The Message Passing Interface (MPI)

TMA4280—Introduction to Supercomputing

NTNU, IMF

February 9. 2018

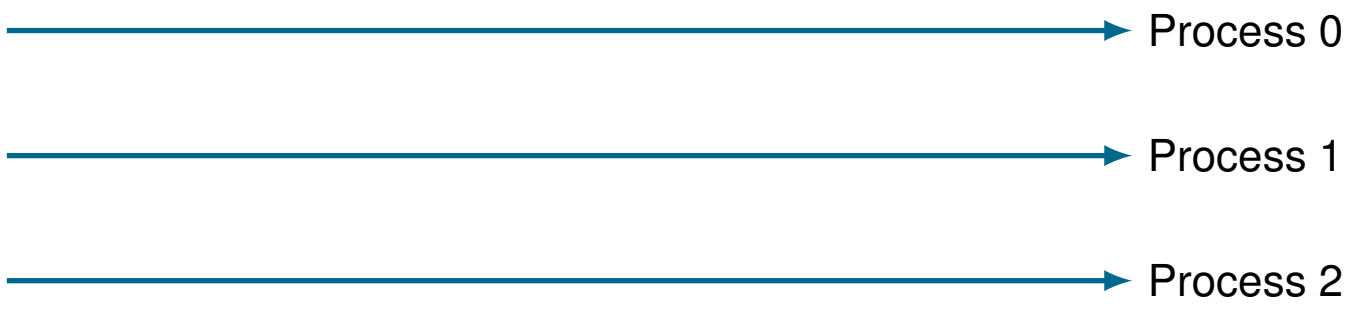
Recap: Parallelism with MPI

An MPI execution is started on a set of processes $\mathcal{P}$ with:

`mpirun -n` N_P `./exe`

with $N_P = card(\mathcal{P})$ the number of processes.

A program is executed in parallel on each process:



where each process in $\mathcal{P} = \{P0, P1, P2\}$ has a access to data in its memory space: remote data exchanged through interconnect.

An ordered set of processes defines a **Group** (`MPI_Group`): the inital group (`WORLD`) consists of **all** processes.

Recap: Parallelism with MPI

An MPI execution is enclosed between calls of function to initialize and finalize the subsystem:

`MPI_Init`, ... { Program body } ..., `MPI_Finalize`.

Processes in a **Group** can communicate through a **Communicator** (`MPI_Comm`) with attributes:

- rank: `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
- size: `MPI_Comm_size(MPI_COMM_WORLD, &size);`

The default communicator for **all** processes is `MPI_COMM_WORLD`.

New **Groups** can be created on subsets of $\mathcal{P}$ to restrict the communication between selected processes.

Example: Monte-Carlo simulation, structure implementation of adjacent exchange.

Recap: Parallelism with MPI

Process communicate using the **Message Passing** paradigm:

1. Envelope: how to process the message,
2. Body: data to exchange.

MPI routines are categorized depending on their relation:

Point-to-Point	one-to-one
Collective	one-to-all all-to-one all-to-all

where **all** is defined by the **Communicator** used.

Point-to-Point operations are the base of **Message Passing**: `MPI_Send`, `MPI_Recv` to exchange data.

Collective operations are implemented in terms of Point-to-Point operations but may be optimized.

Message buffers

`MPI_Send(`*buffer*, *count*, *datatype*, *dest*, *tag*, *comm*`);`

data *envelope*

`buffer`	memory location defining the begining of the buffer
`count`	max length in bytes of the send buffer (**bigger** than actual)
`dest`	receiving process rank
`tag`	arbitrary identifier
`comm`	communicator

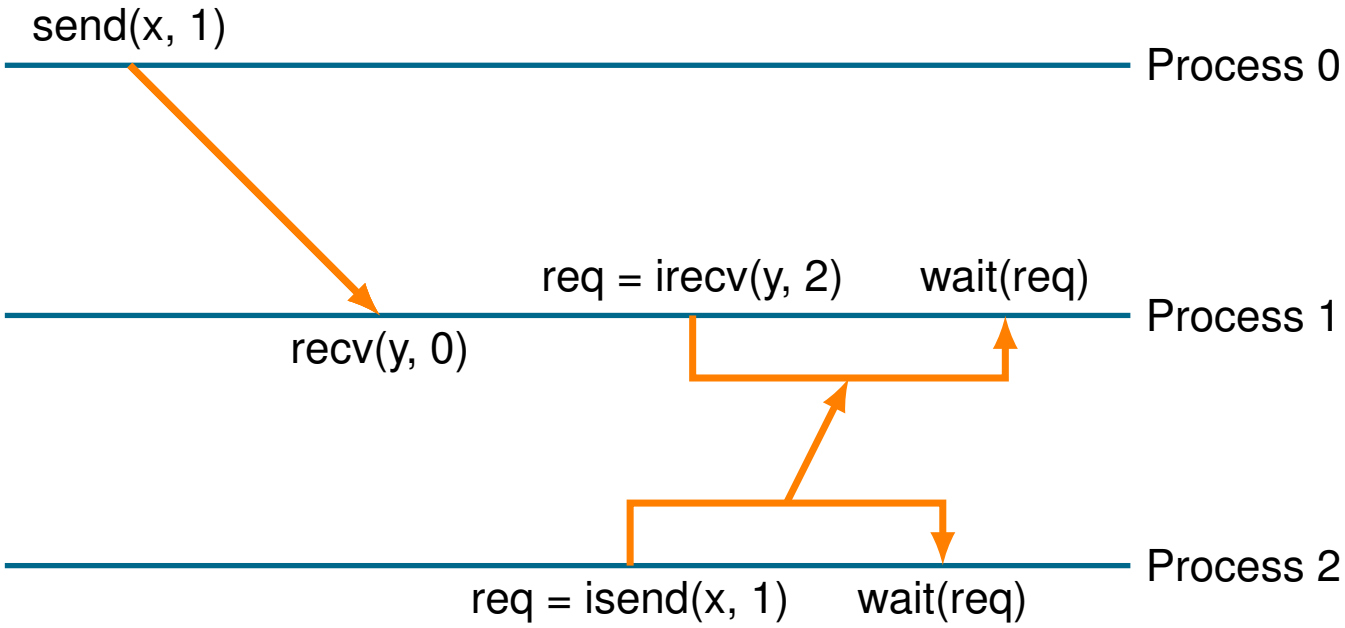
`MPI_Recv(`*buffer*, *count*, *datatype*, *source*, *tag*, *comm*, &*status*`);`

