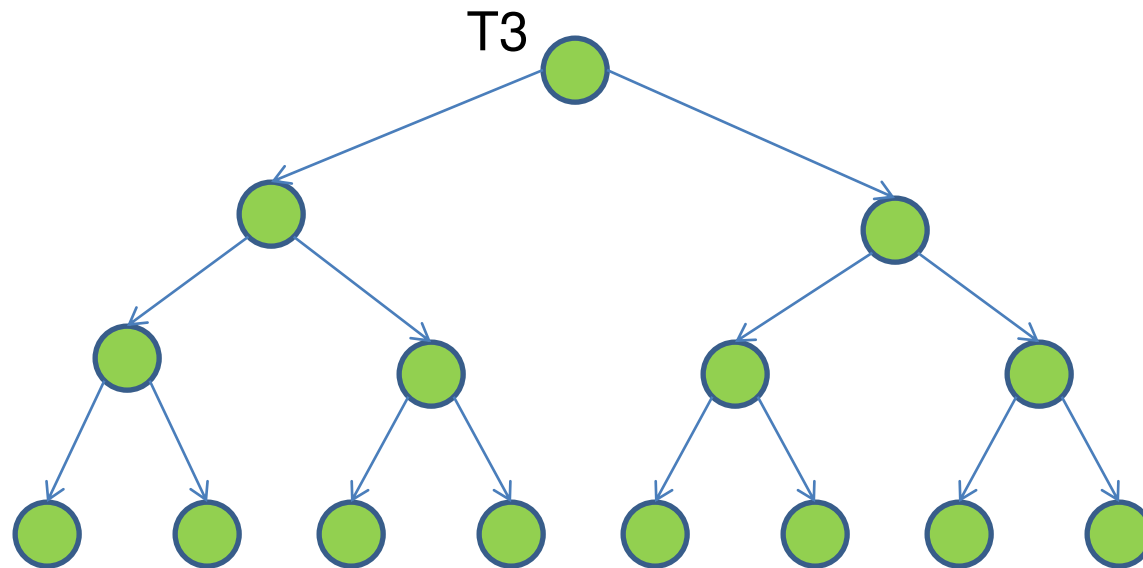


**Note :**

Extreme case, non-constant degree star(tree) can be useful, e.g. gather for very large problems (MPI: No need for intermediate buffering); pipelining can be employed for each child:



## Complete, balanced, binary tree: Structure

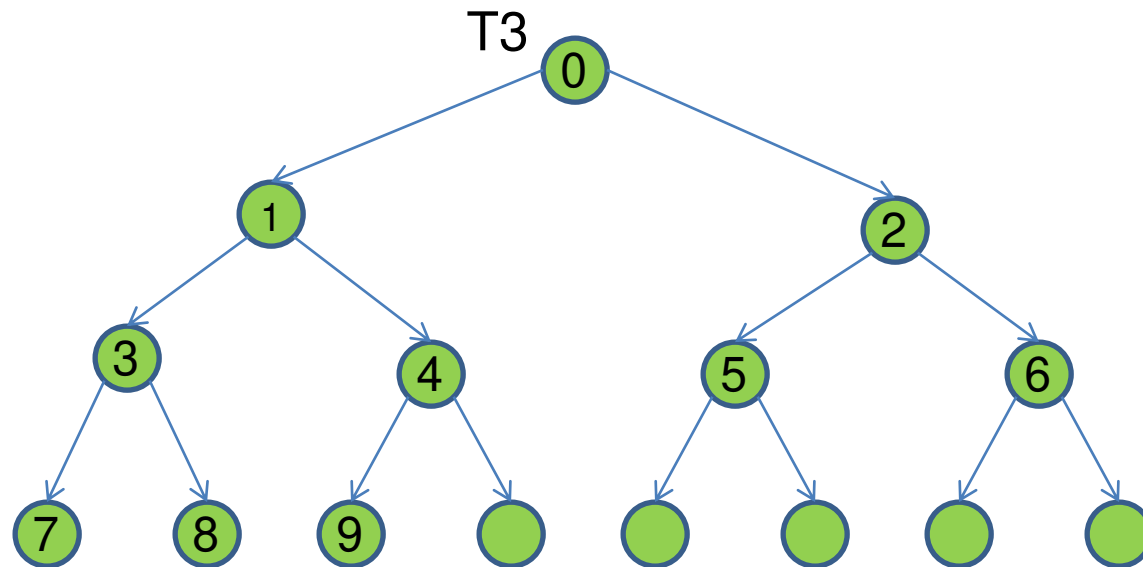


### Properties:

- $T_i$  has  $2^{(i+1)} - 1$  nodes,  $i \geq 0$
- $T_i$  has  $2^i - 1$  interior nodes
- $T_i$  has  $2^i$  leaves
- $T_i$  has  $i+1$  levels



## Complete, balanced, binary tree: Naming (BFS)

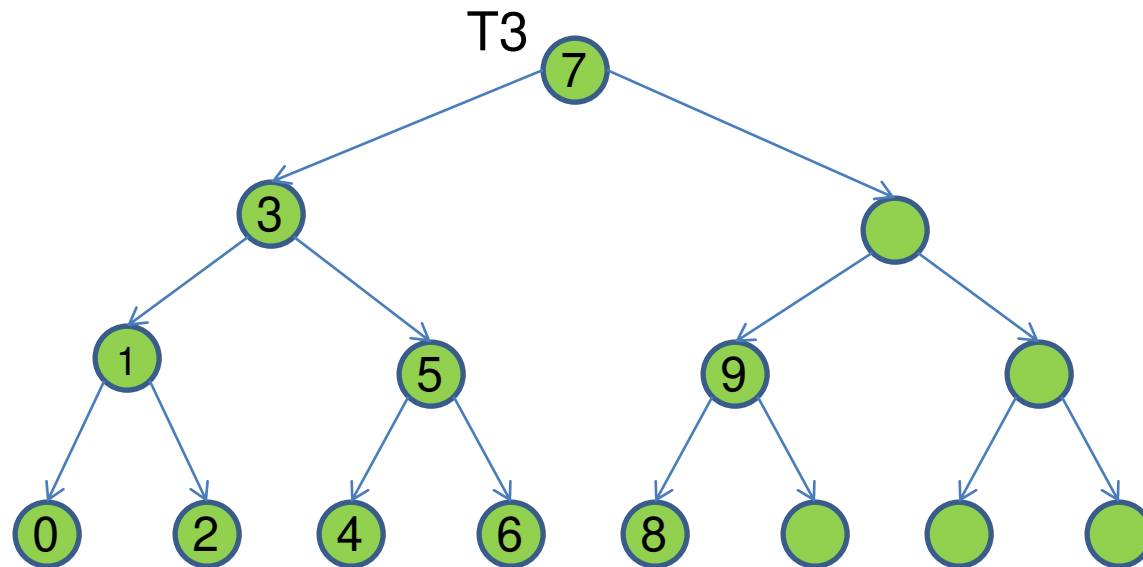


Navigation in  $O(1)$ :

parent is  $(\text{rank}-1)/2$ , children  $2\text{rank}+1$ ,  $2\text{rank}+2$

**For MPI : Not always convenient** , processor ranks of subtrees do not form a consecutive range (e.g., in gather/scatter block reordering necessary; reduction not in rank-order, ...)

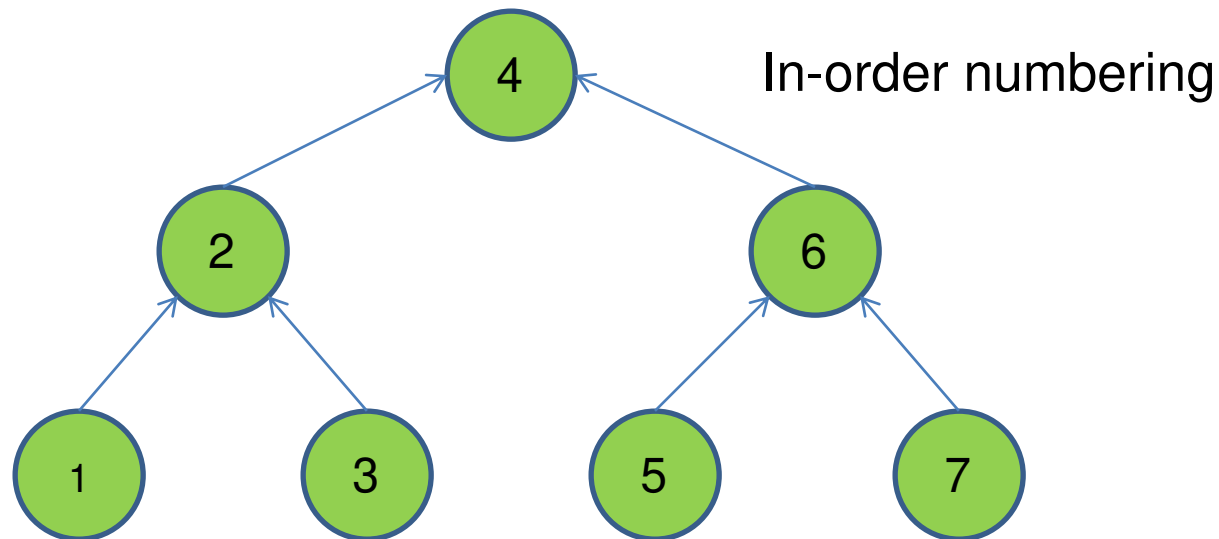
## Complete, balanced, binary tree: Naming (in-order)



**Property** : Processor ranks of subtree from a consecutive range  $[j, \dots, j+2^{k-1}]$  for some  $j$  and  $k$

## Pipelined binary tree

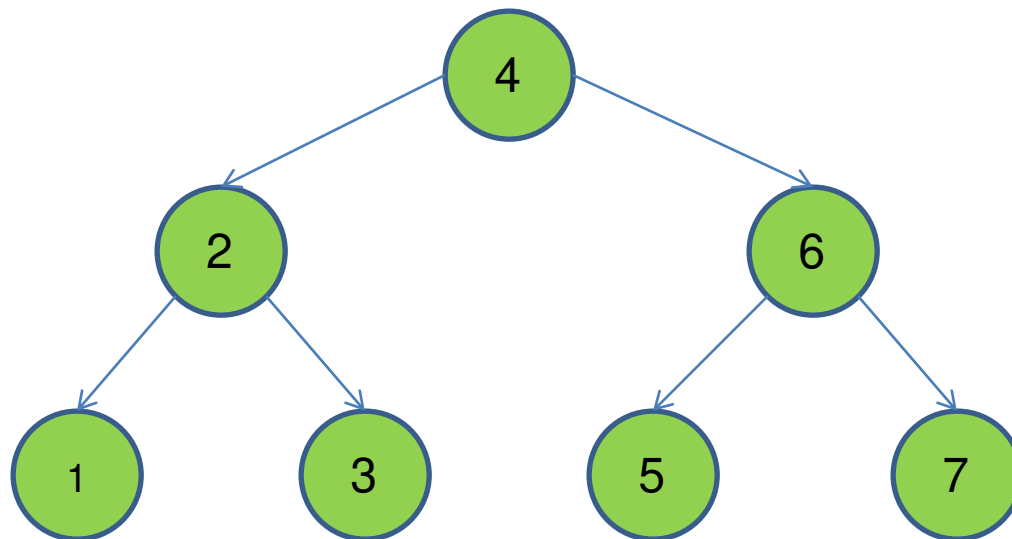
Can be used for broadcast, reduction, scan/exscan; for the latter, inorder numbering of the processors is required (unless MPI operation is commutative)



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

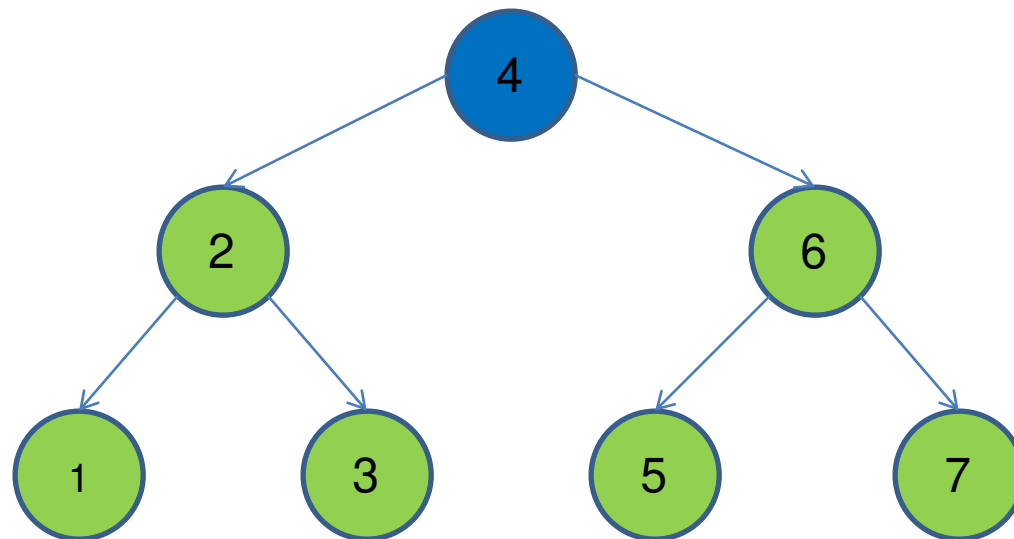
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

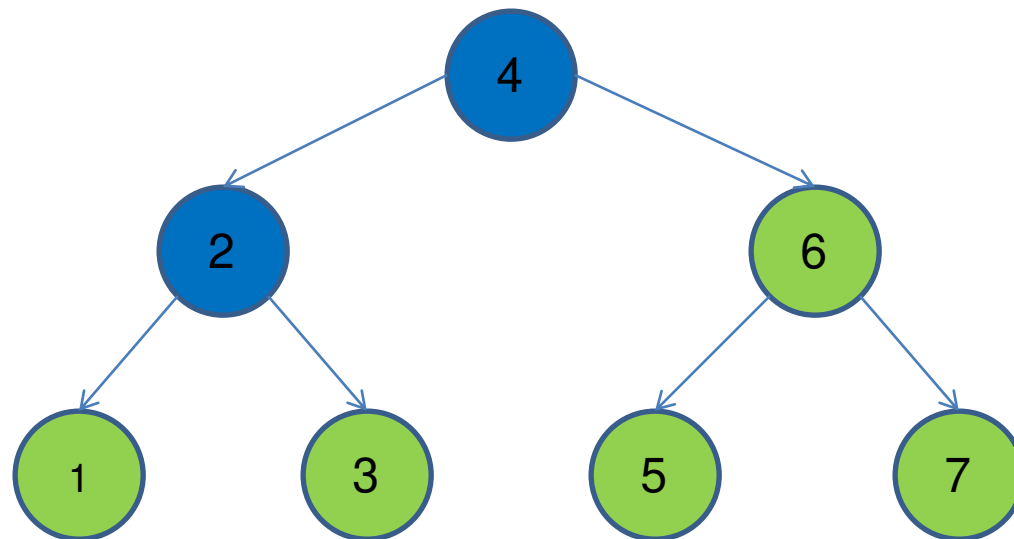
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

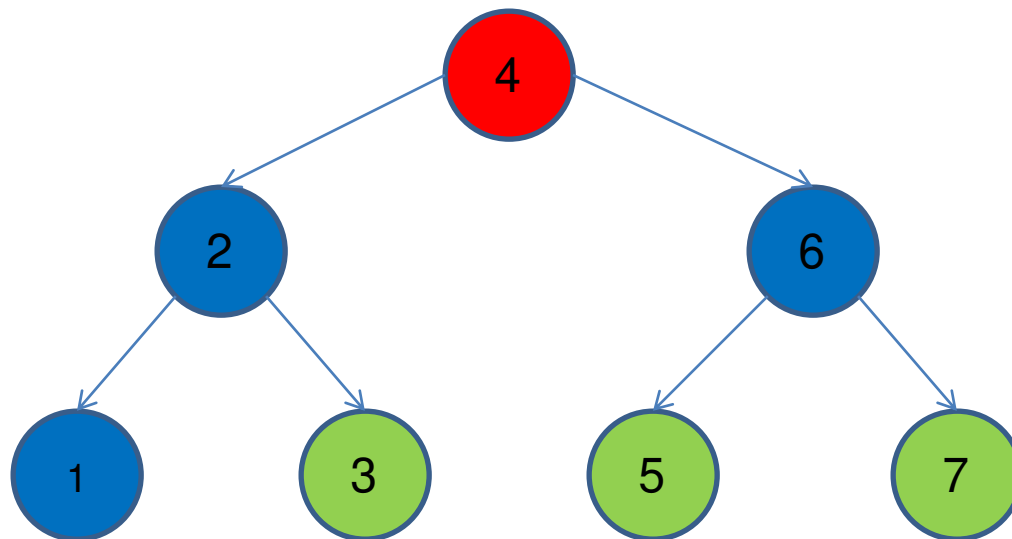
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

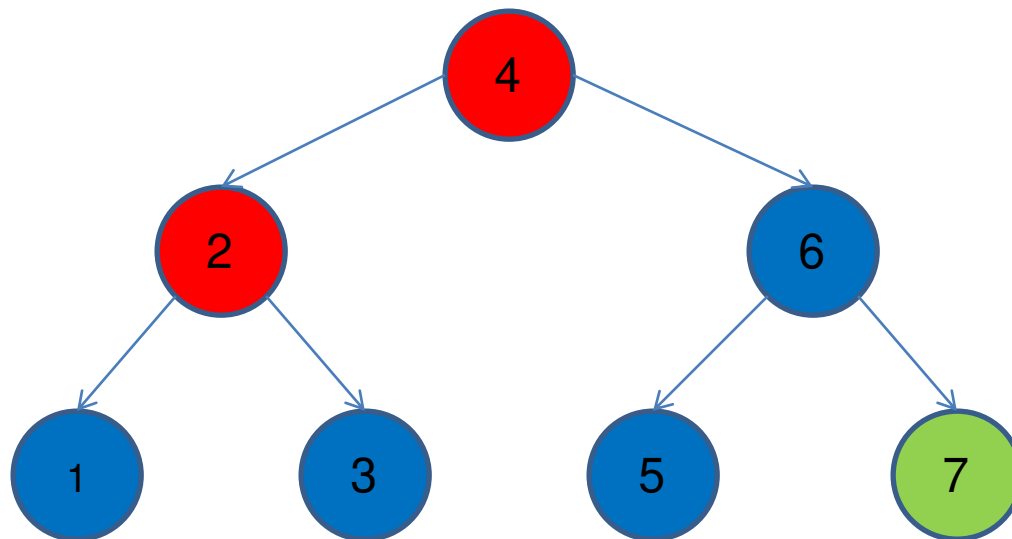
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

M blocks

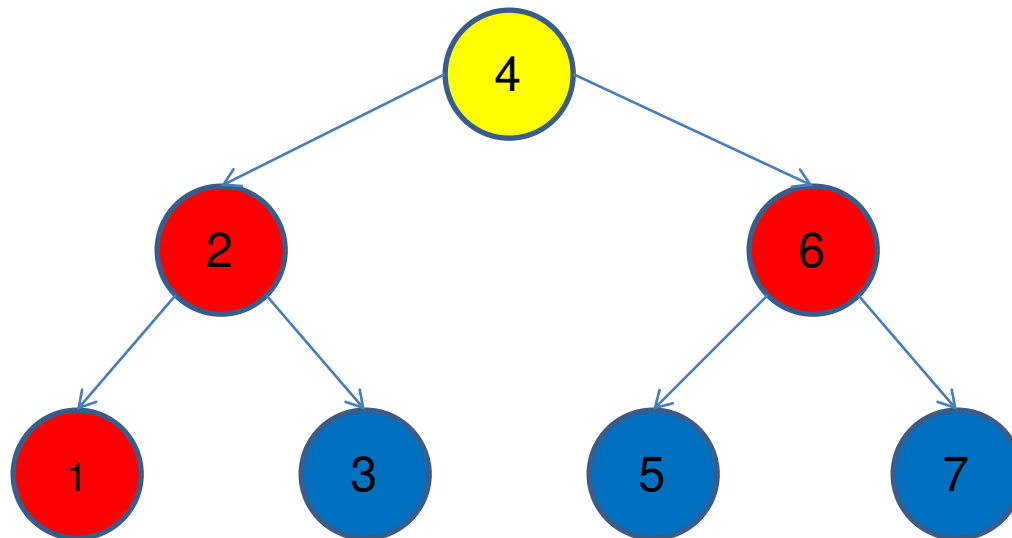




Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

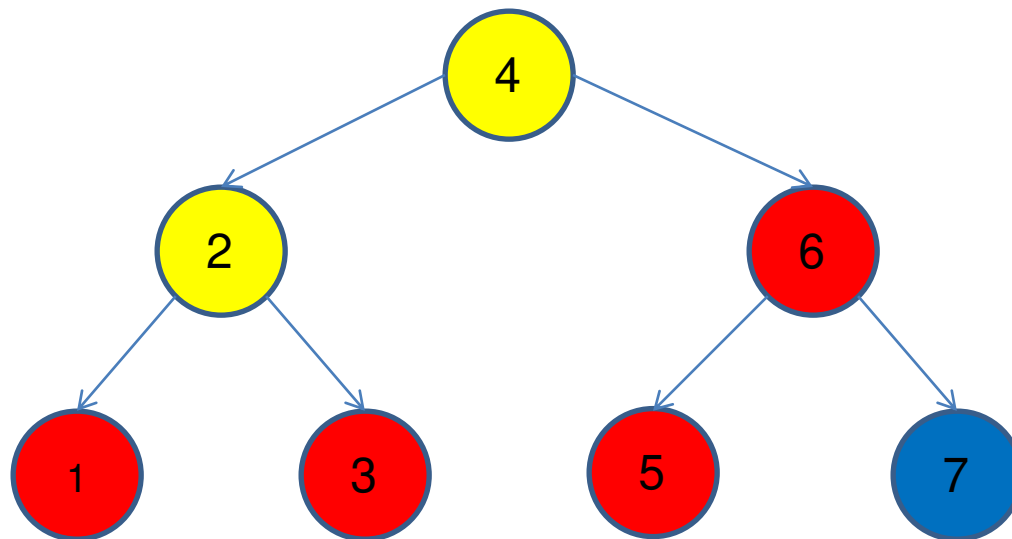
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

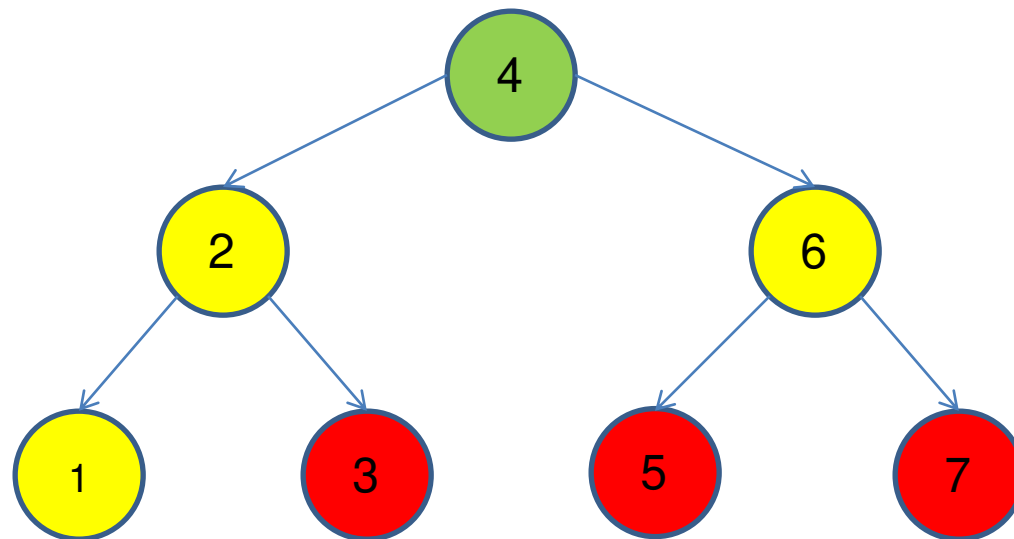
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

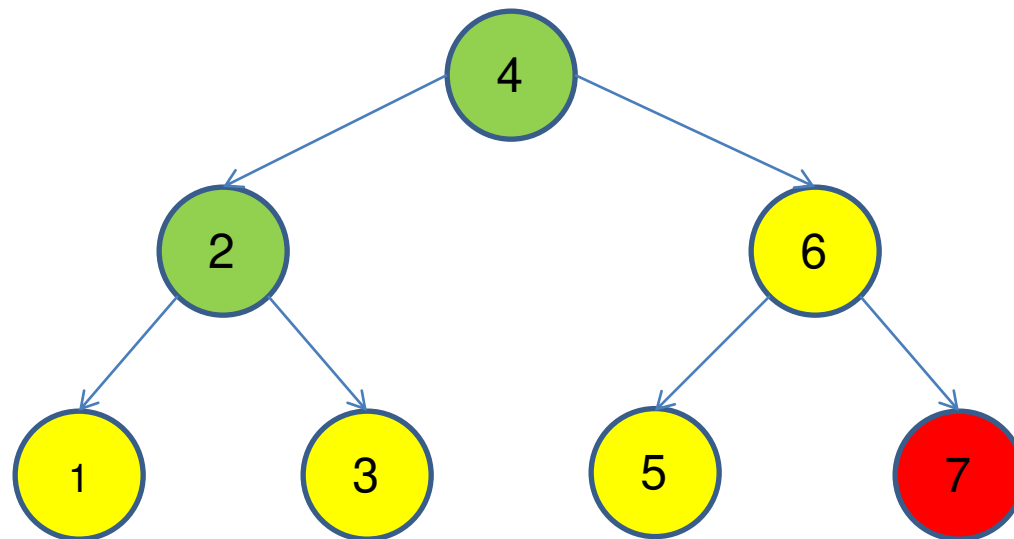
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

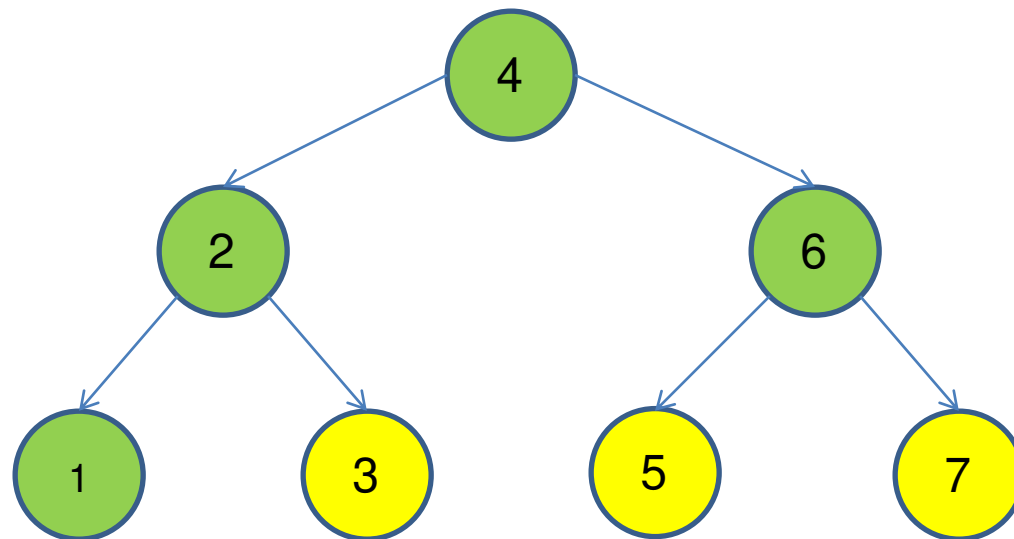
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

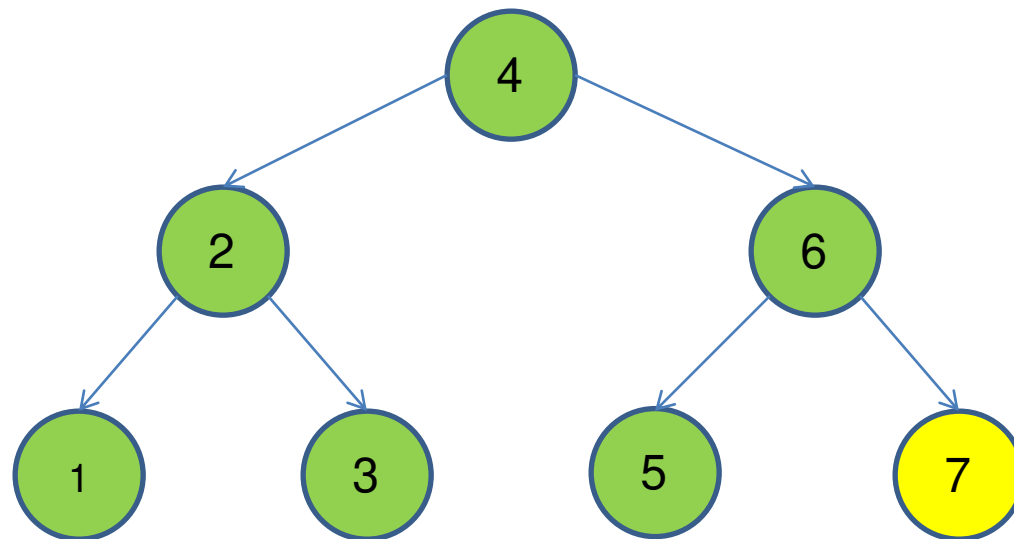
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

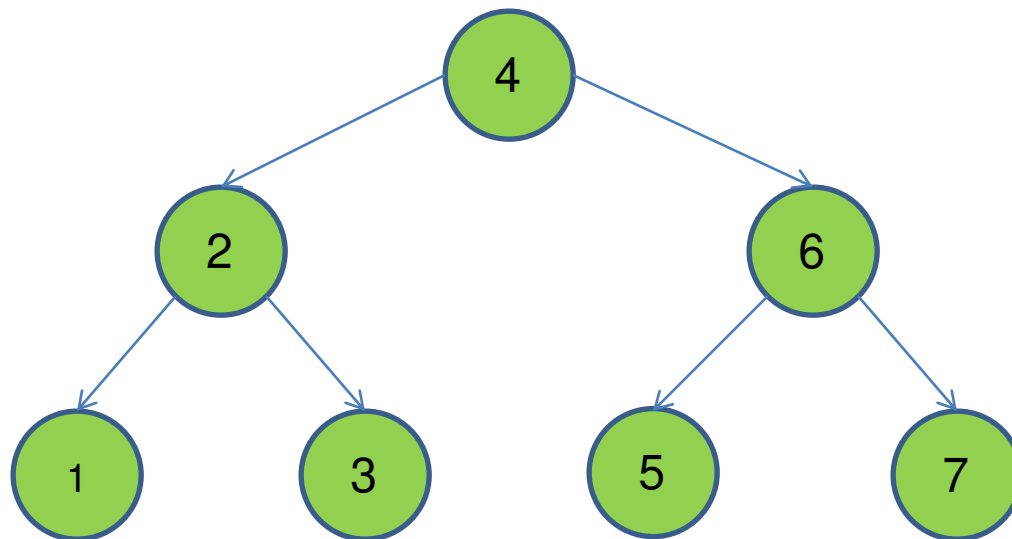
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

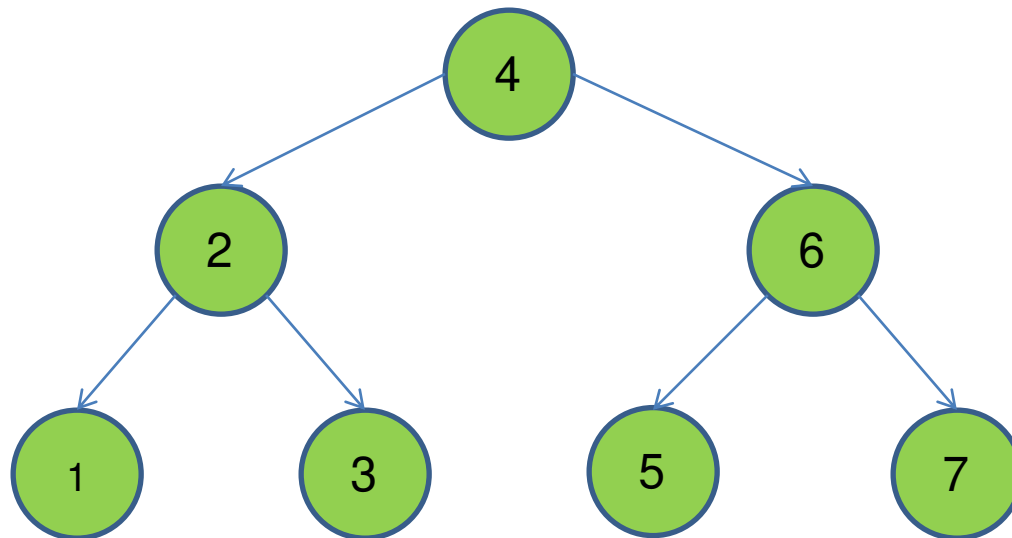
M blocks



Broadcast:  $2(M-1)+x$  rounds

- Root: Send blocks
- Leaf: Receive blocks
- Interior: Receive new block from parent, send previous block to left subtree, send previous block to right subtree

M blocks



Observe:

- Non-root nodes get a new block every second round,  $2(M-1)$
- Last leaf gets first block after  $x=2(\log(p+1)-1)$  rounds



Corollary (pipelining lemma):

$$(k-s)\alpha + 2\sqrt{[s(k-s)\alpha\beta m]} + s\beta m$$

Best possible time for pipelined, binary tree broadcast is

$T_{\text{broadcast}}(m) =$

$$(2\lceil \log((p+1)/4) \rceil)\alpha + 2\sqrt{[4(\lceil \log((p+1)/4) \rceil)\alpha\beta m]} + 2\beta m =$$

$$(2\lceil \log((p+1)/4) \rceil)\alpha + 4\sqrt{[\lceil \log((p+1)/4) \rceil]\alpha\beta m} + \textcolor{red}{2\beta m}$$

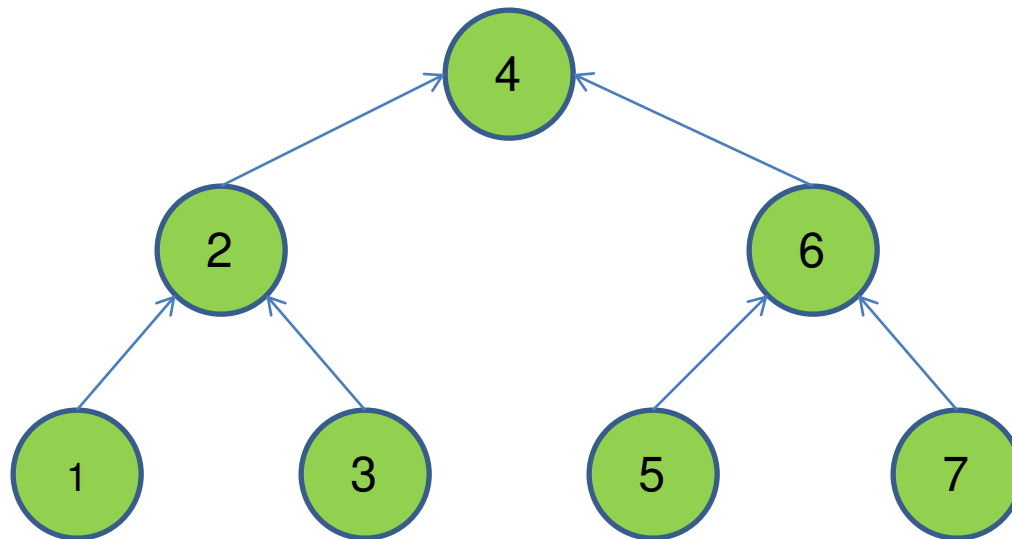
since  $k=2(\lceil \log(p+1) \rceil - 1)$  and  $s=2$ , giving  $k-s = 2(\lceil \log(p+1) \rceil - 2) = 2\lceil \log((p+1)/4) \rceil$

Logarithmic in  $\alpha$ -term

Non-optimal in  $\beta$ -term (factor 2 off)

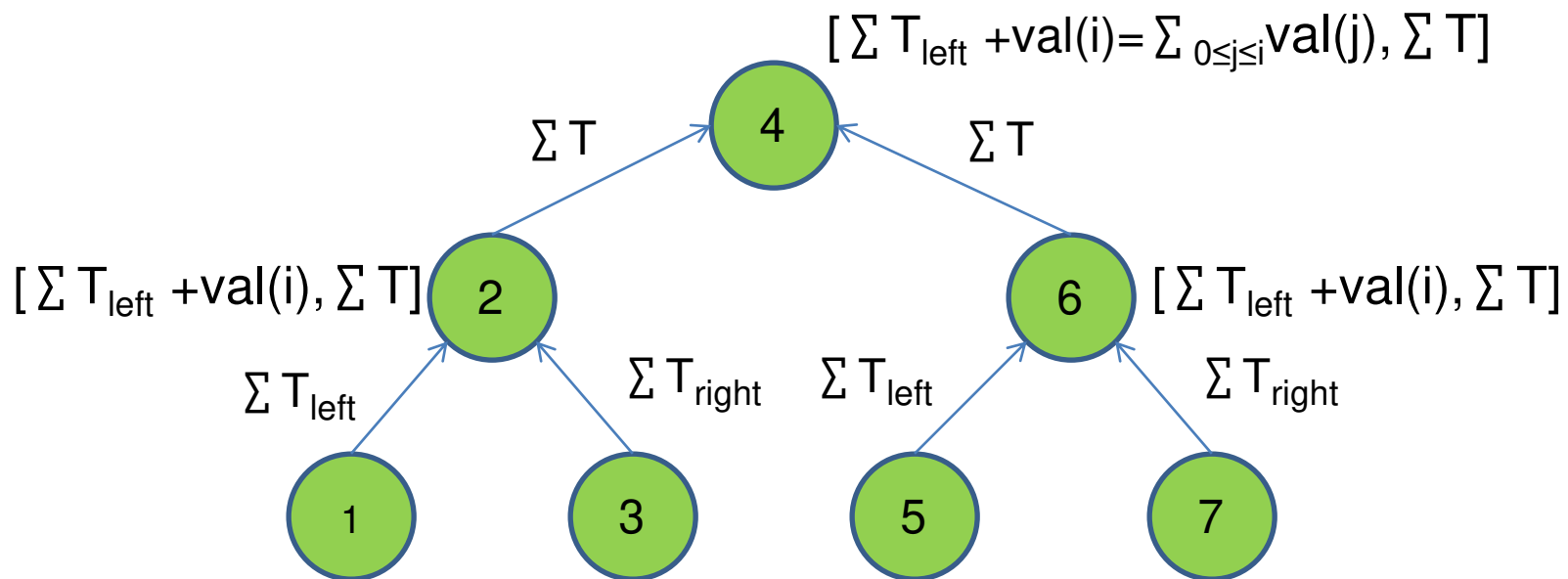
Reduction:  $2(M-1)+2(\log(p+1)-1)$  rounds

- Leaf: Send blocks
- Interior and root: Receive from left subtree, add to partial result, receive from right subtree, add to partial result, send to parent (root does not send)



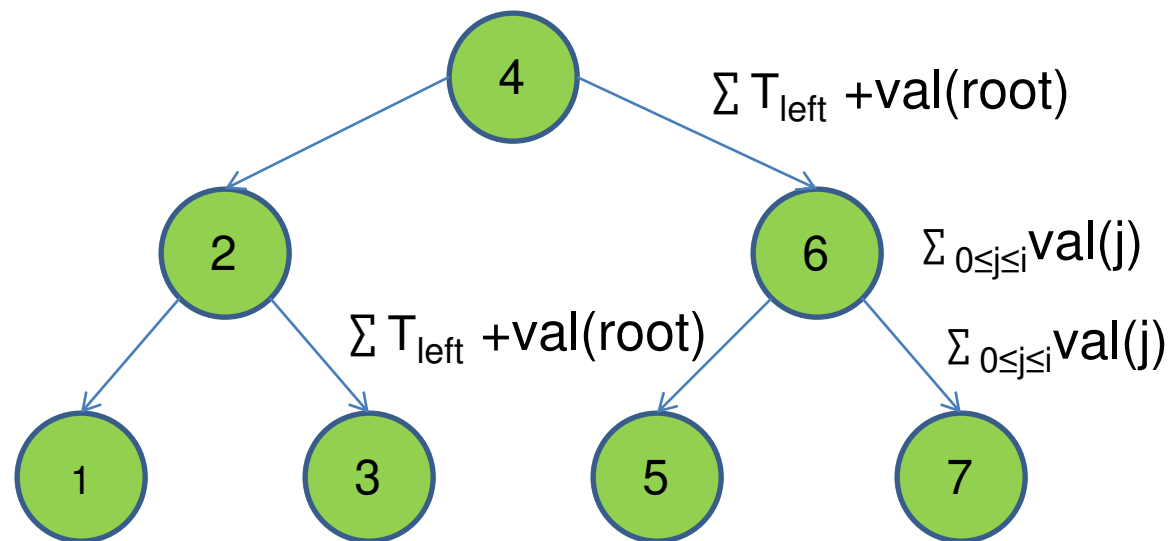
Scan/Exscan: Two phases, up and down

Up-phase: Interior node receives from left subtree, adds to own value, stores partial result, receives from right subtree, computes temporary partial result and sends upwards



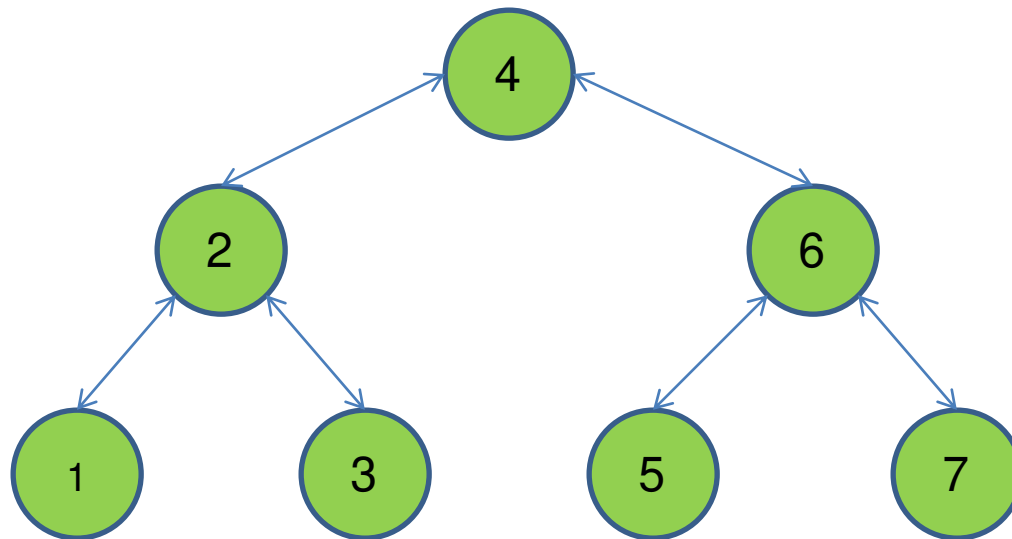
Scan/Exscan: Two phases, up and down

Down-phase: Interior node receives result from parent, sends to left subtree, adds to own partial result, sends complete result to right subtree



Scan/Exscan: Two phases, up and down

Both phases can be pipelined, best time is twice the best time for reduction (some extra overlap of up- and down phase possible with bidirectional communication)



Peter Sanders, Jesper Larsson Träff: Parallel Prefix (Scan)  
Algorithms for MPI. PVM/MPI 2006: 49-57

## Problem with pipelined trees

Problem with pipelined binary trees: bidirectional communication only partially used (receive from parent, send to one child)

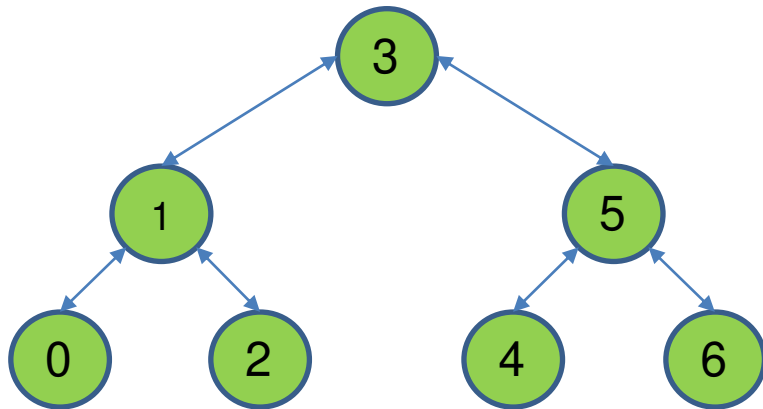
Idea: For operations consisting of both up and down phases, the two phases can be overlapped; bidirectional communication in each step

Example: Allreduce, Scan/Exscan

Doubly-rooted, doubly pipelined allreduce

Pipelined Reduce&Bcast, simultaneously

### Programming exercise



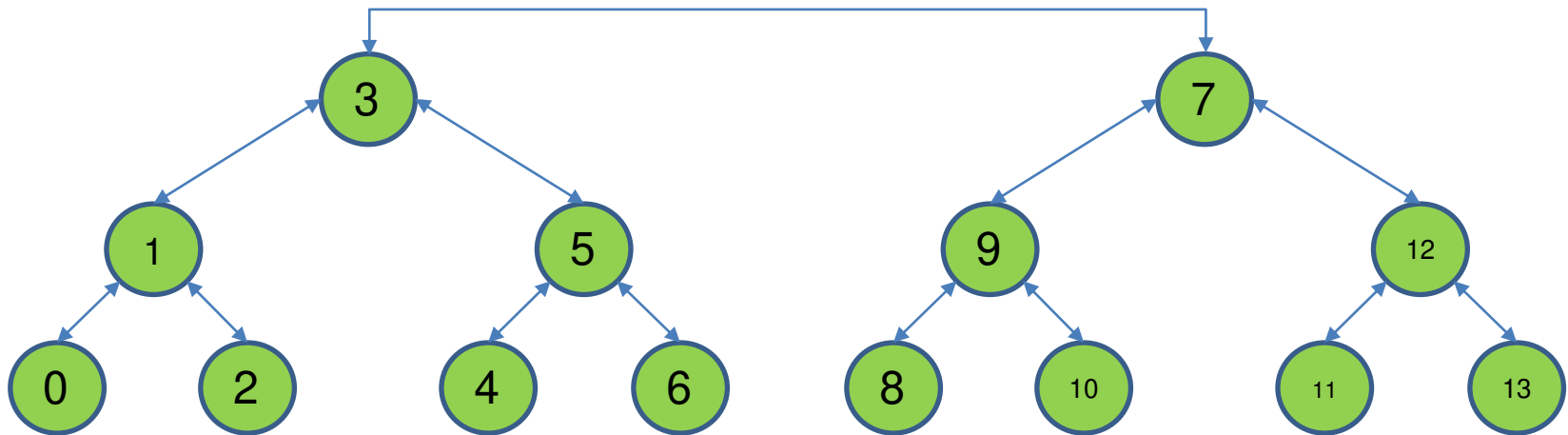
Per round, 3 steps:

1. Sendreceive from first child, local reduce
2. Sendreceive from second child, local reduce
3. Sendreceive from parent

Leaves and root are special: Only Sendreceive with parent; only Sendreceive with children

Doubly-rooted, doubly pipelined allreduce

Pipelined Reduce&Bcast, simultaneously  
With two connected, rooted trees



Leaves are special: Only Sendreceive with parent. Root in one tree communicates with root in other trees (and needs extra reduction)



Doubly-rooted, doubly pipelined allreduce

Pipelined Reduce&Bcast, simultaneously  
With two connected, rooted trees

Implementation detail: After how many rounds will a process at depth  $d$  receive a block from parent? What is the running time of the algorithm with a best possible pipeline block size?

