


Introduction to parallel programming using MPI

Jakub Gałecki, j.galecki@icm.edu.pl

23.04.2025

Presenter bio

Member of the Application and User Support team at ICM

Research relates to application of the finite element method in fluid mechanics

Experience with MPI in the context of physics simulation

One of the maintainers of the ICM HPC software stack

Resources

The MPI Standard `https://www.mpi-forum.org/docs/`

Implementation documentation, e.g., `https://docs.open-mpi.org/en/v5.0.x/index.html`

Gropp W., Lusk E., Skjellum A. 2014. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.

Agenda

Types of CPU parallelism

What is MPI?

Basic definitions

Anatomy of a message

Point-to-point communication

Collective communication

Non-blocking communication

Practical considerations

Exercises

Types of (CPU) parallelism

Type of parallelism	Means of access	Limit
Vector instructions (SIMD)	Compiler, intrinsics	Vector size (512b on modern CPUs)
Instruction-Level Parallelism	Compiler	Number of execution units
SMT (hyperthreading)	Code, OS, IPC	Number of logical/physical cores (2)
Multithreading	Code, OS, IPC	Number of physical cores
Multiple processors (ccNUMA)	-	Number of sockets
Multiple machines (nodes)	**Network communication (MPI)**	**Wallet depth**

SIMD – Single Instruction, Multiple Data

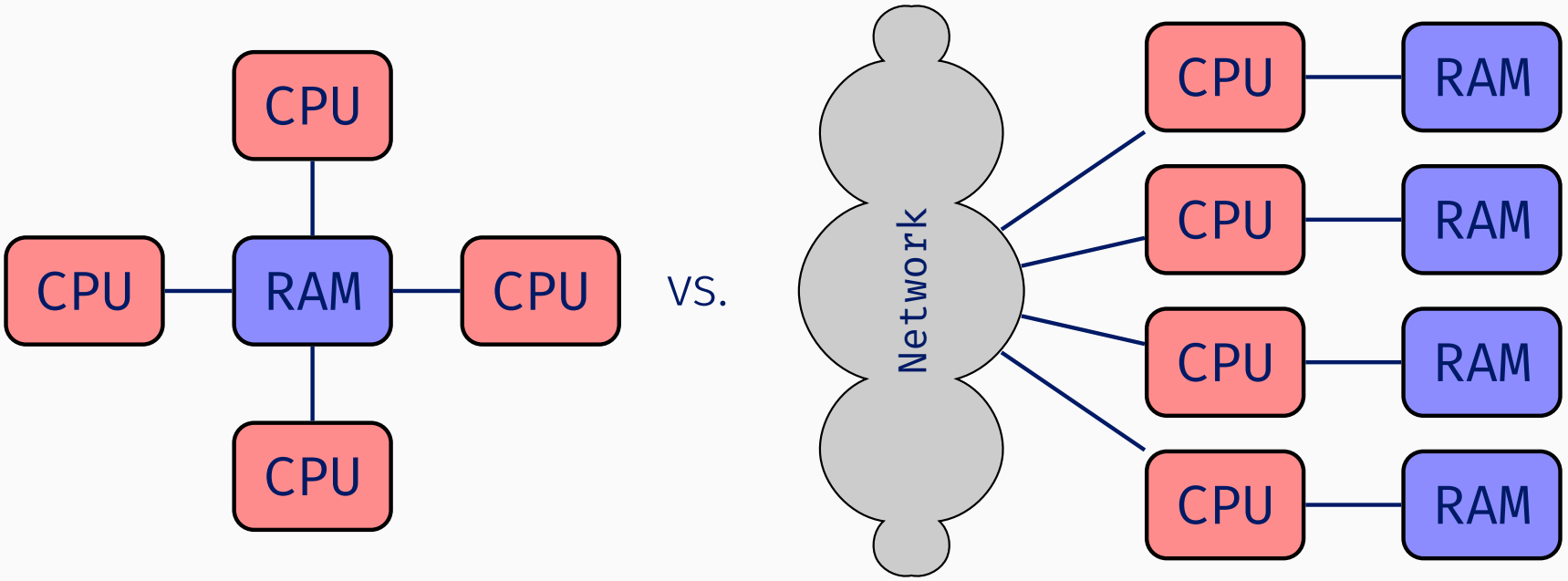
SMT – Simultaneous Multi-Threading

OS – Operating System

IPC – Inter-process communication

ccNUMA – cache-coherent Non-Uniform Memory Access

Shared vs distributed memory



Shared vs distributed memory

$\{\text{thread}\} \subseteq \{\text{processes}\} \subset \text{system}$

Shared memory:

- Threads share the virtual address space – no cost to sharing data
- Issues around synchronization – data races
- Several programming models: OpenMP, TBB, pthreads, ...
- Inherently limited scalability

Distributed memory:

- Processes have separate address spaces – communication is needed to share data
- Ubiquitous standard in HPC: MPI
- Scalability is limited only by the size of your cluster

MPI

Message **P**assing **I**nterface

Standard set by the MPI Forum

Native interface in Fortran, C, ~~C++~~

Current version: 4.1 (Nov 2023)

`https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf`

Competing implementations:

- MPICH – `https://www.mpich.org/`
- OpenMPI – `http://www.open-mpi.org/`
- Others...

Cornerstone of the HPC software stack

MPI programming model

SPMD – Single Program Multiple Data:

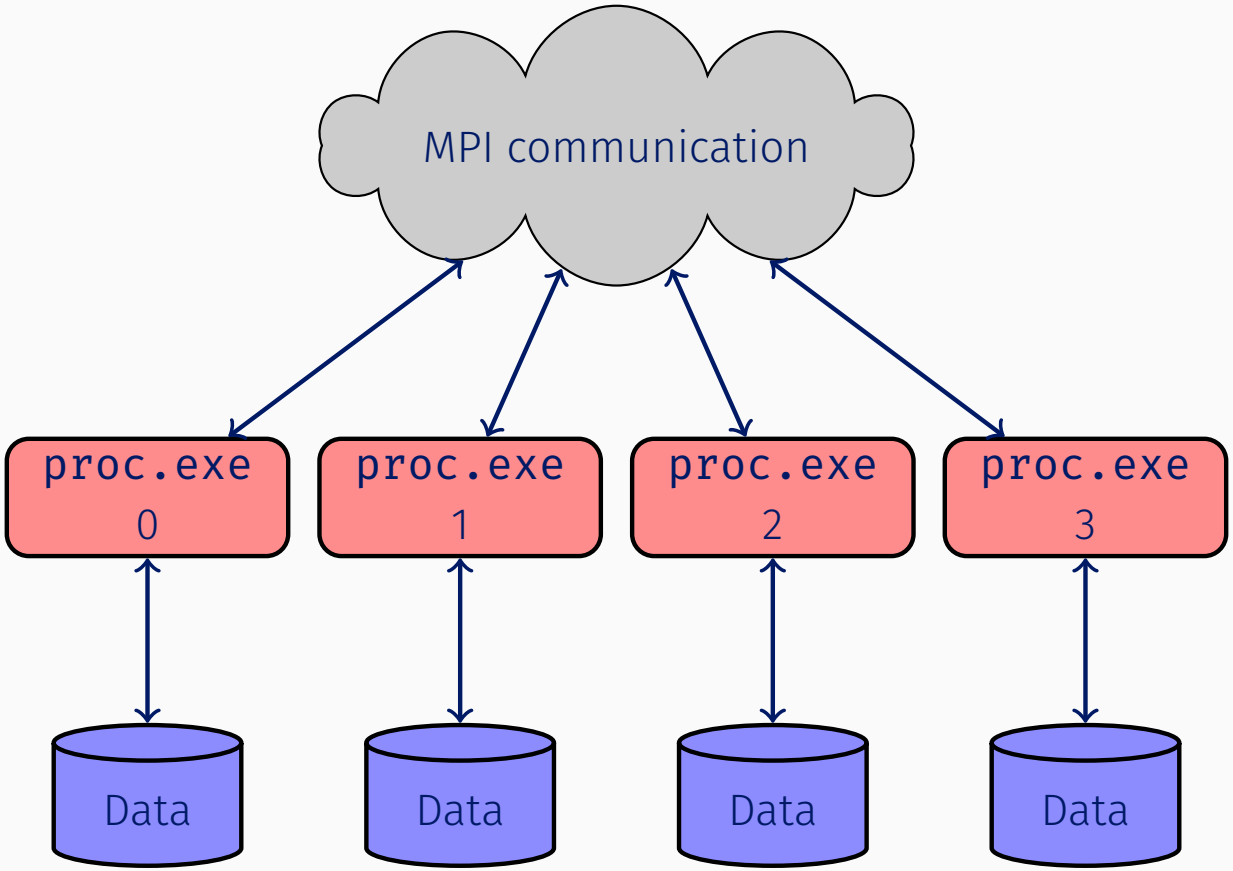
- We execute the launcher, passing the relevant executable, number of instances we wish to launch **n**, program arguments, etc...
- Launcher launches **n** sub-processes, each executing the specified binary
- Sub-processes carry out the actual computation, communicating via the parent process

MPMD also supported, but not very popular

Assumptions:

- Every process has a private virtual address space, backed by a physical address space which may or may not be shared with other processes (abstraction)
- Every exchange of data between the processes is explicit

Model SPMD


MPI communication
proc.exe
0
proc.exe
1
proc.exe
2
proc.exe
3
Data
Data
Data
Data

Basic concepts

Communicator – abstraction describing a set of processes and their associated communication context

Rank – unique ID of a process within a communicator. Rank IDs are always consecutive natural numbers, starting at 0.

2 communicators available by default:

- `MPI_COMM_SELF` – communicator containing only the current process
- `MPI_COMM_WORLD` – communicator containing all processes launched by the launcher

We can create new communicators in order to de-conflict messages (e.g. library vs user)

Initialization and finalization

The MPI C API declaration resides in `mpi.h`

To initialize MPI we must first call:

```
int MPI_Init(int* argc, char*** argv);
```

To finalize the session we must call:

```
int MPI_Finalize();
```

No calls can be made outside of this envelope (with limited exceptions)

Communicator size and rank

To get the size of the communicator:

```
int size;
int err_code = MPI_Comm_size(MPI_COMM_WORLD, &size);
```

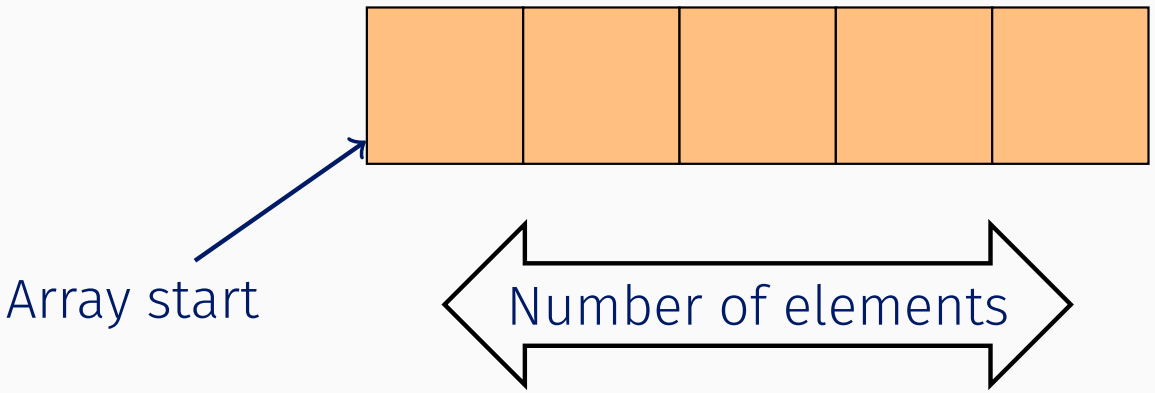
To get the rank of the current process:

```
int my_rank;
int err_code = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Message description

In the point-to-point model, processes simply exchange messages.
The message contents are described by:

- Pointer to the data array (`[const] void*`)
- Number of elements to send/receive (`int`)
- Type of the data (`MPI_Datatype`)



Data types

MPI describes data types using dynamic handles of type `MPI_Datatype`

This lets the user define their own data types (outside the scope of this presentation)

Built-in integer and floating-point types are predefined, e.g.:

- `char` – `MPI_CHAR`
- `int` – `MPI_INT`
- `double` – `MPI_DOUBLE`

C++ libraries commonly use type deduction to avoid having to manually specify this argument

P2P communication

P2P messages are addressed using the following elements:

- Communicator (context)
- Sender (rank)
- Recipient (rank)
- Tag (user-provided ID used to differentiate messages)

In order for a message to be received, these descriptors must match

Wildcards: `MPI_ANY_SOURCE`, `MPI_ANY_TAG`

P2P communication

Send:

```
int MPI_Send(const void* buf, int size, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm);
```

```
const double buf[3] = {3.14, 42., 2.71};
const int dest = 1, tag = 0;
int err = MPI_Send(buf, 3, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
```

Receive:

```
int MPI_Recv(void* buf, int size, MPI_Datatype type,
             int src, int tag, MPI_Comm comm, MPI_Status* status);
```

```
double buf[3];
const int src = 0, tag = 0;
int err = MPI_Recv(buf, 3, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

MPI_Status, MPI_Probe

The last argument of `MPI_Recv` (type `MPI_Status`) provides us with additional information about the received data. e.g., the number of elements which were actually received (`MPI_Get_count`).