data *envelope*

`buffer`	memory location defining the begining of the buffer
`count`	maxlength in bytes of received data
`datatype`	MPI data type
`source`	sending process rank
`tag`	arbitrary identifier
`comm`	communicator
`status`	metadata: actual length

Recap: Parallelism with MPI

Two types of MPI commmunication: block and non-blocking.



1. blocking: the function returns once the buffer is processed (what "processed" (*)
2. non-blocking: the function returns immediately

MPI implementations use an internal buffer:

— data may be buffered internally on both sides,

— sending process can modify the send buffer once it is copied to the internal buffe regardless of receiver status,

Recap: Parallelism with MPI

(∗) What “processed” means depends on the communication mode:

`MPI_Send`, `MPI_SSend`, `MPI_RSend`, `MPI_BSend`

.

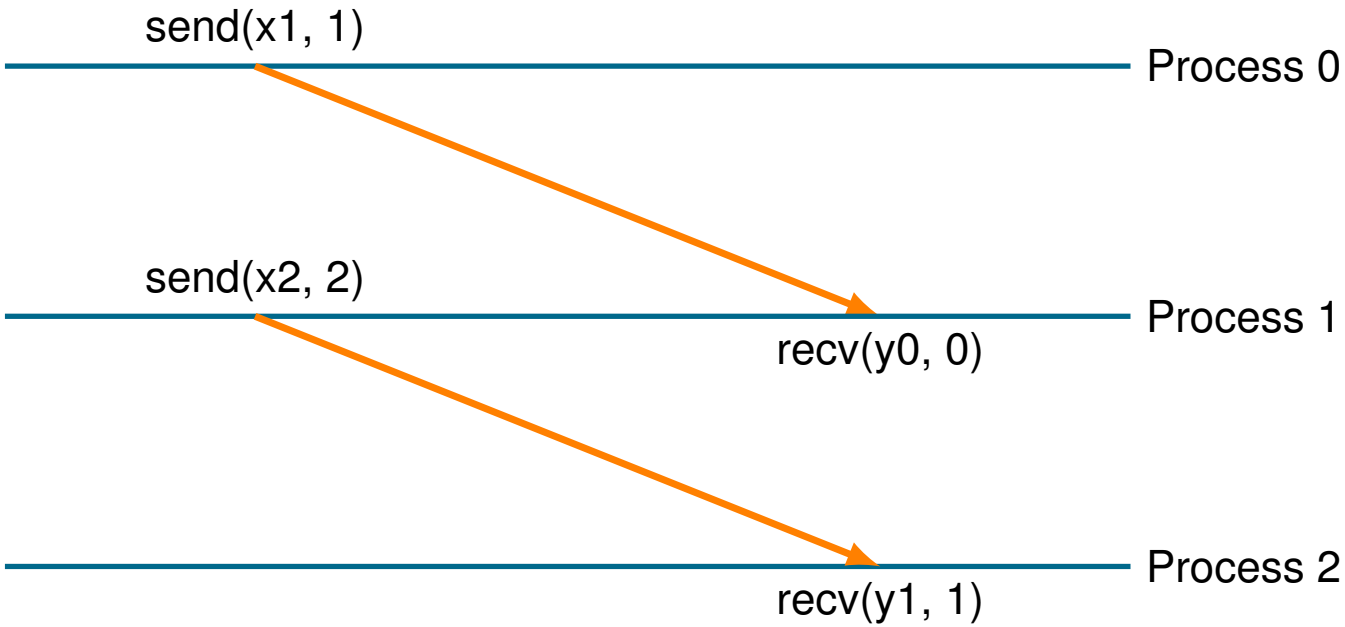
MPI may provide several versions of a routine for different communication modes

- Synchronous: send blocks until handshake is done and matching receive has started.
- Ready: send blocks until matching receive has started (no handshake).
- Buffered: send returns as soon as data is buffered internally.
- Standard: Synchronous or Buffered depending on message size and resources.

