Parallel Computing and the MPI environment

Claudio Chiaruttini
Dipartimento di Matematica e Informatica
Centro Interdipartimentale per le Scienze Computazionali (CISC)
Università di Trieste
http://www.dmi.units.it/~chiarutt/didattica/parallela

Summary

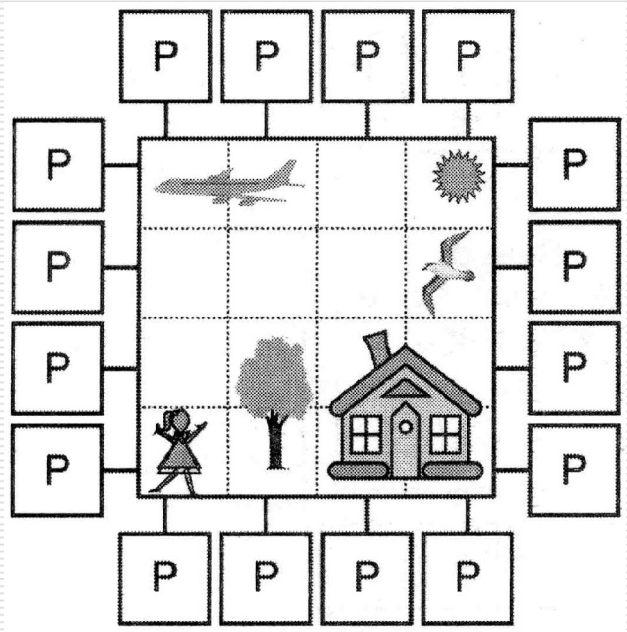
1) Parallel computation, why?
2) Parallel computer architectures
3) Computational paradigms
4) Execution Speedup
5) The MPI environment
6) Distributing arrays among processors

Why parallel computing?

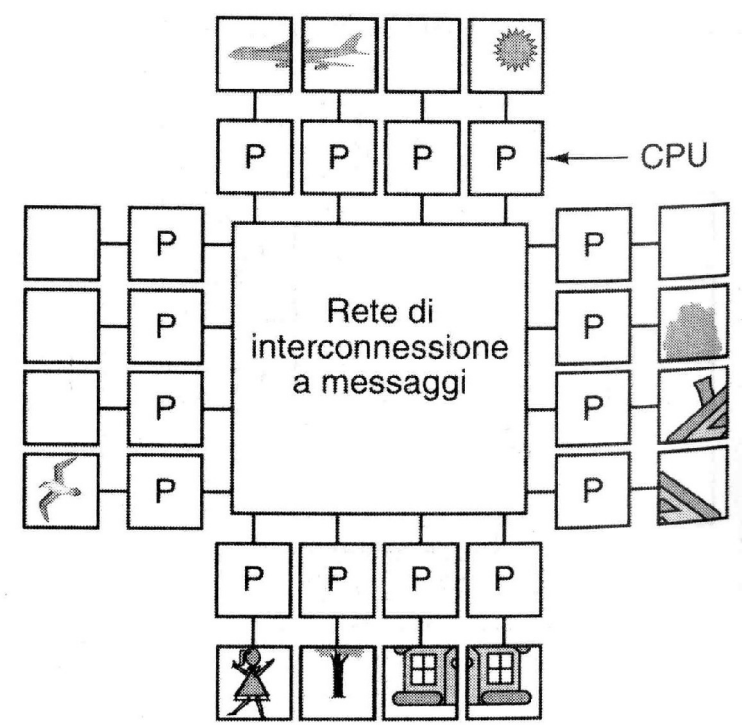
- Solve problems with greater speed
- Run memory demanding programs

