

# Collective Communications

## Some references

- **Parallel Algorithms**, H. Casanova, A. Legrand, Y. Robert
- **Parallel Programming – For Multicore and Cluster System**, T. Rauber, G. Rünger

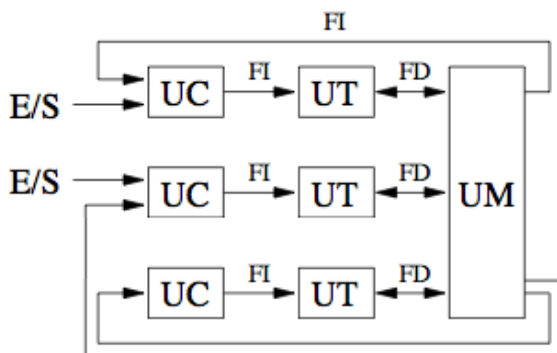
# MIMD: Multiple Instructions stream, multiple data stream

Multi-Processor Machines

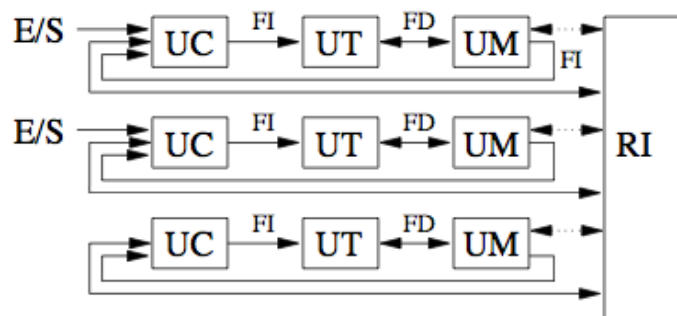
Each processor runs its own code asynchronously and independently

**Two sub-classes**

**Shared memory**



**Distributed memory**

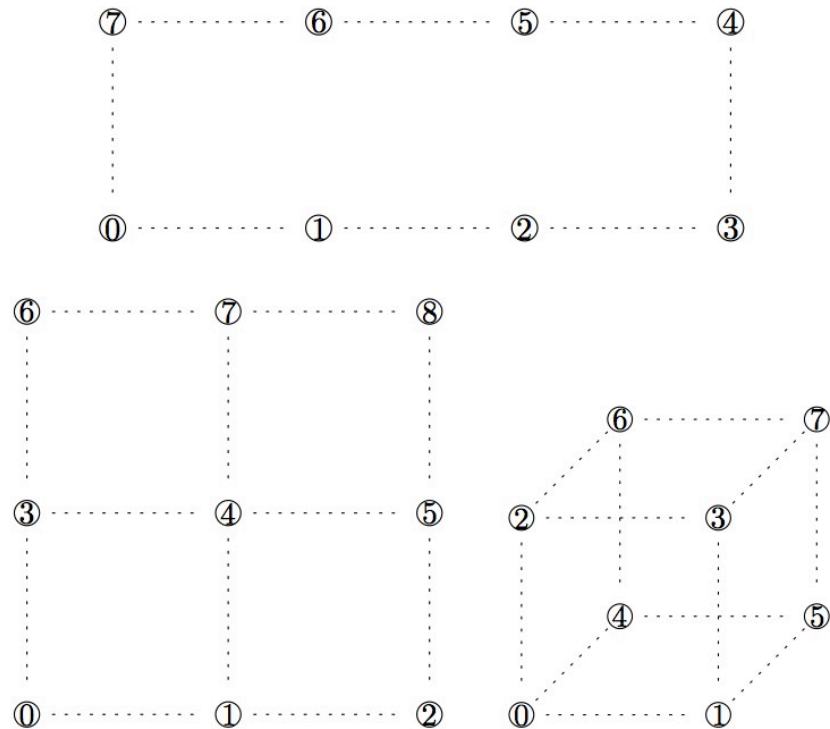


A mix between SIMD and MIMD: SPMD (Single Program, Multiple Data)

## Collectives communications

- Interactions between parts of a parallel program mapped in a set of processors happen following **well defined schemes** between groups of processors/cores
  - Not only point-to-point communications
- To write parallel algorithms, we need **collectives operations**
  - Broadcast, scatter, gather, all-to-all, ...
  - Used in most parallel applications
- MPI provides many of them
  - They should be designed to use efficiently hardware resources (processors, network, memory interfaces, bus, ...)
- Optimizing these operations can
  - Improve global performance of programs
  - Reduce the development cost of applications
  - Improve parallel software quality
- If possible, take the hardware architecture into account
- So why should we take a look at the way they are designed ?

# Topologies



## Communication costs

Global communications are usually written using point-to-point communications

### Difficulty to find accurate models

- MPI implementations have different optimisations depending of the message sizes
- Smart optimizations taking into account special hardware/software features

### Here we use a simplified model

- Time =  $L + m/B$  (without contentions)
- L: startup (or latency) time
- B: bandwidth ( $b = 1/B$ )
- m: message size

#### ■ Store-and-forward

- If we suppose that a message of length  $m$  is sent from  $P_0$  to  $P_q$ , then the communication cost is

$$T_c(m) = q(L + m b)$$

# Suppositions about communications

## Several options

- Send() and Recv() are both blocking  
Called “rendez-vous” mode
- Recv() is blocking, but Send() is not  
Pretty standard  
MPI supports it
- Recv() and Send() are both non-blocking  
Pretty standard too  
MPI supports it as well