Bonus programming exercise

Jesper Larsson Träff: A Doubly-pipelined, Dual-root Reduction-to-all Algorithm and Implementation. CoRR abs/2109.12626 (2021)

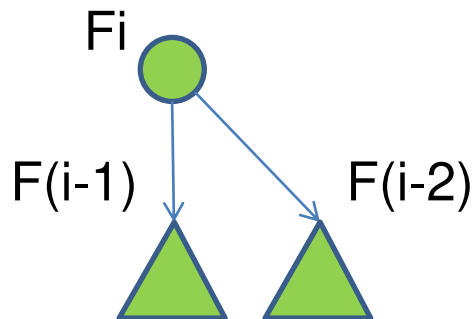
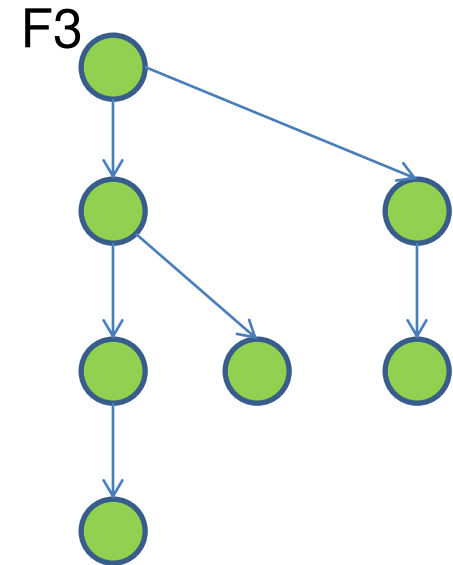
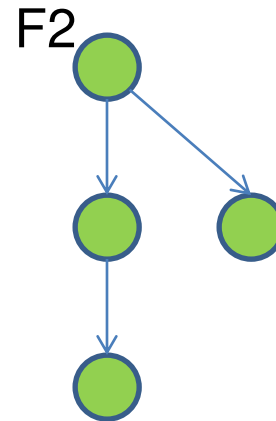
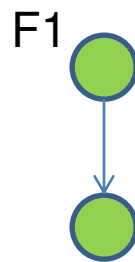
## Problem with balanced binary tree

Broadcast in  $T_i$ :

Last leaf becomes data after  $2i$  rounds (latency  $k$  in pipelining lemma)

Possible solution: Imbalanced binary tree, left subtree deeper than right subtree

## An imbalanced binary tree: Fibonacci tree



Properties:

- $F_i$  has  $\text{Fib}(i+3)-1$  nodes
- $F_i$  has depth  $i$ ,  $i \geq 0$ ,
- $F_i$  has  $i+1$  levels

Lemma:  $F_i = \text{Fib}(i+3) - 1$  where  $F_i$  is the number of nodes in  $i$ 'th Fibonacci tree

Proof: Recall that by definition  $\text{Fib}(0) = 0$ ,  $\text{Fib}(1) = 1$ , and  $\text{Fib}(i) = \text{Fib}(i-1) + \text{Fib}(i-2)$  for  $i \geq 2$ . By definition  $F_i = 1 + F(i-1) + F(i-2)$

i	0	1	2	3	4	5	6	7	8	9	10
Fib(i)	0	1	1	2	3	5	8	13	21	34	55
$F_i$	1	2	4	7	12	20	33	54	...		

Claim follows by induction. Base:  $F_0 = 1 = \text{Fib}(3) - 1$ ,  $F_1 = 2 = \text{Fib}(4) - 1$ . Assume claim holds for  $i-1$ , .... Then  $F_i = 1 + F(i-1) + F(i-2) = 1 + \text{Fib}(i+2) - 1 + \text{Fib}(i+1) - 1 = \text{Fib}(i+2) + \text{Fib}(i+1) - 1 = \text{Fib}(i+3) - 1$

Recall: For the  $i$ 'th Fibonacci number  $\text{Fib}(i) = 1/\sqrt{5}(\phi^i - \phi'^i)$  where  $\phi = (1+\sqrt{5})/2$  and  $\phi' = (1-\sqrt{5})/2$

Exercise: Proof by induction

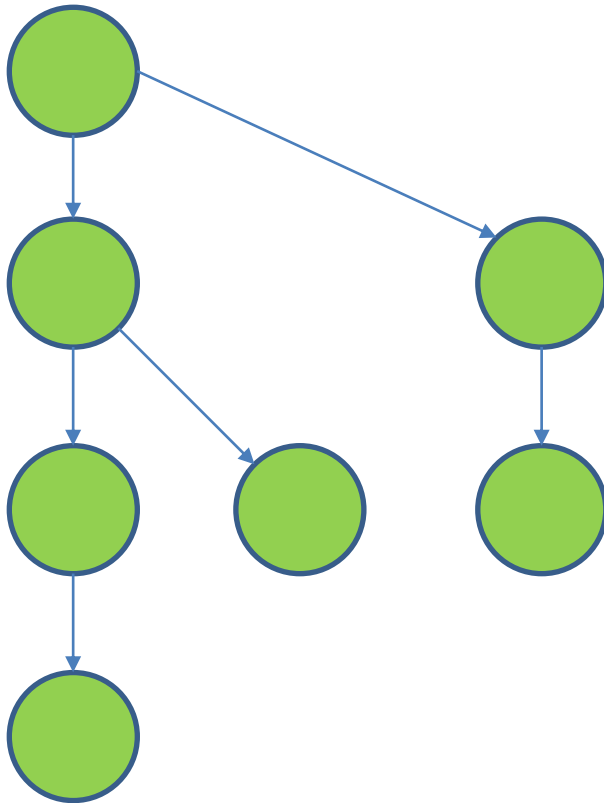
Hint: Can be found explicitly using generating functions (see Wilf, Knuth, and others: powerful technique)

$\text{Fib}(i+2)-1 \geq p$  if  $1/\sqrt{5} \phi^{i+2} \geq p \Leftrightarrow i+2 \geq \log_{\phi} p \Leftrightarrow i \geq \log_{\phi} p - 2$

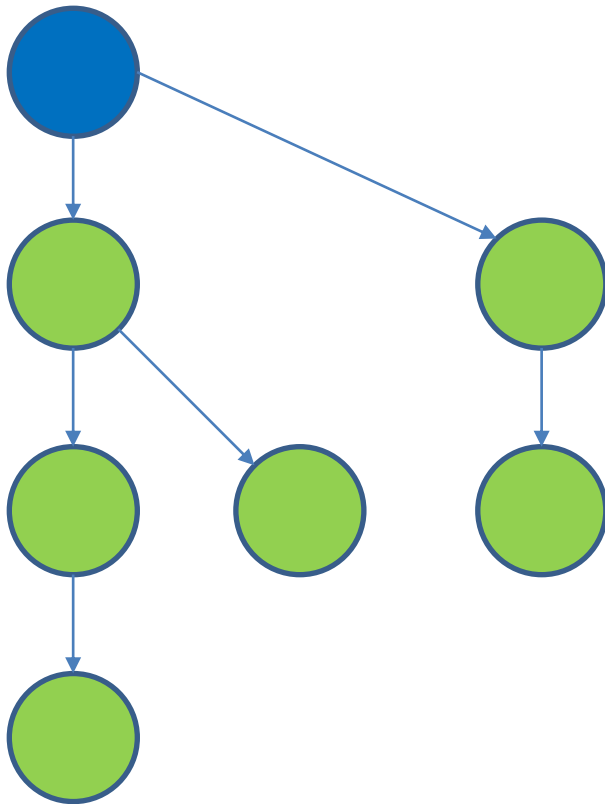
Fibonacci trees introduced explicitly for broadcast in

Jehoshua Bruck, Robert Cypher, Ching-Tien Ho: Multiple Message Broadcasting with Generalized Fibonacci Trees. SPDP 1992: 424-431

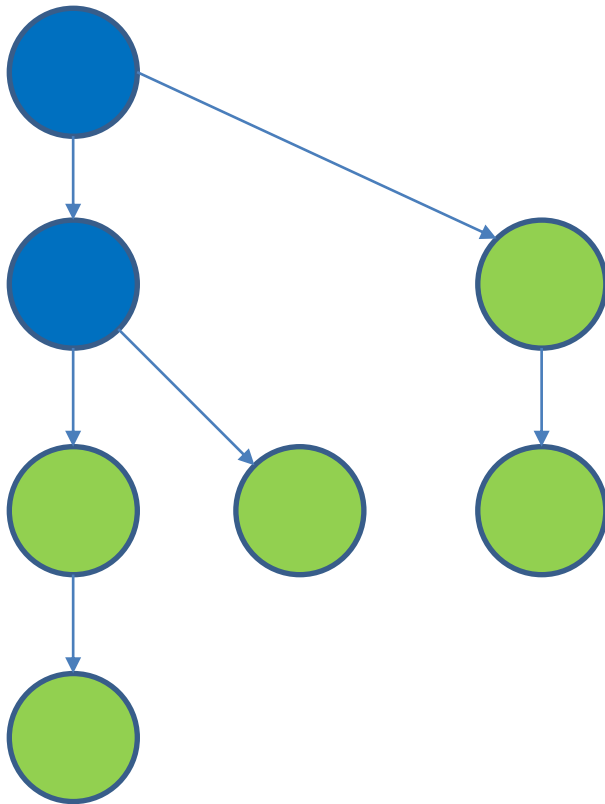
## Broadcast in Fibonacci tree



## Broadcast in Fibonacci tree

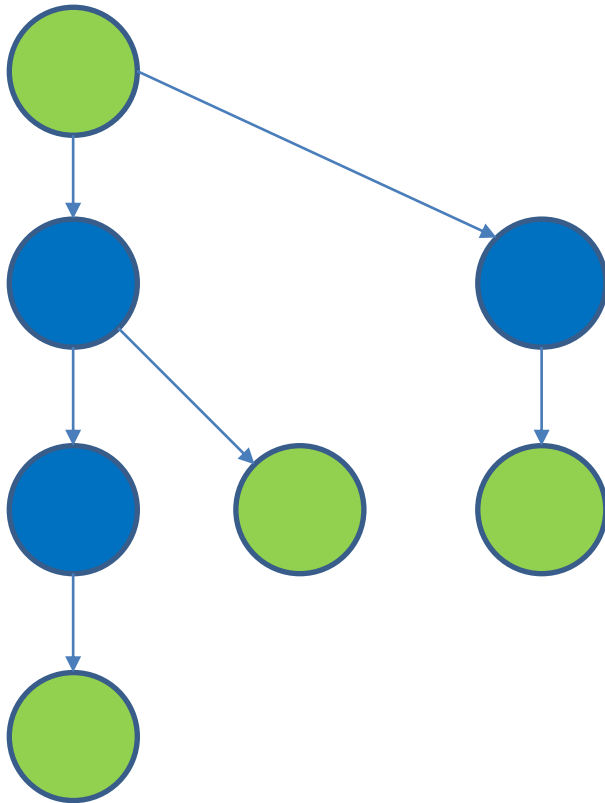


## Broadcast in Fibonacci tree

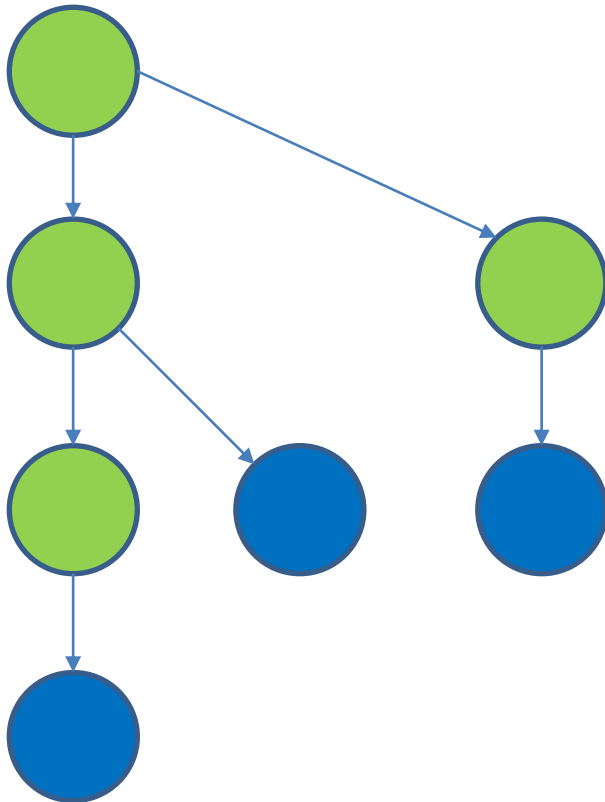




## Broadcast in Fibonacci tree



## Broadcast in Fibonacci tree



- All leaves become data at the same time
- Every node becomes a new block every second round

## Broadcast (reduction) trees: Properties

- Binomial tree  $B_i$ : All leaf nodes receive data after  $i$  rounds
- Fibonacci tree  $F_i$ : All leaf nodes receive data after  $i$  rounds
- Binary tree  $T_i$ : First leaf receives data after  $i$  rounds, last leaf after  $2i$  rounds

Exercise : Prove by induction

For small data: Binomial tree is round optimal, Fibonacci almost round optimal, binary tree factor 2 off

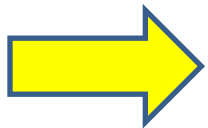
Binary tree and Fibonacci tree can be pipelined (binomial tree not)

Fixed-degree trees can be pipelined; all nodes (except root and leaves) have the same amount of work

## Two (pipelined) binary trees

Binary tree **drawbacks**:

- Nodes receive a new block only every second round
- Leaves only receiving



Capabilities of communication model not fully exploited

**Idea** :

Use two binary trees instead of one. Interior node of one tree is leaf of other, vice versa; in each round processors receive a new block from parent in either tree, send a previous block to one of its children

Could perhaps work for incomplete binary trees with even number of nodes?

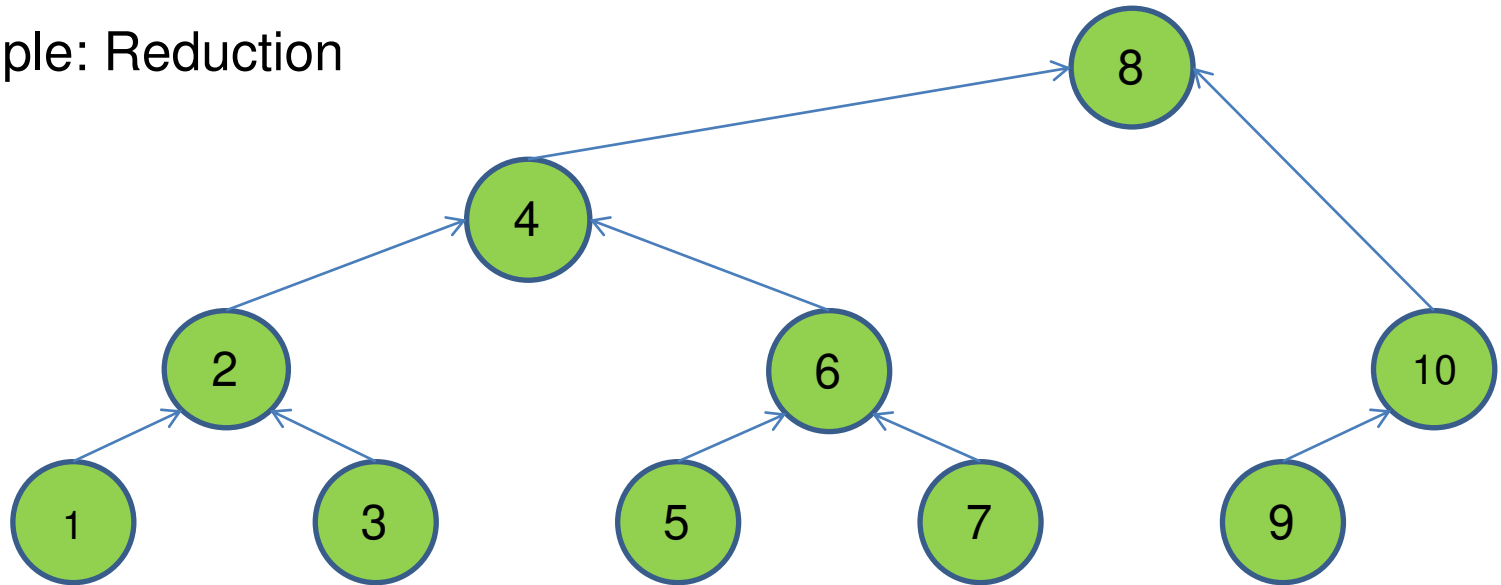
### Idea :

Use two binary trees instead of one. Interior node of one tree is leaf of other, vice versa; in each round processors receive a new block from parent in either tree, send a previous block to one of its children

Each processor associated with two nodes: leaf in one tree, interior node in other tree

Peter Sanders, Jochen Speck, Jesper Larsson Träff: Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing* 35(12): 581-594 (2009)

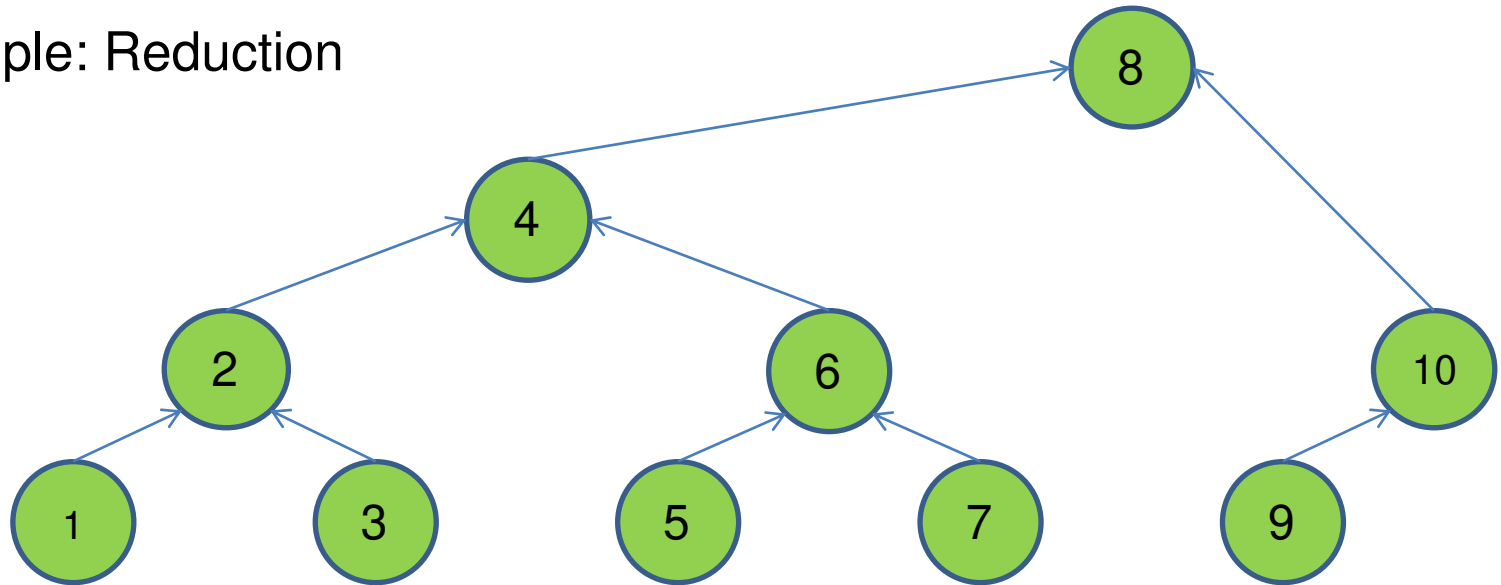
## Example: Reduction



Incomplete binary tree with even number of nodes: Same number of interior nodes and leaf nodes.

Tree in in-order numbering, processors numbered from 1, ..., p

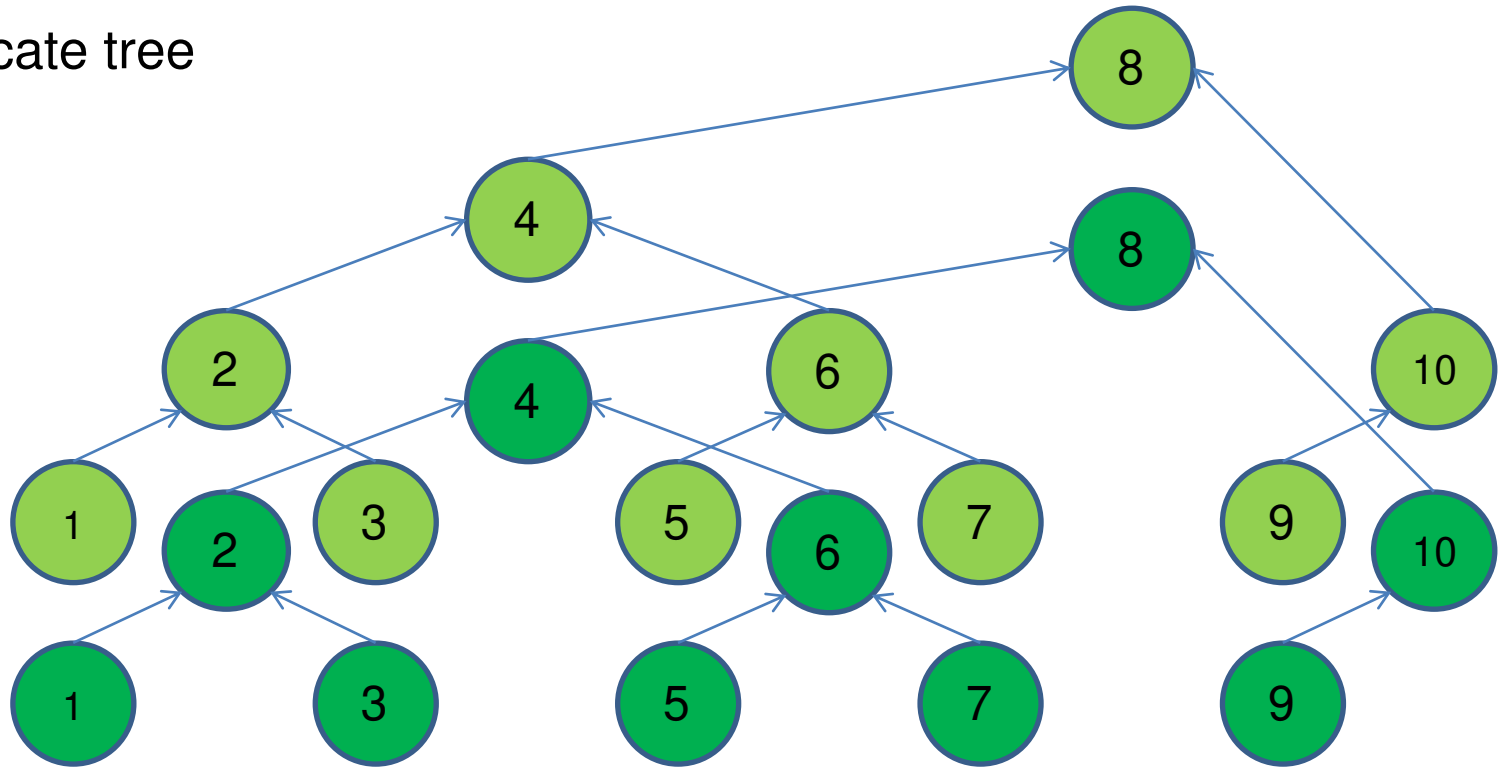
## Example: Reduction



Incomplete binary tree with even number of nodes: Same number of interior nodes and leaf nodes.

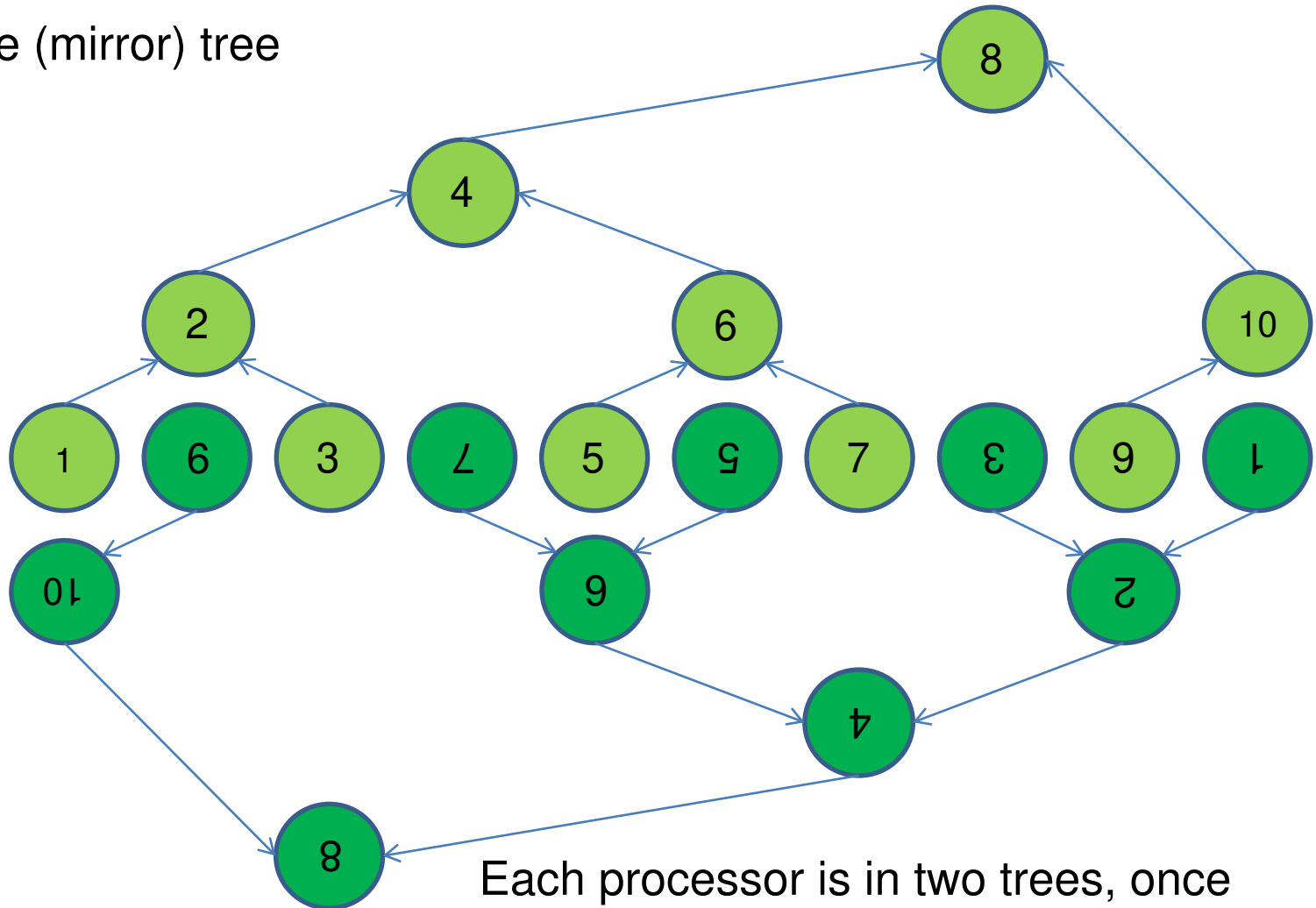
Construction: Given  $p$ , remove processor 0 (root), construct in-order tree over remaining processors, if  $p-1$  is even. (If  $p-1$  is odd, remove one more processor, add as virtual root)

## Duplicate tree





Rotate (mirror) tree



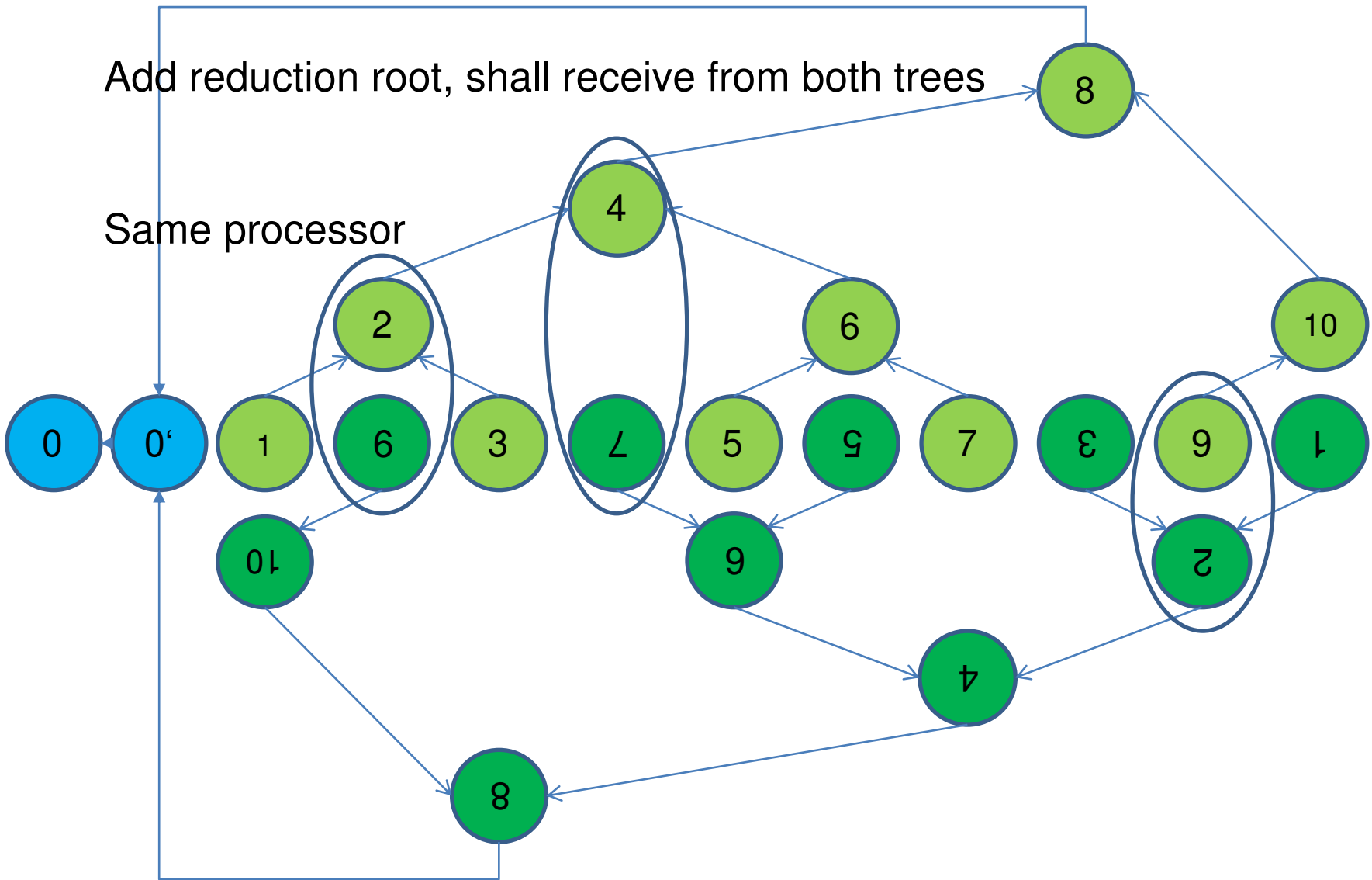
Each processor is in two trees, once with rank, once with rank' = p-rank





Add reduction root, shall receive from both trees

Same processor



Both trees can work simultaneously,  $m/2$  data reduction in either by pipelining. Pipelining lemma with  $s=2$  and  $k=2(\log(p+1)-1)$  gives

$$T_{\text{reduce}}(m) = (2\lceil\log((p+1)/4)\rceil)\alpha + 2\sqrt{4(\lceil\log((p+1)/4)\rceil)\alpha\beta m/2} + \frac{2\beta m}{2} = (2\lceil\log((p+1)/4)\rceil)\alpha + 2\sqrt{2(\lceil\log((p+1)/4)\rceil)\alpha\beta m} + \beta m$$

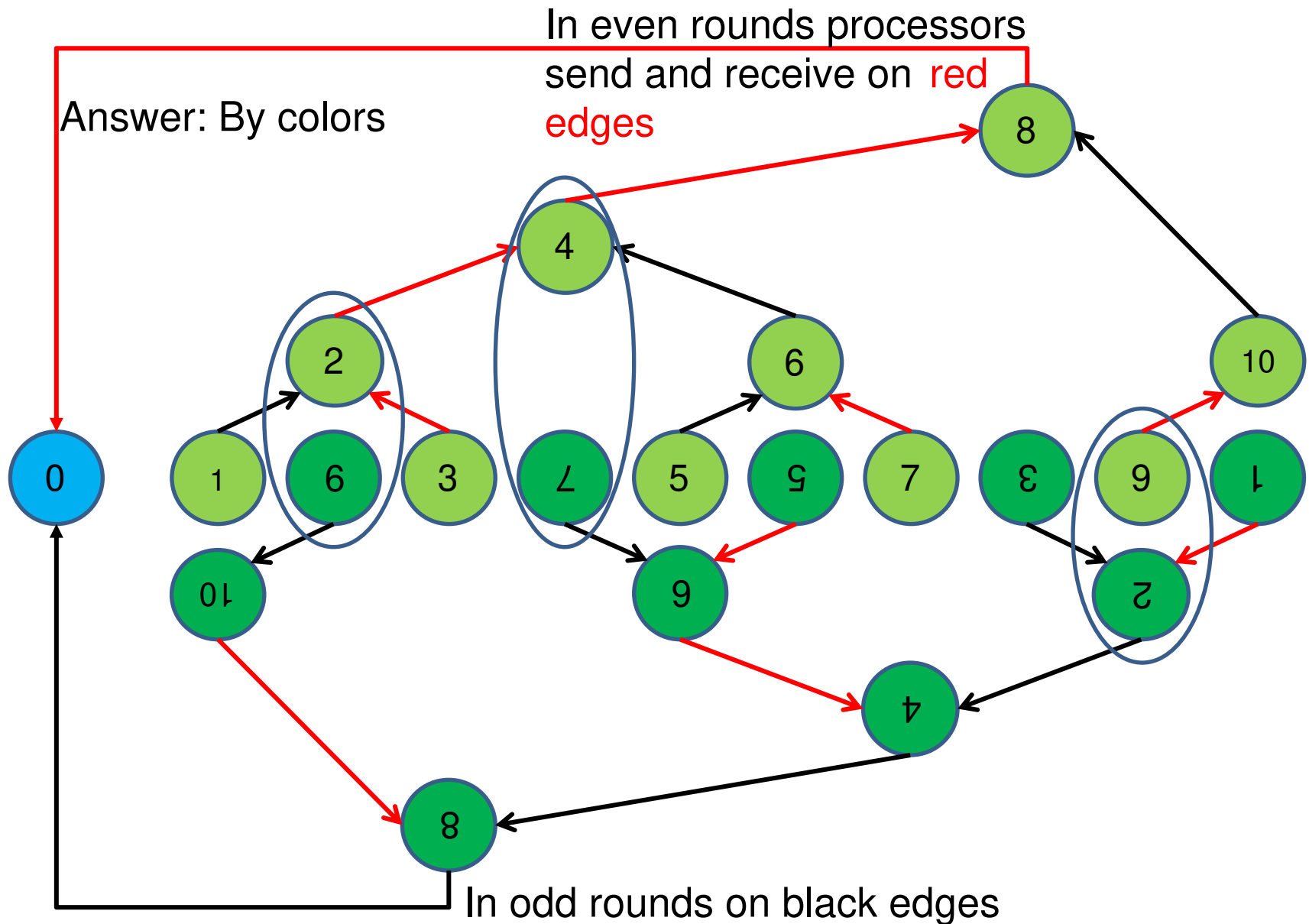
Optimal in  $\beta$ -term, logarithmic in  $\alpha$ -term

**Remark:** Currently best known reduction time with logarithmic latency

Care needed not to exploit commutativity

**Problem:**

How to schedule communication such that in each round each processor sends to a parent (in either tree) and receives from a child?



Coloring (2-tree scheduling) lemma:

There is an edge 2-coloring of the 2-tree such that for each node

- The colors of the edges to parents in upper and lower trees are different
- The colors from the (up to two) children are different

Proof:

Construct **schedule graph B** as follows: Let  $\{s_1, \dots, s_p\}$  be a set of sending,  $\{r_1, \dots, r_p\}$  a set of receiving processor nodes; there is an edge between  $s_i$  and  $r_j$  iff  $s_i$  is a child of  $r_j$  in either upper or lower tree.

Graph B is clearly bipartite and each node  $s_i$  or  $r_j$  has degree at most two. B is therefore edge 2-colorable.

**Note** (recall?):

A bipartite edge 2-coloring can be computed in  $O(n+m)$  time  
(Cole, Ost, Schirra, 2001)

Richard Cole, Kirstin Ost, Stefan Schirra: Edge-Coloring  
Bipartite Multigraphs in  $O(E \log D)$  Time. *Combinatorica* 21(1): 5-  
12 (2001)

Too expensive (algorithmic latency), not parallel



Main theorem:

Using the mirroring construction for 2-trees, the color of the edge to the parent of an interior tree node  $v$ ,  $1 \leq v \leq p$ , in the upper tree is computed in  $O(\log p)$  steps by calling  $\text{EdgeColor}(p, \text{root}, v, 1)$ :

```
EdgeColor(p, root, v, H) {
    if (v==root) return 1;
    while ((v & H)==0) H <<= 1;
    u = ((v & (H<<1))!=0 || v+H>p) ? v-H : v+H;

    c = (u>v) ? 1 : 0;
    return EdgeColor(p, root, u, H) ^ (p/2 mod 2) ^ c;
}
```

 Find parent

Proof: In paper (room for improvement)

$\text{EdgeColor}(10,8,2,1) = \text{EdgeColor}(10,8,4,2) = \text{EdgeColor}(10,8,8,4) = 1$

$\text{EdgeColor}(10,8,4,1) = \text{EdgeColor}(10,8,8,4) = 1$

$\text{EdgeColor}(10,8,6,1) = \text{EdgeColor}(10,8,4,2) \text{ XOR } 1 =$   
 $\text{EdgeColor}(10,8,8,4) \text{ XOR } 1 = 0$

$\text{EdgeColor}(10,8,8,1) = 1$

$\text{EdgeColor}(10,8,10,1) = \text{EdgeColor}(10,8,8,2) \text{ XOR } 1 = 0$

```
EdgeColor(p, root, v, H) {
    if (v==root) return 1;
    while ((v & H)==0) H <<= 1;
    u = ((v & (H<<1))!=0 || v+H>p) ? v-H : v+H;
    c = (u>v) ? 1 : 0;
    return EdgeColor(p, root, u, H) ^ (p/2 mod 2) ^ c;
}
```

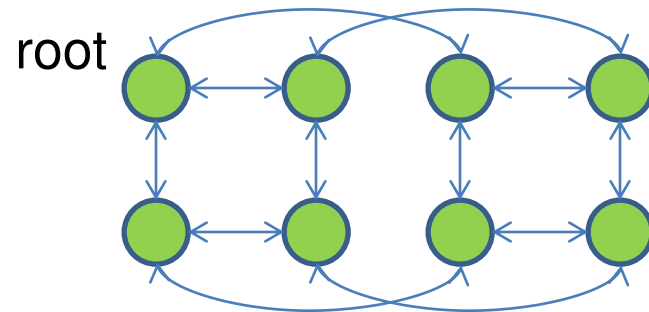
**Final remark** : Latency can be improved by using two Fibonacci trees

Details ... (Peter Sanders and students, personal communication)

## The power of the hypercube: Optimal Broadcast

**Goal:** Broadcast  $M$  blocks in  $M-1+\text{ceil}(\log p)$  rounds

**Idea :** Use allgather like algorithm on hypercube (or circulant graph), each processor broadcasts some of the blocks



Blocks  $\text{buffer}[i]$ ,  $0 \leq i < M$  to be broadcast

Bin Jia: Process cooperation in multiple message broadcast.  
Parallel Computing 35(12): 572-580 (2009)

## The $M-1+\log p$ round hypercube algorithm

```

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

for (j=0; j<M-1+q; j++) {
    MPI_Sendrecv(buffer[s(rank, j)], ...,
                  partner(rank, j), ...,
                  buffer[t(rank, j)], ...,
                  partner(rank, j), ..., comm);
}

```

Partner in  $j$ 'th round is  $(j \bmod q)$ 'th hypercube neighbor:

$$\text{partner}(\text{rank}, j) = \text{rank} \text{ XOR } (2^{j \bmod q})$$

where for now  $q = \log_2 p$

The algorithm is correct if

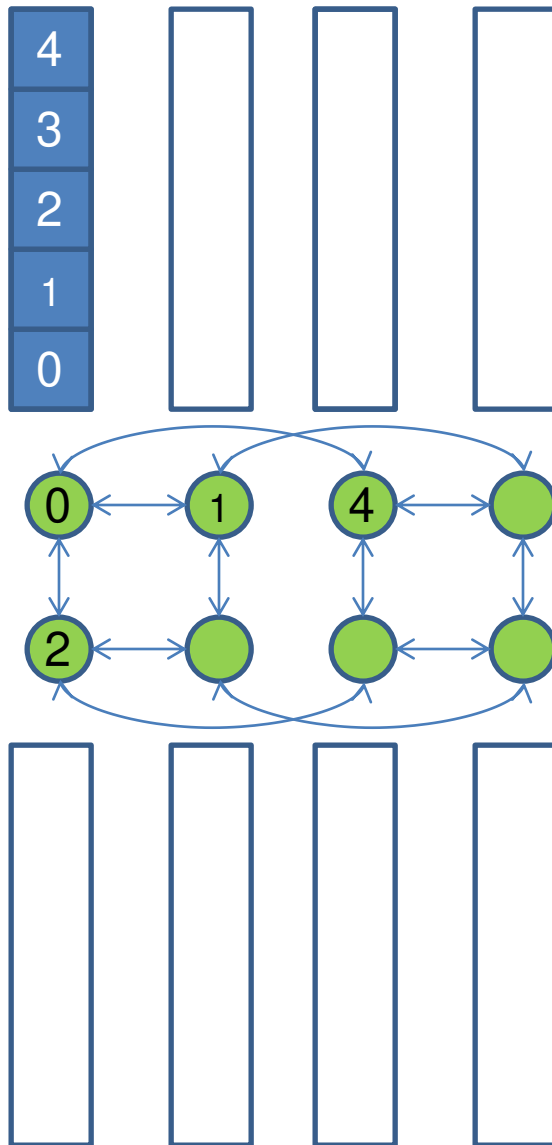
1. The block  $s(\text{rank}, j)$  that processor rank sends in round  $j$  has been received in a previous round  $j' < j$
2. The block  $s(\text{rank}, j)$  that a processor sends in round  $j$  is the block that its partner expects to receive in round  $j$ ,  $t(\text{partner}(\text{rank}, j), j)$
3. The block  $t(\text{rank}, j)$  that a processor receives in round  $j$  is the block that its partner expects sends in round  $j$ ,  $s(\text{partner}(\text{rank}, j), j)$
4. All blocks have been received after  $M-1 + \log_2 p$  rounds by each processor

From now on (wlog)  $\text{root} = 0$  (otherwise: shift towards 0)