Parallel architecture models

Shared memory



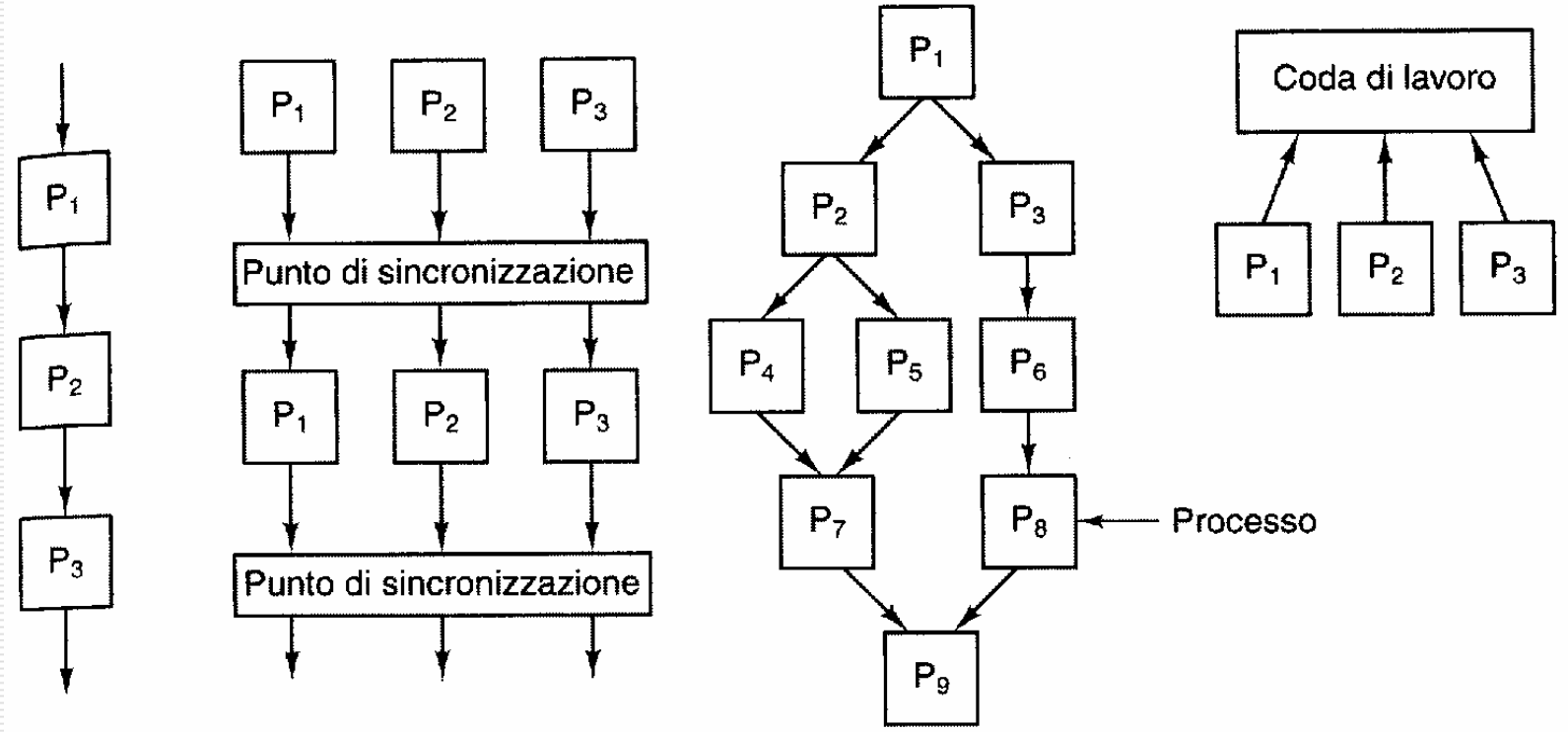
Distributed memory
(Message passing)



Shared vs. Distributed Memory

- Parallel threads in a single process
- Easy programming: extensions to standard languages (OpenMP)
- Several communicating processes
- Difficult programming: special libraries needed (MPI)
- The programmer must explicitely take care of message passing

