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

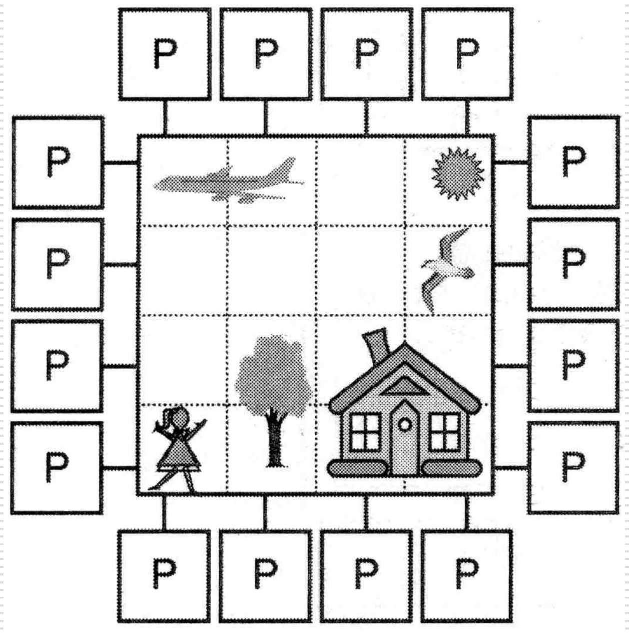
Why parallel computing?

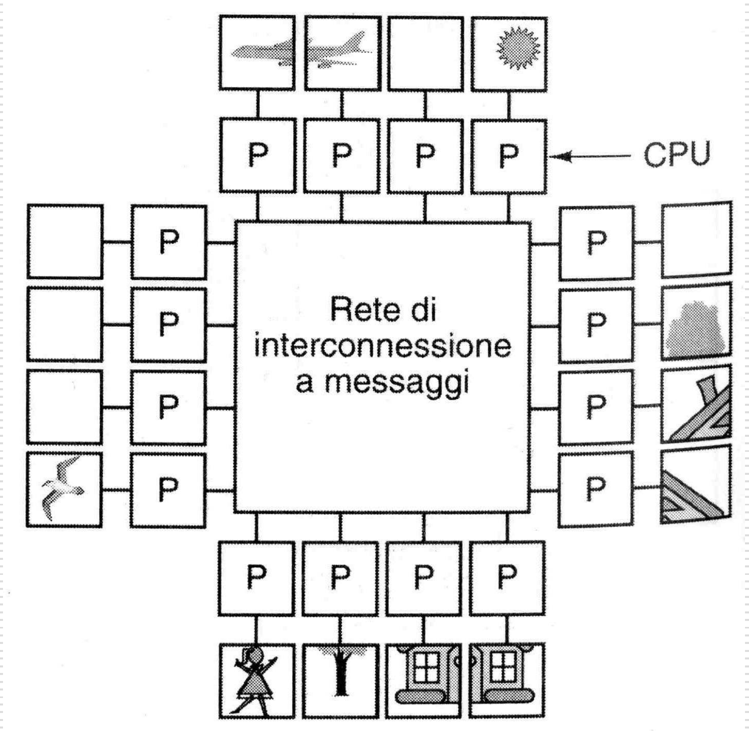
- Solve problems with greater speed
- Run memory demanding programs

Parallel architecture models

Shared memory

Distributed memory
(Message passing)