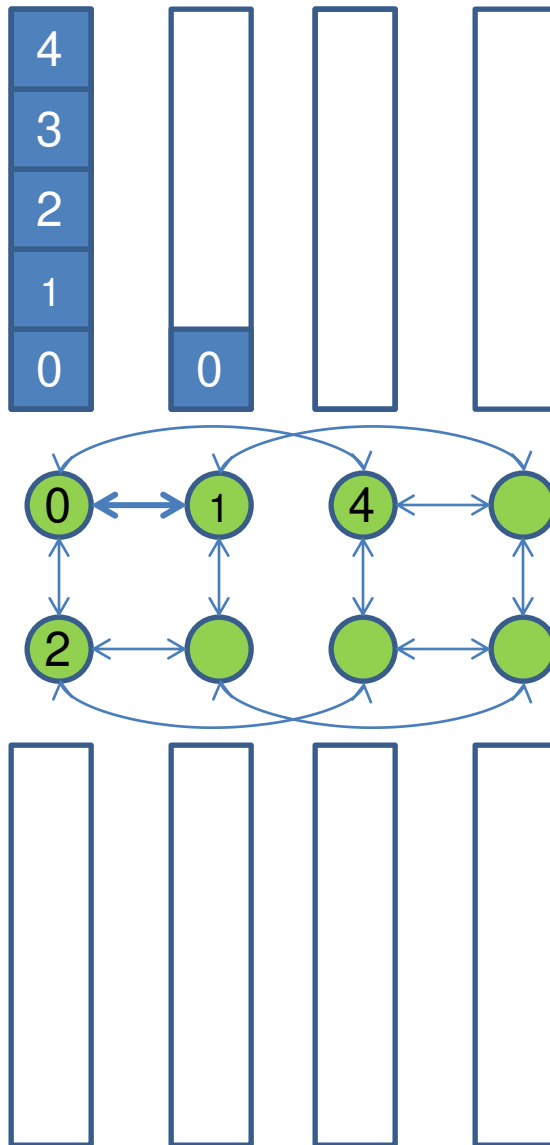
Root sends blocks out one after the other to  $\text{partner}(0,j)$ :

$$s(0,j) = \min(j, M-1)$$

The root never receives; if  $\text{partner}(i,j) = 0$  for processor  $i$  in round  $j$ , processor  $i$  does not send (in MPI: Send to `MPI_PROC_NULL`)

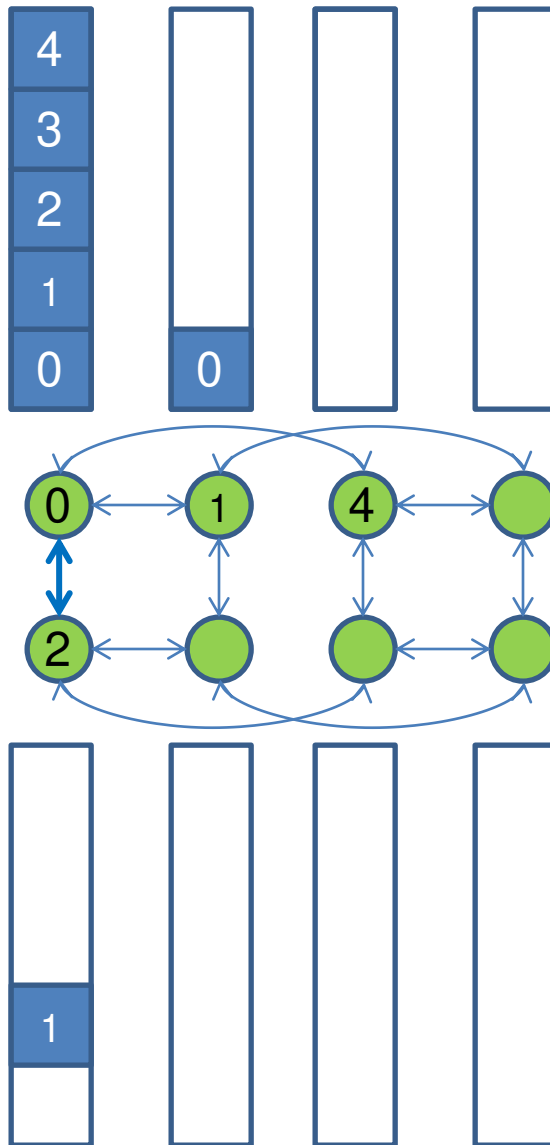






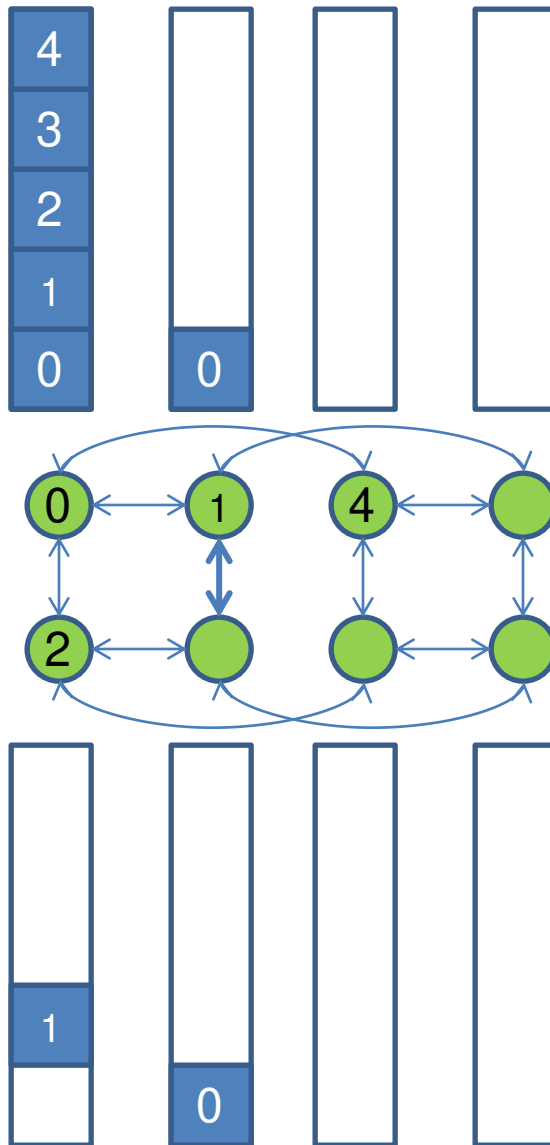
Round 0

Root sends to partner 0:  
flip bit 0



Round 1

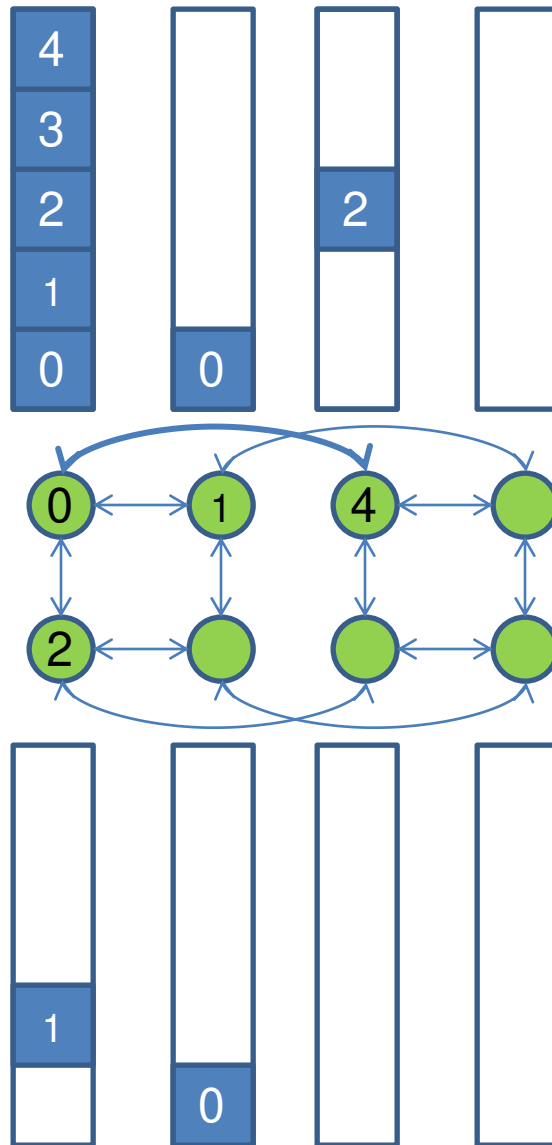
Root sends to partner 1:  
flip bit 1



Round 1

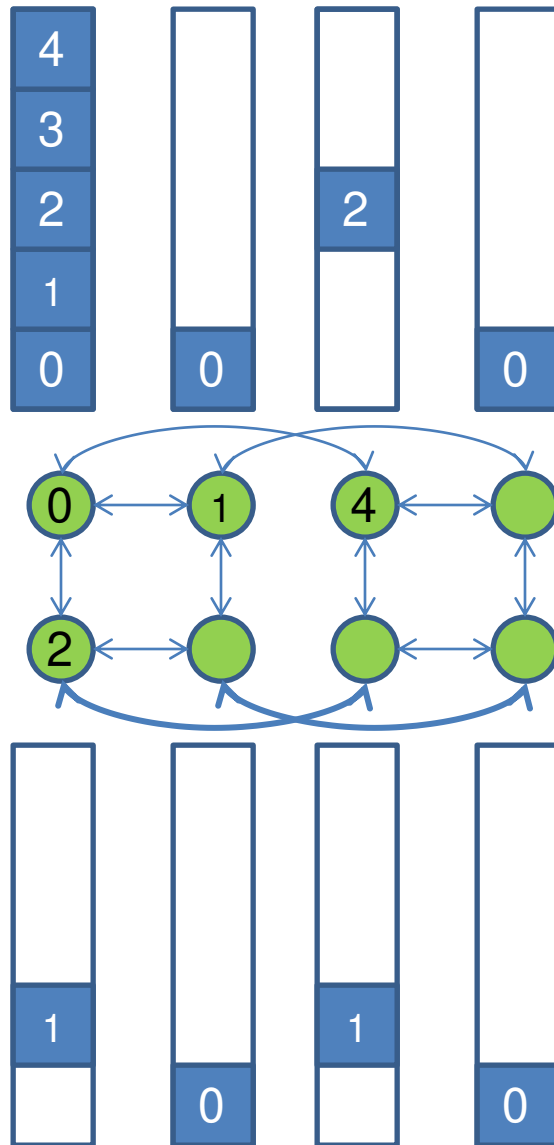
Root sends to partner 1:  
flip bit 1

Processor 1 sends to  
partner 1: flip bit 1



Round 2

Root sends to partner 2:  
flip bit 2



Round 2

Root sends to partner 2:  
flip bit 2

Processors 1,2,3 sends to  
partners: flip bit 2

Set  $q = \log_2 p$ , and define

$\text{bit}(i,j)$ :  $(j \bmod q)$ 'th bit of  $i$  (from least significant bit)

$$s(i,j) = j - q + (1 - \text{bit}(i,j)) \text{NEXTBIT}(i)[j \bmod q]$$

$$t(i,j) = j - q + \text{bit}(i,j) \text{NEXTBIT}(i)[j \bmod q]$$

Observation (by definition of partner):

$$s(i,j) = t(\text{partner}(i,j), j)$$

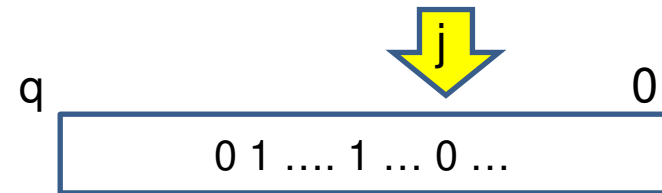
$$t(i,j) = s(\text{partner}(i,j), j)$$

**Remedy**: if  $s(i,j) \geq M$ , send instead block  $M-1$ . If  $t(i,j) \geq M$ , receive instead block  $M-1$ .

**Also**: Blocks with  $s(i,j) < 0$  or  $t(i,j) < 0$  are not sent/received

$\text{NEXTBIT}(i)$  is a precomputed function for each processor  $i$  with

$\text{NEXTBIT}(i)[j]$ : distance from bit  $j$  of  $i$  to next left (more significant) 1-bit in  $i$ , with wrap-around (for technical convenience  $\text{NEXTBIT}(0)[j] = q$ )



**Observation** :  $\text{NEXTBIT}(i)[j] = \text{NEXTBIT}(\text{partner}(i,j))[j]$

Examples:

$\text{NEXTBIT}(010011)[0] = 1$

$\text{NEXTBIT}(010011)[1] = 3$

$\text{NEXTBIT}(010011)[2] = 2$

$\text{NEXTBIT}(010011)[3] = 1$

$\text{NEXTBIT}(010011)[4] = 2$

$\text{NEXTBIT}(010011)[5] = 1$

Lemma:

For any  $i$ ,  $0 \leq i < p$ ,  $\text{NEXTBIT}(i)$  can be computed in  $O(\log p)$  steps

Proof: **Exercise**

Proposition:

$$s(i,j+1) = (1-\text{bit}(i,j))s(i,j) + \text{bit}(i,j)t(i,j)$$

The block that processor  $i$  sends in round  $j+1$  is either the same block as sent in round  $j$ , or the block received in the previous round  $j$ .

A processor therefore does not attempt to send a block that it has not received!



Proposition:

$$s(i,j+1) = (1-\text{bit}(i,j))s(i,j) + \text{bit}(i,j)t(i,j)$$

## Proof:

First note that if  $\text{bit}(i,j+1)=0$ , then

$\text{NEXTBIT}(i)[(j+1) \bmod q] = \text{NEXTBIT}(i)[j \bmod q]-1$ ; and

if  $\text{bit}(i,j+1)=1$ , then  $\text{NEXTBIT}(i)[j \bmod q] = 1$

It follows that

$$s(i,j+1) =$$

$$j + 1 - q + (1-\text{bit}(i,j+1)) \text{NEXTBIT}(i)[(j+1) \bmod q] =$$

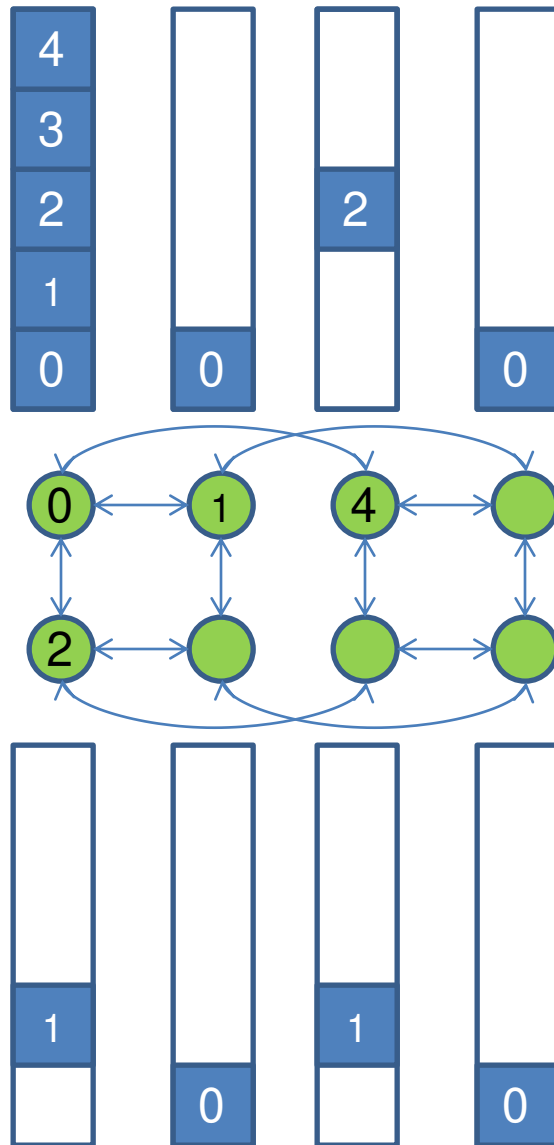
$$j - q + \text{NEXTBIT}(i)[j \bmod q] =$$

$$j - q + (1-\text{bit}(i,j)) \text{NEXTBIT}(i)[j \bmod q] +$$

$$\text{bit}(i,j)\text{NEXTBIT}(i)[j \bmod q] =$$

$$(1-\text{bit}(i,j)) s(i,j) + \text{bit}(i,j)t(i,j)$$

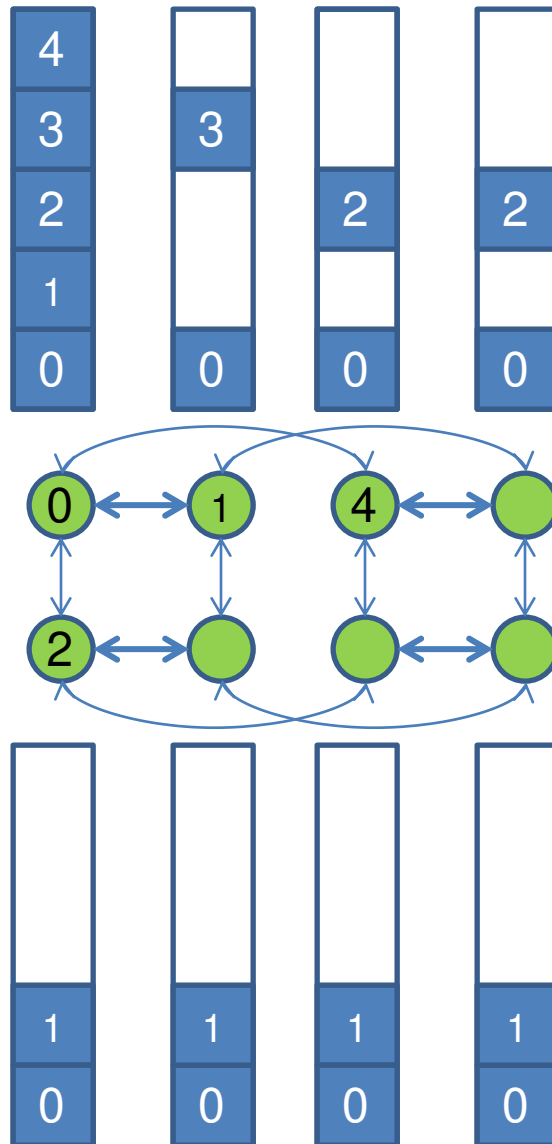
(definition)



Round 2

Root sends to partner 2:  
flip bit 2

Processors 1,2,3 sends to  
partners: flip bit 2



Round 3

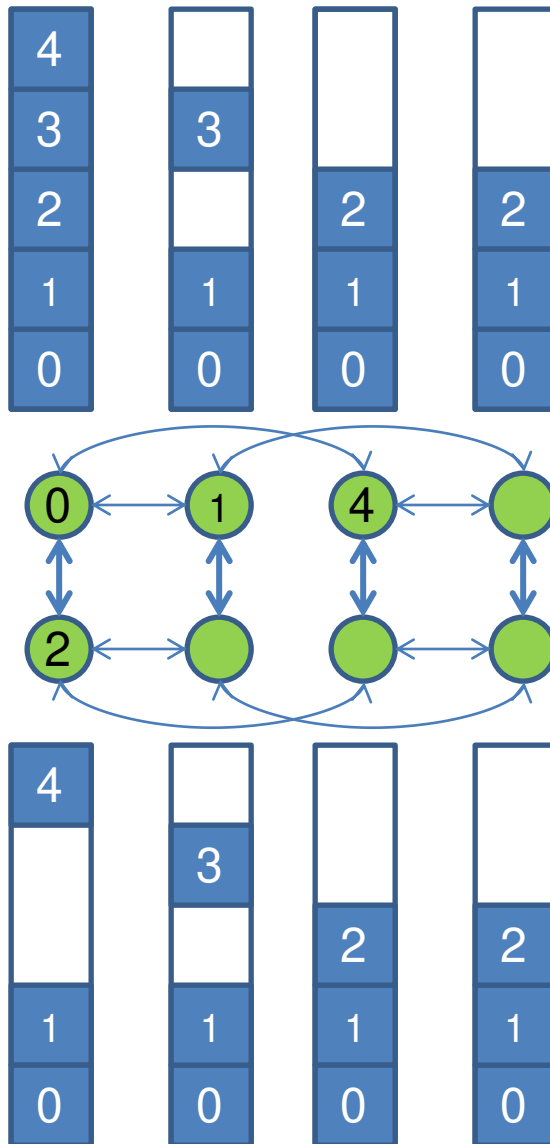
Flip bit  $(3 \bmod 3) = 0$

Processor 4:

$\text{NEXTBIT}(100)[0] = 2$ , so  
 $s(4,3) = 2$ ,  $t(4,3) = 0$

Processor 5:

$\text{NEXTBIT}(101)[0] = 2$ , so  
 $s(5,3) = 0$ ,  $t(5,3) = 2$



Round 4

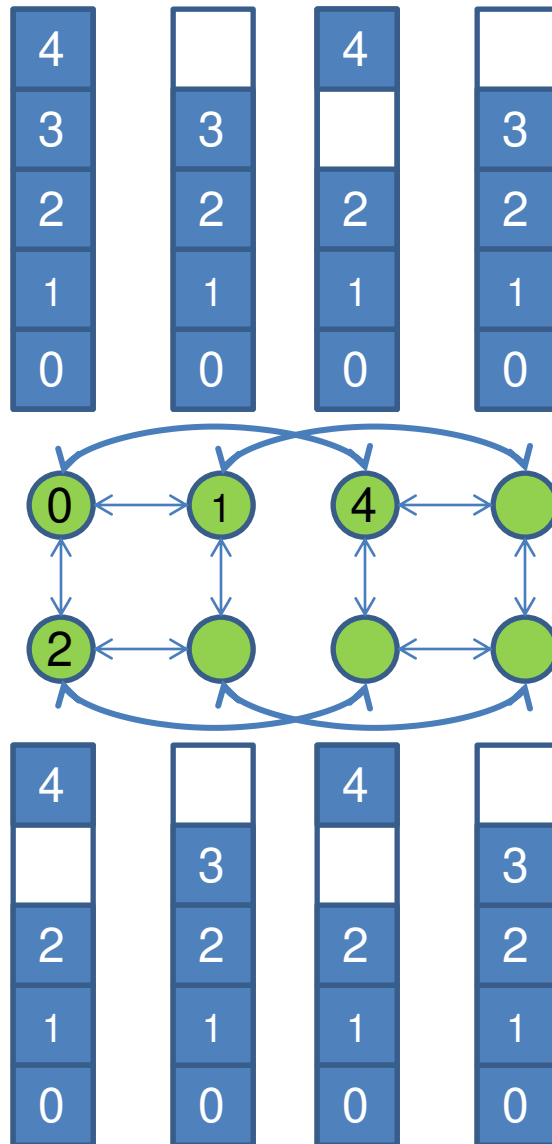
Flip bit  $(4 \bmod 3) = 1$

Processor 4:

$\text{NEXTBIT}(100)[1] = 1$ , so  
 $s(4,4) = 2$ ,  $t(4,4) = 1$

Processor 6:

$\text{NEXTBIT}(110)[1] = 1$ , so  
 $s(6,4) = 1$ ,  $t(6,4) = 2$



Round 5

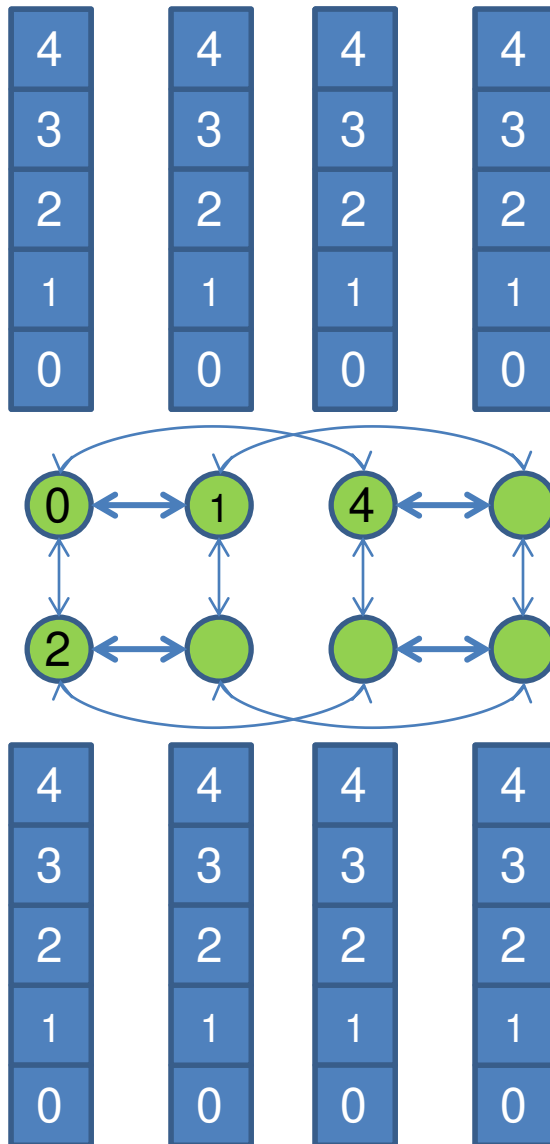
Flip bit  $(5 \bmod 3) = 2$

Processor 2:

$\text{NEXTBIT}(010)[2] = 2$ , so  
 $s(2,5) = 4$ ,  $t(2,5) = 2$

Processor 6:

$\text{NEXTBIT}(110)[2] = 2$ , so  
 $s(6,5) = 2$ ,  $t(6,5) = 4$



Round 6

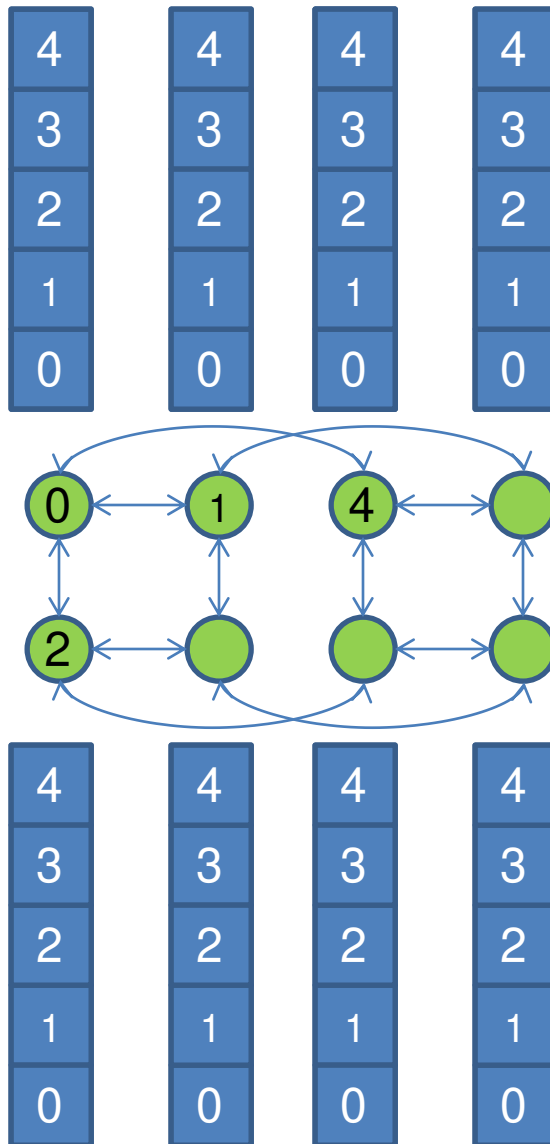
Flip bit  $(6 \bmod 3) = 0$

Processor 2:

$\text{NEXTBIT}(010)[0] = 1$ , so  
 $s(6,2) = 4$ ,  $t(6,2) = 3$

Processor 3:

$\text{NEXTBIT}(011)[0] = 1$ , so  
 $s(3,6) = 3$ ,  $t(3,6) = 4$



Round 6

Flip bit  $(5 \bmod 3) = 0$

Done!  $7 = 5+3-1$  rounds

Processor 7:

NEXTBIT(111)[0] = 1

$t(7,0) = -1$

$s(7,0) = -1$

NEXTBIT(111)[1] = 1

NEXTBIT(111)[2] = 1

$t(7,2) = 0$

NEXTBIT(111)[0] = 1

$t(7,3) = 1$

$s(7,3) = 0$

NEXTBIT(111)[1] = 1

$t(7,4) = 2$

$s(7,4) = 1$

NEXTBIT(111)[2] = 1

$t(7,5) = 3$

$s(7,5) = 2$

NEXTBIT(111)[0] = 1

$t(7,6) = 4$

$s(7,6) = 3$



Processor 5:

$\text{NEXTBIT}(101)[0] = 2$   
 $\text{NEXTBIT}(101)[1] = 1$   
 $\text{NEXTBIT}(101)[2] = 1$   
 $\text{NEXTBIT}(101)[0] = 2$   
 $\text{NEXTBIT}(101)[1] = 1$   
 $\text{NEXTBIT}(101)[2] = 1$   
 $\text{NEXTBIT}(101)[0] = 2$

$t(5,0) = -1$

$t(5,2) = 0$

$t(5,3) = 2$

$t(5,4) = 1$

$t(5,5) = 3$

$t(5,6) = 4$

$s(5,0) = -1$

$s(5,3) = 0$

$s(5,4) = 2$

$s(5,5) = 2$

$s(5,6) = 3$

Proposition: Suppose there is an infinite number of blocks. For round  $j \geq 0$ , define  $G(j,k) = \{i \mid \text{NEXTBIT}(i)[j \bmod q] = k\}$  for  $0 < k \leq q$ , and  $G(j,0) = \{i \mid 0 \leq i < p\}$ .

Claim: After round  $j$ , processors of  $G(j,k)$  have received all blocks  $j-q+k$ , for  $j-q+k \geq 0$ .

It follows that after round  $j = M-2+q$  (last round of algorithm), all processors (in  $G(j,0)$ ) have received blocks  $0, \dots, M-2$ ; half the processors in  $G(j,1)$  have received block  $M-1$ , and the other half have received a block after  $M-1$ . Since blocks after  $M-1$  are handled as block  $M-1$ , all processors have all blocks.



Half the processors have bit  $j+1=1$ , the other half bit  $j+1=0$

Proposition: Suppose there is an infinite number of blocks. For round  $j \geq 0$ , define  $G(j,k) = \{i \mid \text{NEXTBIT}(i)[j \bmod q] = k\}$  for  $0 < k \leq q$ , and  $G(j,0) = \{i \mid 0 \leq i < p\}$ .

Claim: After round  $j$ , processors of  $G(j,k)$  have received all blocks  $j-q+k$ , for  $j-q+k \geq 0$ .

Proof: Induction on the number of rounds  $j$

Base:

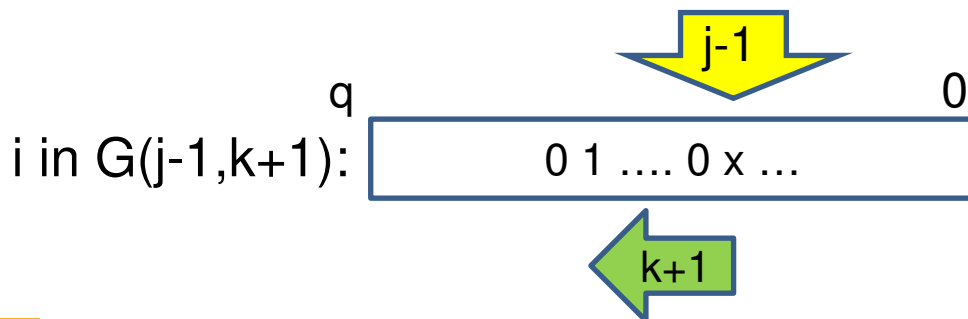
For  $j=0$ ,  $k=q$ . Since  $G(0,q) = \{0,1\}$  and processor 1 receives block  $0=j-q+k$  after round 0, the claim holds

Induction step: Assume claim holds for round  $j-1$ ; recall  $G(j,k) = \{i \mid \text{NEXTBIT}(i)[j \bmod q] = k\}$ . Case analysis on  $k$ .

Case  $0 \leq k < q$ :

In round  $j$  each processor  $i$  in  $G(j-1, k+1)$  sends block  $s(i,j) = j - q + (1 - \text{bit}(i,j)) \text{NEXTBIT}(i)[j \bmod q] = j - q + k$  (case analysis,  $k=0$  and  $k>0$ ) to  $\text{partner}(i,j)$ , and by induction hypothesis  $i$  has block  $j-1-q+k+1 = j-q+k$ .

Since  $G(j,k) = G(j-1, k+1) \cup \{i \mid \text{partner}(i,j) \text{ in } G(j-1, k+1)\}$ , all processors in  $G(j,k)$  have block  $j-q+k$  in round  $j$



Induction step: Assume claim holds for round  $j-1$ ; recall  $G(j,k) = \{i | \text{NEXTBIT}(i)[j \bmod q] = k\}$ . Case analysis on  $k$ .

Case  $k=q$ :

$G(j,q) = \{0, 2^{j \bmod q}\}$ , and processor 0 sends block  $j$  to processor  $2^{j \bmod q}$  in round  $j$

This concludes the proof of the claim, which shows that after  $M-1+q$  rounds of the algorithm (blocks larger than  $M-1$  are sent and received as  $M-1$ , blocks smaller than 0 are neither sent nor received), all processors have received all  $M$  blocks  $0, \dots, M-1$ .

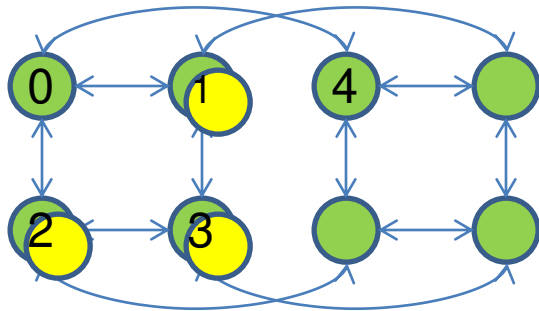
## Beyond hypercube

The hypercube result was known long before Bin Jia, similar to the ideas (“Edge Disjoint Spanning Trees”) in a fundamental paper by Johnsson and Ho (1989).

### Idea :

When  $p$  is not a power of two, let  $q = \text{floor}(\log_2 p)$ , pair each excess process  $i > 2^q$  with processor  $i - 2^q + 1$ , and use the hypercube algorithm on the processor pairs

S. Lennart Johnsson, Ching-Tien Ho: Optimum Broadcasting and Personalized Communication in Hypercubes. IEEE Trans. Computers 38(9): 1249-1268 (1989)



$$\text{co}(i) = \begin{cases} 0 & \text{for } i = 0 \\ 2^q - 1 + i & \text{for } 0 < i \leq p - 2^q \\ i & \text{for } p - 2^q < i < p \\ i - 2^q + q & \text{for } 2^q \leq i < p \end{cases}$$

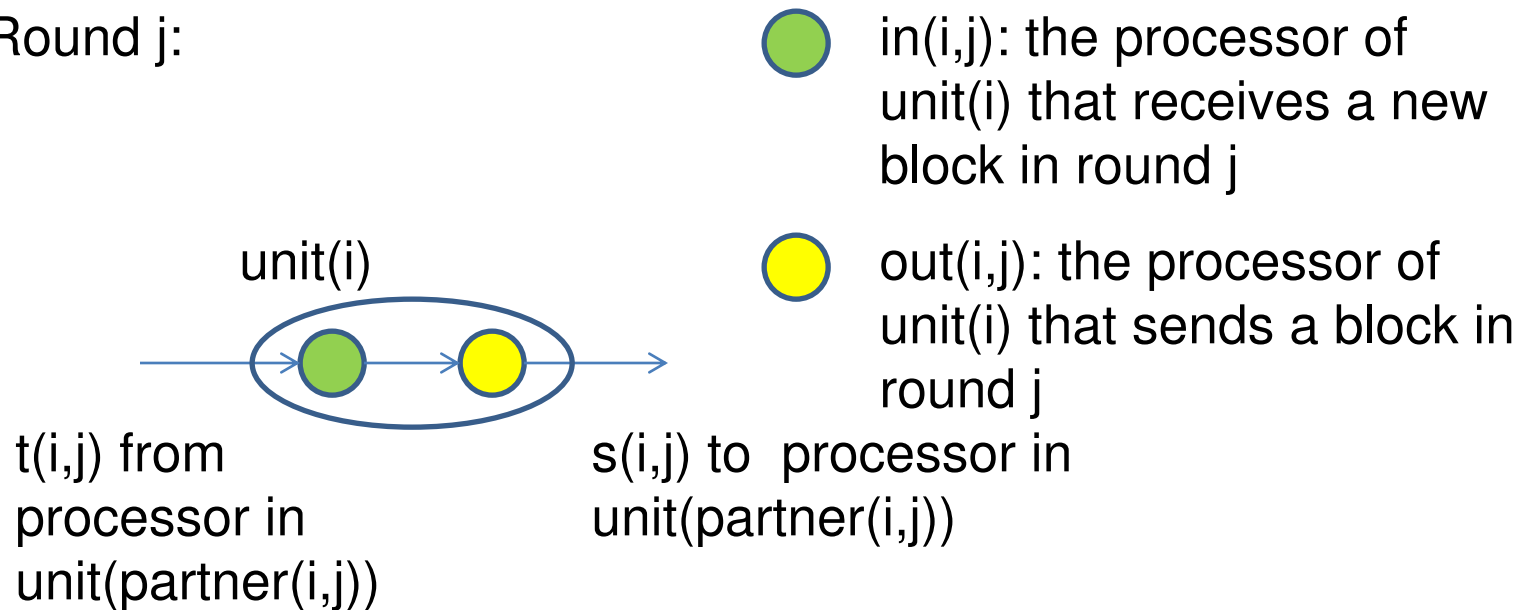
$$\text{rep}(i) = i \text{ for } i < 2^q, \text{ co}(i) \text{ otherwise}$$

$$\text{unit}(i) = \{i, \text{co}(i)\}$$

Each processor  $i$  has a  $\text{co}(i)$  processor (cooperating processor). Together  $i$  and  $\text{co}(i)$  play the role of one processor in the hypercube algorithm

Each pair has a representative  $\text{rep}(i)$

Round  $j$ :



Previous block sent from  $in(i,j)$  to  $out(i,j)$

**Note :**

$in(i,j)$  and  $out(i,j)$  may switch from round  $j$  to round  $j+1$



$$\text{in}(i,0) = \text{rep}(i)$$

$$\text{out}(i,0) = \text{co}(\text{rep}(i))$$

$$\text{in}(i,j) = \text{co}(\text{out}(i,j))$$

$$\text{out}(i,j+1) = (1 - \text{bit}(\text{rep}(i),j)) \text{out}(i,j) + \text{bit}(\text{rep}(i),j) \text{in}(i,j)$$

### Rationale:

$\text{out}(i,j)$  sends block  $s(\text{rep}(i),j)$  to  $\text{in}(\text{partner}(\text{rep}(i),j),j)$ , and  $\text{in}(i,j)$  receives block  $t(\text{rep}(i),j)$  in round  $j$ .

By the proposition  $s(\text{rep}(i),j+1)$  is  $s(\text{rep}(i),j)$  if  $\text{bit}(\text{rep}(i),j)=0$ , thus processor  $\text{out}(i,j)$  can continue as  $\text{out}(i,j+1)$ .

If  $\text{bit}(\text{rep}(i),j)=1$ ,  $s(\text{rep}(i),j+1) = t(\text{rep}(i),j)$ , which was received by  $\text{in}(i,j)$ , thus  $\text{out}(i,j+1)$  **shall switch** to  $\text{in}(i,j)$

$$\begin{aligned} \text{in}(i,0) &= \text{rep}(i) \\ \text{out}(i,0) &= \text{co}(\text{rep}(i)) \end{aligned}$$

$$\text{in}(i,j) = \text{co}(\text{out}(i,j))$$

$$\text{out}(i,j+1) = (1 - \text{bit}(\text{rep}(i),j)) \text{out}(i,j) + \text{bit}(\text{rep}(i),j) \text{in}(i,j)$$

Thus, the role of each processor in  $\text{unit}(i)$  for round  $j+1$  can be computed in  $O(1)$  time from the role in round  $j$ .

It remains to determine the role of the processors in  $\text{unit}(\text{partner}(i,j))$

```

for (j=0; j<M-1+q; j++) {
    if (co(rank)==rank) { // singleton unit
        MPI_Sendrecv(buffer[s(rank,j)],...,
                     in(partner(rank,j),j),...,
                     buffer[t(rank,j)],...,
                     out(partner(rank,j),j),...,comm);
    } else if (rank==out(rank,j)) { // out processor
        MPI_Sendrecv(buffer[s(rep(rank),j)],...,
                     in(partner(rep(rank),j),j),...,
                     buffer[j-q-1],...,
                     in(rank,j),...,comm);
    } else { // in processor
        MPI_Sendrecv(buffer[j-q-1],...,
                     out(rank,j),...,
                     buffer[t(rep(rank),j)],
                     out(partner(rep(rank),j),j),...,comm);
    }
}

```

One extra round (for processor in units with two processors)

```
if (co(rank) != rank) { // non-trivial unit
    if (rank == out(rank, j)) { // out processor
        MPI_Sendrecv(buffer[M-1], ...,
                      co(rank), ...,
                      buffer[M-2], ...,
                      co(rank), ..., comm);
    } else { // in processor
        MPI_Sendrecv(buffer[M-2], ...,
                      co(rank), ...,
                      buffer[M-1],
                      co(rank), ..., comm);
    }
}
```

To determine role of processors in partner unit, it is necessary to compute the number of role switches in constant time. This is

$$u(i,j) = \text{SWITCH}(i)[q-1] (j-1)/q + \text{SWITCH}(i)[(j-1) \bmod q]$$

SWITCH(i) is a q-element array that stores for each bit position j of i, the number of 1-bits from 0 to j (included)

Lemma:

For any i,  $0 \leq i < p$ , SWITCH(i) can be computed in  $O(\log p)$  steps

Easy exercise

Recall  $u(i,j) = \text{SWITCH}(i)[q-1] (j-1)/q + \text{SWITCH}(i)[(j-1) \bmod q]$   
 Let  $u'(i,j) = u(i,j) \bmod 2$ :  $u'(i,j)$  is the parity of the number of role switches for process  $i$  up to round  $j$ . Then

$$\text{out}(i,j) = (1-u'(i,j)) \text{co}(\text{rep}(i)) + u'(i,j) \text{rep}(i)$$

Let  $v(i,j) = (u(i,j) + j/q) \bmod 2$ . Since  $\text{rep}(i)$  and  $\text{partner}(\text{rep}(i),j)$  differs by only one bit (for each  $j$ ), the partner roles can be computed as

$$\text{out}(\text{partner}(\text{rep}(i),j),j) = (1-v(i,j)) \text{co}(\text{partner}(\text{rep}(i),j)) + v(i,j) \text{partner}(\text{rep}(i),j)$$

Proposition:

Before round  $j$ , processor  $\text{in}(i,j)$  has received block  $j-q-1$  if  $j-q-1 \geq 0$ , and processor  $\text{out}(i,j)$  has received block  $s(\text{rep}(i),j)$  if  $s(\text{rep}(i),j) \geq 0$

Proof: Induction on  $j$ . In round 0,  $\text{in}(1,0)$  receives block 0 from processor 0, and  $\text{out}(1,1) = \text{in}(1,0)$ , and  $s(\text{rep}(i)) < 0$  for the other  $\text{rep}(i) \neq 1$ . Hence, the induction base holds.

Assume the proposition holds for  $j$ . Case analysis on bit  $j+1$  of  $\text{rep}(i)$ :

- $\text{rep}(i)_{(j+1) \bmod q} = 1$ : Here  $\text{out}(i,j+1) = \text{in}(i,j)$ , and  $s(\text{rep}(i),j+1) = t(\text{rep}(i),j)$ , and  $\text{in}(i,j)$  has received block  $t(\text{rep}(i),j)$  in round  $j$ . Also,  $\text{in}(i,j+1) = \text{out}(i,j)$ , and  $s(\text{rep}(i),j) = (j+1)-q-1$
- $\text{rep}(i)_{(j+1) \bmod q} = 0$ : Here  $\text{out}(i,j+1) = \text{out}(i,j)$  and  $s(\text{rep}(i),j+1) = s(\text{rep}(i),j)$ . Also,  $\text{in}(i,j+1) = \text{in}(i,j)$  which receives block  $t(\text{rep}(i),j)$  in round  $j$ ; and  $t(\text{rep}(i),j) = j+1-q-1$ .

Proposition:

Before round  $j$ , processor  $\text{in}(i,j)$  has received block  $j-q-1$  if  $j-q-1 \geq 0$ , and processor  $\text{out}(i,j)$  has received block  $s(\text{rep}(i),j)$  if  $s(\text{rep}(i),j) \geq 0$

This shows correctness of the final, extra exchange step. Together with the previous proposition, the main theorem follows



### Main Theorem:

In the fully connected, 1-ported communication model, broadcast of  $M$  blocks can be done optimally in  $M-1+\text{ceil}(\log p)$  communication rounds. In the linear cost model

$$T_{\text{broadcast}}(m) = (\text{ceil}(\log p)-1)\alpha + 2\sqrt{[\text{ceil}(\log p)-1]\alpha\beta m} + \beta m$$

### Note :

There are at least three other algorithms in the literature achieving the optimal bound; Bin Jia's is arguably the most elegant and easy to implement.

Another algorithms achieving the same result is:

Jesper Larsson Träff, Andreas Ripke: Optimal broadcast for fully connected processor-node networks. J. Parallel Distrib. Comput. 68(7): 887-901 (2008)

**Not elegant** , needs  $O(p \log^2 p)$  step precomputation to determine schedule of  $\text{ceil}(\log p)$  entries for process all processes.

**Challenge:** process local,  $O(\log p)$  step schedule computation for local process  $i$ ,  $0 \leq i < p$

**Advantage:** Can be used for allgather as well, exploiting allgather  $\approx p$  bcast, in particular to give an optimal allgather algorithm

Master thesis?

## Recent, major improvement

Jesper Larsson Träff: Fast(er) Construction of Round-optimal n-Block Broadcast Schedules. CLUSTER 2022: 142-151

Jesper Larsson Träff: Brief Announcement: Fast(er) Construction of Round-optimal n-Block Broadcast Schedules. SPAA 2022: 143-146

- Sublinear,  $O(\log^3 p)$  precomputation time per processor
- Implementation for Broadcast and Allgather(v)

Possible to do in  $O(\log p)$  precomputation?

