HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 12

Fabian Wermelinger
Harvard University
CS205

Thursday, March 2nd 2023

LAST TIME

- Introduction to MPI and its history
- MPI communicators and groups
- Point-to-point and collective communication
- Synchronous and asynchronous communication
- Anatomy of a MPI message

TODAY

Main topic: ***Blocking MPI communication***

Details:

- MPI operations and MPI procedures
- Blocking point-to-point communication in MPI
- Communication modes
- Blocking collective MPI communication
- Examples of collective procedures

AGENDA CHECK:

- Reading assignment for Tuesday is about non-blocking point-to-point communication and the intro section for collective communication → MPI standard sections 3.7 and 6.1. The standard can be found in the Git repository: https://code.harvard.edu/CS205/main/blob/master/manuals/mpi40-report.pdf.

MPI OPERATIONS

- Sending and receiving messages is the basic MPI communication mechanism.
- An *MPI operation* is a sequence of steps performed by the MPI library to establish and enable data transfer and/or synchronization.
- This sequence consists of *four* stages:

  Initialization

  Hands over the argument list to the *operation* but not the content of the data buffers (if any).

  Starting

  Hands over the control of the data buffers (if any) to the associated operation. ***Note:*** *initiation* refers to the combination of the initialization and starting stages.

  Completion

  Returns control of the content of the data buffers and indicates that the output buffers and arguments (if any) have been updated. ***Note:*** an MPI operation is *complete* when the MPI procedure implementing the completion returns.

  Freeing

  Returns control of the rest of the argument list (e.g., the data buffer address and array arguments).

- MPI *operations* are available in one or more of these forms: ***blocking***, ***non-blocking*** and ***persistent***.

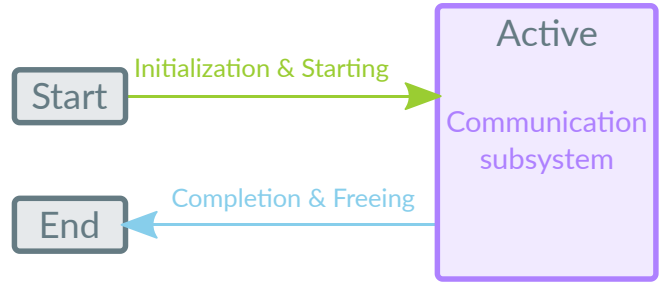
FORMS OF MPI OPERATIONS

Blocking operation

For a ***blocking operation***, all four stages are combined in a *single procedure call*.

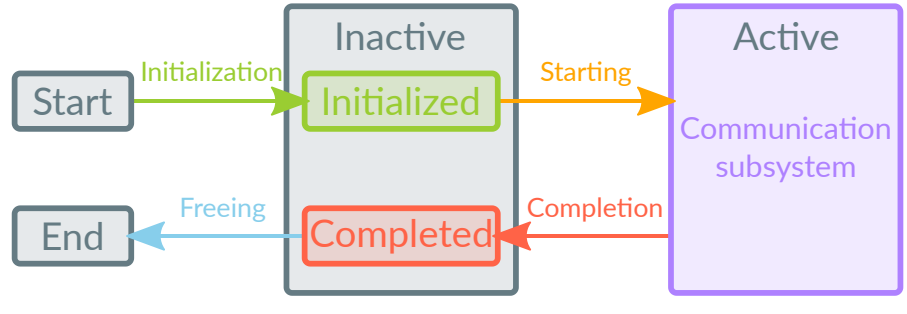
Non-blocking operation

For a ***non-blocking operation***, the initialization and starting stages are *combined into a single non-blocking* procedure call and the completion and freeing stages are combined into a *separate, single procedure call, which can be blocking or non-blocking*.



Persistent operation

For a ***persistent operation***, there is a separate procedure for each of the four stages. Each of these stages may be *blocking or non-blocking*. Persistent operations will not be part of CS205.



Additionally, an MPI operation can be ***collective*** or ***non-collective***

MPI PROCEDURES

- An MPI procedure (or routine) is a function from the API. MPI procedures all have the form `MPI_Xxxx` (or `PMPI_Xxxx` for the profiling interface).
- MPI procedures can be ***local*** or ***non-local***:

  Non-local procedure

  An MPI procedure is *non-local* if returning may require, during its execution, some specific *semantically related MPI procedure* to be called on another MPI process → *most completing procedures are non-local*.

  Local procedure

  An MPI procedure is *local* if it is not non-local → *most incomplete procedures are local*.
- An MPI operation is implemented as a set of one or more MPI procedures.
- There are MPI procedures that are not operation-related (e.g. `MPI_Comm_rank` or `MPI_Wtime`).
- A ***completing*** MPI procedure indicates that at least one associated operation has *finished its completion stage*, which implies that the user can *rely* on the content of the output data buffers and modify the content of input and output data buffers of such operation(s).
- An ***incomplete*** MPI procedure *is not* a completing procedure.
- A ***non-blocking*** MPI procedure is *incomplete and local*.
- A ***blocking*** MPI procedure is not *non-blocking*.
- ***See Sections 2.3 and 2.4 in the MPI 4.0 specification.***

EXAMPLES OF MPI PROCEDURES

Blocking procedures:

- Completing and non-local:
  - `MPI_Send`
  - `MPI_Recv`
  - `MPI_Bcast`
- Incomplete and non-local:
  - `MPI_Mprobe`
  - `MPI_Bcast_init`
- Completing and local:
  - `MPI_Bsend`
  - `MPI_Rsend`
  - `MPI_Mrecv`

Non-blocking procedures:

- Incomplete and local:
  - `MPI_Isend`
  - `MPI_Irecv`
  - `MPI_Ibcast`
  - `MPI_Improbe`

BLOCKING POINT-TO-POINT COMMUNICATION

- Standard (blocking) sending and receiving of messages in MPI is achieved with `MPI_Send` and `MPI_Recv`, respectively.
- They are both *non-local* procedures. A `MPI_Send` *requires* a matching `MPI_Recv` to complete and vice versa.

```
1 int MPI_Send(
2     const void *buf,        // message body
3     int count,              // message body
4     MPI_Datatype datatype, // message body
5     int dest,               // message envelope
6     int tag,                // message envelope
7     MPI_Comm comm           // message envelope
8 )
```