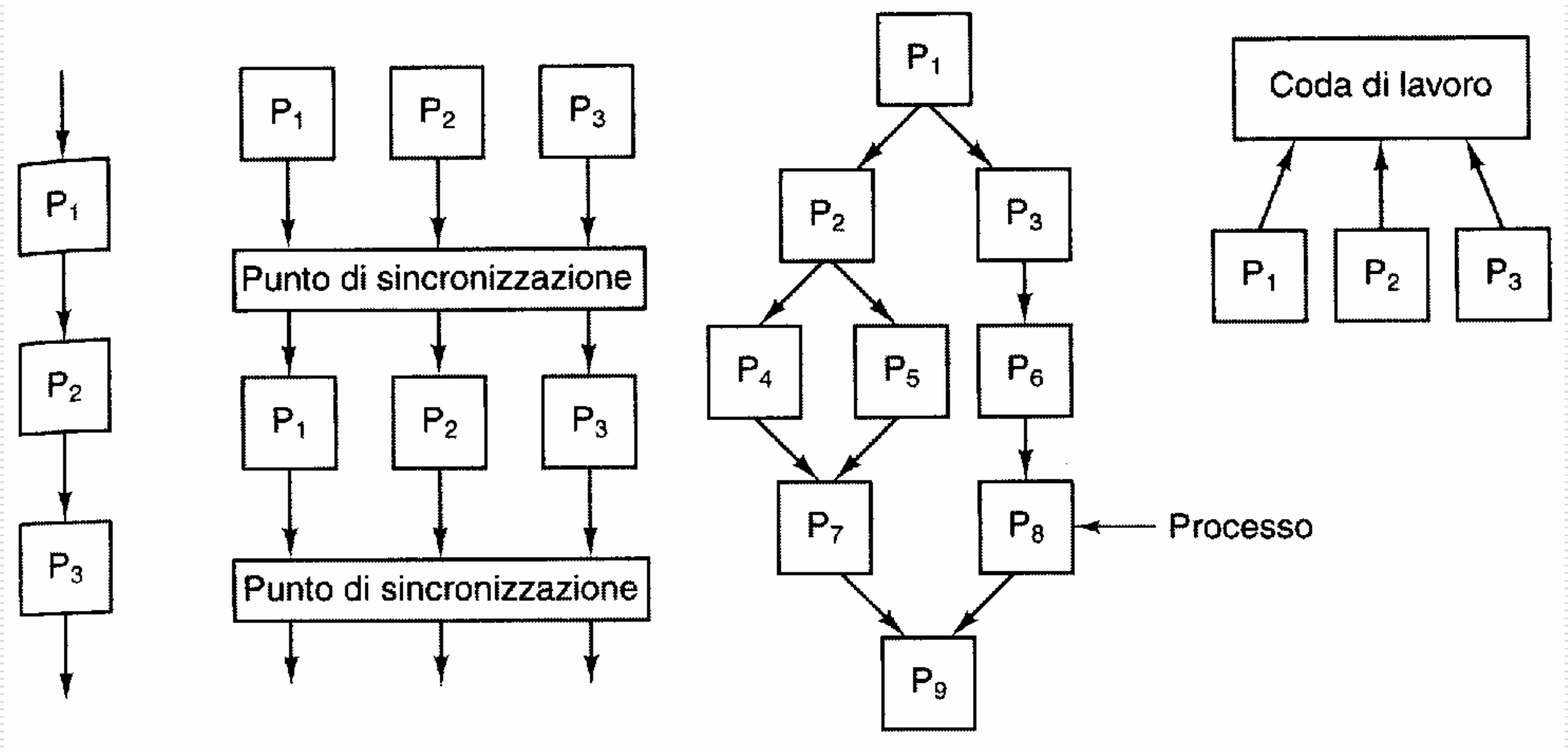


Shared vs. Distributed Memory

- 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 (1, Amdahl)