## Supposition about concurrency

**An important question:** can the processor perform several operations at the same time?

Generally we suppose that the processor is able to send, receive, and compute at the same time

- MPI\_IRecv()
- MPI\_Isend()
- Compute something

### We need these three operations to be independent

- We can not send the result of a computation before it is computed
- We can not send what we receive (*forwarding*) unless we pipeline the communication

When we write parallel algorithms (in pseudo-code), we write concurrent activities with the || sign



# Virtual topology versus physical topology

- We have chosen that our virtual topology is a ring
  - We suppose that the topology is a ring too
- Maybe an other virtual topology is more adapted to the physical one we have for our cluster
- The ring of processes allows to have simple algorithms
- With quite good performances
- Good candidate for our first approach of parallel algorithmics

## Some global operations

- One-to-all broadcast and reduction
- All-to-all broadcast and reduction
- All-Reduce operation and prefix sum
- Scatter and Gather
- Personalized all-to-all communication
- Circular shift



## Broadcast (one-to-all communication)



- **Input**

- Message M is stored on root processor

- **Output**

- Message M is stored locally on every processors

## Reduction (all-to-one reduction)



$$M := M_0 \oplus M_1 \oplus M_2 \oplus M_3$$

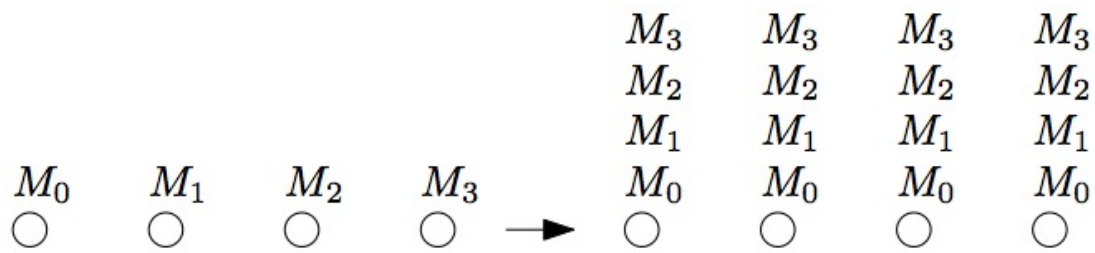
- **Input**

- The p messages  $M_k$  for  $k = 0, 1, \dots, p-1$
- Message  $M_k$  is stored locally on processor k
- An associative operation (+, x, max, min)

- **Output**

- The “sum” is stored on root processor

## All-to-all broadcast



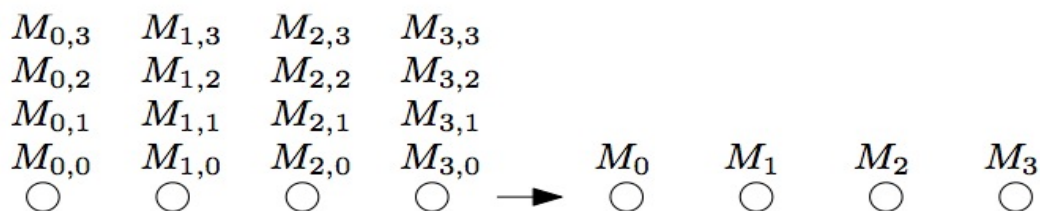
### • Input

- The  $p$  messages  $M_k$  for  $k = 0, 1, \dots, p-1$
- Message  $M_k$  is stored locally processor  $k$

### • Output

- The  $p$  messages  $M_k$  for  $k = 0, 1, \dots, p-1$  are stored locally on every processors

## All-to-all reduction



### • Input

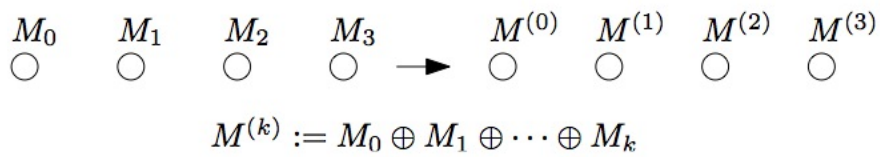
- The  $p^2$  messages  $M_{r,k}$  for  $r, k = 0, 1, \dots, p-1$
- Message  $M_{r,k}$  is stored locally on processor  $r$
- An associative operation  $(+, \times, \max, \min)$

### • Output

- The “sum” is stored on the root processor

$$M_r := M_{0,r} \oplus M_{1,r} \oplus \dots \oplus M_{p-1,r}$$

## Prefix sum



### • Input

- The  $p$  messages  $M_k$  for  $k = 0, 1, \dots, p-1$
- Message  $M_k$  is stored locally on processor  $k$
- An associative operation  $(+, \times, \max, \min)$

### • Output

- The “sum” is stored locally on processor  $k$  for all  $k$

$$M^{(k)} := M_0 \oplus M_1 \oplus \dots \oplus M_k$$

## Scatter



### • Input

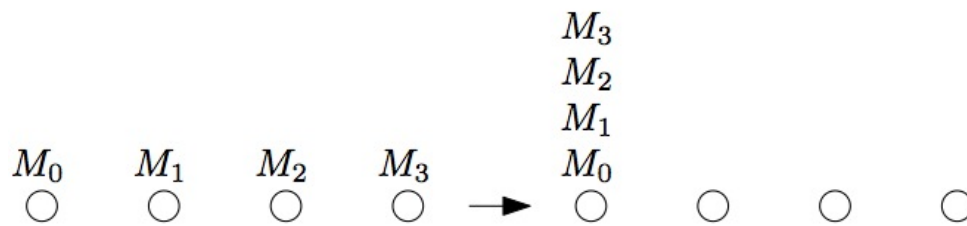
- The  $p$  messages  $M_k$  for  $k = 0, 1, \dots, p-1$  are stored locally on root processor

