New Features in MPI 4.0

Rolf Rabenseifner, Tobias Haas
rabenseifner@hlrs.de tobias.has@hlrs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de
MPI Forum

- **MPI-1 Forum**
  - MPI-1.0 — May 1994
  - MPI-1.1 — June 1995

- **MPI-2 Forum**
  - MPI-1.2 — July 18, 1997: mainly clarifications.
  - MPI-2.0 — July 19, 1997: extensions to MPI-1.2.

- **MPI-3 Forum → MPI-4 Forum**
  - Started Jan. 14-16, 2008 (1st meeting in Chicago)
  - MPI-2.1 — June 23, 2008
    - mainly combining MPI-1 and MPI-2 books to one book
  - MPI-2.2 — September 4, 2009: Clarifications and a few new functions
  - MPI-3.0 — September 21, 2012: Important new functionality
  - MPI-3.1 — June 4, 2015: Errata & new: Nonblocking I/O, MPI_AINT
  - MPI-4.0 — June 9, 2021: Several new functionalities
    (not printed)
  - MPI-4.1 — scheduled for end 2023

© 2000-2022 HLRS, Rolf Rabenseifner

MPI course → Chap.1 Overview

Slide ~48 in the HLRS MPI course
Acknowledgments for the HLRS MPI course

This talk is based on our HLRS MPI-3.1/4.0 five-day course

→ All course slides + exercises: https://www.hlrs.de/training/self-study-materials/mpi-course-material

→ Used in many training courses: https://www.hlrs.de/training/ & https://vsc.ac.at/training

→ Course acknowledgments also apply:

  – The MPI-1.1 part of this course is partially based on the MPI course developed by the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
  – Thanks to the EPCC, especially to Neil MacDonald, Elspeth Minty, Tim Harding, and Simon Brown.
  – Course Notes and exercises of the EPCC course can be used together with these slides.
  – The MPI-2.0 part is partially based on the MPI-2 tutorial at the MPIDC 2000 by Anthony Skjellum, Purushotham Bangalore, Shane Hebert (High Performance Computing Lab, Mississippi State University, and Rolf Rabenseifner (HLRS)
  – Some MPI-3.0 detailed slides are provided by the MPI-3.0 ticket authors, chapter authors, or chapter working groups, Richard Graham (chair of MPI-3.0), and Torsten Hoefler (additional example about new one-sided interfaces)
  – Thanks to Claudia Blaas-Schenner from TU Wien (Vienna) and many other trainers and participants for all their helpful hints for optimizing this course over so many years.
  – **Thanks to Tobias Haas from HLRS for his Python binding of the exercises.** Thanks to Claudia Blaas-Schenner and David Fischak from TU Wien (Vienna) for their additional hints on the Python bindings. Additional background was a first draft from the HiDALGO project at HLRS.
Large counts
Large Counts with MPI_Count, …

- MPI uses different integer types
  - int and INTEGER
  - MPI_Aint = INTEGER(KIND=MPI_ADDRESS_KIND)
  - MPI_Offset = INTEGER(KIND=MPI_OFFSET_KIND)
  - MPI_Count = INTEGER(KIND=MPI_COUNT_KIND)

- sizeof(int) ≤ sizeof(MPI_Aint) ≤ sizeof(MPI_Offset) ≤ sizeof(MPI_Count)

- All count arguments are int or INTEGER.

- Real message sizes may be larger due to datatype size.

- MPI_Type_get_extent, MPI_Type_get_true_extent, MPI_Type_size, MPI_Type_get_elements return MPI_UNDEFINED if value is too large

- MPI_Type_get_extent_x, MPI_Type_get_true_extent_x, MPI_Type_size_x, MPI_Type_get_elements_x return values as MPI_Count

- MPI_Xxxx_c(...) in C: additional interfaces with large counts
- MPI_Xxxx(...) !(_c) in Fortran: overloaded interfaces with large counts

Two exceptions with explicit _c in Fortran: MPI_Op_create_c & MPI_Register_datarep_c
• Language independent definition

• C interface

• Fortran 2008 interface through mpi_f08 module

• Old Fortran interface through mpi module and mpif.h

MPI 3.1 page 28
MPI 4.0 page 37

3.2.4 Blocking Receive
The syntax of the blocking receive operation is given below.

```c
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

- `OUT buf`: initial address of receive buffer (choice)
- `IN count`: number of elements in receive buffer (non-negative integer)
- `IN datatype`: datatype of each receive buffer element (handle)
- `IN source`: rank of source or MPI_ANY_SOURCE (integer)
- `IN tag`: message tag or MPI_ANY_TAG (integer)
- `OUT status`: communicator (handle)
- `status object (Status)`