Probably “yes”

## Yet more algorithms with round and bandwidth optimal broadcast

Amotz Bar-Noy, Shlomo Kipnis, Baruch Schieber: Optimal multiple message broadcasting in telephone-like communication systems. Discrete Applied Mathematics 100(1-2): 1-15 (2000)

Oh-Heum Kwon, Kyung-Yong Chwa: Multiple message broadcasting in communication networks. Networks 26(4): 253-261 (1995)

## In LogP model (k-item broadcast)

Richard M. Karp, Abhijit Sahay, Eunice E. Santos, Klaus E. Schauser: Optimal Broadcast and Summation in the LogP Model. SPAA 1993: 142-153

## Alltoall communication

```
MPI_Alltoall(sendbuf, scount, stype,  
             recvbuf, rcount, rtype, comm) ;
```

Each MPI process has an individual (personalized) block of data (“sendbuf[i]”) to each other process in comm (including itself)

Each MPI process receives an individual (personalized) block of data (“recvbuf[i]”) from each other process in comm (including itself)

Alltoall, personalized alltoall, total exchange, transpose, ...

## Bisection width lower bound for alltoall communication

### Definition:

The bisection width of a network is the minimum number edges that have to be removed to partition the network into two parts with  $\text{floor}(p/2)$  and  $\text{ceil}(p/2)$  nodes, respectively.

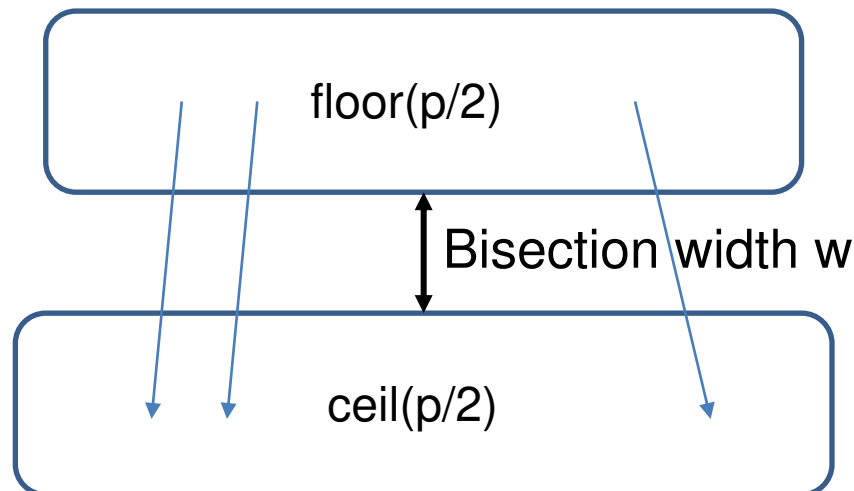
### Observation:

Let the bisection width of a  $k$ -ported, bidirectional network be  $w$ . Any alltoall algorithm that sends/receives each block separately requires at least

$$\text{floor}(p/2) \times \text{ceil}(p/2)/w/k$$

communication rounds (and, trivially, at least  $p-1$  rounds)

Argument: In any partition of  $p$  into roughly equal sized parts,  $\lceil p/2 \rceil/k$  of the  $(p-1)/k$  communication rounds need to cross the cut. In each round,  $\lfloor p/2 \rfloor$  processors want to communicate, but there is a minimum cut that can accommodate at most  $w$  simultaneous communication operations. Thus the number of rounds is at least  $\lceil p/2 \rceil \times \lfloor p/2 \rfloor / w/k$



## Bisection width facts

- Fully connected network:  $w = \text{floor}(p/2) \times \text{ceil}(p/2)$
- Linear array (ring): 1 (2)
- 2-dimensional torus with equal dimension sizes:  $w = 2\sqrt{p}$  (for  $\sqrt{p} > 2$ )
- 3-dimensional torus with equal dimension sizes:  $w = 2(p^{1/3})^2$
- d-dimensional torus with equal dimension sizes:  $2(\sqrt[d]{p})^{d-1}$  (for  $\sqrt[d]{p} > 2$ , if  $=2$ , divide cut size by 2)
- d-dimensional hypercube:  $2^{d-1} = 2^{\log p - 1}$

Note: d-dimensional hypercube is isomorphic to a d-dimensional torus with dimension size 2!



## Direct alltoall, fully-connected network

```
MPI_Alltoall(sendbuf, scount, stype,  
             recvbuf, rcount, rtype, comm) ;
```

Direct algorithm for fully connected networks: Each processor sends to and receives from each other processor, including itself

```
for (i=1; i<=size; i++) {  
    prev = (rank-i+size)%size;  
    next = (rank+i)%size;  
    MPI_Sendrecv(sendbuf[next], ..., next, ...,  
                 recvbuf[prev], ..., prev, ..., comm) ;  
}
```

Direct algorithm,  $p-1$  communication rounds ( **no** communication in last round):

- High latency,  $(p-1)\alpha$
- Optimal in bandwidth term, since  $(p-1)/p m$  units of data have to be sent and received by each processor
- **Exploits fully bidirectional, send-receive communication**

$$T_{\text{alltoall}}(m) = (p-1)\alpha + (p-1)/p \beta m$$

```
for (i=1; i<=size; i++) {
    prev = (rank-i+size)%size;
    next = (rank+i)%size;
    MPI_Sendrecv(sendbuf[next], ..., next, ...,
                  recvbuf[prev], ..., prev, ..., comm) ;
}
```

Less tightly coupled alltoall: All scheduling decisions left to MPI library and communication subsystem

```
MPI_Request request[2*size];
MPI_Status status[2*size];

for (i=1; i<=size; i++) {
    prev = (rank-i+size)%size;
    next = (rank+i)%size;
    MPI_Irecv(recvbuf[prev], ..., prev, ..., comm,
              &request[2*(i-1)]);
    MPI_Isend(sendbuf[next], ..., next, ..., comm,
              &request[2*i-1]);
}
MPI_Waitall(2*size, request, status);
```

**Possible MPI performance problem: Too many outstanding requests for large size communicators**

## Telephone exchange algorithm (1-factoring)

Lemma: The fully connected network (complete graph with self-loops) can be partitioned into  $p$  subgraphs in each of which each node has degree exactly 1 (allowing self-loops): 1-factors

Proof:

Define for each node  $u$  the  $i$ 'th partner as

$$v_i(u) = (i - u + p) \bmod p.$$

Since  $v_i(v_i(u)) = (i - (i - u + p) \bmod p) \bmod p = u$  (\*), which shows that each node has degree 1, the  $i$ 'th factor can be defined as the set of edges  $(u, v_i(u))$  for  $i=0, \dots, p-1$

(\*) if  $i - u \geq 0$ , then  $(i - u + p) \bmod p = i - u$ , and  $u \bmod p = u$ ; if  $i - u < 0$ , then  $(i - u + p) \bmod p = i - u + p$

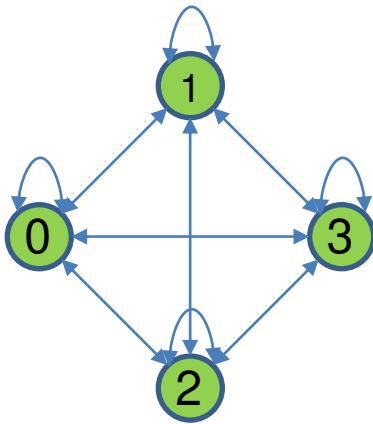
## 1-factor self-loop algorithm for alltoall

```
for (i=0; i<size; i++) {  
    int partner = (i-rank+size)%size;  
  
    MPI_Sendrecv(sendbuf[partner], ..., partner,  
                 recvbuf[partner], ..., partner,  
                 comm, MPI_STATUS_IGNORE);  
}
```

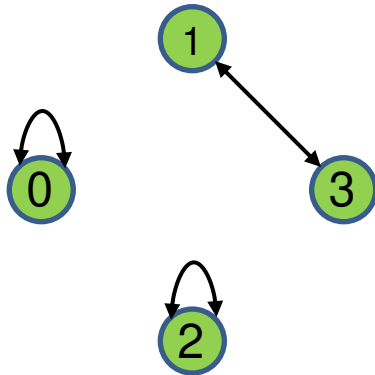
Works for all  $p$  (even, odd, not power-of-two, ...)

$$T_{\text{alltoall}}(m) = p\alpha + \beta m$$

Example,  $p=4$



Example,  $p=4$



Round 0:

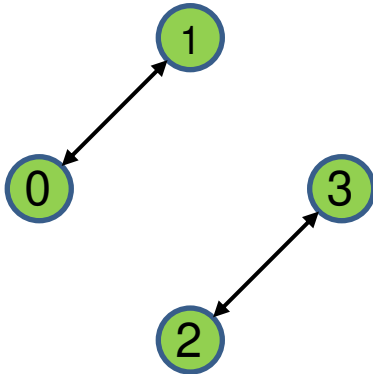
$$v_0(0) = (0-0) \bmod p = 0$$

$$v_0(1) = (0-1) \bmod p = 3$$

$$v_0(2) = (0-2) \bmod p = 2$$

$$v_0(3) = (0-3) \bmod p = 1$$

Example,  $p=4$



Round 1:

$$v_1(0) = (1-0) \bmod p = 1$$

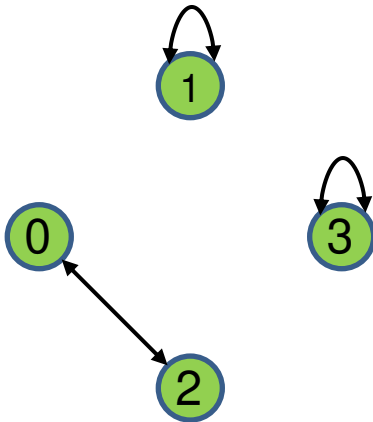
$$v_1(1) = (1-1) \bmod p = 0$$

$$v_1(2) = (1-2) \bmod p = 3$$

$$v_1(3) = (1-3) \bmod p = 2$$



Example,  $p=4$



Round 2:

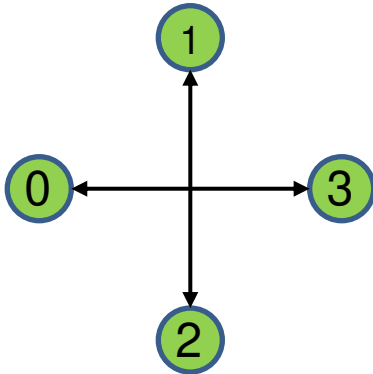
$$v_2(0) = (2-0) \bmod p = 2$$

$$v_2(1) = (2-1) \bmod p = 1$$

$$v_2(2) = (2-2) \bmod p = 0$$

$$v_2(3) = (2-3) \bmod p = 3$$

Example,  $p=4$



Round 3:

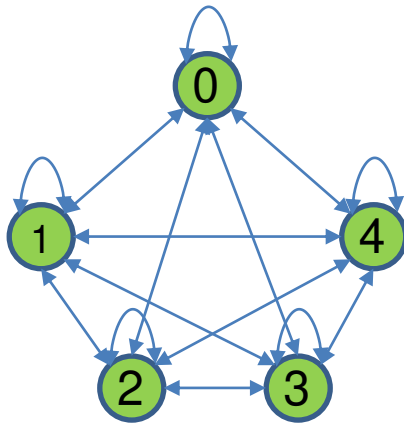
$$v_3(0) = (3-0) \bmod p = 3$$

$$v_3(1) = (3-1) \bmod p = 2$$

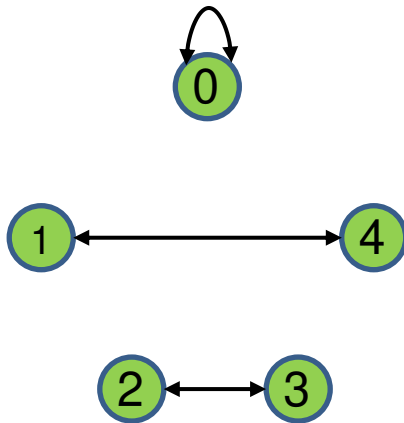
$$v_3(2) = (3-2) \bmod p = 1$$

$$v_3(3) = (3-3) \bmod p = 0$$

Example,  $p=5$



Example,  $p=5$



Round 0:

$$v_0(0) = (0-0) \bmod p = 0$$

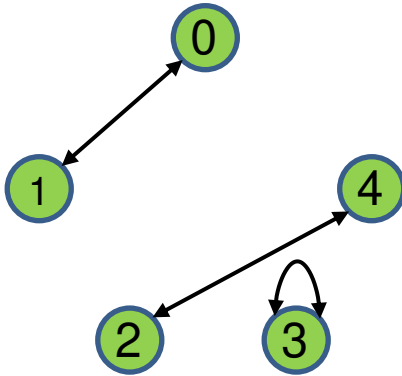
$$v_0(1) = (0-1) \bmod p = 4$$

$$v_0(2) = (0-2) \bmod p = 3$$

$$v_0(3) = (0-3) \bmod p = 2$$

$$v_0(4) = (0-4) \bmod p = 1$$

Example,  $p=5$



Round 1:

$$v_1(0) = (1-0) \bmod p = 1$$

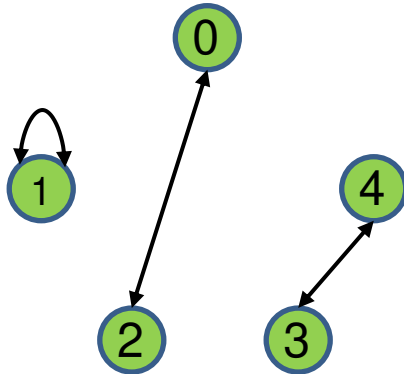
$$v_1(1) = (1-1) \bmod p = 0$$

$$v_1(2) = (1-2) \bmod p = 4$$

$$v_1(3) = (1-3) \bmod p = 3$$

$$v_1(4) = (1-4) \bmod p = 2$$

Example,  $p=5$



Round 2:

$$v_2(0) = (2-0) \bmod p = 2$$

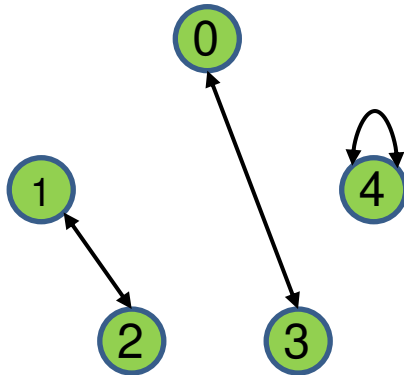
$$v_2(1) = (2-1) \bmod p = 1$$

$$v_2(2) = (2-2) \bmod p = 0$$

$$v_2(3) = (2-3) \bmod p = 4$$

$$v_2(4) = (2-4) \bmod p = 3$$

Example,  $p=5$



Round 3:

$$v_3(0) = (3-0) \bmod p = 3$$

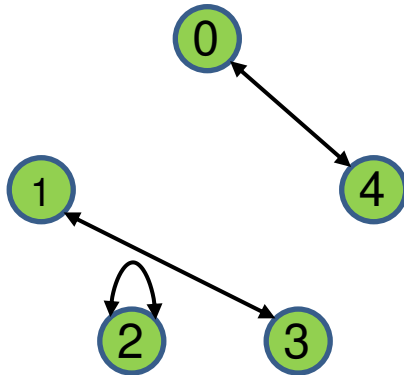
$$v_3(1) = (3-1) \bmod p = 2$$

$$v_3(2) = (3-2) \bmod p = 1$$

$$v_3(3) = (3-3) \bmod p = 0$$

$$v_3(4) = (3-4) \bmod p = 4$$

Example,  $p=5$



Round 4:

$$v_4(0) = (4-0) \bmod p = 4$$

$$v_4(1) = (4-1) \bmod p = 3$$

$$v_4(2) = (4-2) \bmod p = 2$$

$$v_4(3) = (4-3) \bmod p = 1$$

$$v_4(4) = (4-4) \bmod p = 0$$



Improvements:

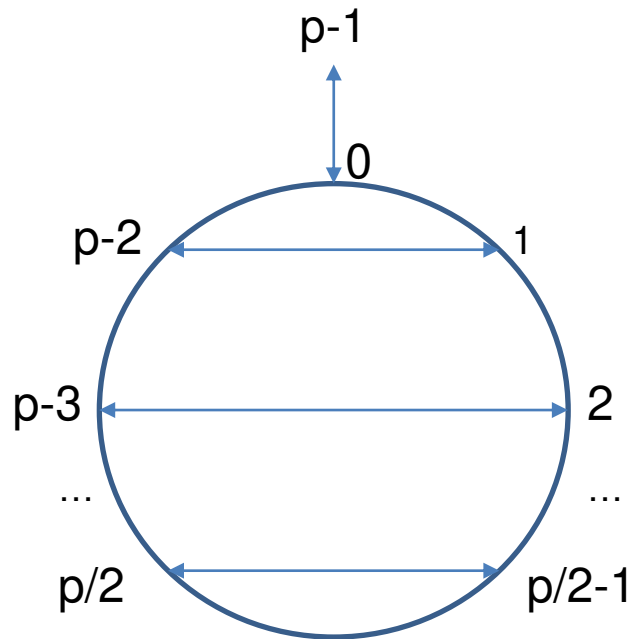
- $p$  even:  $p-1$  communication rounds needed
- $p$  odd:  $p$  communication rounds needed

Lemma: For even  $p$ , there exist a 1-factorization of the fully connected network into  $(p-1)$  1-factors. For  $p$  odd, there exist an almost 1-factorization into  $p$  almost 1-factors (each factor has one un-paired node)

Proof: Folklore, see for instance

Eric Mendelsohn, Alexander Rosa: One-factorizations of the complete graph - A survey. Journal of Graph Theory 9(1): 43-65 (1985)

Even  $p$  construction: The  $i$ 'th 1-factor has edges  $(i, p-1)$  and  $((j+i) \bmod (p-1), (p-1-j+i) \bmod (p-1))$  for  $j=1, \dots, p/2-1$ . Each node clearly has degree 1, and each edge of the network is in exactly one factor

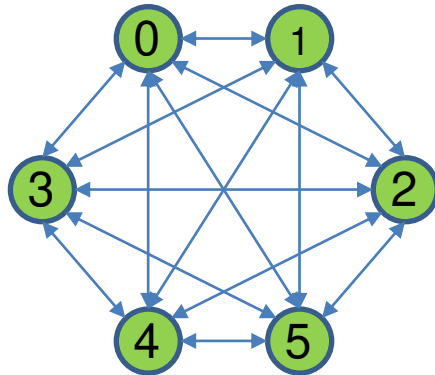


Round  $i$ ,  $0 \leq i < p-1$ , for each node  $u$ :

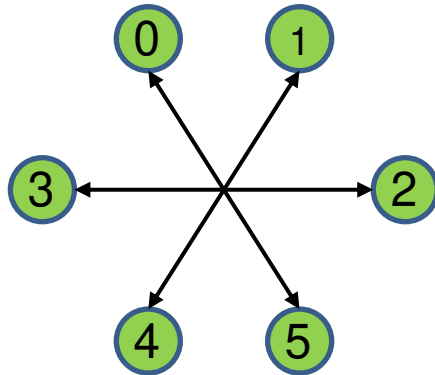
```
if (u==p-1) v = i; else {
    uu = (p-1-u-i+p-1)%(p-1);
    if (uu==0) v = p-1;
    else
        v = ((p-1-uu)+i)%(p-1);
}
```

Rotate along circle, in round  $i$ , node  $u$  communicates with node  $v$

Example,  $p=6$

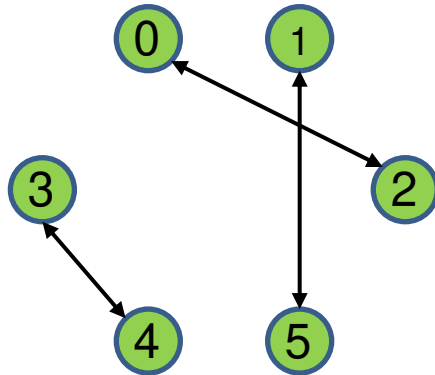


Example,  $p=6$



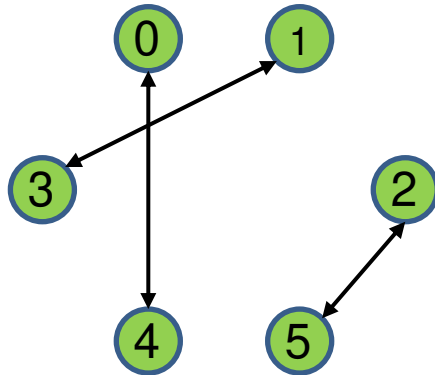
Round 0:

Example,  $p=6$



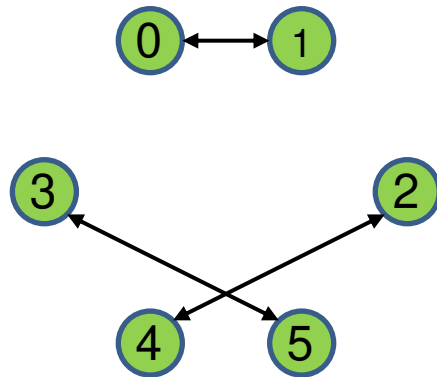
Round 1:

Example,  $p=6$



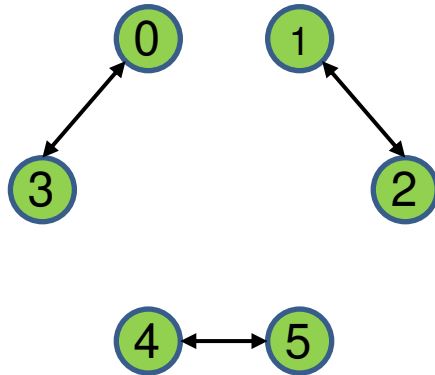
Round 2:

Example,  $p=6$



Round 3:

Example,  $p=6$

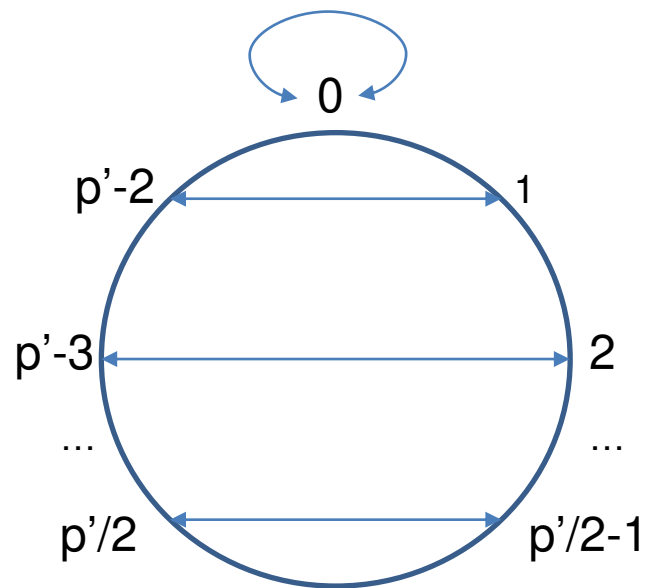


Round 4:

$$T_{\text{alltoall}}(m) = (p-1)\alpha + \beta m(p-1)/p$$



Odd  $p$  construction: Use even  $p$  construction for  $p+1$ , remove in each round the edge to virtual node  $p-1$  (or use previous construction)

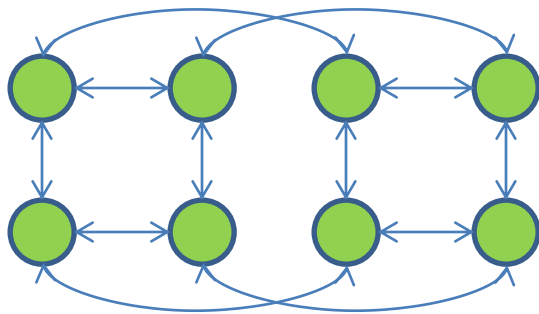


$p$  odd,  $p' = p+1$

Folklore: Even  $p$ ,  $p$  power of two, use  $v = u \text{ XOR } i, i=1, \dots, p-1$

## The power of the hypercube: Fewer communication rounds

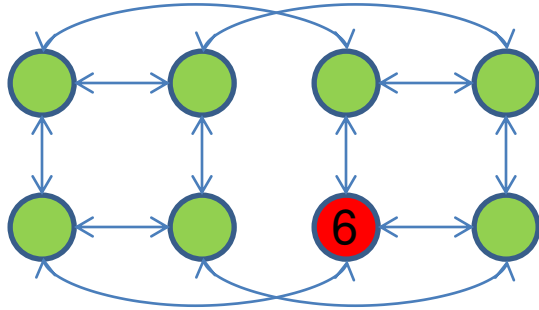
Message combining:



In round  $k$ ,  $0 \leq k < d$ , each processor sends/receives  $2^k$  blocks per processor of  $2^{d-k-1}$  dimensional neighboring hypercube

Neighboring hypercube of processor  $i$  in round  $k$ :  
flip bit  $d-1-k$

Total amount of data per processor per round:  
 $2^k 2^{d-k-1} m/p = 2^{d-1} m/p = m/2$

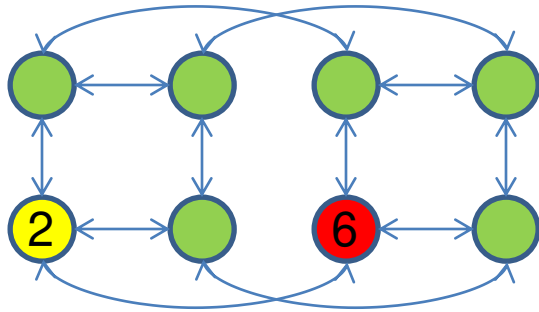


Combine messages:

In round  $k$ ,  $d > k \geq 0$ , each processor sends/receives  $2^k$  blocks per processor of  $2^{d-k-1}$  dimensional neighboring hypercube

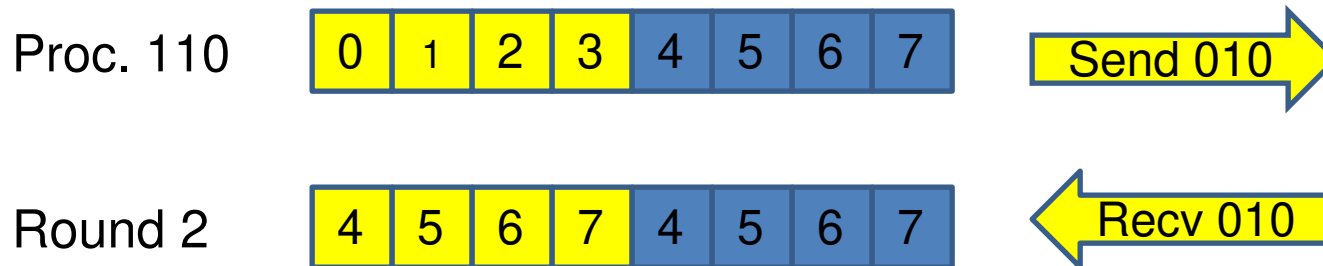
Proc. 110

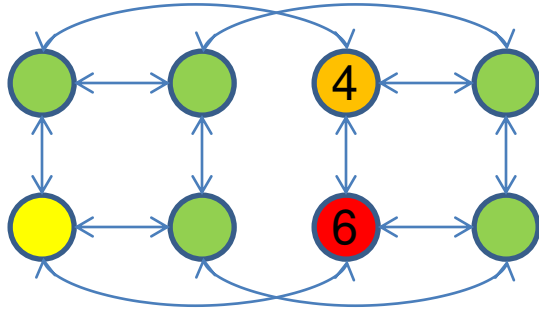




Combine messages:

In round  $k$ ,  $d > k \geq 0$ , each processor sends/receives  $2^k$  blocks per processor of  $2^{d-k-1}$  dimensional neighboring hypercube

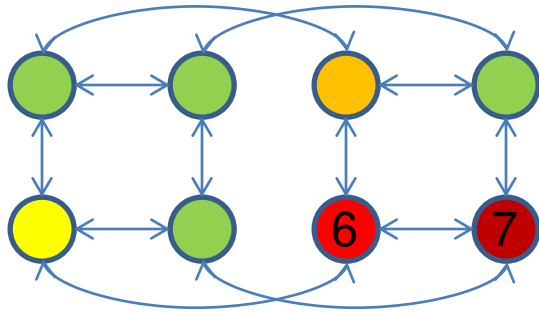




Combine messages:

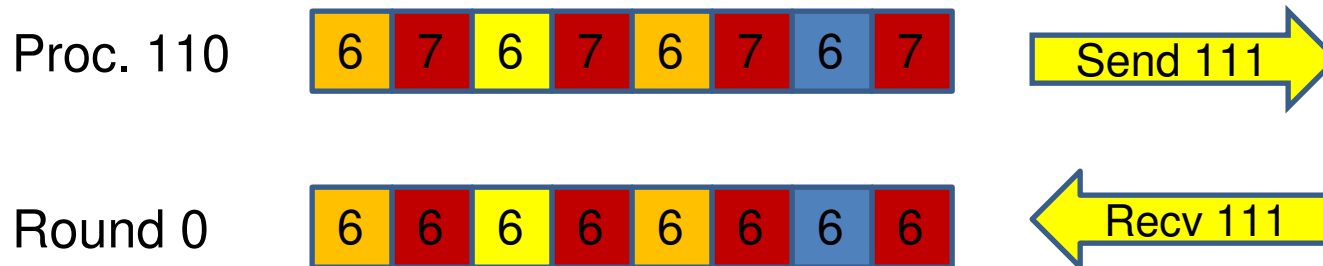
In round  $k$ ,  $d > k \geq 0$ , each processor sends/receives  $2^k$  blocks per processor of  $2^{d-k-1}$  dimensional neighboring hypercube





Combine messages:

In round  $k$ ,  $d > k \geq 0$ , each processor sends/receives  $2^k$  blocks per processor of  $2^{d-k-1}$  dimensional neighboring hypercube



$$T_{\text{alltoall}}(m) = (\log p)(\alpha + \beta m/2)$$

This tradeoff is optimal (see later)

## The power of the circulant graph

The hypercube result does not generalize to fully connected networks (non-power of 2 number of processors).

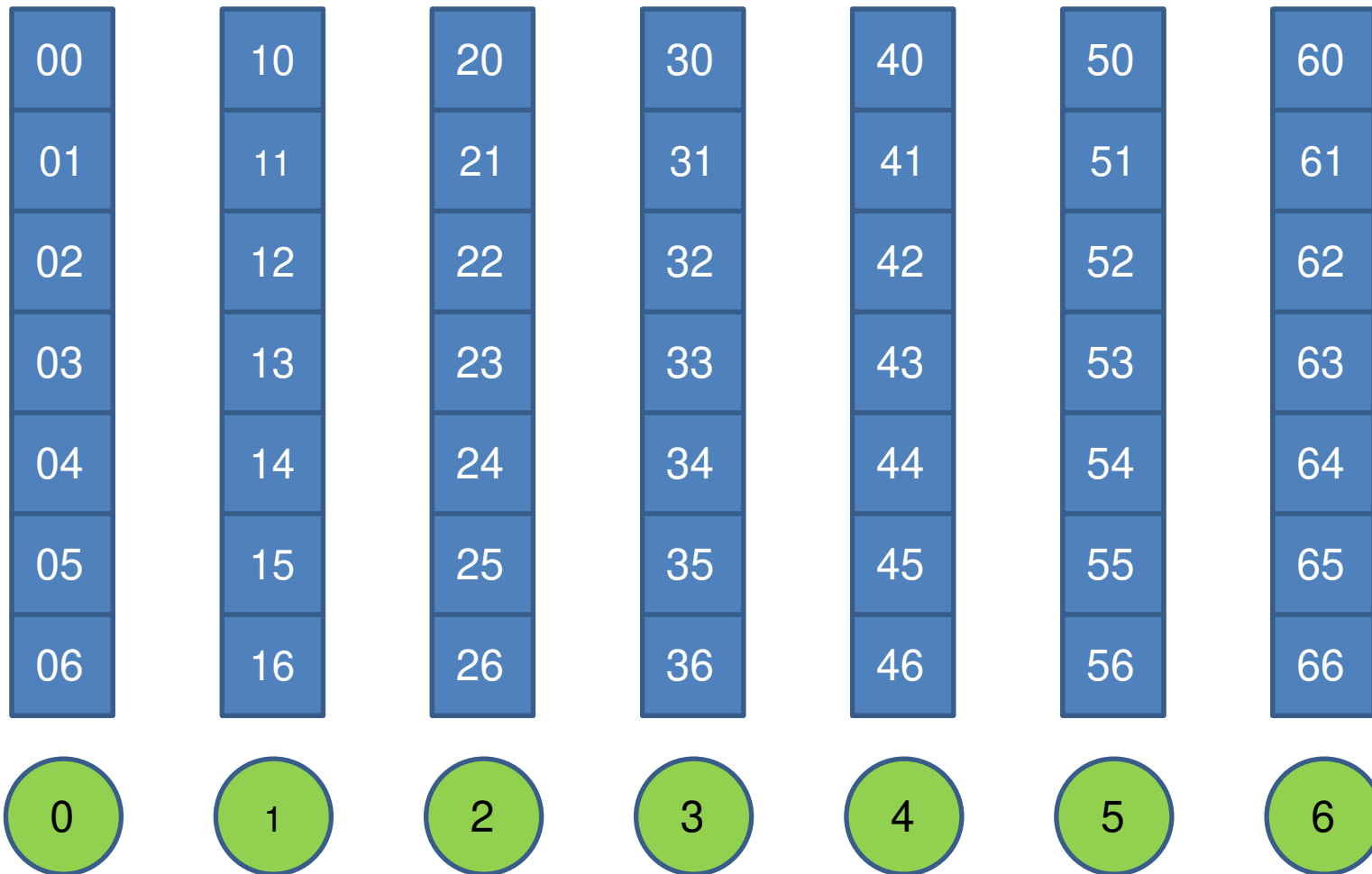
But the pattern (circulant graph) used in Dissemination Allgather algorithm does!

J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, D. Weathersby:  
Efficient Algorithms for All-to-All Communications in Multiport  
Message-Passing Systems. IEEE Trans. Parallel Distrib. Syst.  
8(11): 1143-1156 (1997)



Three (3) steps:

1. Processor local reordering of send blocks to get a symmetric situation for all processors
2. Routing in  $\text{ceil}(\log p)$  rounds with message combining
3. Local reordering to get blocks into rank order

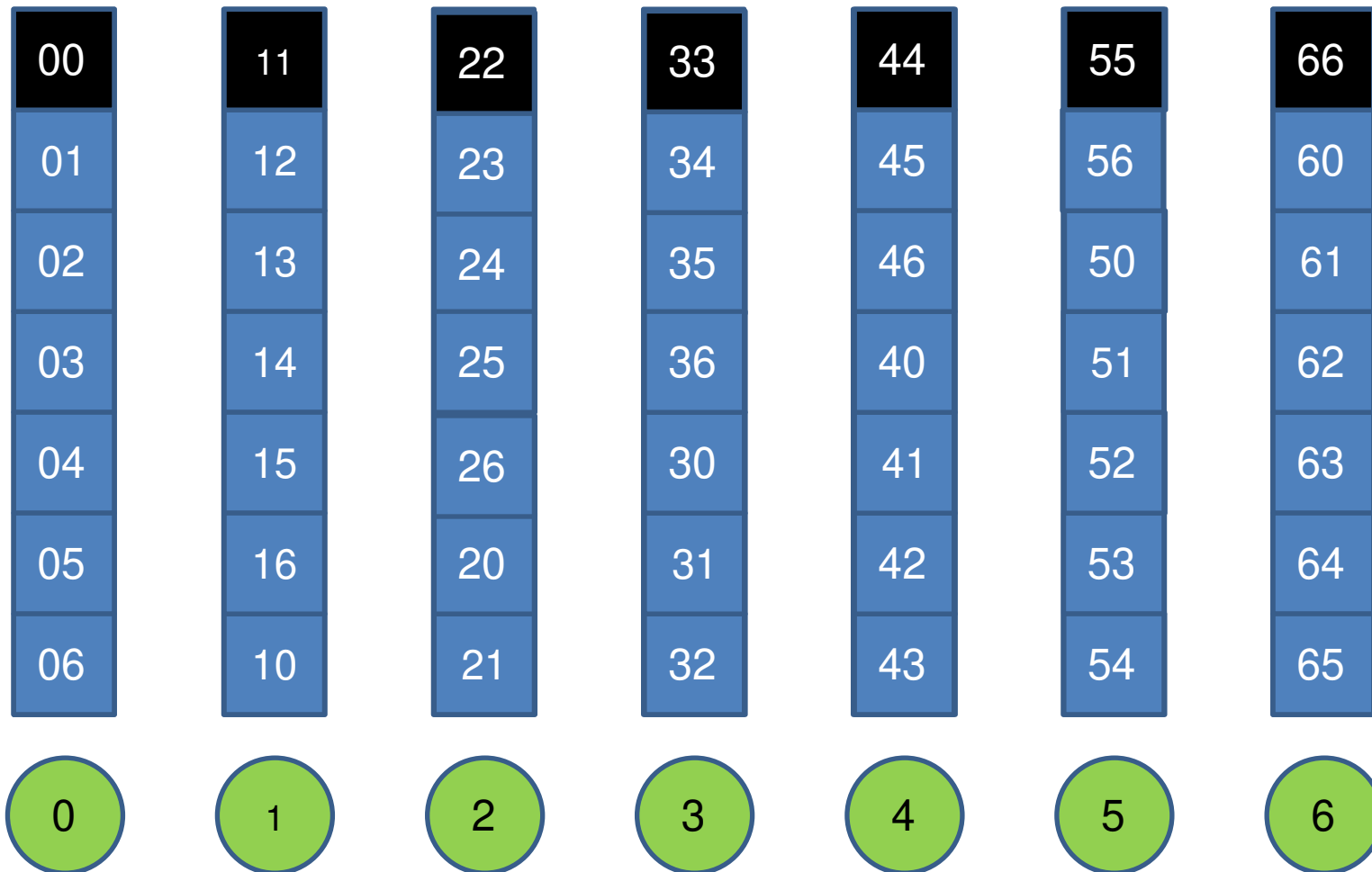
$ij$ Block from processor  $i$  to processor  $j$ 

Step 1:

Local rotate upwards; processor  $i$  by  $i$  positions



Block at right destination processor



After step 1, symmetric situation:

For each process  $i$ : Block in row  $j$  has to be sent to processor  $(i+j) \bmod p$  (“shift”)

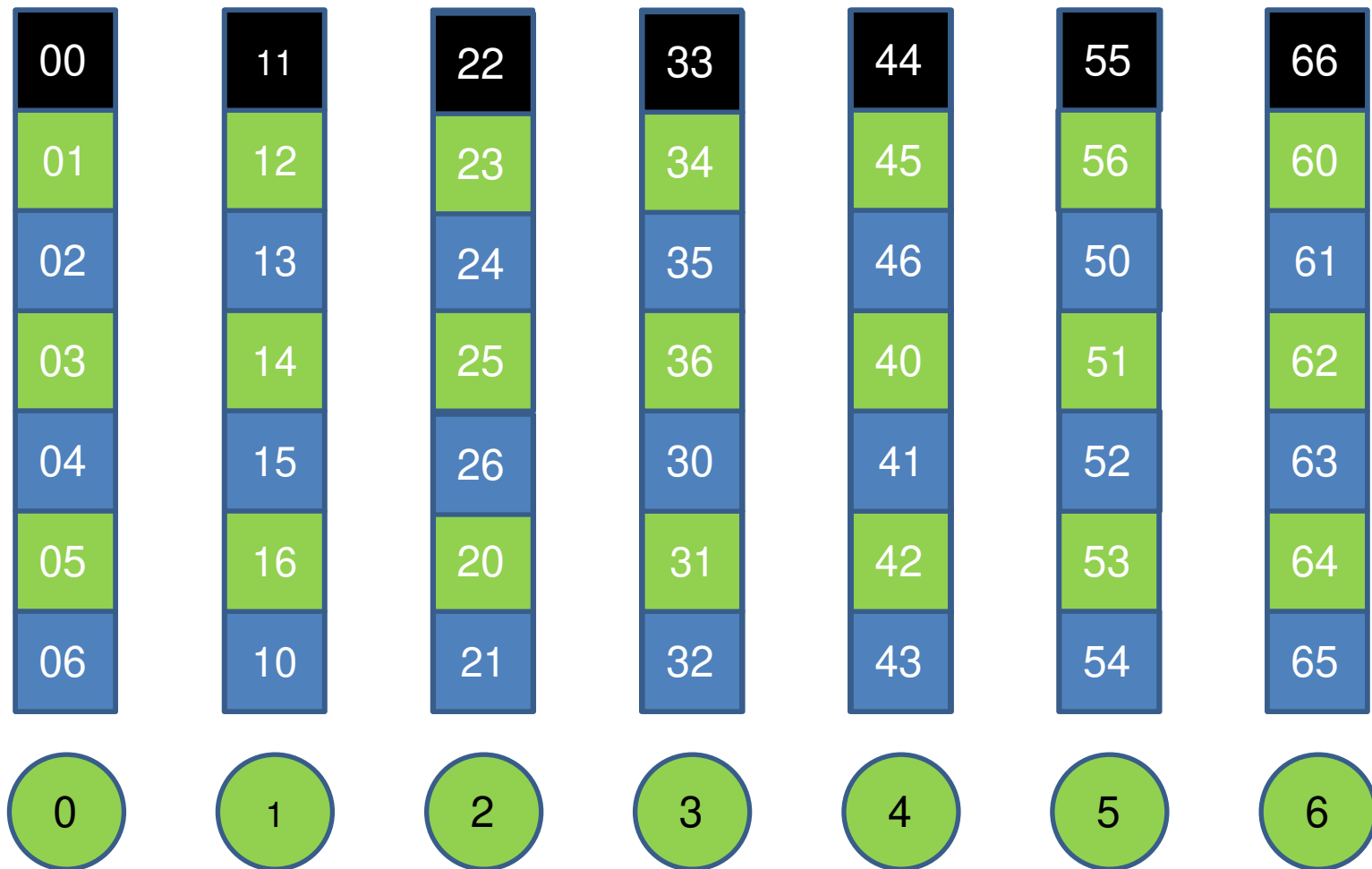
Idea :

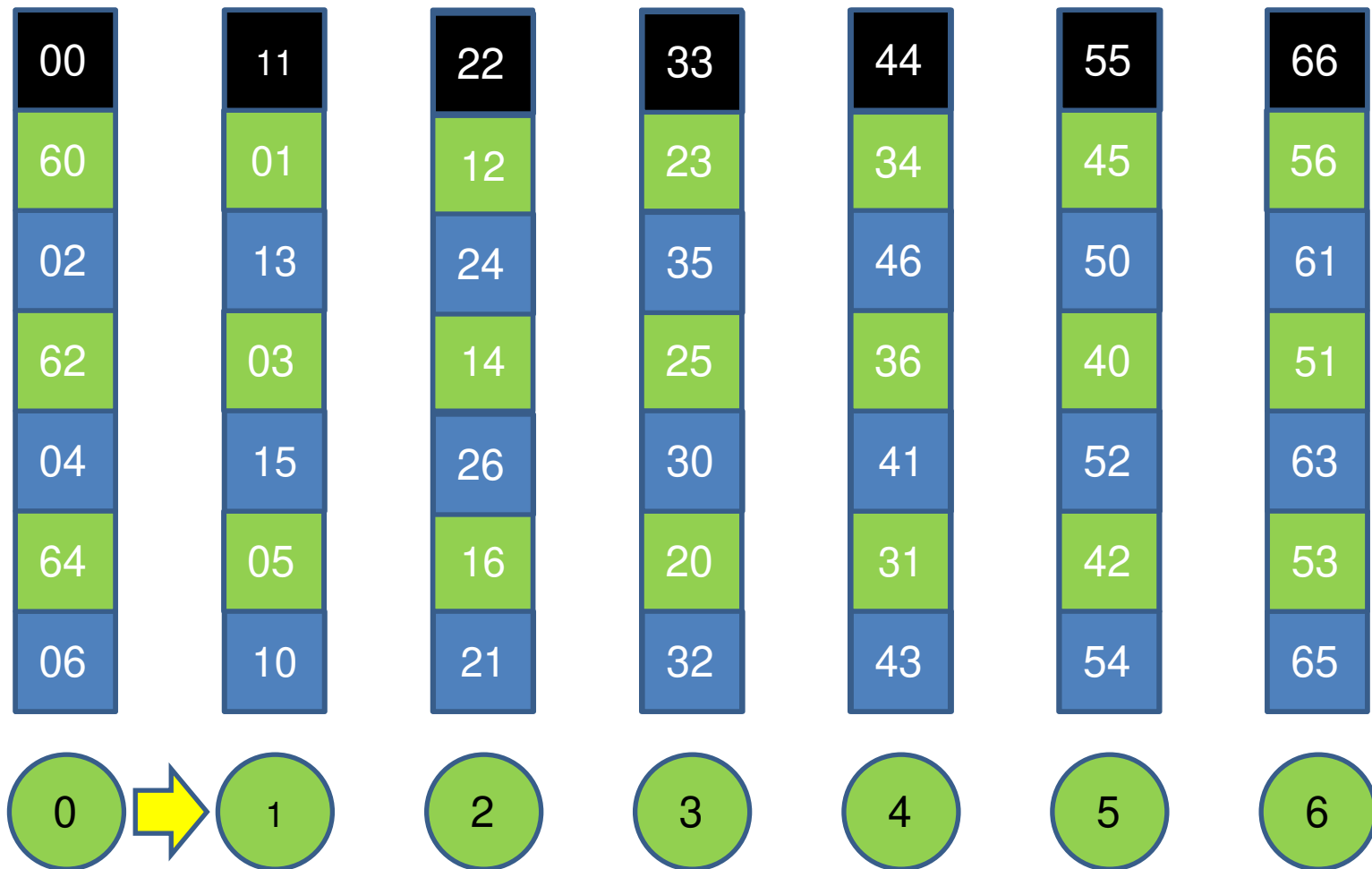
Write row index  $j$  as a binary number (e.g.  $j = 5 = (101)_2 = 2^2 + 2^0$ ), shift according to the 1-bits