- A `MPI_Send` ***matches*** a `MPI_Recv` *only if* `dest`, `source`, `tag` and `comm` match.
- The return value is an error code. See `man MPI` for a full list of the codes. If the procedure is successful, it returns `MPI_SUCCESS`.

```
1 int MPI_Recv(
2     void *buf,              // message body
3     int count,              // message body
4     MPI_Datatype datatype, // message body
5     int source,             // message envelope
6     int tag,                // message envelope
7     MPI_Comm comm,          // message envelope
8     MPI_Status *status
9 )
```

- To receive messages with arbitrary tags use `MPI_ANY_TAG` and to receive from any source use `MPI_ANY_SOURCE`.
- If the `tag` or `source` is unknown, use the `status` object to query the meta data.
- Often the status is not needed, use `MPI_STATUS_IGNORE` in this case.

MPI TYPE MATCHING AND TYPE CONVERSION

MPI_Datatype	C type
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG_INT	long long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

See Table 3.2 in MPI 4.0 standard

- MPI supports *heterogeneous* environments. For example, you can communicate between Fortran and C codes.
- When you write MPI code, you must use these data types for the message data in the body.
- MPI will not convert types. This send/receive pair will not work:

```
1 MPI_Send(buf, 1, MPI_INT,   1, 99, MPI_COMM_WORLD);
2 MPI_Recv(buf, 1, MPI_FLOAT, 0, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- MPI will not convert types (see above) but it may convert the *representation* if necessary (endianness).
- MPI allows you to specify your own types as well, see Chapter 5 in the MPI 4.0 standard.
- `MPI_BYTE` *will not transform a type representation (its endianness).* The *binary* representation on the receiver is the same as the binary representation on the sender. Dangerous if you communicate between little and big endian machines! `MPI_BYTE` ≠ `MPI_CHAR`.

BLOCKING POINT-TO-POINT EXAMPLE

```cpp
#include <mpi.h>
#include <iostream>

// exchange rank IDs between 2 processes
int main(int argc, char* argv[])
{
    int rank, recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (0 == rank) {
        MPI_Send(&rank, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, 1, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        MPI_Send(&rank, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    std::cout << "Rank " << rank << " got ID " << recv << " from other rank\n";

    MPI_Finalize();
    return 0;
}
```

Single Program Multiple Data (SPMD):

Proc 0: `rank = 0`
Proc 1: `rank = 1`
→ Same program

Different data (different execution paths)

Can you explain what happens in the SPMD program on the left? Is this code bug free?

Output:

```
$ mpirun -n 2 ./mpi_p2p
Rank 0 got ID 1 from other rank
Rank 1 got ID 0 from other rank
```

Is the output on the left expected?

BLOCKING POINT-TO-POINT EXAMPLE

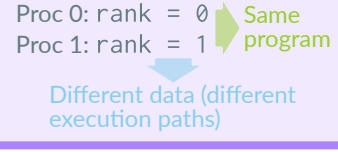
```cpp
#include <mpi.h>
#include <iostream>

// exchange rank IDs between 2 processes
int main(int argc, char* argv[])
{
    int rank, recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (0 == rank) {
        MPI_Send(&rank, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, 1, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        MPI_Send(&rank, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    std::cout << "Rank " << rank << " got ID " << recv << " from other rank\n";

    MPI_Finalize();
    return 0;
}
```

Single Program Multiple Data (SPMD):



Can you explain what happens in the SPMD program on the left? Is this code bug free?

→ The two highlighted lines are a bug in this code! MPI code written like this is not correct.

The (blocking) routines we use are non-local. The blocking `MPI_Send` will never be matched with a `MPI_Recv`. The execution will not make progress → ***deadlock***. Yet, our program ***did not*** deadlock. Why?

MPI COMMUNICATION MODES

- *We were just lucky that our code did not deadlock before*. Sending a larger message would not have worked (or if many more processes were involved than just two).
- The procedure involved made use of what is called ***standard mode***.
- In standard communication mode, MPI decides whether outgoing messages will be ***buffered***. If MPI decides to buffer the message, the `MPI_Send` call may ***complete*** before a matching `MPI_Recv` is invoked. *This is the reason the code did not deadlock in the previous slide → the default buffer size in any MPI implementation is very small!*
- ***Communication modes*** allow you to choose the *communication protocol*.

`MPI_Send` = ***standard mode***

- How standard mode behaves *is not defined in the standard* and therefore *implementation dependent*.
- Only ***guarantee***: when the procedure returns to caller, the send buffer is *safe to be used*.
- `MPI_Send` is a *non-local* procedure. Successful completion may depend on the occurrence of a matching receive.

MPI COMMUNICATION MODES

There are ***three*** additional communication modes (see Section 3.4 in the standard):

Buffered send: The `MPI_Bsend` uses a temporary *user provided* memory buffer to *copy* the message data to. Once the copy is complete, `MPI_Bsend` returns regardless whether a matching receive has been posted. This is a *local* procedure.

```
int MPI_Bsend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm)
```

Synchronous send: The `MPI_Ssend` does not make use of buffering. It can be started whether or not a matching receive was posted. It will only complete when the matching `MPI_Recv` is posted and the message delivered. This is a *non-local* procedure.

```
int MPI_Ssend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm)
```

Ready send: The `MPI_Rsend` may *only* be started if a matching `MPI_Recv` is already posted (error otherwise). It allows to reduce overhead in the communication protocol *but* ***you*** must ensure that the `MPI_Recv` is posted already.

