CINECA

HIGH PERFORMANCE
COMPUTING CINECA
ITALY

Parallel Programming with MPI

Part II - Introduction to Collective communications

HPC SCHOOL
COSTA RICA

CINECA

RISC2

CeNAT
Centro Nacional de Alta Tecnología

ALESSANDRO MARANI

a.marani@cineca.it || San José 2023

AGENDA

COLLECTIVE COMMUNICATIONS
Definition, general rules, MPI_Barrier

MESSAGE PASSING COLLECTIVES
Broadcast, Scatter, Gather and other examples

REDUCTION WITH COLLECTIVES
MPI_Reduce and MPI_Allreduce

COLLECTIVE AND PERFORMANCE
Overview and strategies

KABRÉ
SUPERCOMPUTADORA

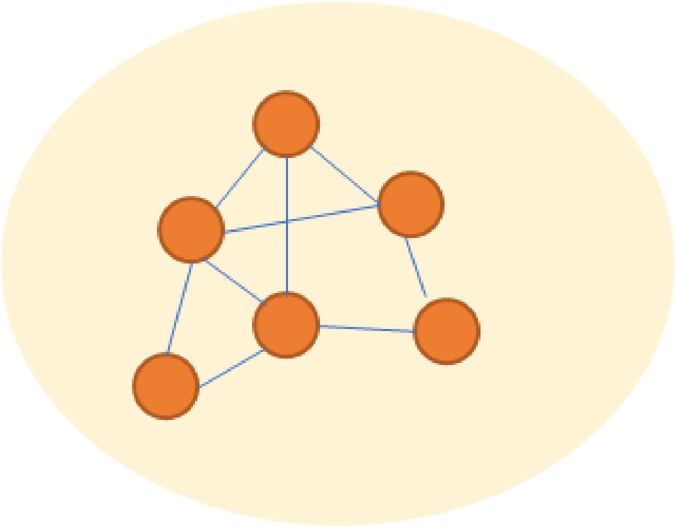
Collective communications

MPI collective communications

- Communications involving groups of processes are called ***collectives*****.**
- The following characteristics apply to collective calls:
  - The calls occur between processes in the communicator and every process ***must*** call the collective function.
  - They do not interfere with point-to-point calls.
  - No tags are required.
  - Receive buffers must match in size.
- MPI 1.0-2.0 collective calls are blocking. MPI-3 introduced non-blocking collectives.
- Designed to replace loops of point-to-point calls and as well as being more concise will be more efficient.

MPI does not define the behaviour when processes do not take part in the collective call. Possibilities include:

- The program crashes
- Deadlock
- Wrong results



Typical use cases for collectives

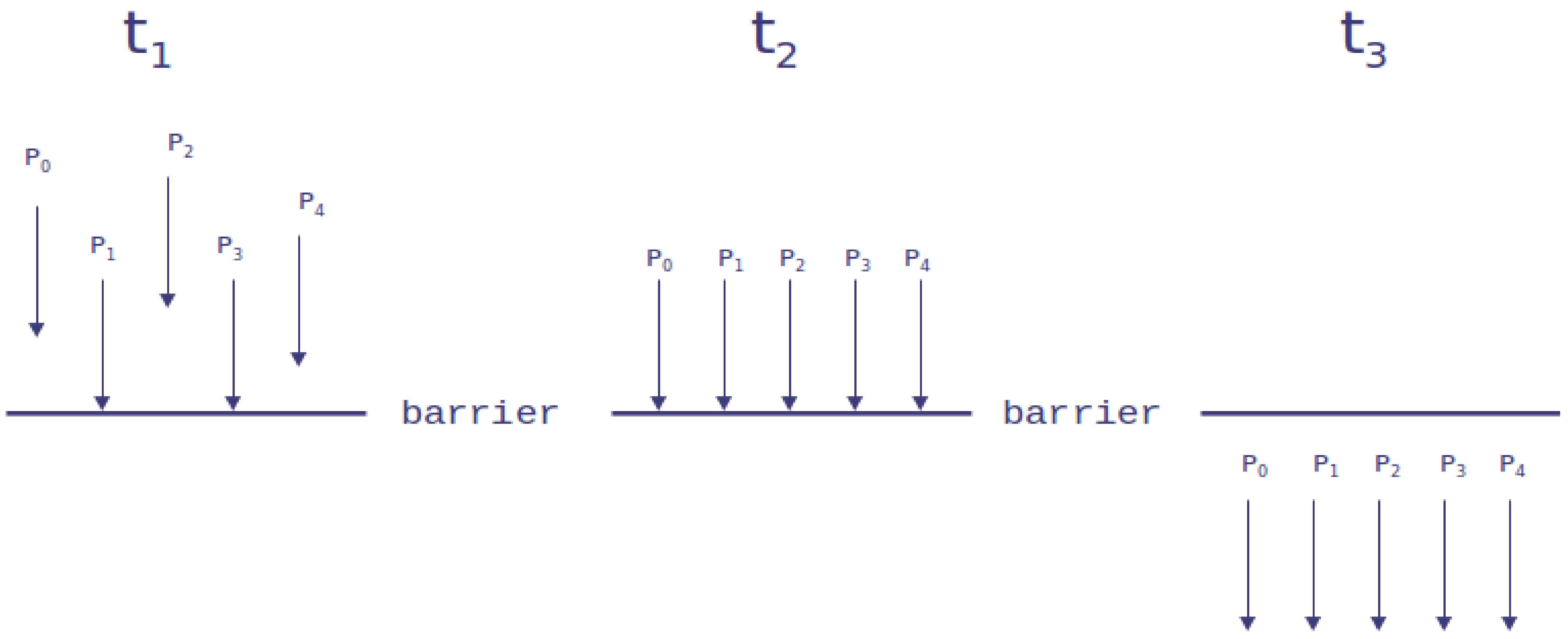
- Reading in data from a file and transferring it other tasks.
- Synchronizing data amongst all tasks.
- Calculating a value based on the data from all tasks.
- Synchronization of tasks.