### • Output

- Message  $M_k$  is stored locally processor  $k$  for all  $k$



## Gather



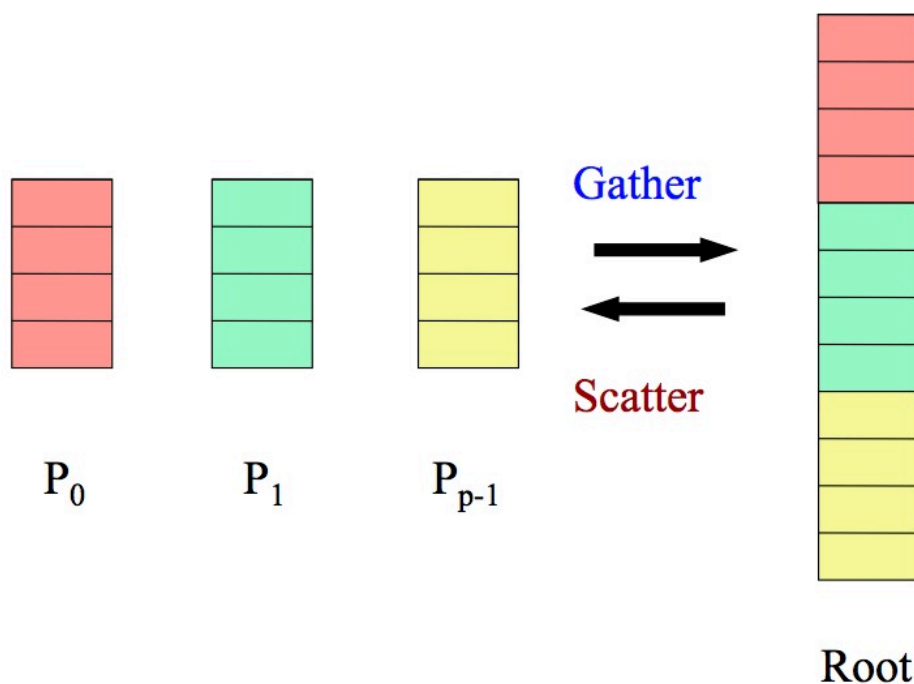
- **Input**

- The  $p$  messages  $M_k$  for  $k = 0, 1, \dots, p-1$
- Message  $M_k$  is stored locally on processor  $k$

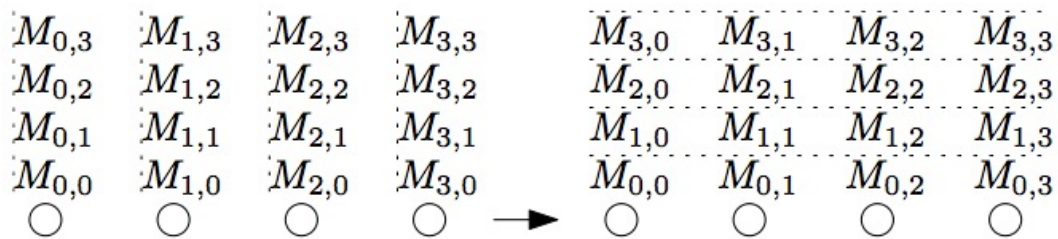
- **Output**

- The  $p$  messages  $M_k$  are stored locally on root processor

## Scatter/Gather



## Personnalized All-to-all (transposition)



- **Input**

- The  $p^2$  messages  $M_{r,k}$  for  $r, k = 0, 1, \dots, p-1$
- Message  $M_{r,k}$  is stored locally on processor  $r$

- **Output**

- The  $p$  messages  $M_{r,k}$  are stored locally processor  $k$  for all  $k$

## Circular shift



- **Input**

- The  $p$  messages  $M_k$  for  $k = 0, 1, \dots, p-1$  are stored locally on each processor

- **Output**

- Message  $M_{(k-1)\%p}$  is stored locally on  $k$  for each  $k$



# ALGORITHMS ON A RING OF PROCESSORS

## Ring of processors

Each process is identified by his rank

- MY\_NUM ( )

We have a way of finding the total number of processes

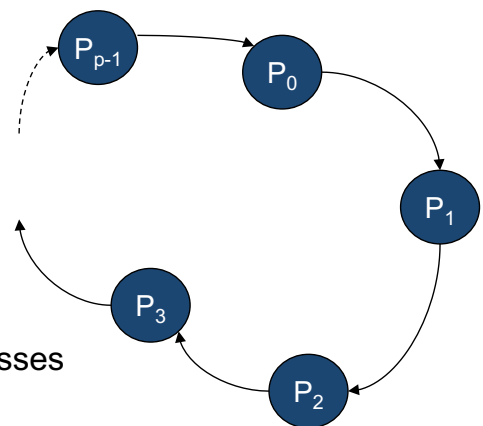
- NUM\_PROCS ( )

Each process can send message to each successor

- SEND(addr, L)

And receive a message to its predecessor

- RECV(addr, L)



## Broadcast

We want to write a program in which  $P_k$  sends the same message of length  $m$  to all other processors

-Broadcast ( $k$ ,  $addr$ ,  $m$ )