```
int MPI_Rsend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm)
```

There is ***only one*** `MPI_Recv` that matches any of `MPI_Send`, `MPI_Bsend`, `MPI_Ssend` or `MPI_Rsend`.

BLOCKING POINT-TO-POINT EXAMPLE (REVISIT)

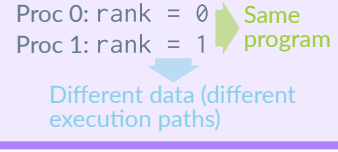
```cpp
#include <mpi.h>
#include <iostream>

// exchange rank IDs between 2 processes
int main(int argc, char* argv[])
{
    int rank, recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (0 == rank) {
        MPI_Send(&rank, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, 1, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        MPI_Recv(&recv, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&rank, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
    }
    std::cout << "Rank " << rank << " got ID " << recv << " from other rank\n";

    MPI_Finalize();
    return 0;
}
```

Single Program Multiple Data (SPMD):



The ***order*** with which you invoke MPI procedures matters.

The order of (blocking) MPI procedures matters! This is correct MPI code (deadlock free).

The `MPI_Send` used on rank 0 is matched with a `MPI_Recv` on rank 1 (both non-local procedures).
Rule: when you write MPI code, try to *post* the matching `MPI_Recv` of an MPI operation as early as possible.

SEMANTICS OF POINT-TO-POINT COMMUNICATION

A valid MPI implementation guarantees certain general properties of point-to-point communication:

Order

Messages are ***non-overtaking***. If a sender sends two messages in succession to the same destination and both match the same receive, then this operation cannot receive the second message if the first one is still pending. *If a process is multi-threaded then this order cannot be guaranteed.*

Progress

If a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions on the system.

Fairness

MPI makes *no guarantee* of fairness in the handling of communication. It is possible that messages sent from one process are continuously overtaken by messages sent from another process with the same destination. *It is your responsibility to prevent starvation in such situations.*

Resource limitation

Any pending communication operation consumes system resources. Errors may occur when lack of resources prevent the execution of an MPI call.

BLOCKING POINT-TO-POINT EXAMPLE

```cpp
#include <mpi.h>

int main(int argc, char* argv[])
{
    constexpr int n = 1000000;
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // allocate a memory buffer for MPI
    int bufsize = 0;
    MPI_Pack_size(n, MPI_INT, MPI_COMM_WORLD, &bufsize);
    bufsize += MPI_BSEND_OVERHEAD;
    char *mpibuf = new char[bufsize];
    MPI_Buffer_attach(mpibuf, bufsize);

    // send/receive buffers
    int *sendbuf = new int[n];
    int *recvbuf = new int[n];

    if (0 == rank) {
        MPI_Bsend(sendbuf, n, MPI_INT, 1, 98, MPI_COMM_WORLD); // local
        MPI_Recv(recvbuf, n, MPI_INT, 1, 99, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        MPI_Bsend(sendbuf, n, MPI_INT, 0, 99, MPI_COMM_WORLD); // local
        MPI_Recv(recvbuf, n, MPI_INT, 0, 98, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    // clean up
    MPI_Buffer_detach(mpibuf, &bufsize); // blocking
    delete[] mpibuf;
    delete[] sendbuf;
    delete[] recvbuf;

    MPI_Finalize();
    return 0;
}
```

MPI buffered send:

- Same example as before but *larger message size* and we explicitly allocate a buffer that MPI can use to buffer messages (for `MPI_Bsend`).
- `MPI_Bsend` is ***local***, therefore this code is correct → *no deadlock!*
- If the allocated buffer *is too small*, MPI will *not deadlock* but exit *with an error message*.
- If you change `MPI_Bsend` to `MPI_Send` (standard mode) or `MPI_Ssend` (synchronous mode) the code will deadlock.

BLOCKING POINT-TO-POINT EXAMPLE

```cpp
#include <cassert>
#include <iostream>
#include <mpi.h>
#include <string>

int main(int argc, char* argv[])
{
    assert(argc == 2); // pass some string as argument
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (0 == rank) {
        char message[1024];
        MPI_Status status;
        MPI_Recv(message, 1024, MPI_CHAR, 1, 99, MPI_COMM_WORLD, &status);

        // get the count of characters
        int count;
        MPI_Get_count(&status, MPI_CHAR, &count);
        assert(count < 1024);
        message[count + 1] = '\0'; // terminating null
        std::cout << "Rank " << rank << " received: " << message << " ("
                  << count << " characters)\n";
    } else {
        const std::string message(argv[1]);
        MPI_Send(
            message.c_str(), message.size(), MPI_CHAR, 0, 99, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Unknown message size on receiver:

- The `count` variable in `MPI_Recv` denotes the *maximum* number of elements in the receive buffer.
- You may receive less elements in a message.
- You can use the `MPI_Get_count` procedure to query the *actual* number of elements received.
- You need a `MPI_Status` object for the received message for this to work.
- See `man MPI_Recv`.