`MPI_Probe` lets us examine an incoming message before actually receiving it:

```
MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status);
```

If we wish, we can ignore the status by passing

`MPI_STATUS_IGNORE`

How exactly is MPI sending my message?

2 types of behavior:

- Synchronous – data is send directly from the user-provided buffer
- Buffered – data is first copied to an intermediate buffer

Mode	MPI function	Behavior
Synchronous	`MPI_Ssend`	Fully synchronous
Buffered	`MPI_Bsend`	Buffered (requires configuring a buffer)
Standard	`MPI_Send`	Implementation decides
Ready send	`MPI_Rsend`	Must be called after the corresponding receive

All of these calls are **blocking**, meaning control will return to the caller only after the message buffer is no longer being used by MPI

Collective communication

Collective communication – all ranks in the communicator participate

Describes a global pattern of data exchange

Examples:

- Barrier
- Broadcast
- Gather, AllGather
- Scatter
- All-to-all
- Reduce, AllReduce
- Scan

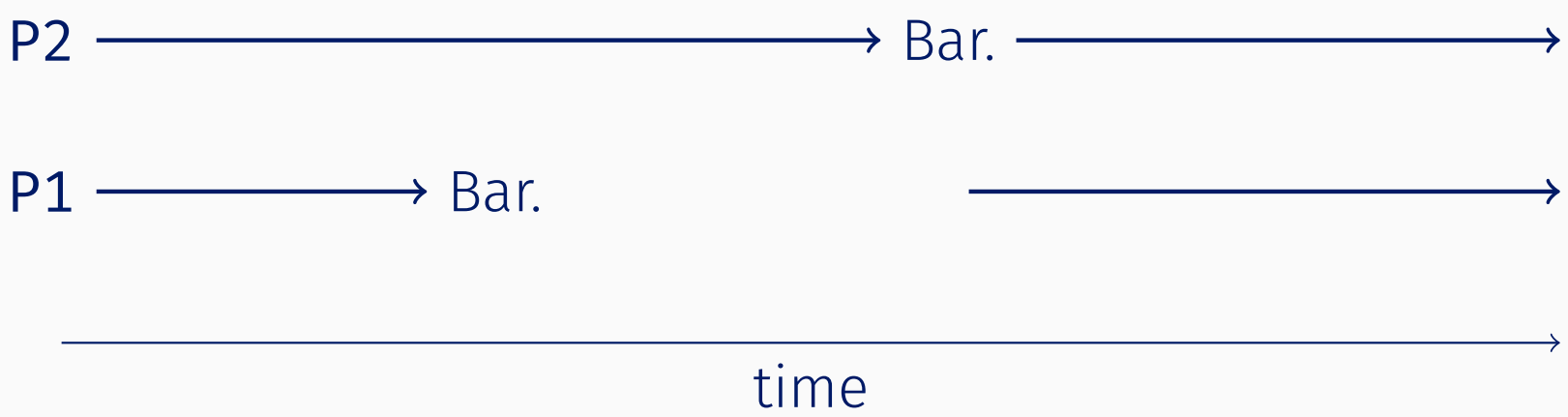
Barrier

Blocks the caller until all ranks reach the barrier

Should be used sparingly

Useful e.g. for debugging, profiling, I/O sync

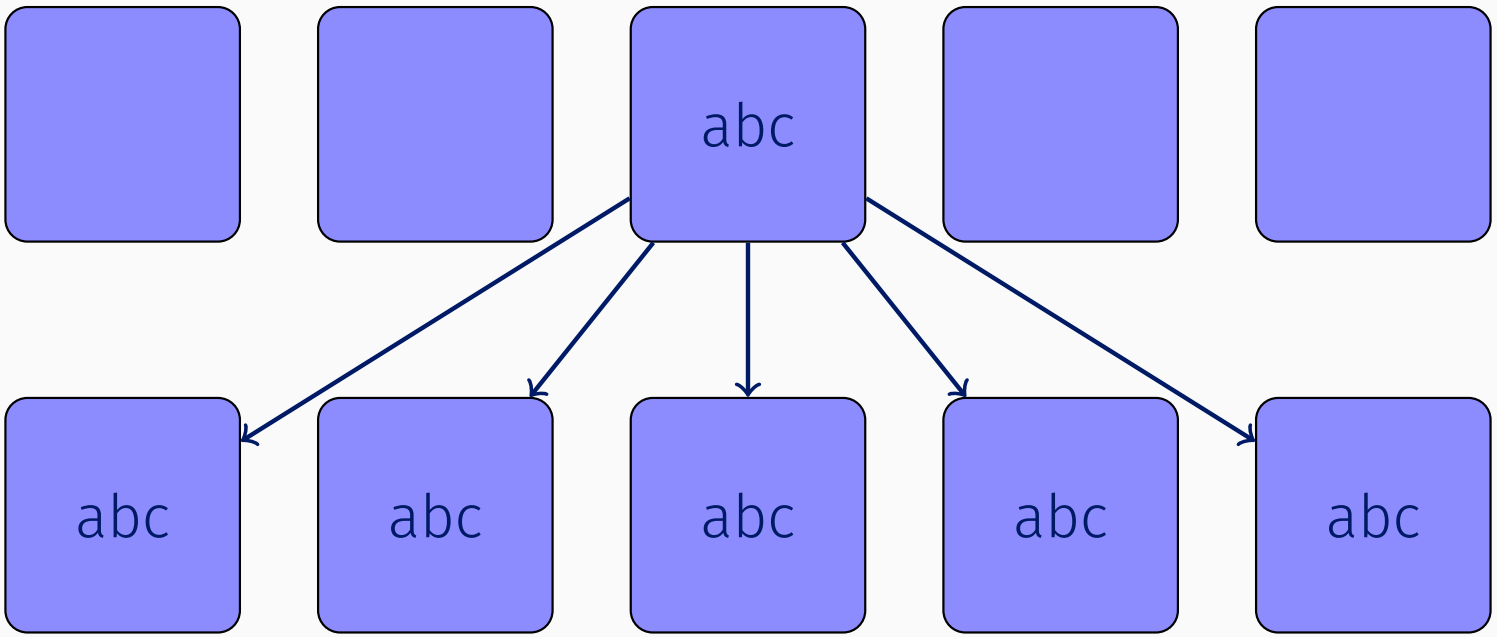
```
int MPI_Barrier(MPI_Comm comm);
```



Broadcast

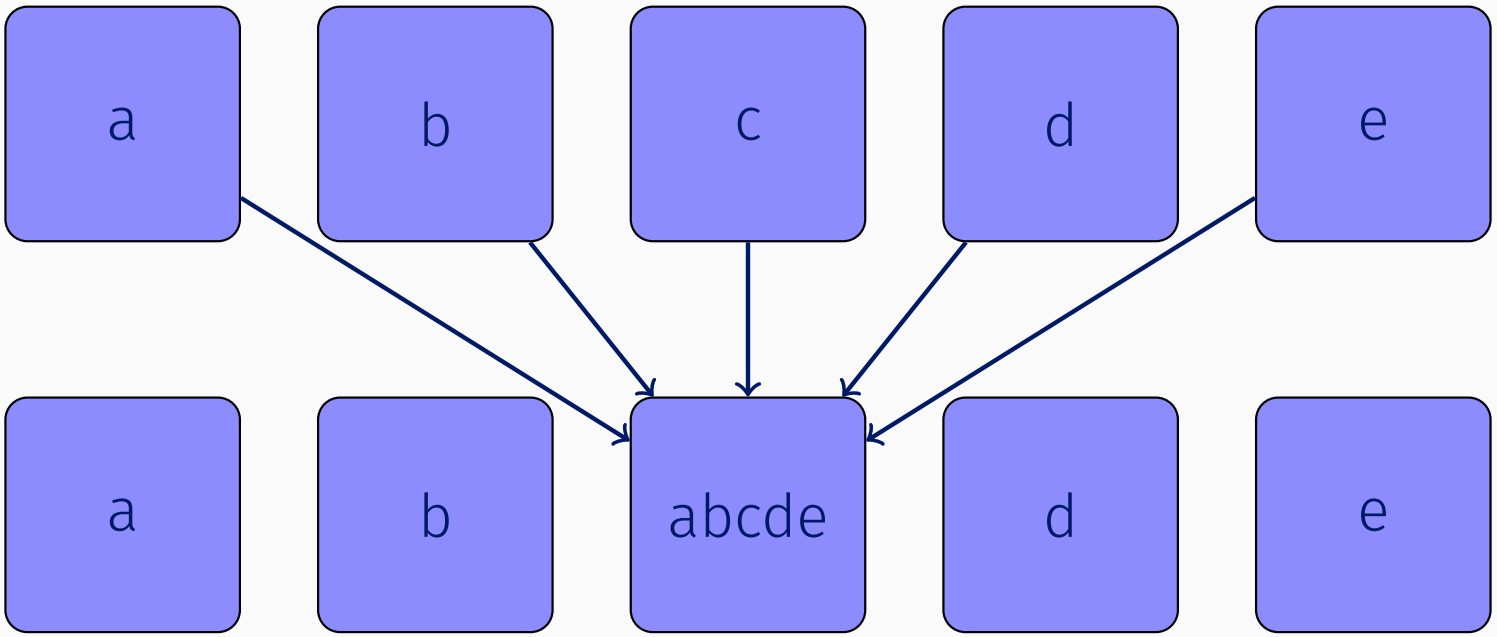
Broadcast – one rank sends the same data to all other ranks

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



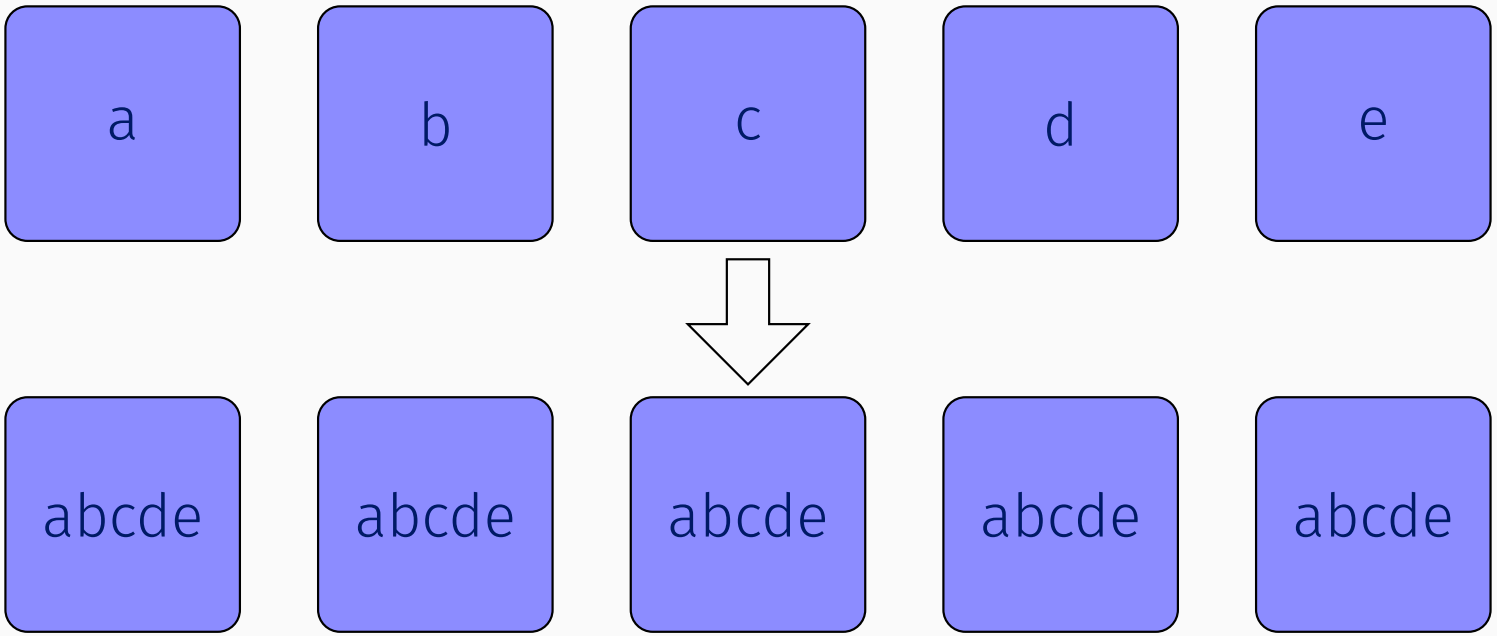
Gather – one rank gathers data from other ranks

```
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm);
```



