HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 13

Fabian Wermelinger
Harvard University
CS205

Tuesday, March 7th 2023

LAST TIME

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

TODAY

Main topic: ***Non-blocking MPI communication***

Details:

- A few more topics on blocking procedures:
  - Probing messages
  - Receiving matched messages (MPI 3.0 or higher only)
  - Shift operations (with virtual topologies)
- MPI message protocols
- Non-blocking semantics
- Example for 1D diffusion equation
- Non-blocking collectives

AGENDA CHECK:

- Quiz 3 this Thursday covers lectures 9-13 in schedule. 15 minutes quiz with 12 multiple choice/true/false questions (2 extra simple questions added to help you smooth out possible mistakes in harder questions).
- Project M3: in-class presentations for your teams project topics. Should be more detailed than high-level overview. Draft for parallelization plan should be presented and it should be argued why parallelization of the problem is justified → https://harvard-iacs.github.io/2023-CS205/pages/project.html#M3

UNKNOWN MESSAGE SIZE

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

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

    std::vector<double> msg;
    if (0 == rank) {
        msg.resize(rand() % 2048, 1.0);
        const int size = msg.size();
        MPI_Send(&size, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
        MPI_Send(msg.data(), msg.size(), MPI_DOUBLE,
                 1, 321, MPI_COMM_WORLD);
    } else if (1 == rank) {
        int size;
        MPI_Recv(&size, 1, MPI_INT,
                 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        msg.resize(size);
        MPI_Recv(msg.data(), msg.size(), MPI_DOUBLE,
                 0, 321, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << "Rank " << rank << " received " << size
                  << " double elements\n";
    }

    MPI_Finalize();
    return 0;
}
```

- What if the message size is unknown and you must know the size of an incoming message ***before*** you have received the message.
- ***Note:*** in the string example in the previous lecture *we have assumed a fixed receive buffer size of 1024 characters.*
- Sending two messages is not a good solution because of the additional overhead (especially since the first message is very small→latency dominated).

PROBING AN INCOMING MESSAGE

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

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

    std::vector<double> msg;
    if (0 == rank) {
        msg.resize(rand() % 2048, 1.0);
        MPI_Send(msg.data(), msg.size(), MPI_DOUBLE,
                 1, 123, MPI_COMM_WORLD);
    } else if (1 == rank) {
        int size;
        MPI_Status status;
        MPI_Probe(0, 123, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_DOUBLE, &size);
        msg.resize(size);
        MPI_Recv(msg.data(), msg.size(), MPI_DOUBLE,
                 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << "Rank " << rank << " received " << size
                  << " double elements\n";
    }

    MPI_Finalize();
    return 0;
}
```

- The `MPI_Probe` operation allows to check for incoming messages *without retrieving them*.
- The populated `MPI_Status` can be used to obtain the count of messages elements using `MPI_Get_count` afterwards.
- Probing a message to get the message status is more efficient than actually sending (small) messages processes that need to know the incoming message size.
- See Section 3.8 in the MPI 4.0 standard.

RECEIVING MESSAGES OF A MATCHING PROBE

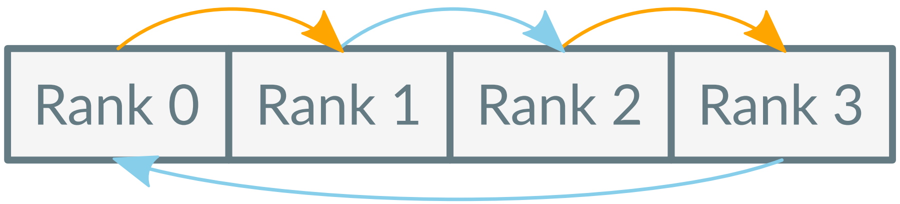
```cpp
#include <cstdlib>
#include <iostream>
#include <mpi.h>
#include <vector>

int main(int argc, char *argv[])
{
    int rank, prov;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &prov);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    srand(rank);

    std::vector<double> msg;
    if (0 == rank) {
        msg.resize(rand() % 2048, 1.0);
        MPI_Send(msg.data(), msg.size(), MPI_DOUBLE,
                 1, 123, MPI_COMM_WORLD);
    } else if (1 == rank) {
        // the following code could be multi-threaded
        int size;
        MPI_Status status;
        MPI_Message handle;
        MPI_Mprobe(0, 123, MPI_COMM_WORLD, &handle, &status);
        MPI_Get_count(&status, MPI_DOUBLE, &size);
        msg.resize(size);
        MPI_Mrecv(msg.data(), msg.size(), MPI_DOUBLE,
                  &handle, MPI_STATUS_IGNORE);
        std::cout << "Rank " << rank
                  << " (matching) received " << size
                  << " double elements\n";
    }

    MPI_Finalize();
    return 0;
```

- On a multi-threaded process it is difficult to process messages with `MPI_Probe` because the *list of incoming messages is global to all threads on the process (shared memory).*
- One thread could receive the message while another thread probes the same message for further processing.
- A `MPI_Mprobe` (*matching probe*) returns a *message handle* which allows to *exclusively receive the matching message* (no other probe or receive operation on another thread may fetch this message) → *MPI 3.0 or higher only.*
- You *must* use a matching receive operation `MPI_Mrecv` to get the message with the matching *message handle.*

SHIFT OPERATIONS ACROSS CHAIN OF PROCESSES

- A common communication pattern is to send a message to one rank and receive a message from another (the source and destination could also be the same).
- If blocking semantics are used, then the send and receive operations must be ***ordered correctly*** to *prevent cyclic dependencies* that may lead to deadlock.
- Such shift operations are often needed on a lattice. ***Example:*** exchanging messages on a Cartesian topology of MPI processes (e.g. in HW3).
- For virtual topologies, see Chapter 8 in the MPI 4.0 standard (useful for HW3).
- See lecture codes: `mpi_cyclic_shift`