Important: do not assume anything about the implementation, different handling of buffers can hide bugs!

Parallelism

Pairwise send–receive:



A safe choice for Point-to-Point communication and optimized by vendors for their hardware.

Collective operations

In general,

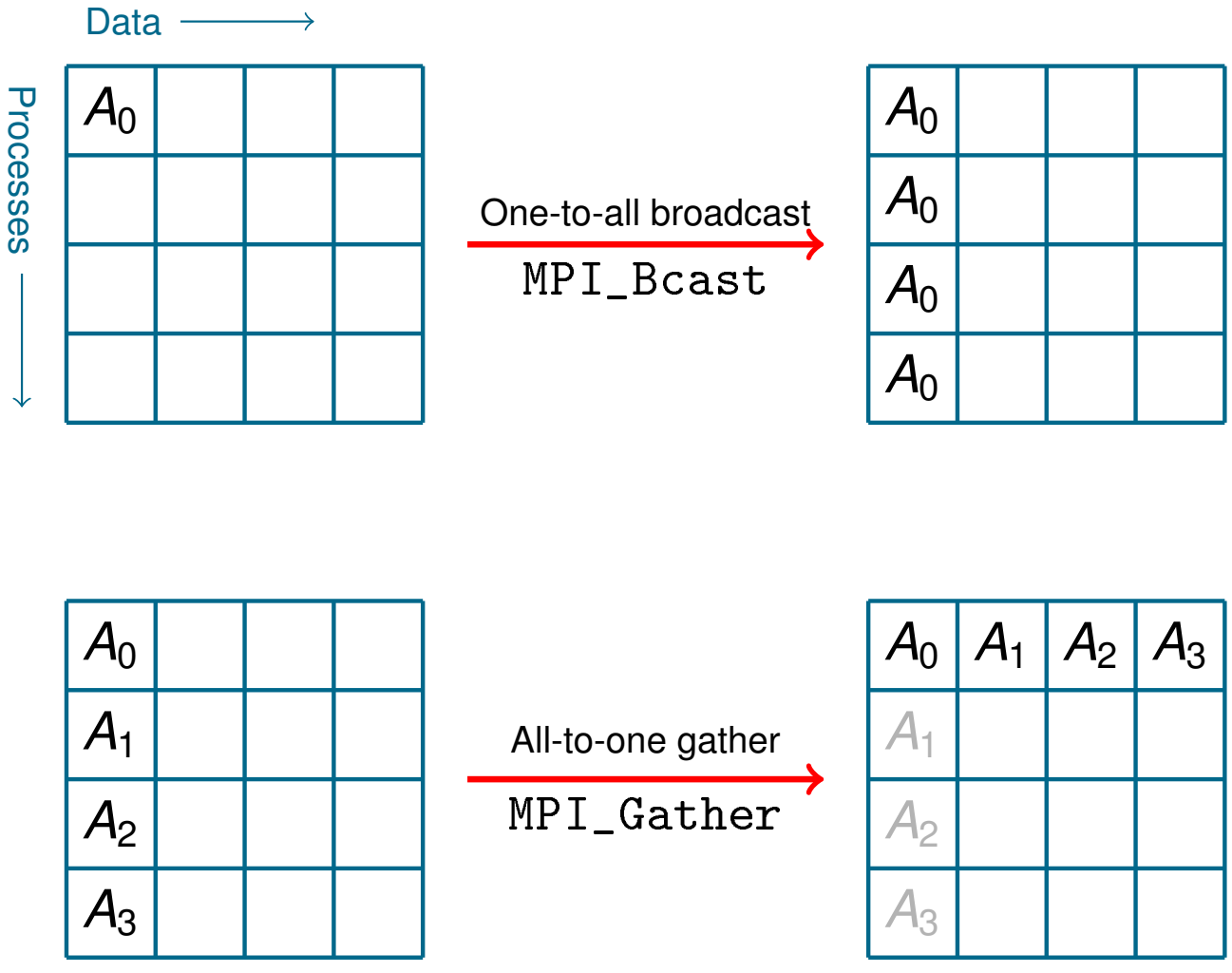
- MUST involves all of the processes in a group, and
- are more efficient and less tedious to use compared to point-to-point communication.

1. Barrier synchronization
2. Broadcast (one-to-all)
3. Scatter (one-to-all)
4. Gather (all-to-one)
5. Global reduction operations: min, max, sum (all-to-all)
6. All-to-all data exchange
7. Scan across all processes