```c
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)
```

```fortran
MPI_Recv(buf, count, datatype, source, comm, status)
```

- `TYPE(*)`: dimension
- `INTEGER, INTENT(IN)`: count, source, tag
- `TYPE(MPI_Datatype)`: datatype
- `TYPE(MPI_Comm)`: comm
- `INTEGER, OPTIONAL, INTENT(OUT)`: ierror

```fortran
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
```

- `INTEGER`: COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS
- `INTEGER`: IERROR

Large count version in MPI-4.0
- `MPI_Recv_c(...)` in C
- `MPI_Recv(...) !(_c)` in Fortran

No large count in mpi / mpif.h

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf#page=60
https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf#page=77
# The Fortran support methods

<table>
<thead>
<tr>
<th>Fortran support method</th>
<th>MPI-1.1</th>
<th>MPI-2</th>
<th>MPI-3</th>
<th>MPI-4.0</th>
<th>MPI-next</th>
<th>MPI-…</th>
<th>far future</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE mpi_f08</td>
<td>x</td>
<td>x</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>USE mpi</td>
<td>x</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2b</td>
<td>2b</td>
<td>1</td>
</tr>
<tr>
<td>INCLUDE `mpif.h´</td>
<td>3</td>
<td>3</td>
<td>2a</td>
<td>2a/b</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Level of Quality:**

- **5** – valid and consistent with the Fortran standard (Fortran 2008 + TS 29113)\(^1\)
- **4** – valid and only partially consistent
- **3** – valid and small consistency (e.g., without argument checking)
- **2** – use is strongly (a) discouraged or (b) partially frozen (i.e., not with all new functions)
- **1** – deprecated
- **0** – removed
- **x** – not yet existing

\(^1\) For full consistency, Fortran 2003 + TS29113 is enough. Fortran 2018 and later versions include TS 29113. Without TS29113, same partial consistency as with the mpi module.
**MPI_Put**

- **C/C++**: `int 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)`

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

- **Fortran**: `MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, ierror)`

  ```fortran
  mpi_f08: TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
           INTEGER, INTENT(IN) :: origin_count, target_count
           INTEGER, INTENT(IN) :: target_rank
           TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
           INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
           TYPE(MPI_Win), INTENT(IN) :: win
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
  ```

  `mpi & mpif.h`: `<type> ORIGIN_ADDR(*)
  INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
  INTEGER TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP`

- **Python**: `win.Put(((origin_buf, origin_count, origin_datatype), target_rank, (target_disp, target_count, target_datatype)))`

  The course-slides include also the mpi4py binding, which are not part of the MPI standard.
Window Creation with MPI_Win_create

- **C/C++:**
  ```c
  int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
  ```

  ```c
  int MPI_Win_create_c(void *base, MPI_Aint size, MPI_Aint disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
  ```

- **Fortran:**
  ```fortran
  MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
  ```