Computational paradigms



Pipeline Phases Divide & Conquer Master–Worker

Execution Speedup

- Definition (for p processors): $Speedup = T_1 / T_p$
- Amdahl law

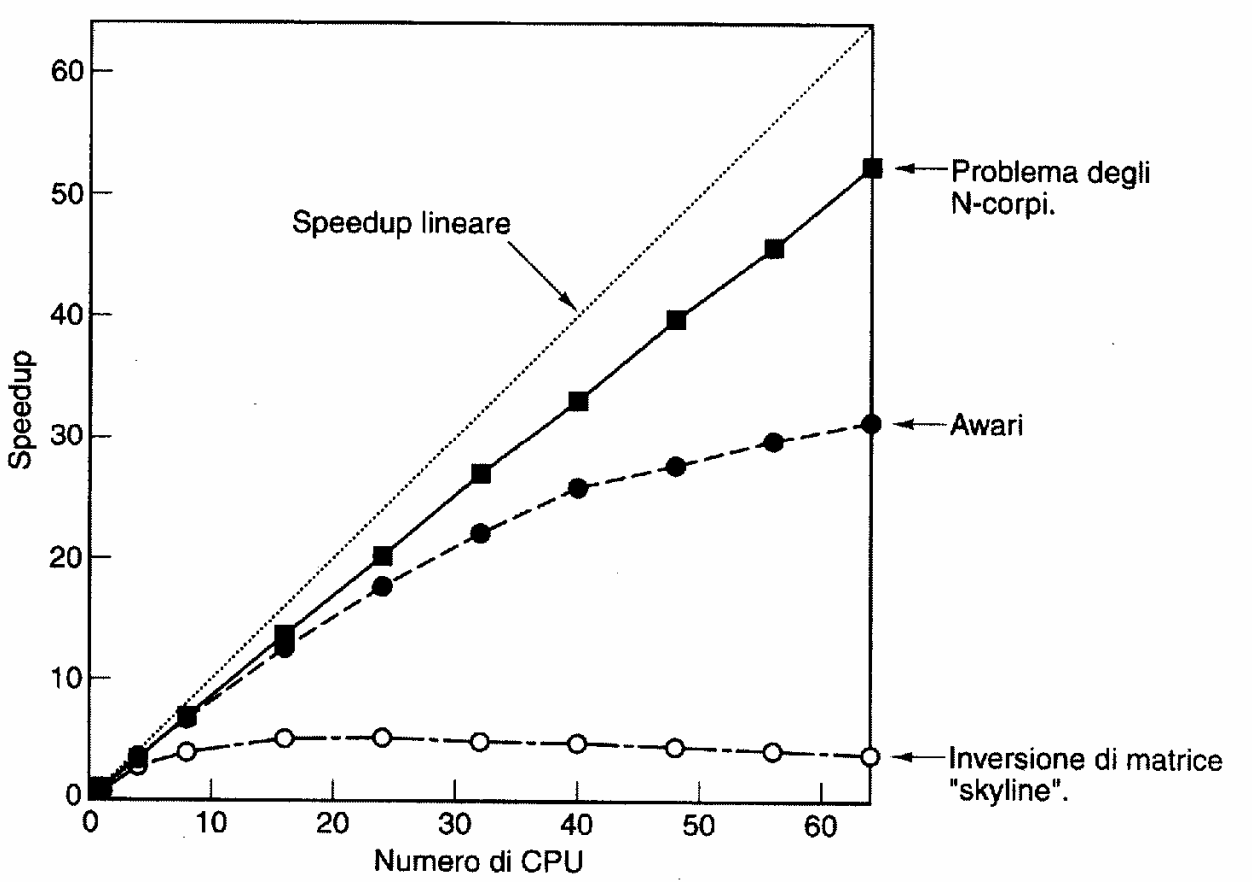
  f: sequential fraction of execution time

$$Speedup = \frac{p}{1+(p-1)f}$$

Maximum S*peedup* : 1/f , for $p \rightarrow \infty$

seq. fraction f	max. speedup
10%	10
1%	100

Scalability of algorithms



Other metrics

- Speedup
  how much do we gain in time
  $S_p = T_1 / T_p \qquad \geq 1$
- Efficiency
  how much do we use the machine
  $E_p = S_p / p \qquad \leq 1$
- Cost
  $C_p = pT_p$
- Effectiveness
  benefit / cost ratio
  $F_p = S_p / C_p = E_p S_p / T_1$

Programming models

- A simple program

```
DO i=1,n
  s(i)=sin(a(i))
  r(i)=sqrt(a(i))
  l(i)=log(a(i))
  t(i)=tan(a(i))
END DO
```

- Functional decomposition:
  each process computes one function on all data
- Domain decomposition:
  each process computes all functions on a chunk of data
  → Scales well

MPI: Message Passing Interface

- **Message structure:**
  Content: data, count, type
  Envelope: source/dest, tag, communicator
- **Basic functions:**
  MPI_SEND: data to destination
  MPI_RECV: data from source

Basic functions

```
PROGRAM Hello_world
! REMARK: no communication here!

INCLUDE "mpif.h" ! MPI library
INTEGER:: err

CALL MPI_INIT(err)
  ! Initialize communication

WRITE (*,*) "Hello world!"

CALL MPI_FINALIZE(err)
  ! Finalize communication
END Hello_world
```