- Even ranks first send to odd ranks and odd ranks receive first.
- Once completed, the odd ranks send to even ranks.




```cpp
// source and destination (periodic boundaries)
const int left = (rank + size - 1) % size;
const int right = (rank + size + 1) % size;

std::vector<double> send(100, rank), recv(100);
if (0 == rank % 2) { // even ranks send first
    MPI_Send(send.data(), send.size(), MPI_DOUBLE,
             right, 123, MPI_COMM_WORLD);
    MPI_Recv(recv.data(), recv.size(), MPI_DOUBLE,
             left, 321, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
    MPI_Recv(recv.data(), recv.size(), MPI_DOUBLE,
             left, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(send.data(), send.size(), MPI_DOUBLE,
             right, 321, MPI_COMM_WORLD);
}
```

SHIFT OPERATIONS ACROSS CHAIN OF PROCESSES

- MPI provides a *send-receive* operation that *combines* sending and receiving to/from processes (the two processes may be the same).
- MPI will take care of the cyclic dependencies and therefore avoid deadlock.
- This operation is often used on a Cartesian topology of processes together with `MPI_Cart_shift` to determine the source and destination ranks.

```
int MPI_Sendrecv(const void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 int dest,
                 int sendtag,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int source,
                 int recvtag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

MPI send-receive operation:

```
// source and destination (periodic boundaries)
const int left = (rank + size - 1) % size;
const int right = (rank + size + 1) % size;

std::vector<double> send(100, rank), recv(100);
MPI_Sendrecv(send.data(), send.size(), MPI_DOUBLE, right, 123,
             recv.data(), recv.size(), MPI_DOUBLE, left,  123,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Manual send and receive pairs:

```
// source and destination (periodic boundaries)
const int left = (rank + size - 1) % size;
const int right = (rank + size + 1) % size;

std::vector<double> send(100, rank), recv(100);
if (0 == rank % 2) { // even ranks send first
    MPI_Send(send.data(), send.size(), MPI_DOUBLE,
             right, 123, MPI_COMM_WORLD);
    MPI_Recv(recv.data(), recv.size(), MPI_DOUBLE,
             left, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
    MPI_Recv(recv.data(), recv.size(), MPI_DOUBLE,
             left, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(send.data(), send.size(), MPI_DOUBLE,
             right, 123, MPI_COMM_WORLD);
}
```

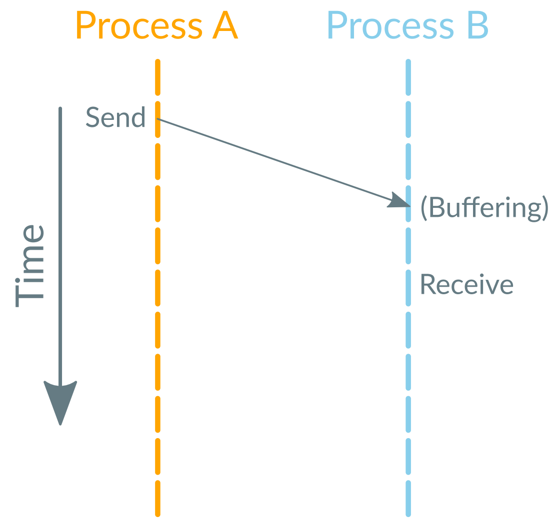
Note: there is also a `MPI_Sendrecv_replace` for which the send and receive buffer is the same (MPI will allocate internal buffer space for the operation).

MESSAGE PASSING PROTOCOLS

- Nearly all MPI implementations employ a *two-level protocol* for implementing the peer communication procedures.
- The idea is to optimize short messages for ***latency*** and large messages for ***bandwidth***.
- If the message is short, MPI attempts to send the message ***eagerly***. This includes sending both, the envelope and the message body, at *the same time*. This protocol requires a pre-allocated and pinned buffer on both, the sender and receiver side. Copying the messages into the pre-allocated buffers incurs additional copy overhead.
- The large message protocol involves a ***rendezvous*** where the two processes perform a *handshake* first followed by sending the message data. This protocol registers the involved memory buffers upon successful handshake which generates additional *overhead* but allows for a zero-copy data transfer. The larger overhead is amortized by the large data transfer.
- Most MPI implementations are forced to use a *rendezvous* protocol for large messages due to the operating system and/or network limitations.
- Sending large messages *eagerly* requires hardware support. HPC nodes have a network interface controller (NIC) with support for **R**emote **D**irect **M**emory **A**ccess (RDMA) that allows to *bypass the operating system*. Sending large messages eagerly can still cause memory exhaustion on the receiver side and increases copy overhead (some zero-copy optimizations exist however).

EAGER PROTOCOL

- An *asynchronous* protocol that allows a send operation to complete *without* acknowledgement from the receiver side.
- The sender assumes that the receiver can store the message.
- It is the receivers responsibility to provide a storage buffer, *especially when the matching receive operation has not been posted yet.*
- Some MPI implementations may provide a small internal buffer to implement the eager protocol (***Recall:*** this was the reason the two `MPI_Send` calls did not deadlock in the previous lecture).
- The eager protocol is generally used for smaller messages (few kB) and may be limited by the number of MPI ranks as well.