  ```fortran
  mpi_f08:
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
  INTEGER, INTENT(IN) :: disp_unit
  or INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: disp_unit
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Win), INTENT(OUT) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
  ```

- **Python:**
  ```python
  win = MPI.Win.Create(memory, disp_unit, info, comm)
  ```

- Large count version, new in MPI-4.0
- Overloaded large count version since MPI-4.0

© 2000-2022 HLRS, Rolf Rabenseifner

MPI course → Chap.10 One-sided Communication  Slide ~334 in the HLRS MPI course
New persistent collectives
→ new terms „*nonblocking & co““
Non-Blocking Communications

Separate communication into **three phases**:

- Initiate nonblocking communication
  - returns immediately
  - routine name starting with MPI_I...

  → it is local,
  i.e., it returns independently of any other process’ activity

- Do some work (perhaps involving other communications?)

- Wait for nonblocking communication to **complete**, i.e.,
  - the send buffer is read out, or
  - the receive buffer is filled in

1) The definition of nonblocking is clarified

---

Complete rewording of MPI-4.0
Section 2.4 Semantic Terms
2.4.1 MPI Operations
2.4.2 MPI Procedures

MPI-1.1 – MPI-3.1:
→ nonblocking = incomplete

MPI-4.0:
→ nonblocking = incomplete AND local

“I” stands for
- Immediate (=local)
- and Incomplete

\[\text{nonblocking}^{1)}\]
Nonblocking operations consist of:

- A nonblocking procedure call: it returns immediately and allows the sub-program to perform other work
- At some later time the sub-program must **test** or **wait** for the completion of the nonblocking operation
Visiting MPI Chapter 2 Terms and Conventions

Operations and Procedures, (non)blocking / (non-)local

• MPI operations consist of four stages:
  – Initialization, starting, completion, freeing

• MPI operations can be
  – **Blocking**: all four stages are combined in a single complete/blocking procedure. → which returns when operation has completed.
  – **Nonblocking**: → next slide
  – **Persistent**: → 2nd next slide

• MPI procedures can be
  – **Non-local**: returning may require, during its execution, some specific semantically-related MPI procedure to be called on another MPI process.
  – **Local**: is not non-local. (See also discussion of “weak local”)

• MPI procedures (if they implement an operation or parts of it) can be
  – **Completing**: on return, all resources (e.g., buffers or array args) can be reused.
  – **Incomplete**: return before resources can be reused.
  – **Nonblocking**: incomplete AND local / **Blocking**: Completing OR non-local.

• Examples:
  – **Nonblocking**: • Incomplete & local: MPI_Isend, MPI_Irecv, MPI_Ibcast, MPI_Send_init
  – **Blocking**: • Completing & non-local: MPI_Send, MPI_Recv, MPI_Bcast
  • Incomplete & non-local: MPI_Mprobe, MPI_Bcast_init
  • Completing & local: MPI_Bsend, MPI_Rsend, MPI_Mrecv

Orthogonal concept, although in most cases:
• Incomplete/nonblocking communication proc. → local
• Complete/blocking communication proc. → non-local (with some exceptions)

The semantics of all operation-related MPI procedures is listed in Annex A.2 (since MPI-4.0)
Nonblocking operations consist of:

- A nonblocking procedure call: it is **incomplete** & returns **immediately** and allows the sub-program to perform other work → stages: **initialization + starting** = initiation

- At some later time the sub-program must **test** or **wait** for the completion of the nonblocking operation → stages: **completion + freeing**

**Nonblocking** synchronous send

New in MPI-4.0
Persistent Requests

For communication calls with identical argument lists in each loop iteration (only buffer content changes):

- **MPI_( ,B,S,R)Send_init** and **MPI_Recv_init**
  - Creates a persistent MPI_Request handle
  - Status of the handle is initiated as **inactive**
  - **Local** calls (does not communicate)
  - It only setups the argument list
- **MPI_Bcast_init** …, also for collective operations
  - **Blocking & collective** calls (may communicate)
- **MPI_Start**(request [,ierror] ) / **MPI_Startall**(cnt, requests [,ierror] )
  - Starts the communication call(s) as nonblocking call(s), i.e., handle gets **active**
- To be completed with regular MPI_Wait… / MPI_Test… calls → **inactive**
- **MPI_Request_free** to finally free such a handle

Usage sequence: init Loop(Start Wait/Test) Request_free

- Persistent inactive request → **active**
- Completes an active request handle → **inactive**

Recommendation: Never free an **active** request handle. Active request handles should be completed with WAIT or TEST.

New in MPI-4.0

Goal: Enables **additional optimizations** within the MPI library

Caused all these new definitions of the terms

New in MPI-4.0
Partitioned Point-to-Point Communication
Partitioned Point-to-Point Communication

- MPI-4.0:
  *Partitioned communication is “partitioned” because it allows for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation.*

- A point-to-point operation (i.e., send or receive)
  - can be split into partitions,
  - and each partition is filled and then “send” with MPI_Pready by a thread;
  - And same for receiving.

- Technically provided as a new form of persistent communication.
#define PARTITIONS 8
#define COUNT 6

double message[PARTITIONS*COUNT];
MPI_Count count_send = COUNT, count_recv=COUNT/2;
int source = 0, dest = 1, tag = 1, flag = 0, rank, thread_provided;
MPI_Request request;

MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &thread_provided);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* Sender part (rank 0) */
if (rank == 0){
    MPI_Psend_init(message, PARTITIONS, count_send, MPI_DOUBLE, dest, tag,
                    MPI_COMM_WORLD, MPI_INFO_NULL, &request);
    MPI_Start(&request);
    #pragma omp parallel for shared(request) num_threads(8)
    for(int i = 0; i < PARTITIONS; ++i){ /* 1 partition per thread */
        /* compute and fill partition message[COUNT*i...COUNT*(i+1)-1], then mark ready: */
        MPI_Pready(i, request);
    }
    while(!flag){
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
    }
    MPI_Request_free(&request);
}
/* Receiver part (rank 1) */
else if (rank == 1){
    /* We split every partition by half, i.e. count per partition divided by two, number or partitions increased by 2 */
    MPI_Precv_init(message, PARTITIONS*2, count_recv, MPI_DOUBLE, source, tag,
                   MPI_COMM_WORLD, MPI_INFO_NULL, &request);
    MPI_Start(&request);
    #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
    for (int j=0; j< PARTITIONS*2; j+=2){
        int part1_complete = 0, part2_complete = 0;
        int work1_complete = 0, work2_complete = 0;
        while(work1_complete == 0 || work2_complete == 0){
            /* test partition #j and #j+1 */
            if(!part1_complete) {MPI_Parrived(request, j, &part1_complete);}
            if(part1_complete && !work1_complete){
                /* Do work using partition j data */
                work1_complete = 1;
            }
            if(!part2_complete) {MPI_Parrived(request, j+1, &part2_complete);}
            if(part2_complete && !work2_complete){
                /* Do work using partition j+1 data */
                work2_complete = 1;
            }
        }
    }

    /* Need to complete request since MPI_PARRIVED doesn't. */
    MPI_Wait(&request, MPI_STATUS_IGNORE); /* Alternative: MPI_Test in loop and do useful work, see previous slide*/
    MPI_Request_free(&request);
}
Comments on Partitioned Communication

- Sequence is

