HLRS & Stony Brook University

High-Performance
Computing Center
Stuttgart

MPI - Introduction

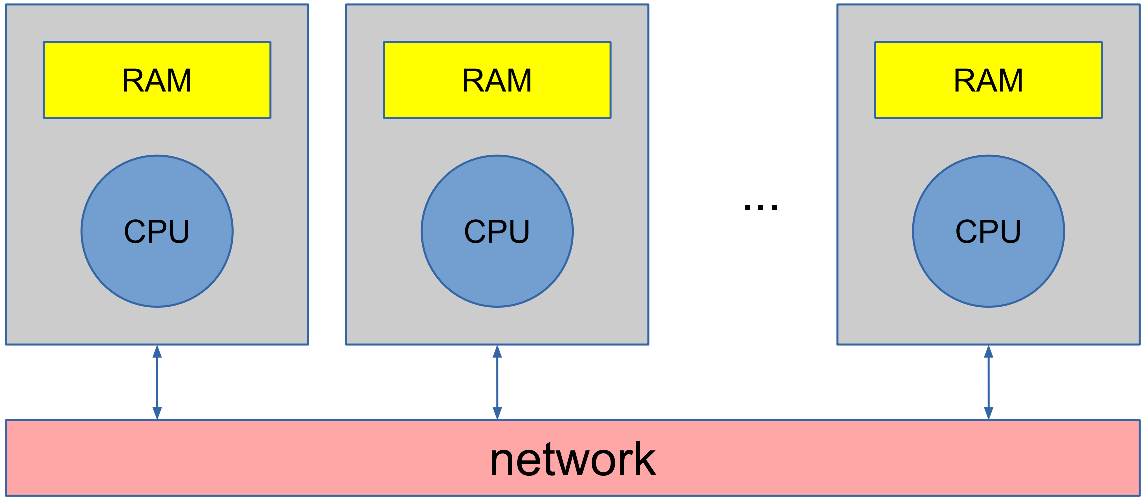
EuroMPI/USA 2025 Tutorial
Christoph Niethammer & Joseph Schuchart

Overview

- Motivation
- History of the MPI Interface
- MPI Basics
- Point to Point (P2P) communication
- Collective operations
- MPI and GPU
- Process Topologies
- MPI + Threads & Partitioned communication
- MPI one-sided communication / RMA

Motivation

- Supercomputer built as distributed memory systems with O(1000) nodes
- Many different HPC hardware vendors
- Software portability



History of the MPI interface

- 1994: meeting of 40 universities and companies: Standardization MPI-1.0
  based on experiences of existing communication APIs → Foundation of the ***MPI-Forum***
- 1995: MPI-1.1 minor corrections
- 1997: MPI-1.2 corrections, rationals, minor corrections
- 1997: MPI-2.0 large body of changes: MPI-IO, one-sided, dynamic process management, ...
- 2008: MPI-1.3: ammendments and cleanup in MPI-1
- 2009: MPI-2.1 lots of corrections on MPI-2
- 2009: MPI-2.2 corrections and simplifications including deprecation of the C++ Interface
- 2012: MPI-3.0 many new features: non-blocking collectives, RMA (aka true one-sided), new Fortran Interface
- 2015: MPI-3.1 minor corrections; few new routines
- 2021: MPI-4.0 Partitioned Communication, „Big-Count“
- 2023: MPI 4.1 corrections
- 2025: MPI 5.0 MPI ABI
- 2025+: MPI next … MPI Forum just started working on it :)

MPI Implementations

HLRS

The MPI Standard defines the syntax and semantics of communication functions but not the actual implementation!
There are different ways implementing MPI:

- **MPICH**:
  from Argonne National Labs (ANL)
  the first available implementation
  foundation of many MPIs

NEC's MPI-SX

MVAPICH2

Intel MPI

HPE/Cray's MPI

IBM's MPI (BG/L, discont')

- **Open MPI:**
  from Labs, Industry & Universities
  the „new" implementation

Sun's (Oracle's) HPC Cluster Tools

Fujitsu's MPI for K-Computer

Bull MPI

IBM Spectrum MPI

Nvidia HPC-X

Note: MPI includes an ABI specification since MPI 5.0 allowing interoperability between implementations

Message Passing Interface (MPI)

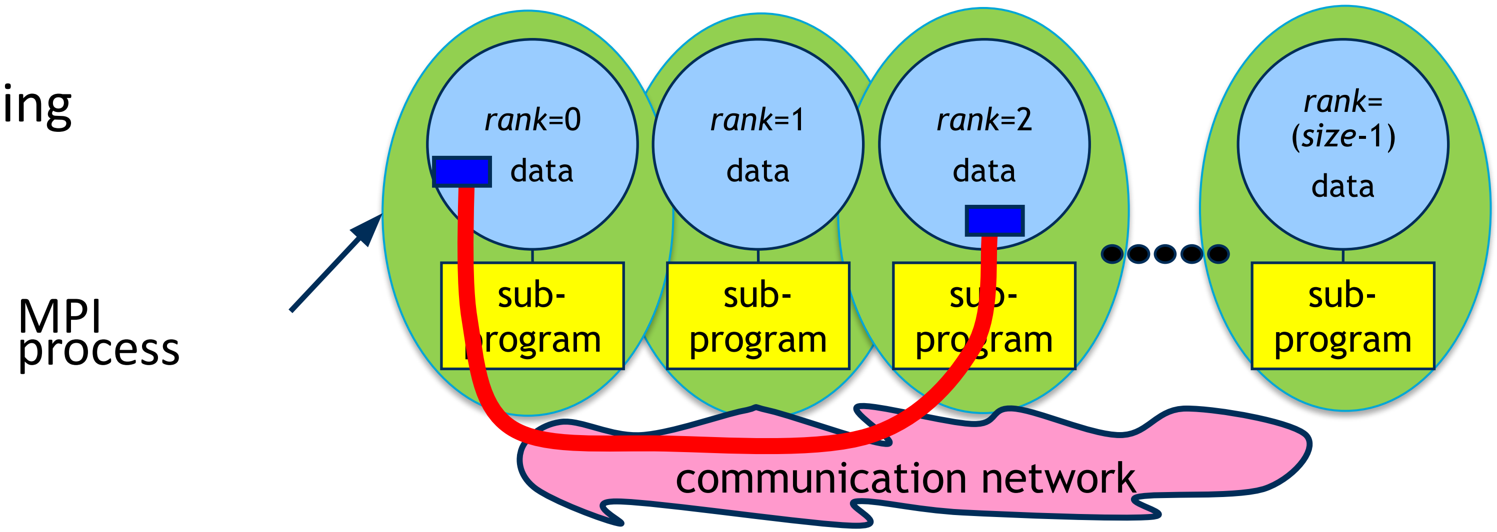
Basics

MPI Basics

- Each processor in a message passing program runs a sub-program referred to as **MPI process**
  - Written in a conventional sequential language, e.g., C or Fortran,
  - Typically the same on each processor (SPMD) but can also be different (MPMD)

- All communication, work and data distribution is based on value of **rank** *that identifies a MPI process in a* ***Group of MPI processes***

- Communication based on exchanging messages between MPI processes
  → message passing



Functionalities of the MPI

An abundance of communication functionality:

- Powerful functions to **group and structure MPI processes**, e.g. for **topologies** or structural close-ness of underlying physical problem at hand!
- **Blocking Point-to-Point (P2P)**, e.g. MPI_Send/MPI_Recv:

  Return to caller, as soon as the buffer may be reused by the caller!
- **Non-Blocking (Immediate) P2P**, e.g. MPI_Isend/MPI_Irecv:
  Communication „in background", for overlapping of computation & communication!
- **Collective Communications**, e.g. MPI_Bcast, MPI_Reduce, but also MPI_Allgatherv, i.e. „All" means all processes of so-called communicator participate, „gather" means data is gathered from all processes and „v" means each process gathers variable amount of data!
- **Remote-Memory-Access (RMA)**: Direct memory read, or write or even atomic increment of remote memory; with the proper Hardware-support this is way faster than P2P Communication...
- **Parallel File-IO**, e.g. MPI_File_open, MPI_File_read_all...

Simple MPI program example

H L R S

```
#include "mpi.h"
int main (int argc, char * argv[]) {
  int rank, size, sndbuf, rcvbuf;
  MPI_Comm comm;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_dup(MPI_COMM_WORLD, &comm);
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  if (0 == rank) {
    sndbuf=42;
    MPI_Send(&sndbuf, 1, MPI_INT, 1, 4711, comm);
  } else if (1 == rank) {
    MPI_Recv(&rcvbuf, 1, MPI_INT, 0, 4711, comm, &status);
  }
  MPI_Finalize ();
}
```

Initialize MPI with the **„world programming model“**

Create a communication context for a group of MPI processes, i.e., **communicator**

Addminitrative functions

Communication, here P2P between MPI processes with rank 0 and 1 in the group of comm

MPI program generation

- The compiler's knowledge about the MPI API comes from the interface definitions in the header file (#include 'mpi.h') in C or the module (use mpi_f08) in Fortran
- At the end the linker has to link in the MPI library

```
gcc -I <MPI_INSTALLATION_DIR>/include \
          -o mpi_prog mpi_prog.c \
          -L <MPI_INSTALLATION_DIR>/lib -lmpi
```

Path to include directory

Path to library directory

The library to be linked to, here: `libmpi.so`

MPI libraries come with compiler wrappers, e.g., `mpicc` as replacements for the original compiler commands to simplify this

```
mpicc -o mpi_prog mpi_prog.c
```

Note: For applications using the cmake build system, the FindMPI module (find(MPI)) can be used.