We shall now look at examples of the various types of collective communication

Barriers

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

MPI_Barrier stops all processes until they are synchronized.
Useful for making sure that all ranks are at the same time point in the program.



⚠ Severe performance impact if used too often.

Message passing collectives

A small example

Write a program that initializes an array of two elements as (2.0, 4.0) only on task 0, and then sends it to all the other tasks

How can we do that with the knowledge we got so far?

Point-to-point solution

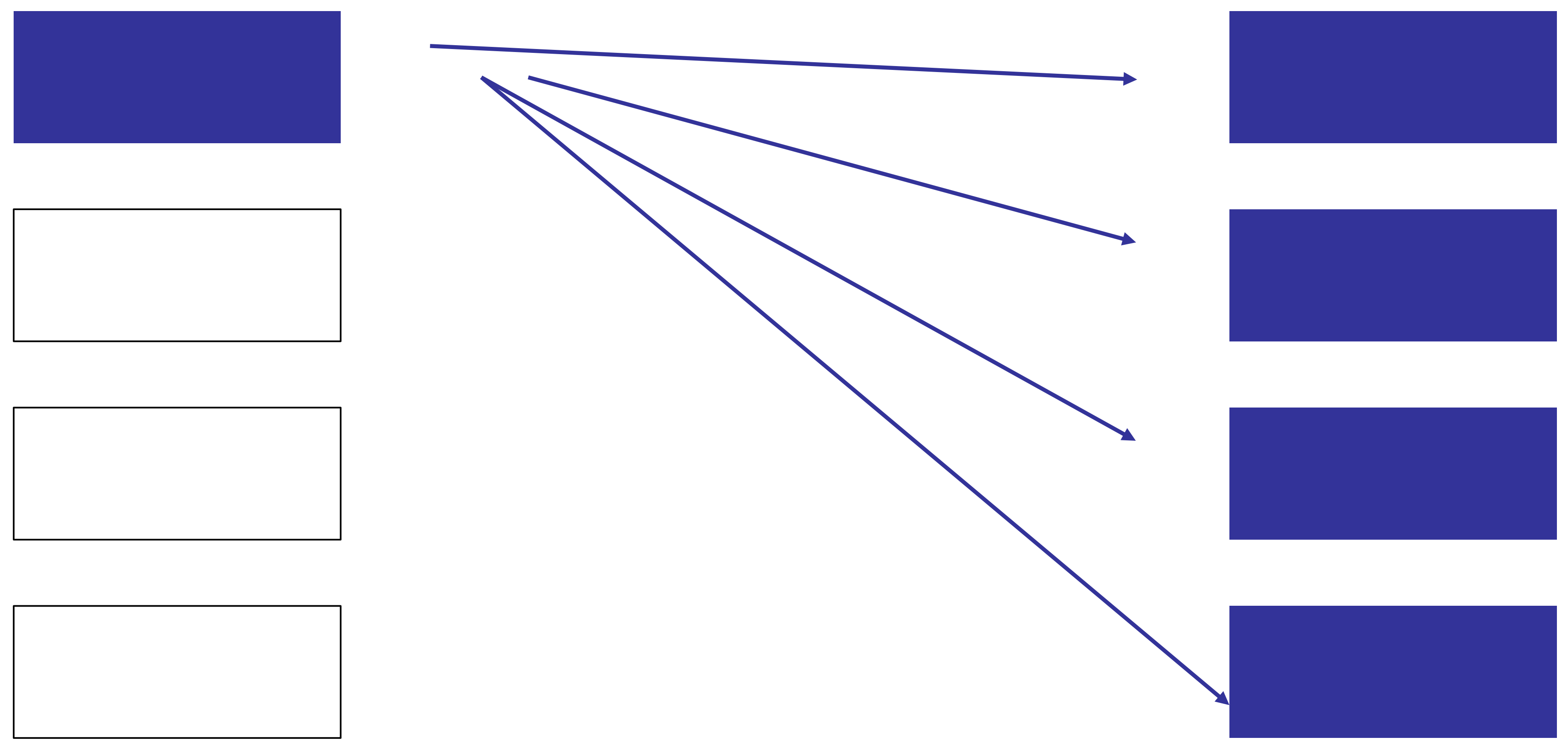
```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv) {
  int my_rank, procs, i;
  MPI_Status status;
  float a[2];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  if ( my_rank == 0 ) {
     a[0] = 2.0;
     a[1] = 4.0;
     }
  if ( my_rank == 0 ) then {
     for (i=1;i<procs;i++)
         MPI_Send(a,2,MPI_FLOAT,i,0,MPI_COMM_WORLD);
      }
  else {
     MPI_Recv(a,2,MPI_FLOAT,0,0,MPI_COMM_WORLD,&status);
     }
  printf(“%d : a[0]=, %f, a[1]=, %f\n”,my_rank,a[0],a[1]);
  MPI_Finalize();
  return 0;
}
```

Broadcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type,
              int root, MPI_Comm comm);
```

One process (*root*) sends a message to all the other ranks in the communicator.
Note that all processes must specify the same root and communicator.

Typical use case is during user input: rank 0 reads input from disk and then broadcasts to other ranks.



Broadcast solution

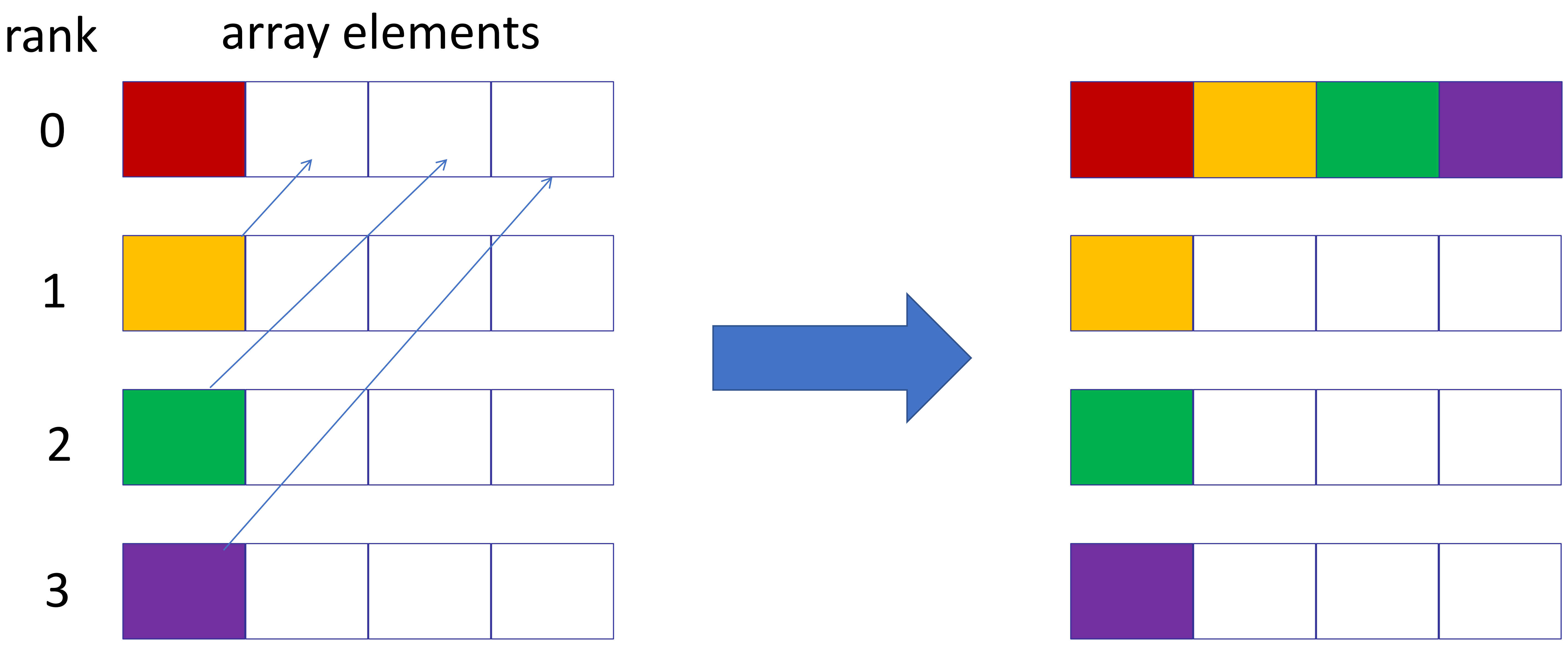
```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv) {
  int my_rank, procs;
  MPI_Status status;
  float a[2];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  if ( myid ==0 ) {
     a[0] = 2.0;
     a[1] = 4.0;
     }

  MPI_Bcast(a,2,MPI_FLOAT,0,MPI_COMM_WORLD);