Step 2:  $\text{ceil}(\log p)$  rounds

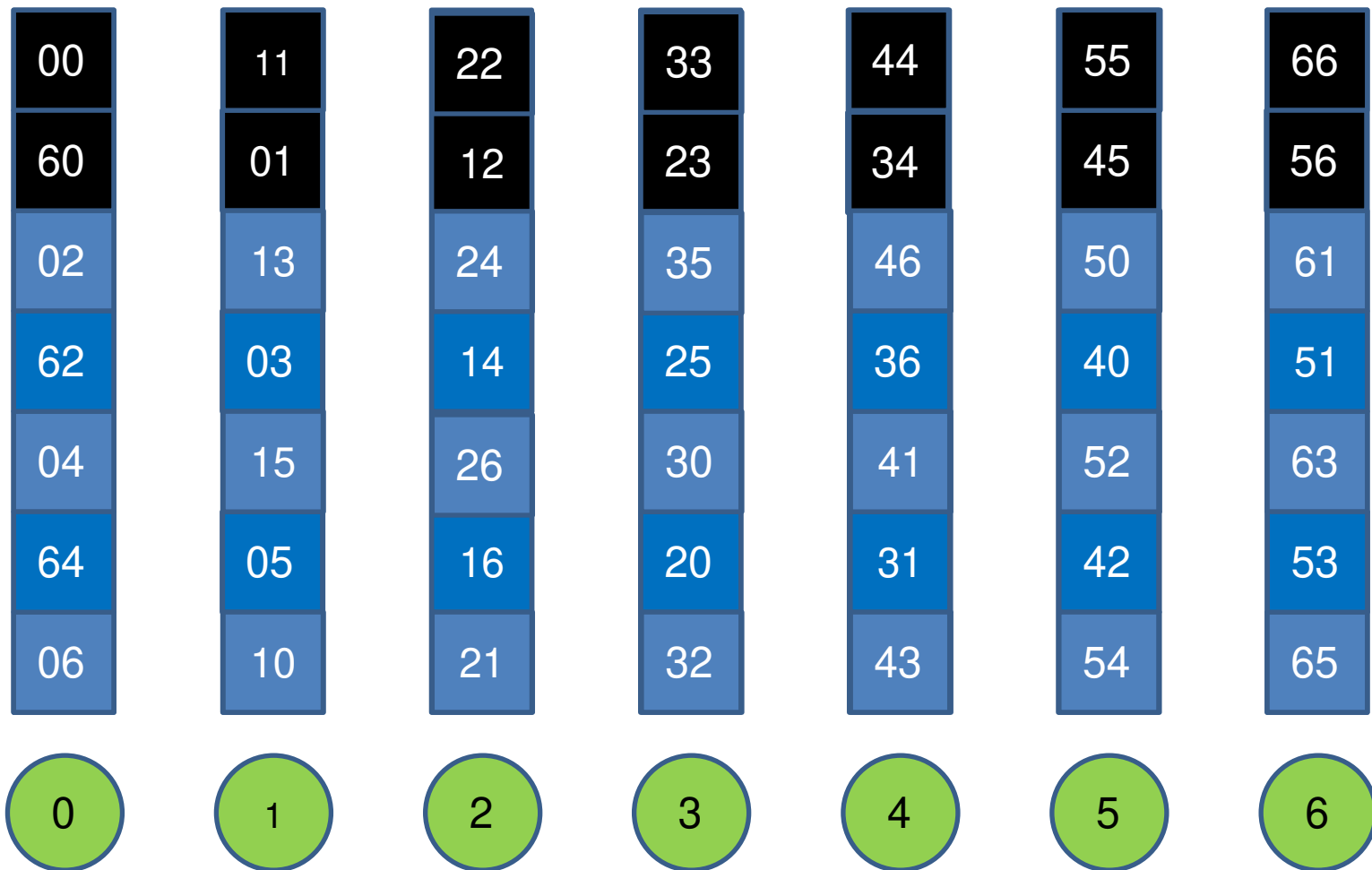
Round  $k$ :

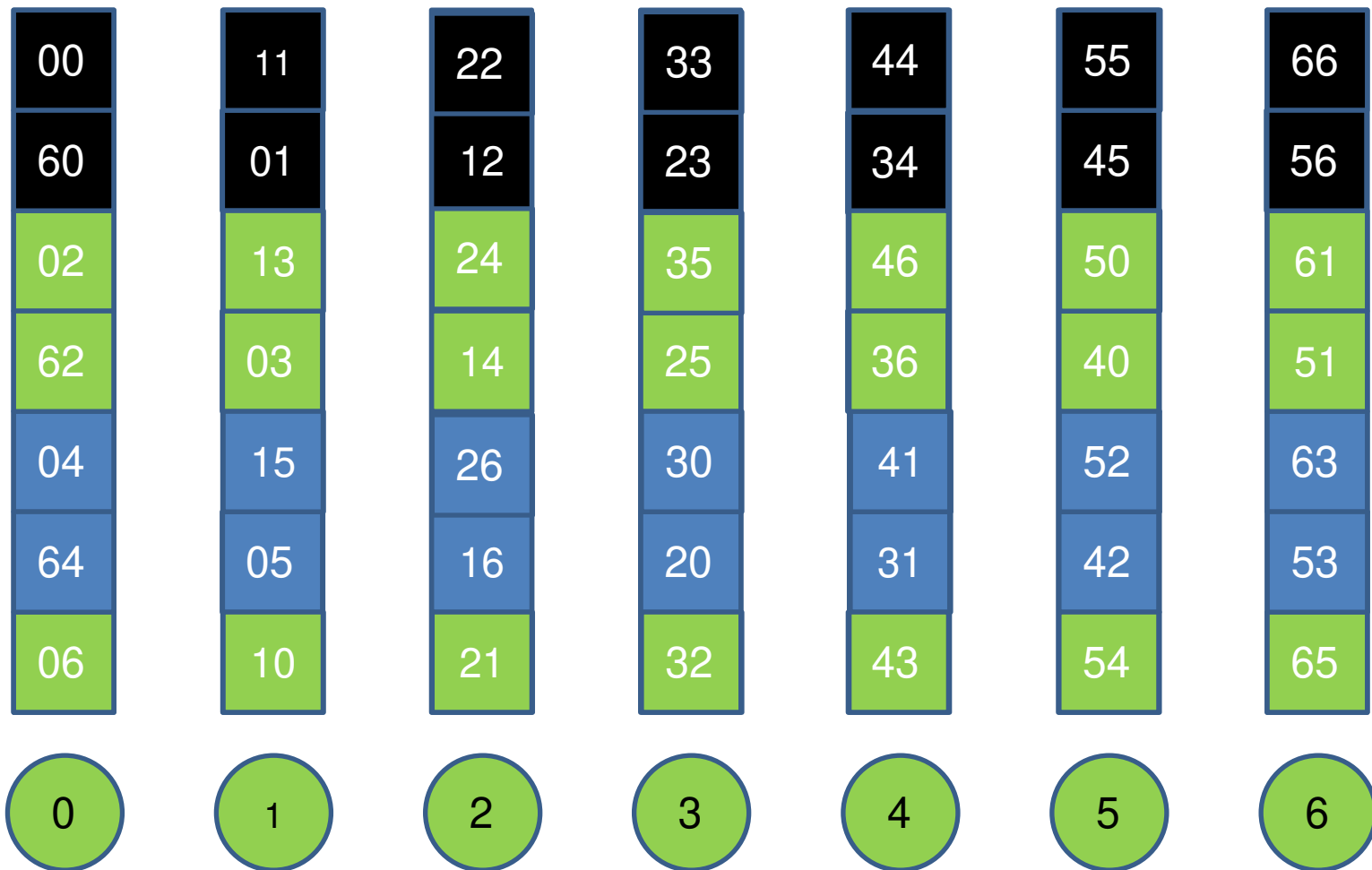
For process  $i$ , all blocks destined to a processor where bit  $k=1$  are combined and sent together to processor  $(i + 2^k) \bmod p$

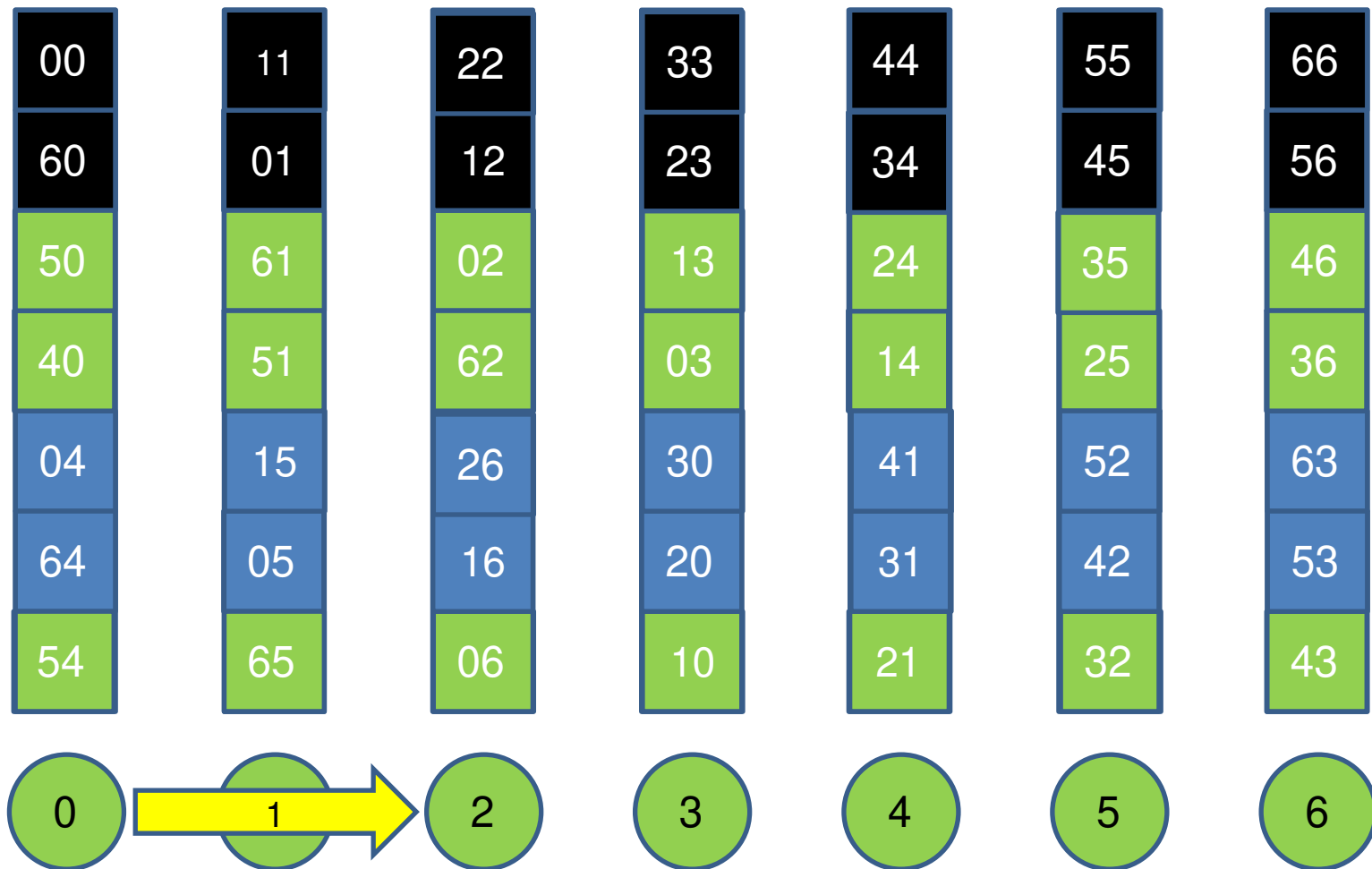


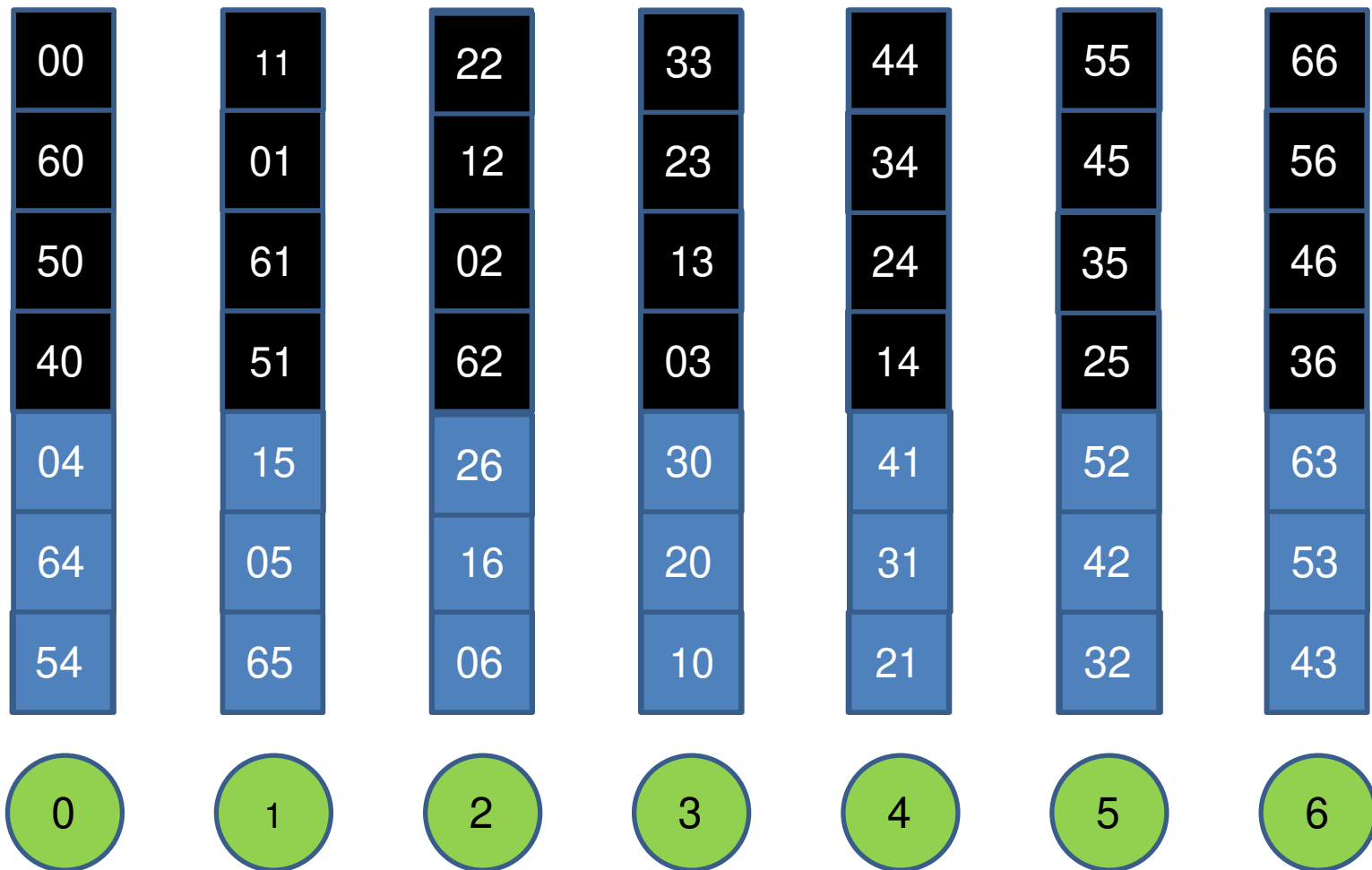


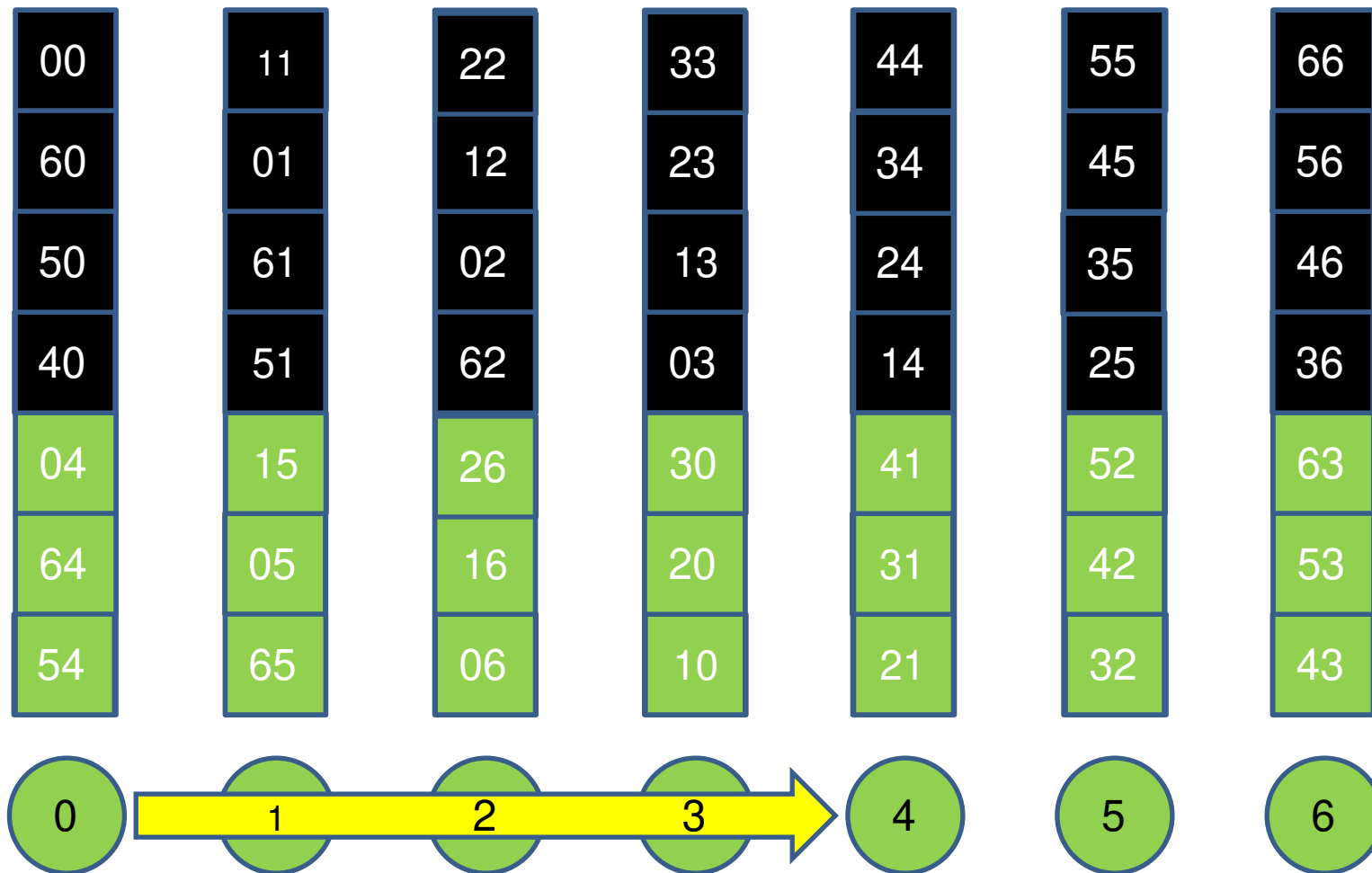


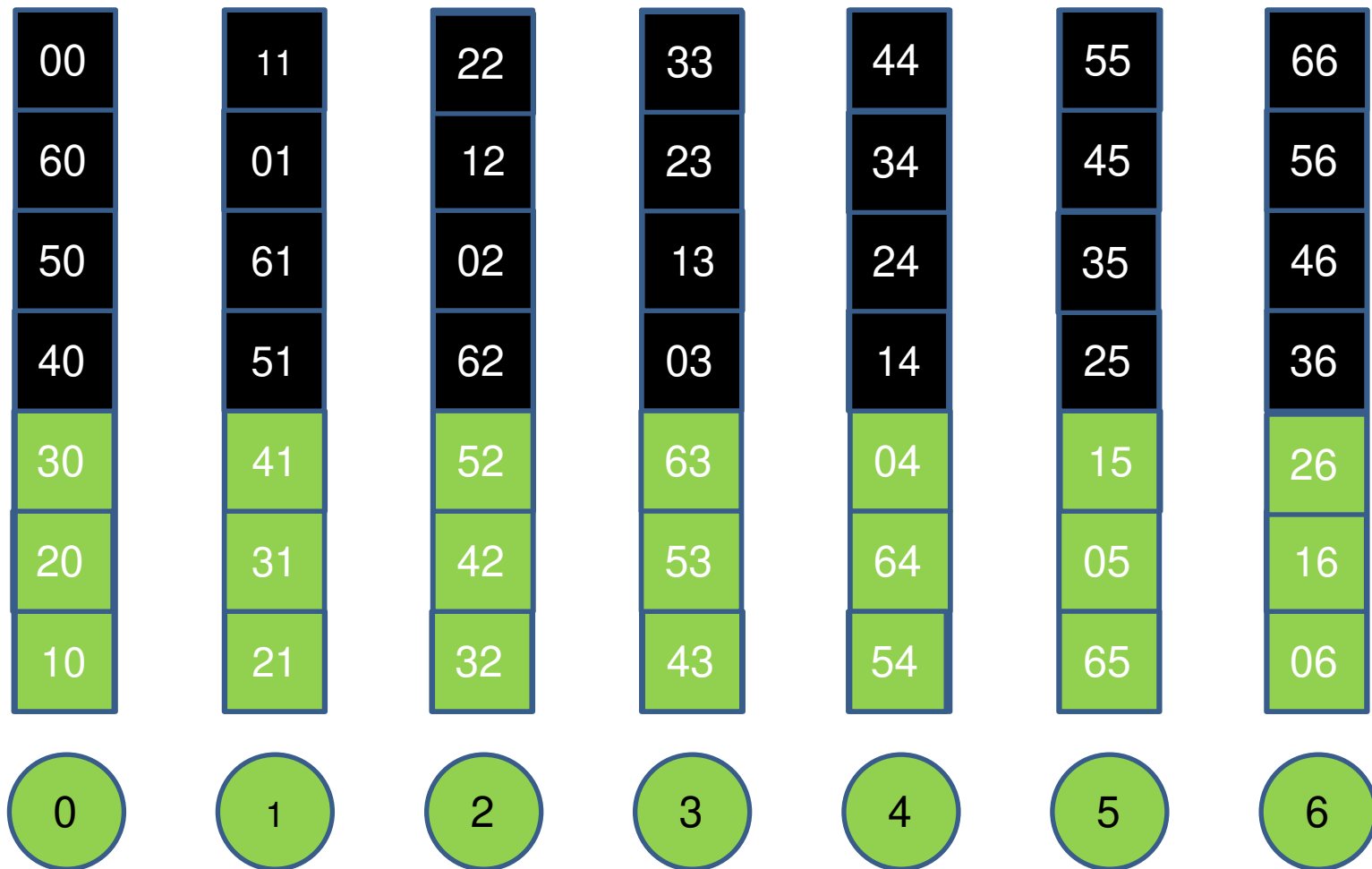


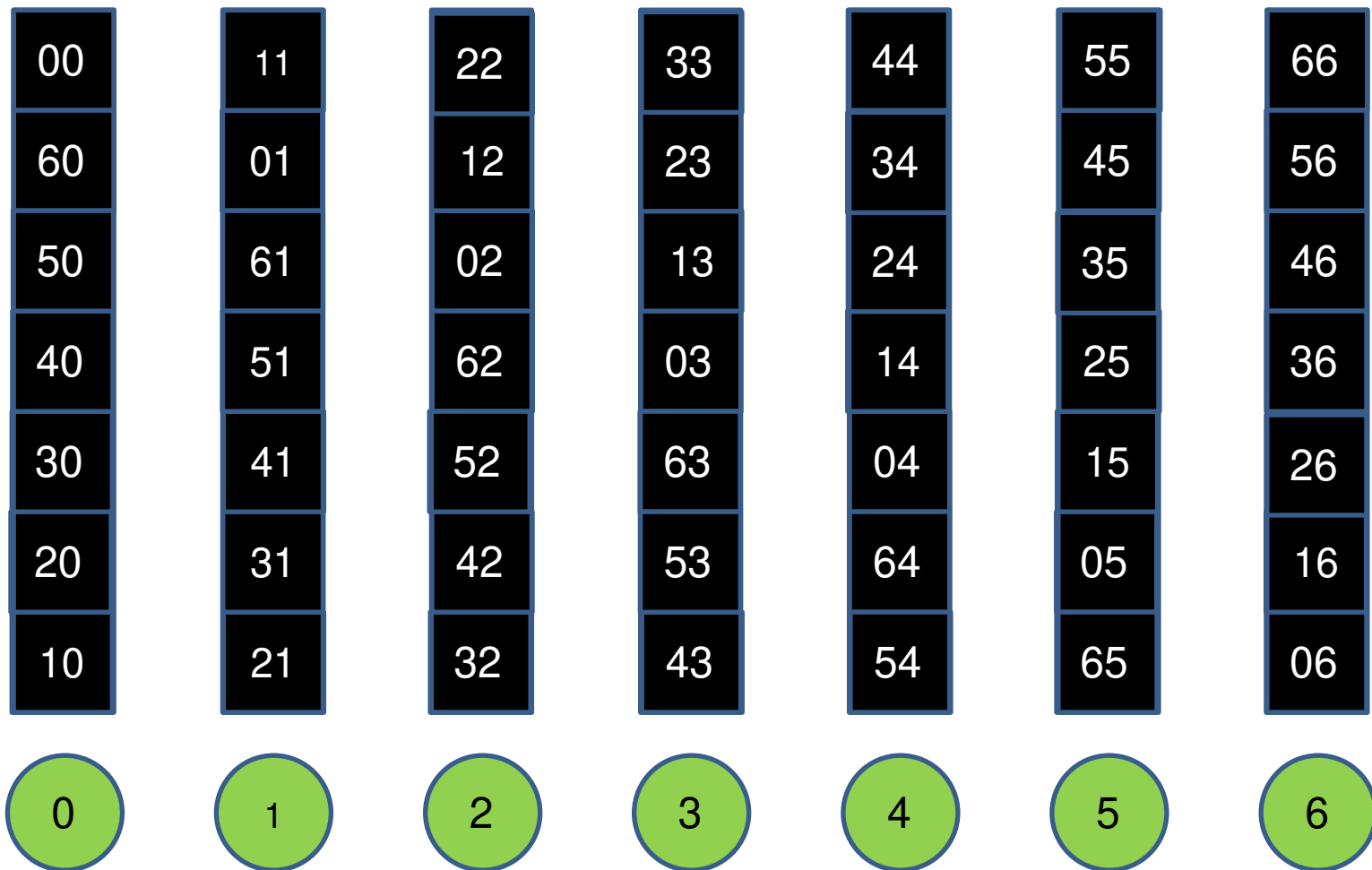








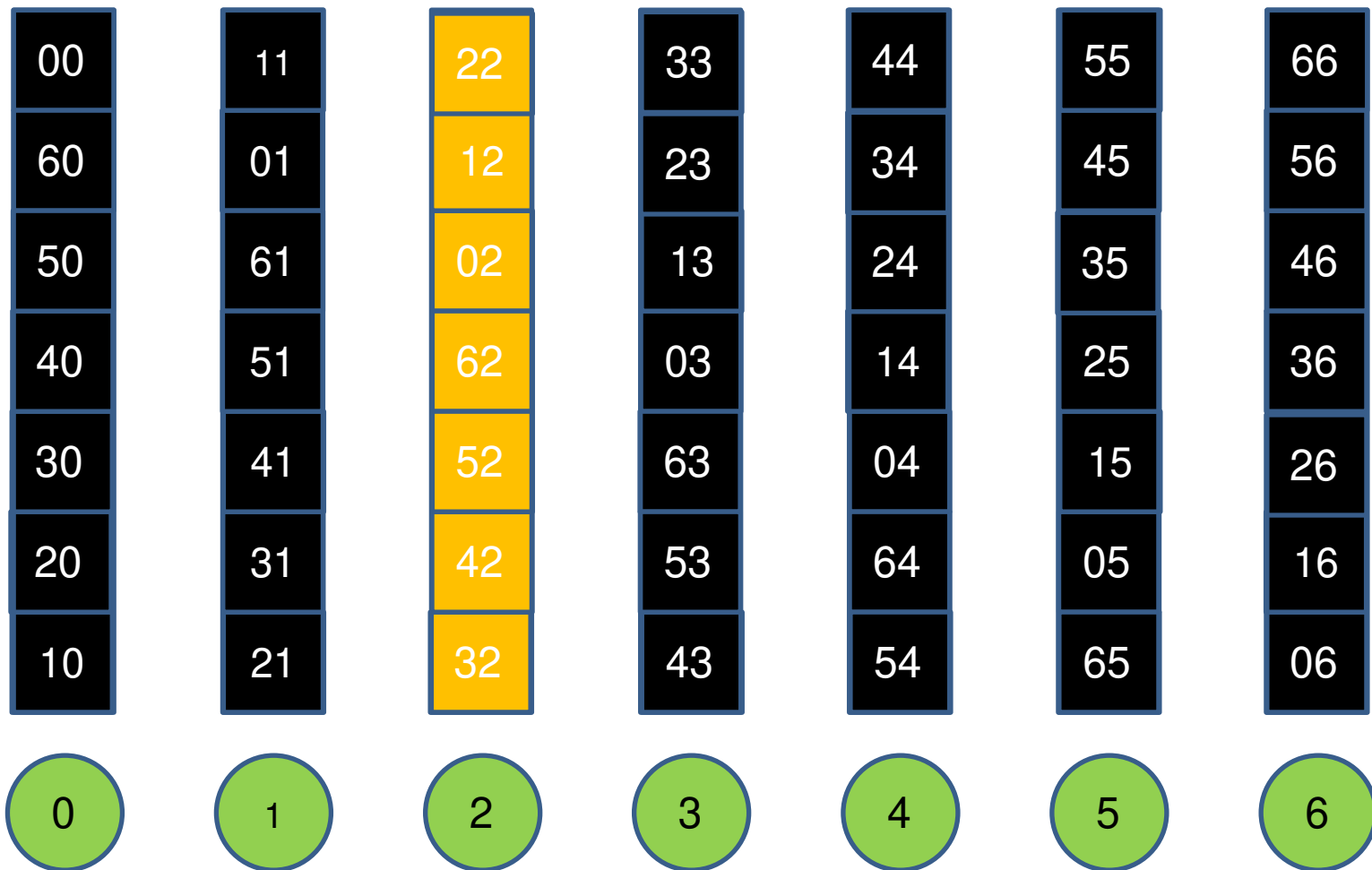


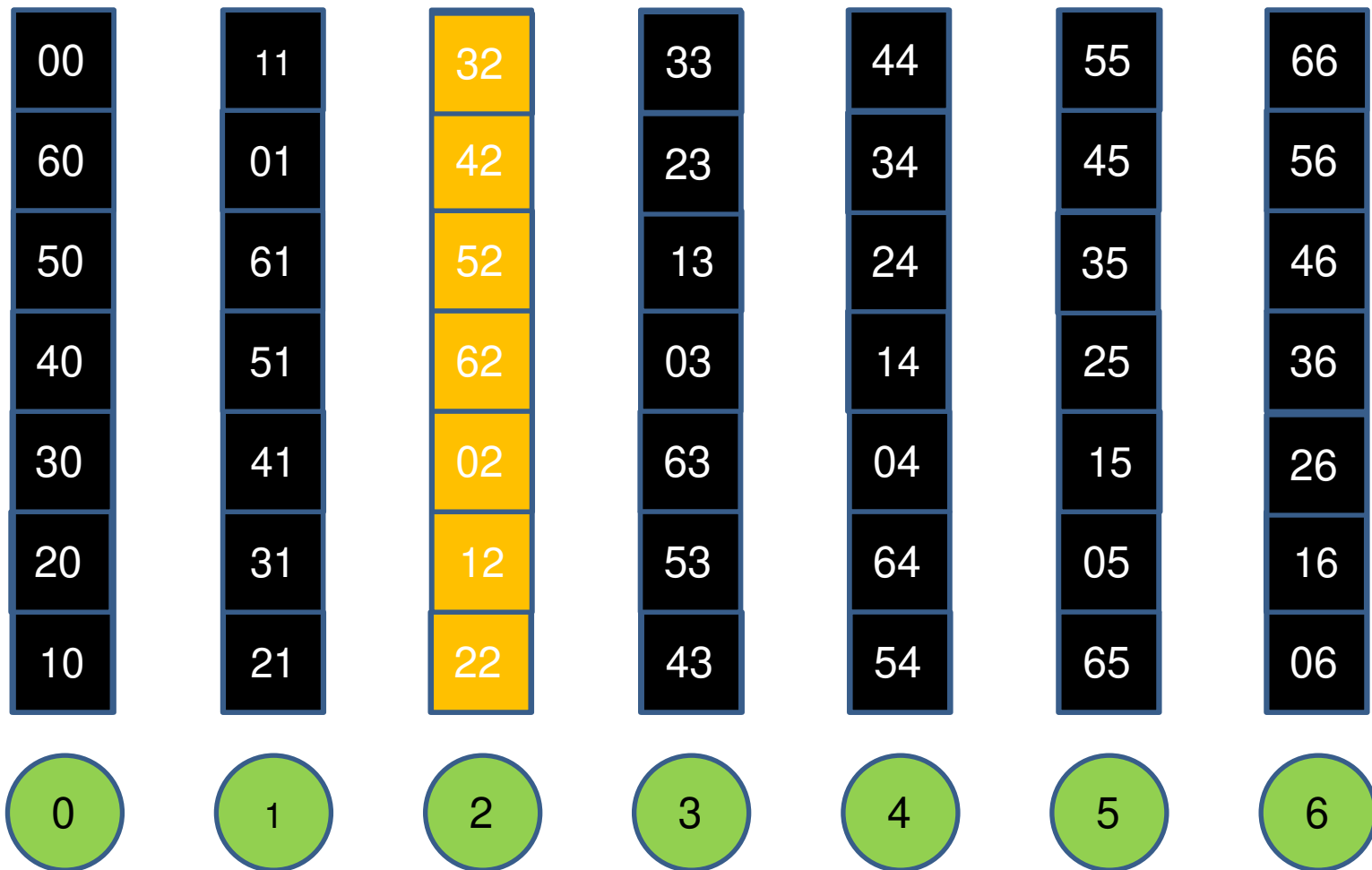


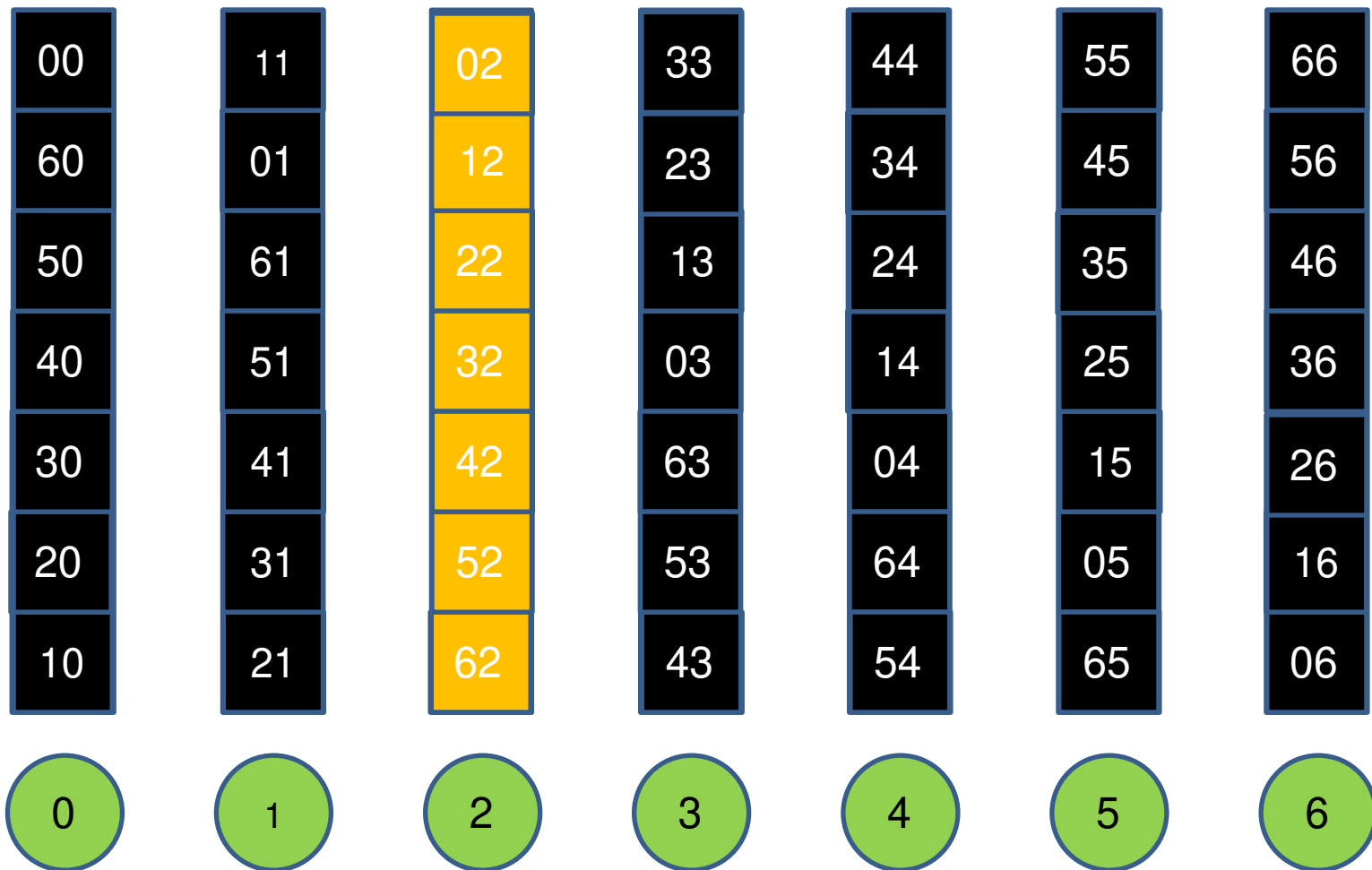
Step 3:

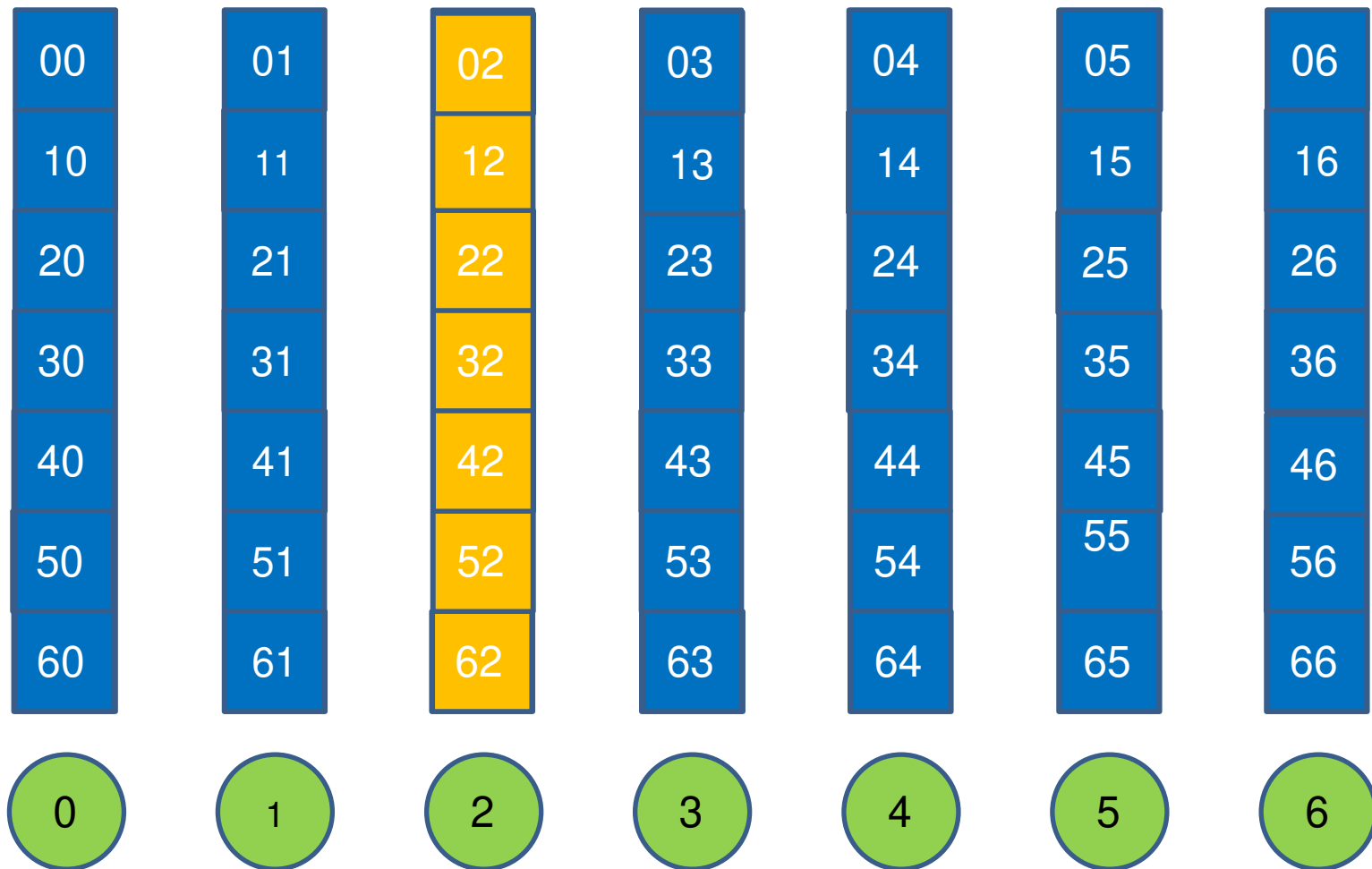
Local reverse and rotate downwards, processor  $i$  by  $i+1$  positions











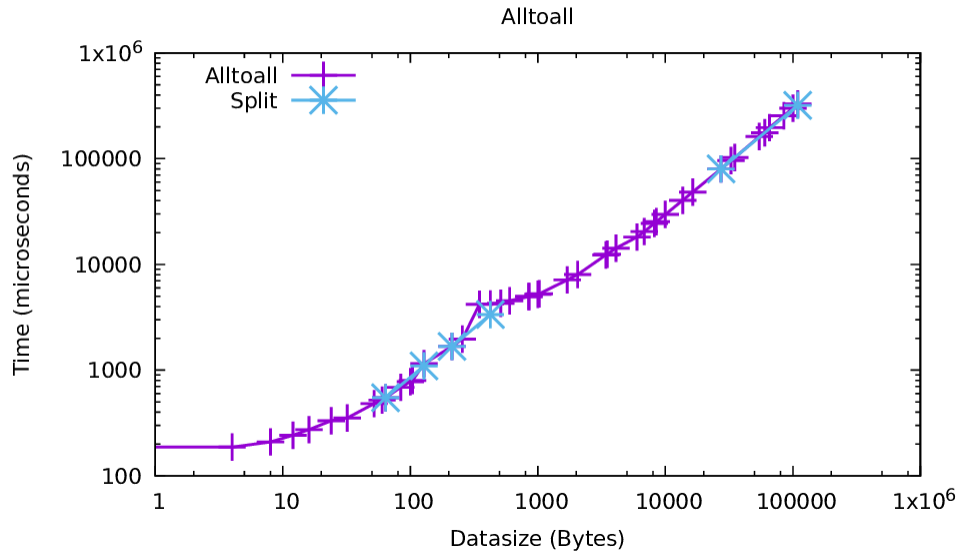
Total time:

$\text{ceil}(\log p)$  communication rounds (Step 2),  $O(m)$  copy-shift (Step 1),  $O(m)$  reverse-shift (Step 3)

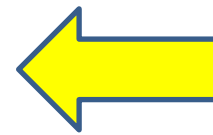
$$\text{Talltoall}(m) = \text{ceil}(\log p)(\alpha + \beta \text{floor}(m/2)) + O(m)$$

Algorithm is used in some MPI libraries (mpich, mvapich, OpenMPI) for small  $m$ .