Pros:

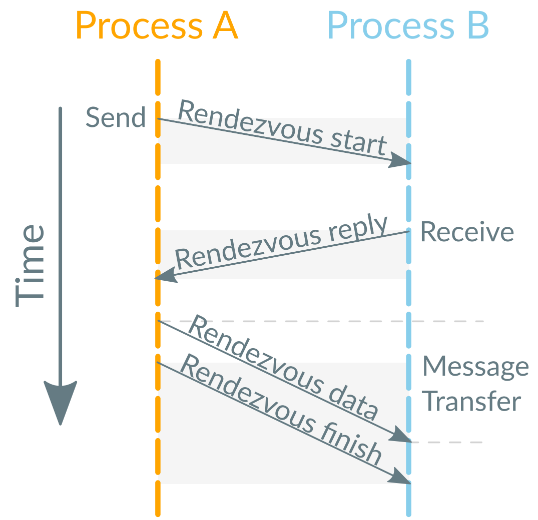
- Reduced synchronization delays (less overhead → optimized for latency).
- Allows to bypass OS by using additional hardware that supports RDMA (NIC).

Cons:

- Not scalable in general (additional pinned buffers increase memory footprint).
- Additional copy overhead is expensive if messages are large.

RENDEZVOUS PROTOCOL

- No assumptions are made that there is buffer space to store the message.
- Requires handshaking between sender and receiver:
    1. Sender sends message envelope to receiver.
    2. Envelope is received and stored by the receiver.
    3. Receiver replies to sender by posting a receive operation.
    4. Sender starts sending the message body.
    5. When done, the sender sends a receipt packet to the receiver indicating completion of the transfer.



MPI implementations often use:

- Eager for *small messages* (dominated by latency)
- Rendezvous for *large messages* (dominated by bandwidth)

Rule: *post receive operations as early as possible.*

Pros:

- Scalable compared to eager protocol → optimizes for bandwidth.
- Better memory management (less memory footprint).

Cons:

- Inherent synchronization (more overhead).
- More complex implementation details.

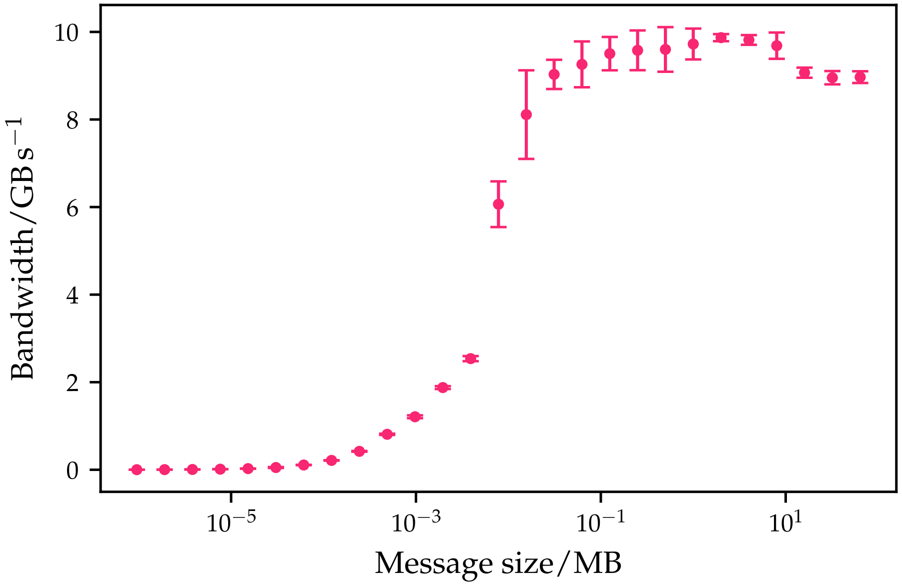
BANDWIDTH AND LATENCY

Communication time to transmit a message of size S:

$$T_{\text{comm}} = l + \frac{1}{\beta} S$$

- l: latency (time to send a message of size $S = 0$)
- β: peak bandwidth

- Latency in the rendezvous protocol includes *control messages* required for successful message transmission.
- Bandwidth performance is a function of the message size S.
- A parallel application that *scales* well hides T_{comm} with useful computation.
- See Culler et al. (1996) and Alexandrov et al. (1997) for two more detailed models.

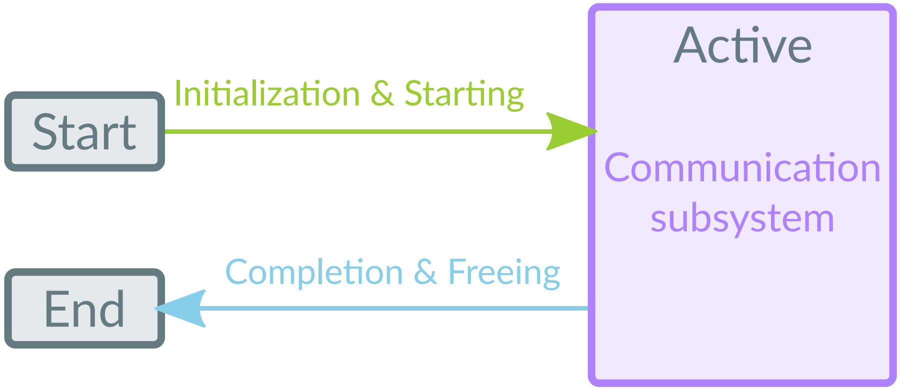


SAN bandwidth measurements on Cray XC50 system.

NON-BLOCKING MPI OPERATION

Recall from last lecture:

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.*



Blocking point-to-point:

- The message buffer is safe to be used after the procedure returns (completion).
- In case of a send, the message data may have not fully arrived on the receiver side (e.g. `MPI_Bsend` and eager protocol).