  ```
  Init (Start Pready/Rarrived Wait/Test)* Free
  ```

  e.g.

  ```
  MPI_Psend[recv]_init (MPI_Pstart MPI_Pready MPI_Wait)* MPI_Request_free
  ```

- **MPI_PSEND_INIT** must be combined with **MPI_PRECV_INIT**.
- Matching rules are the same as for normal pt-to-pt communication. In doubt, order of initialization is used to break ties.
- Buffers must have **same** size for send and receive.
- Partitioning on sender/receiver may differ (as in the example).
- **PREADY must** be used to mark partition to be sent.
- **MPI_PARRIVED(request,partition,flag)** may be used to check
  - if partition is complete,
  - but does not complete the request (must be done with **MPI_TEST/MPI_WAIT**).
The new sessions model
World Model and Sessions Model

- **The World Model**
  - MPI_COMM_WORLD can be used between MPI_Init and MPI_Finalize
  - Exactly one call to MPI_Init and MPI_Finalize
  - Problem, if several independent software layers want to use MPI:
    - Each layer can duplicate MPI_COMM_WORLD using MPI_COMM_DUP()
    - But there is no rule on which layer calls MPI_Init and which one MPI_Finalize

- **The Sessions Model**
  - Each independent software layer xxx can initialize and finalize MPI, e.g., as follows:
    - As part of layer_xxx_init
      - MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &session);
      - MPI_Group_from_session_pset(session, "mpi://WORLD", &xxx_world_group);
      - MPI_Comm_create_from_group(xxx_world_group, "stringtag_xxx", MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &xxx_world_comm);
      - MPI_Group_free(&xxx_world_group);
    - As part of layer_xxx_finalize
      - MPI_Comm_free(&xxx_world_comm);
      - MPI_Session_finalize(&session);
  - Caution: MPI objects derived from different MPI Session handles shall not be intermixed with each other in a single MPI procedure call.

- An MPI application may use the World Model (not more than once) together with the Sessions Model (with several overlapping or non-overlapping sessions)

E.g., each independent software layer initiates its own session and communicator
Environment inquiry – implementation information (1)

Inquire start environment

- Predefined info object MPI_INFO_ENV (in the World Model) or info handle created with MPI_Info_create_env (in the Sessions Model) holds arguments from
  - mpiexec, or
  - MPI_COMM_SPAWN

New in MPI-3.0

New in MPI-4.0

see a few slides later

Slide ~232 in the HLRS MPI course
Sessions Model – Summary

- The Sessions Model → a method to init/finalize MPI within independent application components / software layers

New in MPI-4.0
New ways for hardware-based split of communicators
Splitting into smaller shared memory islands, e.g., NUMA nodes or sockets

- Subsets of shared memory nodes, e.g., one `comm_sm` on each socket with `size_sm` cores (requires also sequential ranks in `comm_all` for each socket!)

```c
MPI_Comm_split_type (comm_all, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm_large);
MPI_Comm_rank (comm_sm_large, &my_rank_sm_large);
MPI_Comm_size (comm_sm_large, &size_sm_large);
MPI_Comm_split (comm_sm_large, /*color*/ my_rank_sm_large / size_sm_large, 0, &comm_sm);
MPI_Win_allocate_shared (... , comm_sm, ...);
```

- Most MPI libraries have an non-standardized method to split a communicator into NUMA nodes (e.g., sockets): (see also Current support for split types in MPI implementations or MPI based libraries)
  - OpenMPI: choose split_type as OMPI_COMM_TYPE_NUMA
  - HPE: `MPI_Info_create (&info); MPI_Info_set(info, "shmem_topo", "numa"); // or "socket"
    MPI_Comm_split_type(comm_all, MPI_COMM_TYPE_SHARED, 0, info, &comm_sm);
  - mpich: `split_type=MPIX_COMM_TYPE_NEIGHBORHOOD, info_key= "SHMEM_INFO_KEY" and value= "machine", "socket", "package", "numa", "core", "hwthread", "pu", "l1cache", ..., or "l5cache"

- Two additional standardized split types:
  - `MPI_COMM_TYPE_HW_GUIDED`
  - `MPI_COMM_TYPE_HW_UNGUIDED`

- See also Exercise 3.
MPI_Neighbor communication: Examples / bug-fixes
Periodic MPI_NEIGHBOR_ALLTOALL in direction $d$
with 4 processes

This figure represents one direction $d$.
Of course, it is valid for any direction.

grey array entries are used only if periods[d] == non-zero in C or .TRUE. in Fortran.

© 2000-2022 HLRS, Rolf Rabenseifner

MPI course → Chap.9-(2) Virtual topologies → Neighborhood comm & MPI_BOTTOM
As if ...

After MPI_NEIGHBOR_ALLTOALL on a Cartesian communicator returned, the content of the recvbuf is as if the following code is executed:

```c
MPI_Cartdim_get(comm, &ndims);
for( /*direction*/ d = 0; d < ndims; d++) {
    MPI_Cart_shift(comm, /*direction*/ d, /*disp*/ 1, &rank_source, &rank_dest);
    MPI_Sendrecv(sendbuf[d*2+0], sendcount, sendtype, rank_source, /*sendtag*/ d*2,
                  recvbuf[d*2+1], recvcount, recvtype, rank_dest,    /*recvtag*/ d*2,
                  comm, &status); /* 1st communication in direction of displacement -1 */
    MPI_Sendrecv(sendbuf[d*2+1], sendcount, sendtype, rank_dest,    /*sendtag*/ d*2+1,
                  recvbuf[d*2+0], recvcount, recvtype, rank_source, /*recvtag*/ d*2+1,
                  comm, &status); /* 2nd communication in direction of displacement +1 */
}
```

The tags are chosen to guarantee that both communications (i.e., in negative and positive direction) cannot be mixed up, even if the MPI_SENDRECV is substituted by nonblocking communication and the MPI_ISEND and MPI_IRECV calls are started in any sequence.
Wrong implementations of periodic MPI_NEIGHBOR_ALLTOALL with only 2 and 1 processes

![Diagram showing the incorrect implementation of MPI_NEIGHBOR_ALLTOALL with only 2 and 1 processes.]

Wrong results with openmpi/4.0.1-gnu-8.3.0 and cray-mpich/7.7.6 with 2 and 1 processes:
Communication pattern of MPI_NEIGHBOR_ALLGATHER

- The send_buf is only one element, which is sent to the neighbor processes in all directions.
- The recv_buf represents one direction \( d \). Of course, this figure is valid for any direction.
- The green recv_buf elements are recvbuf[2\(d+0\)] and recvbuf[2\(d+1\)].
- Grey array entries are used only if periods[d] == non-zero in C or .TRUE. In Fortran.

Clarified in MPI-4.0.
Other small new MPI-4 features
Info handles revisited

- New nonblocking MPI_Comm_idup_with_info complementing blocking MPI_Comm_dup_with_info

- Use MPI_Info_get_string instead of deprecated MPI_Info_get_valuelen and MPI_Info_get

- MPI_Comm|File|Win_set_info + MPI_Comm|File|Win_get_info were clarified:
  - The MPI library may or may not set or recognize some (system specific) hints

Additional text in MPI-4.0
**MPI_Info Object**

- An **MPI_Info** is an opaque object that consists of a set of (key,value) pairs
  - Both key and value are **strings**
  - A **key** should have a **unique** name within one info handle
  - Several keys are reserved by standard / implementation
  - Portable programs may use **MPI_INFO_NULL** as the info argument
  - Vendor keys are also portable, may be ignored by other libraries
  - Several sets of vendor-specific keys may be used

- Allows applications to **pass environment-specific information**

- Allow applications to **provide assertions** regarding their usage of MPI objects and operations → to improve performance or resource utilization

- Several functions provided to manipulate the info objects

- Used in:
  - **Process Creation**,
  - **Window Creation**,
  - **MPI-I/O**,
  - **MPI_Comm_(i)dup_with_info**, **MPI_INFO_ENV**

- The key/value list returned by MPI_Comm/File/Win_get_info in the handle may differ from a those set by the application during Comm/File/Win creation or stored with MPI_Comm/File/Win_set_info: The MPI library may or may not set or recognize some (system specific) hints.

---

**Example:**