  printf("%d : a[0]=, %f, a[1]=, %f\n",my_rank,a[0],a[1]);
  MPI_Finalize();
  return 0;
}
```

Typical error is to put the bcast in an "if" condition.
Remember: all tasks must make the call!

Gather

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

One process, the *root* process, collects data elements from all the processes and stores them in rank order.

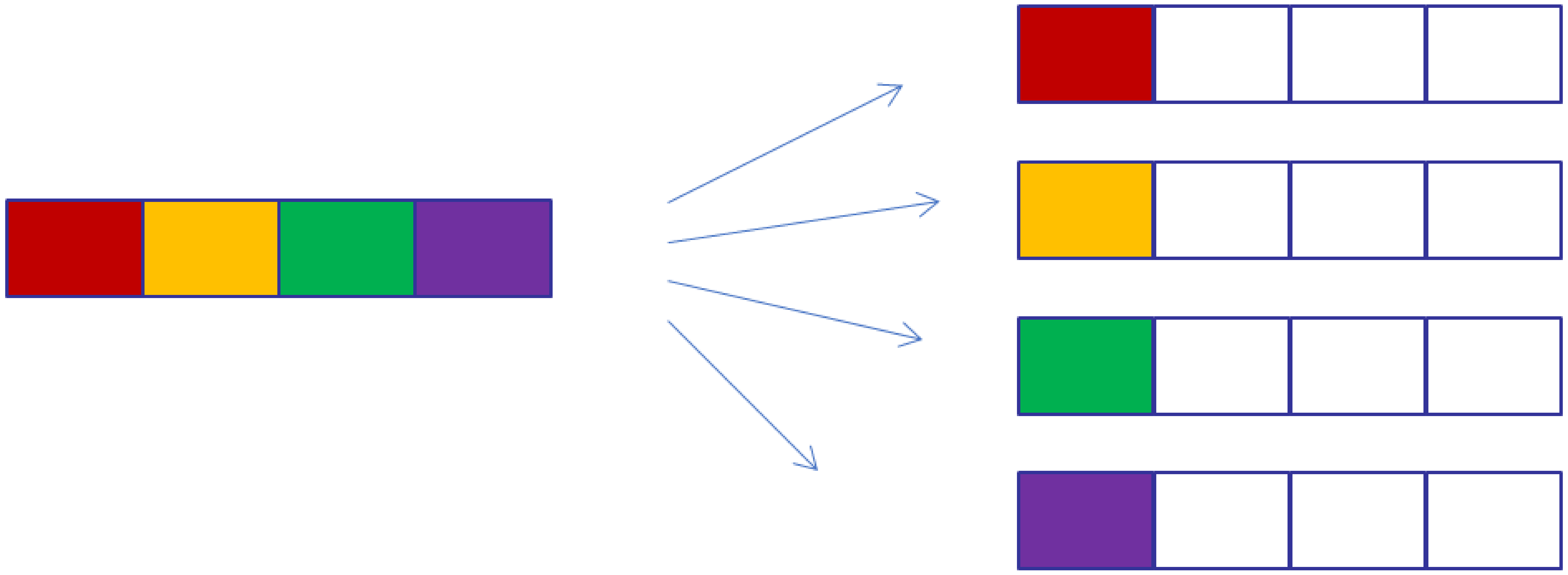


NOTE: *recvcount* is as big as the single array block expected to be received, NOT as the final array! Single blocks will be stored contiguously in memory

Scatter

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