Non-blocking point-to-point:

- The message buffer ***cannot*** be modified or safely read upon return of the procedure.
- The *starting and initialization procedure* returns *immediately* and is *incomplete*.
- Non-blocking communication requires ***request*** objects to query the state of the operation.
- You can use the request objects together with `MPI_Wait` to *wait* for completion or `MPI_Test` to *test* for completion.

You can use a blocking receives to match a non-blocking send and vice versa.

SEMANTICS OF NON-BLOCKING COMMUNICATION

The semantics of non-blocking communication suitably extend the definitions of last lecture. See Section 3.7.4 and 3.5 in the MPI 4.0 standard, respectively.

Order

Non-blocking communication operations are *ordered* according to the execution order of the calls that *initiate* the communication. The ***non-overtaking*** requirement of blocking communication is extended with this definition for non-blocking communication.

Progress

A call to `MPI_Wait` that *completes* a receive will eventually terminate and return if a matching send has been *started*, unless the send is satisfied by another receive. If the matching send is non-blocking, then the matching receive should *complete* even if no call is executed by the sender to *complete* the send. The same applies for a `MPI_Wait` that *completes* a send. If a `MPI_Test` that *completes* a receive is repeatedly called with the same arguments, and a matching send has been *started*, then the call will eventually return `flag = true`, unless the send is satisfied by another receive. The same applies to a `MPI_Test` that *completes* a send.

NON-BLOCKING POINT-TO-POINT COMMUNICATION

- Standard (non-blocking) sending and receiving of messages in MPI is achieved with `MPI_Isend` and `MPI_Irecv`, respectively.
- They are both (*all* non-blocking procedures are) *local* and *incomplete*. The "`I`" in the procedure name stands for *incomplete* and *immediate*. A send operation requires a matching receive operation and vice versa. You can match non-blocking send and receive operations with blocking receive and send operations, respectively.

```c
int MPI_Isend(
    const void *buf,        // message body
    int count,              // message body
    MPI_Datatype datatype,  // message body
    int dest,               // message envelope
    int tag,                // message envelope
    MPI_Comm comm,          // message envelope
    MPI_Request *request
)
```

```c
int MPI_Irecv(
    void *buf,              // message body
    int count,              // message body
    MPI_Datatype datatype,  // message body
    int source,             // message envelope
    int tag,                // message envelope
    MPI_Comm comm,          // message envelope
    MPI_Request *request
)
```

- The same rules apply to the non-blocking version as for the blocking counterpart but a `MPI_Request` object is required this time for both procedures!
- The message status is obtained later by calling `MPI_Wait` or `MPI_Test` (or their variations).

NON-BLOCKING POINT-TO-POINT COMMUNICATION

What are the benefits?

- Non-blocking communication is important both *for reasons of correctness and performance*
- For complex communication patterns, the use of only blocking communication (without buffering) is difficult because ***you*** must ensure that each send is matched with a receive in an *order that avoids deadlock*.
- For communication patterns that are determined only at runtime, this is even more difficult.
- Non-blocking communication allows to *overlap* with different communication operations to avoid *serialization* of such operations.
- Non-blocking communication allows to hide communication latency with *useful work*.
- The same *four* communication modes can be used with non-blocking sends: *standard*, *buffered*, *synchronous* and *ready*.
- The communication model implies that communication is initiated by the sender. The communication will generally have *lower overhead* if a receive operation is already posted when the sender initiates the communication (data can be sent directly to the receive buffer and there is no need to *queue* a pending send request).
  → ***Rule:*** *post receive operations as early as possible.*

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;
}
```

- Discussed in last lecture.
- This code may deadlock depending on the MPI implementation.

NON-BLOCKING POINT-TO-POINT ALTERNATIVE

```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);
    MPI_Request request;
    if (0 == rank) {
        MPI_Irecv(&recv, 1, MPI_INT, 1, 99, MPI_COMM_WORLD, &request);
        MPI_Send(&rank, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
    } else {
        MPI_Irecv(&recv, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, &request);
        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;
}
```

- This code uses non-blocking receive procedures.
- The code *will not deadlock*.
- Is the code correct?

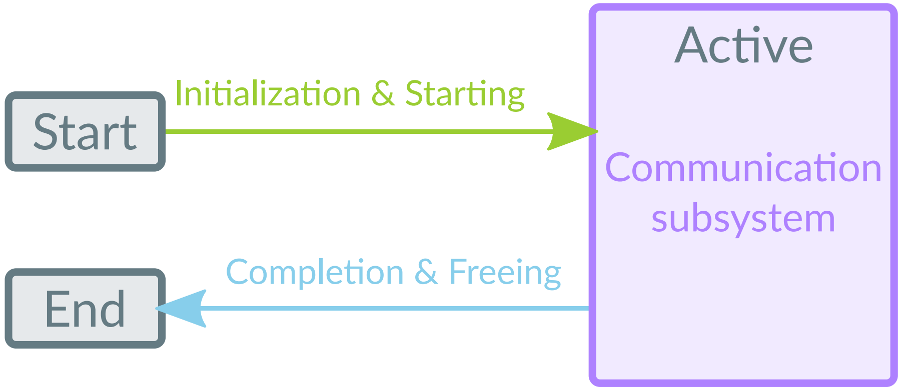
NON-BLOCKING POINT-TO-POINT ALTERNATIVE

