Parallel Scientific Computing

Course AMS301 — Fall 2023 — Lecture 2

Design of parallel algorithms
Advanced communications with MPI

Axel Modave

Parallel algorithms — Recap with one example

Sequential algorithm

for $n = 0 \cdots 99$ **do**
$\quad S = S + x_n \cdot y_n$
end

Parallel algorithms with 2 processes

Data: each process knows half of the entries of $\mathbf{x}$ et $\mathbf{y}$.

On each process $p = 0, 1$**:**

for $n = (50 \cdot p) \cdots (50 \cdot (p+1) - 1)$ **do**
$\quad S^{(p)} = S^{(p)} + x_n \cdot y_n$
end

Communication: process 1 sends $S^{(1)}$ to process 0.

On process 0**:** $S = S^{(0)} + S^{(1)}$

Result: process 0 knows S.

Process = set of instructions + memory space + communication resources

Strategy ***"divide and conquer"*** $\longrightarrow$ the global problem is divided into subdomains

The communication introduces a **synchronization**.

Parallel algorithms — General approach

The design of parallel algorithms is more complicated
than the design of sequential algorithms.

- To design a **sequential algorithm**, we have to
  - define a sequence of instructions to be process in a particular order by a sequential machine.

  ... that's it. :-)

- To design a **parallel algorithm**, we have to
  - distribute the operations/data between the nodes of the parallel machine, *(each node has its own sequence of instructions and its own data)*

    $\longrightarrow$ ***Define a partitioning / Mapping***
  - specify the data transfers between the nodes, and the order of operations.

    $\longrightarrow$ ***Define a communication pattern***

Design of parallel algorithms — Partitioning [1/4]

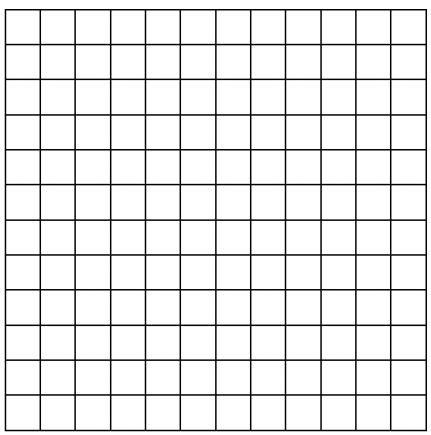
The operations and the associated data are partitioned over the processes, which can be executed concurrently *(i.e. independently)*.
The goal of this phase is to exhibit the parallelism.

Domain decomposition

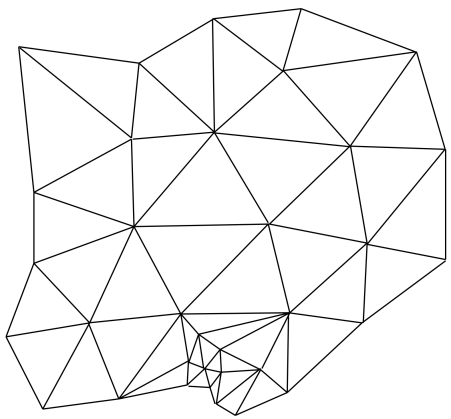
- The data are partitioned over subdomains, then we deal with the operations that uses these data.
- The analysis is performed in this order: data, operations, communications.

Examples:

Structured grid



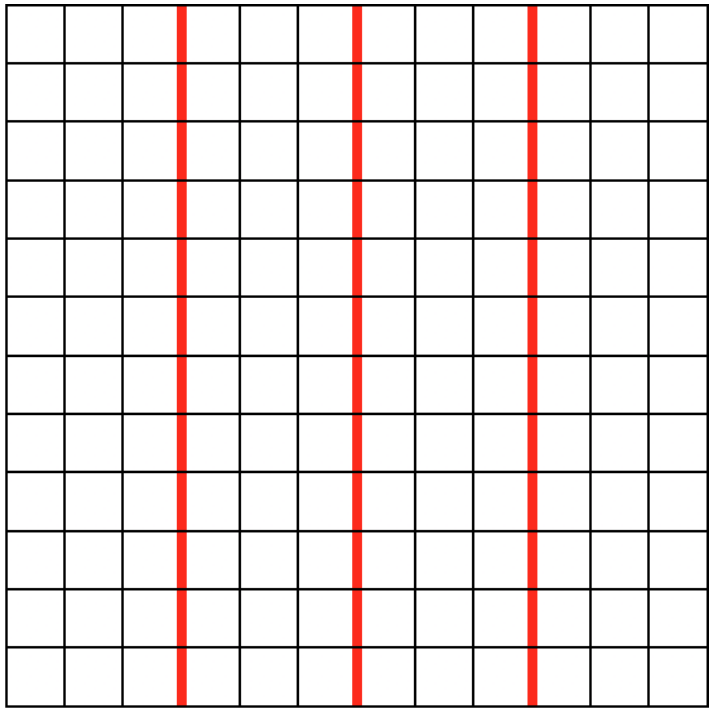
Unstructured mesh



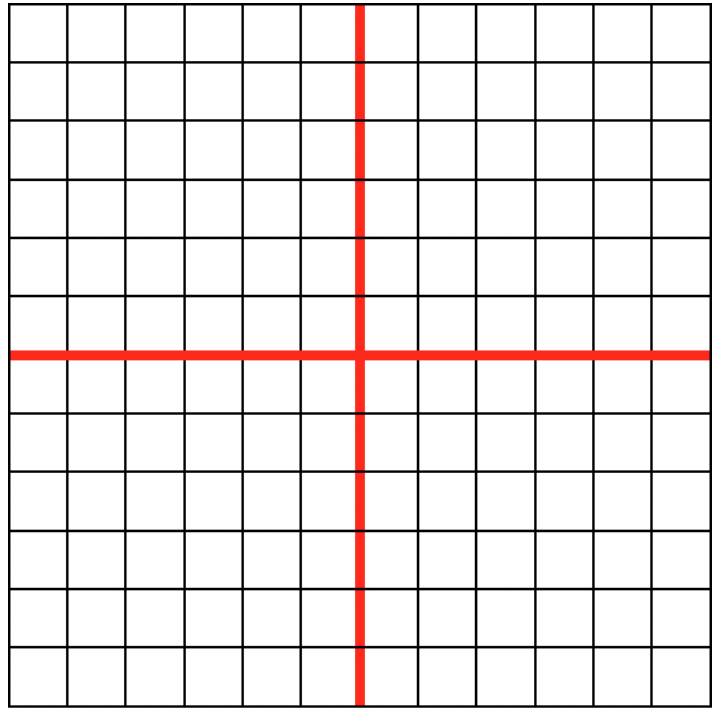
Cloud of points

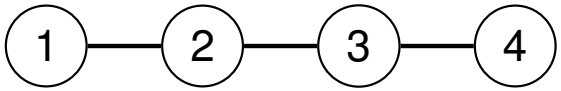
Design of parallel algorithms — Partitioning [2/4]