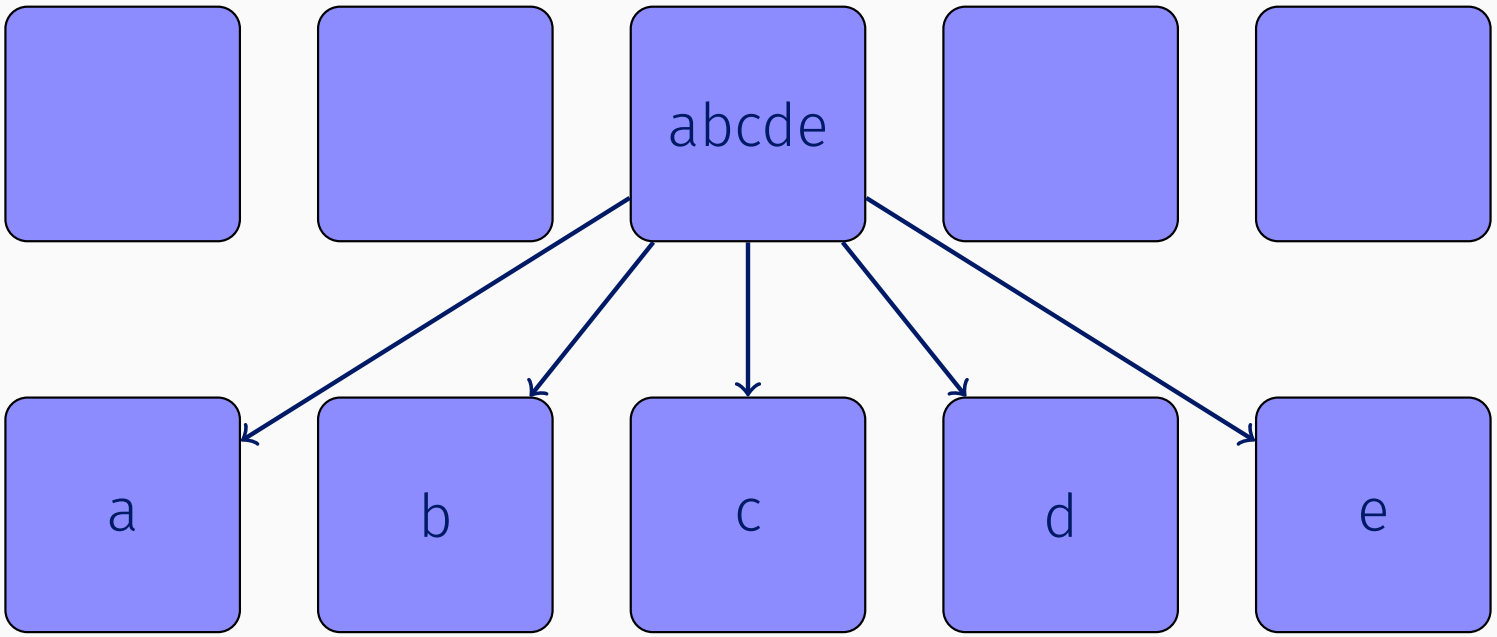
AllGather = Gather + Broadcast

```
int MPI_Allgather(const void *sendbuf, int  sendcount,
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm);
```