```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);
    MPI_Request request;
    if (0 == rank) {
        MPI_Irecv(&recv, 1, MPI_INT, 1, 99, MPI_COMM_WORLD, &request);
        MPI_Send(&rank, 1, MPI_INT, 1, 98, MPI_COMM_WORLD);
    } else {
        MPI_Irecv(&recv, 1, MPI_INT, 0, 98, MPI_COMM_WORLD, &request);
        MPI_Send(&rank, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
    }
    MPI_Wait(&request, MPI_STATUS_IGNORE); // required synchronization
    std::cout << "Rank " << rank << " got ID " << recv << " from other rank\n";
    MPI_Finalize();
    return 0;
}
```

- This code uses non-blocking receive procedures.
- The code *will not deadlock.*
- Is the code correct? ***This code is!***

COMMUNICATION COMPLETION

- All non-blocking communication procedures require a *request* object.
- The *completion* of a send or receive operation ***guarantees*** that the communication buffers are safe to be used again.
- The `MPI_REQUEST_NULL` designates a *null handle*. A `MPI_Request` object is set to `MPI_REQUEST_NULL` by `MPI_Wait` or `MPI_Test` when the operation has completed.



Completion guarantees that the communication buffer is safe to be used again for read/write operations.

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

- The call to `MPI_Wait` returns when the operation identified by `request` is *complete*.
- `request` will be deallocated and set to `MPI_REQUEST_NULL` upon completion.
- This procedure is ***non-local***.

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

- The call to `MPI_Test` returns `flag = true` when the operation identified by `request` is *complete*.
- `request` will be deallocated and set to `MPI_REQUEST_NULL` upon completion.
- This procedure is ***local***.

COMMUNICATION COMPLETION: WAIT MULTIPLE

- It is convenient to be able to wait or test for any, some or all pending operations in a list.
- MPI provides variations of `MPI_Wait` and `MPI_Test` for this purpose.

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

- Blocks until *one* of the active handles in the list has *completed*.
- The list may contain `MPI_REQUEST_NULL` handles.
- Returns the `index` in the array and the `status` of the completed operation.

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

- Blocks until *all* active handles in the list *complete*.
- The list may contain `MPI_REQUEST_NULL` handles.
- The list of requests and statuses must be of the same length.

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

- Blocks until *at least one* of the active handles in the list have *completed*.
- Returns `outcount` indices of the completed operations and statuses.
- All arrays must have at least length `incount` (the procedure does not allocate memory).

COMMUNICATION COMPLETION: TEST MULTIPLE

The same applies when you want to test instead. These are ***local*** procedures, while the wait procedures on the previous slide are ***non-local***.

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

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

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

Behaves like `MPI_Waitsome` except that it returns immediately (local procedure).

- Both `MPI_Waitsome` and `MPI_Testsome` fulfill a ***fairness requirement:*** if a request for a receive repeatedly appears in a list of requests passed to the procedures then the receive will eventually succeed, unless the send is satisfied by another receive.
- The same applies for send requests.

NON-BLOCKING POINT-TO-POINT EXAMPLE

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

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

    MPI_Request send_requests[2];
    if (0 == rank) {
        int send = 101;
        MPI_Isend(&send, 1, MPI_INT, 1, 123, MPI_COMM_WORLD, &send_requests[0]);
        send = 102;
        MPI_Isend(&send, 1, MPI_INT, 1, 321, MPI_COMM_WORLD, &send_requests[1]);
        MPI_Waitall(2, send_requests, MPI_STATUSES_IGNORE);
    } else if (1 == rank) {
        int recv;
        MPI_Recv(&recv, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << "Received first message: " << recv << '\n';
        MPI_Recv(&recv, 1, MPI_INT, 0, 321, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << "Received second message: " << recv << '\n';
    }
    MPI_Finalize();
    return 0;
}
```

What is wrong in this code?

NON–BLOCKING REDUCTION EXAMPLE

Blocking:

```cpp
#include <mpi.h>

double reduction_I(double val,
                   const int root,
                   const MPI_Comm comm)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (root == rank) {
        double recv;
        for (int i = 0; i < size; ++i) {
            if (root == i) { continue; }
            MPI_Recv(&recv, 1, MPI_DOUBLE, i,
                     123, comm, MPI_STATUS_IGNORE);
            val += recv;
        }
    } else {
        MPI_Ssend(&val, 1, MPI_DOUBLE, root, 123, comm);
    }
    return val;
}
```

Apart from the $\mathcal{O}(n)$ reduction, what else is bad in this code?

Non-blocking:

```cpp
#include <mpi.h>
#include <numeric>
#include <vector>

double reduction_II(double val,
                    const int root,
                    const MPI_Comm comm)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (root == rank) {
        std::vector<MPI_Request> recv_reqs(size,
                                           MPI_REQUEST_NULL);
        std::vector<double> recv(size, val);
        for (int i = 0; i < size; ++i) {
            if (root == i) { continue; }
            MPI_Irecv(&recv[i], 1, MPI_DOUBLE, i,
                      123, comm, &recv_reqs[i]);
        }
        MPI_Waitall(recv_reqs.size(), recv_reqs.data(),
                    MPI_STATUSES_IGNORE);
        val = std::accumulate(recv.begin(), recv.end(), 0.0);
    } else {
        MPI_Ssend(&val, 1, MPI_DOUBLE, root, 123, comm);
    }
    return val;
}
```

- Communication subsystem manages messages (they may overlap for different processes).
- Can we still improve?

NON-BLOCKING REDUCTION EXAMPLE

Non-blocking:

```cpp
#include <mpi.h>
#include <numeric>
#include <vector>

double reduction_II(double val,
                    const int root,
                    const MPI_Comm comm)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (root == rank) {
        std::vector<MPI_Request> recv_reqs(size,
                                           MPI_REQUEST_NULL);
        std::vector<double> recv(size, val);
        for (int i = 0; i < size; ++i) {
            if (root == i) { continue; }
            MPI_Irecv(&recv[i], 1, MPI_DOUBLE, i,
                      123, comm, &recv_reqs[i]);
        }
        MPI_Waitall(recv_reqs.size(), recv_reqs.data(),
                    MPI_STATUSES_IGNORE);
        val = std::accumulate(recv.begin(), recv.end(), 0.0);
    } else {
        MPI_Ssend(&val, 1, MPI_DOUBLE, root, 123, comm);
    }
    return val;
}
```

- Communication subsystem manages messages (they may overlap for different processes).
- Can we still improve?