On a ring, the **naive algorithm** consists in sending message to the neighbor processor and so on and so forth, with **no parallel communication**

It should not be written like this if the physical topology is not a ring

- MPI uses some kind of tree



## Broadcast

**Broadcast( $k, addr, m$ )**

```
q = MY_NUM( )
```

```
p = NUM_PROCS( )
```

```
if (q == k)
```

```
    SEND(addr, m)
```

```
else
```

```
    if (q == k-1 mod p)
```

```
        RECV(addr, m)
```

```
    else
```

```
        RECV(addr, m)
```

```
        SEND(addr, m)
```

```
    endif
```

```
endif
```

- Assumes a blocking receive
- Send can be non-blocking
- The broadcast time is the following  
 $(p-1)(L+m \cdot b)$



## Optimized broadcast

- How to improve performance?
- We can split the message in smaller packets
  - $r$  packets where  $m$  can be divided by  $r$
- The root process sends  $r$  messages
- **The model of the broadcast can be computed like this**
  - Consider the last process to obtain the last packet of the message
  - We need  $p-1$  steps for the first packet to reach its destination, thus  $(p-1)(L + m b / r)$
  - The the next  $r-1$  packets arrive one after an other  $(r-1)(L + m b / r)$
  - Thus a total of  $(p + r - 2) (L + mb / r)$

## Optimized broadcast, contd.

The next question is, what is the value  $r$  that that minimizes

$$(p + r - 2) (M + m b / r) ?$$

- We can see the previous expression as  $(c+ar)(d+b/r)$ , with four constant values  $a, b, c, d$
- The non-constant part of the expression is thus  $ad.r + cb/r$ , that should be minimized
- This value is minimized for  $\text{sqrt}(cb / ad)$

thus we have

$$r_{\text{opt}} = \text{sqrt}(m(p-2)b / L)$$

With the optimal time

$$(\text{sqrt}((p-2)L) + \text{sqrt}(mb))^2$$

that tends towards  $mb$  when  $m$  is large (independent of  $p$  !)

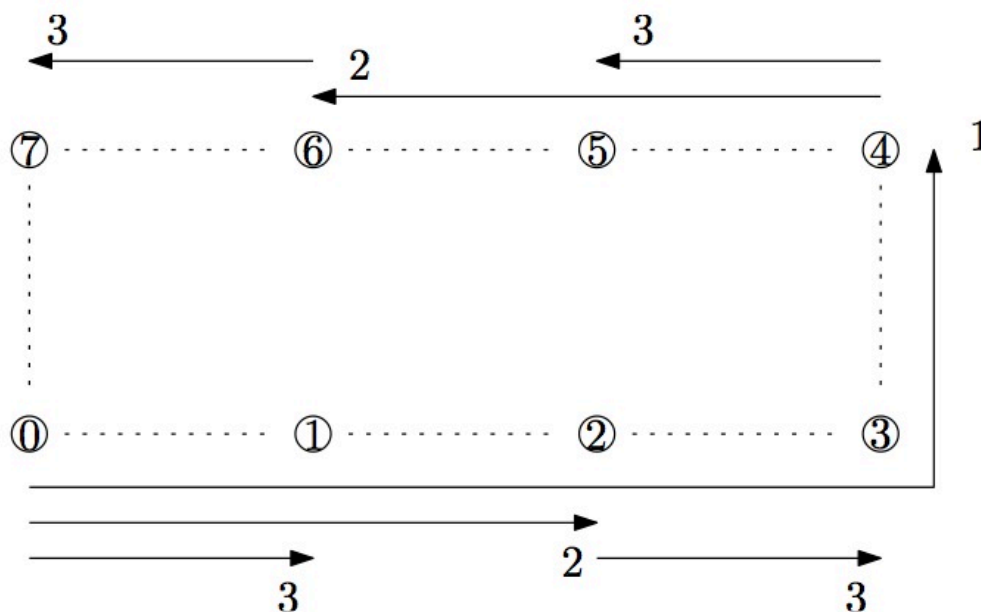
## Classical network principle

We have seen that if we cut a (large) message into a large number of (small) messages, then we can send the message through several jumps (in our case  $p-1$ ) virtually as fast as sending it to just one jump

### This is the fundamental principle of IP networks

- Messages are divided into several IP frames
- They are sent on several routers
- But the execution time is limited by the slowest router time

## Other solution: Recursive Doubling



Double the number of active processes at each step

## Scatter

- Process  $k$  sends a different message to all other processes (including it)
  - $P_k$  stores messages for  $P_q$  at address  $addr[q]$ , including a message to  $addr[k]$
- At the end of the execution, each processor has the message it received in `msg`
- The principle of the algorithm is just pipelining the communications starting with the message intended for  $P_{k-1}$ , the most distant process



## Scatter

### **Scatter( $k, msg, addr, m$ )**

```
q = MY_NUM( )
p = NUM_PROCS( )
if (q == k)
    for i = 0 to p-2
        SEND(addr[k+p-1-i mod p], m)
    msg ← addr[k]
else
    RECV(tempR, L)
    for i = 1 to k-1-q mod p
        tempS ↔ tempR
        SEND(tempS, m) || RECV(tempR, m)
    msg ← tempR
```

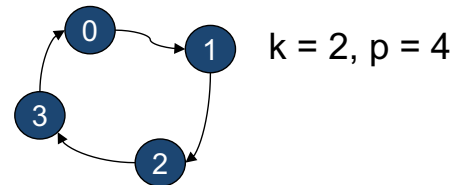
Same execution time than broadcast  
 $(p-1)(L + m b)$