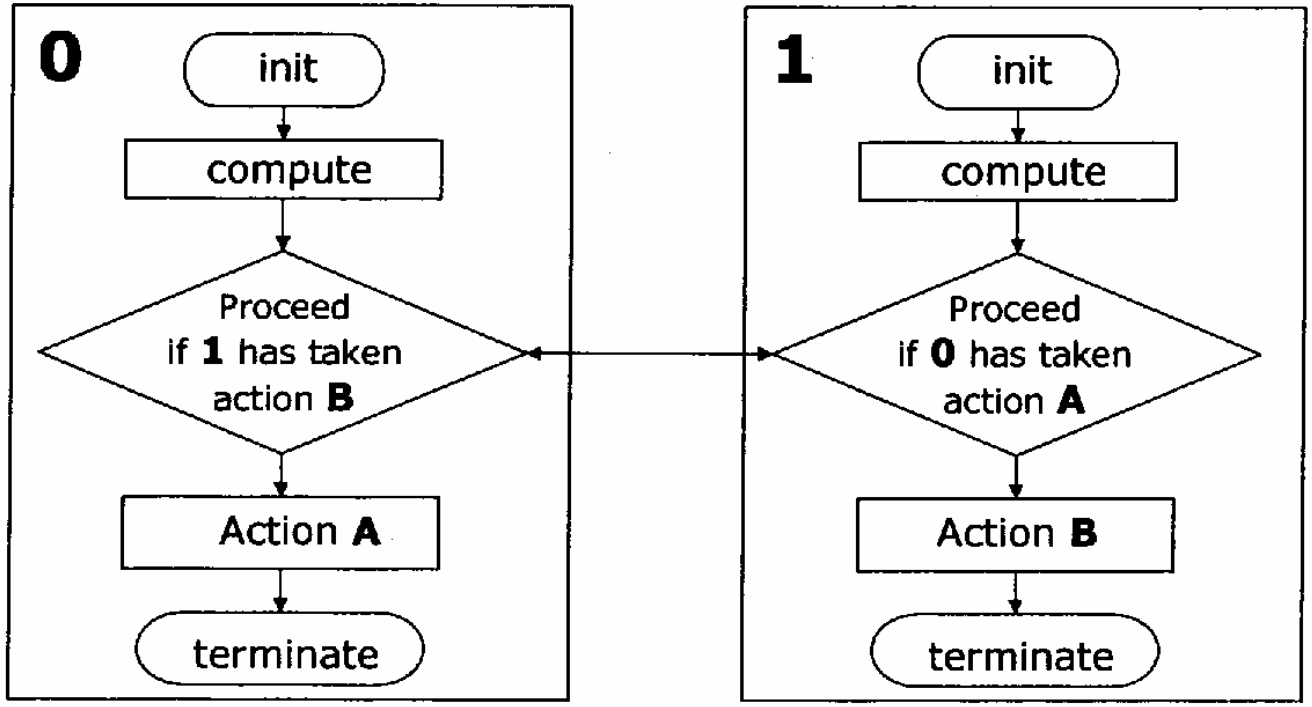
```
PROGRAM Hello_from
! Each process has its own rank

INCLUDE "mpif.h"
INTEGER:: err, nproc, myid

CALL MPI_INIT (err)
CALL MPI_COMM_SIZE  &
      (MPI_COMM_WORLD, nproc,err)
   ! get the total number of processes
CALL MPI_COMM_RANK  &
      (MPI_COMM_WORLD, myid, err)
   ! get the process rank

WRITE(*,*) "Hello from",myid," of",nproc

CALL MPI_FINALIZE(err)
END Hello_from
```

Sending and receiving messages

```
PROGRAM Send_Receive
.....
INTEGER:: err, nproc, myid
INTEGER,DIMENSION(MPI_STATUS_SIZE):: status
REAL,DIMENSION(2):: a
  .....
IF (myid==0)  THEN
  a = (/ 1, 2 /)          ! Process # 0 holds the data
  CALL MPI_SEND (a, 2, MPI_REAL,  &! Content: buffer, count and type
                      1, 10, MPI_COMM_WORLD, err) &! Envelope
ELSE IF (myid==1)  THEN
  CALL MPI_RECV (a, 2, MPI_REAL,        &! Data buffer, count and type
                     0, 10, MPI_COMM_WORLD, status, err)  &! Envelope
  WRITE (*,*) myid, ": a =", a
END IF
.....
END PROGRAM Send_Receive
```

Deadlocks

Deadlock occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress.