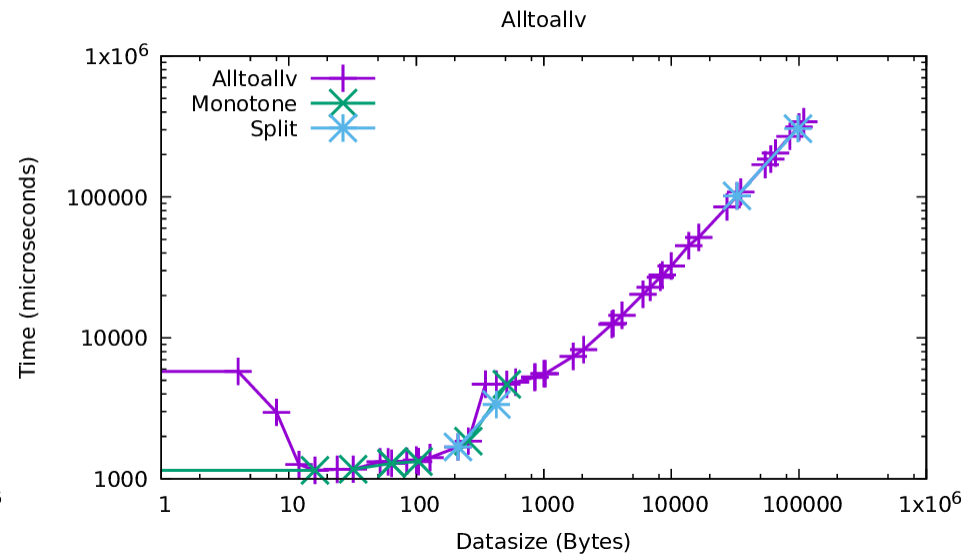
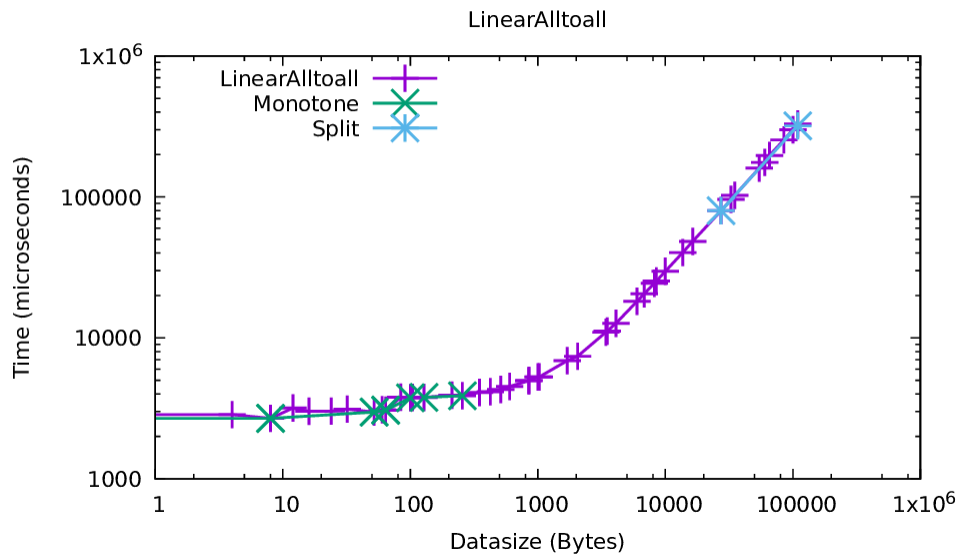
**Drawback:** Many copy/pack-unpack operations (Steps 1 and 3)

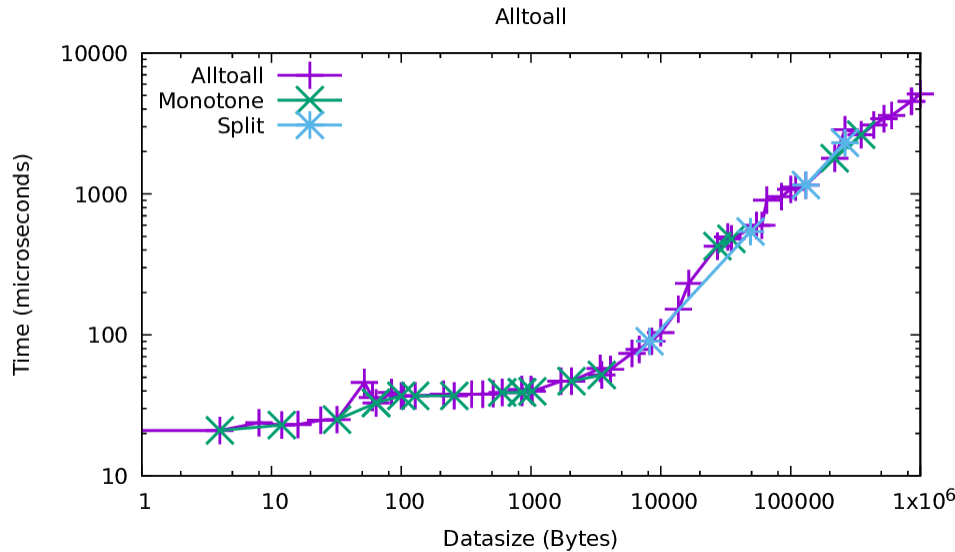


Hydra, Intel MPI 2018, 36x32  
processes,  
MPI\_COMM\_WORLD

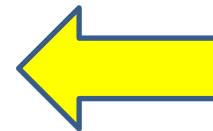


Employs better than  
linear algorithm

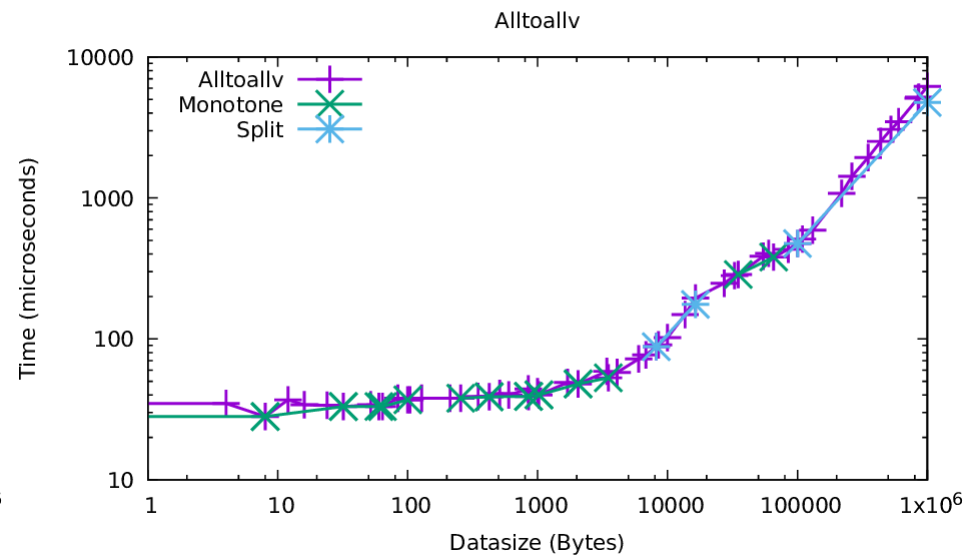
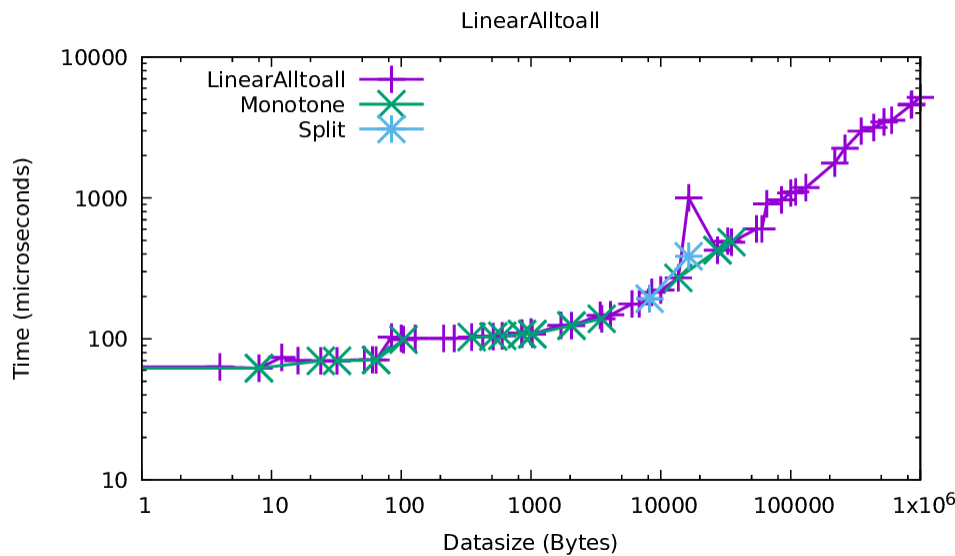


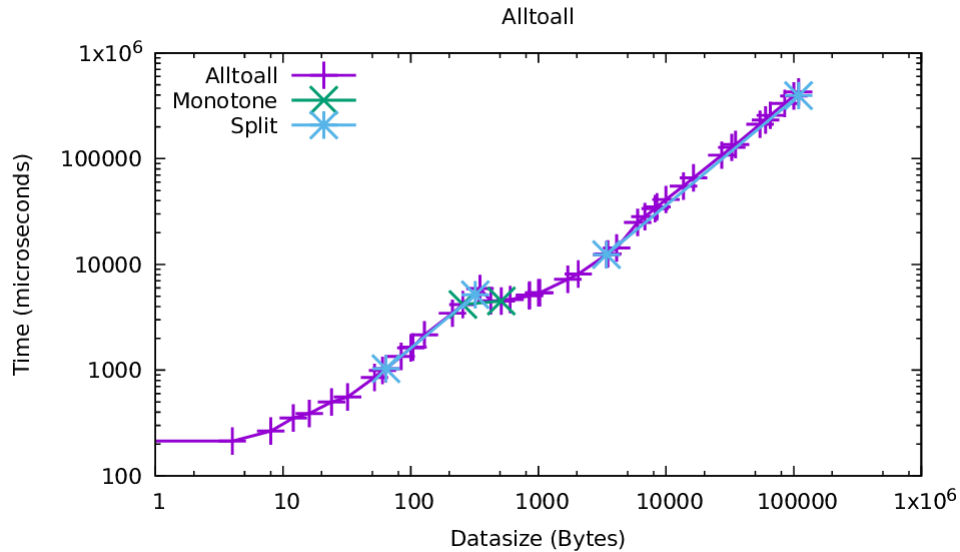


Hydra, Intel MPI 2018, 36x1  
processes,  
MPI\_COMM\_WORLD

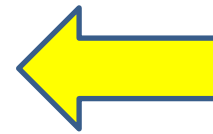


Employs better than  
linear algorithm

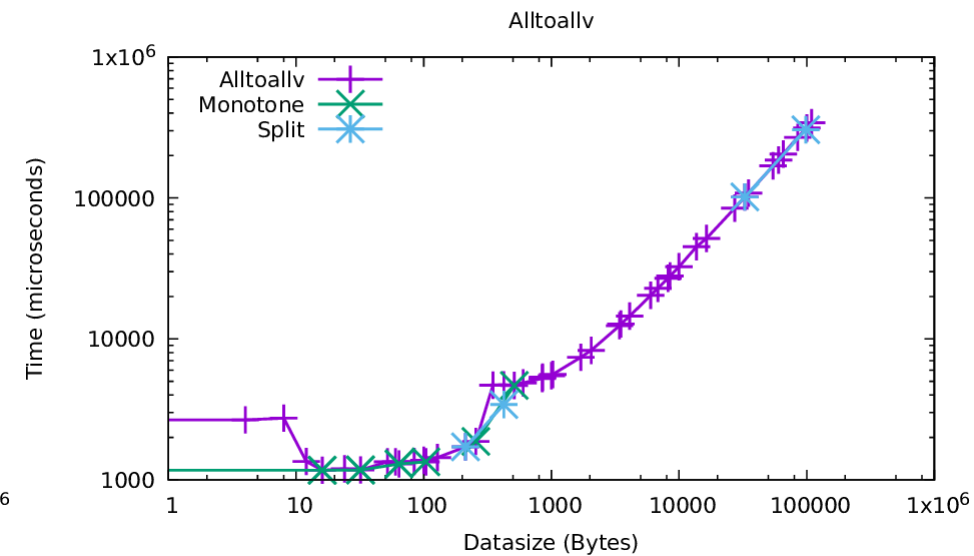
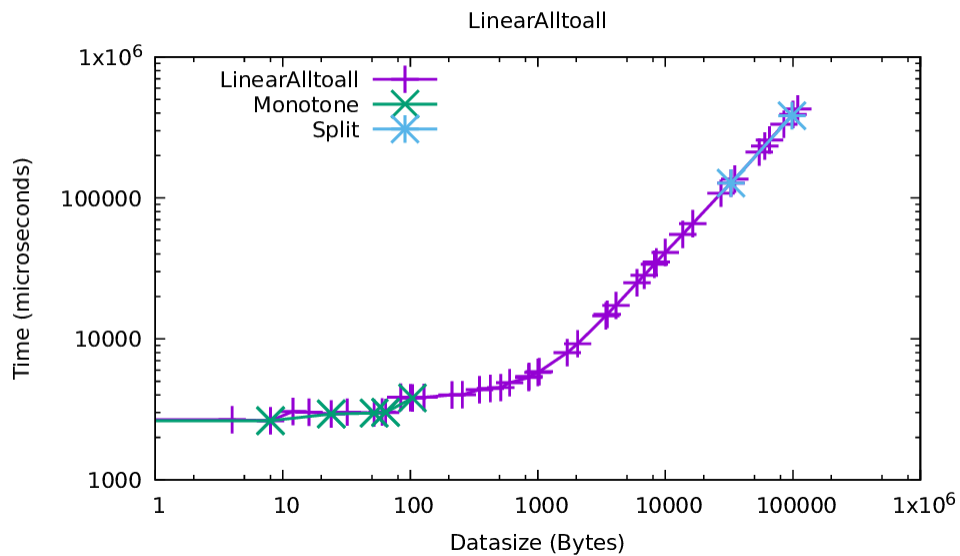




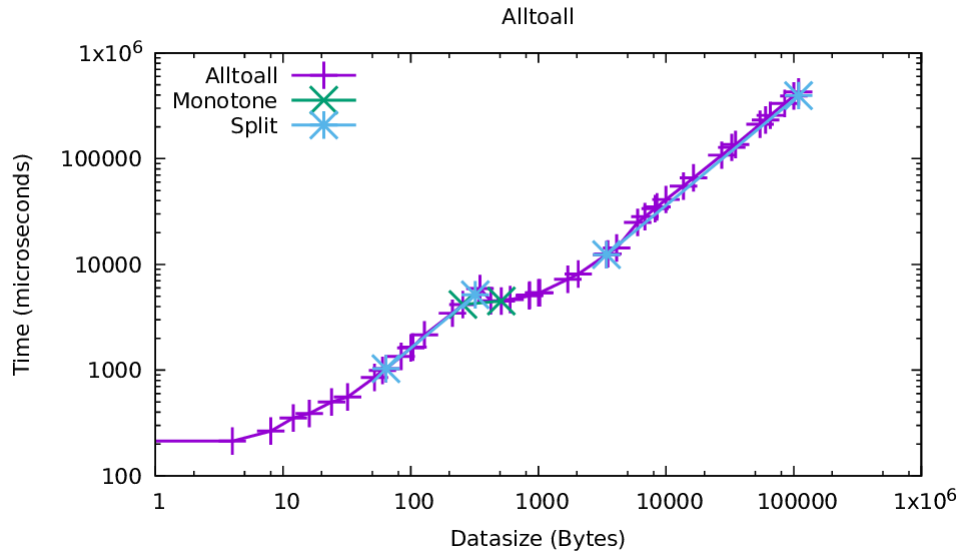
Hydra, Intel MPI 2018, 36x32 processes, cyclic comm



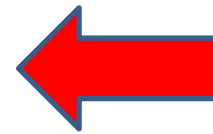
Employs better than linear algorithm



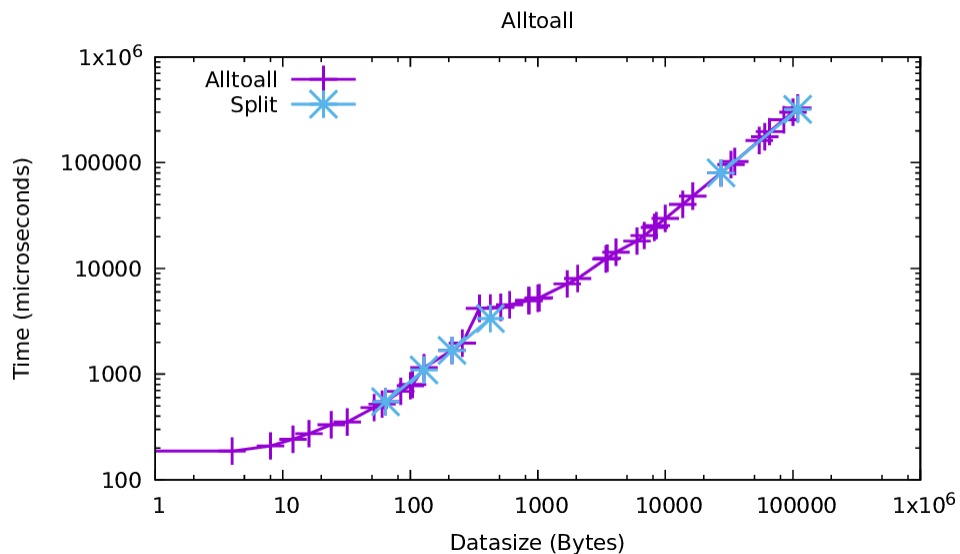




Hydra, Intel MPI 2018, 36x32 processes, cyclic comm



Somewhat/too sensitive to communicator/process placement?



Hydra, Intel MPI 2018, 36x32 processes,  
MPI\_COMM\_WORLD

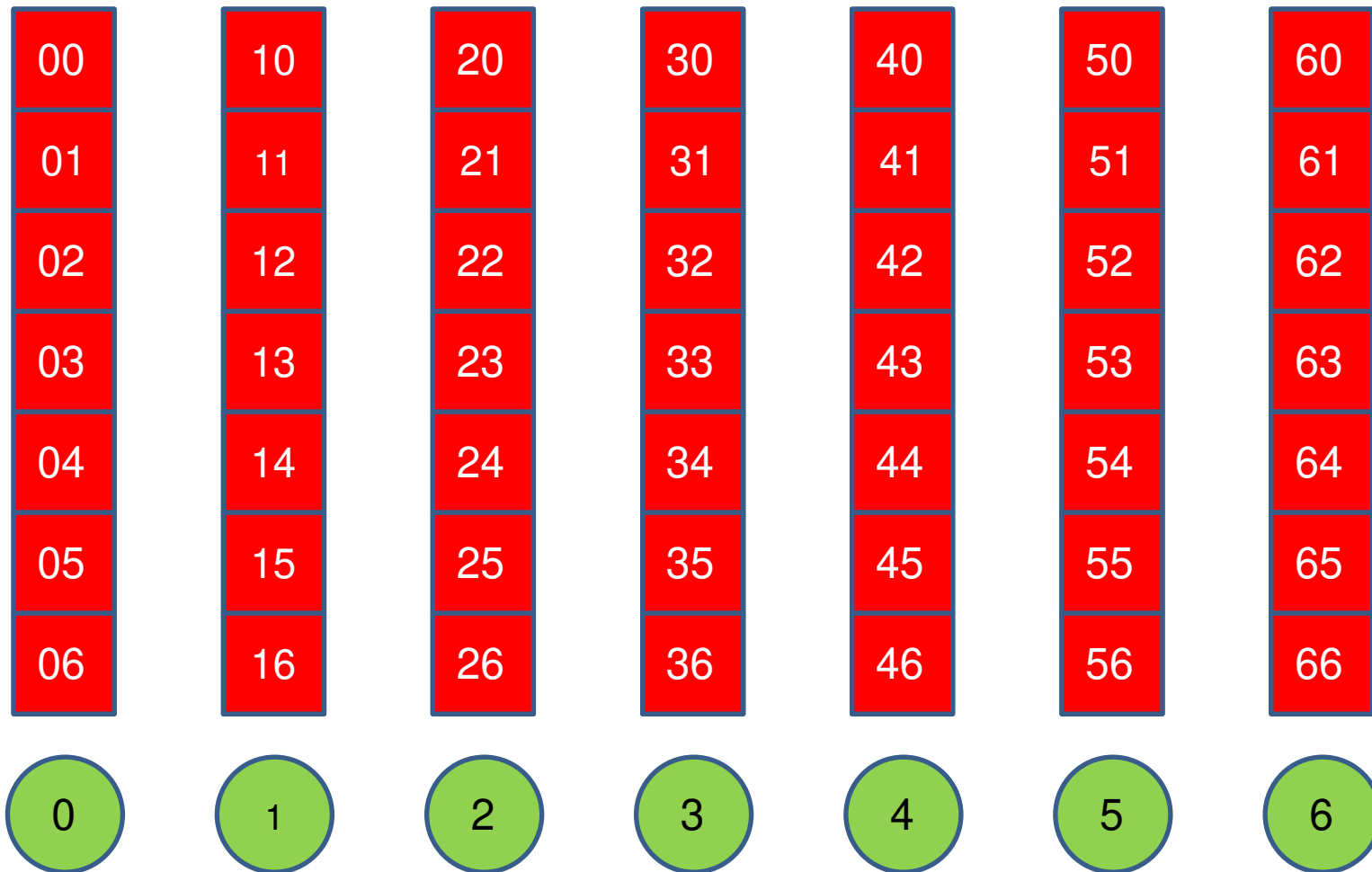
**Note :**

Step 1 and Step 3 can be eliminated: Reverse communication direction, shift implicitly by communication step; MPI derived datatypes to avoid explicit packing/unpacking

Jesper Larsson Träff, Antoine Rougier, Sascha Hunold:  
Implementing a classic: zero-copy all-to-all communication with  
mpi datatypes. ICS 2014: 135-144

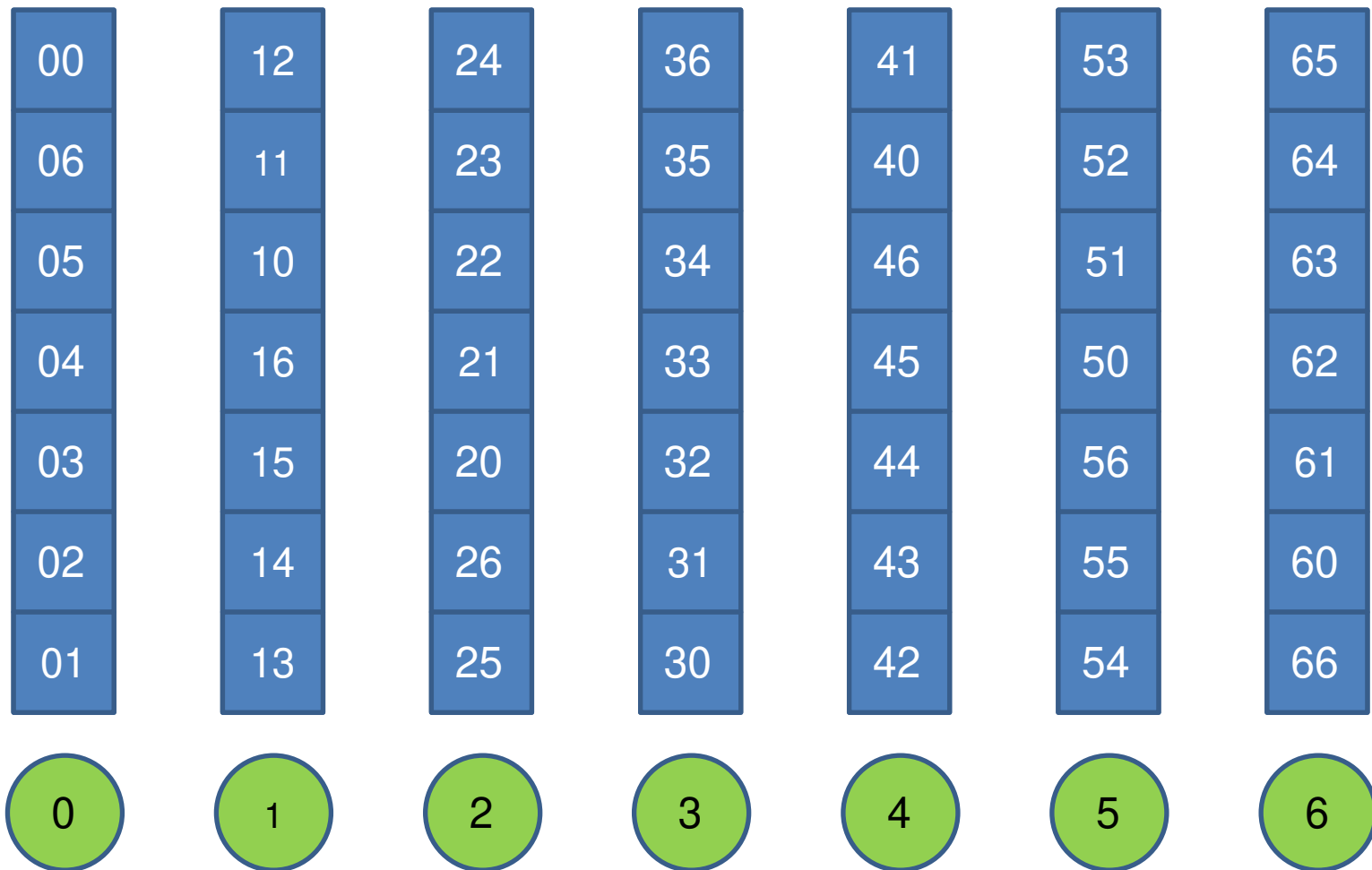
Next example shows how to eliminate Step 3

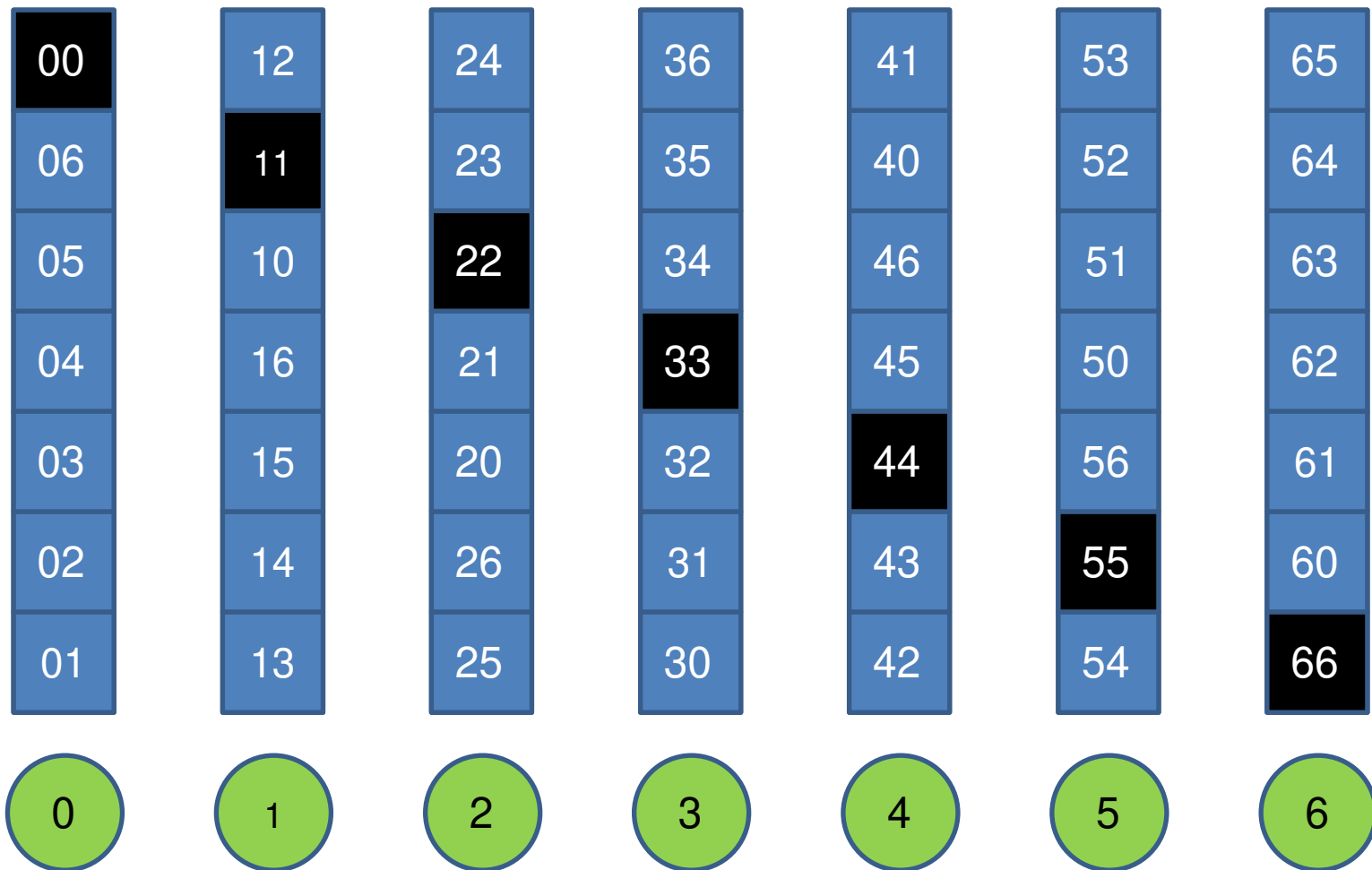
MPI sendbuf



Step 1:

Copy **sendbuf** blocks into **recvbuf** in reverse order





Step 2:  $\text{ceil}(\log p)$  rounds

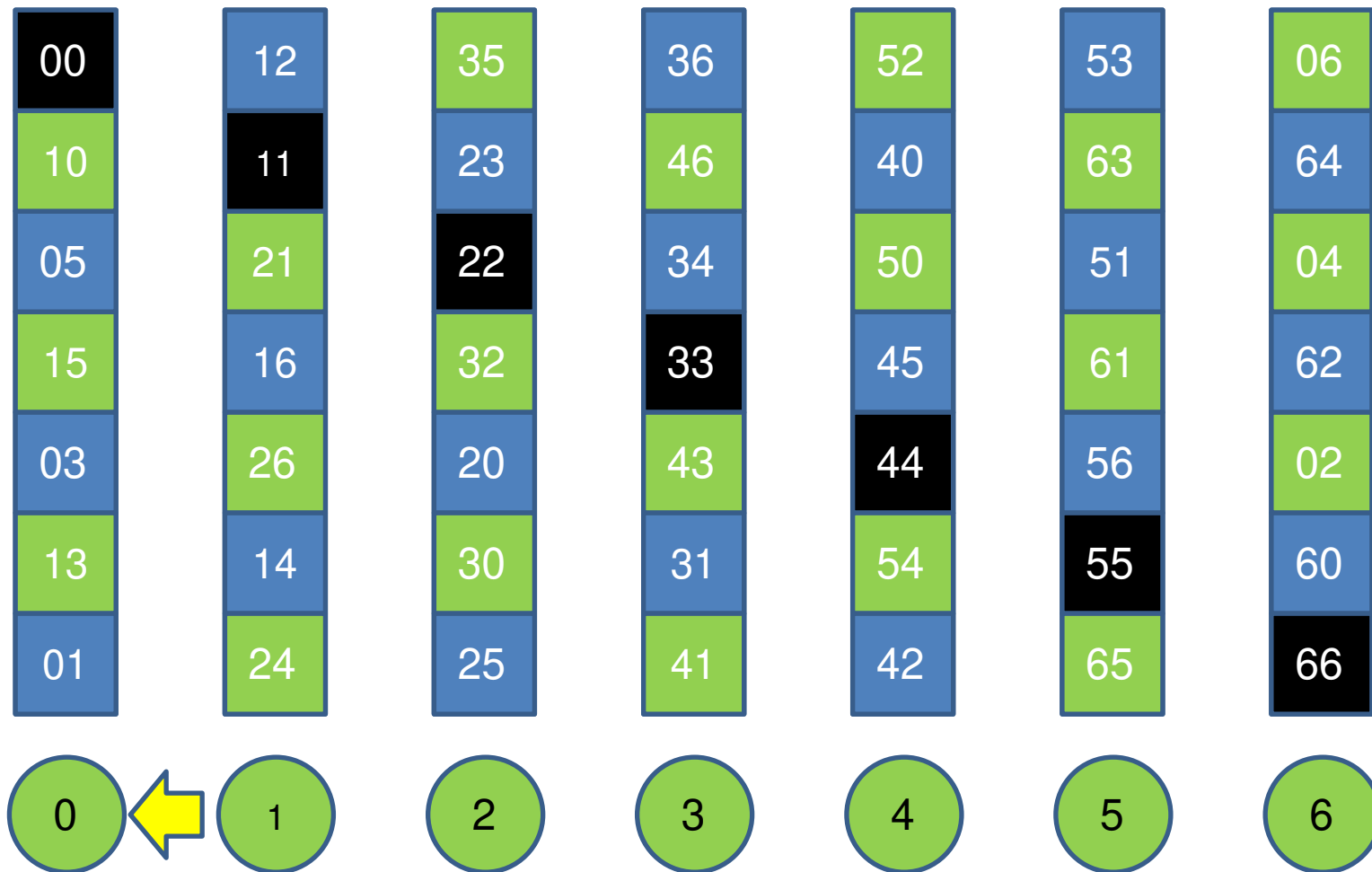
Round  $k$ :

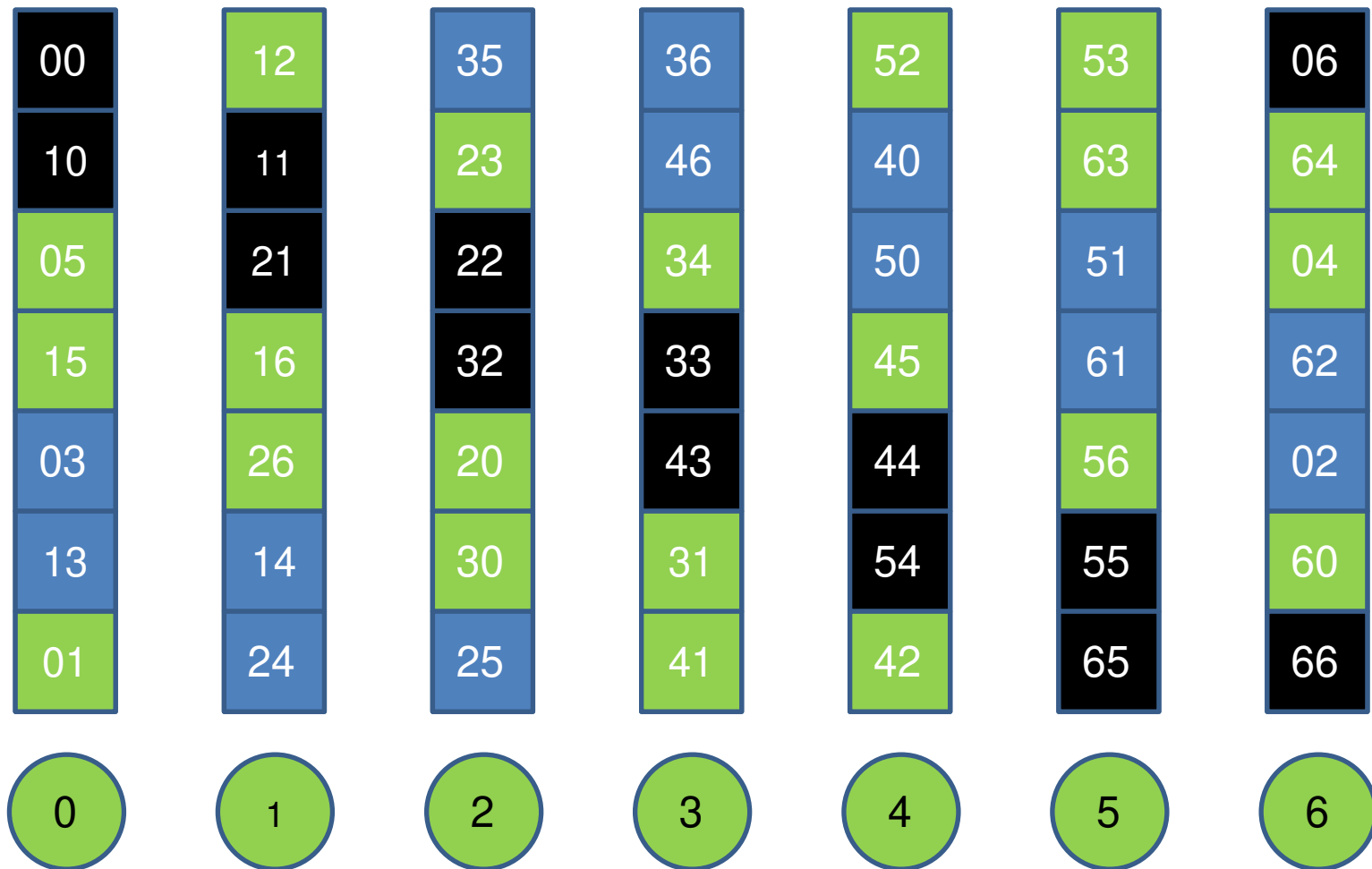
For process  $i$ , all blocks destined to a processor with bit  $k=1$  are sent to processor  $(i - 2^k) \bmod p$

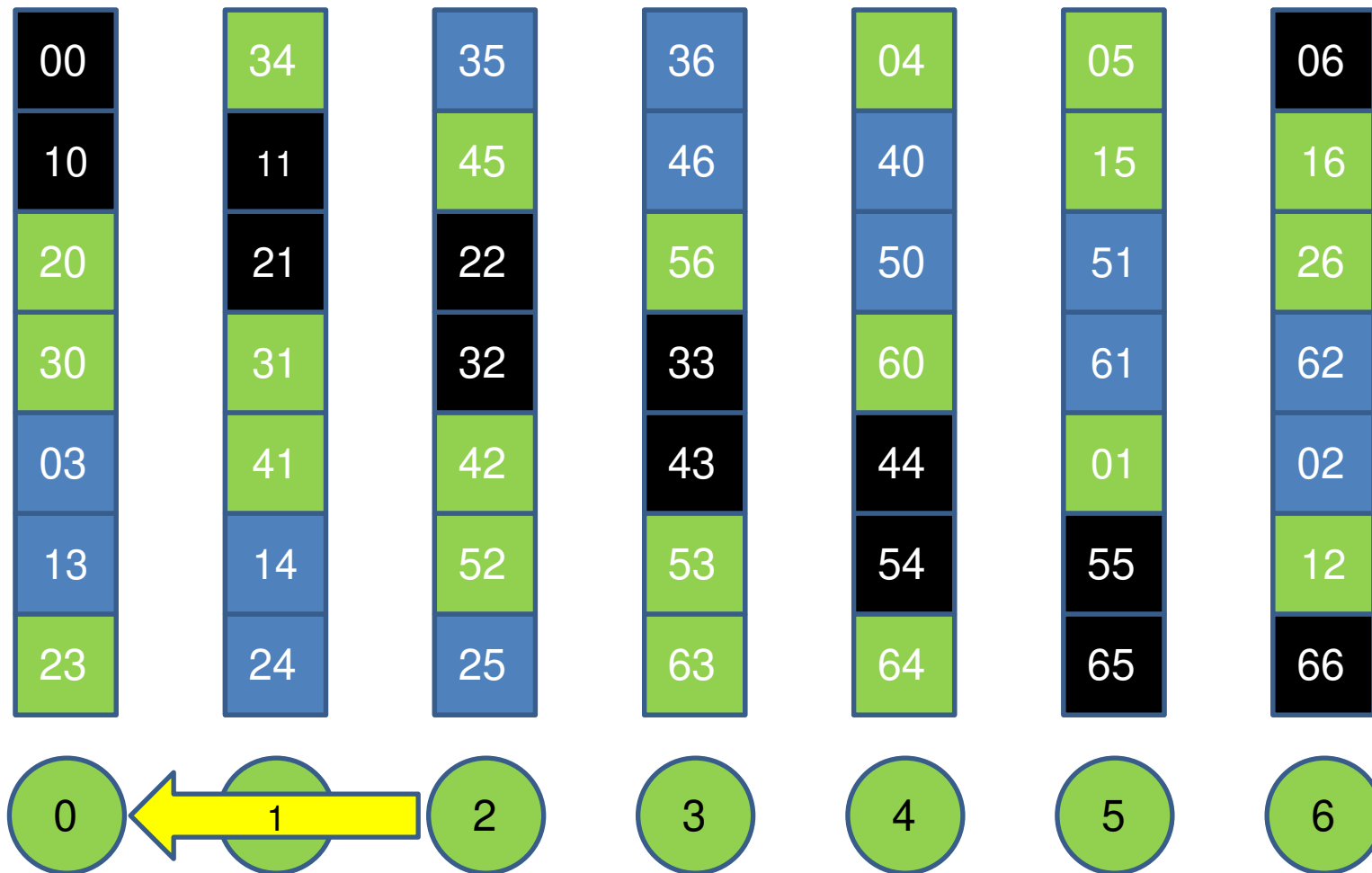
Block  $j$  is in row  $(i+j) \bmod p$ , block  $j$  is sent if  $j \text{ XOR } 2^{k-1}$