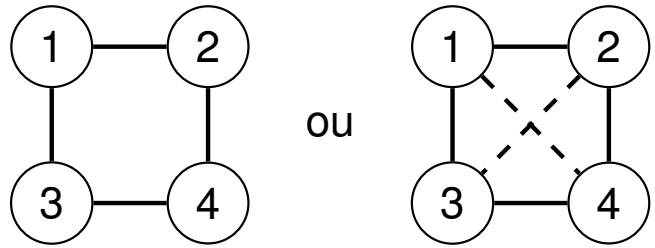
1D partition



2D partition

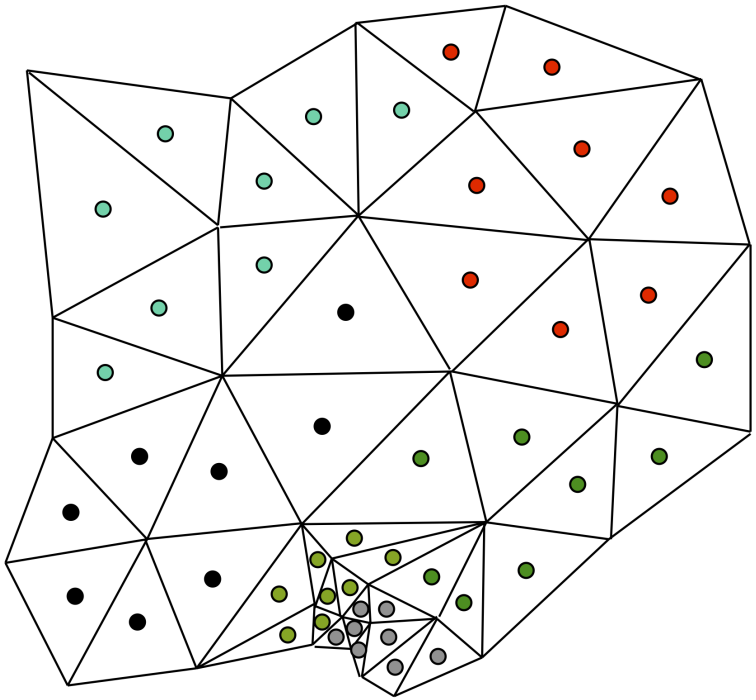




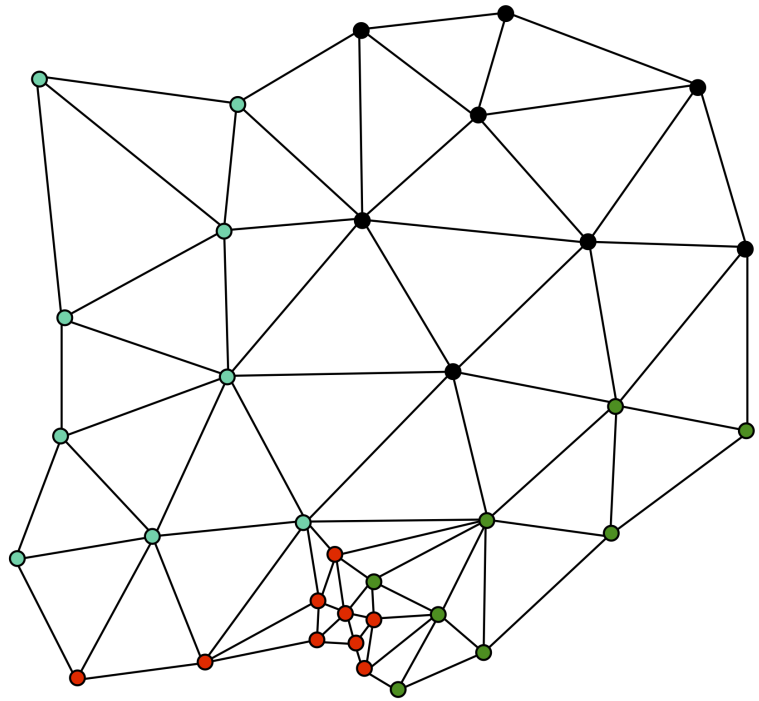


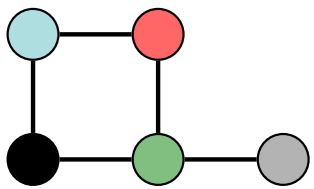
Design of parallel algorithms — Partitioning [3/4]

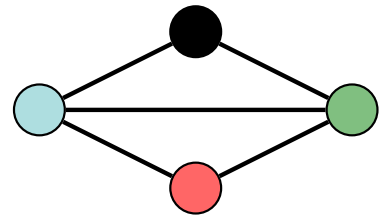
Partition by mesh cells



Partition by nodes







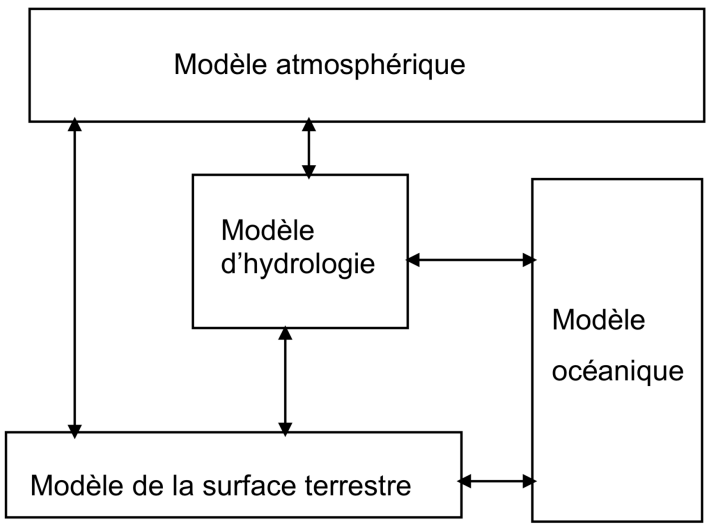
The theory of graphs can be used to define and describe the mesh partitioning process in a clean way.

Functional decomposition

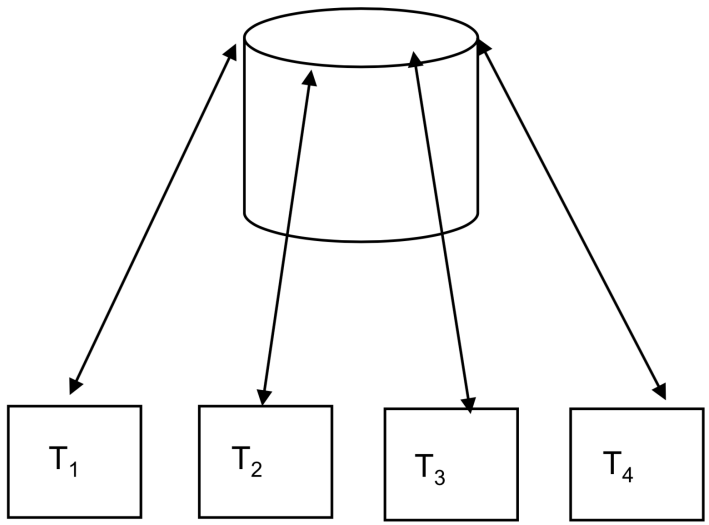
- The operations are partitioned into elementary blocs, then we deal with the data associated to each elementary bloc.
- The analysis is performed in this order: operations, data, communications.