Executing an MPI program

- Requires
  - Starting of MPI processes on various resources
  - Propagating connectivity information between the MPI processes
- MPI startup command:

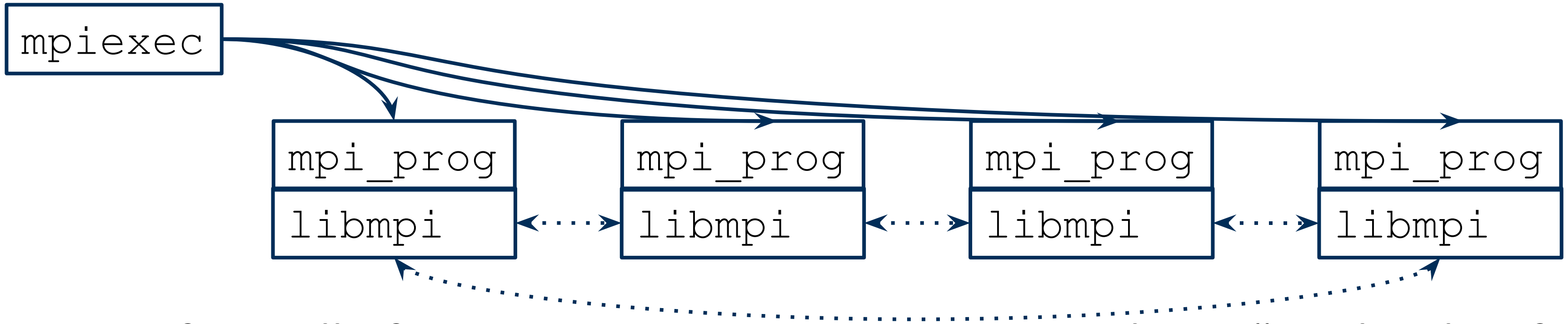
  mpiexec -n <numprocs> <additional options> <program>

Note: MPI implementations provide many additional options to mpiexec, e.g., for controlling process placement

MPI program execution example using Open MPI

The following command starts the program execution of `mpi_prog` using 4 processes, passing 2 arguments.

```
mpiexec -n 4 ./mpi_prog arg1 arg2
```



- Upon first call of `MPI_Init()` processes „get to know“ each other following the “world model” When the processes return from `MPI_Init()`, MPI_COMM_WORLD is defined and a processe’s rank in it is known.
- **Where** the processes are executed, i.e., on which node / which core, is MPI implementation-dependent. All MPI implementations allow to control this *mapping*.

Availability of MPI for other programming languages

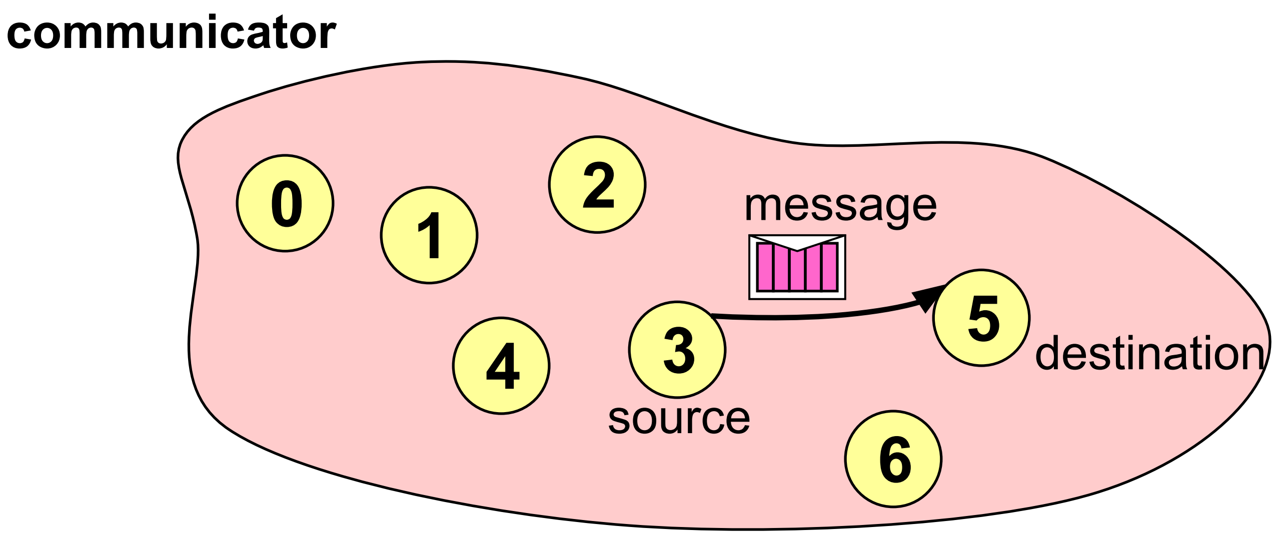
- MPI defines only a C and Fortran interface
- There exist many „Language Bindings“ – typically based on the C-Interface
  - C++: via the C-Interfaces or, Boost.MPI (MPI 1.1), MPL (MPI 3.1), ...
  - Rust: rsmpi (MPI 3.1, https://docs.rs/mpi/latest/mpi/)
  - Python: mpi4py (https://mpi4py.readthedocs.io/)
  - Java (z.B., in Open MPI via import mpi.*)
- All of those require the availability of an MPI library installation (C/Fortran)

Message Passing Interface (MPI)

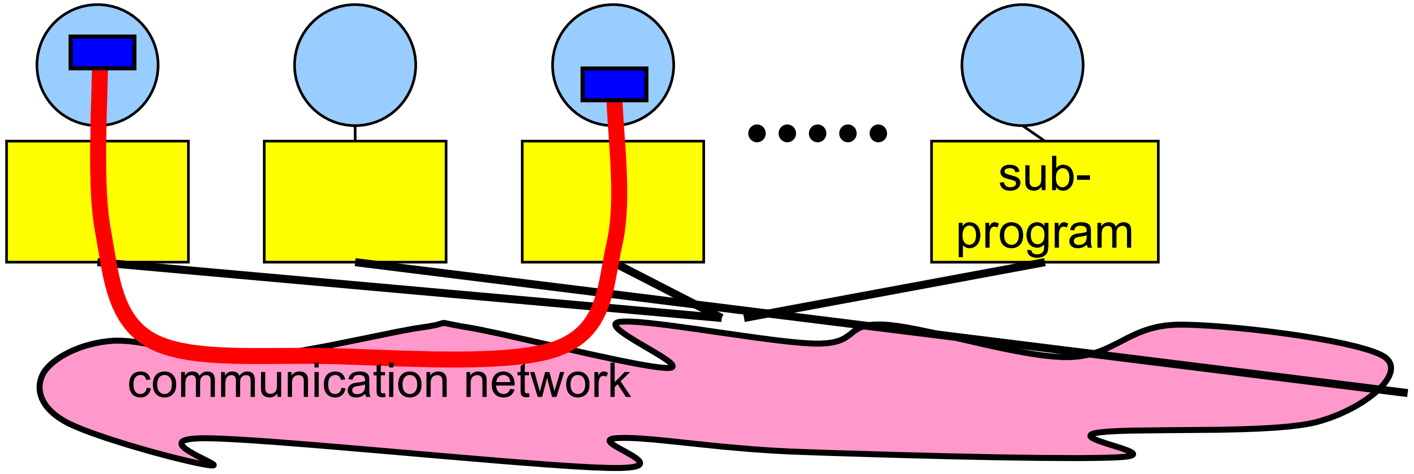
Point to Point (P2P) communication

Point to Point (P2P) communicatoin

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator.



Messages



- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:

– sending process	– receiving process i.e., the ranks
– source location	– destination location
– source data type	– destination data type
– source data size	– destination buffer size
– source tag	– recv tag

Point to Point (P2P) Communication

HLRS

Simplest form of MPI communication is **blocking point-to-point**

- One process sends, another process receives: „two-sided Communication“.
- The MPI Standard defines for each call the API as:

```
int MPI_Send(void * buf, int count, MPI_Datatype ddt, int rank, int tag, MPI_Comm comm);
```

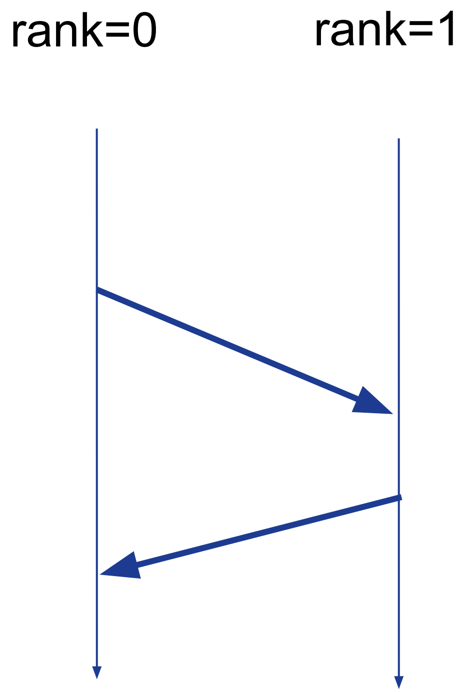
Send the buffer pointed to by `buf` with `count` elements of type `ddt` to process `rank` within the communicator `comm` identified by `tag`.
The call returns as soon as `buf` may be reused by the application !

```
int MPI_Recv(void * buf, int count, MPI_Datatype ddt, int rank, int tag, MPI_Comm comm,
     MPI_Status * status);
```

Receive into the buffer pointed to by `buf` with max. `count` elements of type `ddt` from process `rank` within the communicator `comm` identified by `tag`.
The call returns as soon as `buf` contains the complete data!