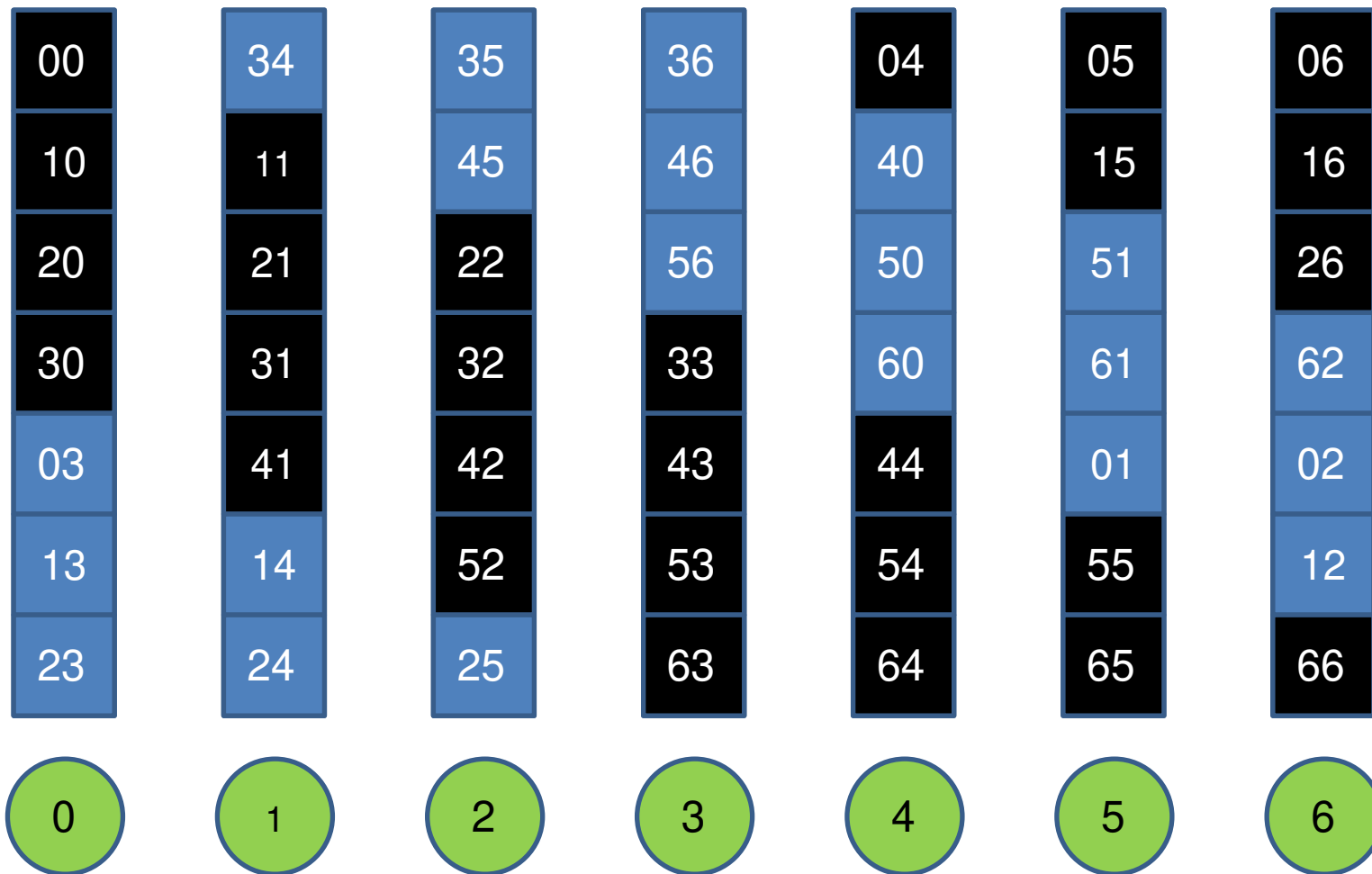


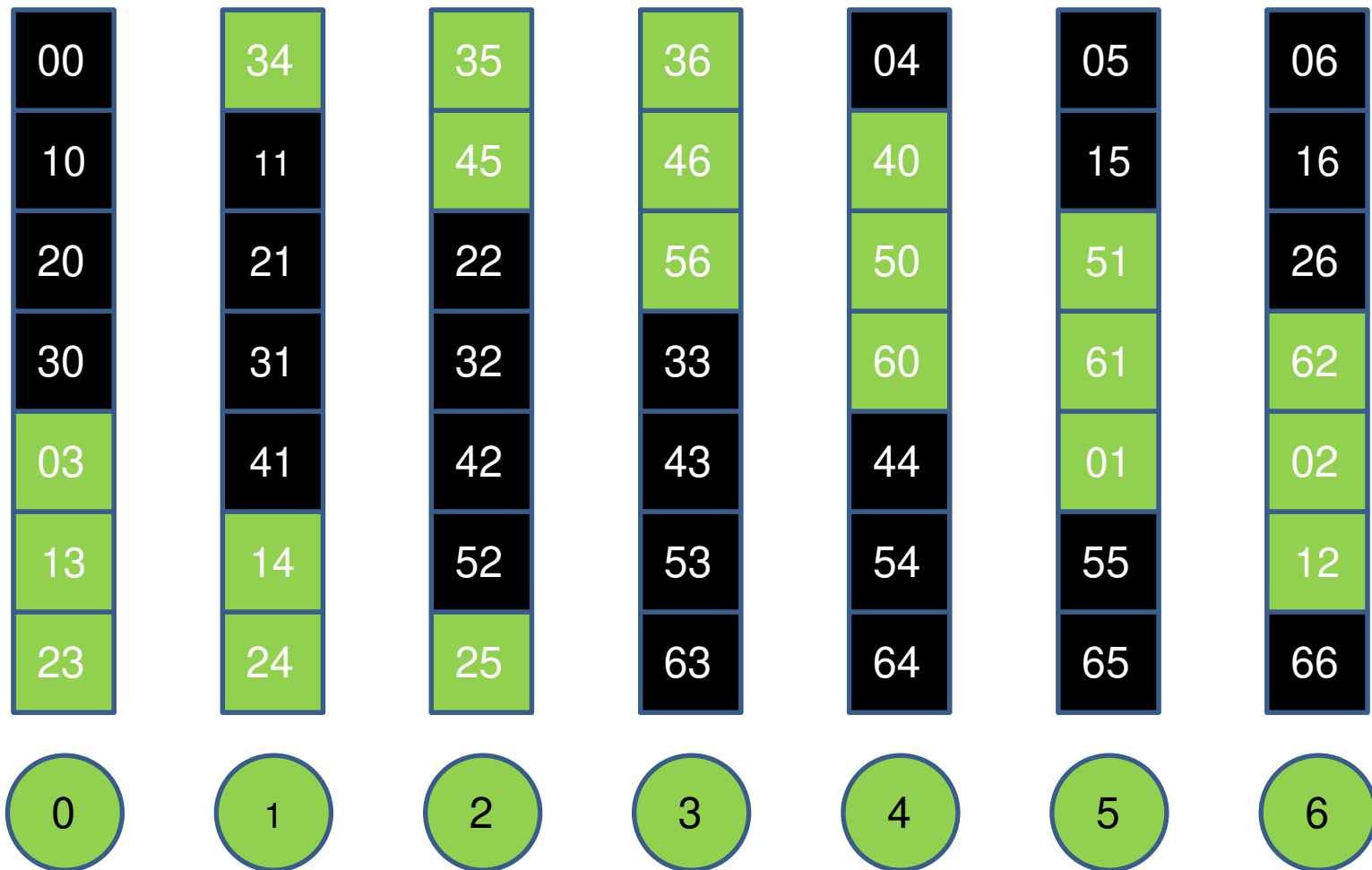


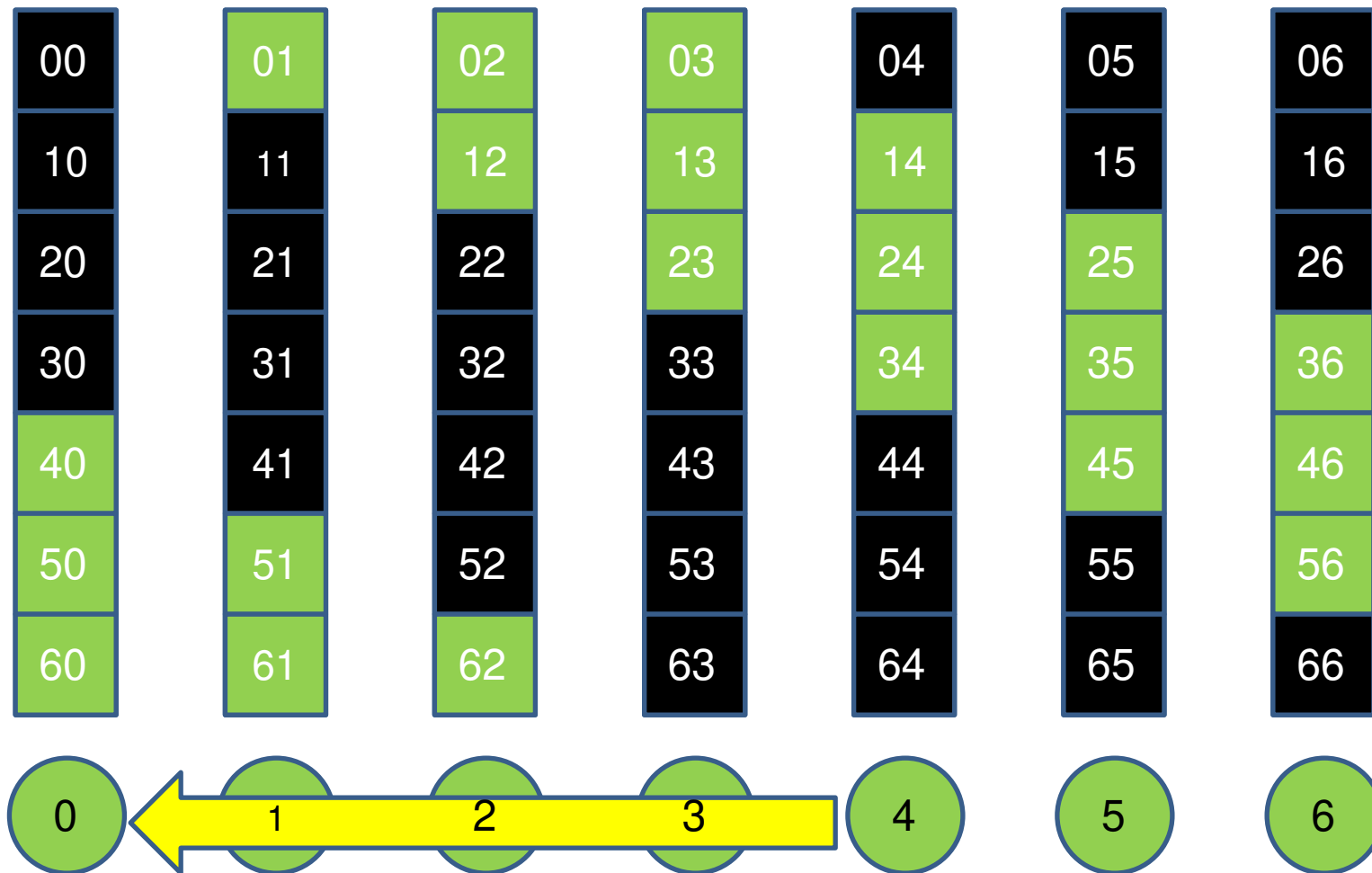


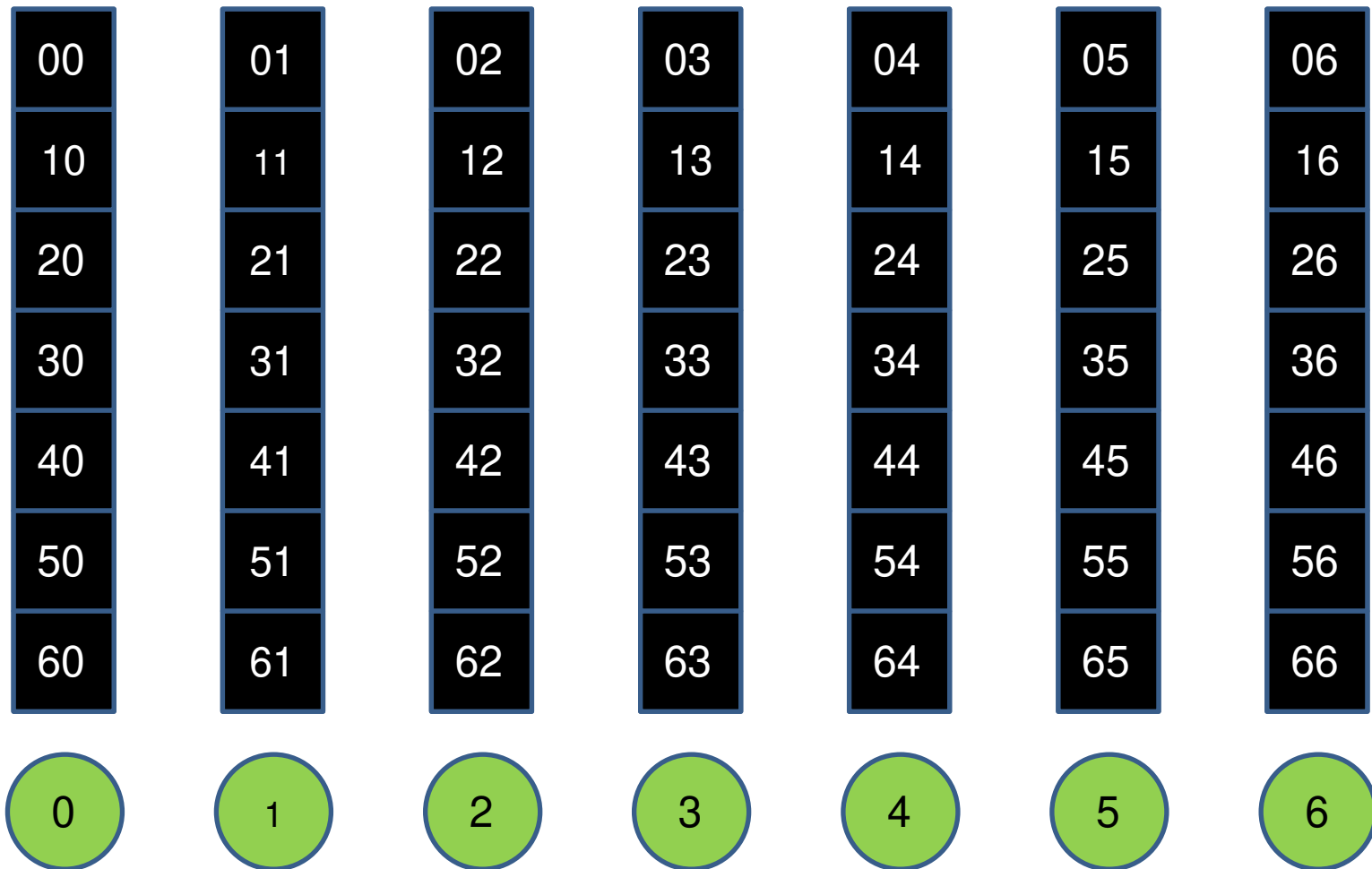












Done! (no step 3)

Step 1 and all intermediate pack/unpack eliminated by using MPI derived datatype and double buffering per block (see paper)



## Bandwidth/Latency trade-offs for alltoall communication

In fully connected, k-ported, bidirectional networks:

### Theorem:

- Any allgather algorithm requires at least  $\text{ceil}(\log_{k+1} p)$  communication rounds in the k-ported communication model
- Any alltoall algorithm requires at least  $\text{ceil}(\log_{k+1} p)$  communication rounds in the k-ported communication model

### Theorem:

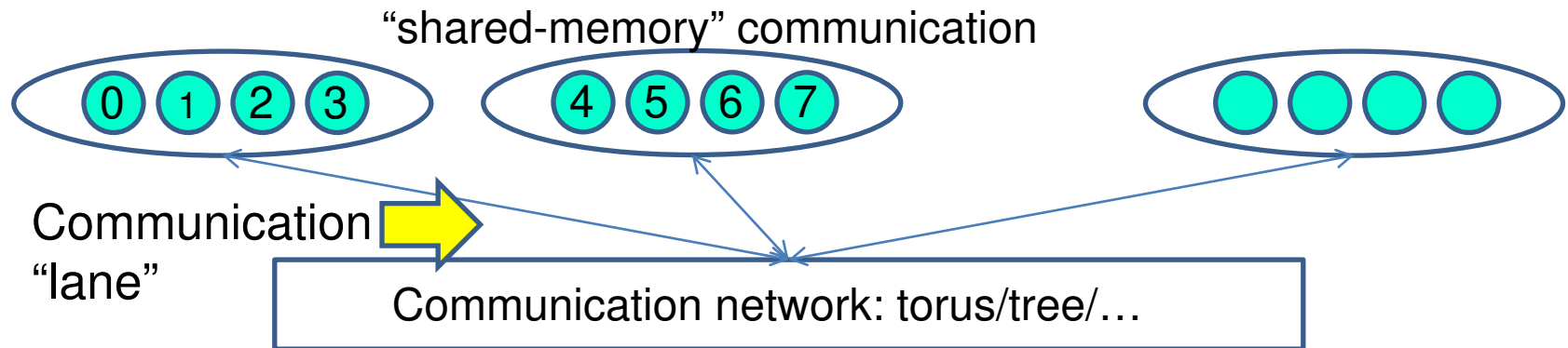
- Any alltoall algorithm that transfers exactly  $(p-1)/p$  m Bytes requires at least  $p-1$  communication rounds
- Any alltoall algorithm that uses  $\text{ceil}(\log_{k+1} p)$  communication rounds must transfer at least  $m / ((k+1) \log_{k+1} p)$  Bytes

## Trade-off results from

J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, D. Weathersby:  
Efficient Algorithms for All-to-All Communications in Multiport  
Message-Passing Systems. IEEE Trans. Parallel Distrib. Syst.  
8(11): 1143-1156 (1997)

## Hierarchical communication systems

Many/most HPC systems have a (two-level) hierarchical communication system, e.g.



All processors (cores) inside node share inter-node communication bandwidth to network

## Hierarchical SMP/multi-core cluster:

- Intra-node communication via shared memory: Somewhat like fully connected, 1-ported, bidirectional
- Inter-node communication via network: bidirectional communication with network characteristics, e.g., torus, fat tree (hierarchical), fully connected ( **rare** ), ... If more than one communication “lane” (connections to network, network switches), up to k processes can communicate out of node concurrently, and/or bandwidth for a single process can be increased by a factor of k, where k is the number of “lanes”

Goal: Exploit full network bandwidth out of/into nodes

Linear-array/ring algorithms for MPI collectives can often be embedded efficiently into hierarchical network

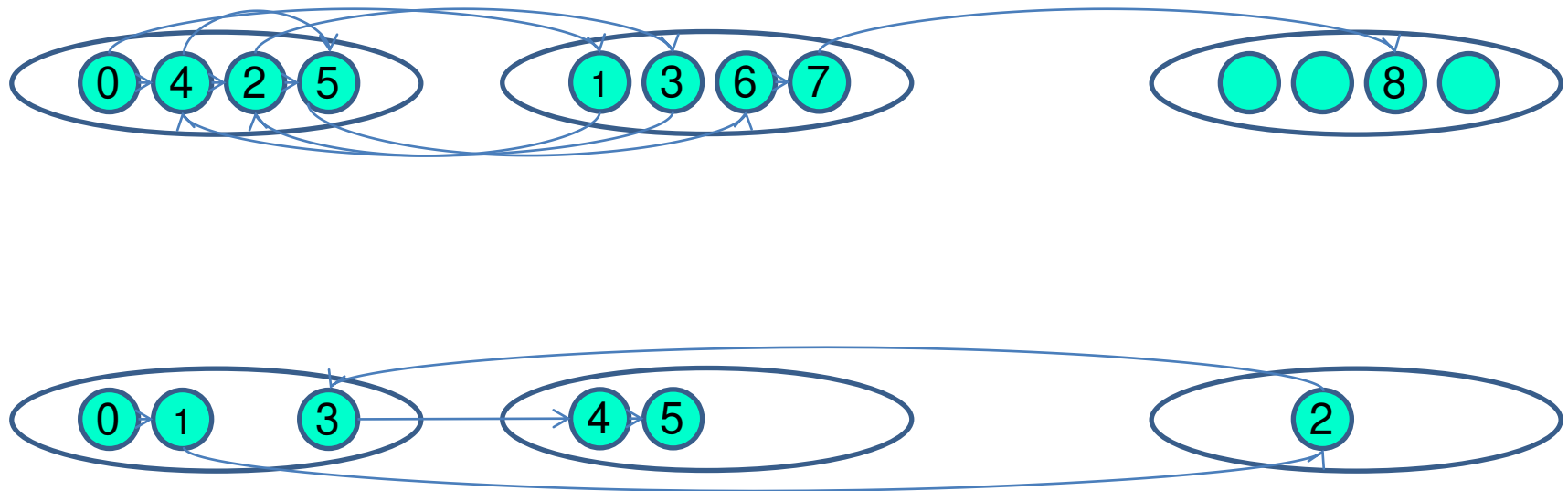


No conflicts on inter-node communication, so all linear algorithms can be used

Example:

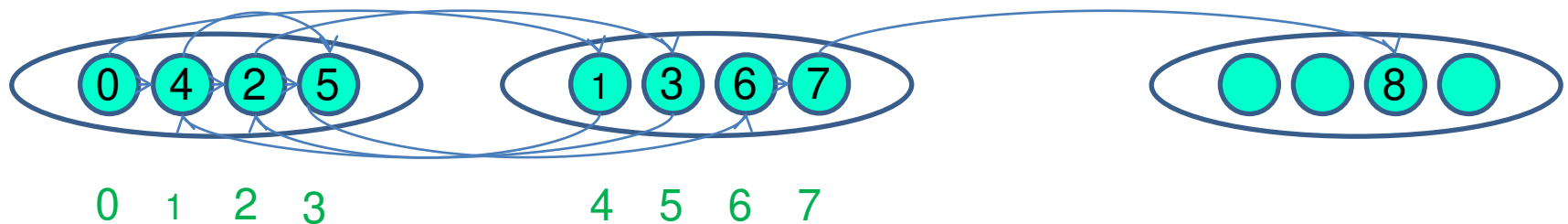
Jesper Larsson Träff, Andreas Ripke, Christian Siebert, Pavan Balaji, Rajeev Thakur, William Gropp: A Pipelined Algorithm for Large, Irregular All-Gather Problems. IJHPCA 24(1): 58-68 (2010)

**But:** MPI allows creation of arbitrary subsets of processes (communicators), can lead to contention/serialization on inter-node network



```
MPI_Comm_split(comm,color=0,key=random,&newcomm);
```

For “commutative” collectives (Broadcast, Allgather, Gather/Scatter): Use linear, virtual process numbering



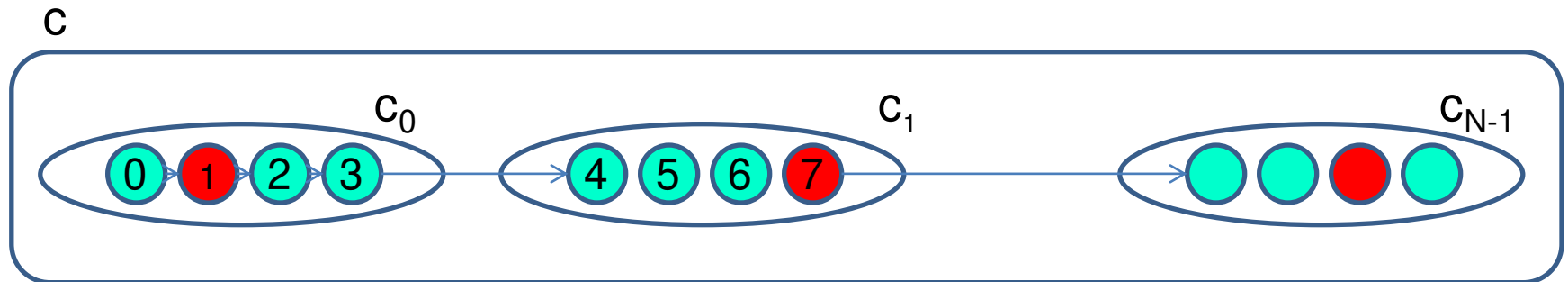
Reduction: If operator is commutative, use linear, virtual ordering. Scan/Exscan: Reordering not possible

**Complication:** Data reordering may be required at different steps (e.g., MPI\_Gather must receive blocks in rank order)

## Hierarchical collectives

$c$ : communicator over  $N$  nodes

- $c_0, c_1, c_2, \dots, c_{N-1}$ : partition of  $c$  into subcommunicators, each  $c_i$  fully on node,  $\text{size}(c_i) = n_i$  ( $=n$  for homogeneous communicator)
- $C$ : subcommunicator of  $c$  with one process of each  $c_i$



```
MPI_Comm_split_type(c, MPI_COMM_TYPE_SHARED, key,
                    info, &ci);
```



Bcast(c):

1. Bcast(C)
2. In parallel: Bcast( $c_i$ )

**Note** : Rounds at least  
 $\text{ceil}(\log n) + \text{ceil}(\log N) \geq$   
 $\text{ceil}(\log n + \log N) = \text{ceil}(\log p)$   
 since  $n = p/N$

**but**  $\text{ceil}(\log n) + \text{ceil}(\log N) \leq$   
 $\text{ceil}(\log p) + 1$ , **at most one round**  
**off from optimal**

Reduce(c):

1. In parallel: Reduce( $c_i$ )
2. Reduce(C)

If all the processes in  $c_i$  are in order of c

Total amount of data out of/into node: m

**Note** : Time for Bcast( $c_i$ ) may differ since size( $c_i$ ) may be different from size( $c_j$ )

Allreduce(c):

1. In parallel: Reduce( $c_i$ )
2. Allreduce(C)
3. In parallel: Bcast( $c_i$ )

One  $\text{ceil}(\log n)$  operation too much,  $\text{ceil}(\log n) + \text{ceil}(\log N) + \text{ceil}(\log n)$

If all the processes in  $c_i$  are in order of  $c$

Allgather(c):

1. In parallel: Gather( $c_i$ )
2. Allgatherv(C)
3. Bcast( $c_i$ )

**Note** : To solve regular problem, algorithm for irregular problem is needed when  $\text{size}(c_i) \neq \text{size}(c_j)$

Similar observations for Gather/Scatter, Scan/Exscan

Alltoall(c):

1. In parallel: Gather( $c_i$ )
2. Alltoallv(C)
3. In parallel: Scatter( $c_i$ )

**Note** : To solve regular problem, algorithm for irregular problem is needed when  $\text{size}(c_i) \neq \text{size}(c_j)$

Scan(c):

1. In parallel: Reduce( $c_i$ )
2. Exscan(C)
3. In parallel: Scan( $c_i$ )

If all the processes in  $c_i$  are in order of c

Writing own, hierarchical collectives, avoiding explicit data reordering

Jesper Larsson Träff, Antoine Rougier: MPI Collectives and Datatypes for Hierarchical All-to-all Communication.

EuroMPI/ASIA 2014: 27

Jesper Larsson Träff, Antoine Rougier: Zero-copy, Hierarchical Gather is not possible with MPI Datatypes and Collectives.

EuroMPI/ASIA 2014: 39

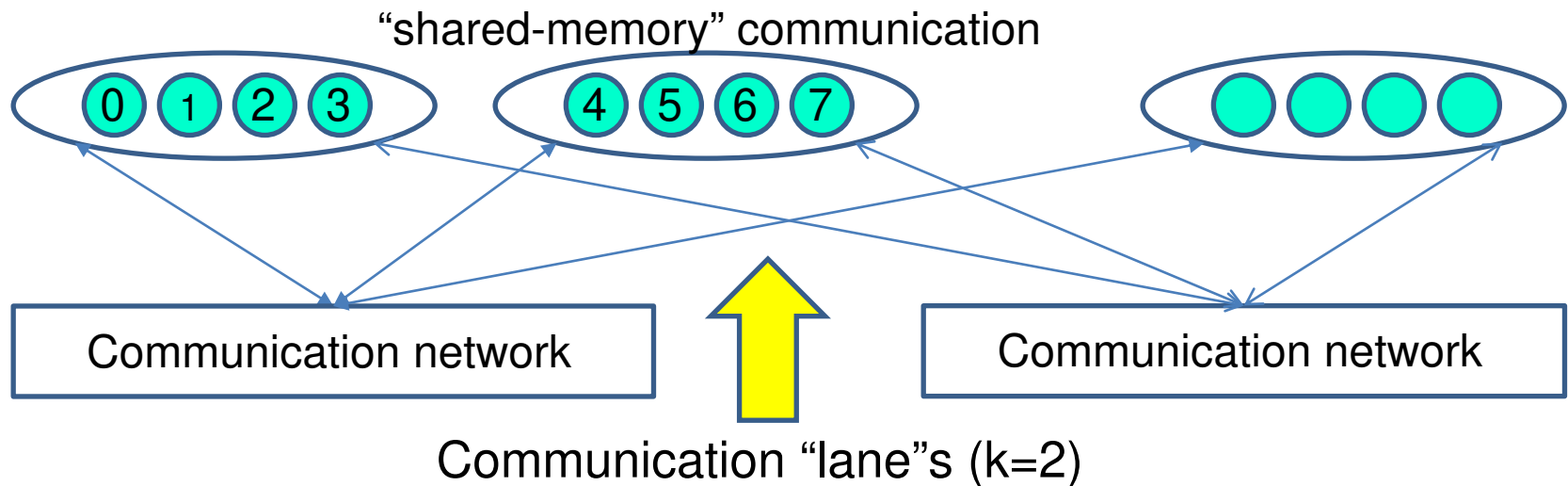
A good MPI library internally implements collective operations in a hierarchical fashion (shared memory part, network part, ...)

## Hierarchical collectives

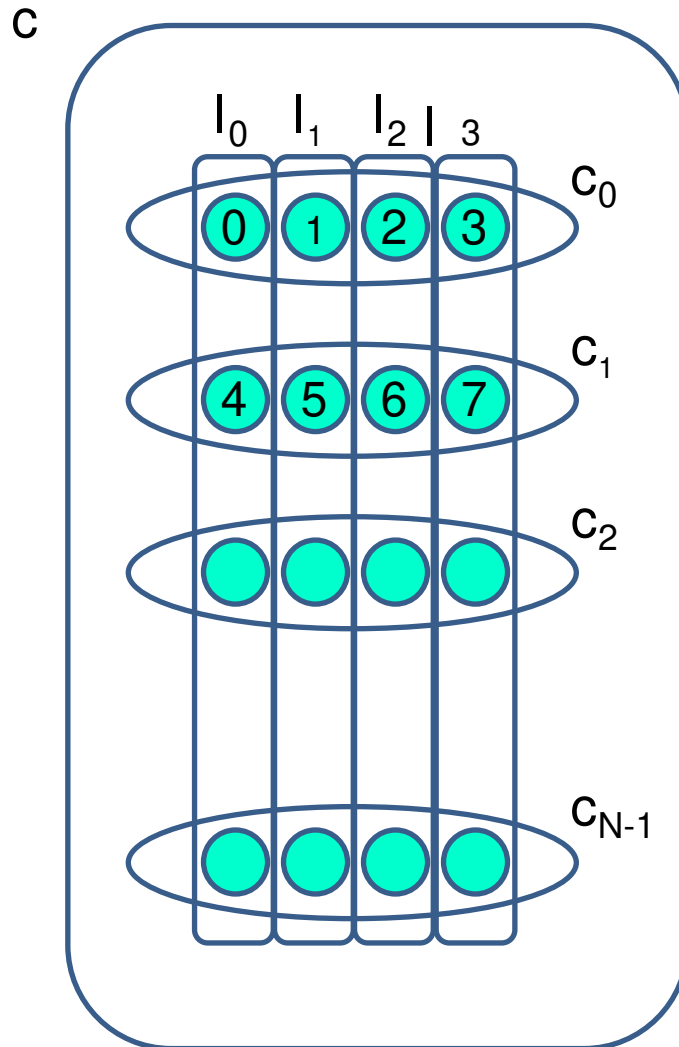
### Goals:

- Avoid contention on connection(s) to inter-node network
- Avoid sending/receiving same data more than once per node
- Achieve same number of communication rounds as best, non-hierarchical algorithm
- Achieve same asymptotic bandwidth ( $\beta$ -term) as best, non-hierarchical algorithm

## Hierarchical, full-lane collectives



k-lane model: k processors per node can communicate concurrently with full network bandwidth (with more than k: serialization, linear, proportional slowdown)



Full-lane collectives:  
Split communicator  $c$  into  $c_i$   
node communicators and  $l_i$   
“lane” communicators

Works when  $n_i = n$  for all nodes  
( $c$  is a regular communicator )

Bcast(c):

1. Scatter( $c_{\text{root}}$ )
2. In parallel over lanes: Bcast( $l_i$ ) with blocksize  $m/n$
3. In parallel on nodes: Allgather( $c_i$ )

One  $\text{ceil}(\log n)$  operation too much,  $\text{ceil}(\log n) + \text{ceil}(\log N) + \text{ceil}(\log n)$

Reduce(c):

1. In parallel on nodes: Reduce-scatter( $n_i$ )
2. In parallel over lanes: Reduce( $c_i$ ) with vector size  $m/n$
3. Gather( $c_{\text{root}}$ )

Total amount of data out of/into node:  $m$



Allreduce(c):

1. Reduce-scatter( $c_i$ )
2. Allreduce( $l_i$ )
3. Allgather( $c_i$ )

Rounds at most  $2\text{ceil}(\log n) + 2\text{ceil}(\log N) \leq \text{ceil}(\log p) + 2$ . Data per process at most  $2m$  (check!)

Same as best known homogeneous Allreduce

Allgather(c):

1. Allgather( $l_i$ )
2. Allgather( $c_i$ )

**Note** : Rounds at least  $\text{ceil}(\log n) + \text{ceil}(\log N) \geq \text{ceil}(\log n + \log N) = \text{ceil}(\log p)$  since  $n=p/N$

**but**  $\text{ceil}(\log n) + \text{ceil}(\log N) \leq \text{ceil}(\log p) + 1$ , **at most one round off from optimal**

Gather(c):

1. Gather( $l_i$ )
2. Gather( $c_{\text{root}}$ )

Scatter(c):

1. Scatter( $c_{\text{root}}$ )
2. Scatter( $l_i$ )

MPI implementation: Use derived datatypes to avoid copying into intermediate buffers

Alltoall(c):

1. Alltoall(l<sub>i</sub>)
2. Alltoall(c<sub>i</sub>)

Reduce-scatter(c):

1. Reduce-scatter(c<sub>i</sub>)
2. Reduce-scatter(l<sub>i</sub>)

MPI implementation: Use derived datatypes to avoid copying into intermediate buffers

Experimental question: How do hierarchical and full-lane collective implementations compare to standard, homogeneous algorithms? How do hierarchical and full-lane collective implementations compare to the collectives in common MPI libraries?

Themes for Master theses

Jesper Larsson Träff, Sascha Hunold: Decomposing MPI Collectives for Exploiting Multi-lane Communication. CLUSTER 2020: 270-280

Jesper Larsson Träff: Decomposing Collectives for Exploiting Multi-lane Communication. CoRR abs/1910.13373 (2019)

Code available at

[www.par.tuwien.ac.at/Downloads/TUWMPI/tuw\\_lanecoll.zip](http://www.par.tuwien.ac.at/Downloads/TUWMPI/tuw_lanecoll.zip)

## Another lecture: Algorithms for 2d-ported, d-dim. tori

Lower bounds to meet:

- Diameter
- Bisection
- Edge congestion

Algorithms look different, tricky, still open problems (alltoall)

First approximation (van de Geijn): Use combinations of linear ring algorithms along the dimensions

## Algorithms summary (communication costs only)

p-processor linear array

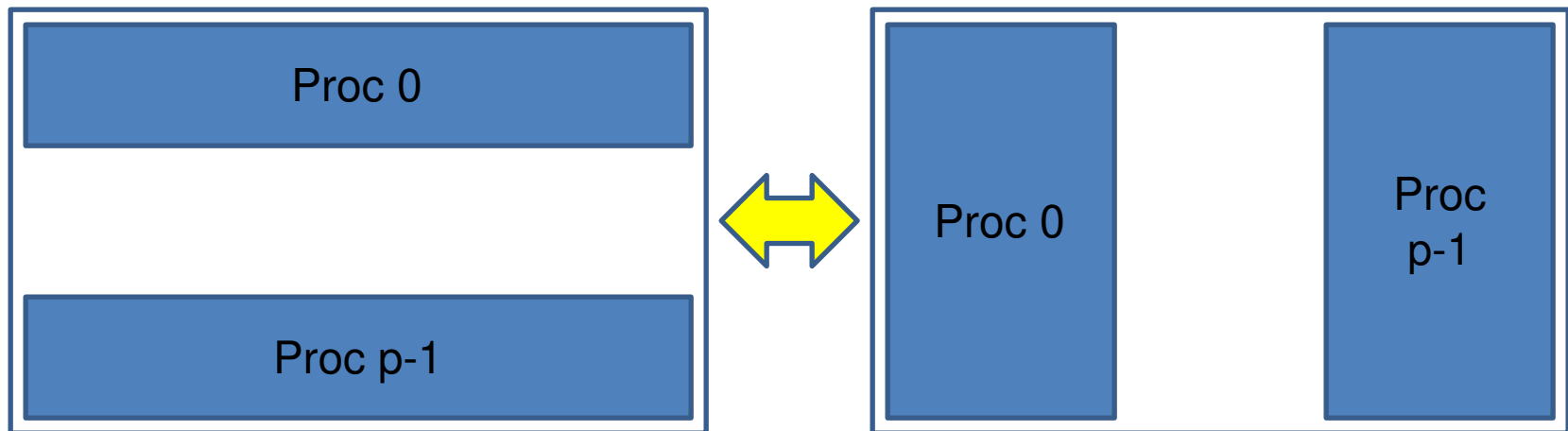
Collective	$T(m)$ in linear cost model	
Gather/Scatter	$(p-1)\alpha + (p-1)/p \beta m$	
Allgather	same	
Reduce-scatter	same	
Bcast/Reduce	$(p-2)\alpha + o(pm) + \beta m$	Pipelining
Scan/Exscan	same	
All-to-all	Not covered	

Fully connected (tree, hypercube):

Collective	$T(m)$ in linear cost model	
Gather/Scatter	$\text{ceil}(\log p)\alpha + (p-1)/p \beta m$	Binomial tree
Allgather	same	Circulant graph
Reduce-scatter		
Bcast	$2\text{ceil}(\log p) \alpha + o(pm) + 2\beta m$	Binary tree pipe
	$2\text{ceil}(\log p) \alpha + 2o(pm) + \beta m$	2-trees
	$\text{ceil}(\log p)\alpha + o(pm) + \beta m$	Pipelining, Bin-Jia
Scan/Exscan	$\text{ceil}(\log p)(\alpha + \beta m)$	Hillis-Steele
	$4\text{ceil}(\log p) \alpha + 2o(pm) + 2\beta m$	2-trees
All-to-all	$(p-1) (\alpha + 1/p \beta m)$	Direct, fully conn., 1-factor
	$\text{ceil}(\log p)(\alpha + \beta \text{floor}(m/2))$	Circulant Bruck

## Back to MPI (Example: Changing data distributions)

Matrix-vector multiplication algorithms with different layouts





## The MPI collective interfaces

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

`buffer`: address of data (address)

`count`: number of elements (int)

`type`: MPI datatype describing layout (= static structure) of element



Contiguous buffer of consecutive elements



Noncontiguous buffer  
(homogeneous)

## The MPI collective interfaces

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

`buffer`: address of data (address)

`count`: number of elements (int)

`type`: MPI datatype describing layout (= static structure) of element



Contiguous buffer of consecutive elements



Noncontiguous buffer  
(heterogeneous: different types of elements)

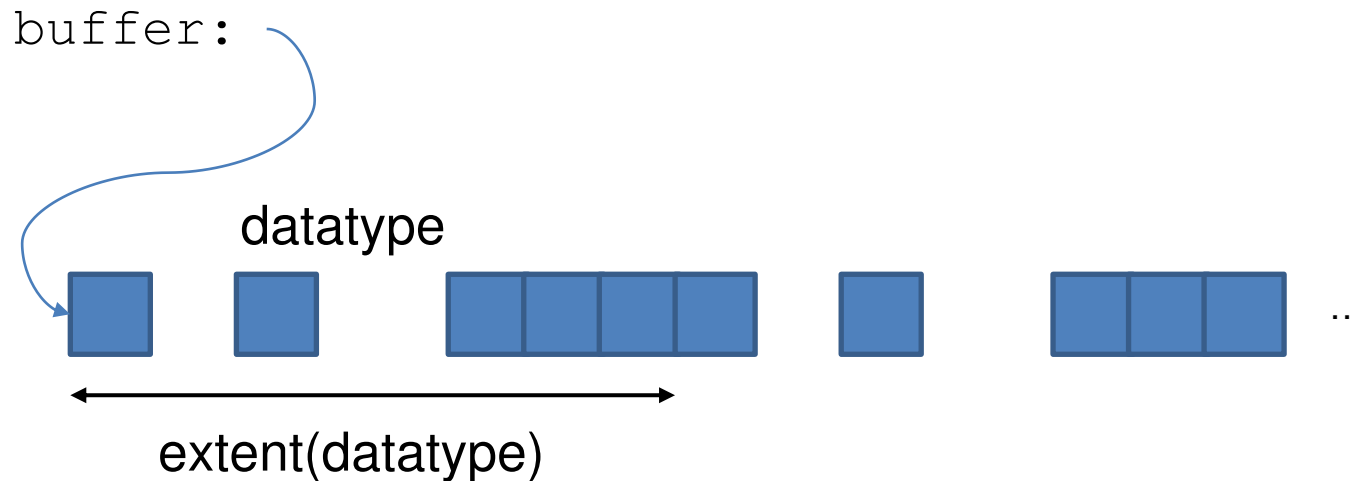
## MPI typed data communication

All communication operations: Sequence of typed, basic elements are sent and received in **predefined order**

Basic elements (MPI predefined datatypes):  
int (MPI\_INT), char (MPI\_CHAR), double (MPI\_DOUBLE), ...

**Order** , **position** and **number** of elements is determined by `count` and `datatype` arguments:

Count repetitions of the layout described by datatype, the  $i$ 'th repetition,  $0 \leq i < \text{count}$  is at relative offset  $i * \text{extent}(\text{datatype})$  in buffer



**Important** (very):

extent(datatype) is just an **arbitrary** (almost...) unit associated with the datatype, used for calculating offsets

Datatypes always have extent. Default rules for how the extent is set, but can also be set explicitly. Sometimes very powerful tool

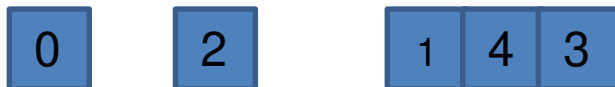
## MPI datatypes (layouts), formally

Data layout: Finite sequence of (basic element, offset) pairs

Type map: Finite sequence of (basic type, offset) pairs

Type signature : Finite sequence of (basic type)

Sequence: Basic elements in layout has an order. Elements are communicated in that order



## Type map (layout)



← True extent: Difference between offset of first and last element →

## Corresponding type signature



Basetype elements, corresponding to C (Fortran) int, double, float, char, ...

## MPI typed data communication, again

Sequences of elements described by the type signature of the count and datatype arguments are communicated.

MPI communication is **correct** if signature of sent elements **matches** the signature of received element

Matching:

- In collective operations, send and receive signatures must be identical
- In point-to-point and one-sided communication, send signature must be a prefix of receive signature

These requirements are not checked; it is user responsibility to ensure that types are used correctly. No type casting is performed in communication

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

`sendbuf, sendcount, sendtype`: Describes **one block** of data elements to be sent to one other processor

`recvbuf, recvcount, recvtype`: Describes **one block** of data elements to be received from some other processor

**Block of data elements** : `count` repetitions of type with `j`'th repetition of `i`'th block at offset  $(j+i*\text{count})*\text{extent}(\text{type})$  from `buf`,  $0 \leq i < \text{size}$ ,  $0 \leq j < \text{count}$



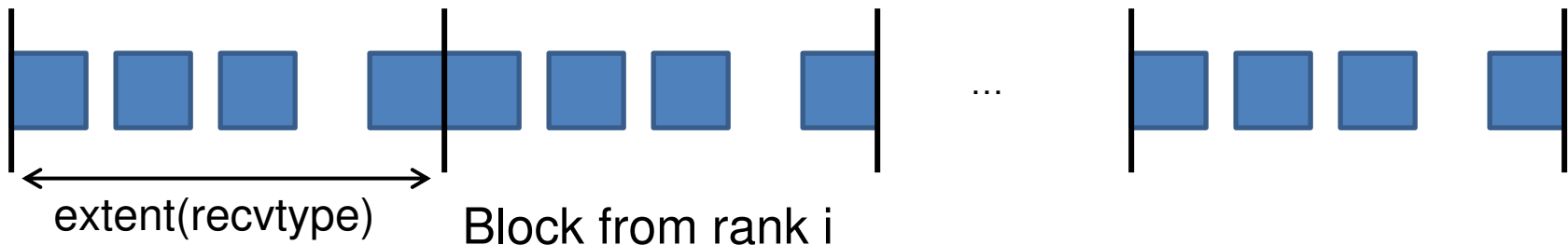
```
MPI_Gather (sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype, root, comm) ;
```

**sendbuf** (rank i): Example: different send/recv types, with  
**same signature**



**recvbuf** (root):

Root **must** allocate memory for  $\text{size}(\text{comm})$  blocks of  $\text{recvcount} \times [\text{true}] \text{extent}(\text{recvtype})$  elements, otherwise **BIG TROUBLE**



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

## Consistency rules:

- Signature of sendblock **must be exactly same** as signature of corresponding receive block (sequence of basic elements sent by one processor must be same as sequence expected by receiving process)
- Consistent values for other arguments, e.g. **same root**, **same op**, ...

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

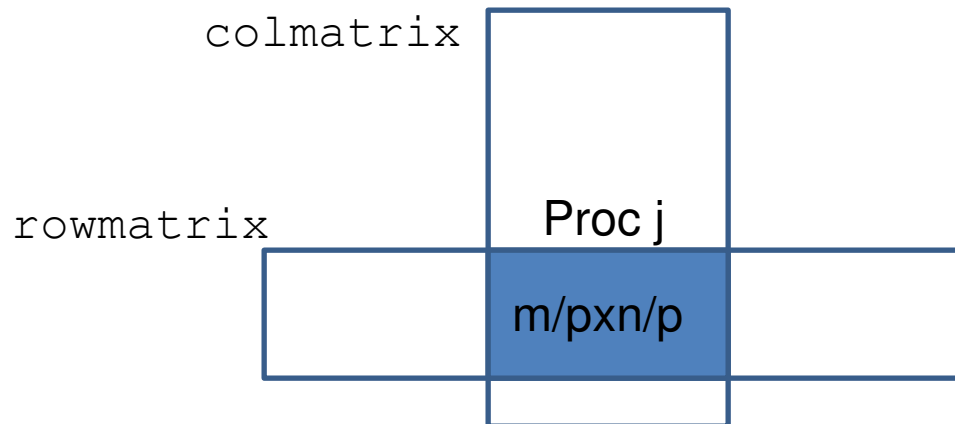
## Consistency rules:

- Signature of sendblock **must be exactly same** as signature of corresponding receive block (sequence of basic elements sent by one processor must be same as sequence expected by receiving process)

**Recall:** Point-to-point and one-sided models are less strict: Send signature must be a **prefix** of receive signature

Change distribution

Process  $i$  shall send block of  $m/p \times n/p$  elements to process  $j$ ,  
 $j=0, \dots, p-1$ : Transpose of submatrices: alltoall communication

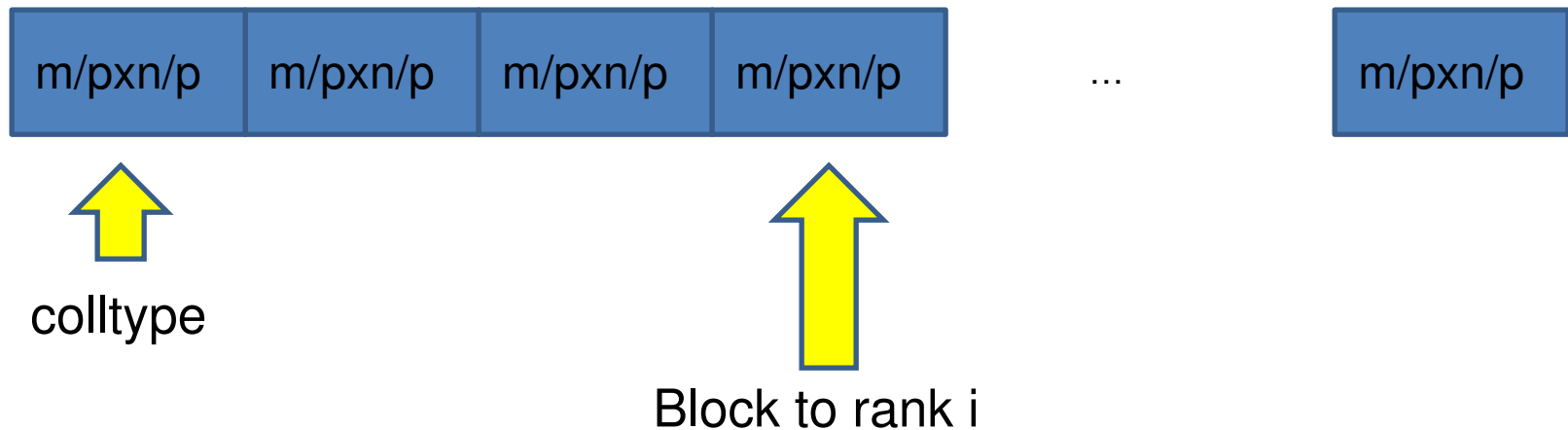


**Recall:**

Matrices in C stored in row order

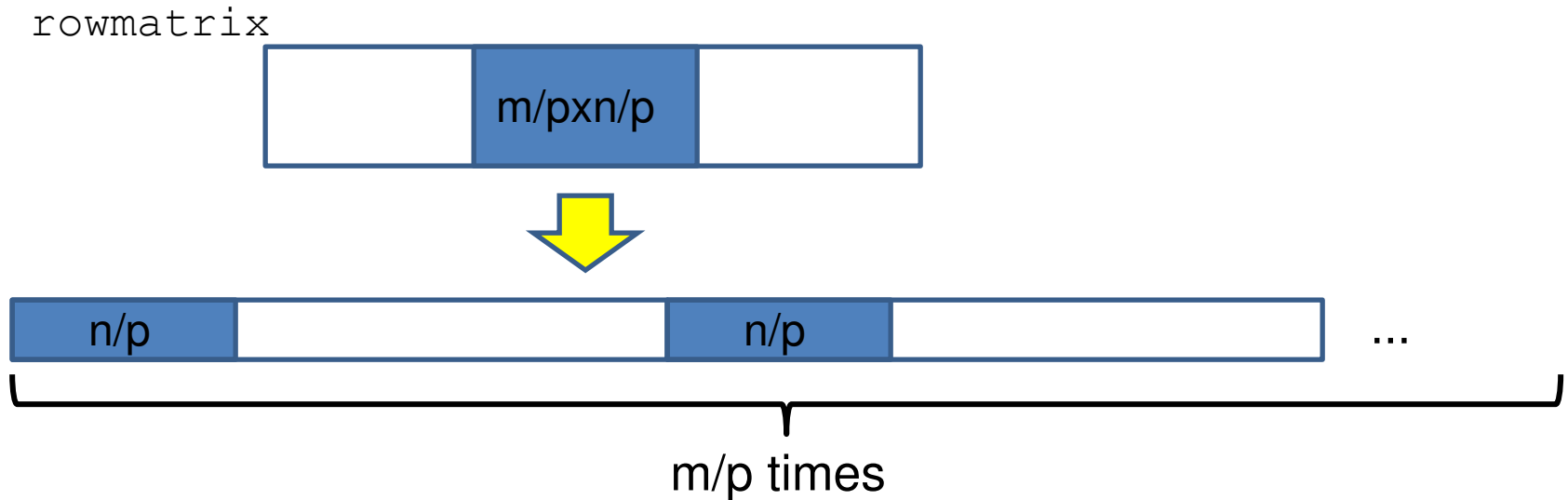
```
MPI_Alltoall(rowmatrix, 1, coltype,
              colmatrix, m/p*n/p, MPI_DOUBLE, comm);
```

```
MPI_Alltoall(rowmatrix, 1, coltype,  
             colmatrix, m/p*n/p, MPI_DOUBLE, comm) ;
```



Process  $i$  stores  $m/p \times n$  matrix in row major.