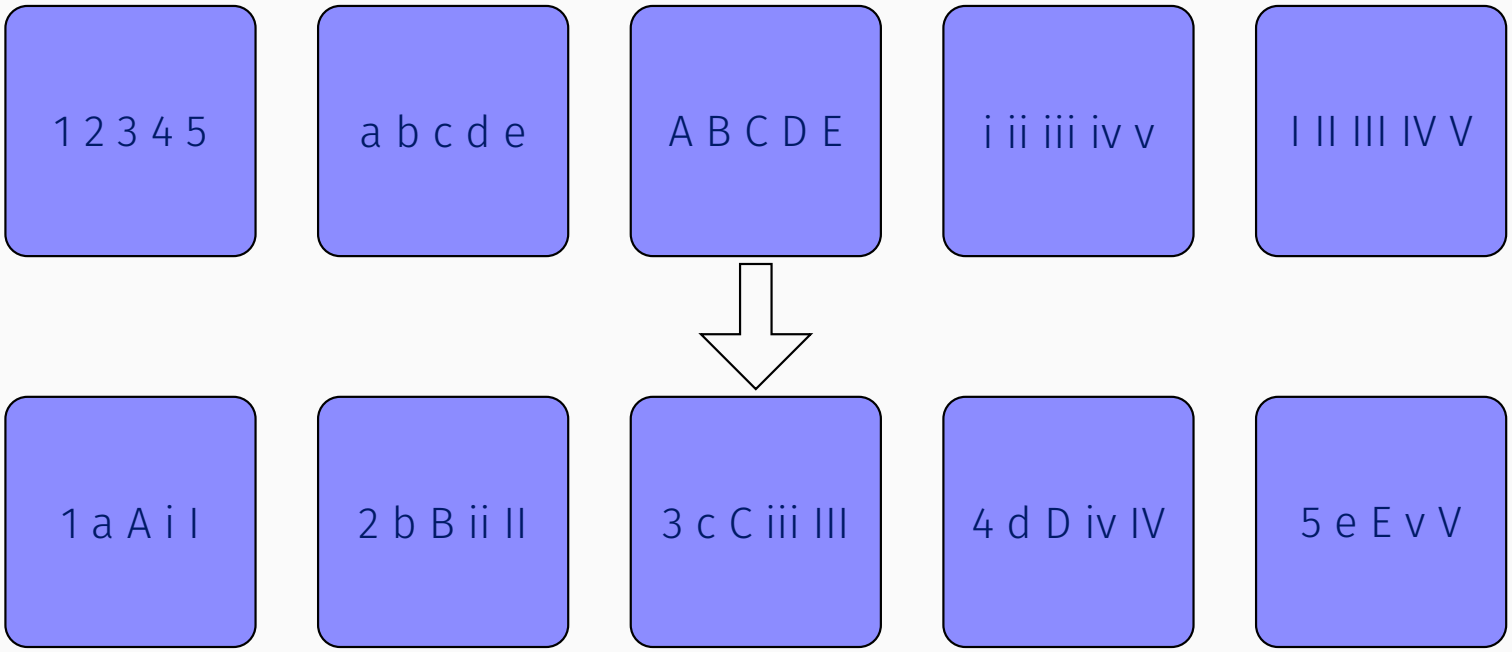
Scatter – one rank distributes its data among other ranks

```
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```



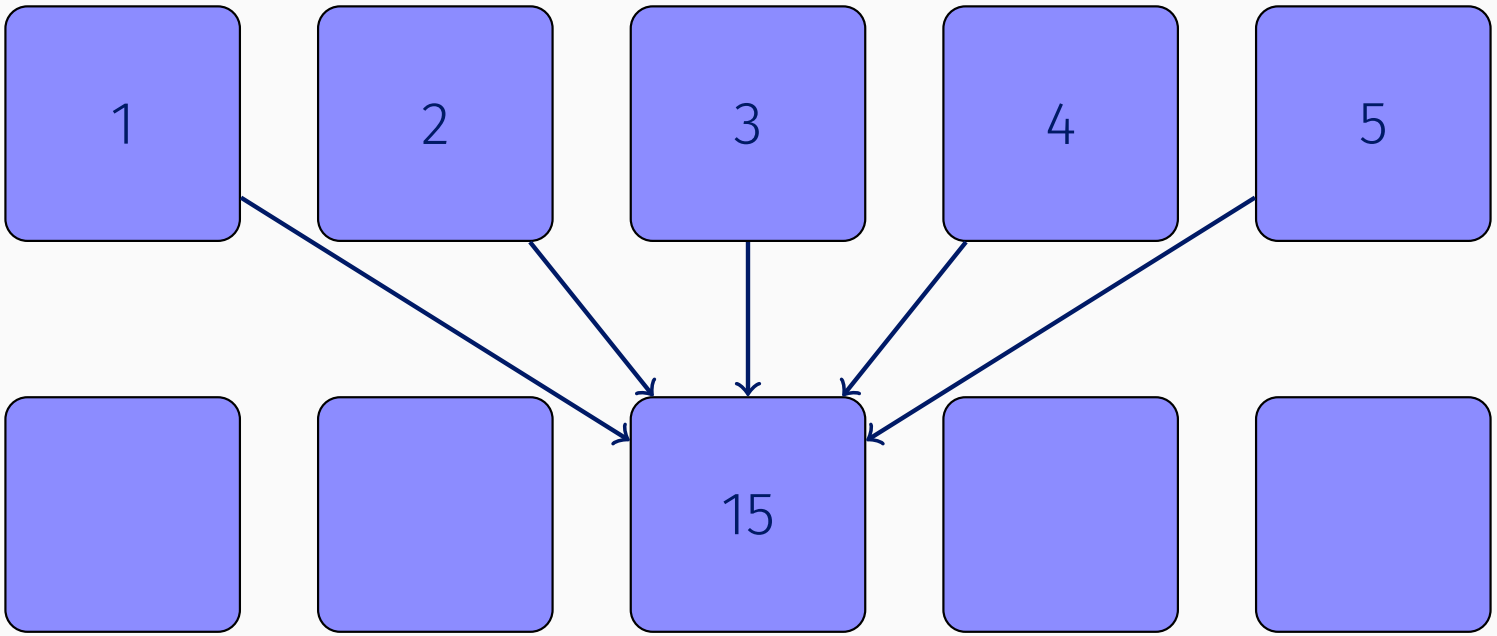
All-to-all – all ranks distribute data among other ranks