COMPUTING π WITH BLOCKING POINT-TO-POINT

```cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const size_t n = 100000000;

    // domain decomposition
    const size_t chunk_size = (n + size - 1) / size;
    const size_t my_start = rank * chunk_size;
    const size_t my_end = std::min((rank + 1) * chunk_size, n);

    // partial sum
    double sum = 0.0; // rank local sum (local to each process)
    for (size_t k = my_start; k < my_end; ++k) {
        sum += (1.0 - 2.0 * (k % 2)) / (2.0 * k + 1.0);
    }

    // reduction
    if (0 == rank) {
        double pi = sum;
        double recv; // receive buffer
        for (int r = 1; r < size; ++r) {
            MPI_Recv(&recv, 1, MPI_DOUBLE, r, 123,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            pi += recv;
        }
        std::cout << "pi = " << 4.0 * pi << '\n';
    } else {
        MPI_Send(&sum, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Recall problem 4 in HW1:

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

- Let us compute π with MPI this time.
- We need to divide the sum among MPI ranks instead of threads.
- There are no race conditions here!
- To reduce the partial sums, each rank needs to communicate its result with the root rank in the communicator (it could be any other rank as well).
- Note that the reduction phase involves as many steps as there are ranks. The way it is done here will scale badly if you have many ranks! It is a *collective* operation.

BLOCKING COLLECTIVE COMMUNICATION

What are they:

- Commonly used operations that involve ***all*** ranks in a communicator.
- When a blocking collective procedure returns, it is safe to use the communication buffers again (*completion*).
- You cannot assume that ranks in the communicator are *synchronized* when a blocking collective procedure returns. The *barrier* is the only collective procedure that guarantees synchronization upon return.
- Some of the collective calls have a single *originating* or *receiving* rank. This rank is called ***root***. Examples are `MPI_Bcast` (originating) or `MPI_Reduce` (receiving).
- Because communication is *collective*, *all* ranks must call the same collective primitive, otherwise ***deadlock***.

Overview:

All to One:

- `MPI_Reduce`
- `MPI_Gather`

One to All:

- `MPI_Bcast`
- `MPI_Scatter`

All to All:

- `MPI_Allgather`
- `MPI_Alltoall`
- `MPI_Allreduce`
- `MPI_Barrier`

Other:

- `MPI_Exscan`
- `MPI_Scan`

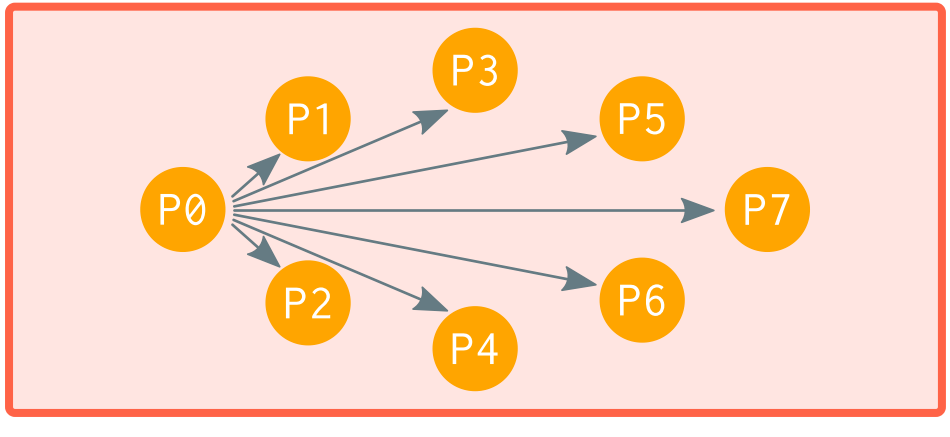
BLOCKING COLLECTIVE COMMUNICATION

Why do we have collective procedures in the API:

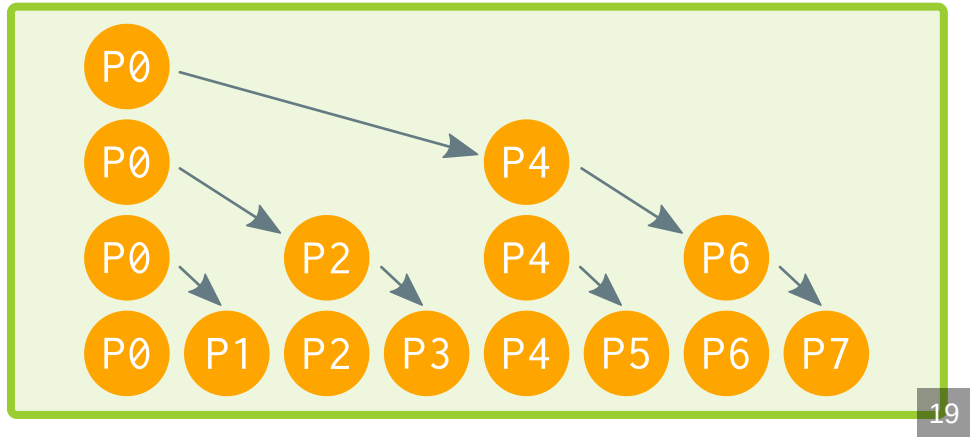
- We can implement any collective with point-to-point procedures.
- Collectives exist because we could implement them in *many different ways*. Some are *good* and others are *bad* from a *performance* perspective.
- MPI does not want you to implement your own (these are common operations). Instead, the MPI implementation *will pick the best automatically* depending on the *hardware* you are running on, the size of the communicator or due to algorithmic preference.

Example: `MPI_Bcast` (broadcast) — the root rank distributes data to all other ranks

Naive: the root rank must deal with 7 `MPI_Send` calls. This will scale badly if many ranks are involved.



Log-step: in a tree based algorithm, root must deal only with $\log_2(n)$ `MPI_Send` calls, where n is the total number of ranks in the communicator. *Here other ranks help out broadcasting the data.*



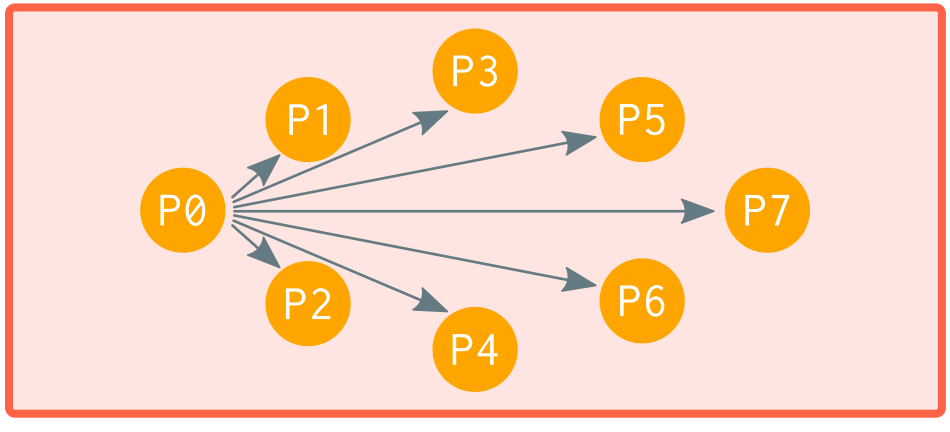
BLOCKING COLLECTIVE COMMUNICATION

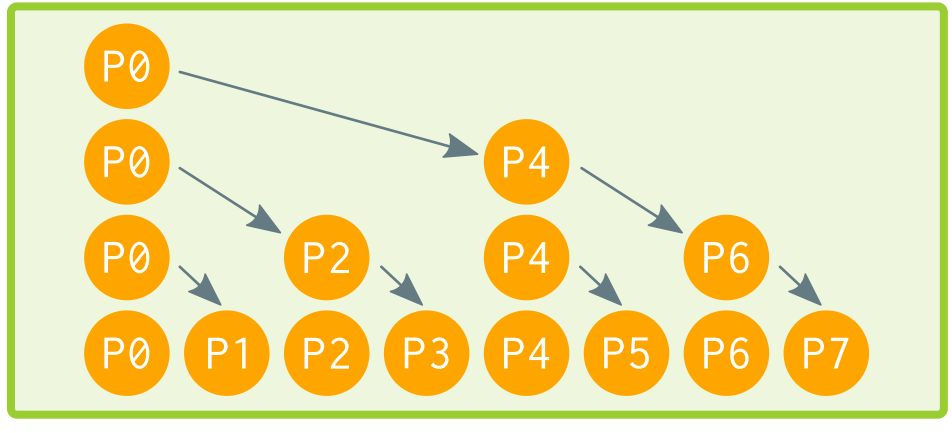
Why do we have collective procedures in the API:

Example: `MPI_Bcast` (broadcast) — the root rank distributes data to all other ranks

Naive: the root rank must deal with 7 `MPI_Send` calls. This will scale badly if many ranks are involved.

Log-step: in a tree based algorithm, root must deal only with $\log_2(n)$ `MPI_Send` calls, where n is the total number of ranks in the communicator. *Here other ranks help out broadcasting the data.*





Note:

- For ***8 ranks:*** 3 versus 7 `MPI_Send` calls *may not* feel like a big improvement.
- For ***1024 ranks:*** 10 versus 1023 `MPI_Send` calls is a ***100x*** improvement!

REDUCTION OPERATION

Blocking MPI reduction:

```
1 int MPI_Reduce(const void *sendbuf,
2                void *recvbuf,
3                int count,
4                MPI_Datatype datatype,
5                MPI_Op op,
6                int root,
7                MPI_Comm comm)
```

`sendbuf`	input data to reduce
`recvbuf`	output result (*root* only)
`count`	number of elements in `sendbuf`
`datatype`	type of data elements
`op`	reduction operator
`root`	root rank ID
`comm`	reduction communicator

- A reduction is a *collective* operation (***all*** ranks must participate).
- You can reduce the data *in-place*. In this case `recvbuf` acts as *input-output* and `sendbuf` must be set to `MPI_IN_PLACE`. See `man MPI_Reduce`.

Basic MPI reduction operators:

`MPI_Op`	Meaning
`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_LAND`	logical and
`MPI_BAND`	bit-wise and
`MPI_LOR`	logical or
`MPI_BOR`	bit-wise or
`MPI_LXOR`	logical exclusive or
`MPI_BXOR`	bit-wise exclusive or
`MPI_MAXLOC`	max value and location
`MPI_MINLOC`	min value and location

- The operation must be *associative*.
- The operation can be *commutative* but not required by the standard.
- You can define your own operator with `MPI_Op_create` (not covered in class).

COMPUTING π WITH BLOCKING REDUCTION

```cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const size_t n = 100000000;

    // domain decomposition
    const size_t chunk_size = (n + size - 1) / size;
    const size_t my_start = rank * chunk_size;
    const size_t my_end = std::min((rank + 1) * chunk_size, n);

    // partial sum
    double sum = 0.0; // rank local sum (local to each process)
    for (size_t k = my_start; k < my_end; ++k) {
        sum += (1.0 - 2.0 * (k % 2)) / (2.0 * k + 1.0);
    }

    // reduction
    double pi = 0.0;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (0 == rank) {
        std::cout << "pi = " << 4.0 * pi << '\n';
    }

    MPI_Finalize();
    return 0;
}
```

Recall problem 4 in HW1:

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

- Less code and more clear!
- MPI takes care of the reduction for us.
- In this example the result is reduced to rank 0.
- *All* ranks must call `MPI_Reduce` otherwise the code will deadlock.

COMPUTING π WITH BLOCKING IN-PLACE REDUCTION

```cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const size_t n = 100000000;

    // domain decomposition
    const size_t chunk_size = (n + size - 1) / size;
    const size_t my_start = rank * chunk_size;
    const size_t my_end = std::min((rank + 1) * chunk_size, n);

    // partial sum
    double sum = 0.0; // rank local sum (local to each process)
    for (size_t k = my_start; k < my_end; ++k) {
        sum += (1.0 - 2.0 * (k % 2)) / (2.0 * k + 1.0);
    }

    // reduction
    MPI_Reduce((0 == rank) ? MPI_IN_PLACE : &sum,
               &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (0 == rank) {
        std::cout << "pi = " << 4.0 * sum << '\n';
    }

    MPI_Finalize();
    return 0;
}
```

Recall problem 4 in HW1:

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

- You could save allocating another `double` by using the `MPI_IN_PLACE` handle.
- This is not always clearer.
- In this example, allocating one more `double` is not a big deal.

FIND MAX/MIN LOCATION WITH REDUCTION

```cpp
#include <cstdlib>
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    srand48(rank);
    const double value = drand48();
    std::cout << "Rank " << rank << ": value = " << value << '\n';

    // packed value and rank ID
    struct DoubleInt {
        double val;
        int rank;
    } send, recv;
    send.val = value;
    send.rank = rank;

    MPI_Reduce(&send, &recv, 1, MPI_DOUBLE_INT,
        MPI_MINLOC, 0, MPI_COMM_WORLD);
    if (0 == rank) {
        std::cout << "Minimum: value = " << recv.val;
                  << " on rank " << recv.rank << '\n';
    }

    MPI_Finalize();
    return 0;
}
```

- The `MPI_MAXLOC` and `MPI_MINLOC` reduce operators are used to determine a global maximum and/or minimum of a value including its location (on which rank).
- For this operation, MPI defines special data types that encode the value type as well as the index type (`int`). These are
  - `MPI_FLOAT_INT`
  - `MPI_DOUBLE_INT`
  - `MPI_LONG_INT`
  - `MPI_2INT`
  - `MPI_SHORT_INT`
  - `MPI_LONG_DOUBLE_INT`

  See Section 6.9.3 in the MPI 4.0 standard.

Output:

```
$ mpirun -n 4 ./mpi_minloc
Rank 1: value = 0.0416303
Rank 2: value = 0.912433
Rank 3: value = 0.783235
Rank 0: value = 0.170828
Minimum: value = 0.0416303 on rank 1
```

MPI BARRIER

```c
#include <mpi.h>

int main(int argc, char *argv[])
{
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // some code

    if (0 == rank) {
        // all ranks must call this procedure
        MPI_Barrier(MPI_COMM_WORLD);
    }
    // Deadlock: other ranks do not call
    //           MPI_Barrier

    // some other code

    MPI_Finalize();
    return 0;
}
```

- A barrier *synchronizes* all processes in a communicator
- The procedure completes once *all processes have executed the routine.*
- Your code will deadlock if not all ranks in a group participate.
- The barrier on the left must not be inside the `if`-block for this example!

MPI BROADCAST

A broadcast sends data from the `root` rank (not necessarily rank 0) to all ranks in the group, including itself.

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

A naive implementation to broadcast a message (inefficient, don't do this)

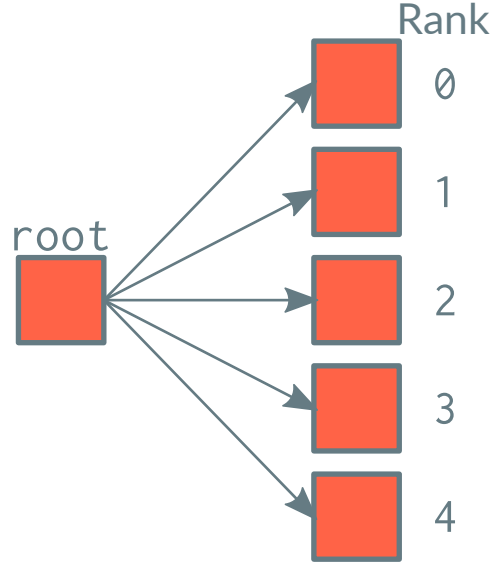
```c
#include <mpi.h>

void bcast(void *buffer, int count, MPI_Datatype datatype,
           int root, MPI_Comm comm)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == root) {
        for (int i = 0; i < size; ++i) {
            if (i == rank) {
                continue
            }
            MPI_Send(buffer, count, datatype, i, 123, comm);
        }
    } else {
        MPI_Recv(buffer, count, datatype, root, 123, comm,
            MPI_STATUS_IGNORE);
    }
}
```

MPI internally uses a tree structure to achieve a more efficient implementation.

Example visualization for an `MPI_Bcast`:



See Section 6.4 in the MPI 4.0 standard.

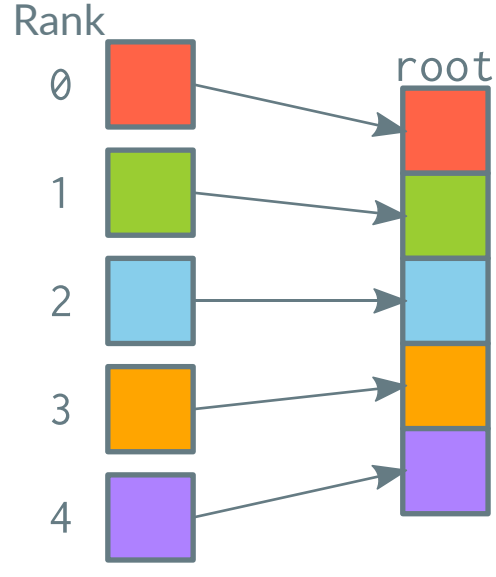
MPI GATHER

In a gather operation, each rank sends data to the root rank. The root process receives and stores the messages in *rank order*.

```
1 int MPI_Gather(const void *sendbuf,
2                int sendcount,
3                MPI_Datatype sendtype,
4                void *recvbuf,
5                int recvcount,
6                MPI_Datatype recvtype,
7                int root,
8                MPI_Comm comm)
```

- The `recvbuf`, `recvcount` and `recvtype` arguments are only significant for the `root` process.
- The `recvcount` argument denotes the number of elements to be received from any other process.
- If there are n ranks, then the total number of elements in the receive buffer is given by `n * recvcount`. The n messages are concatenated in *rank order*.

Example visualization for an `MPI_Gather`:



- See Section 6.5 in the MPI 4.0 standard.
- There is a more flexible version that allows to receive a different number of elements from each process. See `man MPI_Gatherv`

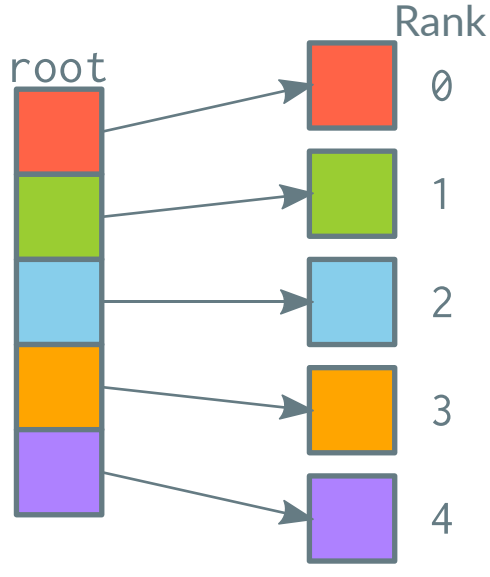
MPI SCATTER

The scatter operation is the *inverse* of the gather operation.