$m/p \times n/c$  submatrix consists of  $n/c$  element rows strided  $n$  elements apart.



This layout can be described as MPI vector data type

## Legal use of datatypes: Different send and receive types

```
MPI_Alltoall(rowmatrix, 1, coltype,  
             colmatrix, m/p*n/p, MPI_DOUBLE, comm) ;
```

### Process root

```
MPI_Bcast(buffer, 1, coltype, root, comm) ;
```

### Non-root

```
MPI_Bcast(buffer, 1, rowtype, root, comm) ;
```

Number of basic elements **must be same** for all pairs of processes exchanging data; sequence of basic datatypes (int, float, float, int, char, ...) must be same: **Type signature**



`coltype`: derived datatype that describes  $m/p$  rows of  $n/p$  element column of  $n$ -element vector

```
MPI_Type_vector(m/p, n/p, n, MPI_DOUBLE, &coltype);

MPI_Type_commit(&coltype);
```



is wrong



Extent of datatype is used to compute offset of next block in `MPI_Alltoall`





`coltype`: derived datatype that describes  $m/p$  rows of  $n/p$  element column of  $n$ -element vector

```
MPI_Type_vector(m/p, n/p, n, MPI_DOUBLE, &coltype);

MPI_Type_commit(&coltype);
```

Correct extent for redistribution

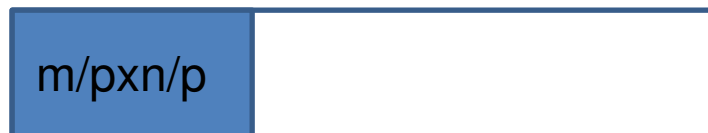




`coltype`: derived datatype that describes  $m/p$  rows of  $n/p$  element column of  $n$ -element vector

```
MPI_Type_vector(m/p, n/p, n, MPI_DOUBLE, &cc);
MPI_Type_create_resized(cc, 0, n/p*sizeof(double),
                        &coltype);
MPI_Type_commit(&coltype);
```

Correct extent for redistribution





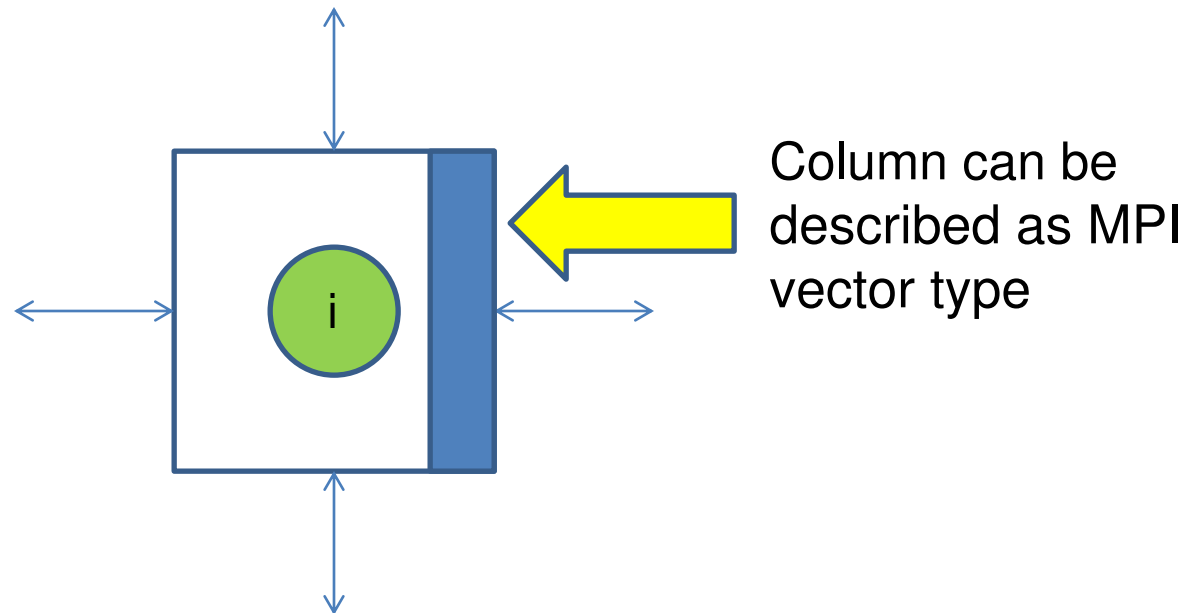
Change distribution

Process  $i$  shall send block of  $m/p \times n/p$  elements to process  $j$ ,  
 $j=0, \dots, p-1$

Process sends  $\text{rowmatrix} + i * 1 * \text{extent}(\text{coltype})$  to process  $i$ ,  
receives  $\text{colmatrix} + i * m/p * n/p * \text{extent}(\text{MPI\_DOUBLE})$  from  
process  $i$

```
MPI_Alltoall(rowmatrix, 1, coltype,  
             colmatrix, m/p*n/p, MPI_DOUBLE, comm);
```

## Example: 2d-stencil 5-point computation



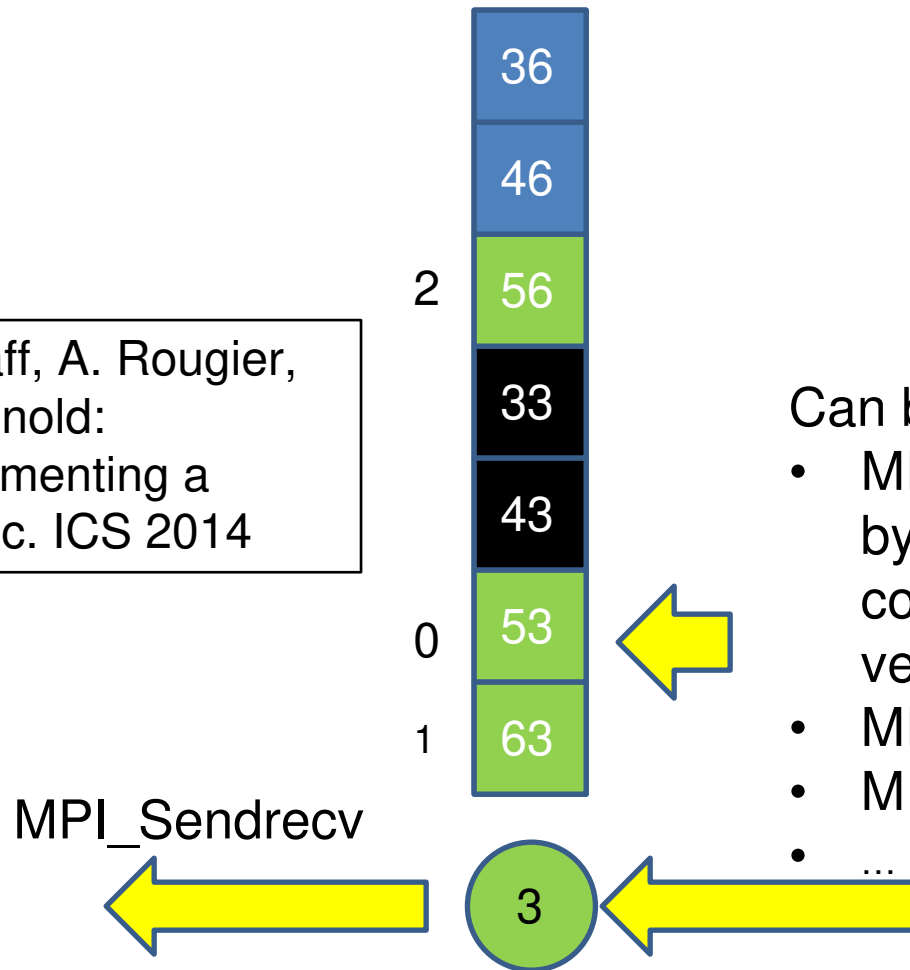
Each process  $i$  communicates with 4 neighbors and exchanges data at the border of own submatrix:

- Vector data type for first and last column
- Communication pattern can be implemented with point-to-point, one-sided, and **MPI 3.0** neighborhood collective communication

## Example: Round k of Bruck all-to-all algorithm

More in

J. Träff, A. Rougier,  
S. Hunold:  
Implementing a  
classic. ICS 2014



Can be described as:

- MPI vector followed by contig followed by contig followed by vector
- MPI indexed
- MPI struct
- ...

## (Research) Questions:

- Are all descriptions of the same data layout equally good: Performance?
- Is there a best possible description?
- How expensive is it to compute a best description?

Robert Ganian, Martin Kalany, Stefan Szeider, Jesper Larsson Träff: Polynomial-Time Construction of Optimal MPI Derived Datatype Trees. IPDPS 2016: 638-647

Alexandra Carpen-Amarie, Sascha Hunold, Jesper Larsson Träff: On the Expected and Observed Communication Performance with MPI Derived Datatypes. EuroMPI 2016: 108-12

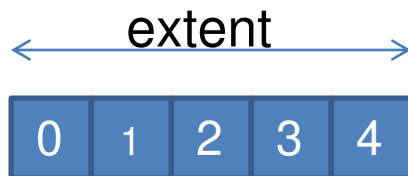
Master thesis

## MPI derived datatypes to describe application data layouts

- Basic type: MPI\_INT, MPI\_FLOAT, MPI\_DOUBLE, MPI\_CHAR, ... - or previously defined, derived datatype

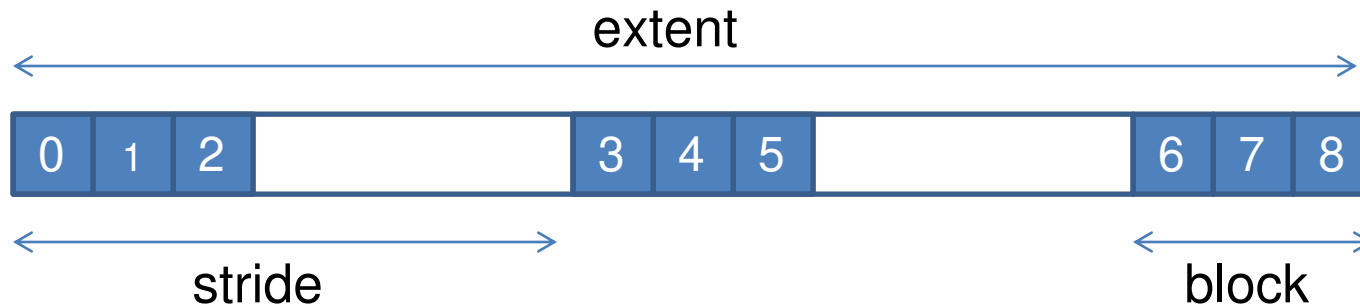
New derived datatypes built from previously defined ones (do not have to be committed) using the MPI type constructors

```
MPI_Type_contiguous(count, oldtype, &newtype);
```



Constructor defines type signature (order of elements)

```
MPI_Type_vector(n,block, stride, oldtype, &newtype);
```



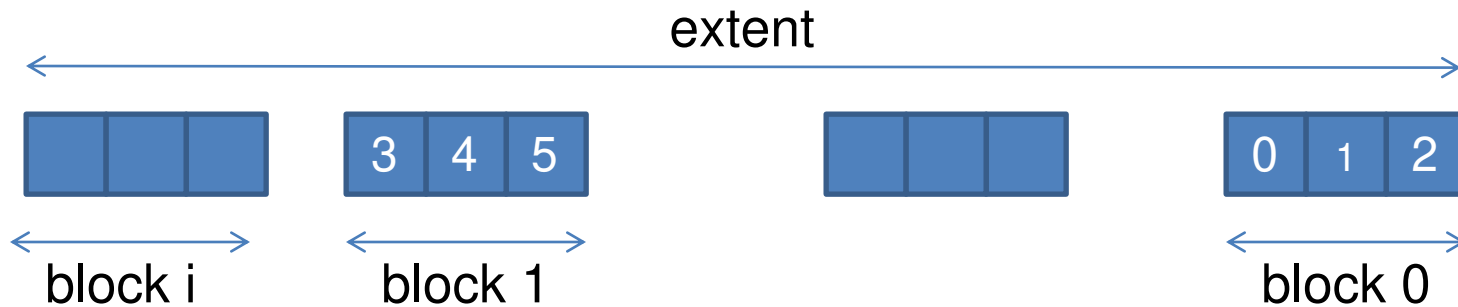
### Note :

Vector extent does not include stride (full block) of last block.  
 Both block(count) and stride are in units of extent(oldtype).  
 Constructor defines type signature (order of elements)

`MPI_Type_create_hvector` allows stride in bytes



```
MPI_Type_create_indexed_block(n,block,displacement[],
                             oldtype,&newtype);
```



```
MPI_Type_create_indexed(n,block[],displacement[],
                        oldtype,&newtype);
```



Also constructors with byte displacements

```
MPI_Type_create_struct(n,block[],displacement[],
                      oldtype[],&newtype);
```



- Derived datatypes make it possible to (recursively) describe any layout of data in memory
- Derived datatypes can be used in all communication operations: point-to-point, one-sided, collective
- Essential in MPI-IO

Before use in communication:

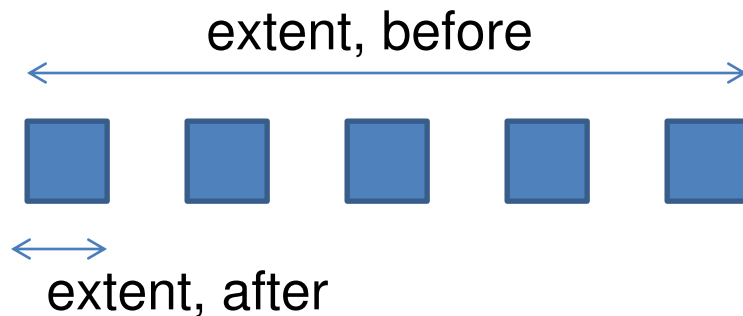
```
MPI_Type_commit(&type);
```

After use:

```
MPI_Type_free(&type);
```

## Advanced datatype usage

```
MPI_Type_create_resized(oldtype, lb, extent, &newtype);
```



```
MPI_Type_size(type, &size); // number of bytes consumed
```

```
MPI_Type_get_extent(datatype, &lb, &extent);
```

```
MPI_Type_get_true_extent(type, &lb, &extent);
```

```
MPI_Type_get_true_extent(type, &lb, &extent);
```

True extent: Difference between the highest and the lowest offset of two basic datatypes in the datatype. The true lower bound is the lowest offset of a basic datatype in the datatype

Use true extent for (safer) buffer allocation

MPI\_BOTTOM: Special (null) buffer address, can be used as buffer argument. Datatype offsets are absolute addresses.

with care

Use

## Communication with datatypes

**Question** : Overlapping elements?

**sendbuf, sendcount, sendtype** : Layouts, also between different blocks of elements, may overlap (some elements are sent multiple times)

**recvbuf, recvcount, recvtype** : Each element once in layout of total receive buffer, overlap illegal

Reason: Determinism

MPI\_Pack/Unpack: Functionality to pack/unpack a structured buffer to a consecutive sequence of elements (type signature)

Performance expectation:

```
MPI_Pack(buffer, count, type, &outbuf, &outsize, ...);  
MPI_Bcast(outbuf, outsize, MPI_PACKED, ...);
```

not slower than (since they are semantically equivalent)

```
MPI_Bcast(buffer, count, type, ...);
```

**But:**

Pack/unpack still useful for point-to-point communication of incomplete data (incremental un/packing) and “type safe unions”)

**Note** : MPI\_Pack/Unpack work on units of datatypes. The functionality is **not sufficient** for accessing arbitrary sequences of elements in a layout described by derived datatypes.

Pipelining requires (unexposed) library-internal functionality

Tarun Prabhu, William Gropp: DAME: A Runtime-Compiled Engine for Derived Datatypes. EuroMPI 2015: 4:1-4:10

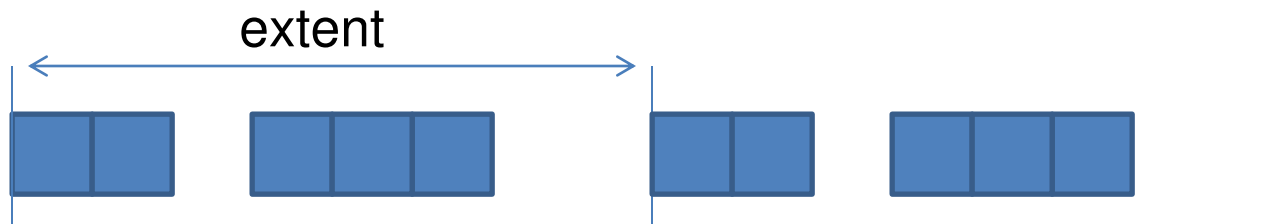
Timo Schneider, Fredrik Kjolstad, Torsten Hoefler: MPI datatype processing using runtime compilation. EuroMPI 2013: 19-24

Timo Schneider, Robert Gerstenberger, Torsten Hoefler: Micro-applications for Communication Data Access Patterns and MPI Datatypes. EuroMPI 2012: 121-131

Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, Falk Zimmermann: Flattening on the Fly: Efficient Handling of MPI Derived Datatypes. PVM/MPI 1999: 109-116

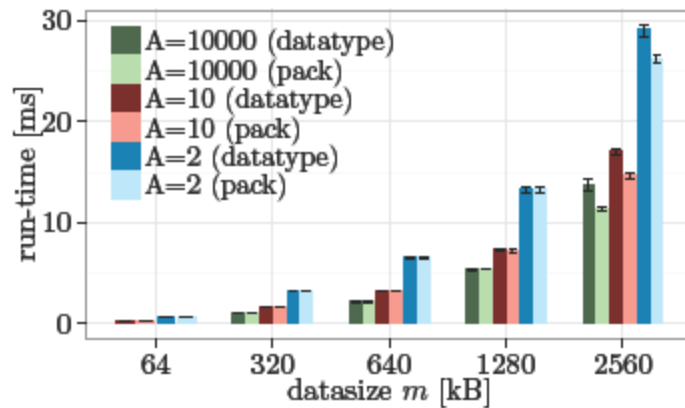
Is this true?

Check with different basic layouts. Example: [Alternating layout](#):  
Two blocks with A1 and A2 elements with strides B1 and B2

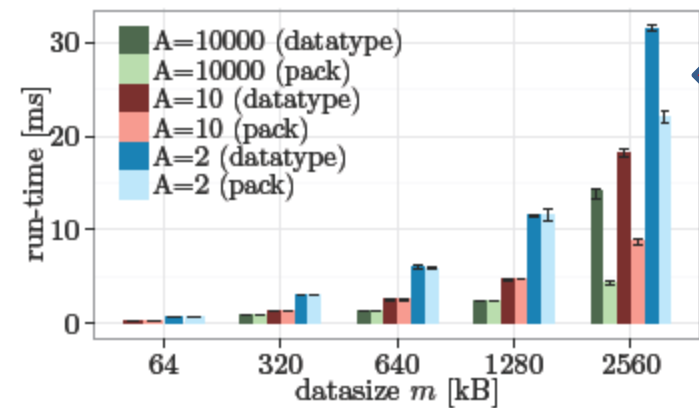




16 processes on one node



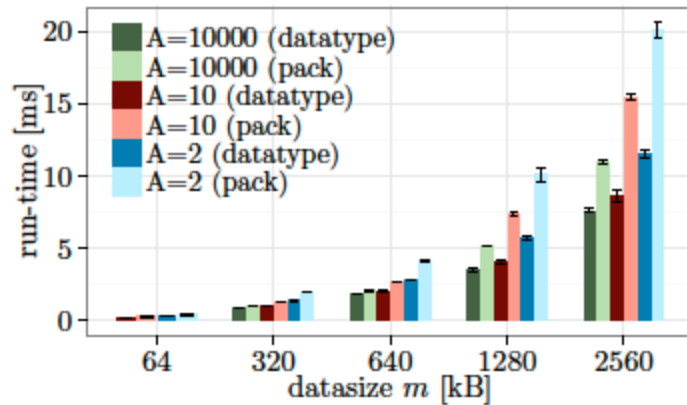
32 processes on 32 nodes



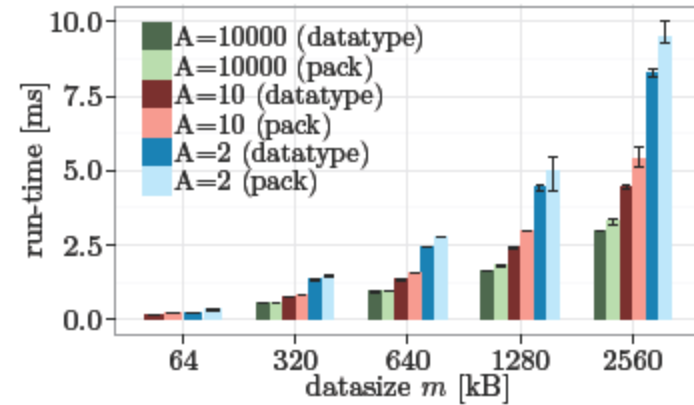
MPI library: mvapich 2-2.1

MPI\_Bcast benchmark (root packs, non-roots unpack)

16 processes on one node



32 processes on 32 nodes



MPI library: NECmpi 1.3.1

Note also absolute performance difference between the two libraries

MPI\_Bcast benchmark (root packs, non-roots unpack)

## Benchmarking datatypes under expectations:

Alexandra Carpen-Amarie, Sascha Hunold, Jesper Larsson Träff:  
On the Expected and Observed Communication Performance with  
MPI Derived Datatypes. EuroMPI 2016: 108-120

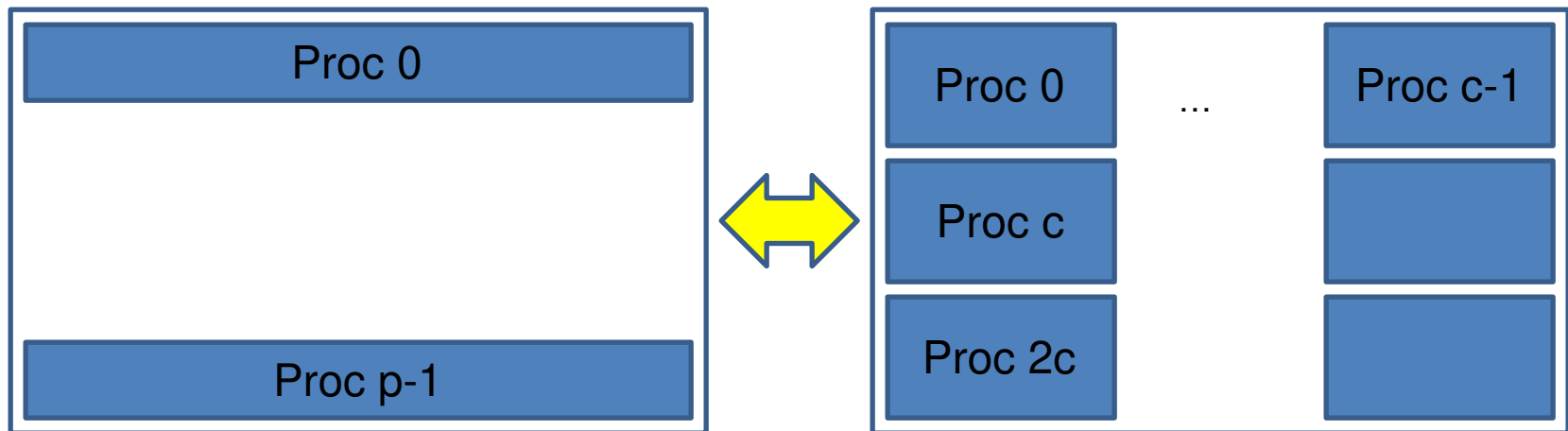
## On the limited support in MPI specification for programming with datatypes:

Jesper Larsson Träff: A Library for Advanced Datatype  
Programming. EuroMPI 2016: 98-107

Master thesis

## Example: Changing distributions (2)

Process  $i$  in comm shall send an  $m/p_{xn}/c$  block to process  $i\%c+i/r$  in Cartesian communicator rcomm



### Problem:

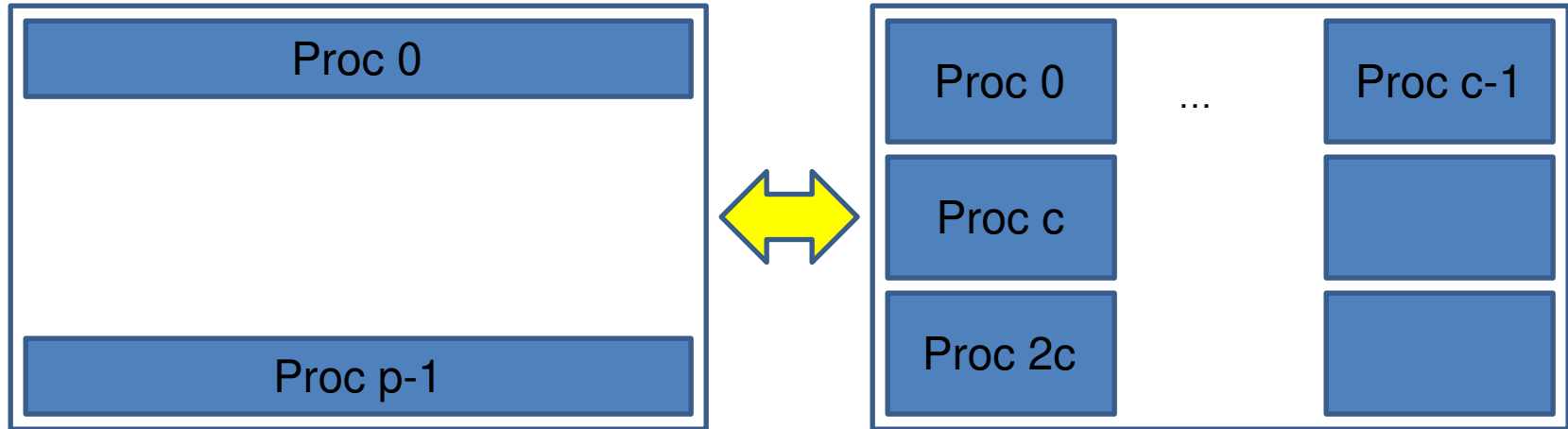
Two communicators, but MPI communication is always relative to one communicator

## Solution: Translating ranks between groups

```
MPI_Comm_rank(comm, &rank);  
MPI_Comm_group(comm, &group);  
MPI_Comm_group(rccomm, &rcgroup);  
  
int rcrank;  
MPI_Group_translate_rank(group, 1, &rank, rcgroup,  
                        &rcrank);
```

### Problem:

Each process sends blocks only to c other processes



- Each rank in comm sends  $c$  blocks of  $m/p$  rows and  $n/c$  columns to  $c$  consecutive processes
- Each rank in rcomm receives  $m/p$  rows and  $n/c$  columns from  $c$  consecutive processes

## Solution: Irregular alltoall and datatypes

```
for (i=0; i<size; i++) {
    scount[i] = 0; sdisp[i] = 0;
    rcount[i] = 0; rdisp[i] = 0;
}

MPI_Comm_rank(comm, &rank);
for (i=0; i<c; i++) {
    int oldrank, colrank = rank/r+i;
    MPI_Group_translate_rank(rcgroup, 1, colrank,
                           group, &oldrank);

    scount[oldrank] = 1;
    sdisp[oldrank] = i*extent(coltype);
}
...
MPI_Alltoallv(rows, scount, sdisp, coltype, ..., comm);
```

## Solution: Irregular alltoall and datatypes

```
...
MPI_Cart_coord(rccomm,rank,2,rccoord);
int rowrank = rccoord[0]*r;

for (i=0; i<c; i++) {
    rcount[rowrank] = m/p*n/c;
    rdisp[rowrank]  = i*m/p*n/c;
    rowrank++;
}

MPI_Alltoallv(rows,scount,sdisp,coltype,
              rowscols,rcount,rdisp,MPI_DOUBLE,
              comm);
```

**Home exercise** : Use a neighborhood collective instead



## Performance problem (alltoallv abuse):

- Each process only sends and receives to/from  $c$  neighbors
- $c$  (mostly)  $O(\sqrt{p})$  – sparse neighborhood

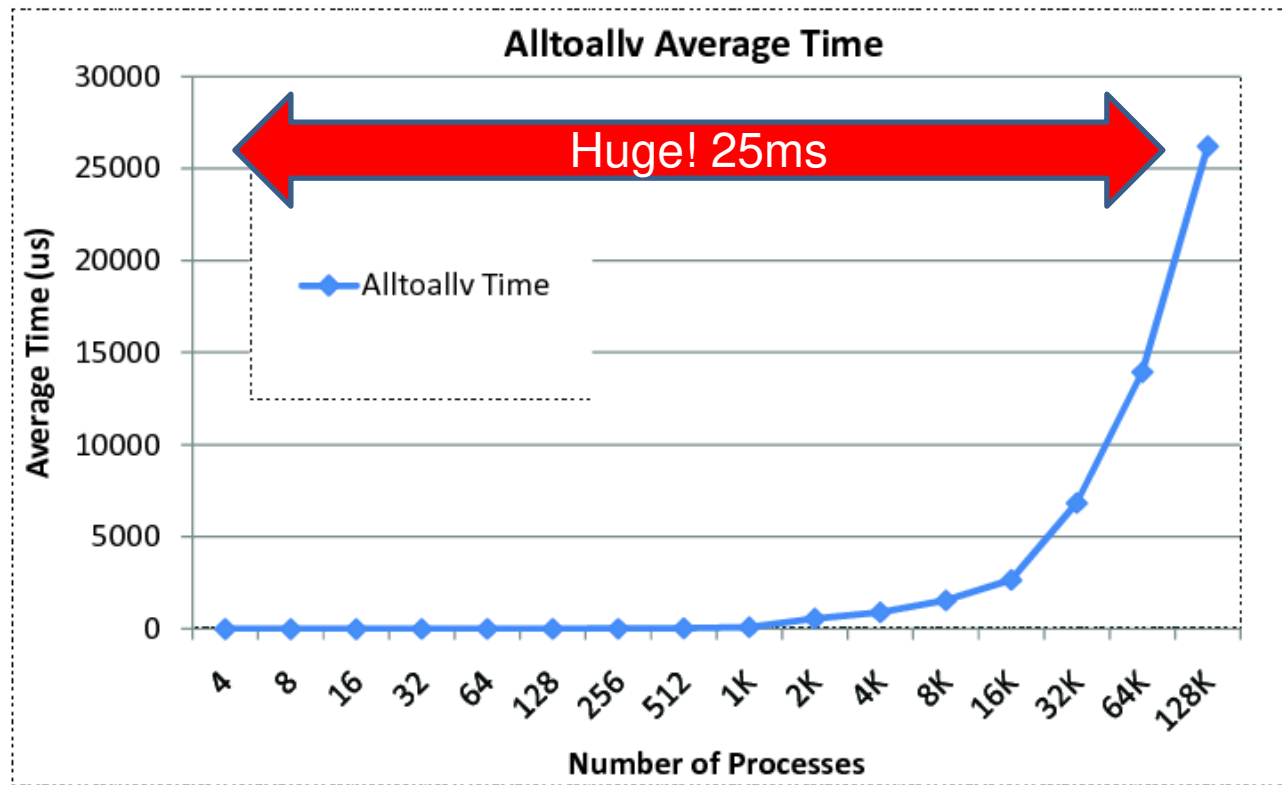
## Other problems :

- What if  $c$  and  $r$  do not divide  $n$  and  $m$ ?
- What if  $p$  does not factor nicely into  $r$  and  $c$ ?

## Partial solutions :

- “Padding”
- Irregular collectives; but some datatype functionality seems missing (there is only the fully general, expensive `MPI_Alltoallw`)

Experiment: What is the cost of MPI\_Alltoallv when no data are exchanged,  $\text{scount}[i]=0$ ,  $\text{rcount}[i]=0$ ?



On Argonne  
National  
Laboratory  
BlueGene/P  
system

P. Balaji et al.: MPI on Millions of Cores. Parallel Processing Letters 21(1): 45-60 (2011)

## New collectives in MPI 3.0

Topological/sparse/neighbor collectives can express collective patterns on sparse neighborhoods

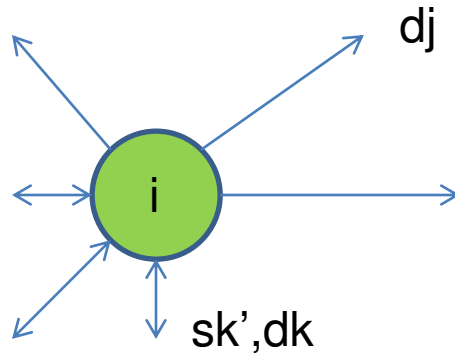
Neighborhoods expressed by process topologies

MPI\_Cart: Each process has 2d outgoing edges, 2d incoming edges

MPI\_Graph: Each process has outgoing and incoming edges as described by communication graph in MPI\_Dist\_graph\_create

Algorithmic flavor totally different from previous, global neighborhood collectives

`MPI_Neighbor_allgather(v)`

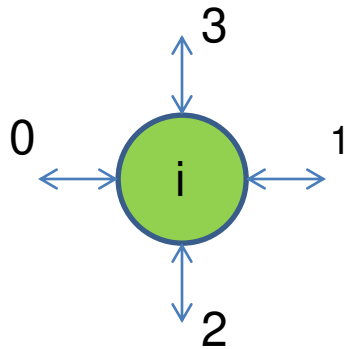


Neighborhood defined by  
general communication graph

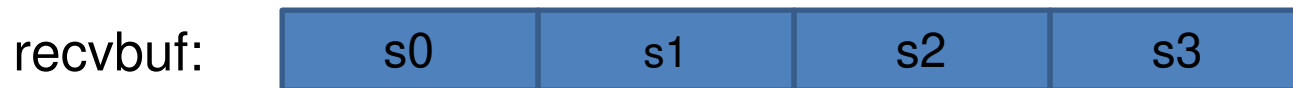


sendbuf sent to all destinations; individual block received from  
each source into recvbuf

## MPI\_Neighbor\_allgather(v)

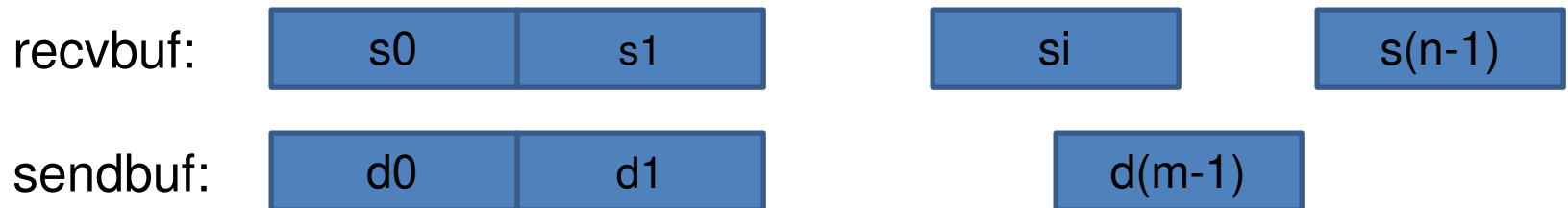
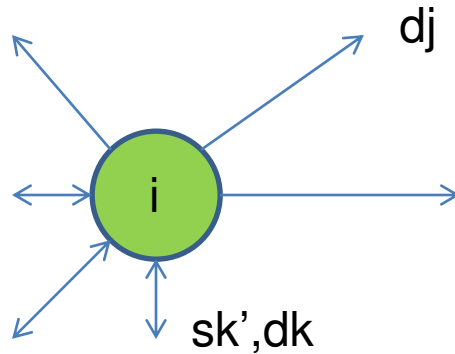


Cartesian neighborhood, neighbors in dimension order, -1 dist, +1 dist



sendbuf sent to all destinations; individual block received from each source into recvbuf

`MPI_Neighbor_alltoall(v,w)`



Individual block from `sendbuf` sent to each destination;  
individual block received from each source into `recvbuf`

## Algorithms for neighborhood (sparse) collectives

Not much is known, flavor totally different from global, dense collectives, more difficult, hard optimization problems for optimality results; see, e.g.,

Torsten Hoefler, Timo Schneider: Optimization principles for collective neighborhood communications. SC 2012: 98

Seyed Hessam Mirsadeghi, Jesper Larsson Träff, Pavan Balaji, Ahmad Afsahi: Exploiting Common Neighborhoods to Optimize MPI Neighborhood Collectives. HiPC 2017: 348-357

**MPI design questions** : Are the neighborhood collectives too powerful? Are they useful? Better compromises possible?

## Creation of virtual communication graph/neighborhoods

```
MPI_Dist_graph_create(comm,...,&graphcomm);  
MPI_Dist_graph_create_adjacent(comm,...,&graphcomm);
```

specify neighborhoods in a fully distributed fashion. Order of adjacent (in and out edges) implementation dependent, but must be fixed, as returned by calls to

```
MPI_Dist_neighbors_count(graphcomm,  
                        &indegree,&outdegree,  
                        &weighted);  
MPI_Dist_graph_neighbors(graphcomm,  
                        maxindeg,sources,sweights,  
                        maxoutdeg,destinats,dweights);
```

This order of edges is used in neighborhood collectives

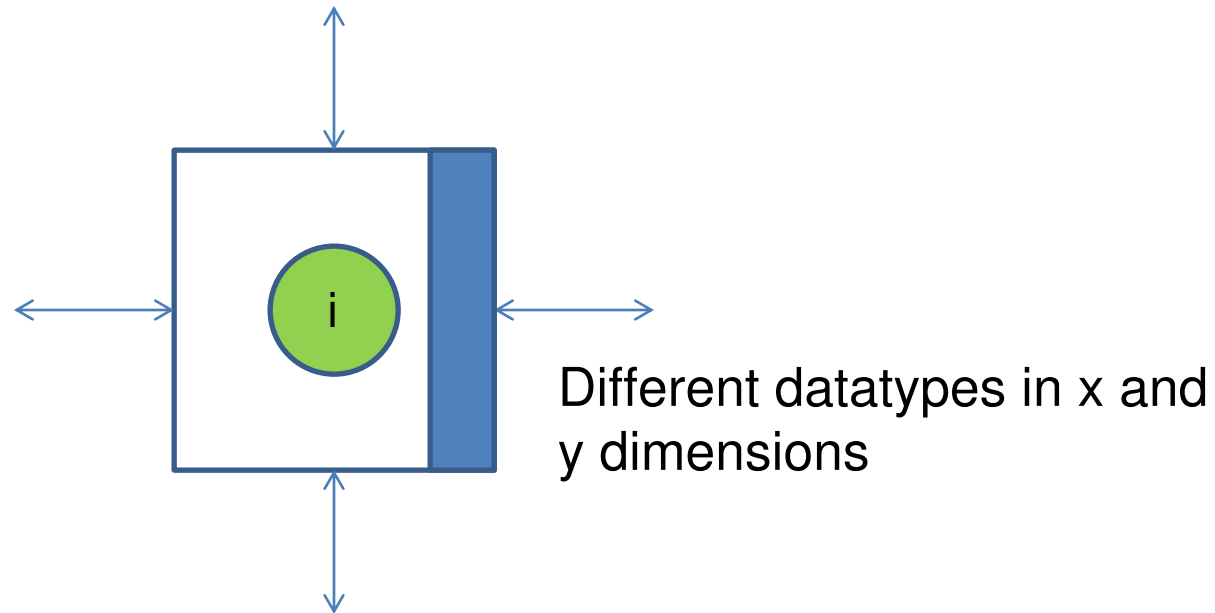


## WARNING

Never use the “old” (MPI 1) graph topology interface, `MPI_Graph_create()` etc.

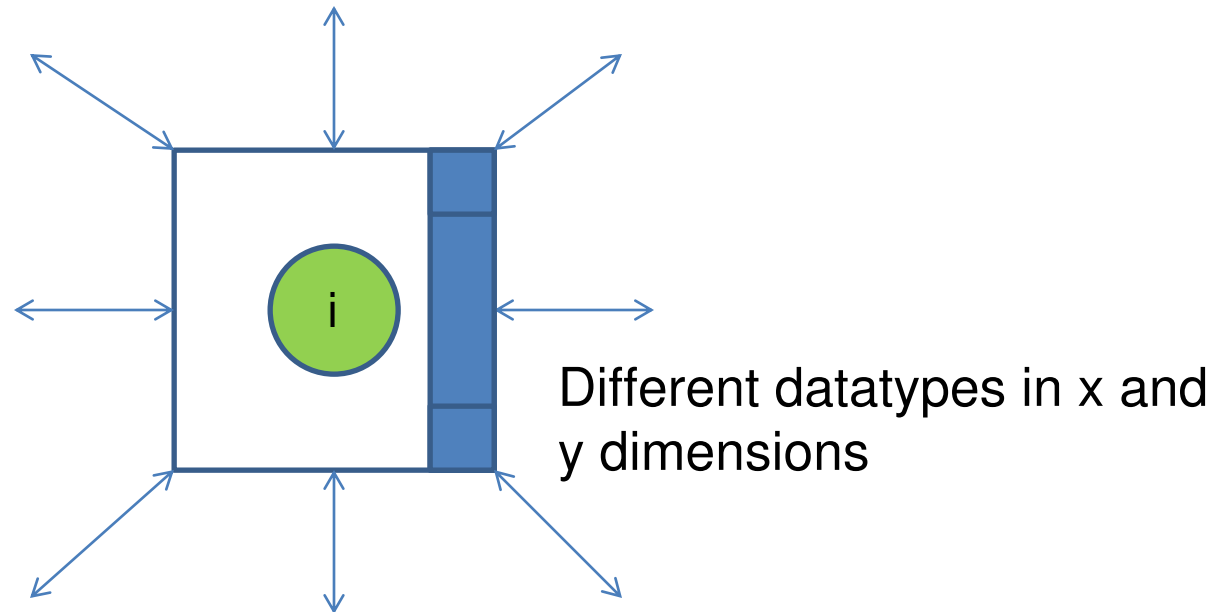
- Non-scalable
- Inconsistent
- May/will be deprecated

## Example: 2d stencil 5-point computation



- Cartesian neighborhood
- MPI\_Neighbor\_alltoallw: Why?

## Example: 2d stencil 9-point computation



- **Non-Cartesian** neighborhood. Have to use distributed graphs
- `MPI_Neighbor_alltoallw`: Why?

## Non-blocking collectives (from MPI 3.0)

In analogy with non-blocking point-to-point communication, all collective operations (also sparse) have non-blocking counterparts

MPI\_Ibarrier  
MPI\_Ibcast  
MPI\_Iscatter/Igather  
MPI\_Iallgather  
MPI\_Ialltoall  
...

MPI\_Ineighbor\_allgather  
MPI\_Ineighbor\_allgatherv  
MPI\_Ineighbor\_alltoall  
MPI\_Ineighbor\_alltoallv  
MPI\_Ineighbor\_alltoallw

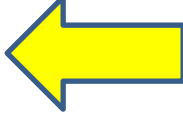
Check for completion: MPI\_Test (many variants)

Enforce completion: MPI\_Wait (many variants)

## Example:

```
MPI_Request request; // MPI request object

MPI_Iallgather(sendbuf, sendcount, sendtype,
               recvbuf, recvcount, recvtype, comm, &req);

// compute  Potential for overlap

MPI_Status status; // MPI status object:
// most fields undefined for non-blocking collectives
MPI_Wait(&req, &status);
```

Semantics as for blocking collectives: Locally complete (with the implications this has) after wait

## Non-blocking collective vs. point-to-point communication

MPI\_Send



MPI\_Irecv



All combinations  
permitted

MPI\_IBcast



MPI\_Bcast



Blocking and non-blocking  
collectives do not match.  
**Why?**

**Answer:** Blocking and non-blocking collective may use a different algorithm. Specification should not forbid such implementations

## Blocking collectives:

**Assumption**: Used in a synchronized manner, MPI process busy until (local) completion

Objective: Fast algorithms (latency, bandwidth)

## Non-blocking collectives:

**Assumption**: Used asynchronously, MPI process can do sensible things concurrently, postponed check for completion

Objective: Algorithms that permit overlap, can tolerate skewed arrival patterns, can exploit hardware offloading

## MPI and collective communication algorithms summary

- MPI is the reference communication interface in HPC
- Much to learn from the MPI design
- Good basis to study concepts in interfaces and algorithms for large-scale, parallel systems
- Many interesting research problems
- Application programmers need to know and understand MPI well to program effectively
- Much is known about efficient algorithms for collective communication operations in types of networks; **but not everything**
- Good synthesis for not fully-connected networks needed