```c
MPI_Info info_noncontig;
MPI_Info_create(&info_noncontig);
MPI_Info_set(info_noncontig, "alloc_shared_noncontig", "true");
MPI_WinAllocateShared(…, info_noncontig, …);
```

**New in MPI-4.0:**

- Use MPI_Info_get_string instead of deprecated MPI_Info_get_valuelen and MPI_Info_get.
Wildcarding

- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter.

- With info assertions
  - "mpi_assert_no_any_source" = "true" and/or
  - "mpi_assert_no_any_tag" = "true"

  stored on the communicator using MPI_Comm_set_info(),
  - an MPI application can tell the MPI library that it will never use MPI_ANY_SOURCE and/or MPI_ANY_TAG on this communicator
  → may enable lower latencies.

- Other assertions:
  - "mpi_assert_exact_length" = "true" → receive buffer must have exact length
  - "mpi_assert_allow_overtaking" = "true" → message order need not to be preserved
Error handler revisited

• “MPI calls that are not related to any MPI objects are considered to be attached to the communicator MPI_COMM_SELF when using the World Model”
  – If you want to change the initial error handler
    • MPI_ERRORS_ARE_FATAL is the default
    • May be changed when calling mpirun / mpiexec
  then you must change it for both, MPI_COMM_WORLD and MPI_COMM_SELF

• New error handler MPI_ERRORS_ABORT
  – aborts only all processes of the related communicator

• Many other small additions / clarifications / …, see
  – MPI-4.0 Appendix B.1.2 Changes in MPI-4.0, items 4, 19-21, 26-27
Error Handling → “assembler for parallel computing”

2-level-concept with error codes and error classes, see MPI-3.1/MPI-4.0 Sect. 8.3-5/9.3-5

Most important aspects:
• The communication should be reliable (same rule as for processor and memory)
• If the MPI program is erroneous → no warranties:
  – by default: abort, if error detected by MPI library
  otherwise, unpredictable behavior
  i.e., error handler MPI_ERRORS_ARE_FATAL is the default
  – C/C++: MPI_Comm_set_errhandler (comm, MPI_ERRORS_RETURN);
    Fortran: call MPI_Comm_set_errhandler( comm, MPI_ERRORS_RETURN, ierr)
directly after MPI_INIT with both comm = MPI_COMM_WORLD and MPI_COMM_SELF, then
  • ierror returned by each MPI routine (except MPI window and MPI file routines)
  • undefined state after an erroneous MPI call has occurred
    (only MPI_Abort(…) should be still callable)
  – Exception: MPI-I/O has default MPI_ERRORS_RETURN
    • Default can be changed through MPI_FILE_NULL:
    • MPI_File_set_errhandler (MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL)
    • See MPI-3.1 Sect. 13.7, page 555 / MPI-4.0 Sect. 14.7, page 719, and course Chapter 7
  – MPI_ERRORS_ARE_FATAL aborts the process and all connected processes
  – MPI_ERRORS_ABORT aborts only all processes of the related communicator

© 2000-2022 HLRS, Rolf Rabenseifner
MPI course → Chap.7 Error Handling
Send-Receive in one routine

- MPI_Sendrecv & MPI_Sendrecv_replace
  - Combines the triple “MPI_Irecv + Send + Wait” into one routine

- Nonblocking MPI_Isendrecv & MPI_Isendrecv_replace
  - Whereas blocking MPI_Sendrecv was used to prevent
    - serializations and
    - deadlocks,
  - the nonblocking MPI_Isendrecv can be used, e.g.,
    to parallelize the existing communication calls in multiple directions
    \(\rightarrow\) e.g., to minimize idle times if only some neighbors are delayed
Use cases for nonblocking operations

- To prevent **serializations** and **deadlocks** (as if overlapping of communication with other communication)

> New in MPI-4.0

Now also described in the intro of MPI-4.0 Section 3.7 Nonblocking Communication

3.7 Nonblocking Communication

Nonblocking 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 the programmer must ensure that each send is matched with a receive in an order that avoids **deadlock**. For communication patterns that are determined only at run time, this is even more difficult. Nonblocking communication can be used to avoid this problem, allowing programmers to express complex and possibly dynamic communication patterns without needing to ensure that all sends and receives are issued in an order that prevents deadlock (see Section 3.5 and the discussion of “safe” programs). Nonblocking communication also allows for the **overlap** of communication with different communication operations, e.g., to prevent the **serialization** of such operations, and for the **overlap** of communication with computation. Whether an implementation is able to accomplish an effective (from a performance standpoint) overlap of operations depends on the implementation itself and the system on which the implementation is running. Using nonblocking operations **permits** an implementation to overlap communication with computation, but does not require it to do so.