- Definition (for *N* processors): $Speedup = T_1 / T_N$
- Amdahl law

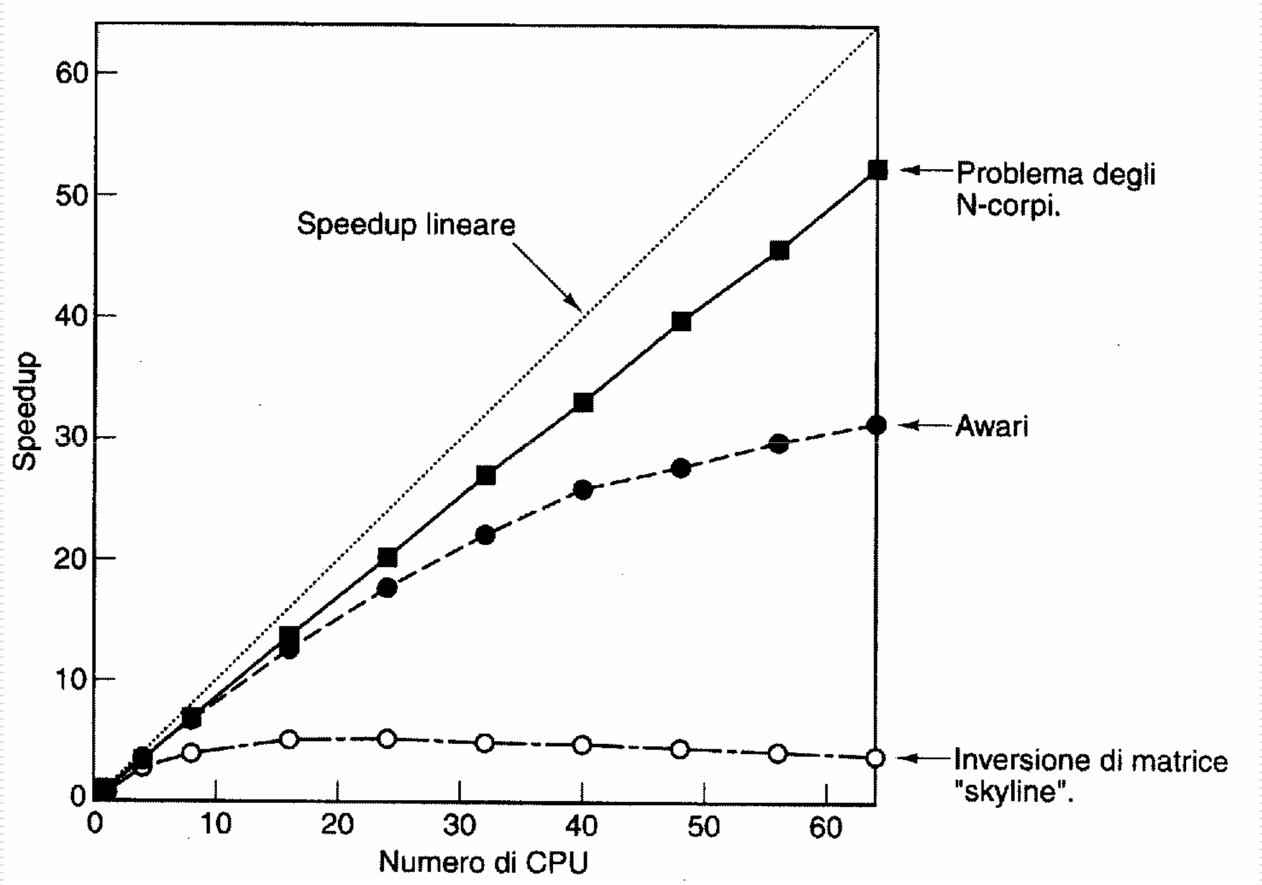
  s: sequential fraction of program

$$Speedup = \frac{N}{1+(N-1)s}$$

  Maximum Speedup : 1/*s* , for $N \rightarrow \infty$

seq. fraction *s*	max. Speedup
10%	10
1%	100

Scalability of algorithms



Execution Speedup (2, Gustafson)

- Amdahl: ***constant problem size*** (T_1)
- Actually:
  - the size of the problem scales linearly with the number of processors
  - the sequential fraction *s* remains nearly constant
- Gustafson: ***constant run time*** (T_N)
- Definition: $\textit{Scaled speedup} = s + (1 - s) \times N$

Execution Speedup (3)

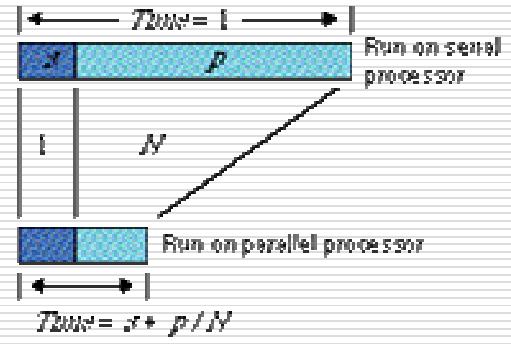


FIGURE 2a. Fixed-Size Model: Speedup = 1 / (s + p / N)

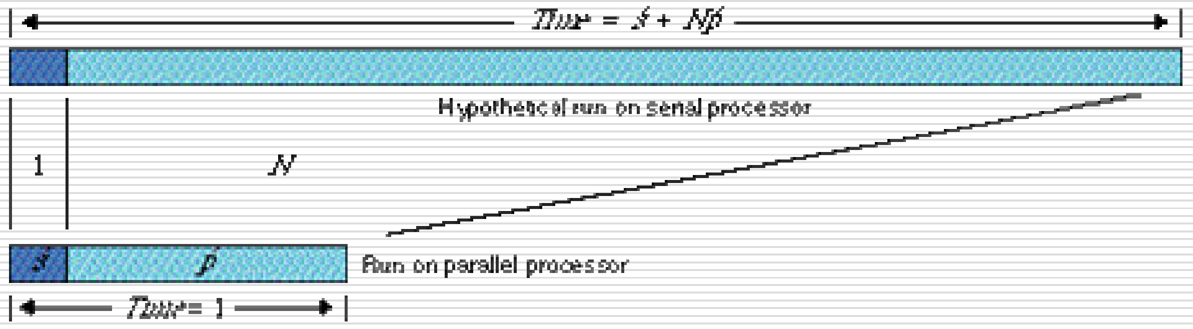

Time = s' + Np'
Hypothetical run on serial processor
1
N
s'
p'
Run on parallel processor
Time = 1

FIGURE 2b. Scaled-Size Model: Speedup = s' + Np'

Parallel performance metrics

- **Speedup** $S_N = T_1 / T_N \qquad \geq 1$
  how much do we gain in time
- **Efficiency** $E_N = S_N / N \qquad \leq 1$
  how much do we use the machine
- **Cost** $C_N = N \times T_N$
- **Effectiveness** $F_N = S_N / C_N = E_N \times S_N / T_1$
  benefit / cost ratio

Programming models

- A simple program

```
for (i=0; i<n; i++)
{ s[i]=sin(a[i]);
  r[i]=sqrt(a[i]);
  l[i]=log(a[i]);
  t[i]=tan(a[i]);
}
```

- 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

```
/* -----------------------------------*/
/* Hello world!                      */
/* -----------------------------------*/
/* REMARK: no communication!
*/
#include <stdio.h>
#include <mpi.h> /* MPI library */

int main (int argc, char *argv[])
{ int err;
  err = MPI_Init(&argc, &argv);
   /* initialize communication */

  printf("Hello world!\n");

  err = MPI_Finalize();
   /* finalize communication   */
}
```