The *root* sends a message. The message is split into *n* equal segments, the *i*-th segment is sent to the *i*-th process in the group and each process receives this message.

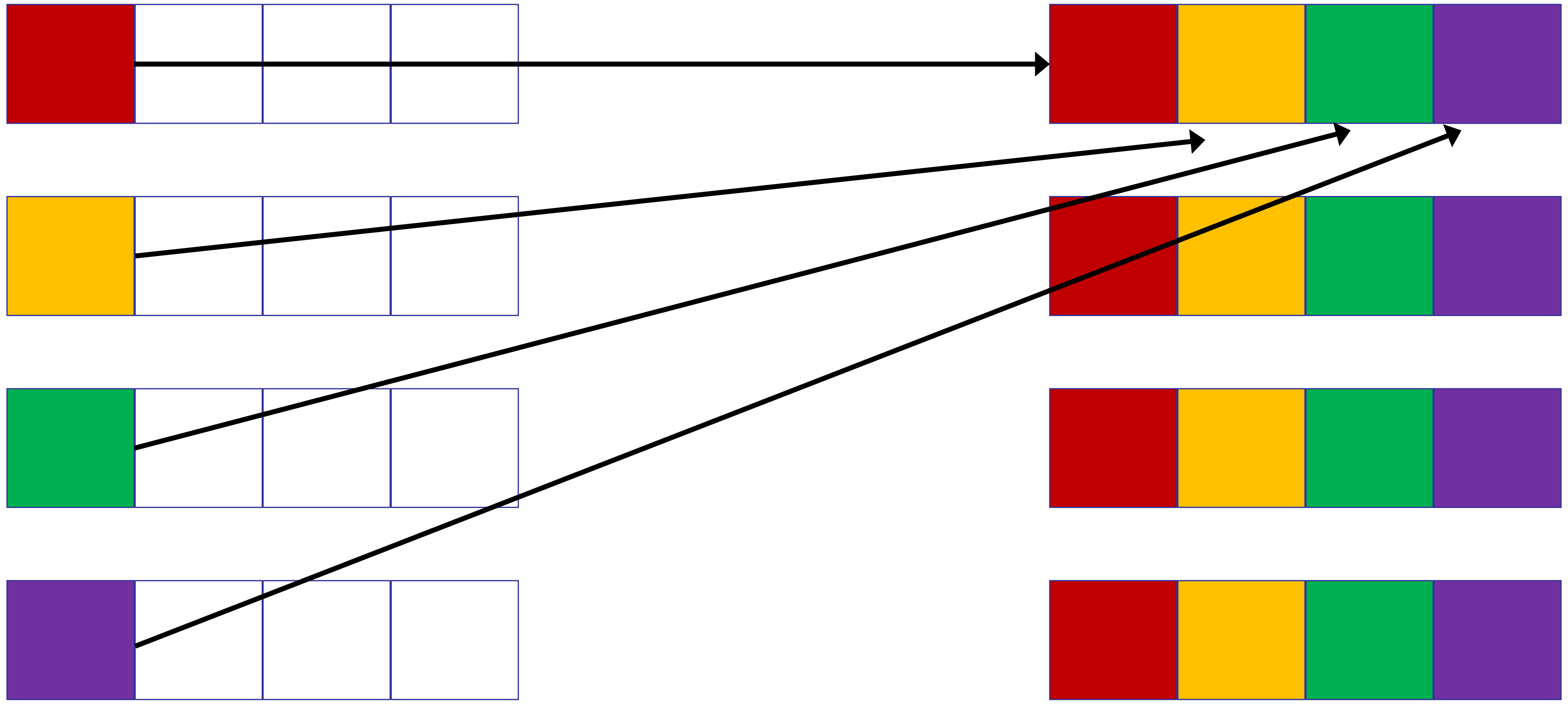


NOTE: *sendcount* is as big as the single array block expected to be sent, NOT as the full array! Single blocks will be read as contiguous in memory

Allgather

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

Combinations are possible - for example, MPI_Allgather which does a Gather followed by a Broadcast.

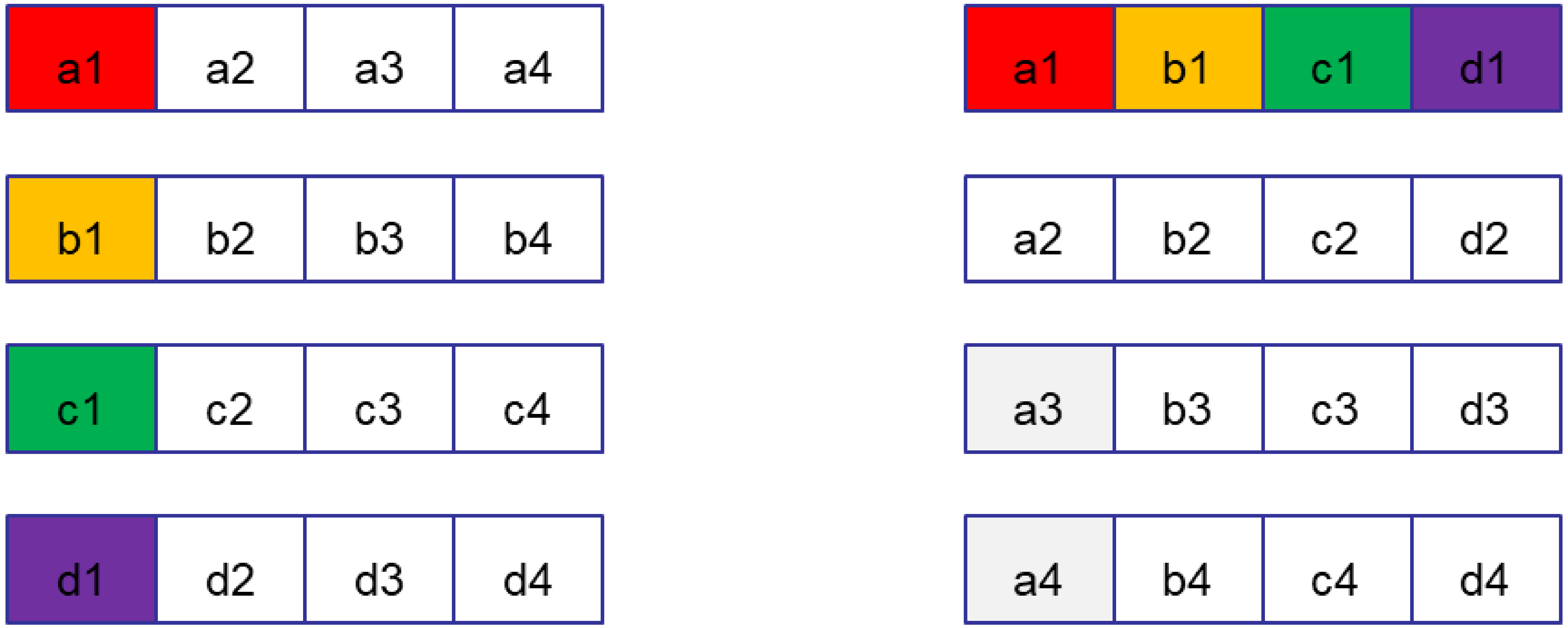