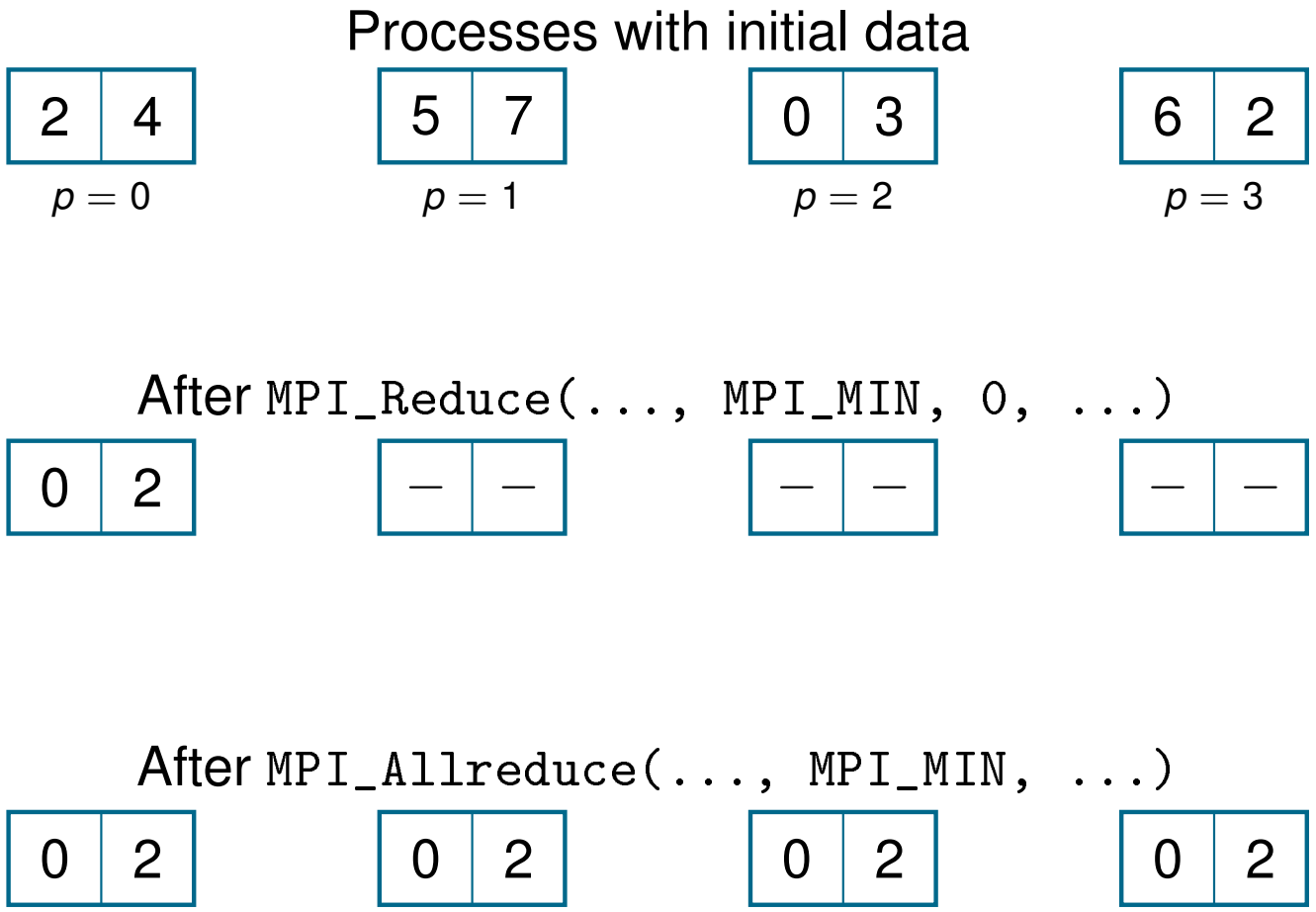
An example (synchronization between processes):

```
MPI_Barrier(comm);
```

Collective operations



Global reduction



Global reduction

`MPI_Reduce`(*sbuf*, *rbuf*, *count*, *datatype*, *op*, *root*, *comm*);

data *envelope*

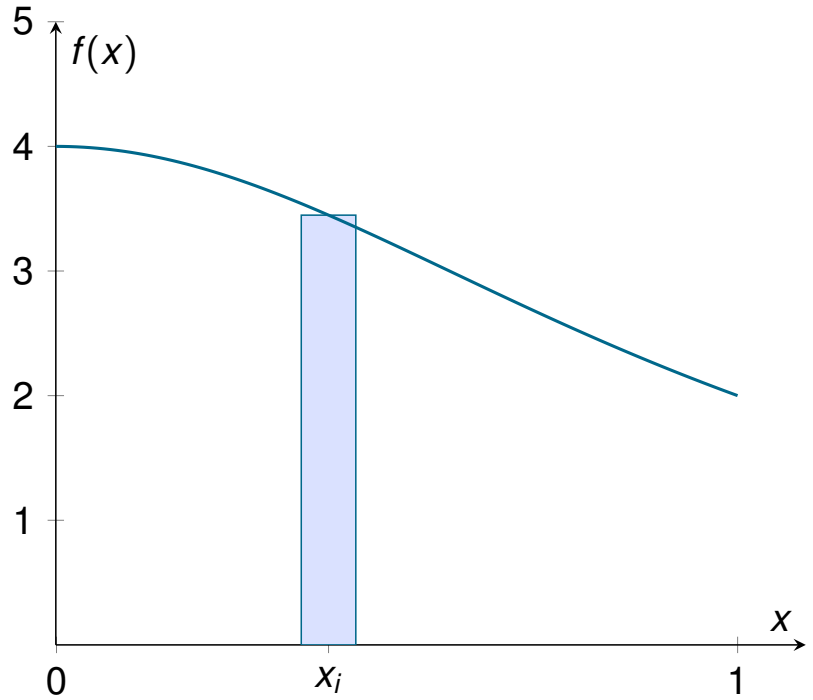
`MPI_Allreduce`(*sbuf*, *rbuf*, *count*, *datatype*, *op*, *comm*);

data *envelope*

Examples of predefined operations (C):