Window creation or allocation

Four different methods

- Using existing memory as windows
  - MPI_Alloc_mem, MPI_Win_create, MPI_Win_free, MPI_Free_mem

- Allocating new memory as windows
  - MPI_Win_allocate

- Allocating shared memory windows – usable only within a shared memory node
  - MPI_Win_allocate_shared, MPI_Win_shared_query

- Using existing memory dynamically
  - MPI_Win_create_dynamic, MPI_Win_attach, MPI_Win_detach

MPI_Alloc_mem, MPI_Win_allocate, and MPI_Win_allocate_shared:

- Memory alignment must fit to all predefined MPI datatypes
  - alternative minimum alignment through info key "mpi_minimum_memory_alignment"
Lock/Unlock

- Does not guarantee a sequence
- agent may be necessary on systems without (virtual) shared memory
- Portable programs can use lock calls to windows in memory allocated **only** by `MPI_Alloc_mem`, `MPI_Win_allocate`, or `MPI_Win_attach` or `MPI_Win_allocate_shared`
- RMA completed after `MPI_Unlock` at both origin and target
- No concept of an exposure epoch
  - like window is permanently exposed
  - local load/stores must be enclosed in a local lock/unlock epoch

Arrow diagram:
- `Origin1`:
  - lock
  - put
  - unlock
- `Origin2`:
  - lock
  - put
  - unlock
- `Target`:
  - lock
  - get
  - unlock
- `window`
- `synchronization`
- `communication`
MPI_Request_free

• MPI_Request_free for *active* communication request:
  – Marks a request handle for deallocation
  – Deallocation will be done after *active* communication completion
  – May be used only for *active* send-request to substitute MPI_Wait, but *strongly discouraged* and dangerous when there is no other 100% guarantee that the send-buffer can be reused.
    • Active send handle is produced with MPI_I(s,b,r)send or MPI_(S,B,R)send_init + MPI_Start
      – *Should never be used* for *active* receive requests.

• Conclusion:
  MPI_Request_free really useful only for *inactive* persistent requests i.e., after such Loop(Start Wait/Test), i.e., not after Start
MPI_Cancel

- Marks a active nonblocking communication handle for cancellation.
- MPI_Cancel is a local call, i.e., returns immediately.
- **Subsequent call to MPI_Wait must return irrespective of the activities of other processes.**
- **Either the cancellation or the communication succeeds, but not both.**
- MPI_Test_cancelled(wait_status, flag [, ierror])
  - flag = true → cancellation succeeded, communication failed
  - flag = false → cancellation failed, communication succeeded

- **Comment:** Do not use it – may be reason for worse performance
- **MPI_Cancel of send requests is deprecated**
MPI_SIZEOF(…) – Fortran only API

• MPI_SIZEOF(…) was introduced in MPI-2.0
  – in combination with MPI_Type_match_size
  – as alternative to (recommended)
    • MPI_TYPE_CREATE_F90_INTEGER
    • MPI_TYPE_CREATE_F90_REAL
    • MPI_TYPE_CREATE_F90_COMPLEX
    to generate basic datatype handles
    for KIND-parameterized Fortran types

• MPI_SIZEOF is deprecated
Other changes …

- Tools chapter
  - New in MPI-4.0
  - MPI-4.0 Appendix B.1.2 *Changes in MPI-4.0*, items 30-32
Semantic changes & warnings
Chapter 18 – Semantic Changes and Warnings

18.1 Semantic Changes

This section describes semantics that have changed in a way that would potentially cause an MPI program to behave differently when using this version of the MPI Standard without changing the program's code.

18.1.1 Semantic Changes Starting in MPI-4.0

MPI_COMM_DUP and MPI_COMM_IDUP no longer propagate info hints from the input communicator to the output communicator. This behavior can be achieved using MPI_COMM_DUP_WITH_INFO and MPI_COMM_IDUP_WITH_INFO.

The default communicator where errors are raised when not involving a communicator, window, or file was changed from MPI_COMM_WORLD to MPI_COMM_SELF.

18.2 Additional Warnings

This section describes additional changes that could potentially cause a program that relies on the semantics described in a previous version of the MPI Standard to behave differently than with this version of MPI. The changes in this section are limited in scope and unlikely to impact most programs.

18.2.1 Warnings Starting in MPI-4.0

The limit for length of MPI identifiers was removed. Prior to MPI-4.0, MPI identifiers were limited to 30 characters (31 with the profiling interface). This limitation was initially introduced to avoid exceeding the limit on some compilation systems.

Annex B – Change-Log

18.x.1 Fixes to Errata in Previous Versions of MPI
Some future MPI-4.1 / 5.0 plans
Active Working Groups ➔ Important efforts