Examples:

Coupled codes



Task server



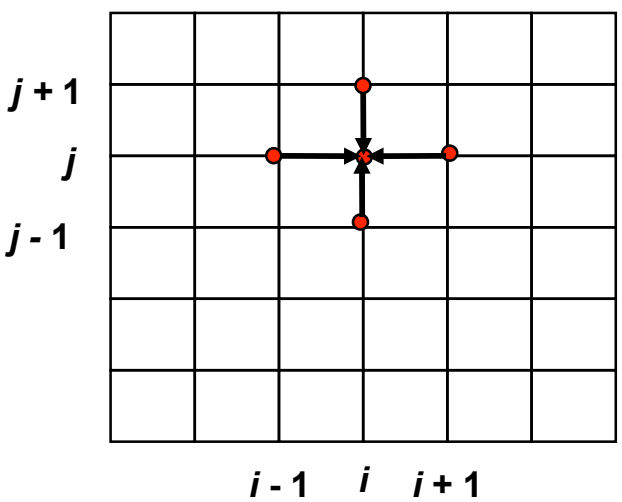
The interactions between the processes are identified
and a communication pattern is chosen.

Local interaction: One task communicates with another one *(or few others)*.

Example: 5-points finite difference scheme

Discretization of the equation $\partial_t u - \Delta u = 0$ with the 5-point scheme in space:

$$\frac{u_{i,j}^{\ell+1} - u_{i,j}^{\ell}}{h_t} - \frac{4u_{i,j}^{\ell} - u_{i-1,j}^{\ell} - u_{i+1,j}^{\ell} - u_{i,j-1}^{\ell} - u_{i,j+1}^{\ell}}{h_x^2} = 0$$



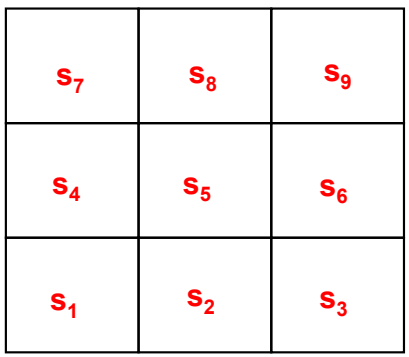
for $\ell = 0 \ldots \ell_{final}$ **do**
  for $i, j = \ldots$ **do**
    Send $u_{i,j}^{\ell}$ to each neighbor
    Receive $u_{i-1,j}^{\ell}, u_{i+1,j}^{\ell}, u_{i,j-1}^{\ell}, u_{i,j+1}^{\ell}$ from neighbors
    Compute $u_{i,j}^{\ell+1}$
  end
end

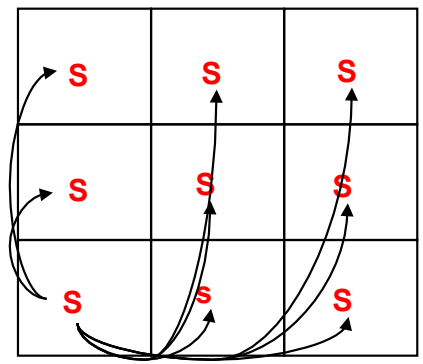
Design of parallel algorithms — Communications [2/5]

Global interaction: one *(or more)* task communicates with all the others.

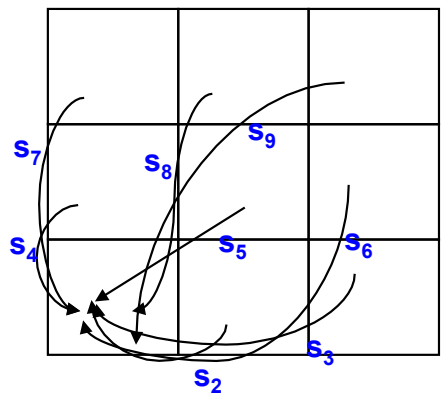
Example: parallel scalar product $S = \mathbf{x} \cdot \mathbf{y}$

1. The vectors $\mathbf{x}$ and $\mathbf{y}$ is partitioned over the tasks.
2. The partial scalar products S_i are computed by the tasks.
3. They are collected by one task, and the sum S is computed.
4. The final value S is sent to all the tasks.





Diffusion



Reduction

The two last steps are standard global operations:

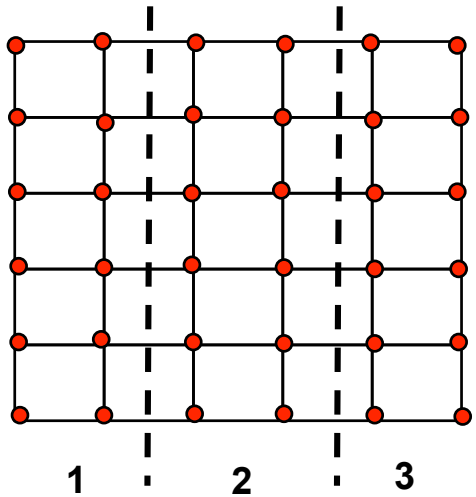
- The operation **"diffusion"** distributes one value to a set of tasks.
- The operation **"reduction"** collects values that are distributed over a set of tasks, and a basis global operations is performed *(i.e. addition or maximum)*.

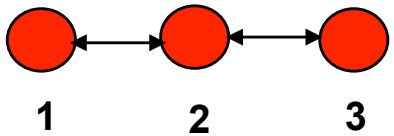
Design of parallel algorithms — Communications [3/5]

Structured communications: The communication pattern is regular and does not evolve over time.