Exchange of Send Buffer and  
Receive Buffer (Pointer)

Send and receive in  
parallel, with a non-  
blocking send



# Scatter



**Scatter(k,msg,addr,m)**

```
q = MY_NUM()
p = NUM_PROCS()
if (q == k)
  for i = 0 to p-2
    SEND(addr[k+p-1-i mod p],m)
  msg ← addr[k]
else
  RECV(tempR,L)
  for i = 1 to k-1-q mod p
    tempS ↔ tempR
    SEND(tempS,m) || RECV(tempR,m)
  msg ← tempR
```

**Proc q=2**

```
send addr[2+4-1-0 % 4 = 1]
send addr[2+4-1-1 % 4 = 0]
send addr[2+4-1-2 % 4 = 3]
msg = addr[2]
```

**Proc q=3**

```
recv (addr[1])
// loop 2-1-3 % 4 = 2 times
send (addr[1]) || recv (addr[0])
send (addr[0]) || recv (addr[3])

msg = addr[3]
```

**Proc q=0**

```
recv (addr[1])
// loop 2-1-2 % 4 = 1 time
send (addr[1]) || recv (addr[0])

msg = addr[0]
```

**Proc q=1**

```
// loop 2-1-1 % 4 = 0 time
recv (addr[1])

msg = addr[1]
```

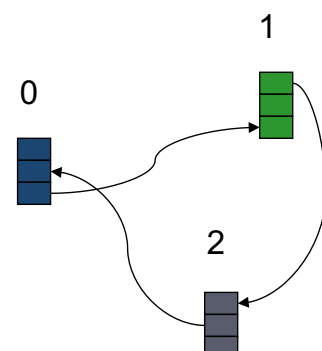
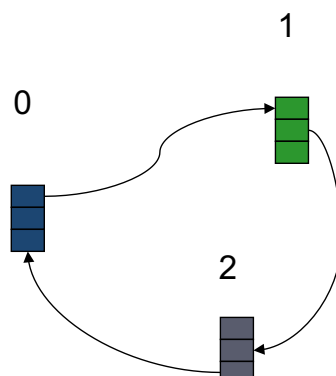


# All-to-all

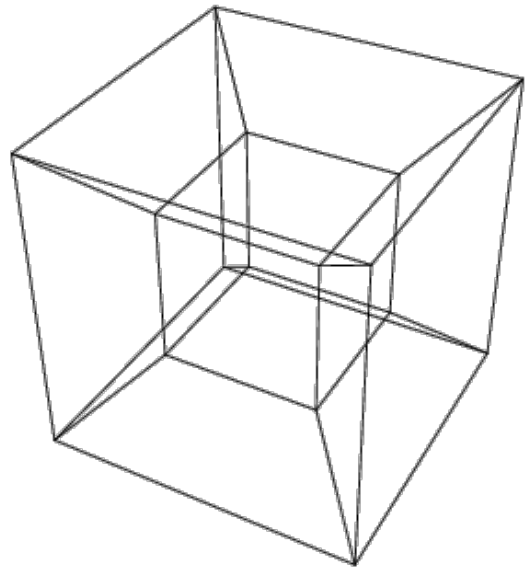
**All2All(my\_addr, addr, m)**

```
q = MY_NUM()
p = NUM_PROCS()
addr[q] ← my_addr
for i = 1 to p-1
  SEND(addr[q-i+1 mod p],m) || RECV(addr[q-i mod p],m)
```

Same execution time than  
scatter  
 $(p-1)(L + m b)$



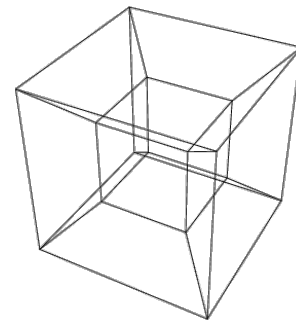
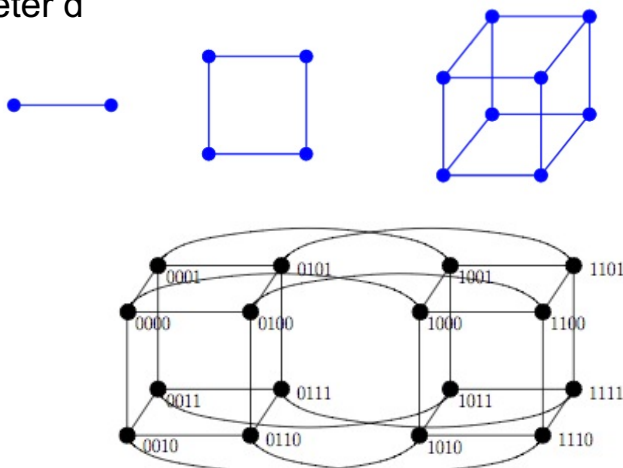




# ALGORITHMS ON HYPERCUBE

## Reminder on hypercubes

- $d$  dimensional graph
- $2^d$  nodes with  $d$  neighbor each
- A 0-cube is a simple node simple, a 1-cube a row of processors, a 2-cube a mesh, etc
- $\log(p)$  dimensions if  $p$  processors
- Diameter  $d$