Example: MPI Ping-Pong

```
#include "mpi.h"
int main (int argc, char * argv[]) {
  int rank, size, buf = 42;
  MPI_Comm comm;
  MPI_Init(&argc, &argv);
  MPI_Comm_dup(MPI_COMM_WORLD, &comm);
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  if (0 == rank) {
    MPI_Send(&buf, 1, MPI_INT, 1, 4711, comm);
    MPI_Recv(&buf, 1, MPI_INT, 1, 4711, comm, MPI_STATUS_IGNORE);
  } else if (1 == rank) {
    MPI_Recv(&buf, 1, MPI_INT, 0, 4711, comm, MPI_STATUS_IGNORE);
    MPI_Recv(&buf, 1, MPI_INT, 0, 4711, comm, &status);
  }
  MPI_Finalize ();
}
```

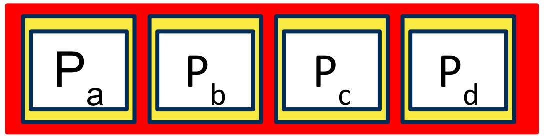


MPI Basics – Communicator & Rank

The **rank** of each process is **defined within a group of a communicator**!
Predefined communicators are made available, e.g., in the “world model” by `MPI_Init`.
The ranks of all processes within the communicator do not change

The number of processes started with `mpirun` match exactly the range from `0` to sizeof(MPI_COMM_WORLD)`-1`

```
mpirun -np 4 ./mpi_prog
```



Pre-defined communicators in the “world model”:

- `MPI_COMM_WORLD`: all processes
- `MPI_COMM_SELF`: The “own” process
- `MPI_COMM_NULL`: no process included

MPI communicators

- Each communication is relative to a specific **communicator** that provides the communication context!
- The predefined `MPI_COMM_WORLD` in the “world model” is a starting point used very often, ...
- There’s different methods to create communicators:

1. `MPI_Comm_dup` duplicates the passed communicator (i.e. inside of your own library – or for usage with threads! The newly created communicator is semantically different from the old comm.
2. Using `MPI_Comm_split_type` one may include/exclude specific processes from the newly created communicator.
```
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank, &comm);
// Splits into communicators that allow shared memory access (from MCW)
```
3. By a way of moving processes to groups, any kind of grouping is achieved, e.g. here multiple „Servers“ and 42 „Clients“ per server.
```
MPI_Comm_group (MCW, &group_mcw); // mcw is MPI_COMM_WORLD
for (i=0; i < size/42; i++) servers[i] = i*42;
MPI_Group_excl (group_mcw, i, servers, group_client);
MPI_Group_incl (group_mcw, i, servers, group_server);
```
4. For even more flexibility groups can be obtained from MPI Sessions in the “sessions model“ (see MPI 4.0)

Ranks

- The rank identifies different processes.
- The rank is the basis for any work and data distribution.

C

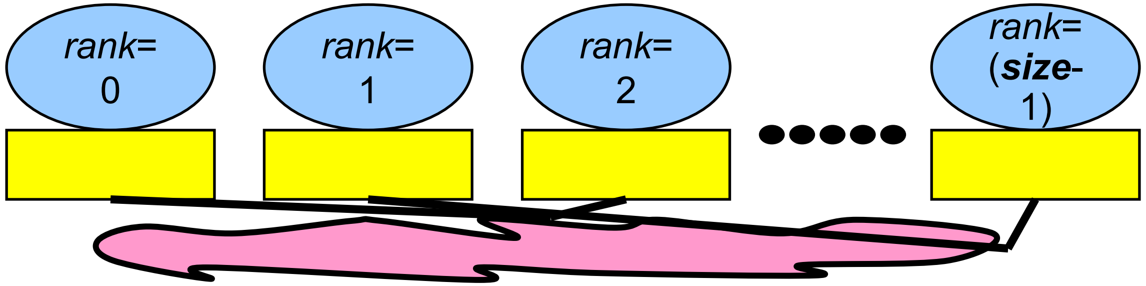
Fort
ran

Python

- C/C++: int MPI_Comm_rank(MPI_Comm comm, int **rank*)
- Fortran: MPI_COMM_RANK(comm, *rank*, *ierror*)

  mpi_f08: TYPE(MPI_Comm) :: comm
  INTEGER :: rank; INTEGER, OPTIONAL :: ierror

  mpi & mpif.h: INTEGER comm, rank, ierror
- Python: *rank* = comm.Get_rank()



CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierror)

Point to Point (P2P) Communication

Standard blocking send & receive


```
MPI_Send (&buf, len...);    MPI_Recv (&buf, len...);
```

When `MPI_Recv` returns, we know the message is received in full.
When `MPI_Send` returns, we **do not know**, whether the receiver has received, or even yet called `MPI_Recv`!

The message may be buffered by MPI, or be on the network
MPI could therefore block, if the message is too long!

Point to Point (P2P) communication protocols

Example: Standard blocking send & receive


```
MPI_Send (&buf, len...);
long_computation();
```

```
long_computation();
MPI_Recv (&buf, len...);
```

„Eager Protocol“:

If buffer to be send is small (ca. < 64 kB): copy the message into MPI-internal buffer, send to network and return to application

→ sender continues with `long_computation`!

„Rendezvous Protocol“

If buffer to be sent is too big (e.g. > 64 kB): send first message fragment “eagerly”, then wait until the receiver is ready for the rest, i.e., until the call of MPI_Recv!

Deadlock example

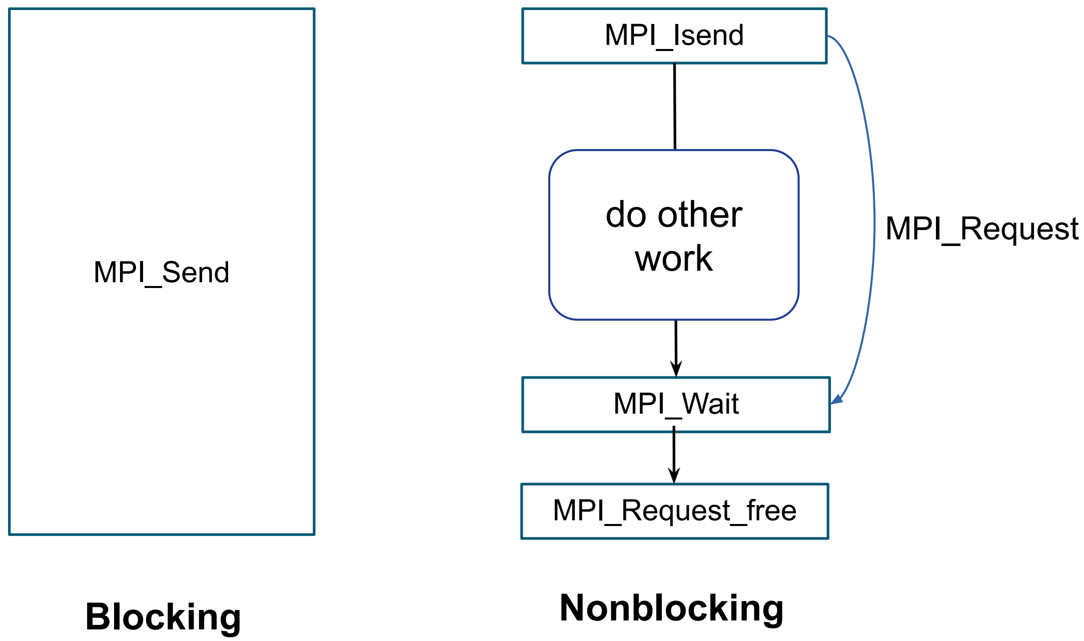
```
#include "mpi.h"
int main (int argc, char * argv[]) {
  int rank, size, sndbuf = 42, rcvbuf;
  MPI_Comm comm;
  MPI_Init(&argc, &argv);
  MPI_Comm_dup(MPI_COMM_WORLD, &comm);
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  int to = (rank + 1) % size
  int from = (rank + size -1) % size
  MPI_Send(&sndbuf, 1, MPI_INT, to, 4711, comm);
  MPI_Recv(&rcvbuf, 1, MPI_INT, from , 4711, comm, MPI_STATUS_IGNORE);

  MPI_Finalize ();
}
```

Deadlock!

Nonblocking operations

Nonblocking operations consist of:

- A **nonblocking procedure** call: it returns immediately and allows the
- code to perform other work
- At some **later** time the sub-program must **test or wait for the completion** of the nonblocking operation



Nonblocking (immediate) P2P

```
int MPI_Isend(void * buf, int count, MPI_Datatype ddt, int rank, int tag,
              MPI_Comm comm, MPI_Request * request);
```

Send non-blocking, i.e. `buf` with `count` elements of type `ddt` is send in “the background”!
The call returns immediately with an additional **request**.
However, **the buffer `buf` may not be touched until a corresponding `MPI_Wait()`** has been called.

Each request of a non-blocking call **has** to be finalized using either

```
MPI_Wait(),MPI_Test(),            // Wait for one request…
MPI_Waitall(),MPI_Testall(),      // Wait for all requests to finish
MPI_Waitsome(),MPI_Testsome(),    // Wait for possibly multiple
MPI_Waitany(),MPI_Testany()       // Wait for one of many requests
```

Nonblocking (immediate) P2P

If you want to overlap communication & computation, to „perfectly“ hide communication, then nonblocking immediate P2P is a **must**.

An example:


```
MPI_Isend
  (&buf, 1000, MPI_INT, 1, \
   4711, MPI_COMM_WORLD, &req);
long_computation();
MPI_Wait (&req, &status);
/* Now buf may be used again */
```

```
MPI_Irecv
  (&buf, 1000, MPI_INT, 0, \
   4711, MPI_COMM_WORLD, &req);
long_computation();
MPI_Wait (&req, &status);
/* Now buf may be used again */
```

MPI Datatypes

MPI provides datypes for message data

- Allow type transformation in inhomogeneous systems, e.g., different MPI processes run on different architectures (little/big endian)
- Writing data in platform independent file formats
- **Predefined MPI datatypes** for C and Fortran datatypes:

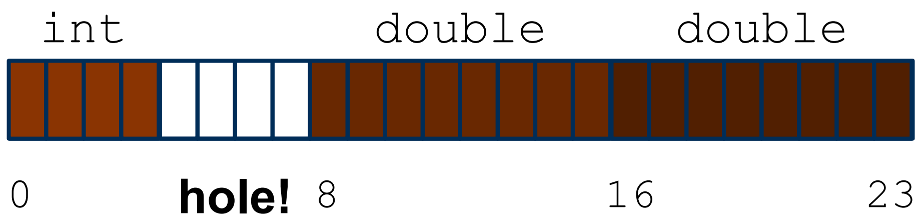
MPI datatype	C datatype
MPI_CHAR	char
MPI_INT	int
MPI_LONG	long
MPI_DOUBLE	double
MPI_UNSIGNED	unsigned int
…	

MPI Derived Datatypes

Out of a base data type, one may generate new derived datatype:

```
typedef struct {
  int location;
  double real;
  double imag;
} complex_loc_t cl;
```

Has the following layout in memory



In order to send data of `complex_loc_t` type, one has to describe it using base- and derived types (here only base):

```
array_of_types[0]     = MPI_INT;
array_of_blocklen[0] = 1;
array_of_disp[0]      = 0;
array_of_types[1]     = MPI_DOUBLE;
array_of_blocklen[1] = 2;
array_of_disp[1]      = &(cl.real)-&(cl.location);
MPI_Type_struct(2, array_of_blocklen, array_of_disp,
                array_of_types, &complex_loc_ddt);
MPI_Type_commit(&complex_loc_ddt); // Now one may send!
```

Message Passing Interface (MPI)

Collective communication

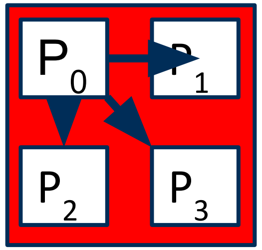
Collective communication

- In collective communication multiple MPI processes are involved in the communication
- The group of involved processes is given by the processes in the group of the communicator
- Examples: Broadcast, Reduction, Scatter, ...