Gather + Broadcast

Alltoall

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

This function makes a redistribution of the content of each process in a way that each process know the buffer of all others.
It is a way to implement matrix data transposition.



This is an expensive call!
use only when needed.

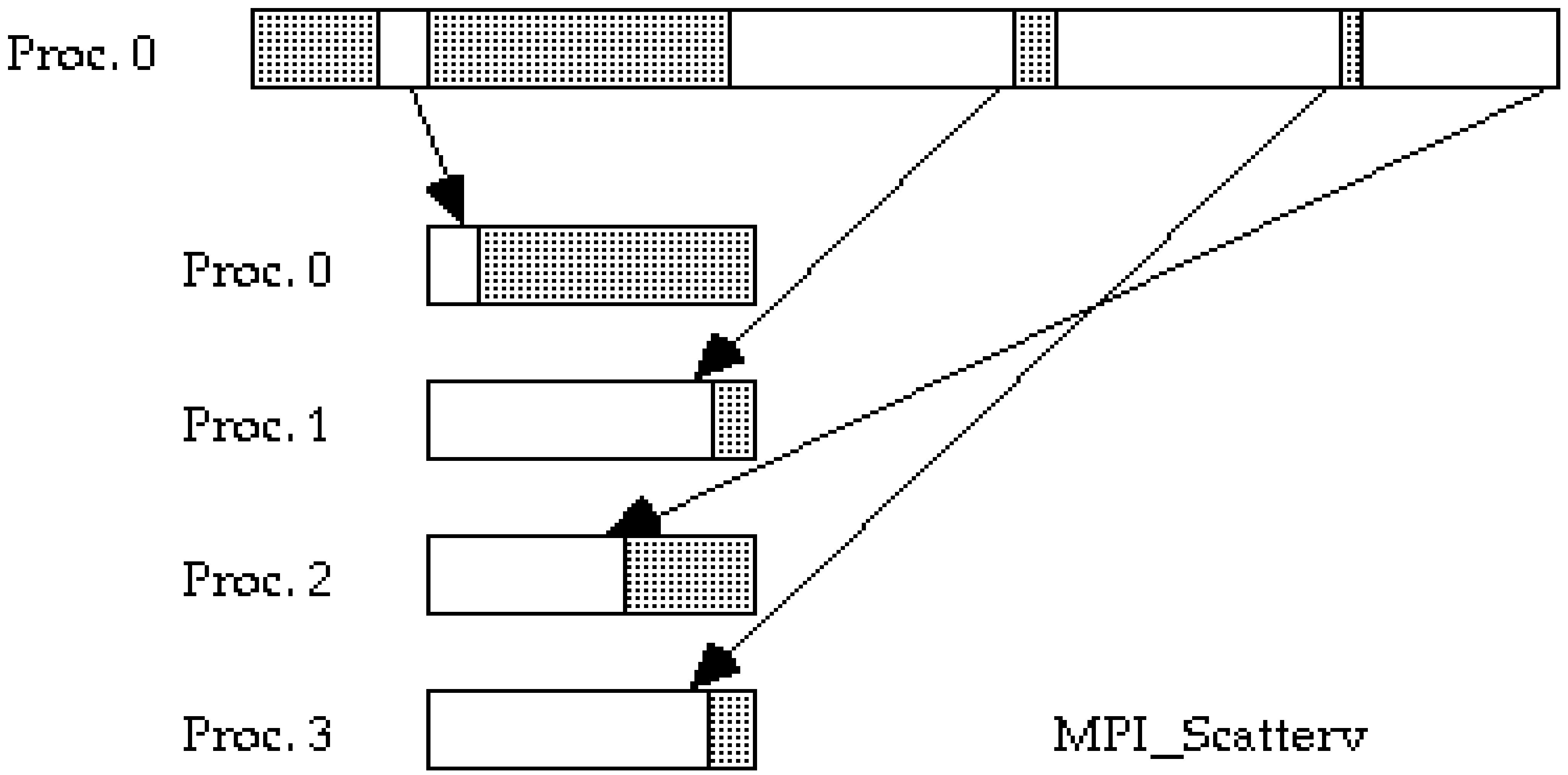
Gather + Broadcast

Other data distribution collectives

- MPI provides a variety of calls of distributing data amongst the processes, some quite complex.

- For example, it’s possible to define the length of arrays to be scattered or gathered.