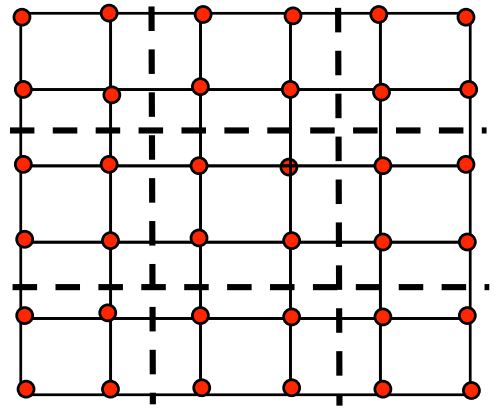
Example: 5-points finite difference scheme

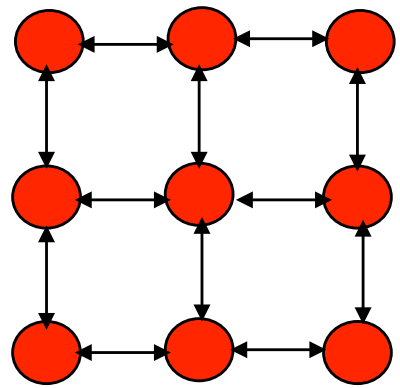
1D partition





2D partition

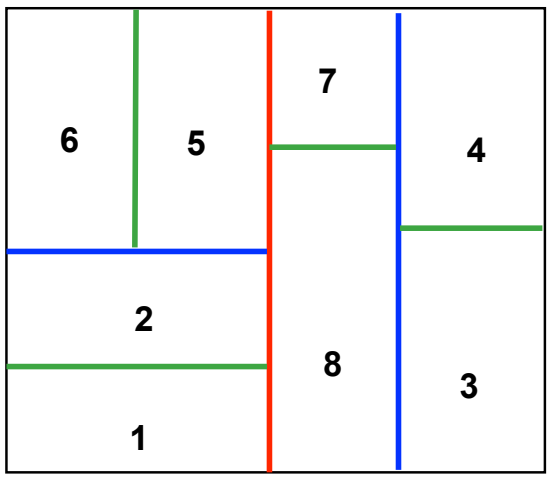


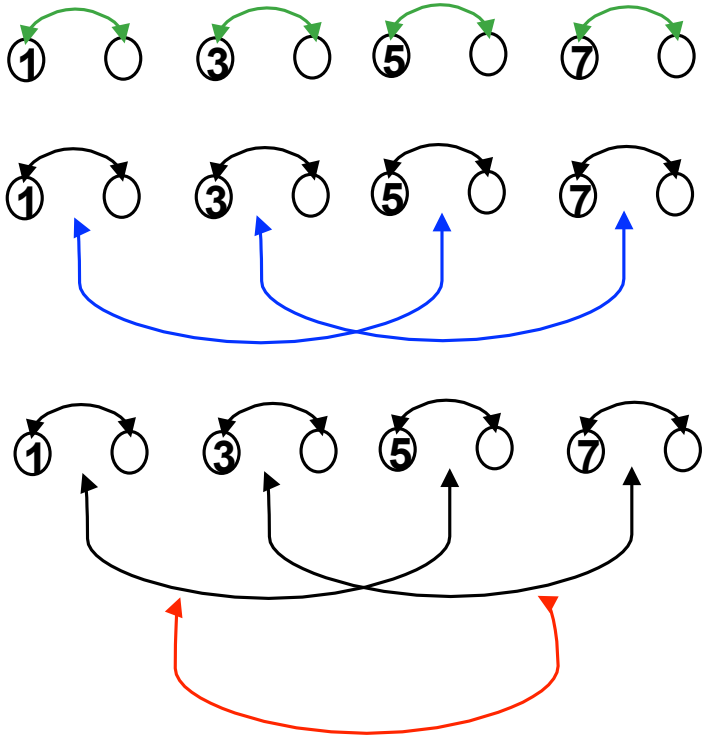


Structured communications: The communication pattern is regular and does not evolve over time.

Example: 5-points finite difference scheme

Recursive partition

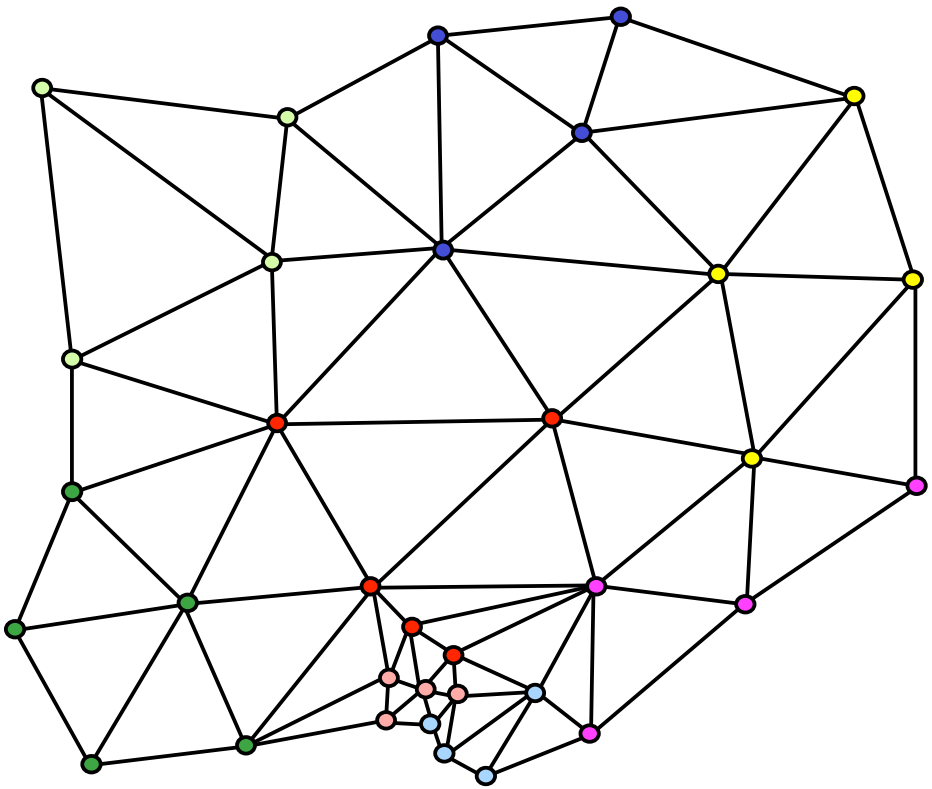




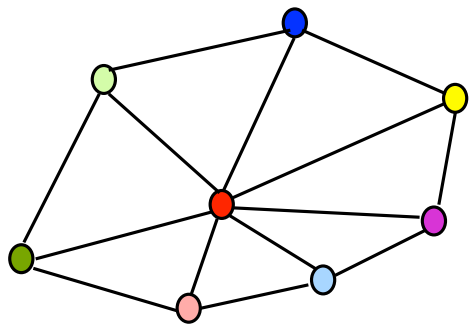
Hierarchical scheme

Unstructured communications: The communication pattern is non-regular or it evolves over time.