- **Synchronizing** collective communication requires that all MPI processes must start the collective MPI operation before one MPI process can finish the operation
- Examples:Allreduce, Alltoall

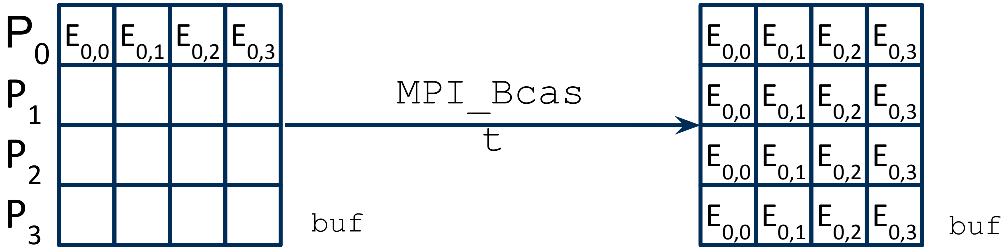
MPI Collective Communication - Broadcast

Broadcast data from one process to all others



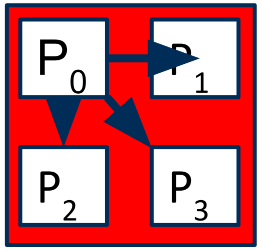

```
int MPI_Bcast(void *buf, int cnt, MPI_Datatype ddt, int root, MPI_Comm comm);
```

The memory pointed to by `buf` of length `cnt*ddt` will be sent from `root` to all processes in `comm` and copied into their memory `buf`.
Attention: `MPI_Bcast()` does not have to synchronize!



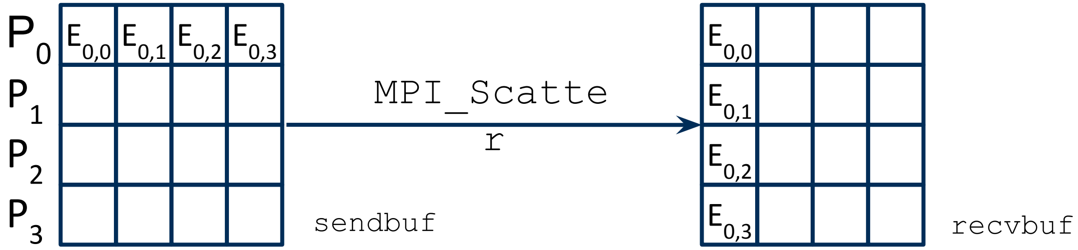
Here:
`cnt=4`
`root=0`

MPI Collective Communication - Scatter



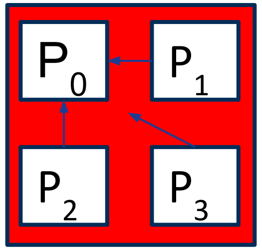

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype send_ddt,
                void *recvbuf, int recvcnt, MPI_Datatype recv_ddt,
                int root, MPI_Comm comm);
```

Process `root` distributes to all processes the data pointed to by `sendbuf` to all processes in `comm`; the data is stored in the receiving processes in `recvbuf`.



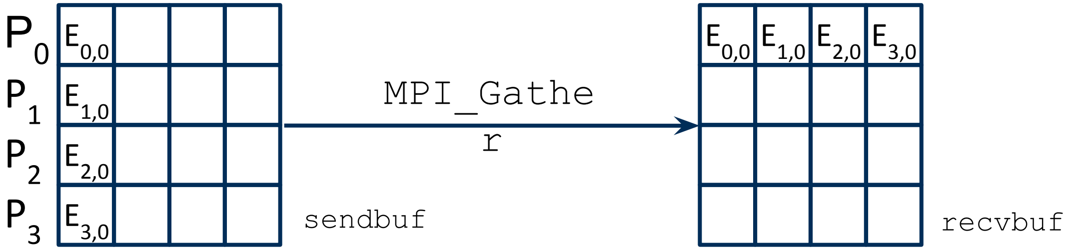
Here:
`sendcnt=1`
`root=0`

MPI Collective Communication - Gather



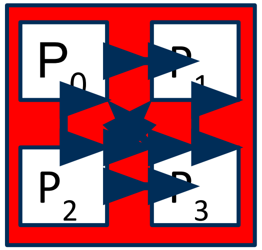

```
int MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype send_ddt,
                void *recvbuf, int recvcnt, MPI_Datatype recv_ddt,
                int root, MPI_Comm comm);
```

Process `root` collects from all processes the data pointed to by `sendbuf` and saves the data in `recvbuf`.



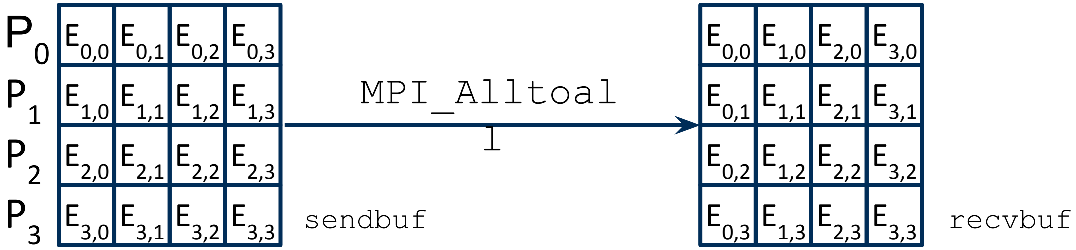
Here:
`sendcnt=1`
`root=0`

MPI Collective Communication - Alltoall




```
int MPI_Alltoall(void *sendbuf, int sendcnt, MPI_Datatype send_ddt,
                 void *recvbuf, int recvcnt, MPI_Datatype recv_ddt,
                 MPI_Comm comm);
```

Matrix-Transposition: Each process sends `sendcnt` data for every other process within communicator, i.e. for 4 processes 4*`sendcnt` elements.

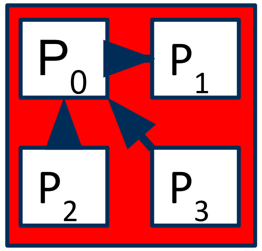


Here:
`sendcnt=1`

Used for Fast Fourier Transformations (FFT)!

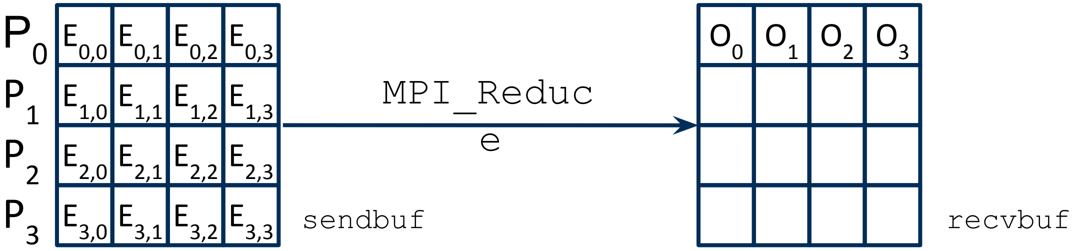
This is a really expensive operation!

MPI Collective Communication - Reduce




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

Process `root` using the operation `op` collects and combines data from `recvbuf`. Possible `ops` depend on the data type, and include `MPI_MIN`, `MAX`, `SUM`, `PROD`, `LAND`, `BAND`, `LOR`, `BOR`, `LXOR`, `BXOR`, `MINLOC`, `MAXLOC`. One may define one's own `MPI_Op`.



Here:
`sendcnt=`**`4`**
`root=0`

And:
`O=Output Vektor`

MPI Collective Communication ...

There are many variants of each of the collective calls:

- `All` variant: Instead of just `root` all processes within `comm` receive the data, e.g. `MPI_Allgather()` and `MPI_Allreduce()`
- `*v` variant: Variable number of elements per rank, i.e. each process may send a different amount of data.
- Immediate variant: Since MPI-3 also immediate, i.e. non-blocking variants of all calls are available, e.g. `MPI_Igather()` and `MPI_Ibarrier()`

The biggest up-point :

- **The MPI implementation may optimize the communication pattern according to the hardware, process mapping and network topology.**
- You may define your own operators

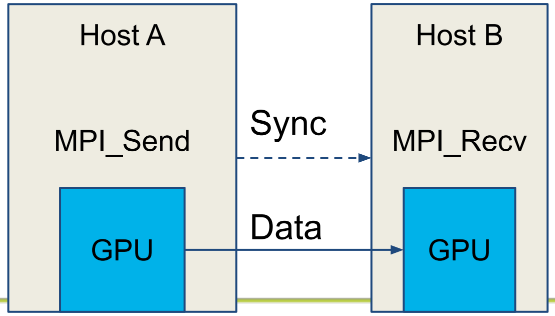
Message Passing Interface (MPI)

MPI and GPU Memory

MPI and GPUs: Implementation-Specifics

- Most MPI implementations support using GPU memory
- All MPI calls happen on the host

```c
/*
 * Program that shows the use of CUDA-aware macro and runtime check.
 */
#include <stdio.h>
#include "mpi.h"

#if !defined(OPEN_MPI) || !OPEN_MPI
#error This source code uses an Open MPI-specific extension
#endif

/* Needed for MPIX_Query_cuda_support(), below */
#include "mpi-ext.h"