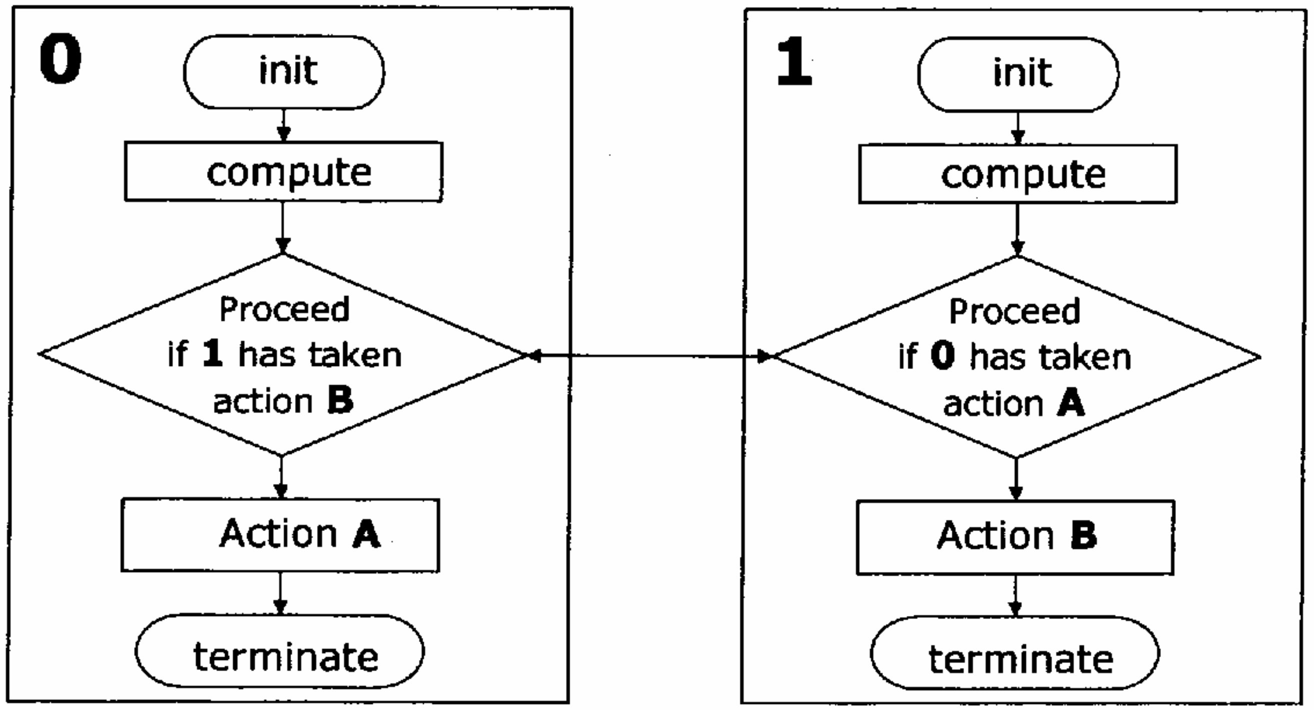
```
/*-----------------------------------------*/
/* Hello from ...                         */
/*-----------------------------------------*/
/* Each process has its own "rank"
*/
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[])
{ int err, nproc, myid;
   err = MPI_Init (&argc, &argv);
   err = MPI_Comm_size
        (MPI_COMM_WORLD, &nproc);
    /*get the total number of processes*/
   err = MPI_Comm_rank
        (MPI_COMM_WORLD, &myid);
    /* get the process rank */
  printf("Hello from %d of %d\n", myid,
        nproc);
  err = MPI_Finalize();
}
```

Sending and receiving messages

```
/*--------------------------------------------------------------------------------------*/
/* Sending and receiving messages                                                       */
/*--------------------------------------------------------------------------------------*/
……………………………………………………………………………………..
int main (int argc, char *argv[])
{ int err, nproc, myid;
  MPI_Status status;
  float a[2];
  ……………………………………………………………………………………
  if (myid==0) {a[0]=1, a[1]=2;         /* Process # 0 holds the data           */
    err = MPI_Send(a, 2, MPI_FLOAT,  /* Content: BUFFER, count and type */
                       1, 10, MPI_COMM_WORLD); /* Envelope                      */
  } else if (myid==1) {
    err = MPI_Recv(a, 2, MPI_FLOAT,         /* Data BUFFER, count and type */
                       0, 10, MPI_COMM_WORLD, &status); /* Envelope       */
    printf("%d: a[0]=%f a[1]=%f\n", myid, a[0], a[1]);
  }……………………………………………………………………………………
}
```

Deadlocks

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


0
init
compute
Proceed if 1 has taken action B
Action A
terminate
1
init
compute
Proceed if 0 has taken action A
Action B
terminate