Avoiding deadlocks 1

```
PROGRAM Deadlock
.....
INTEGER, PARAMETER:: N=100000
INTEGER:: err, nproc, myid
REAL:: a(N), b(N)
.....
IF (myid==0) THEN
  a = 0
  CALL MPI_SEND (a, N, MPI_REAL, 1, &
       10, MPI_COMM_WORLD, err)
  CALL MPI_RECV (b, N, MPI_REAL, 1, &
       11, MPI_COMM_WORLD, status, err)
ELSE IF (myid==1)  THEN
  a = 1
  CALL MPI_SEND (a, N, MPI_REAL, 0, &
       11, MPI_COMM_WORLD, err)
  CALL MPI_RECV (b, N, MPI_REAL, 0, &
       10, MPI_COMM_WORLD, status, err)
END IF
.....
END PROGRAM Deadlock
```

```
PROGRAM NO_Deadlock
! ... by breaking symmetry
.....
INTEGER, PARAMETER:: N=100000
INTEGER:: err, nproc, myid
REAL:: a(N), b(N)
 .....
IF (myid==0) THEN
  a = 0
  CALL MPI_SEND (a, N, MPI_REAL, 1, &
       10, MPI_COMM_WORLD, err)
  CALL MPI_RECV (b, N, MPI_REAL, 1, &
       11, MPI_COMM_WORLD, status, err)
ELSE IF (myid==1) THEN
  a = 1
  CALL MPI_RECV (b, N, MPI_REAL, 0, &
       10, MPI_COMM_WORLD, status, err)
  CALL MPI_SEND (a, N, MPI_REAL, 0, &
       11, MPI_COMM_WORLD, err)
END IF
.....
END PROGRAM NO_Deadlock
```

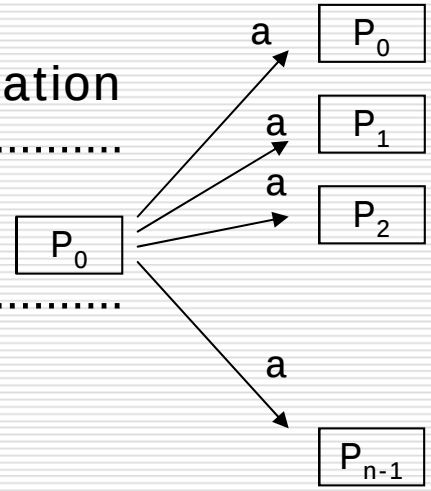
Avoiding deadlocks 2

```
PROGRAM SendRecv
.....
INTEGER, PARAMETER:: N=100000
REAL:: a(N), b(N)
.....
IF (myid==0) THEN
  a = 0
  CALL MPI_SENDRECV (a, N, MPI_REAL, 1, 10   &
                     ! sent data, count, type, destination, tag
                       , b, N, MPI_REAL, 1, 11   &
                     ! received data, count, type, source, tag
                       , MPI_COMM_WORLD, status, err)
ELSE IF (myid==1) THEN
  a = 1
  CALL MPI_SENDRECV (a, N, MPI_REAL, 0, 11   &
                       , b, N, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, err)
                    ! NOTE the taga correspondence
END IF
WRITE(*,*) myid, " received:",  b(1:5), "..."
.....
END PROGRAM SendRecv
```

Overlapping communication and computation

- Start communication in advance, non-blocking send/receive
- Sinchronization to ensure transfer completion

One-to-all: Broadcast



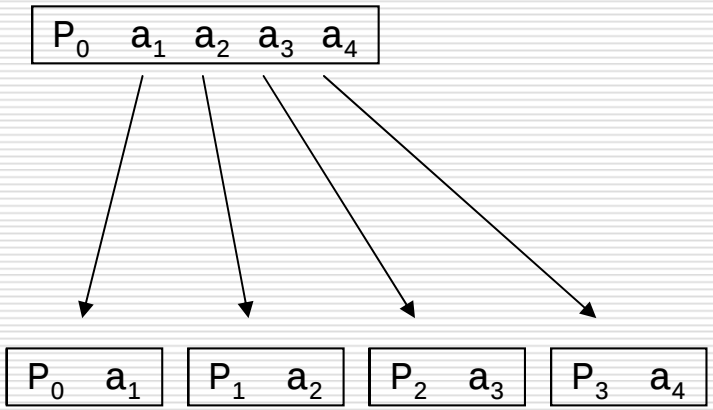

```
PROGRAM Broadcast  ! One-to-many communication
.....
INTEGER:: err, nproc, myid
INTEGER:: root, i, a(10)
.....


root = 0
IF (myid==root)  a = (/ (i, i=,10) /)
CALL MPI_BCAST (a,10,MPI_INTEGER, &! s/d buffs, count,type
                root, MPI_COMM_WORLD, err) ! source, comm.
! REMARK: source and destination buffers have the
      same name, but are in different processor memories
.....
END PROGRAM Broadcast
```