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

    printf("Compile time check:\n");
#if defined(MPIX_CUDA_AWARE_SUPPORT) && MPIX_CUDA_AWARE_SUPPORT
    printf("This MPI library has CUDA-aware support.\n", MPIX_CUDA_AWARE_SUPPORT);
#elif defined(MPIX_CUDA_AWARE_SUPPORT) && !MPIX_CUDA_AWARE_SUPPORT
    printf("This MPI library does not have CUDA-aware support.\n");
#else
    printf("This MPI library cannot determine if there is CUDA-aware support.\n");
#endif /* MPIX_CUDA_AWARE_SUPPORT */

    printf("Run time check:\n");
#if defined(MPIX_CUDA_AWARE_SUPPORT)
    if (1 == MPIX_Query_cuda_support()) {
        printf("This MPI library has CUDA-aware support.\n");
    } else {
        printf("This MPI library does not have CUDA-aware support.\n");
    }
#else /* !defined(MPIX_CUDA_AWARE_SUPPORT) */
    printf("This MPI library cannot determine if there is CUDA-aware support.\n");
#endif /* MPIX_CUDA_AWARE_SUPPORT */

    MPI_Finalize();

    return 0;
}
```



MPI and GPUs: Requesting Device Support

HLRS

- Sessions replace MPI_Init and can be used to request and check for device memory support
- Device memory types documented in a side document
- Info objects = string dictionaries
- Applications should request device memory kind during startup

```
int cuda_device_aware = 0;
int cuda_managed_aware = 0;
int len, flag = 0;
MPI_Info info;
MPI_Session session;
// Usage mode : REQUESTED
MPI_Info_create(&info);
MPI_Info_set(info, "mpi_memory_alloc_kinds",
                   "system,cuda:device,cuda:managed");
MPI_Session_init(info, MPI_ERRORS_ARE_FATAL, &session);
MPI_Info_free(&info);
```

MPI and GPUs: Checking Device Support

- Check whether support for a specific device memory kind is **provided**
  - Application can check that **requested** memory kind is provided

```
// Usage mode : PROVIDED
MPI_Session_get_info(session, &info);
MPI_Info_get_string(info, "mpi_memory_alloc_kinds", &len, NULL, &flag);

if (flag) {
  char *val, *valptr, *kind;
  val = valptr = (char *)malloc(len);
  MPI_Info_get_string(info, "mpi_memory_alloc_kinds", &len, valptr, &flag);
  while ((kind = strsep(&val, ",")) != NULL) {
    if (strcasecmp(kind, "cuda:managed") == 0) {
      cuda_managed_aware = 1;
    } else if (strcasecmp(kind, "cuda:device") == 0) {
      cuda_device_aware = 1;
    }
  }
  free(valptr);
}
```

MPI and GPUs: Asserting Device Memory Usage

- **Assert** that only a specific device memory kind is used
  - MPI can optimize based on this assertion

```
// Usage mode : ASSERTED
MPI_Group wgroup;
MPI_Comm system_comm;
MPI_Group_from_session_pset(session, "mpi://WORLD", &wgroup);
// Create a communicator for operations on system memory
MPI_Info_create(&info);
MPI_Info_set(info, "mpi_assert_memory_alloc_kinds", "system");
MPI_Comm_create_from_group(wgroup,
   "org.mpi-forum.side-doc.mem-alloc-kind.cuda-example.system",
   info, MPI_ERRORS_ABORT, &system_comm);
MPI_Info_free(&info);
```

MPI and GPUs: Usage Example

- Allocate a CUDA device buffer and use the previously created communicator on which we asserted the use of CUDA device memory.

```
  // Example: Use with OpenMP target directive
  double *cuda_buffer;
  cudaMalloc(sizeof(double)*N, &cuda_buffer);

#pragma omp target
{
  …
}
  MPI_Send(cuda_buffer, N, MPI_DOUBLE, peer, cuda_comm);

  cudaFree(&cuda_buffer);
```

Message Passing Interface (MPI)

Process Topologies & Collective communication

MPI Process Topologies

- The linear identifier rank may not reflect the **logical communication pattern** of an application, e.g., in Cartesian process grids or graphs
- Topologies provide here a convenient naming mechanism for MPI processes in a group of processes
- Process topologies express communication path information that can help an MPI runtime in mapping MPI processes onto the underlying hardware topology

- Implemented via virtual topologies in MPI:
  - Cartesian Topology
  - Graph Topology
  - Distributed Graph Topology

- Topology information is associated to a communicator

MPI Cartesian Topologies

Cartesian Topologies represent a n-dimensional process grid

Communicator with an associated Cartesian topology is created by

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
   int dims[], int periods[], int reorder, MPI_Comm *comm_cart)
```

Introduces process coordinates with row-major numbering:

Hardware mapping

Example: 4 processes in 2x2 grid:

(0,0): R0	(0,1): R1
(1,0): R2	(1,1): R3

The following convenience function can factorize a number:

```
int MPI_Dims_create(int nnodes, int ndims, int dims[])
```

BUT: It takes not into account the data layout inside the application, i.e., will not optimize the amount of data to be communicated!

MPI Cartesian Topologies - coordinates

Various helpers for mapping coordinates to ranks in P2P communication functions here:

coordinate ⟶ rank:

```
int MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank)
```

rank ⟶ coordinate:

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
            int coords[])
```

Communication along the coordinate direction may be preformed, e.g., with MPI_Sendrecv:

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
          int disp, int *rank_source, int rank_dest)
```

MPI Graph Topologies

- Most general form to describe process topologies:
  - Nodes: MPI processes
  - Edges: communication
  - Edge weights: additional hints (e.g., bandwidth, latency, …)

- Come in two flavours:
  - Graph:
    - Each process has the full graph information

This does not scale!

  - Distributed Graph
    - Each process only has a local subset of the graph
    - Allow specification of additional weights for edges

MPI Distributed Graph Topologies

Distributed graphs can be created in two ways:

- Any process can specify any part of the graph:

```
int MPI_Dist_graph_create(MPI_Comm comm_old, int n,
    int sources[], int degrees[], int destinations[],
    int weights[], MPI_Info info, int reorder,
    MPI_Comm *comm_dist_graph)
```

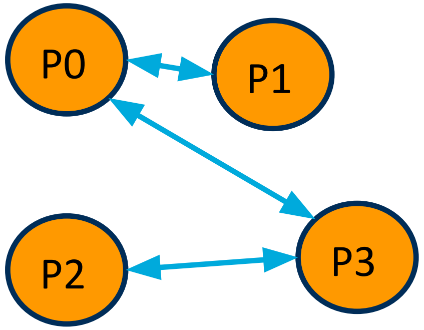
- Each process specifies its neighbours:

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
    int indegree, int sources[], int sourceweights[],
    int outdegree, int destinations[], int destweights[],
    MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

MPI Distributed Graph Topologies

for MPI_Dist_graph_create:

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	1,1,1,1,1,1
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	



for MPI_Dist_graph_create_adjacent:

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

MPI Graph Topologies

Information about the number of neighbours for a specified rank:

```
int MPI_Graph_neighbors_count(MPI_Comm comm,
    int rank, int *nneighbors)
```

Actual neighbours for a specified rank:

```
int MPI_Graph_neighbors(MPI_Comm comm,
    int rank, int maxneighbors, int neighbors[])
```

MPI Neighborhood Collectives

MPI Topologies define neighbours for MPI processes.
One can use collective operations on these neighbours – and replace P2P communication.
E.g., Bcast to or Reduce over all neighbours of a MPI process instead of a loop over all neighbours and performing individual send/recvs.

Related functions are of the form **MPI_Neighbor_<collective>**

Advantage:

- Communication pattern can be optimized by the MPI libraries.
- Easier to understand and simpler code by exposing what is intended (reduce!, gather!, etc.)

Message Passing Interface (MPI)

MPI and Threads (e.g. OpenMP) + Partitioned Communication

MPI and threads

When MPI is used in combination with threads the MPI library has to protect its internal state. Therefore the MPI has to be initiated with

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

Requested/provided can be one of

- MPI_THREAD_SINGLE: only one thread calls MPI functions
- MPI_THREAD_FUNNELED: only the thread that called MPI_Init_thread does MPI
- MPI_THREAD_SERIALIZED: only one thread at a time will call MPI functions
- MPI_THREAD_MULTIPLE: multiple threads may call MPI functions concurrently

Starting an hybrid MPI and threads program

- To start use still mpiexec to start mpi processes
- Take care that processes have enought resources, i.e., cores, assigned
- You want to make sure, that processes (and associated threads) do not move around, especially from one NUMA to another NUMA domain!

Example for starting with Open MPI :

```
$ mpiexec -n 2 –map-by slot:PE=2 –bind-to core -x OMP_NUM_THREADS=2  mpi+openmp_app …
```

2 MPI processes

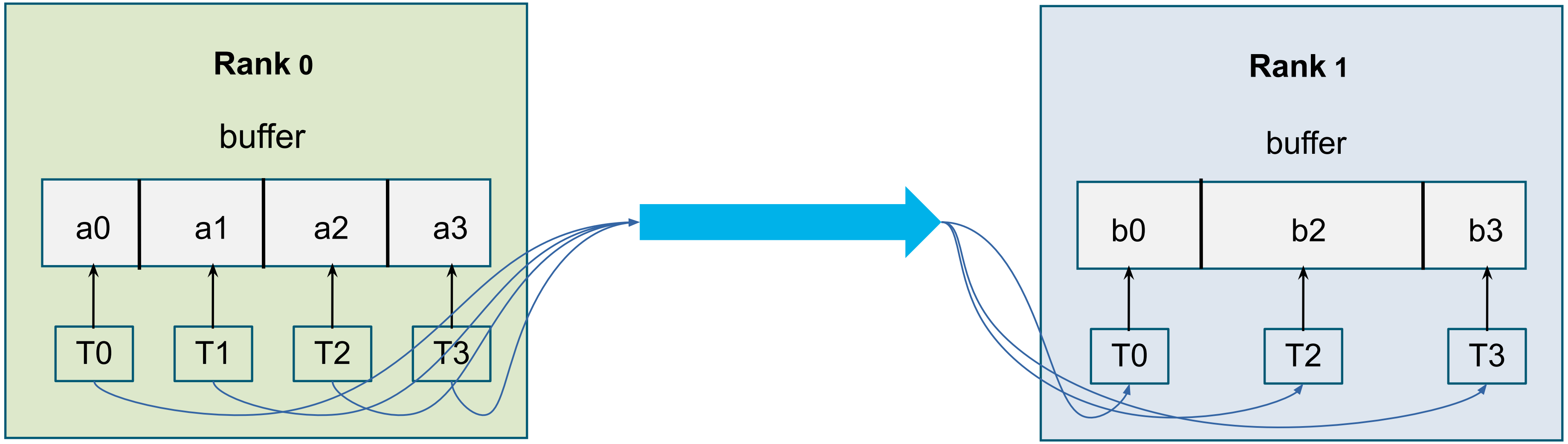
two slots, i.e., cores per MPI process for the threads

ensure MPI processes do not move around

pass environment variable to MPI processes to tell OpenMP to start 2 threads per MPI process