```
int MPI_Alltoall(const void* sendbuf, int sendcount,
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, MPI_Comm comm);
```



Reduce – Global reduction (sum, max, etc.)

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



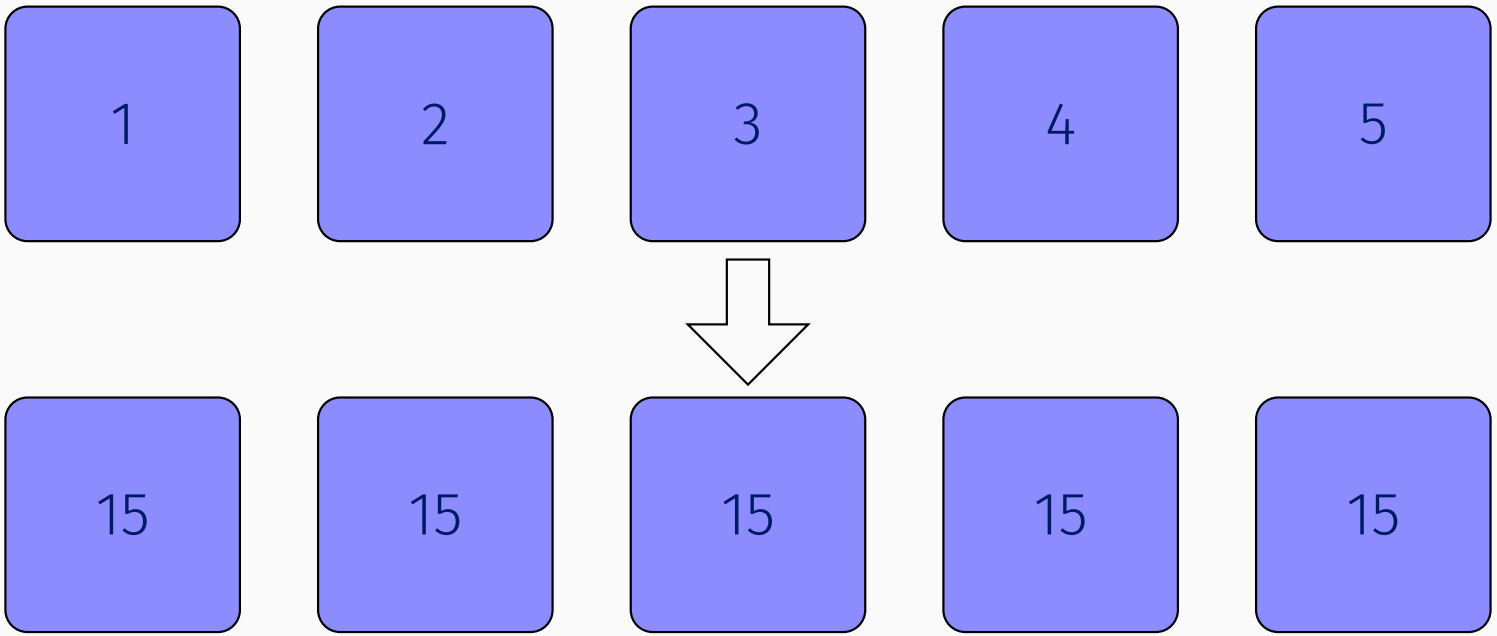
Types of reductions

Define our own or use the presets:

Nazwa	Znaczenie
MPI_MAX	max
MPI_MIN	min
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	binary and
MPI_LOR	
MPI_BOR	
MPI_LXOR	
MPI_BXOR	
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

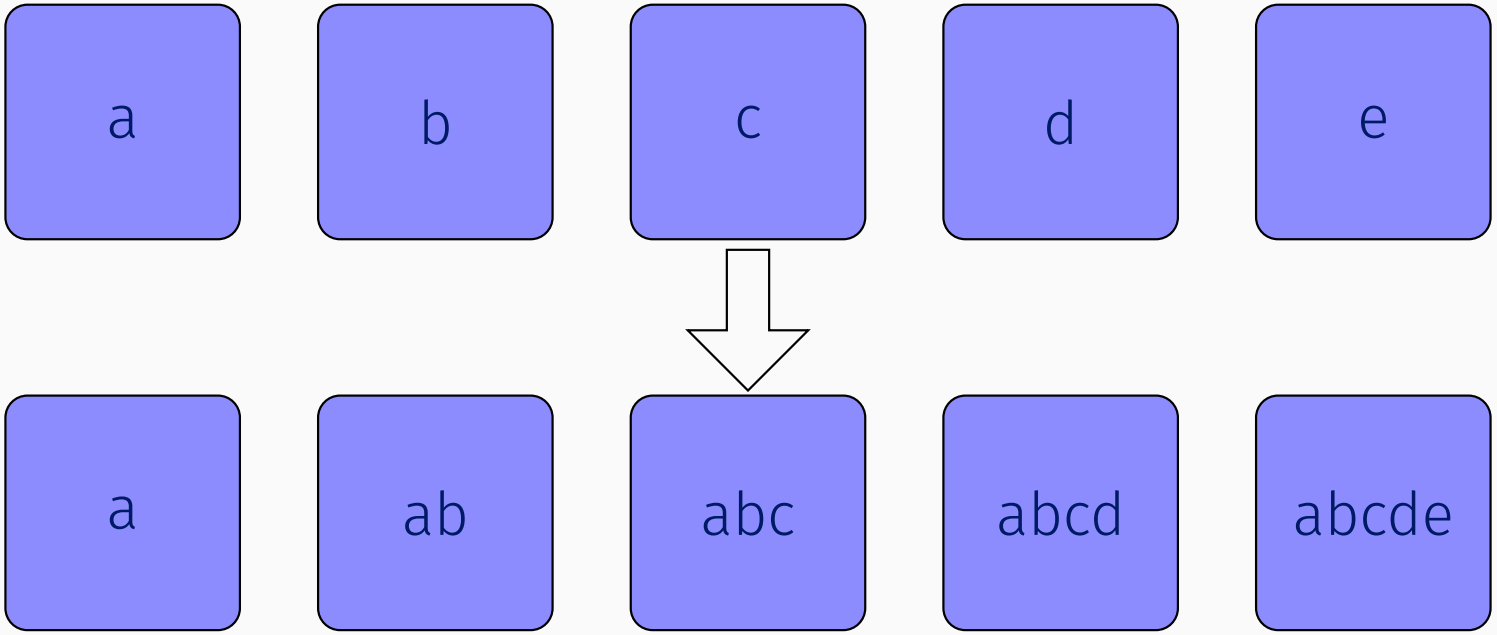
AllReduce = Reduce + Broadcast

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



Scan

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



Non-blocking communication

Control returns to the caller immediately after invoking the function

Communication happens “in the background”

Communication request handle used to query the status

Testing – check for completion

Waiting – block until complete

The buffer can only be reclaimed after completion – user-side memory management

Non-blocking communication is foundational for scalability – **communication and computation overlap**

Non-blocking communication – P2P

Send:

```
int MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request* request);
```

Receive:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request* request)
```

Other modes available: `MPI_Ibsend`, `MPI_Issend`, etc.

Non-blocking collective communication also possible: `MPI_Ibcast`, `MPI_Iscatter`, etc.

MPI_Request

`MPI_Request` – abstraction representing the handle to the non-blocking communication

Testing:

```
int MPI_Test(MPI_Request* request, int* flag, MPI_Status* status);
```

Waiting:

```
int MPI_Wait(MPI_Request* request, MPI_Status* status);
```

Cancellation*:

```
int MPI_Cancel(MPI_Request* request);
```

A data leak occurs if `MPI_Request` is not handled in one of these ways

MPI_Request

Utilities for handling arrays of requests

Test:

```
int MPI_Testall(int count, MPI_Request array_of_requests[],
                int* flag, MPI_Status array_of_statuses[]);

int MPI_Testany(int count, MPI_Request array_of_requests[],
                int* index, int* flag, MPI_Status* status);

int MPI_Testsome(int incount, MPI_Request array_of_requests[],
                 int* outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);
```

Wait:

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status* array_of_statuses);

int MPI_Waitany(int count, MPI_Request array_of_requests[],
                int* index, MPI_Status* status);

int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int* outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[]);
```

Compiling MPI applications

Implementations supply compiler wrappers responsible for setting any flags and linkage of shared objects

- `mpifort` / `mpif77` / `mpif90`
- `mpicc`
- `mpicxx` / `mpic++`*

CMake support: `find_package(MPI)` from the `FindMPI` module

Launching MPI applications

Launcher: `mpiexec` / `mpirun`

```
mpiexec -n 4 my_awesome_app
```

Various options available for setting hardware bindings, etc.

On clusters resources are managed by scheduling systems

MPI natively integrates with slurm – ranks map directly to tasks

Resource requirements can be specified via `sbatch` options, in the job script we simply call

```
srun my_awesome_app
```

MPI + multithreading

MPI offers support for multithreading within MPI processes

4 modes available:

- Single – process only has 1 thread
- Funneled – MPI calls may only occur from the main thread
- **Serialized** – MPI calls will not occur concurrently
- Multiple – MPI calls may occur concurrently (MPI responsible for sync)

When multithreading, MPI should be initialized using `MPI_Init_thread`:

```
int MPI_Init_thread(int* argc, char*** argv,
                    int required, int* provided);
```

MPI + CUDA

As we've described it so far, using MPI with CUDA would require the following steps:

1. Copy data from device to host memory
2. Send data to the destination process
3. On the destination process, copy the data from host to device memory

Server GPUs have networking capabilities, ideally we'd like to send data directly between GPUs

Since the introduction of UVA, we can achieve this simply by passing device memory pointers to MPI calls.

C++ wrappers for MPI

Simpler interface thanks to classes

Automatic type deduction for built-in types

Request lifetime management via RAII

Bundling requests and data

Leverage C++20 ranges for ease and safety

```
Comm comm{MPI_COMM_WORLD};
int rank = comm.rank();
auto send_data = std::vector{1., 2., 3.};
auto recv_data = std::array<double, 3>{};
int dest = 1, src = 2, tag = 42;
auto send_request = comm.send(send_data, dest, tag);
auto recv_request = comm.recv(recv_data, src, tag);
send_request.wait();
recv_request.wait();
```

Summary

Distributed memory programming is the most scalable approach to parallelism

MPI can be used to exchange data between processes which potentially reside on different physical machines

MPI offers many abstractions, such as collective and nonblocking communication which, facilitate correctness and performance

MPI integrates well with slurm

MPI does not preclude – and in fact enables – other types of parallelism

Further topics

Virtual communicator topologies

Distributed I/O

One-sided communication (RMA)

Defining data types and operations

Thanks for listening!

Feel free to reach out at `jgalecki@icm.edu.pl`

Exercises

Write an MPI program which prints the rank of the current process and the number of launched processes.

Is the order of the messages deterministic?

If not, how can we synchronize the printing operation?

2. Distributed inner product

The inner (scalar) product of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ is given by:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{N} x_i \cdot y_i$$

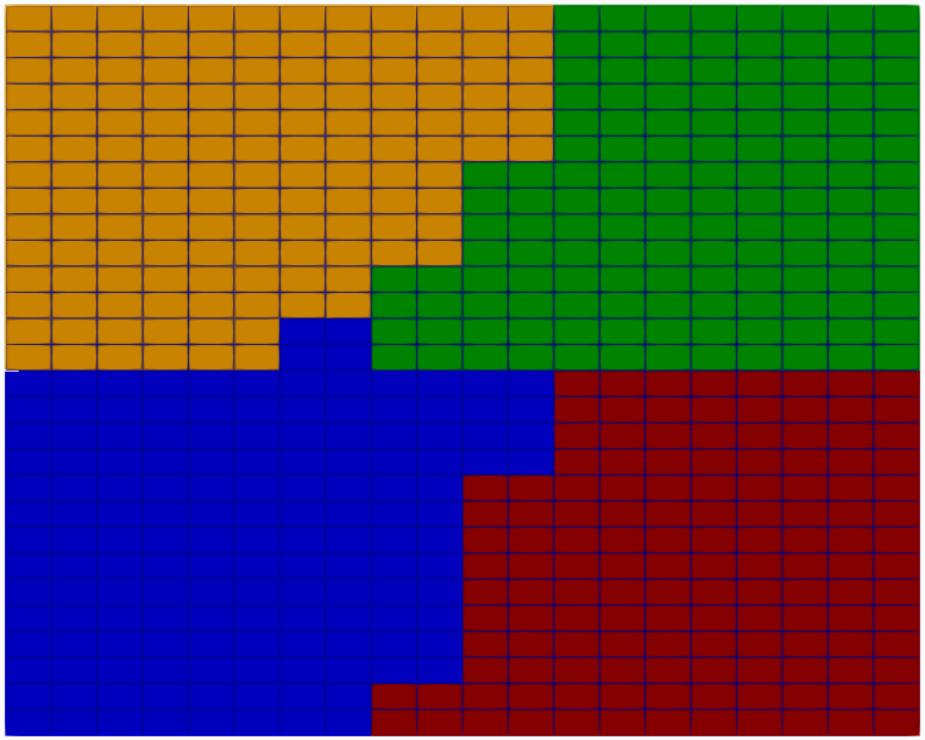
Write an MPI program which computes the inner product of 2 vectors. Assume that both vectors have the same distribution among the ranks.

Does exact way the vectors are distributed among the ranks affect the solution?

What is the simplest way to distribute vectors among ranks (local-global mapping)?

3. Halo exchange

Domain decomposition. In scientific simulations, the computational grid is distributed among MPI ranks. Computation is mostly local to each rank. Interface data must be communicated – this is called a halo exchange.



3. Halo exchange

On each rank, the vector of unknowns consists of 2 sections: owned and shared entries. Typically, the shared section is sorted by the owning partition.

$$\mathbf{x}_1 = \begin{bmatrix} x_1^1 \\ \vdots \\ x_1^n \\ x_2^1 \\ \vdots \\ x_2^m \\ x_3^1 \\ \vdots \\ x_3^l \end{bmatrix}$$

3. Halo exchange

Write a program which performs a halo exchange for a grid with the following topology:

- There are N_R ranks
- Each rank owns N_L cells
- Rank i owns cells $i \cdot N_L$ through $(i+1) \cdot N_L - 1$
- Rank i shares its last cell with rank $i + 1 \, mod \, N_R$

This setup corresponds to a 1D periodic domain, decomposed equally among the ranks