- `MPI_SUM`
- `MPI_PROD`
- `MPI_MIN`
- `MPI_MAX`

Numerical integration



$$A_i = \left(\frac{4}{1+x_i^2}\right) \cdot h, \qquad \text{with} \quad x_i = \left(i + \frac{1}{2}\right) \cdot h$$

where $i = 0, \ldots, n-1$, and $h = 1/n$.

Numerical integration

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x \approx h\sum_{i=0}^{n-1} \frac{4}{1+x_i^2} = \pi_n$$

Calculating pi with MPI in C

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    if (argc < 2) {
        printf("Requires argument: number of intervals.");
        return 1;
    }

    int nprocs, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int nintervals = atoi(argv[1]);
    double time_start;
    if (rank == 0) {
        time_start = MPI_Wtime();
    }
```

Calculating pi with MPI in C (cntd.)

```
    double h = 1.0 / (double)nintervals;
    double sum = 0.0;
    int i;
    for (i = rank; i < nintervals; i += nprocs) {
        double x = h * ((double)i + 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    double my_pi = h * sum;

    double pi;
    MPI_Reduce(&my_pi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double duration = MPI_Wtime() - time_start;
        double error = fabs(pi - 4.0 * atan(1.0));
        printf("pi=%e, error=%e, duration=%e\n", pi, error, duration);
    }

    MPI_Finalize();