```
1 int MPI_Scatter(const void *sendbuf,
2                 int sendcount,
3                 MPI_Datatype sendtype,
4                 void *recvbuf,
5                 int recvcount,
6                 MPI_Datatype recvtype,
7                 int root,
8                 MPI_Comm comm)
```

- The `sendbuf`, `sendcount` and `sendtype` arguments are only significant for the `root` process.
- The `sendcount` argument denotes the number of elements to be sent to any other process.
- If there are n ranks, then the total number of elements in the send buffer is given by `n * sendcount`. The i-th process with rank `i` receives `sendcount` elements starting from position `i * sendcount` in the `sendbuf` buffer on the `root` rank.

Example visualization for an `MPI_Scatter`:



- See Section 6.6 in the MPI 4.0 standard.
- There is a more flexible version that allows to send a different number of elements to each process. See `man MPI_Scatterv`

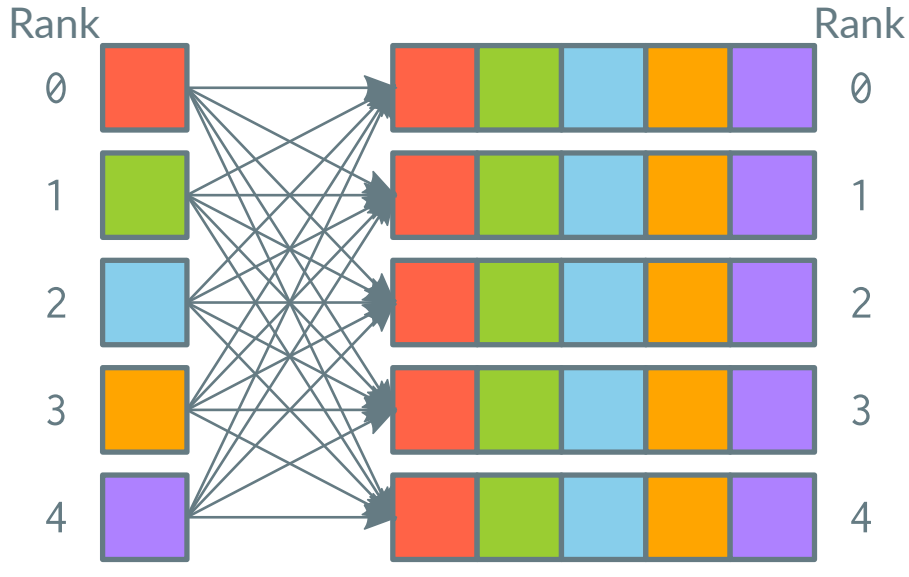
MPI GATHER TO ALL

The same as the gather operation but *all* processes receive the results instead of just `root`.

```
1 int MPI_Allgather(const void *sendbuf,
2                   int sendcount,
3                   MPI_Datatype sendtype,
4                   void *recvbuf,
5                   int recvcount,
6                   MPI_Datatype recvtype,
7                   MPI_Comm comm)
```

- The `recvcount` argument denotes the number of elements to be received from any other process.
- If there are n ranks, then the total number of elements in the receive buffer is given by `n * recvcount`. The n messages are concatenated in *rank order*.

Example visualization for an `MPI_Allgather`:



See Section 6.7 in the MPI 4.0 standard.

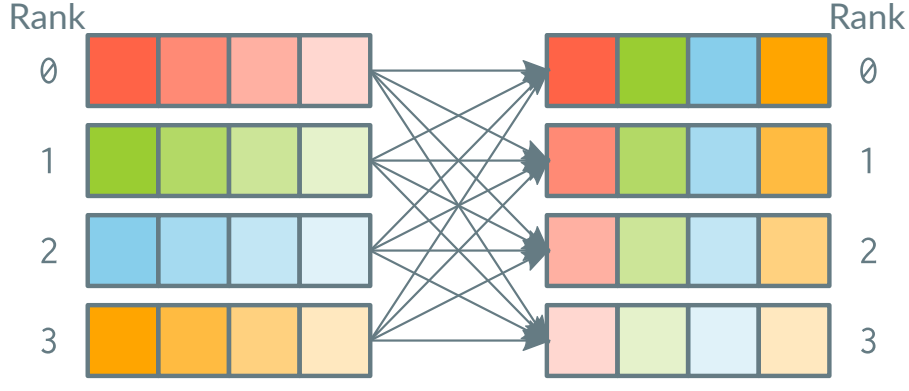
MPI ALL-TO-ALL

The all-to-all operation is an extension of the gather to all operation to the case where each process sends distinct data to each of the receivers.