One-to-all: Scatter/ Gather (1)

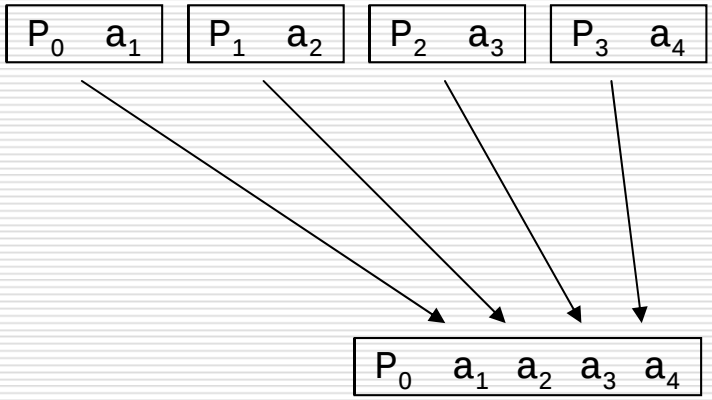
```
PROGRAM ScatterGather
! SCATTER: distribute an array among processes
! GATHER: collect a distributed array in a single process
.....
INTEGER, PARAMETER:: N=16
INTEGER:: err, nproc, myid
INTEGER:: root, i, m, a(N), b(N)
.....
root = 0
m = N/nproc        ! number of elements per process
IF (myid==root)  a = (/ (i, i=1,N) /)
CALL MPI_SCATTER (a, m, MPI_INTEGER,            & ! source buffer
                  b, m, MPI_INTEGER,            & ! destination buffer
                  root, MPI_COMM_WORLD, err) ! source process, communicator
b(1:m) = 2*b(1:m)                                  ! Parallel function computation
CALL MPI_GATHER (b, m, MPI_INTEGER,            &! source buffer
                 a, m, MPI_INTEGER,            &! destination buffer
                 root, MPI_COMM_WORLD, err) !destination process, communicator
  .....
END PROGRAM ScatterGather
```

Scatter/ Gather (2)