    return 0;
}
```

Things to consider

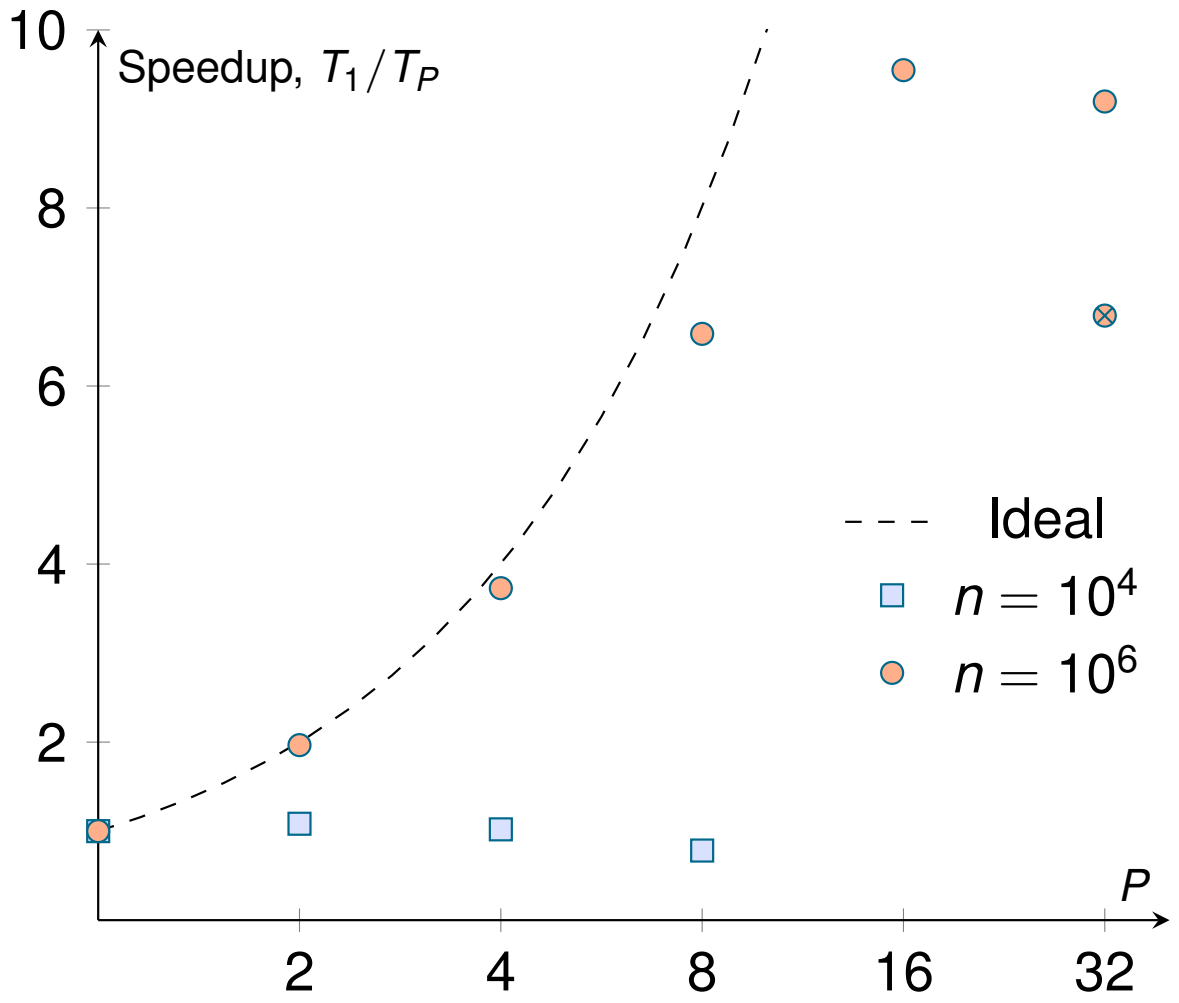
1. Is the program correct, e.g., is the convergence rate as expected?
2. Is the program load-balanced?
3. Do we get the same value of π_n for different values of P?
4. Is the program scalable?

Convergence test

n	error $= \lvert\pi - \pi_n\rvert$
10	$8.33 \cdot 10^{-4}$
10^2	$8.33 \cdot 10^{-6}$
10^3	$8.33 \cdot 10^{-8}$
10^4	$8.33 \cdot 10^{-10}$
10^5	$8.37 \cdot 10^{-12}$

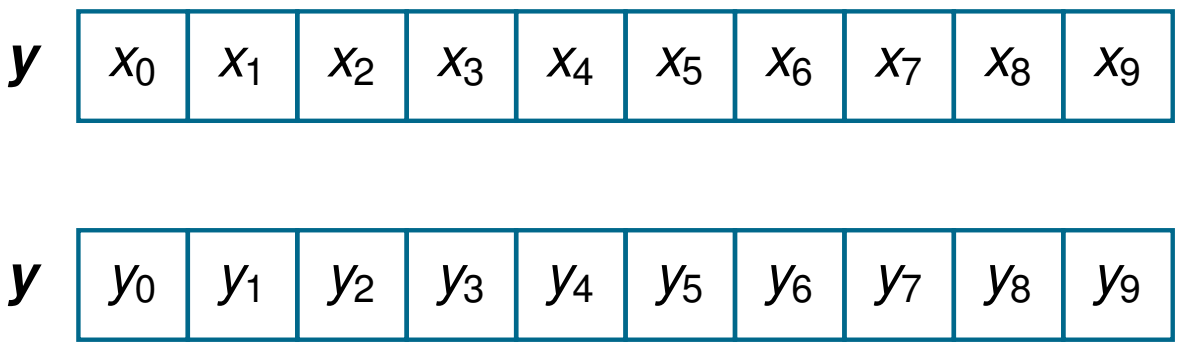
Hence, $|\pi - \pi_n| \sim \mathcal{O}(h^2)$ where $h = 1/n$.

Scalability: timing results on Vilje

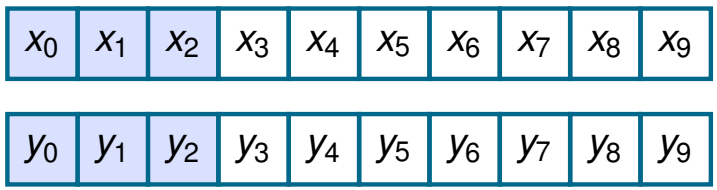


Inner product

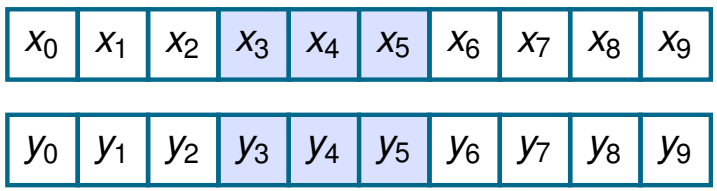
$$\sigma = \boldsymbol{x}^{\mathsf{T}} \boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{y} = \sum_{m=0}^{N-1} x_m y_m$$



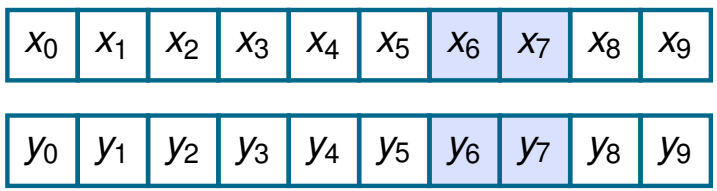
Distribution of work



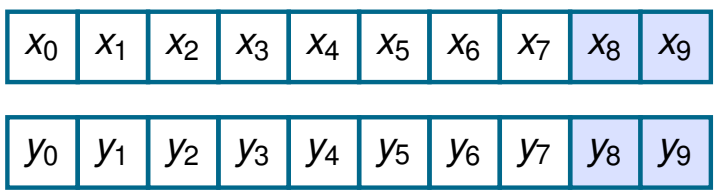
$$\omega_0 = \sum_{m=0}^{2} x_m y_m$$



$$\omega_1 = \sum_{m=3}^{5} x_m y_m$$



$$\omega_2 = \sum_{m=6}^{7} x_m y_m$$



$$\omega_3 = \sum_{m=8}^{9} x_m y_m$$

Program on processor p

$\boldsymbol{x}, \boldsymbol{y}$: vectors of dimension N

$$\omega_p = \sum_{m \in \mathcal{N}_p} x_m y_m$$

Send ω_p to processor $q \neq p$

Receive ω_q from processor q

$$\sigma = \sum_{q=0}^{P-1} \omega_q .$$

Local and global numbering

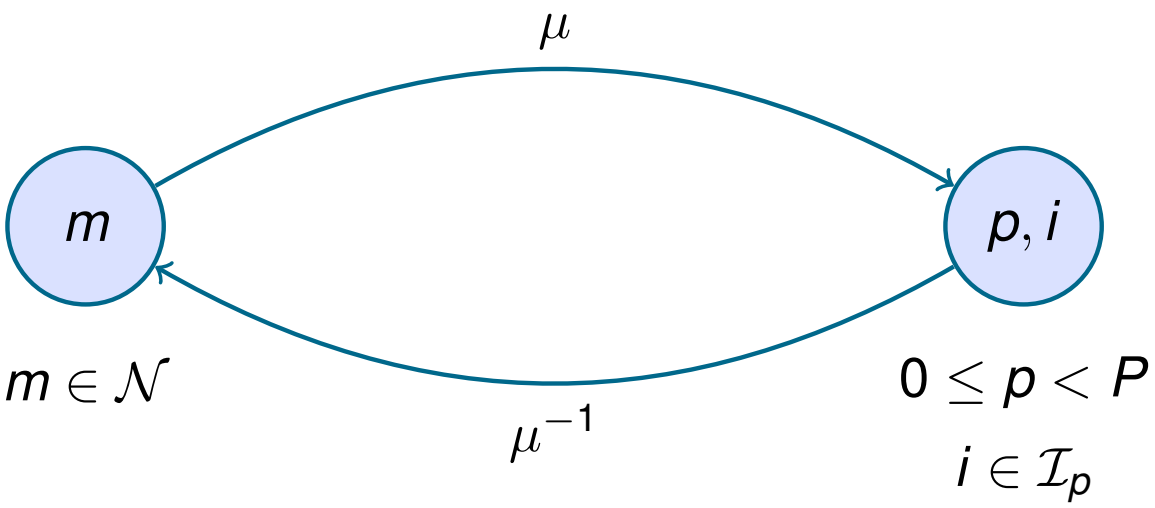
Global indices: $\mathcal{N} = \{0, 1, 2, \ldots, N-1\}$. Cardinality $|\mathcal{N}| = N$.