```
1 int MPI_Alltoall(const void *sendbuf,
2                   int sendcount,
3                   MPI_Datatype sendtype,
4                   void *recvbuf,
5                   int recvcount,
6                   MPI_Datatype recvtype,
7                   MPI_Comm comm)
```

- Each process performs $2n$ independent point-to-point communications (including itself).
- Each process breaks up its local `sendbuf` into n blocks (each with `sendcount` elements). The `recvbuf` is similarly divided.
- Process j sends the k-th block to process k which places the data in the j-th block of its local `recvbuf` buffer.
- The amount of data sent must be equal to the amount of data received (pairwise between every pair of processes).

Example visualization for an `MPI_Alltoall`:



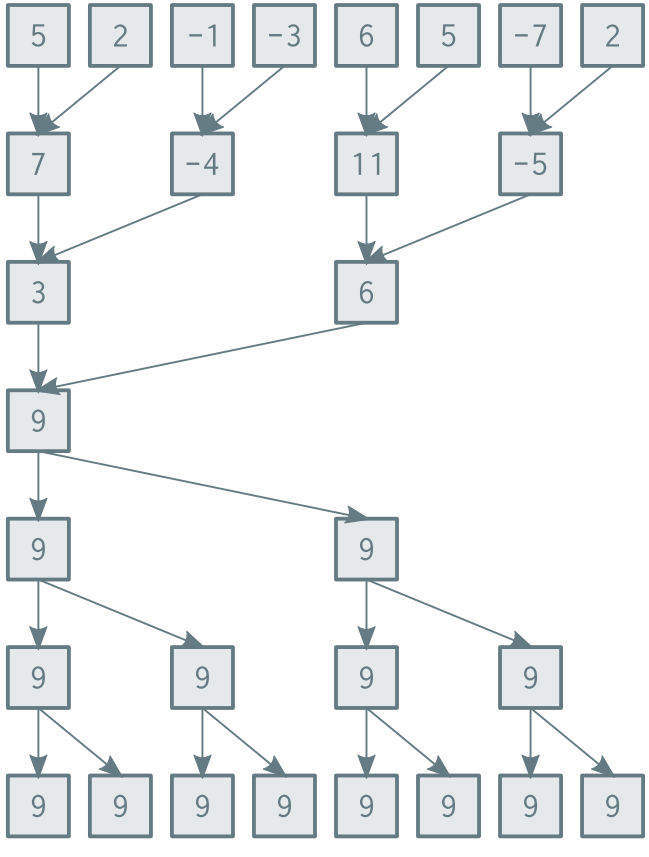
- The result is a ***transpose***.
- The `MPI_Alltoall` is heavily used in the parallel implementation of the Fast-Fourier-Transform (FFTW library).
- This operation supports the `MPI_IN_PLACE` handle for in-place memory operations.
- See Section 6.8 in the MPI 4.0 standard.

MPI ALL-REDUCE

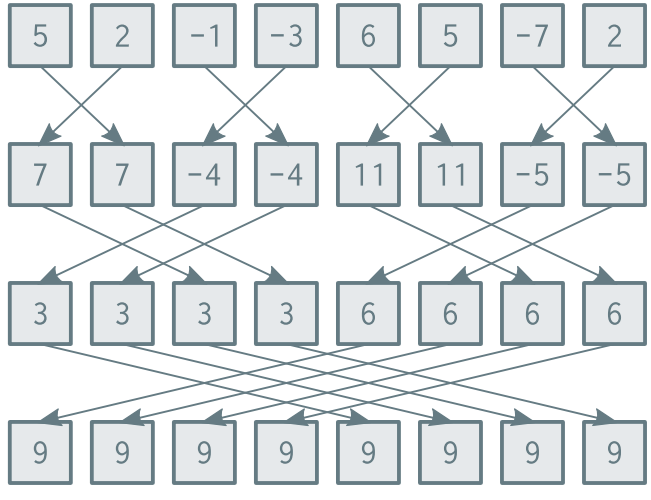
Same as `MPI_Reduce` but result is broadcast to all processes.

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

Naive tree implementation:



Optimized implementation:



MPI INCLUSIVE SCAN (PARTIAL REDUCTION)

For the *inclusive scan* operation, the result on rank i is the partial reduction with data from ranks $k \leq i$ (*inclusive*).

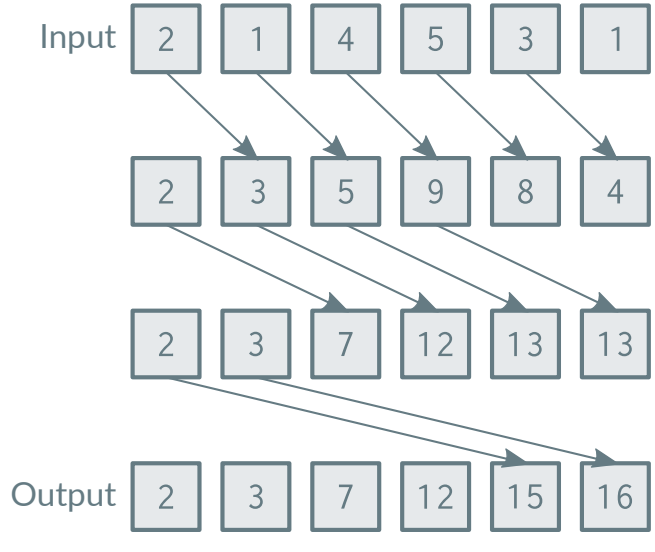
```
1 int MPI_Scan(const void *sendbuf,
2               void *recvbuf,
3               int count,
4               MPI_Datatype datatype,
5               MPI_Op op,
6               MPI_Comm comm)
```

- The `MPI_Op` reduction operator can be any of the ones used for `MPI_Reduce`.
- You can use the `MPI_IN_PLACE` handle for the `sendbuf` argument. In this case the input is taken from the `recvbuf` argument and output replaces this value.

Example:

$$y_i = \sum_{k=0}^{i} x_k$$

Rank	0	1	2	3	4	5
Input	2	1	4	5	3	1
Output	2	3	7	12	15	16



MPI EXCLUSIVE SCAN (PARTIAL REDUCTION)

The *exclusive scan* operation is similar to the inclusive scan, except that the partial reduction does not include the data on rank i, that is $k < i$ (*exclusive*).

```
1 int MPI_Exscan(const void *sendbuf,
2                 void *recvbuf,
3                 int count,
4                 MPI_Datatype datatype,
5                 MPI_Op op,
6                 MPI_Comm comm)
```

- The `MPI_Op` reduction operator can be any of the ones used for `MPI_Reduce`.
- You can use the `MPI_IN_PLACE` handle for the `sendbuf` argument. In this case the input is taken from the `recvbuf` argument and output replaces this value.

Example:

$$y_i = \sum_{k=0}^{i-1} x_k$$

Rank	0	1	2	3	4	5
Input	2	1	4	5	3	1
Output	0	2	3	7	12	15

- An example usage for an `MPI_Exscan` is to compute the file offset when you write compressed data into a file using parallel I/O.
- The size of the compressed buffer is only known on the process that performed the data compression.

RECAP

- The fundamental form of communication in MPI is point-to-point communication.
- It is important to understand the difference between local and non-local procedures and the different communication modes in MPI.
- Collective primitives are implemented with point-to-point communication primitives.
- Collective procedures involve *all* ranks in a group.
- MPI provides optimized algorithms for collective communication.
- Collective primitives implement commonly used algorithms. You ***should not*** implement these algorithms yourself.

Further reading:

- Sections 2.3-2.4, 3.1-3.6 and 6.1-6.11 in *MPI 4.0 Standard*, MPI Forum, 2021 (pdf in `git` repo as well)
- Section 3.1, 3.2 and 3.4 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Chapter 9 in Hager and Wellein, *"Introduction to High Performance Computing for Scientists and Engineers"*, CRC Press, 2011