Scatter



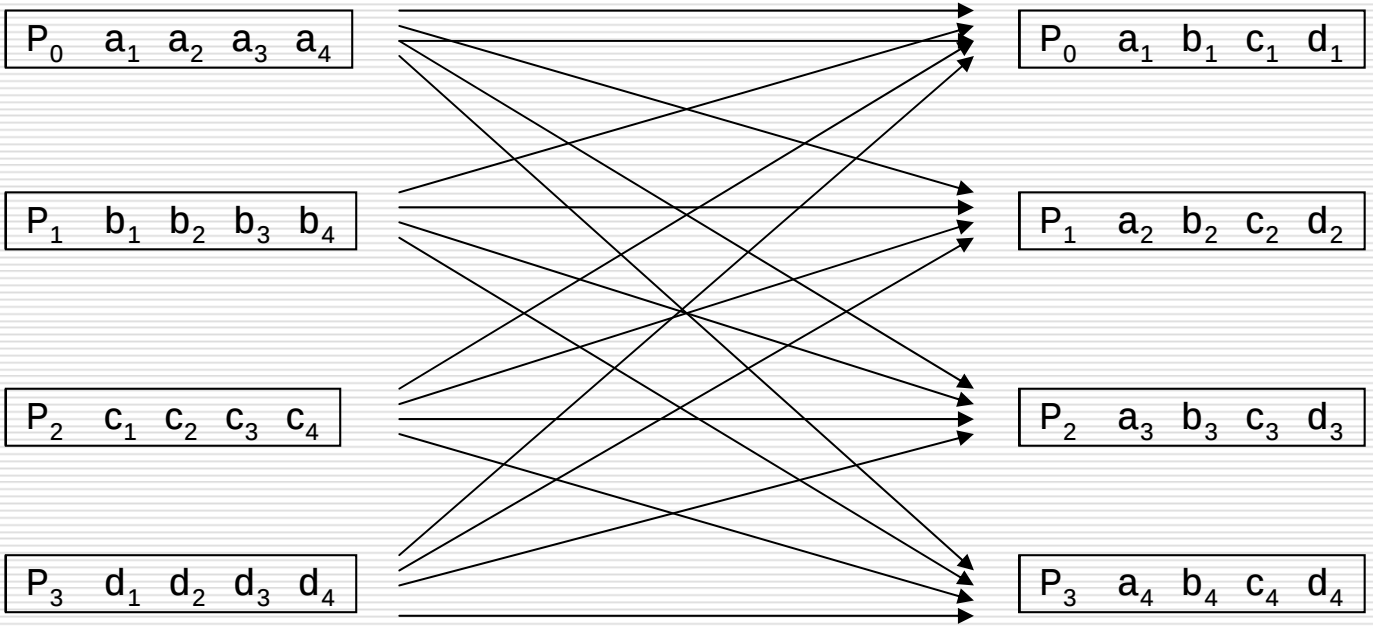
Gather



All-to-all: MPI_Alltoall (1)

```
PROGRAM All_to_all
!------------------------------------------------------------------
! Exchange data among all processors
!------------------------------------------------------------------
.....
INTEGER,PARAMETER:: N=4
INTEGER:: i, m
REAL,DIMENSION(N):: a
.....
a = (/ (N*myid+i, i=1,N) /)
WRITE(*,*) "process", myid, " has:", a
m = N/nproc
CALL MPI_ALLTOALL (a, m, MPI_REAL, & ! Sender
                   a, m, MPI_REAL, & ! Receiver
!                  buff, count, type
                   MPI_COMM_WORLD, err)
! REMARK: count is the number of elements sent from one
!   process to the other
WRITE(*,*) "Process", myid, "received:", a
.....
END PROGRAM All_to_all
```

All-to-all: MPI_Alltoall (2)



Reduction functions (1)

```
PROGRAM Reduce
!-----------------------------------------------------------------------
! Parallel sum, an instance of MPI reduction functions
!-----------------------------------------------------------------------
.....
INTEGER,PARAMETER:: N=4
INTEGER:: i, m, root
REAL,DIMENSION(N):: a, s
.....
a = (/ (i, i=1,N) /)
WRITE(*,*) "process", myid, " has:", a
root = 0
CALL MPI_REDUCE (a, s, N, MPI_REAL, & ! s/D buff., count, type
                 MPI_SUM, root,     & ! Operation, destination
                 MPI_COMM_WORLD, err)
! REMARK: dropping the "root" argument, ALL processors get the result
IF (myid==0) WRITE(*,*) "The sum is", s
.....
```

Reduction functions (2)

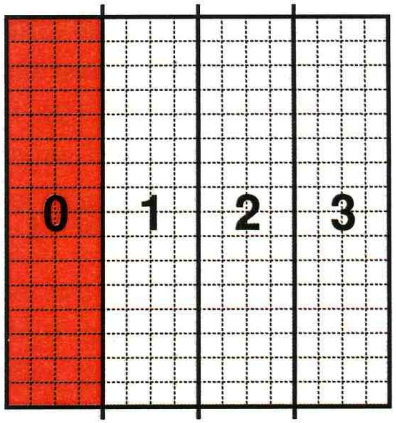
MPI_OP	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Location of maximum
MPI_MINLOC	Location of minimum
MPI_……	……

Distributing arrays among processors

ISSUES:

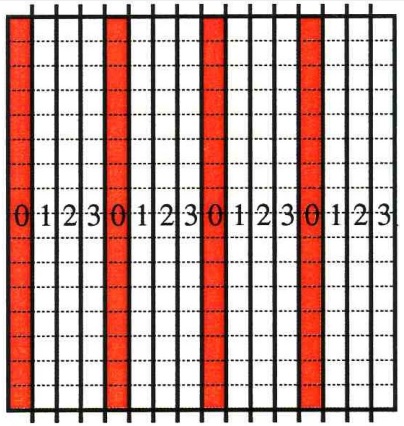
- Load balancing
- Communication optimization