Example: finite element scheme

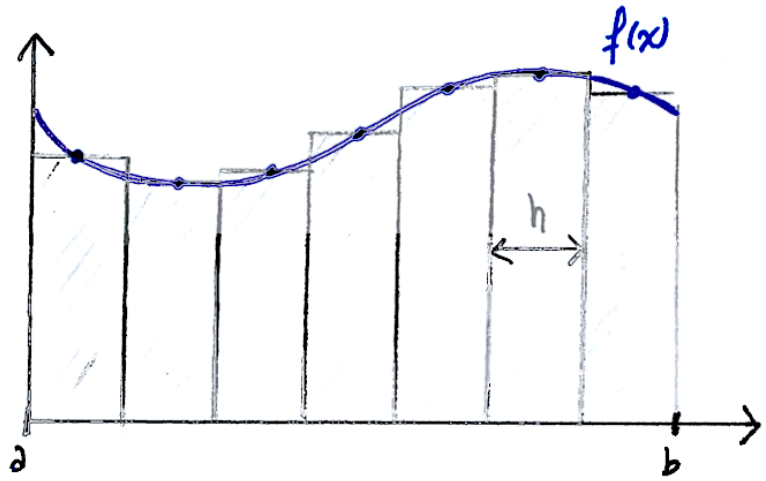


Mesh partition *by nodes*



Graph of tasks

Basic algorithm — Numerical integration (1D) [1/3]

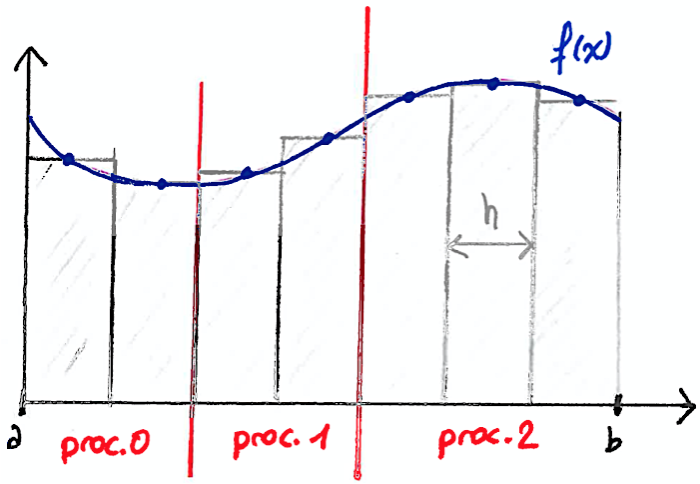


Numerical integration with the midpoint method:

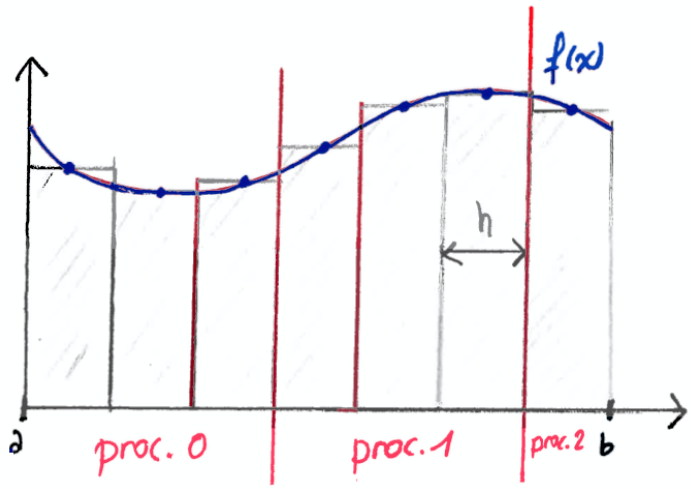
$$\int_a^b f(x)\,dx \;\approx\; \sum_{n=0}^{N-1} h\; f(a + h/2 + nh), \quad \text{with } h = \frac{b-a}{N}$$

Sequential algorithm
$S \leftarrow 0$ **for** $n = 0, \ldots, (N-1)$ **do** $\quad S \leftarrow S + h\; f(a + h/2 + nh)$ **end**

Basic algorithm — Numerical integration (1D) [2/3]



(ou)



Numerical integration with N points and P processes:

$$\int_a^b f(x)\,dx \approx \sum_{p=0}^{P-1} \sum_{n=n_{start}^{(p)}}^{n_{end}^{(p)}} h\, f(a + h/2 + nh), \quad \text{with } h = \frac{b-a}{N}$$

For the second version:

$$n_{start}^{(p)} = p(\lfloor N/P \rfloor + 1)$$
$$n_{end}^{(p)} = \min\left(n_{start}^{(p+1)} - 1, N-1\right) = \min\left((p+1)(\lfloor N/P \rfloor + 1) - 1, N-1\right)$$

Basic algorithm — Numerical integration (1D) [3/3]

Numerical integration with N points and P processes:

$$\int_a^b f(x)\,dx \approx \sum_{p=0}^{P-1} \sum_{n=n_{start}^{(p)}}^{n_{end}^{(p)}} h\ f(a+h/2+nh), \quad \text{with } h = \frac{b-a}{N}$$

Parallel algorithms with P processes

On each process $p = 0, \ldots, (P-1)$:

$S^{(p)} \leftarrow 0$

$n_{start}^{(p)} \leftarrow p(\lfloor N/P \rfloor + 1)$

$n_{end}^{(p)} \leftarrow \min\left((p+1)(\lfloor N/P \rfloor + 1) - 1, N-1\right)$

for $n = n_{start}^{(p)}, \ldots, n_{end}^{(p)}$ **do**

$\quad S^{(p)} \leftarrow S^{(p)} + h\ f(a+h/2+nh)$

end

Global operation (Réduction): parallel addition of the $S^{(p)}$'s.

Result: all processes know S.

Basic algorithm — Finite difference (1D) [1/3]

Considered problem: $\boxed{\partial_t u - \kappa \partial_{xx} u = 0}$ for $x \in]a, b[$ and $t \in]0, t_{fin}]$,

with the initial condition $u = u_{init}(x)$, the boundary condition $u = 0$ and $\kappa > 0$.

Discretization on a regular grid: $u_n^\ell \approx u(t_\ell, x_n)$ with

$$x_n = a + nh_x \quad n = 0, 1, \ldots, N+1 \quad h_x = (b-a)/(N+1)$$
$$t_\ell = \ell h_t \quad \ell = 0, 1, \ldots, L \quad h_t = t_{fin}/L$$

Finite difference scheme (with $2\kappa h_t / h_x^2 < 1$):

$$\frac{u_n^{l+1} - u_n^\ell}{h_t} - \kappa \frac{u_{n-1}^\ell - 2u_n^\ell + u_{n+1}^\ell}{h_x^2} = 0 \quad (n = 1, \ldots, N, \ell = 0, 1, \ldots, L)$$

Sequential algorithm

Data: initial solution $\left\{u_n^0\right\}_{n=1,\ldots,N}$

for $\ell = 0, \ldots, L-1$ **do**

  for $n = 1, \ldots, N$ **do**

    $u_n^{\ell+1} \leftarrow u_n^\ell + \frac{\kappa h_t}{h_x^2}\left[u_{n-1}^\ell - 2u_n^\ell + u_{n+1}^\ell\right]$

  end

end

Basic algorithm — Finite difference (1D) [2/3]