Motivating MPI Partitioned Communication

Hybrid MPI+threads programs want to use fine grained communication to allow various optimizations



BUT: Individual send/recv operations for each thread come with overheads and prevent optimizations

MPI Partitioned P2P Communication

- Introduced in **MPI 4.0**

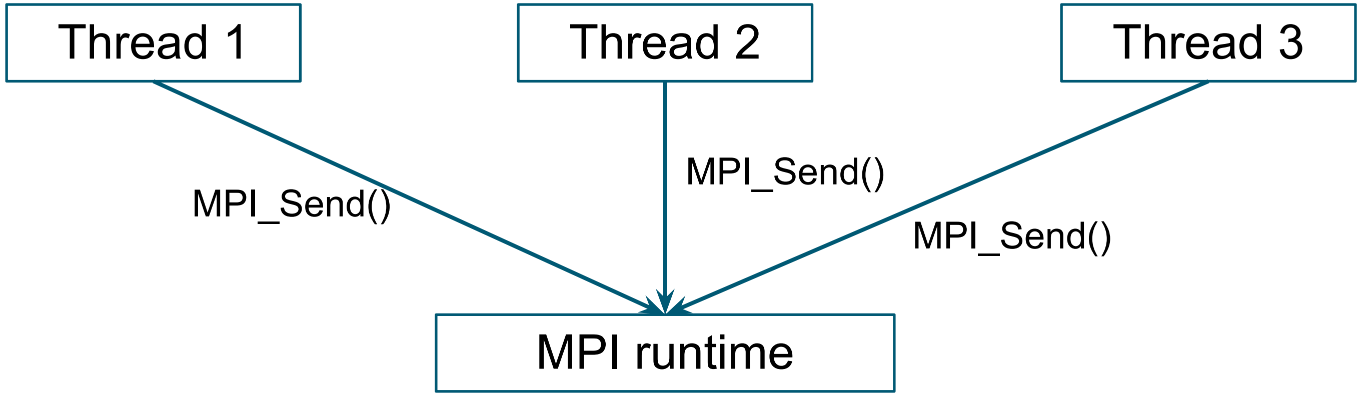
- **Extends** the **persistent P2P** communication

- Allows to **divide a message into a fixed number of *partitions*** for multiple contributions of data, potentially, by multiple actors (e.g. OpenMP threads)

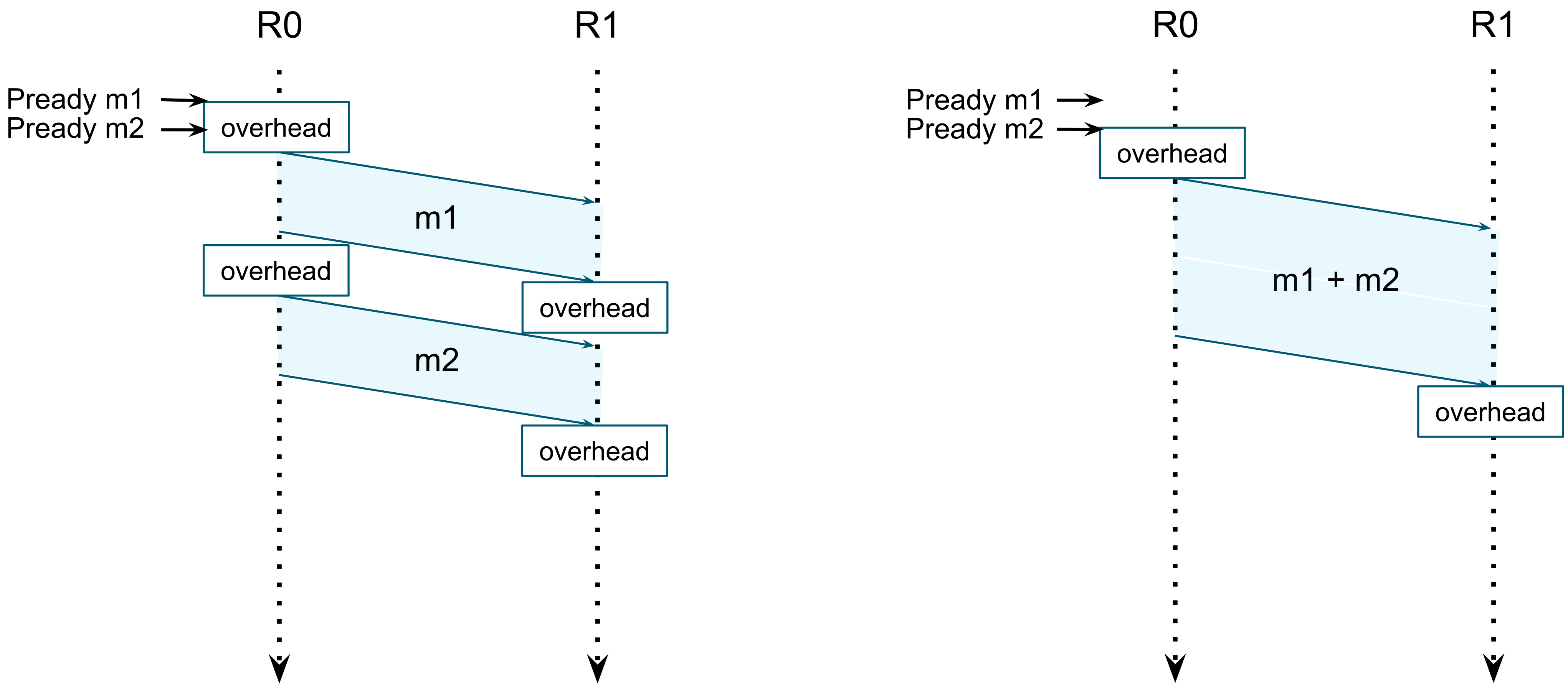
- **Enables various optimizations**:
  - reduced MPI runtime overheads
  - message aggregation
  - relaxed ordering for partitions

- Adds **two new initialization methods** for send and receive, which cannot be intermixed with other send/recv methods

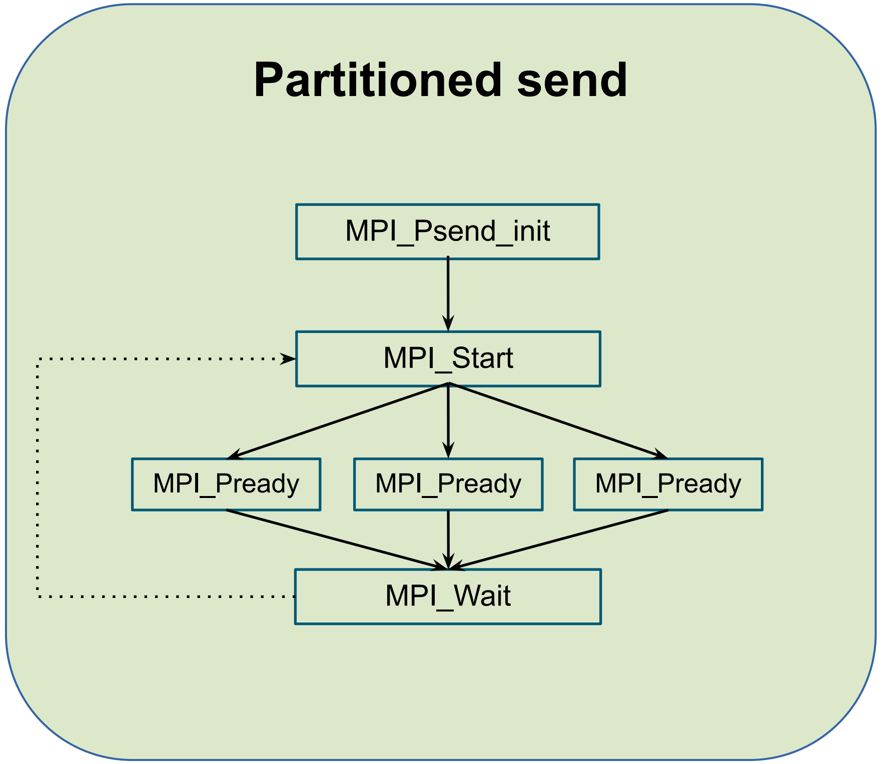
Possible optimizations: reduced runtime overheads



Possible optimizations: aggregation



Partitioned communication API: sending



- **MPI_Psend_init:**
  Initialize partitioned send operation for a buffer with a given number of partitions, returning a partitioned request handle.
- **MPI_Start:**
  Start partitioned communication associated with a partitioned communication request.
- **MPI_Pready:**
  Mark individual partition as ready to be sent. To be called for every partition.
- **MPI_Wait:**
  Complete send operation (one round of sending).