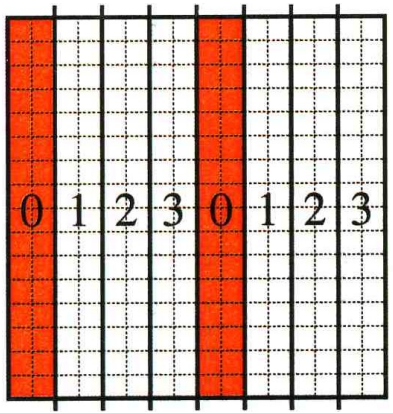
Array distribution 1



Colunm blocks

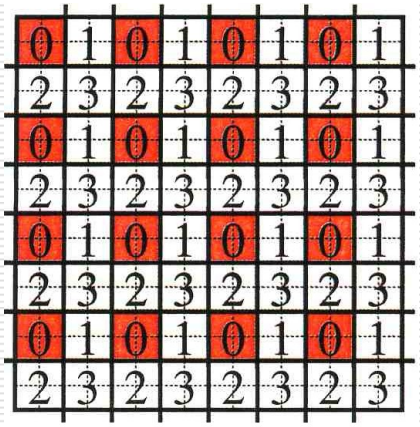
Column cycles





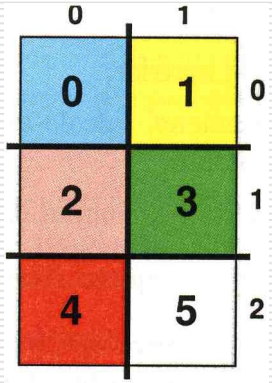
Colunm block cycles

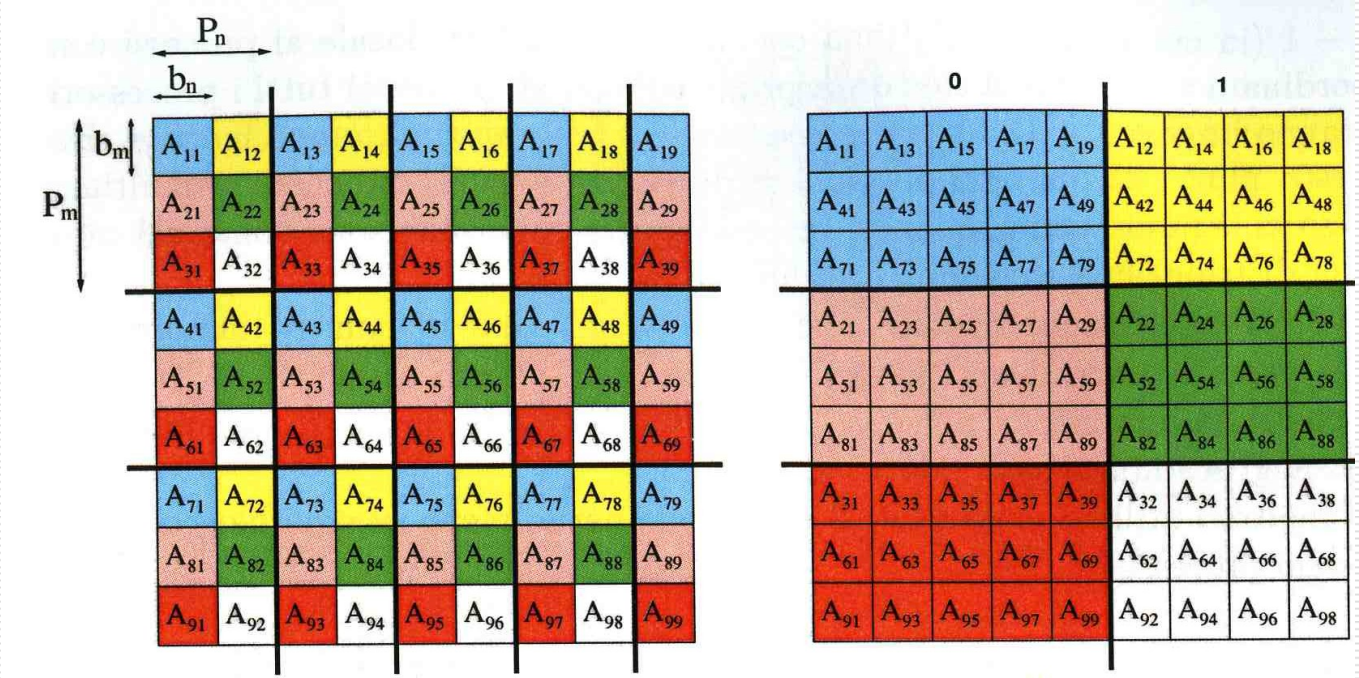
Row and colunm block cycles



Array distribution 2

Grid of processors





In summary

- MPI functions: low-level tools efficiently implemented
- Parallel high-level language compilers (HPF) need improvement
- Software libraries, e.g. ScaLAPACK for Linear Algebra
- Careful algorithm design is needed to exploit the hardware