Avoiding deadlocks 1

```
/*--------------------------------------*/
/* Deadlock                             */
/*--------------------------------------*/
…………………………………………
#define N 100000
int main (int argc, char * argv[])
{int err, nproc, myid;
  float a[N], b[N];
  ………………………………
  if (myid==0) {  a[0]=1, a[1]=2;
    MPI_Send(a, N, MPI_FLOAT, 1,
        10, MPI_COMM_WORLD);
    MPI_Recv(b, N, MPI_FLOAT, 1,
        11, MPI_COMM_WORLD, &status);
  } else if (myid==1) { a[0]=3, a[1]=4;
    MPI_Send (a, N, MPI_FLOAT, 0,
        11, MPI_COMM_WORLD);
    MPI_Recv (b, N, MPI_FLOAT, 0,
        10, MPI_COMM_WORLD, &status);
  }   ……………………………………
}
```

```
/*------------------------------------------*/
/* NO Deadlock                              */
/*------------------------------------------*/
…………………………………………………………
#define N 100000
int main (int argc, char * argv[])
{int err, nproc, myid;
  float a[N], b[N];
 ……………………………………………
  if (myid==0) { a[0]=1, a[1]=2;
    MPI_Send(a, N, MPI_FLOAT, 1,
        10, MPI_COMM_WORLD);
    MPI_Recv(b, N, MPI_FLOAT, 1,
        11, MPI_COMM_WORLD, &status);
  } else if (myid==1) {a[0]=3, a[1]=4;
    MPI_Recv(b, N, MPI_FLOAT, 0,
        10, MPI_COMM_WORLD, &status);
    MPI_Send(a, N, MPI_FLOAT, 0,
        11, MPI_COMM_WORLD);
 }    ……………………………………
}
```

Avoiding deadlocks 2

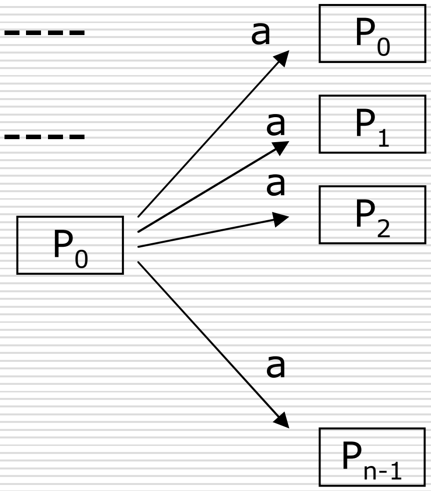
```
/*-------------------------------------------------------*/
/* Send/Receive without deadlocks                                          */
/*-------------------------------------------------------*/
………………………………………………………………………….
#define N 100000
int main (int argc, char * argv[])
{ ……………………………………………………………….
  float a[N], b[N];
  ……………………………………………………………………………….
  if (myid==0) { a[0]=1, a[1]=2;
  MPI_Sendrecv (a, N, MPI_FLOAT, 1, 10
                                  /* dati inviati, numero, tipo, destinatario, tag */
                          , b, N, MPI_FLOAT, 1, 11
                                  /* dati ricevuti, numero, tipo, mittente, tag */
                          , MPI_COMM_WORLD,&status);
  } else if (myid==1) { a[0]=3, a[1]=4;
       MPI_Sendrecv(a, N, MPI_FLOAT, 0, 11
                            , b, N, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &status);
                             /* NOTA: si ossevi la corrispondenza dei tag */
  }  printf("%d: b[0]=%f b[1]=%f\n", myid, b[0], b[1]);
…………………………………………………………………………………….
}
```

Overlapping communication and computation

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

One-to-many: Broadcast

```
//-------------------------------------------------
// BROADCAST: One-to-many communication
//-------------------------------------------------
……………………………………………………………………………………
int main (int argc, char *argv[])
{int err, nproc, myid;
  int root, a[2];
……………………………………………………………………………………
  root = 0;
  if(myid==root) a[0]=1, a[1]=2;
  err = MPI_Bcast (a, 2, MPI_INT,      // s/d buffers, count, type
                   root, MPI_COMM_WORLD); // source, comm.
  /* REMARK: source and destination buffers have the
       same name, but are in different processor memories */
  …………………………………………………………………………………….
}
```