```
MPI_ScatterV()
MPI_GatherV()
```



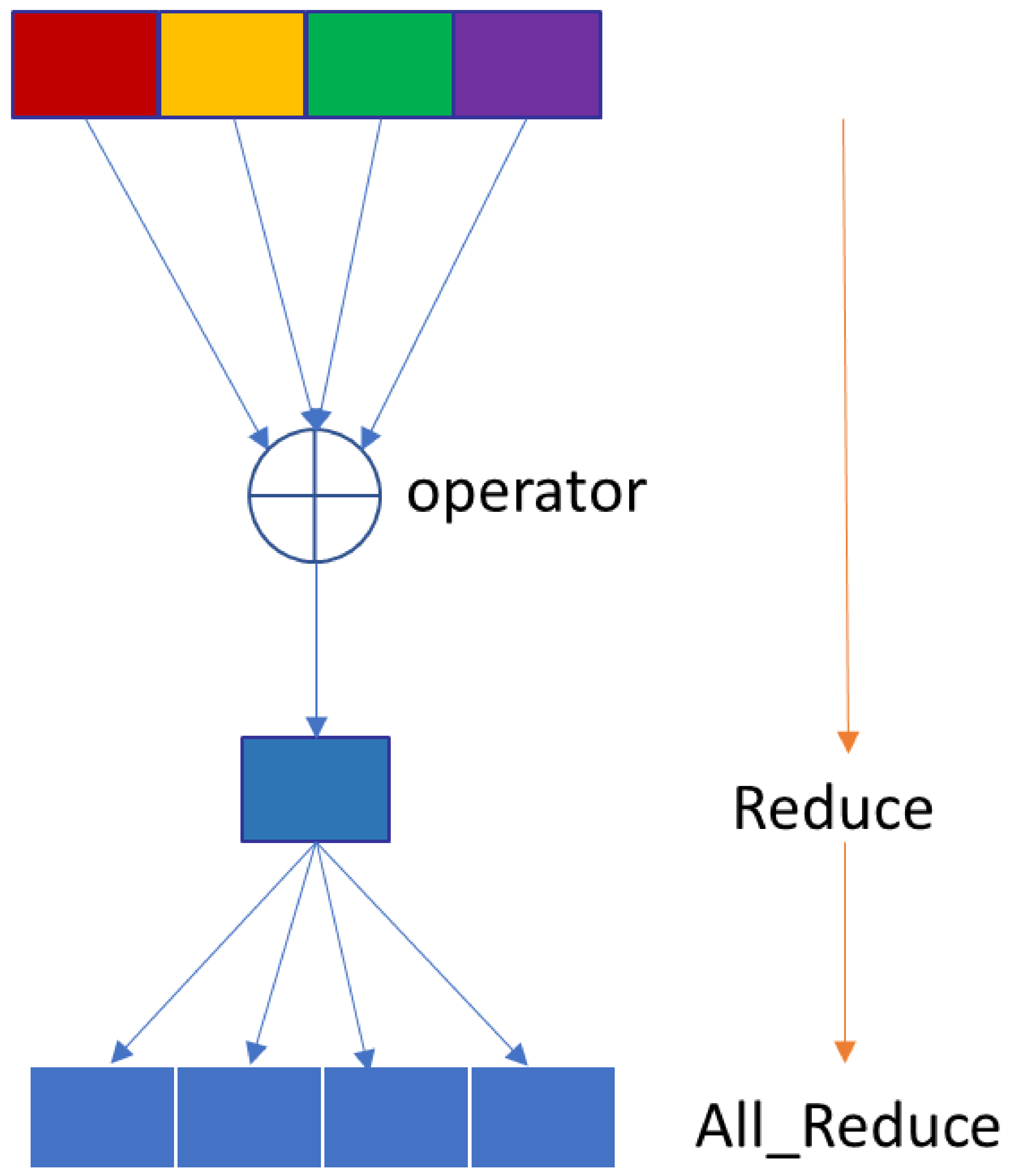
Reduction with collectives

Reduction

- A *reduction* takes values from different parallel processes and generates a single value
  (e.g. a sum, average, etc.).

- To avoid *race-conditions* special MPI calls are provided.

- In MPI the reduce collective function allows us to:
  - collect data from different processes;
  - reduce to a single value via some operation;
  - store the result on a single process (MPI_Reduce) or distribute the value to all processes (MPI_Allreduce).



Reduce and Allreduce

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

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

sendbuf	array that every task sends
Recvbuf	array for storing the result
count	number of elements to be operated
type	MPI type
op	MPI handle for the operation to process
root	rank that will receive the result (missing in Allreduce because the result is scattered among all processes)
comm	communicator involved

Reduction operations

Pre-defined Reduction operations

MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Reduction example

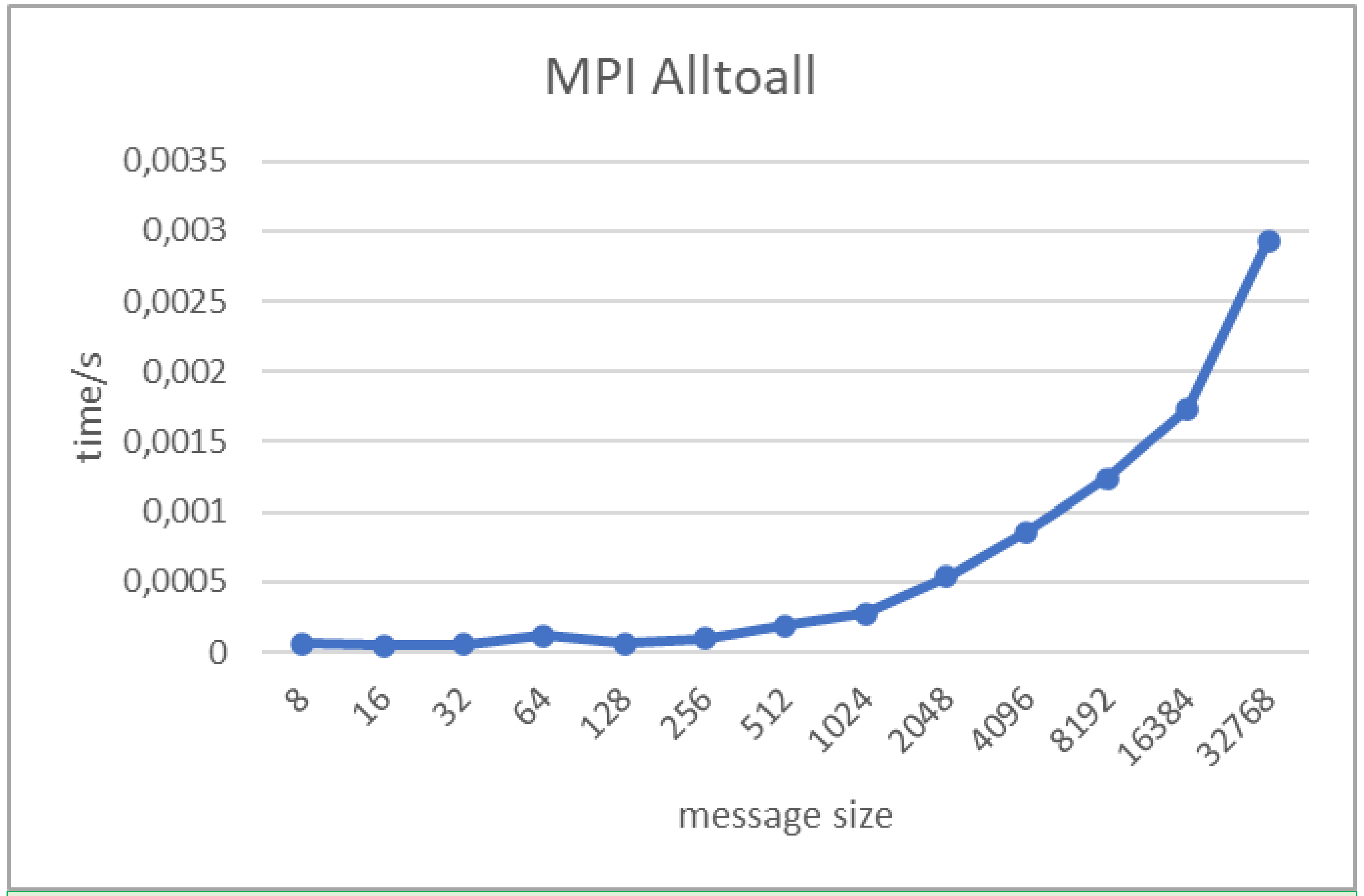
```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv) {
  int my_rank, procs, i;
  MPI_Status status;
  float a[2], res[2];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  a[0] = my_rank;
  a[1] = 2*my_rank;