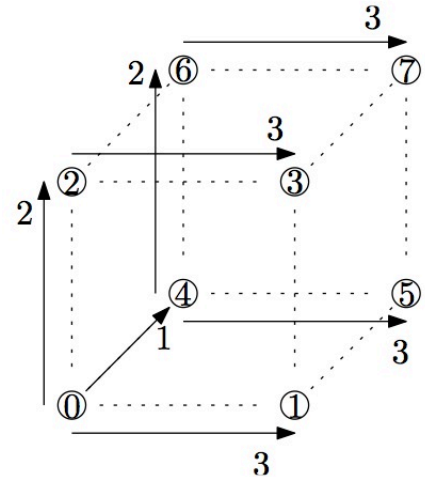
## Broadcast in hypercubes

Same algorithm as the ring one but generalized to  $d$  dimensions

```

1: Assume that  $p = 2^d$ 
2:  $\text{mask} \leftarrow 2^d - 1$  (set all bits)
3: for  $k = d - 1, d - 2, \dots, 0$  do
4:    $\text{mask} \leftarrow \text{mask} \text{ XOR } 2^k$  (clear bit  $k$ )
5:   if  $\text{me AND mask} = 0$  then
6:     (lower  $k$  bits of  $\text{me}$  are 0)
7:      $\text{partner} \leftarrow \text{me XOR } 2^k$  (partner has opposite bit  $k$ )
8:     if  $\text{me AND } 2^k = 0$  then
9:       Send  $M$  to partner
10:    else
11:      Receive  $M$  from partner
12:    end if
13:  end if
14: end for

```



If the root process is not 0

rename processes  $\text{me} = \text{me XOR root}$



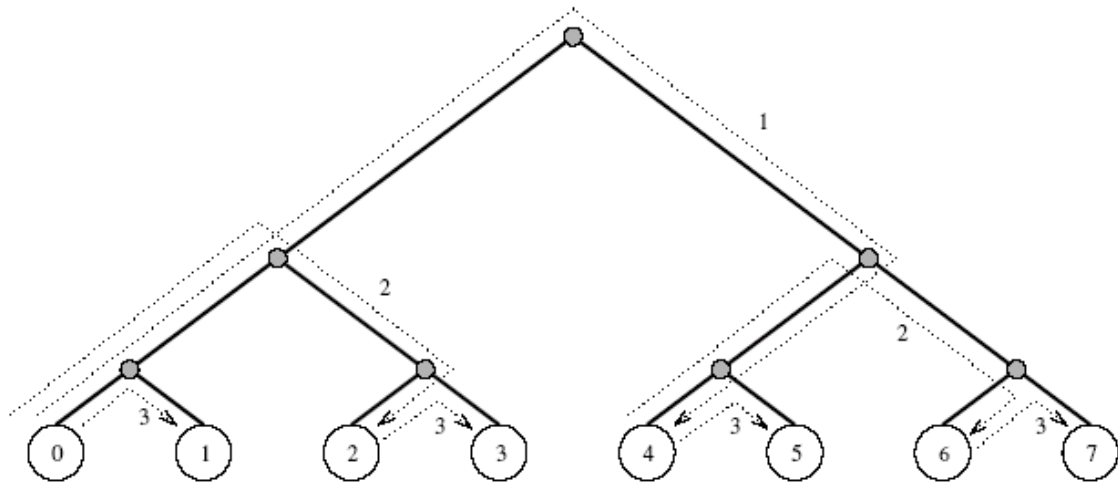
## Broadcast cost

- Number of steps:  $d = \log_2(p)$
- Cost per step:  $L + m/B$
- Total cost:  $(L + m/B) \log_2(p)$

The broadcast cost with  $p^2$  processors is only the double of the broadcast cost with  $p$  processors ( $\log_2(p^2) = 2 \log_2(p)$ )



## Broadcast in a binary tree



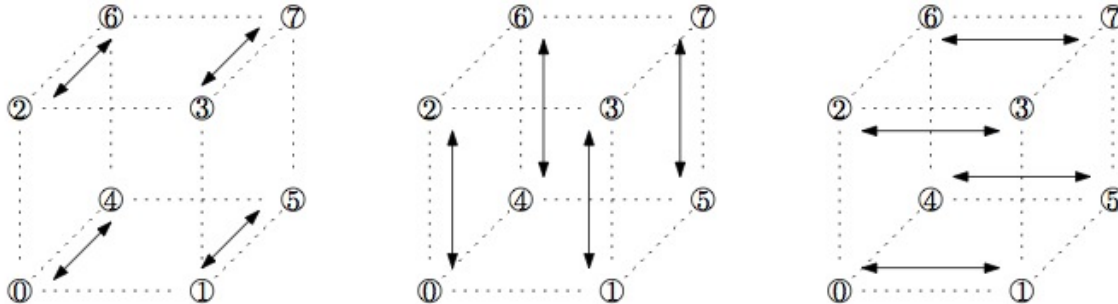
## Reduction (all-to-one)

- Same algorithm as broadcast but reversing the communication order and directions
- Same execution time (adding the reduction cost)
- Combining the incoming message with the local data with the operation

# All-to-all broadcast in a hypercube

## Using the ring algorithm

- For each dimension  $d$  of the hypercube, apply in sequence the algorithm on a ring on the  $2^{d-1}$  links of the current dimension in parallel