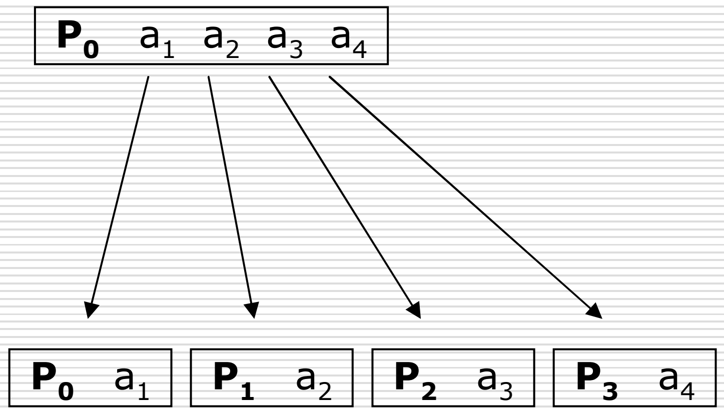


One-to-many: Scatter/Gather

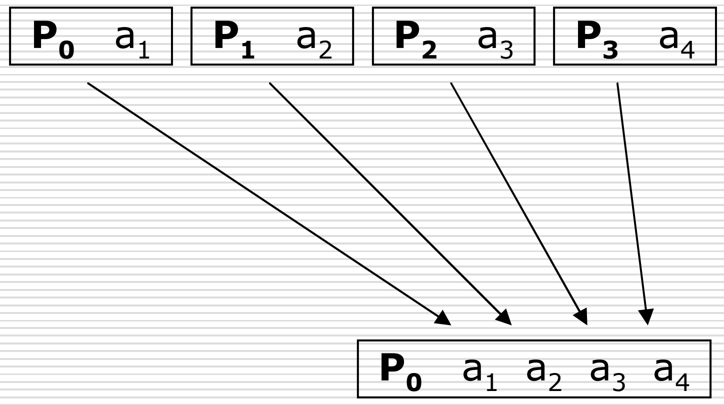
```
/*--------------------------------------------------------------------------------------*/
/* SCATTER: distribute an array among processes                                          */
/* GATHER: collect a distributed array in a single process                               */
/*--------------------------------------------------------------------------------------*/
……………………………………………………………………………………………………………
#define N 16
int main (int argc, char *argv[])
{int err, nproc, myid;
  int root, i, n, a[N], b[N];
  ……………………………………………………………………………………………………..
  root = 0;
  n = N/nproc;        /* number of elements PER PROCESS*/
  if (myid==root)  for (i=0; i<N; i++)  a[i]=i;
  err = MPI_Scatter (a, n, MPI_INT,               /* source buffer                      */
                     b, n, MPI_INT,               /*destination buffer                  */
                     root, MPI_COMM_WORLD); /* source process, communicator      */
  for (i=0; i<n; i++)  b[i] = 2*b[i];             /* parallel function computation      */
  err = MPI_Gather (b, n, MPI_INT,                /* source buffer                      */
                    a, n, MPI_INT,                /* destination buffer                 */
                    root, MPI_COMM_WORLD); /*destination process, communicator */
  ………………………………………………………………………………..
}
```

Scatter/Gather (2)

Scatter



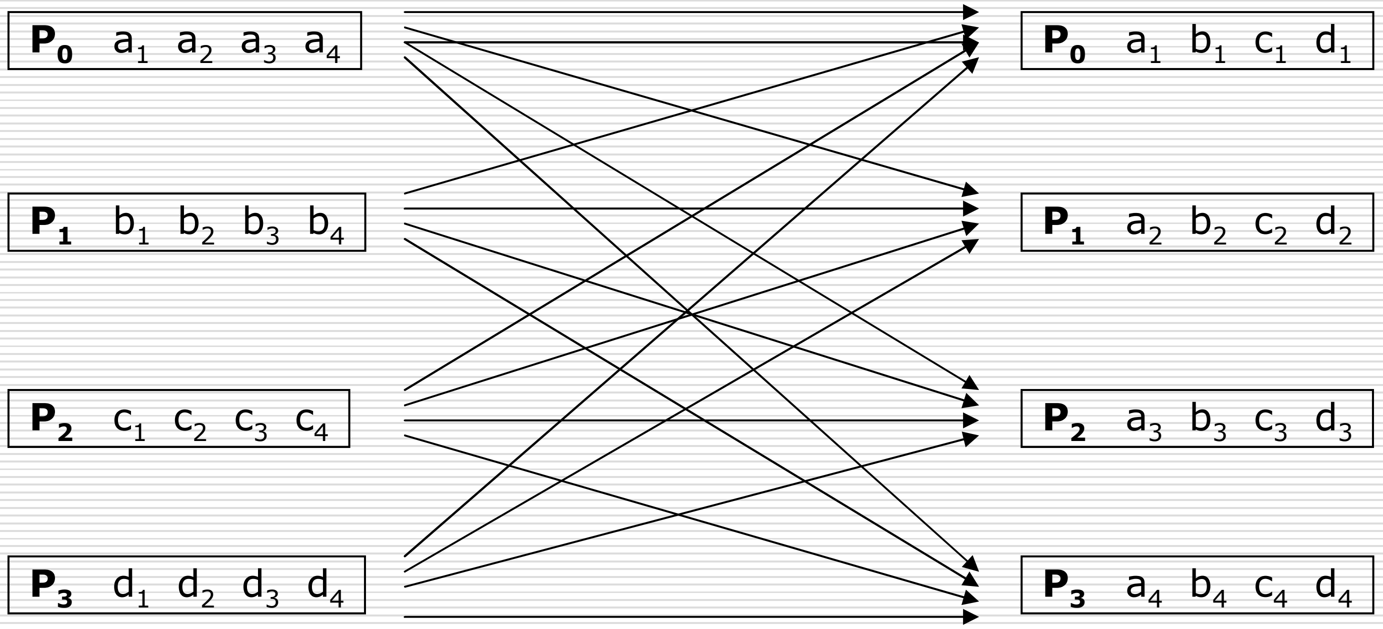
Gather



All-to-all: MPI_Alltoall (1)

```
//--------------------------------------------------------------------
// Exchange data among all processors
//--------------------------------------------------------------------
……………………………………………………………………………………………………
#define N 4
int main (int argc, char *argv[])
{int err, nproc, myid;
……………………………………………………………………………………………………………
int i, m;
int a[N];
…………………………………………………………………………………………………………
for (i=0, i<N, i++)  a[i]=N*myid+i;
printf("process %d has:\n", myid);
for (i=0, i<N, i++)  printf("%d ", a[i]);  printf("\n");
m = N/nproc;
MPI_Alltoall (a, m, MPI_INT,          // Sender
                  a, m, MPI_INT,          // Receiver
                  MPI_COMM_WORLD);
/* REMARK: count is the number of elements sent from one
    process to the other                                              */
printf("process %d has:\n", myid);
for (i=0, i<N, i++)  printf("%d ", a[i]);  printf("\n");
……………………………………………………………………………………………………………
}
```