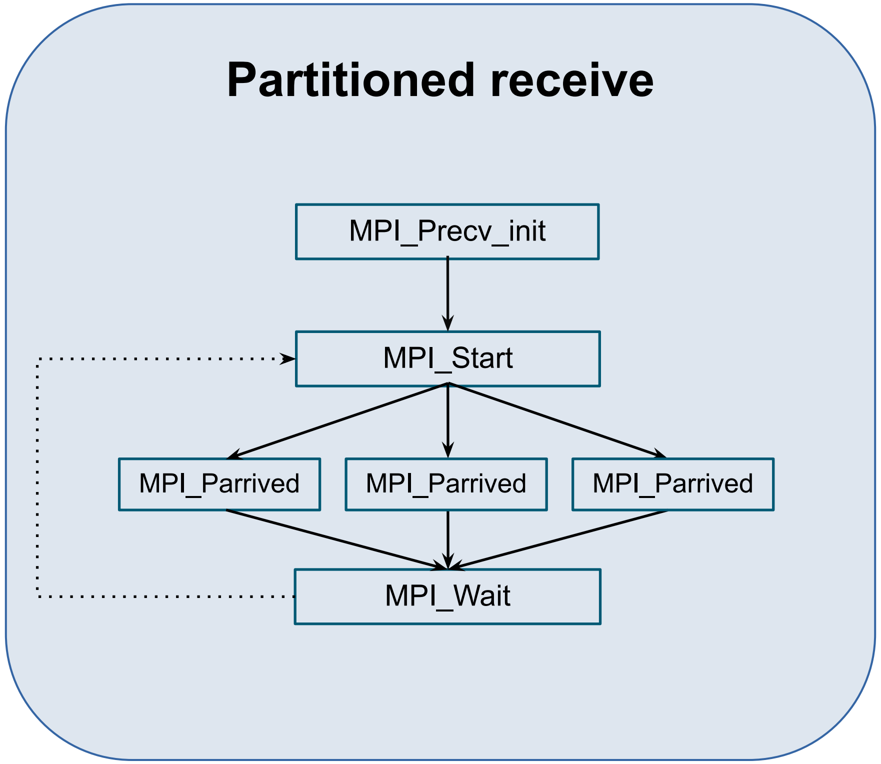
Partitioned communication API: receiving

- **MPI_Psend_init:**
  Initialize partitioned receive operation for a buffer with a given number of partitions, returning a partitioned request handle.

- **MPI_Start:**
  Start partitioned communication associated with a partitioned communication request.

- **MPI_Parrived:**
  Check if an individual partition is already received. To be called for every partition.

- **MPI_Wait:**
  Complete receive operation

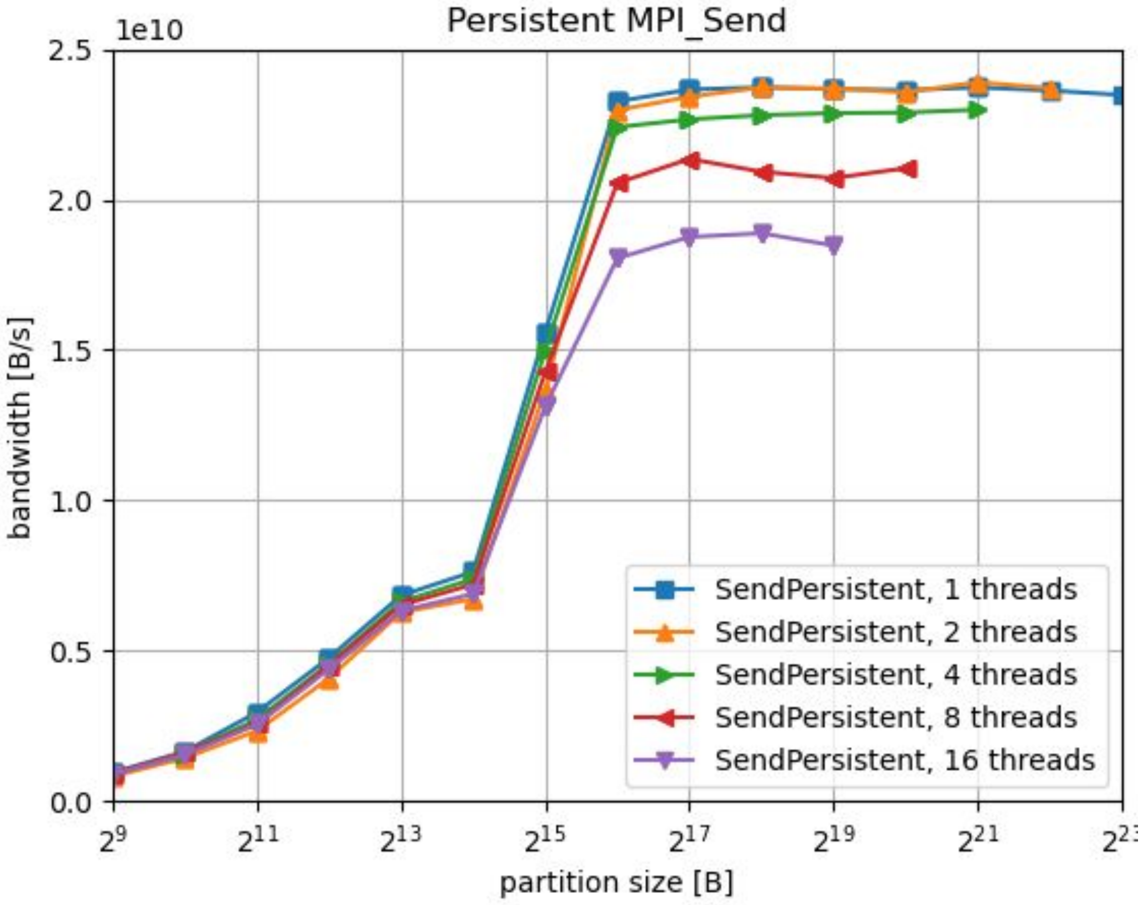


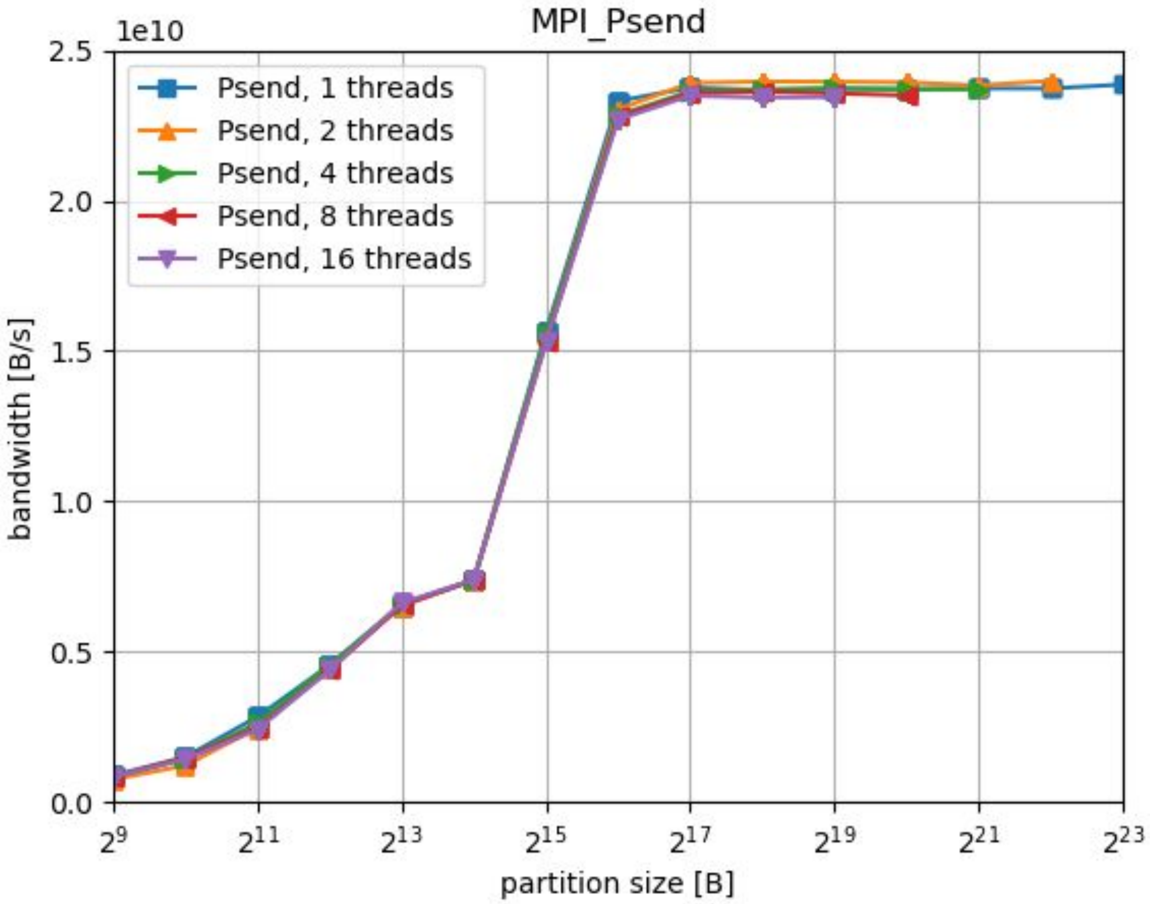
MPI Partitioned example with OpenMP

```
MPI_Psend_init( message,
                partitions_1, COUNT_1,
                MPI_DOUBLE, dest, tag,
                MPI_COMM_WORLD,
                MPI_INFO_NULL, &request );
MPI_Start(& request );
#prama omp parallel for shared(request)
for ( i = 0; i < partitions ; ++ i ) {
  /* compute and fill partition #i  */
  MPI_Pready(i , request );
}
while (! flag ) {
  MPI_Test( &request, &flag,
            MPI_STATUS_IGNORE );
  /* do useful work */
}
MPI_Request_free(& request );
```

```
MPI_Precv_init( message,
                partitions_2, COUNT_2,
                MPI_DOUBLE, src, tag,
                MPI_COMM_WORLD,
                MPI_INFO_NULL, &request );
MPI_Start(& request );
#prama omp parallel for shared(request)
for ( i = 0; i < partitions ; ++ i ) {
  int part_flag = 0;
  while( 0 == part_flag ) {
    /* do something useful */
    MPI_Parrived(request, i, &part_flag );
  }
}
while (! flag ) {
  MPI_Test( &request, &flag,
            MPI_STATUS_IGNORE );
  /* do useful work */
}
MPI_Request_free(& request );
```

Multithreaded communication performance





A. Schneewind, C.Niethammer – EuroMPI 24 – Benchmarking the State of MPI Partitioned Communication in Open MPI

Message Passing Interface (MPI)