Sequential algorithm

Data: initial solution $\{u_n^0\}_{n=1,\dots,N}$

for $\ell = 0, \dots, L-1$ **do**

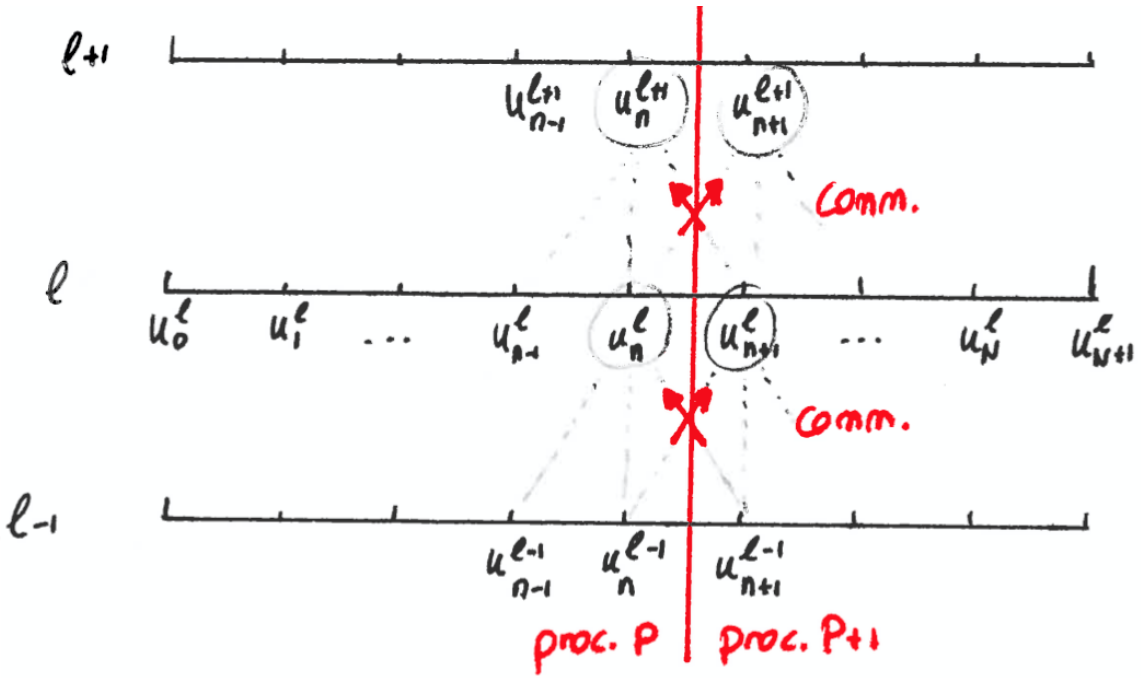
  for $n = 1, \dots, N$ **do**

    $$u_n^{\ell+1} \leftarrow u_n^\ell + \frac{\kappa h_t}{h_x^2}\left[u_{n-1}^\ell - 2u_n^\ell + u_{n+1}^\ell\right]$$

  end

end

Analysis of dependencies



Basic algorithm — Finite difference (1D) [3/3]

Parallel algorithms with P processes

Data: each process knows one part of the init. sol. $\left\{u_n^0\right\}_{n=1,\dots,N}$

On each process $p = 0, \dots, (P-1)$:

$n_{start} \leftarrow \cdots$

$n_{end} \leftarrow \cdots$

for $\ell = 0, 1, \dots$ **do**

  for $n = n_{start}, \dots, n_{end}$ **do**

    $$u_n^{\ell+1} \leftarrow u_n^\ell + \frac{\kappa h_t}{h_x^2}\left[u_{n-1}^\ell - 2u_n^\ell + u_{n+1}^\ell\right]$$

  end

  if $p > 0$ **then**

    Comm. : process p sends $u_{n_{start}}^{\ell+1}$ to process $p-1$

    Comm. : process p receives $u_{n_{start}-1}^{\ell+1}$ from process $p-1$

  end

  if $p < (P-1)$ **then**

    Comm. : process p sends $u_{n_{end}}^{\ell+1}$ to process $p+1$

    Comm. : process p receives $u_{n_{end}+1}^{\ell+1}$ from process $p+1$

  end

end

Result: each process knows one part of the final solution.

Design of parallel algorithms

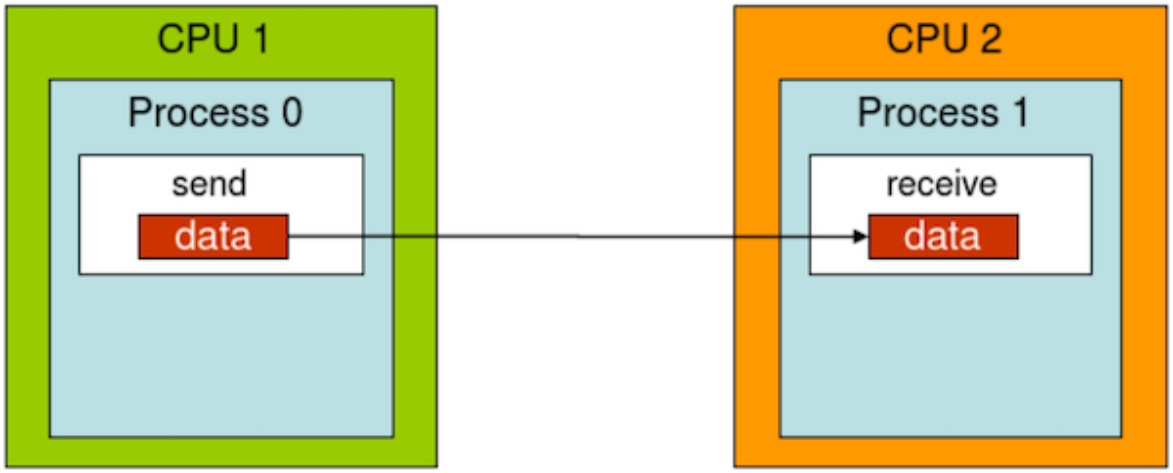
Partitioning and communications

Two basic algorithms

Advanced communications with MPI

MPI — Point-to-point communications

The commands `MPI_Send` and `MPI_Recv` transfer data from one process to another one.