All-to-all: MPI_Alltoall (2)



P_0 a_1 a_2 a_3 a_4
P_1 b_1 b_2 b_3 b_4
P_2 c_1 c_2 c_3 c_4
P_3 d_1 d_2 d_3 d_4
P_0 a_1 b_1 c_1 d_1
P_1 a_2 b_2 c_2 d_2
P_2 a_3 b_3 c_3 d_3
P_3 a_4 b_4 c_4 d_4

Reduction functions (1)

```
//---------------------------------------------------------------------
// Parallel sum, an instance of MPI reduction functions
//---------------------------------------------------------------------
………………………………………………………………………………………………………
#define N 4
int main (int argc, char *argv[])
{int err, nproc, myid;
……………………………………………………………………………………………………………….
int i, root, s, a[N];
……………………………………………………………………………………………………………..
for (i=0, i<N, i++)  a[i]=N;
printf("process %d has:\n", myid);
for (i=0, i<N, i++)  printf("%d ", a[i]);  printf("\n");
root = 0;
MPI_Reduce (a, &s, N, MPI_INT,    // S/D buff., count, type
                    MPI_SUM, root,        // Operation, destination
                    MPI_COMM_WORLD);
// REMARK: dropping the "root" argument, ALL processors get the result
if (myid==root) printf("The sum is %d", s);
………………………………………………………………………………………………………………….
}
```

Reduction functions (2)

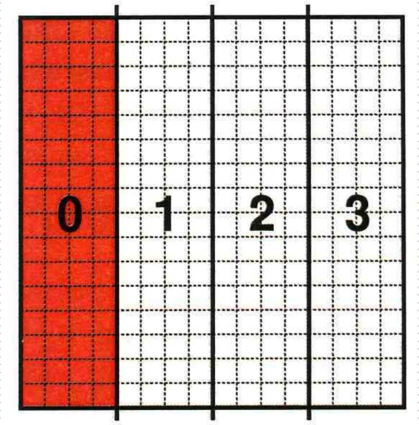
MPI_OP	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Location of maximum
MPI_MINLOC	Location of minimum
MPI_……	……………

Distributing arrays among processors

ISSUES:

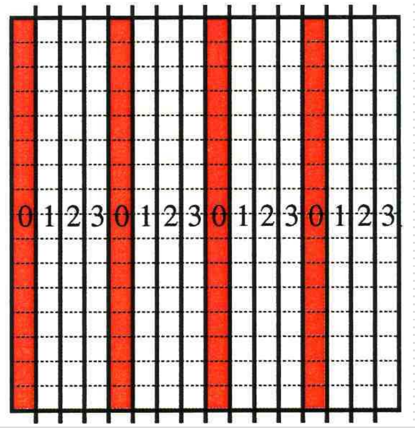
- Load balancing
- Communication optimization