Subdivision:

$$\mathcal{N} = \bigcup_{p=0}^{P-1} \mathcal{N}_p, \qquad \mathcal{N}_p \cap \mathcal{N}_q = \emptyset, \qquad p \neq q.$$

— $\mathcal{N}_p$: a subset of global indices
— $N_p = |\mathcal{N}_p|$: the number of global indices assigned to process p.
— $\mathcal{I}_p = \{0, 1, \ldots, N_p - 1\}$: a *local* index set

Note $|\mathcal{I}_p| = |\mathcal{N}_p| = N_p$.

Local and global numbering



Local and global numbering

Requirements for the numbering of entities:

- The local-to-global numbering is a one-to-one relationship.
- Local indices are found in the interval $\mathcal{I}_p = \{0, 1, \ldots, N_p - 1\}$.
- Global indices are not necessarily alway contiguous: this is the case when the global number of entities is not known.

The entities are usually renumbered globally such that global indices are packed contiguously.

Each process p is then assigned a **range** $[i_0^p, i_{N_p}^p[$ with

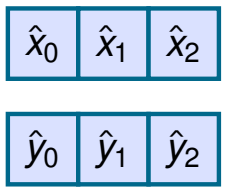
1. i_0^p is the process range *offset*,
2. $i_{N_p}^p$ is the *local size*.

Local and global numbering

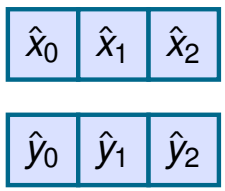
How to compute the offset with MPI?

1. MPI 1: `MPI_Scan`
2. MPI 2: `MPI_Exscan`

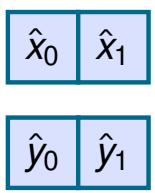
Distribution of work and data



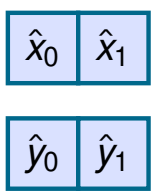
$$\omega_0 = \sum_{i=0}^{2} \hat{x}_i \hat{y}_i$$



$$\omega_1 = \sum_{i=0}^{2} \hat{x}_i \hat{y}_i$$



$$\omega_2 = \sum_{i=0}^{1} \hat{x}_i \hat{y}_i$$