Non-blocking with C/T overlap:

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

double reduction_III(double val,
                     const int root,
                     const MPI_Comm comm)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (root == rank) {
        std::vector<MPI_Request> recv_reqs(size,
                                           MPI_REQUEST_NULL);
        std::vector<double> recv(size, val);
        for (int i = 0; i < size; ++i) {
            if (root == i) { continue; }
            MPI_Irecv(&recv[i], 1, MPI_DOUBLE, i,
                      123, comm, &recv_reqs[i]);
        }
        int ready = 0;
        std::vector<int> indices(size);
        while (size > 1) {
            MPI_Waitsome(recv_reqs.size(), recv_reqs.data(),
                         &ready, indices.data(),
                         MPI_STATUSES_IGNORE);
            for (int i = 0; i < ready; ++i, --size) {
                val += recv[indices[i]];
            }
        }
    } else {
        MPI_Ssend(&val, 1, MPI_DOUBLE, root, 123, comm);
    }
    return val;
}
```

1D DIFFUSION EQUATION

Consider the diffusion equation for the unknown u in *one* space dimension:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}, \quad x \in [0,1]$$

We assume some initial condition $u_0(x)$, *periodic boundary conditions* and a constant diffusivity D.

Spatial discretization using second order centered differences leads to:

$$\frac{\partial u(x_i,t)}{\partial t} \approx D\frac{u(x_{i-1},t) - 2u(x_i,t) + u(x_{i+1},t)}{\Delta x^2},$$

where Δx is a constant grid spacing and $x_i = (i+1/2)\Delta x$ and $i = 0, 1, \ldots, N$. Here N is the number of cells in the discrete domain.

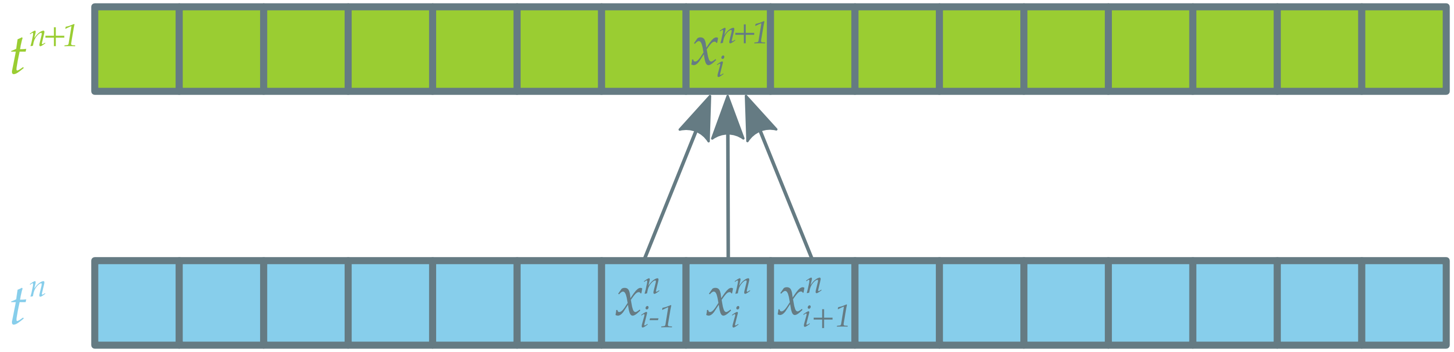
Temporal discretization using the forward Euler method leads to:

$$u(x_i,t^{n+1}) \approx u(x_i,t^n) + \frac{D\Delta t}{\Delta x^2}\left[u(x_{i-1},t^n) - 2u(x_i,t^n) + u(x_{i+1},t^n)\right]$$

1D STENCIL SCHEME

$$u(x_i, t^{n+1}) \approx u(x_i, t^n) + \frac{D\Delta t}{\Delta x^2}[u(x_{i-1}, t^n) - 2u(x_i, t^n) + u(x_{i+1}, t^n)]$$

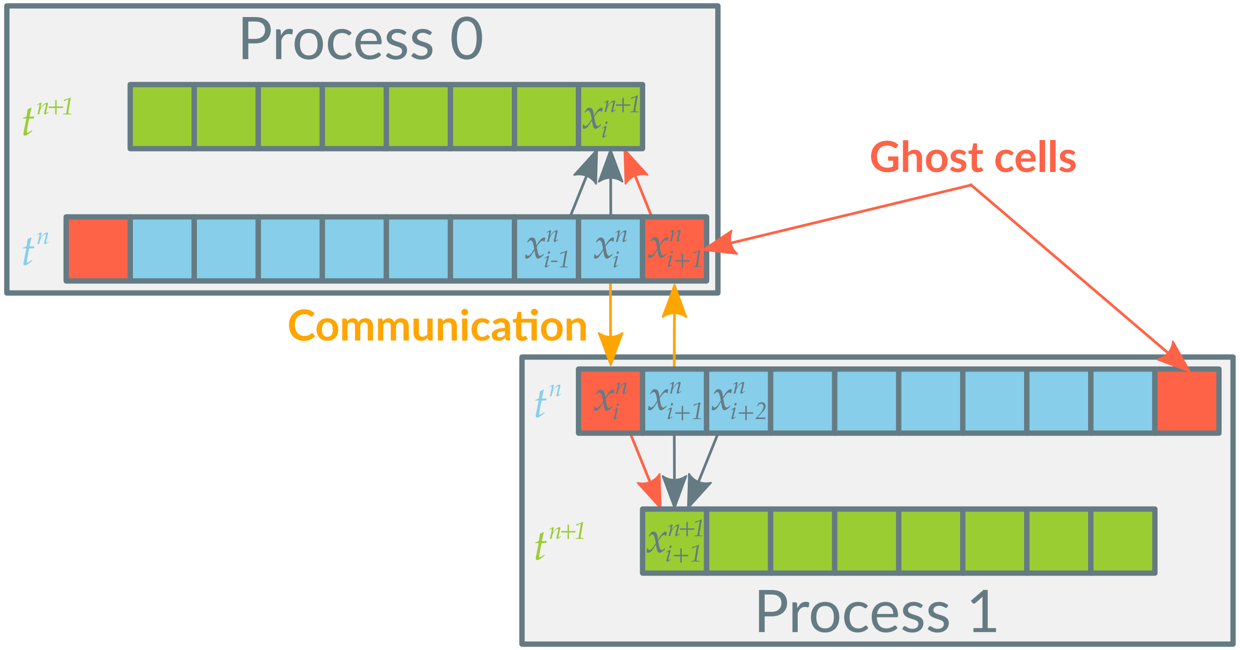
The approximation at discrete time t^{n+1} is obtained with information from time t^n.



- Consider a grid with N cells.
- Each update of cell x_i at the new time $t^{n+1} = t^n + \Delta t$ (where $t^n = n\Delta t$) requires *neighbor* cell values at time t^n.
- A stencil scheme is a numerical method to solve Partial Differential Equations (PDEs).

1D STENCIL SCHEME WITH MPI

Assume we have two MPI processes: each rank hosts a *subdomain* of the full physical domain *(domain decomposition)*→***distributed memory***.

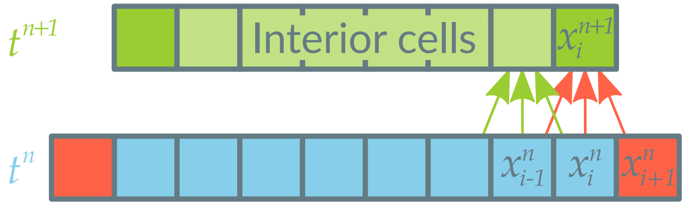


Ranks must *communicate* local data to fill ***ghost cells*** at subdomain boundaries needed to evaluate the stencil scheme on the processes.

BLOCKING STENCIL ALGORITHM