Example:

```
1 if(myRank == 0)
2   MPI_Send(&arraySend[0], 128, MPI_INT, 1, 66, MPI_COMM_WORLD);
3 if(myRank == 1)
4   MPI_Recv(&arrayRecv[0], 128, MPI_INT, 0, 66, MPI_COMM_WORLD, MPI_STATUS_IGNORE
      );
```

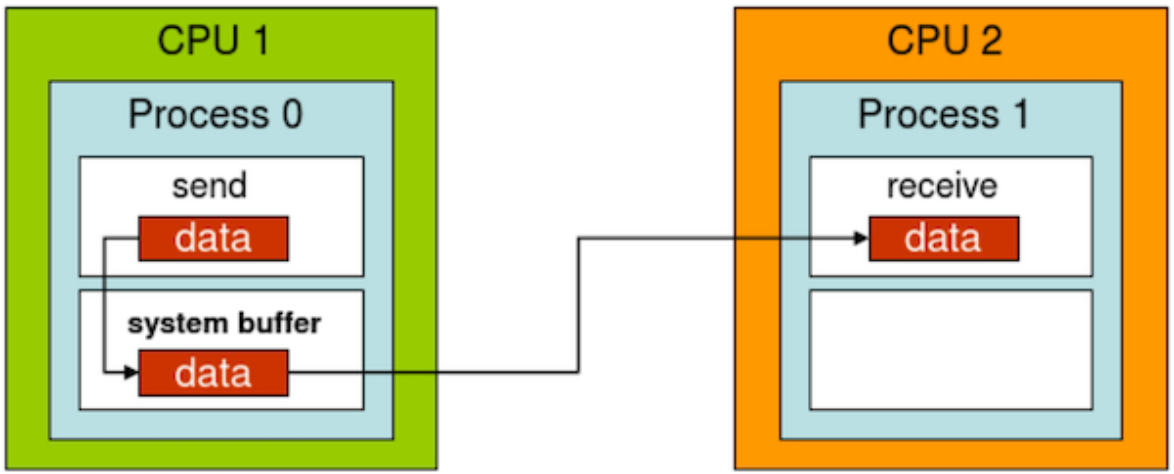
In this example, an array with 128 integers is transferred from process `0` to process `1`. The *tag* associated to this message is `66`.

These commands are ***a priori* blocking**: both processes (`0` and `1`) are blocked while the message is not completely received by process `1`. **WARNING !**

MPI — Point-to-point communications: *The "buffered" protocol*

If the message is very short, the communication is buffered.

```
1 if(myRank == 0){
2   MPI_Send(&arraySend[0], 4, MPI_INT, 1, 66, MPI_COMM_WORLD);
3   ...     // Done even if data not yet received by processus 1
4   ...     // "arraySend" can be modified safely
5 }
6 if(myRank == 1){
7   MPI_Recv(&arrayRecv[0], 4, MPI_INT, 0, 66, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
8   ...     // Done only if data received by processus 1
9 }
```



Process 0: The operations after `MPI_Send` are processed, even if the message is not transferred yet. The values are stored in a buffer.

$\Rightarrow$ The values in `arraySend` can be modified after `MPI_Send` without any problem.

Process 1: This process is blocked while the message is not completely transferred.

MPI — Point-to-point communications: Nonblocking mode

The commands `MPI_Isend` et `MPI_Irecv` transfer data from one process to another one in the nonblocking mode.

```
1  MPI_Request req;
2  if(myRank == 0){
3    MPI_Isend(&arraySend[0], 128, MPI_INT, 1, 66, MPI_COMM_WORLD, &req
       );
4    ...     // Done even if data not yet received by processus 1
5    ...     // "arraySend" cannot be modified safely!
6    MPI_Wait(&req, MPI_STATUS_IGNORE);
7    ...     // Done only if data received by processus 1
8  }
9  if(myRank == 1){
10   MPI_Irecv(&arrayRecv[0], 128, MPI_INT, 0, 66, MPI_COMM_WORLD, &req
       );
11   ...     // Done even if data not yet received by processus 1
12   ...     // "arrayRecv" should not be used here!
13   MPI_Wait(&req, MPI_STATUS_IGNORE);
14   ...     // Done only if data received by processus 1
15 }
```

These commands are **nonblocking**: the operations after each command are processed, even if the message is not completely transferred yet.

⇒ Do not modify `arraySend`, do not use `arrayRecv` if the message is not transferred yet!

The command `MPI_Wait` blocks the operations while the message is not received.

The command `MPI_Barrier` synchronizes the processes.
When a process reaches the command, it is blocked until all processes have reached it.

Description:

```
int MPI_Barrier(MPI_Comm comm)
```

Communicator

Example:

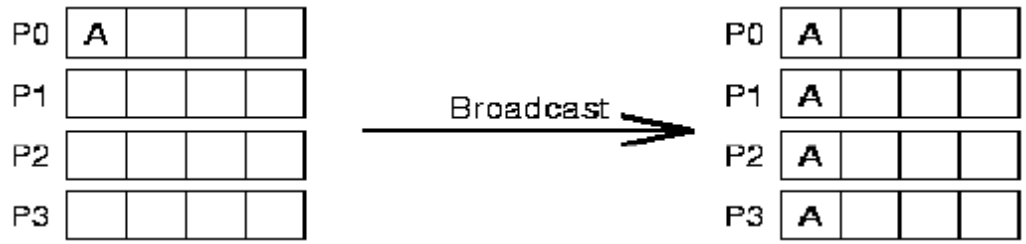
```
1 MPI_Barrier(MPI_COMM_WORLD);
2 double time1 = MPI_Wtime();
3
4 // Lots of operations for all the processus
5
6 MPI_Barrier(MPI_COMM_WORLD);
7 double time2 = MPI_Wtime();
8
9 if(myRank == 0) printf("Duration: %f\n", time2-time1);
```

In this example, the command `MPI_Wtime()` is used to measure the time before and after a sequence of operations performed in parallel. The command `MPI_Barrier` is used to synchronize all processes before measuring the time.

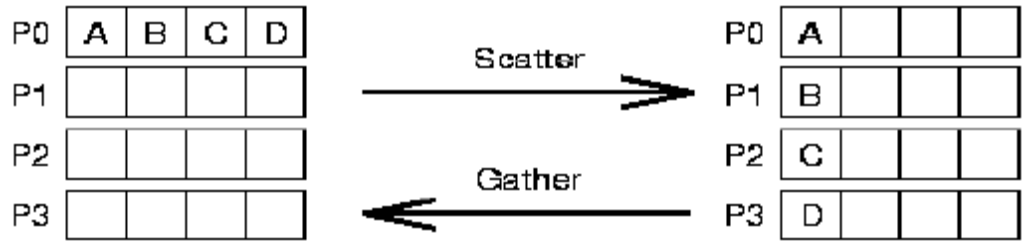
MPI — Collective communications: Data transfer [1/2]

The MPI commands hereafter transfer data between several MPI processes.
These commands synchronize the processes.