$$\omega_3 = \sum_{i=0}^{1} \hat{x}_i \hat{y}_i$$

Program on processor p

$\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}$: vectors of dimension l_p

$$\omega_p = \sum_{i=0}^{l_p-1} \hat{x}_i \hat{y}_i = \sum_{i=0}^{l_p-1} x_{\mu^{-1}(p,i)} y_{\mu^{-1}(p,i)} = \sum_{m \in \mathcal{N}_p} x_m y_m.$$

Send ω_p to processor $q \neq p$

Receive ω_q from processor q

$$\sigma = \sum_{q=0}^{P-1} \omega_q.$$

Global reduction

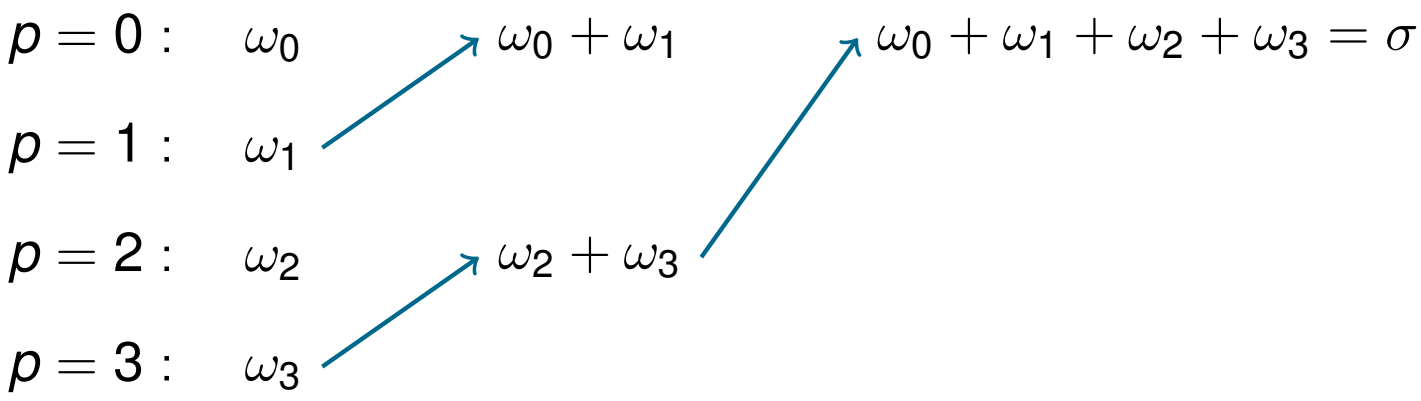
Reduction algorithm for the global sum

$$\sigma = \sum_{q=0}^{P-1} \omega_q.$$

`MPI_Reduce`(ω, σ, `1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD`)
(the answer will be known to process zero), or

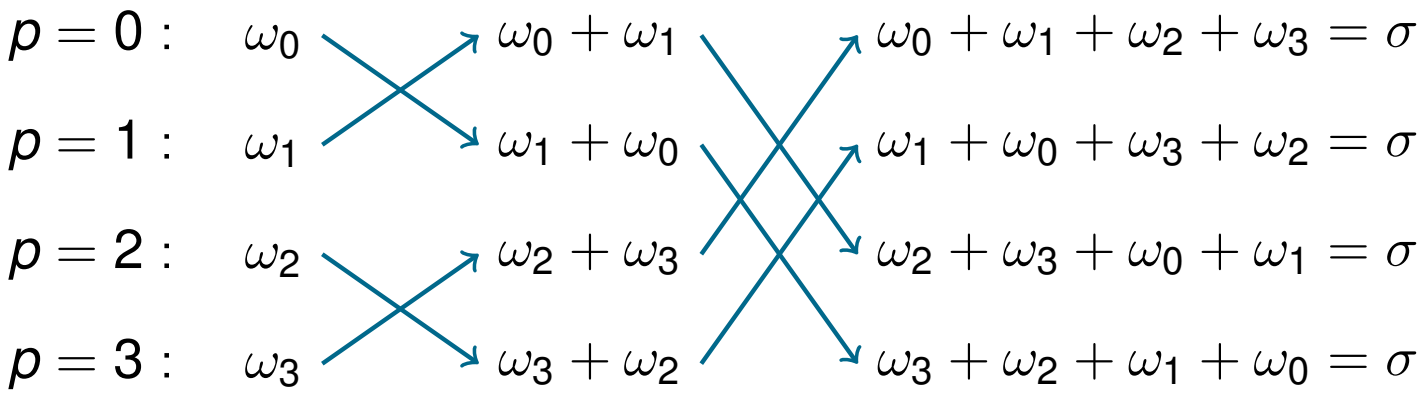
`MPI_Allreduce`(ω, σ, `1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD`) (the answer will be known to every process)

Global sum $P = 4$, `MPI_Reduce`



Can be completed in $\log_2 P$ steps.

Global sum $P = 4$, `MPI_Allreduce`



Can be completed in $\log_2 P$ steps.

Program on processor p

$\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}$: vectors of dimension l_p

$$\sigma = \sum_{i=0}^{l_p-1} \hat{x}_i \hat{y}_i = \sum_{i=0}^{l_p-1} x_{\mu^{-1}(p,i)} y_{\mu^{-1}(p,i)} = \sum_{m \in \mathcal{N}_p} x_m y_m.$$

for $d = 0, \ldots, \log_2 P - 1$

Send σ to processor $q = p \bar{\vee} 2^d$

Receive σ_q from processor $q = p \bar{\vee} 2^d$

$\sigma = \sigma + \sigma_q$

end

Here, $\bar{\vee}$ is exclusive or. p and $p \bar{\vee} 2^d$ differ only in bit d.