# All-to-all broadcast in a hypercube

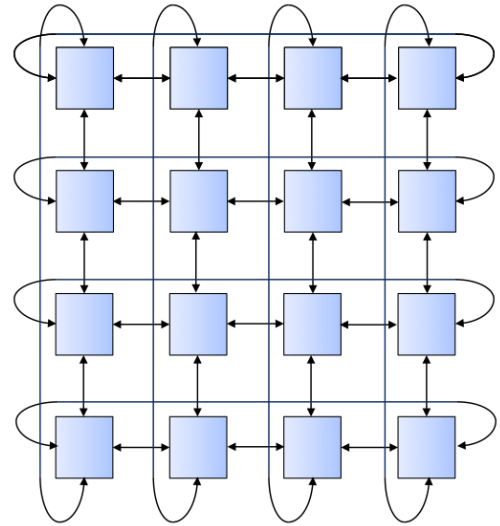
## • Cost

- Number of steps:  $d = \log_2(p)$

- Cost for step  $k = 0, 1, \dots, d-1$ :  $L + \frac{m2^k}{B}$

- Total cost:

$$\sum_{k=0}^{d-1} (L + 2^k \frac{m}{B}) = \log_2(p) * L + (p - 1) * \frac{m}{B}$$



# ALGORITHMS ON A GRID OF PROCESSORS

## Bi-dimensional grid of processors

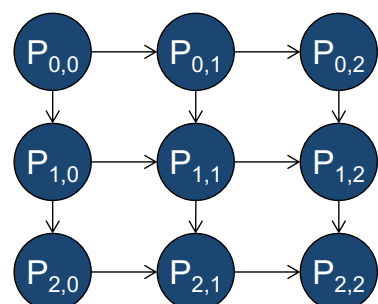
Let  $p = q^2$  processors

They can be seen as being arranged in the form of a square grid

- One can also have a rectangular grid

**Each processor is identified by two indexes**

- i: its row
- j: its column



## Bi-dimensional torus (2D torus)

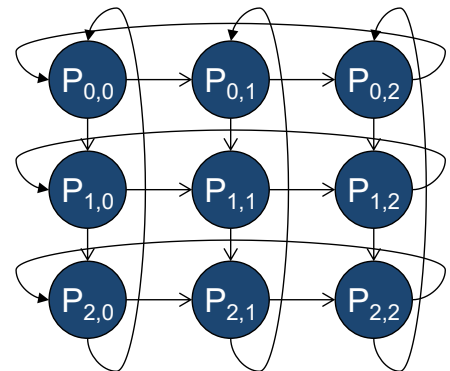
We have links which connect each side of the grid

Each processor belongs to two different rings

- Possibility to use algorithms designed for ring topologies

Mono-directional or bi-directional links

- Depends on what we need for our algorithm and/or physical resources



## Overlaps

In our performance analyzes, it is often assumed that a processor can perform **three activities in parallel**

- Computation
- Send
- Receive

It is also necessary to know whether the links are bi-directional or not

- **Two models**

- **Half-duplex:** two messages on the same link going in opposite directions share the link bandwidth
- **Full-duplex:** it's like having two links between each processor

- To be checked (and to measure and verify sometimes) with the target platform

## Multiple concurrent communications?

- We now have four (logical) links on each processor
- You need to know how many concurrent calls can be made at the same time
  - There can be 4 sends and 4 receives in the model with bi-directional links
  - Assuming that the 4 sends and the 4 receives can take place in parallel, one has a **multi-port model**
  - If we assume 1 send and 1 receive in parallel, we have a **1-port model**
  - Other possible variations
    - k-port (bounded multi-port), inputs/outputs

## Next

### We have several options

- Grid or torus
  - Mono- ou bi-directional links
  - 1-port or multi-port (or k-port)
  - Half- or full-duplex
- 
- We will generally assume a bi-directional and full-duplex torus
  - We will examine the 1-port and multi-port assumptions

"Easy" to modify a performance analysis to stick with the physical resources of the target machines studied

## Is the grid topology realistic?

Some parallel machines are(were) built with physical networks in the form of grids (2D or 3D)

- Examples: Intel Paragon, IBM's Blue Gene/L

If the platform uses a switch with all-to-all communications, then the grid is assumed to be valid

- On the other hand, the assumptions of full-duplex or multi-port are not necessarily valid

We will see that even if the physical platform is a unique shared medium (such as a non-switched Ethernet network), it is sometimes better to think of it as a grid when developing algorithms!



## Communications in a grid

- A process can call two functions to know its position in the grid:

`My_Proc_Row()` and `My_Proc_Col()`

- A process can know how many total processes are in the topology with:

`Num_Procs()`

- Assume that we have a square grid

- There are two point-to-point communications functions:

`Send(dest, addr, L)`

`Recv(src, addr, L)`

- Broadcast functions can be created in rows and columns

`BroadcastRow(i, j, srcaddr, dstaddr, L)`

`BroadcastCol(i, j, srcaddr, dstaddr, L)`

- It is assumed that a call to such a function in a row or column that is not right returns immediately