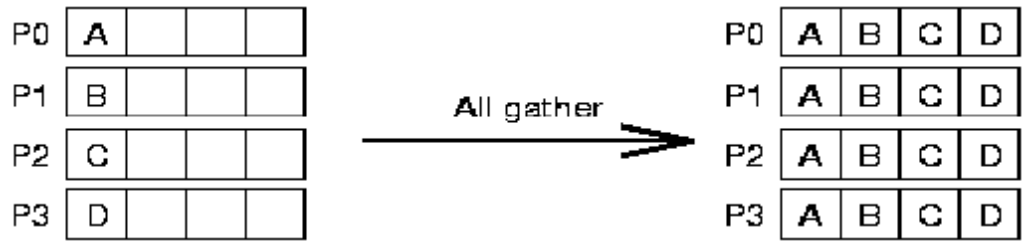
`MPI_Bcast`



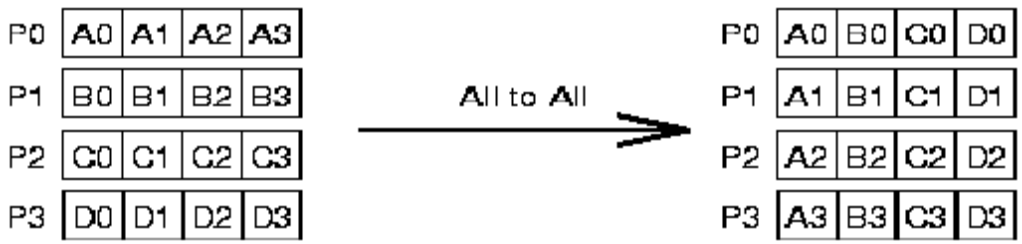
`MPI_Scatter`
`MPI_Gather`



`MPI_Allgather`

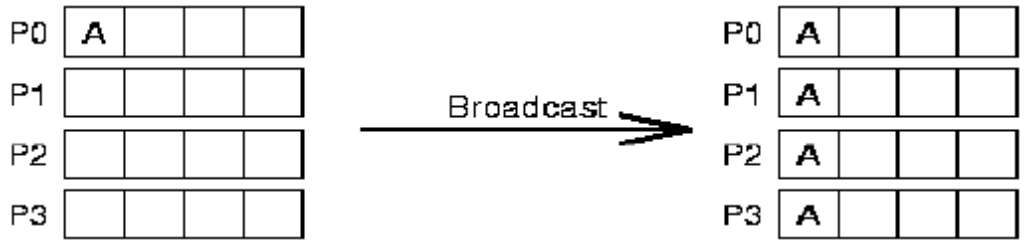


`MPI_Alltoall`



MPI — Collective communications: Data transfer [2/2]

The command `MPI_Bcast` broadcasts a message from one process to all other processes.



Description:

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

Starting address of buffer
Number of entries in buffer
Data type of buffer
Rank of broadcast root
Communicator

Example:

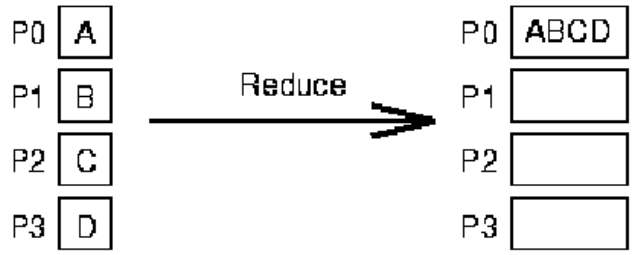
```
1 int param;
2 if(myRank == 0){
3   printf("Enter parameter: ");
4   scanf("%i", param);
5 }
6 MPI_Bcast(&param, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

In this example, an information is requested to the user by process 0, and it is send to the other processes. These processes wait until the info. is send. $\Rightarrow$ **Synchronization**

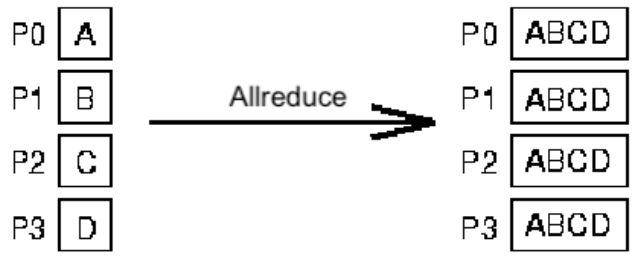
MPI — Collective communications: Global operations [1/2]

With the MPI commands hereafter, data are transferred and basic operation are performed on these data. The MPI processes are synchronized with these commands.

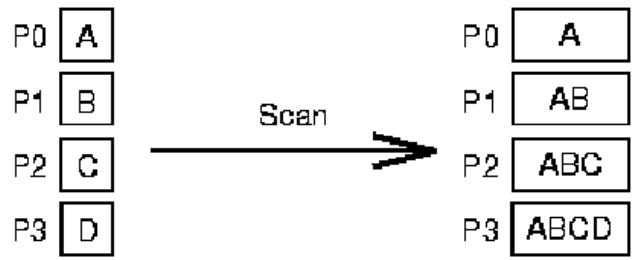
`MPI_Reduce`



`MPI_Allreduce`



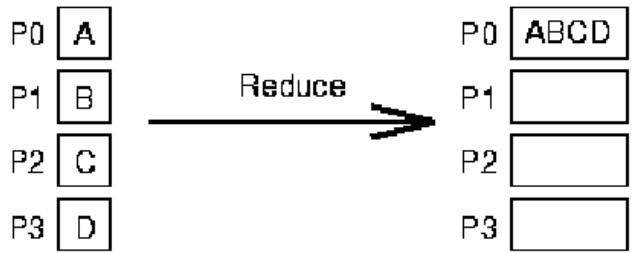
`MPI_Scan`



In the column on the right of these illustrations, one operation has been performed on A, AB, ABC and ABCD.

MPI — Collective communications: Global operations [2/2]

The command `MPI_Reduce` reduces values on all processes to a single value.



Description:

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

Address of send buffer
Address of receive buffer
Number of elements in send buffer
Data type of elements of send buffer
Reduce operation
Rank of root process
Communicator

Several kinds of operations : `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, `MPI_LAND`, ...

Example:

```
1 int myVal = func(myRank);     // "myVal" depends on the processus
2 int maxVal;
3 MPI_Reduce(&myVal, &maxVal, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
```

At the end, process 0 *(only this process)* knows the maximum `myVal` *(stored in* `maxVal`*)*.

Summary

- **Parallel algorithms**
  - Domain/Functional decomposition
  - Local/Global communications
  - Structured/Unstructured scheme
- **Two basic algorithms**
  - Numerical integration (1D)
  - Finite difference (1D)
- **MPI Commands**
  - `MPI_Init`, `MPI_Finalize`
  - `MPI_Comm_size`, `MPI_Comm_rank`
  - `MPI_Send`, `MPI_Recv`
  - `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`
  - `MPI_Barrier`, `MPI_Wtime`
  - `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, ...
  - `MPI_Reduce`, `MPI_Allreduce`, ...