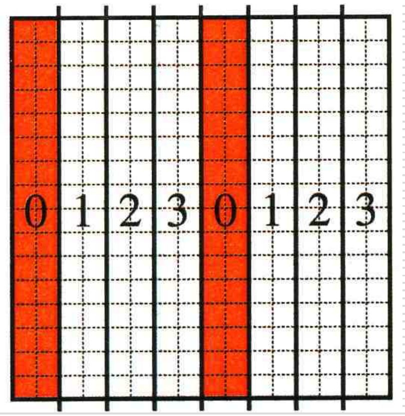
Array distribution 1



Colunm blocks

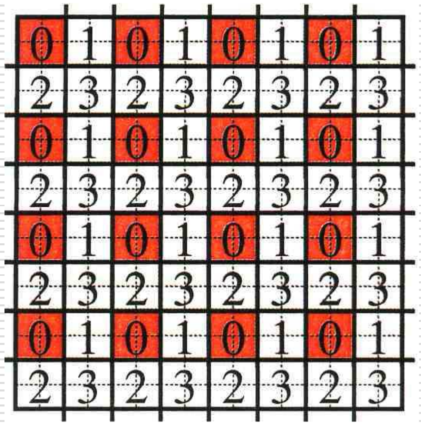
Column cycles





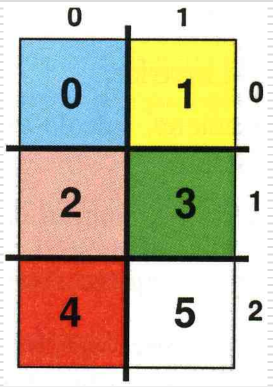
Colunm block cycles

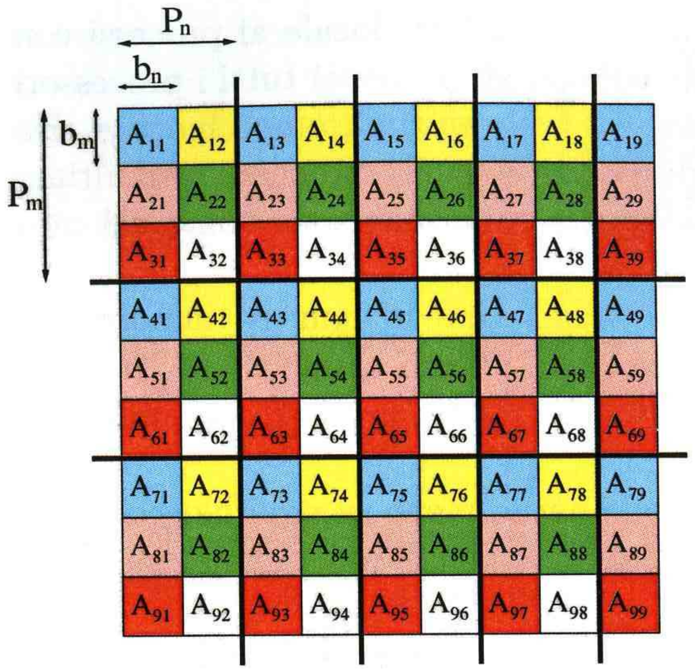
Row and colunm
block cycles



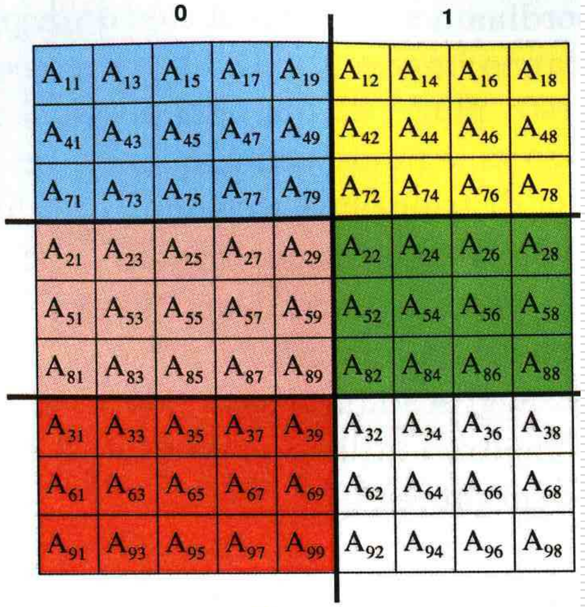
Array distribution 2

Grid of processors





P_n
b_n
b_m
P_m
A_{11} A_{12} A_{13} A_{14} A_{15} A_{16} A_{17} A_{18} A_{19}
A_{21} A_{22} A_{23} A_{24} A_{25} A_{26} A_{27} A_{28} A_{29}
A_{31} A_{32} A_{33} A_{34} A_{35} A_{36} A_{37} A_{38} A_{39}
A_{41} A_{42} A_{43} A_{44} A_{45} A_{46} A_{47} A_{48} A_{49}
A_{51} A_{52} A_{53} A_{54} A_{55} A_{56} A_{57} A_{58} A_{59}
A_{61} A_{62} A_{63} A_{64} A_{65} A_{66} A_{67} A_{68} A_{69}
A_{71} A_{72} A_{73} A_{74} A_{75} A_{76} A_{77} A_{78} A_{79}
A_{81} A_{82} A_{83} A_{84} A_{85} A_{86} A_{87} A_{88} A_{89}
A_{91} A_{92} A_{93} A_{94} A_{95} A_{96} A_{97} A_{98} A_{99}



0
1
A_{11} A_{13} A_{15} A_{17} A_{19} A_{12} A_{14} A_{16} A_{18}
A_{41} A_{43} A_{45} A_{47} A_{49} A_{42} A_{44} A_{46} A_{48}
A_{71} A_{73} A_{75} A_{77} A_{79} A_{72} A_{74} A_{76} A_{78}
A_{21} A_{23} A_{25} A_{27} A_{29} A_{22} A_{24} A_{26} A_{28}
A_{51} A_{53} A_{55} A_{57} A_{59} A_{52} A_{54} A_{56} A_{58}
A_{81} A_{83} A_{85} A_{87} A_{89} A_{82} A_{84} A_{86} A_{88}
A_{31} A_{33} A_{35} A_{37} A_{39} A_{32} A_{34} A_{36} A_{38}
A_{61} A_{63} A_{65} A_{67} A_{69} A_{62} A_{64} A_{66} A_{68}
A_{91} A_{93} A_{95} A_{97} A_{99} A_{92} A_{94} A_{96} A_{98}

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

References

- W.Gropp, E.Lusk, A.Skjellum. Using MPI. Portable Parallel Programming with the Massage-Passing Interface. MIT Press.