One-sided communication / RMA

MPI one-sided communication

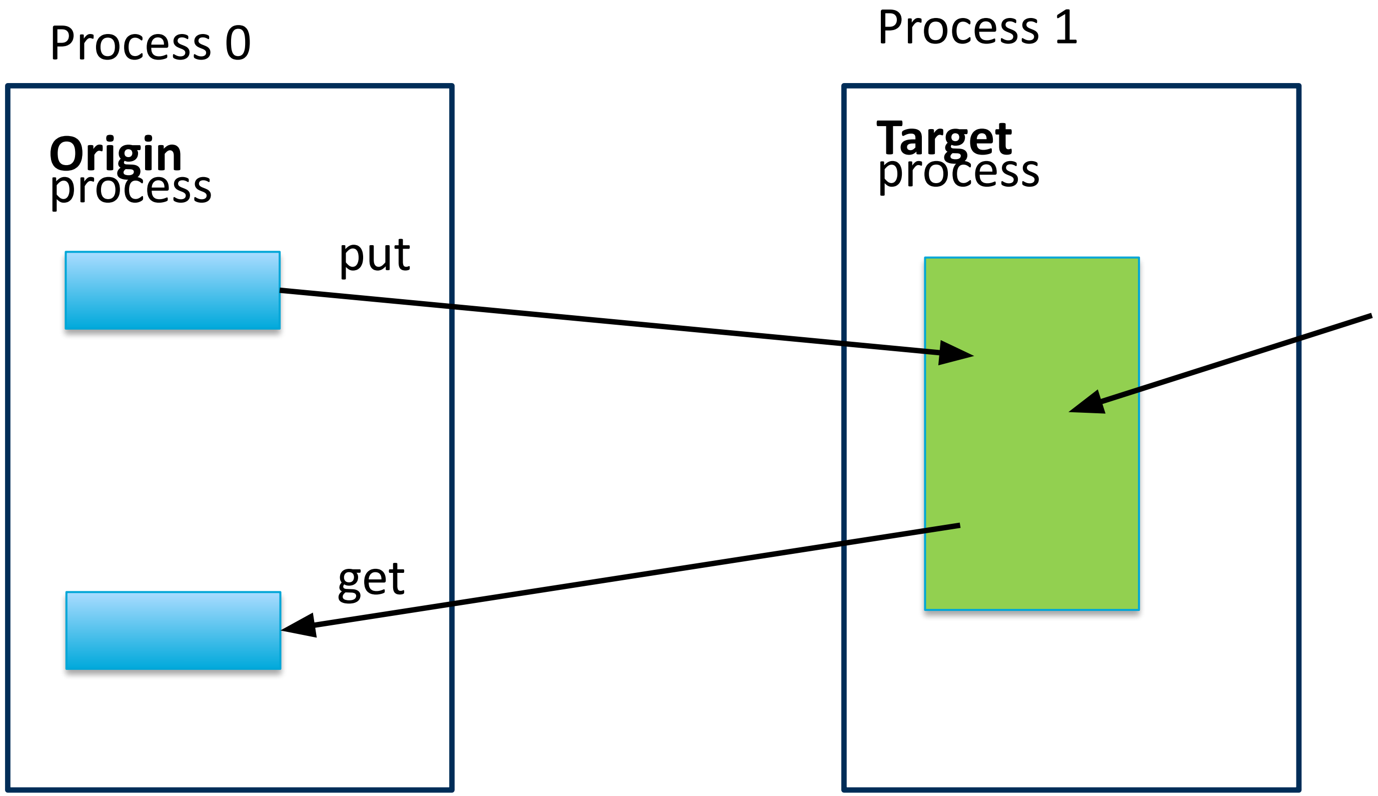
One-sided communication separates communication and synchronization

- **Reduces synchronization**
  Put/Get operations focus only on transfer without synchronization
  Only single process involved in transfer
  Models directly RDMA in the hardware

- **Designed with communication computation overlap in mind:**
  Put/Get are non-blocking

- **Mitigate some scalability problems of P2P**
  Frequently changing/unknown communication partners

MPI one-sided model



Window:
memory portion
accessible from
other processes

MPI one-sided operations

- **Window creation and allocation**
  Each process in a group of processes (Communicator) defines a chunk of his memory (Window), which can be afterwards accessed by all other processes in the group

- **Remote Memory Access (RMA)**
  Access to remote windows:
  - put, get, accumulate, …

- **Synchronization**
  RMA routines are non-blocking and
  must be surrounded by synchronization
  to guarantee that
  - RMA is locally and remotely finished
  - necessary cache operations are implicitly done

MPI window allocation

- Already allocated memory in the application:
  MPI_Win_create
- Create buffer and make it available as window:
  MPI_Win_allocate
  MPI_Win_allocate_shared
- Buffer must be allocated later, e.g., size yet unknown
  MPI_Win_create_dynamic

MPI one-sided data transfer

General Memory transfer:

- MPI_Get
- MPI_Put (race conditions!)

Atomic operations:

- MPI_Accumulate
- MPI_Get_accumulate
- MPI_Fetch_and_op
- MPI_Compare_and_swap

Get/fetch executed before the operation

R-Versions: MPI_Rget, MPI_Rput, …
→ request-based, only with passive synchronization

MPI_Put / MPI_Get

- Non-blocking calls
- Origin specifies arguments for both, origin and target
- Equivalent to P2P transfer via send/recv operations
- target buffer is at address
  target_addr = win_base_target + target_disp_origin * disp_unit_target

```
MPI_Put(const void *origin_addr, int origin_count,    MPI_Datatype
origin_datatype, int target_rank,
    MPI_Aint target_disp, int target_count,
    MPI_Datatype target_datatype, MPI_Win win)

MPI_Get(void *origin_addr, int origin_count,
    MPI_Datatype origin_datatype, int target_rank,
    MPI_Aint target_disp, target_count,
    MPI_Datatype target_datatype, MPI_Win win)
```

MPI one-sided synchronization

- **Active synchronization:**
- Origin and target are involved
  - MPI_Win_fence
  - MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait

- **Passive synchronization:**
- Only origin process calls synchronization functions, target is passive
  - MPI_Win_lock, MPI_Win_unlock

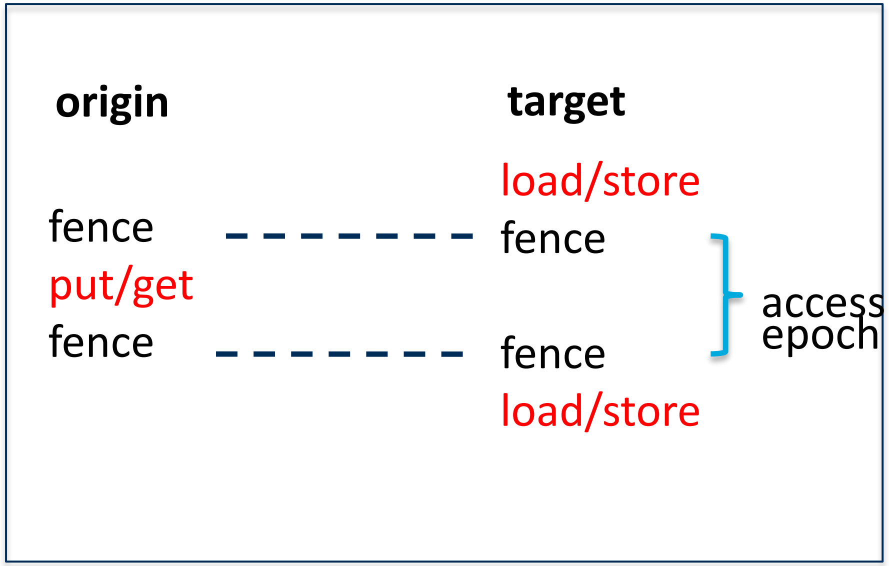
MPI active target synchronization

Barrier like synchronization is provided by

```
int MPI_Win_fence(int assert, MPI_Win win)
```

A fence ensures a consistent memory view before and after the fence for a provided window

Note: MPI_Barrier does not guarantee this!

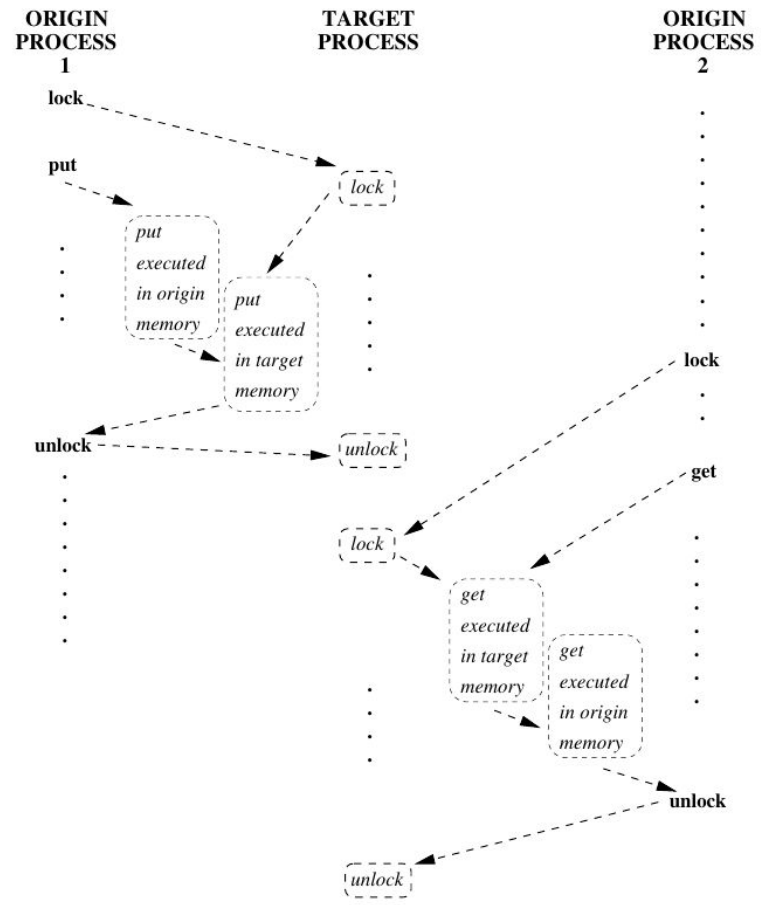


modify memory in window

MPI passive target synchronization

origin maintains locks for the target window via

```
int MPI_Win_lock(
    int lock_type,
    int rank, int assert,
    MPI_Win win)


int MPI_Win_unlock(
    int rank, MPI_Win win)
```



Literatur

- Gropp, B., Lusk, E.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, 3rd Ed., 2014

- Ofizielle Versionenen des Standards: https://www.mpi-forum.org/docs/