```cpp
std::vector<double> curr(N + 2); // +2 ghost cells
std::vector<double> next(N + 2); // +2 ghost cells

// time stepping
for (int step = 0; step < nsteps; ++step) {
    MPI_Sendrecv(&curr[N], 1, MPI_DOUBLE, right, 123,
                 &curr[0], 1, MPI_DOUBLE, left,  123,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&curr[1],     1, MPI_DOUBLE, left,  321,
                 &curr[N + 1], 1, MPI_DOUBLE, right, 321,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // evaluate stencil scheme
    for (int i = 1; i < N + 1; ++i) {
        next[i] = curr[i] +
            Fo * (curr[i - 1] - 2.0 * curr[i] + curr[i + 1]);
    }
    curr.swap(next);
}
```

- Create data structures for N *local* cells plus 2 *ghost cells*.
- Start by exchanging the ghost cells.
- Then evaluate the stencil scheme and advance to the next time step.
- Algorithm is inefficient because of blocking communication.



The interior cells do not depend on the communication and we can proceed computing while ghost cell exchange is active.

NON-BLOCKING STENCIL ALGORITHM

```
std::vector<double> curr(N + 2); // +2 ghost cells
std::vector<double> next(N + 2); // +2 ghost cells

auto applyStencil = [&](int i) {
    next[i] = curr[i] + Fo * (curr[i - 1] - 2.0 * curr[i] + curr[i + 1]);
};

// time stepping
for (int step = 0; step < nsteps; ++step) {
    MPI_Request reqs[4];
    MPI_Irecv(&curr[0],     1, MPI_DOUBLE, left,  123, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&curr[N + 1], 1, MPI_DOUBLE, right, 321, MPI_COMM_WORLD, &reqs[1]);
    MPI_Isend(&curr[1],     1, MPI_DOUBLE, left,  321, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&curr[N],     1, MPI_DOUBLE, right, 123, MPI_COMM_WORLD, &reqs[3]);

    // compute internal domain
    for (int i = 2; i < N; ++i) {
        applyStencil(i);
    }

    MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE); // wait for completion
    applyStencil(1); // complete left boundary
    applyStencil(N); // complete right boundary

    curr.swap(next);
}
```

NON-BLOCKING COLLECTIVES

Why non-blocking collectives?

- Performance can be improved by *overlapping* communication and computation.
- Non-blocking collective operations are optimized algorithms to exploit overlap and avoid synchronization.
- The non-blocking collective communication model is similar to the model used for non-blocking point-to-point communication.
- Non-blocking collectives require a `MPI_Request` object and are *compatible with all completion calls* we have discussed.

Unlike non-blocking point-to-point communication, non-blocking collectives ***cannot be matched with a blocking collective***.

```c
if (0 == rank) {
    MPI_Request req;
    MPI_Ireduce(sendbuf, recvbuf, count,
                datatype, op, root, comm, &req);
} else if (1 == rank) {
    // Illegal code:
    MPI_Reduce(sendbuf, recvbuf, count,
               datatype, op, root, comm);
}
```

NON-BLOCKING COLLECTIVES

Prototypes of non-blocking collectives:

- The function signatures of non-blocking collectives are identical to their blocking counterparts, except that an additional `MPI_Request` argument is necessary.
- All non-blocking procedures are prefixed with "I" for *incomplete* and *immediate*.