  MPI_Reduce(a,res,2,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

  if (my_rank == 0)
    printf("res[0]=, %d, res[1]=, %d\n",res[0],res[1]);

  MPI_Finalize();
  return 0;
}
```

procs=4:
res[0]=6, res[1]=12

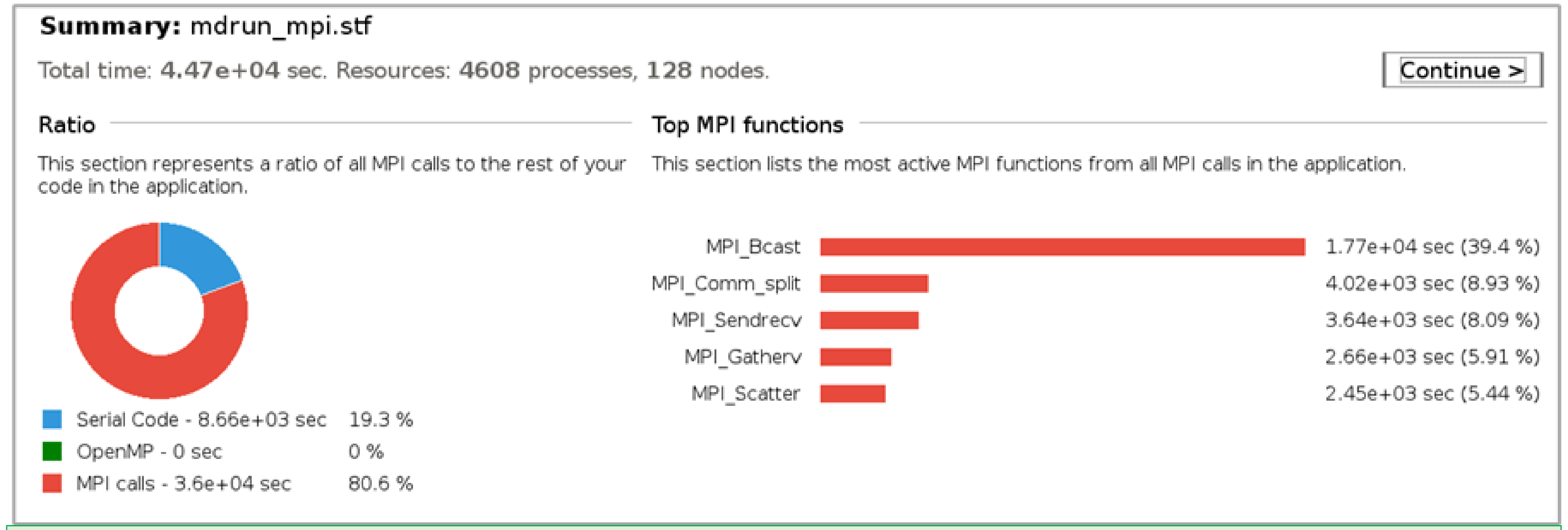
Collectives and performance

Collectives and performances

- MPI vendors work hard to optimise collectives for parallel hardware
- Despite this, parallel scaling is often dictated by MPI collectives
- All-to-all type communications can be particularly time consuming at high message sizes.



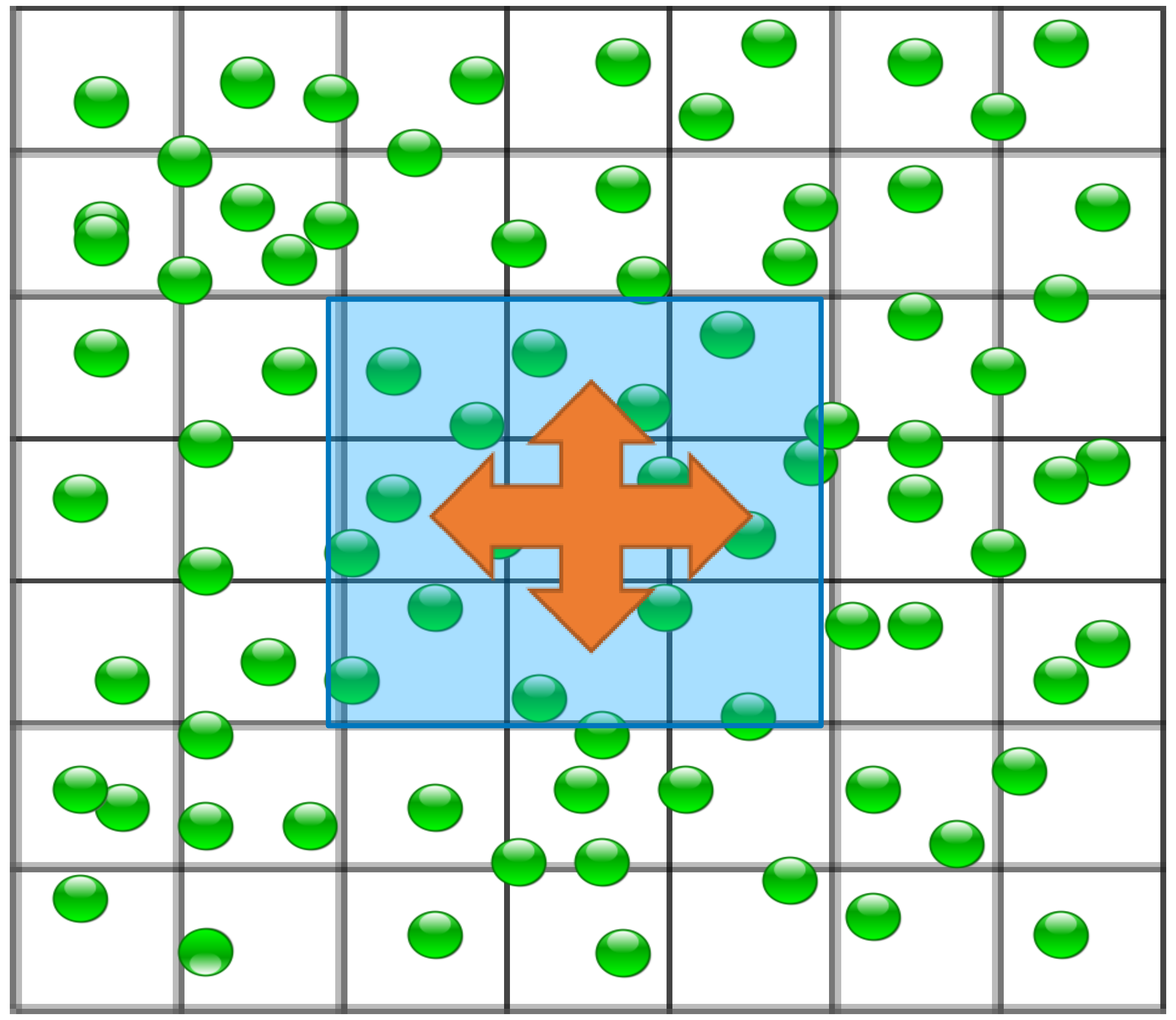
MPI alltoall variation with msg size over 16 nodes (IntelMPI)



Intel APS profiling of GROMACS over 128 nodes (red is MPI, blue is serial code)

Collective performance strategies

- ❑ Avoid unnecessary MPI_Barriers (often used in debugging).
- ❑ Consider non-blocking collectives.
- ❑ Define communicators using a subset of the available processes.
- ❑ Introduce algorithms which avoid many-process collectives (e.g. doman decomposition, meshes etc.)
- ❑ Use hybrid MPI/OpenMP to remove the number of MPI processes in collective calls.



Domain Decomposition - communications are localised, relying on point-to-point communications (e.g. MPI_SendRecv)

Summary

MPI Collective calls are convenient and efficient for communications or synchronizations groups of processes.

Don’t be tempted to write loops of point-to-point calls.

Be aware that at high parallelizations they will probably cause the program to stop parallel scaling.

Consider local-communication algorithms, communicators with fewer tasks or OpenMP to reduce collective costs.

Current MPI Standard to be found at: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
Credits to A. Emerson and many other colleagues from CINECA for the slides