## Row and column broadcast

### If we have a torus

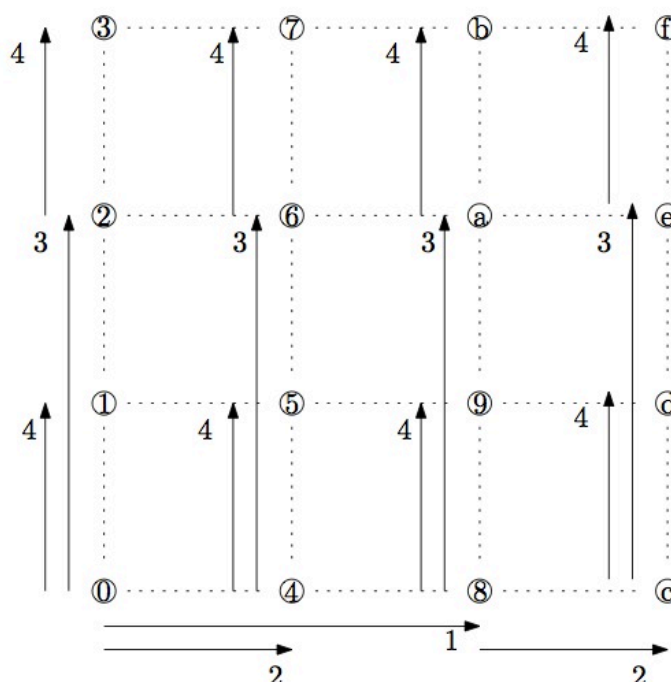
- If one has mono-directional links, one can re-use the broadcast function developed for the rings of processors
- Pipelined or not
- If you have bi-directional links and a multi-port model, you can improve performance by sending data on both sides of the ring
- Asymptotic performances are not changed

### If you have a grid

- If the links are bi-directional, then we can send the messages on both sides from the source processor concurrently or not, depending on whether we have a 1-port or multi-port model
- If the links are mono-directional, one can simply not implement the broadcast

## Broadcast in a grid

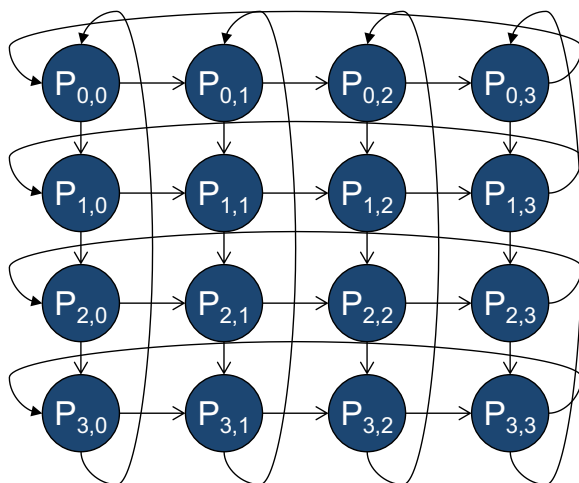
- Use the ring broadcast algorithm on the row where the root is located
- Use the ring broadcast algorithm on all columns in //



## All-to-all in a grid of processors

- Use the ring broadcast algorithm on each row in //
  - Cost (we suppose that we have a  $\sqrt{p} * \sqrt{p}$  grid of processors)
    - Number of steps:  $\sqrt{p} - 1$
    - Time per step:  $L + \frac{m}{B}$
    - Total time:  $(\sqrt{p} - 1) * \left(L + \frac{m}{B}\right)$
- Use the ring broadcast algorithm on each column in //
  - Cost
    - Number of steps:  $\sqrt{p} - 1$
    - Time per step:  $L + \sqrt{p} \frac{m}{B}$
    - Total time:  $(\sqrt{p} - 1) * \left(L + \frac{m}{B}\right)$
- Total time:
 
$$2 * (\sqrt{p} - 1) * L + (p - 1) * \frac{m}{B}$$

## Bi-dimensional matrix distribution



- Let  $a_{i,j}$  be a element of the matrix
- We denote by  $A_{i,j}$  (or  $A_{ij}$ ) the block of matrix A assigned to  $P_{i,j}$

$C_{00}$	$C_{01}$	$C_{02}$	$C_{03}$
$C_{10}$	$C_{11}$	$C_{12}$	$C_{13}$
$C_{20}$	$C_{21}$	$C_{22}$	$C_{23}$
$C_{30}$	$C_{31}$	$C_{32}$	$C_{33}$

$A_{00}$	$A_{01}$	$A_{02}$	$A_{03}$
$A_{10}$	$A_{11}$	$A_{12}$	$A_{13}$
$A_{20}$	$A_{21}$	$A_{22}$	$A_{23}$
$A_{30}$	$A_{31}$	$A_{32}$	$A_{33}$

$B_{00}$	$B_{01}$	$B_{02}$	$B_{03}$
$B_{10}$	$B_{11}$	$B_{12}$	$B_{13}$
$B_{20}$	$B_{21}$	$B_{22}$	$B_{23}$
$B_{30}$	$B_{31}$	$B_{32}$	$B_{33}$

# Cannon matrix product algorithm

Old algorithm

- Designed for systolic architectures (SIMD)
- Adapted to a 2D grid

The algorithm starts with a redistribution of matrices A and B

- Called "*preskewing*"

Then matrices are multiplied together

At the end, the matrices are re-distributed to find their initial distribution

- Called "*postskewing*"



## Cannon Preskewing

### Matrix A

Each block of matrix A is shifted to the left until the process of the first process column contains a block of the diagonal of the matrix

$A_{00}$	$A_{01}$	$A_{02}$	$A_{03}$
$A_{10}$	$A_{11}$	$A_{12}$	$A_{13}$
$A_{20}$	$A_{21}$	$A_{22}$	$A_{23}$
$A_{30}$	$A_{31}$	$A_{32}$	$A_{33}$

$A_{00}$	$A_{01}$	$A_{02}$	$A_{03}$
$A_{11}$	$A_{12}$	$A_{13}$	$A_{10}$
$A_{22}$	$A_{23}$	$A_{20}$	$A_{21}$
$A_{33}$	$A_{30}$	$A_{31}$	$A_{32}$