```c
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
                MPI_Op op, int root, MPI_Comm comm, MPI_Request *request)
```

```c
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

```c
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm,
               MPI_Request *request)
```

```c
int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm, MPI_Request *request)
```

```c
int MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm, MPI_Request *request)
```

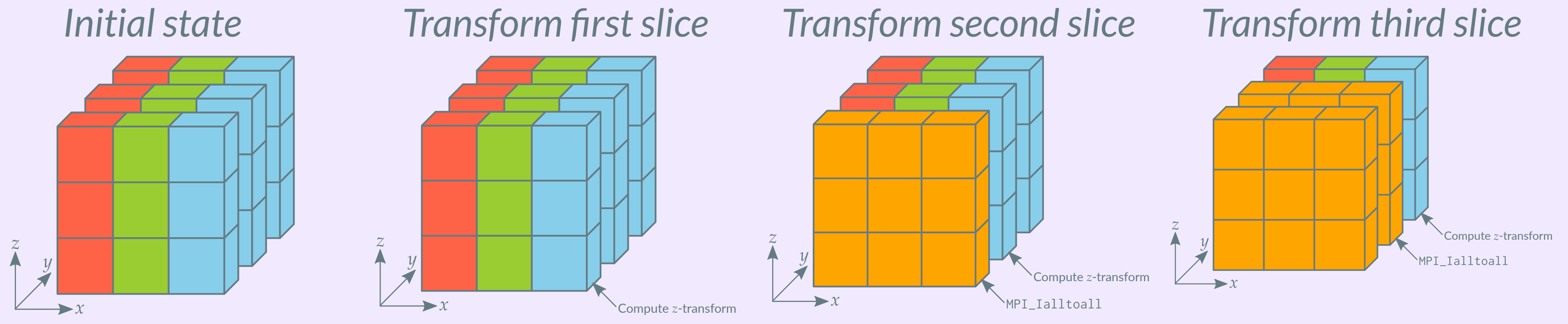
```c
int MPI_Ialltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm, MPI_Request *request)
```

```c
int MPI_Iexscan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
                MPI_Op op, MPI_Comm comm, MPI_Request *request)
```

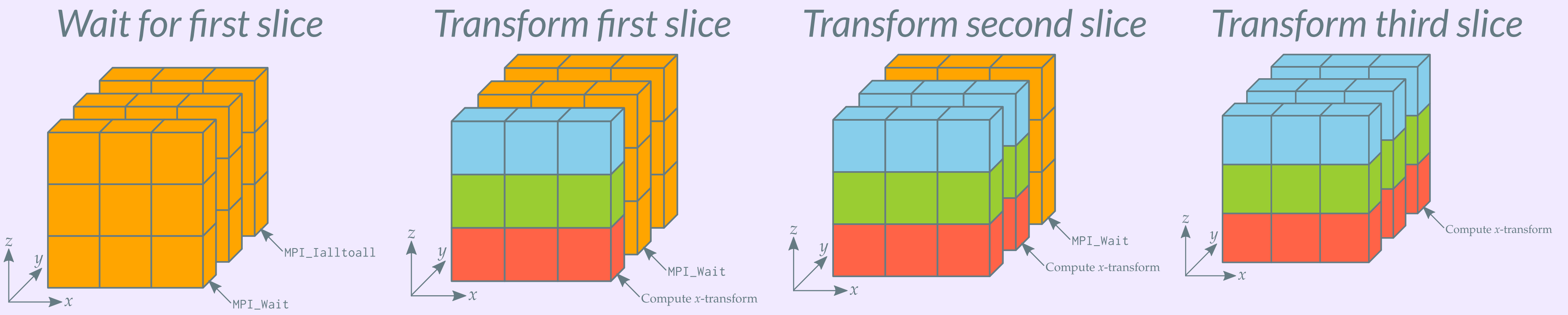
NON-BLOCKING `MPI_Ialltoall` EXAMPLE

A 3D Fast-Fourier-Transform (FFT) can be carried out with three 1D transforms and transposition of the data. Assume we have already transformed along the y-coordinate.

z-transform in the xz-plane



x-transform in the xz-plane



RECAP

- Use probing procedures to query message meta data.
- Understand the different communication protocols used by MPI. Most implementations use eager and rendezvous protocols for small and large messages, respectively.
- Non-blocking MPI procedures allow for optimizations both internal to the MPI implementation and by the user (compute/transfer overlap).
- Many (or most) algorithms allow to do work while communication is in progress.
- The idea behind non-blocking collectives is the same as for non-blocking point-to-point semantics.
- With non-blocking procedures MPI offers many possibilities to optimize your software and hide communication latencies.

Further reading:

- Section 3.7, 6.12 and Chapter 8 in *MPI 4.0 Standard*, MPI Forum, 2021 (pdf in `git` repo as well)
- Section 3.4 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Chapter 10 (especially Section 10.4) in Hager and Wellein, *"Introduction to High Performance Computing for Scientists and Engineers"*, CRC Press, 2011
- D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian and T. von Eicken, *"LogP: A practical model of parallel computation"*, ACM, 1996
- W. Gropp, E. Lusk, N. Doss and A. Skjellum, *"A high-performance, portable implementation of the MPI message passing interface standard"*, Elsevier, 1996
- A. Alexandrov, M. Ionescu, K. Schauser and C. Scheiman, *"LogGP: Incorporating long messages into the LogP model for parallel computation"*, Elsevier, 1997