- Collective, Communicators, Context, Persistent, Partitioned, Groups, Topologies
  ➔ e.g. partitioned collectives, partitioned arrival / any / some
- Fault Tolerance
  ➔ new chapter on User Level Failure Mitigation / Fault Tolerance (ULFM/FT)
- Hardware-Topologies
  ➔ standardized levels for MPI_COMM_TYPE_HW_GUIDED
- Hybrid & Accelerator ➔ See next slide
- Languages ➔ side documents (other timeline), e.g., for other bindings (e.g. C++, Python)
- Remote Memory Access ➔ bug fixes ➔ See next slides
  ➔ completely new API allowing, e.g., offloading to the network interface controller (NIC)
  ➔ simplifying existing interface
  ➔ MPI_WIN.Shared_Query also for the shared memory-part of regular windows
- Semantic Terms
  ➔ apply them to RMA; differentiation between a procedure and a specific call to it
- Progress ➔ See next slides
- Sessions
  ➔ Adding functionality for features currently supporting only for the World Model
  ➔ e.g. dynamic resources, buffered send, …
- Tools ➔ QMPI + handling introspection and debugging interface

See [https://www.mpi-forum.org/mpi-41/](https://www.mpi-forum.org/mpi-41/)
Hybrid & Accelerator

https://github.com/mpiwg-hybrid/hybrid-issues/wiki

• **Active Topics**
  • Continuations proposal #6
  • Clarification of thread ordering rules #117
  • Integration with accelerator programming models:
    – Accelerator info keys #3
    – Stream/Graph Based MPI Operations #5
    – Accelerator bindings for partitioned communication #4
    – Partitioned communication buffer preparation (shared with Persistence WG) #264
  • Asynchronous operations #585
Errata to MPI shared memory
Errata to MPI shared memory

• Problem with MPI-3.0 to MPI-4.0:
The role of assertions in RMA synchronization used for direct shared memory accesses (i.e., without RMA calls) is not clearly defined!
  – Detected & communicated about March 01, 2015
  – Implications for all RMA function on a shared memory window:
    • Users: Always use assert=0
    • Implementors: Always ignore the assert values
    • MPI Forum: Specify valid assertions for shared memory windows

• MPI_Win_sync + any other process-to-process synchronization
  – Rules are unclear
  – AtoUsers in MPI-3.1/MPI-4.0, page 456 lines 22-29/ page 613 line 46 – 614 line 5
  – And through Example MPI-3.1/MPI-4.0, pages 468f/626f, Exa. 11.21/12.21
  → See next slides  (skip them)

General MPI shared memory synchronization rules
(based on MPI-3.1/MPI-4.0, MPI_Win_allocate_shared, page 408/560, lines 43-47/22-26: “A consistent view ...”)

Defining Proc 0 Proc 1
Sync-from → Sync-to

being

MPI_Win_post\(^1\) → MPI_Win_start\(^1\)
or
MPI_Win_complete\(^1\) → MPI_Win_wait\(^1\)
or
MPI_Win_fence\(^1\) ← MPI_Win_fence\(^1\)
or
MPI_Win_sync
Any-process-sync\(^2\) → Any-process-sync\(^2\)
or
MPI_Win_unlock\(^1\) → MPI_Win_lock\(^1\)

and A, B, C are shared variables
and having ...

then it is guaranteed that ...

A=val_1
Sync-from → Sync-to
load(A)
⇒ ... the load(A) in P1 loads val_1
(this is the write-read-rule)

load(B)
Sync-from → Sync-to
B=val_2
⇒ ... the load(B) in P0 is not affected by the store of val_2 in P1
(read-write-rule)

C=val_3
Sync-from → Sync-to
C=val_4
load(C)
⇒ ... that the load(C) in P1 loads val_4
(write-write-rule)

\(^1\) Must be paired according to the general one-sided synchronization rules.
\(^2\) ”Any-process-sync” may be done with methods from MPI (e.g. with send→recv as in MPI-3.1/MPI-4.0 Example 11/12.21, but also with some synchronization through MPI shared memory loads and stores, e.g. with C++11 atomic loads and stores).
\(^3\) No rule for MPI_Win_flush (according current forum discussion)
“Any-process-sync” & MPI_Win_sync on shared memory

- If the shared memory data transfer is done without RMA operation, then the synchronization can be done by other methods.
- This example demonstrates the rules for the unified memory model if the data transfer is implemented only with load and store (instead of MPI_Get or MPI_Put) and the synchronization between the processes is done with MPI communication (instead of RMA synchronization routines).

**Process A**

```fortran
MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK, win)
DO ...
  X = ...
  MPI_F_SYNC_REG(X)
  MPI_Win_sync(win)
  MPI_Send
  MPI_Recv
  MPI_Win_sync(win)
  MPI_F_SYNC_REG(X)
  MPI_Send
  MPI_Win_sync(win)
  MPI_F_SYNC_REG(X)
  MPI_Rececv
  local_tmp = X
  MPI_Win_sync(win)
  MPI_F_SYNC_REG(X)
  MPI_Send
  MPI_Win_sync(win)
  MPI_F_SYNC_REG(X)
  MPI_Rececv
END DO

MPI_WIN_UNLOCK_ALL(win)
```

**Process B**

```fortran
MPI_WIN_LOCK_ALL(MPI_MODE_NOCHECK, win)
DO ...
  MPI_Win_sync(win)
  MPI_Send
  MPI_Rececv
  MPI_Win_sync(win)
  MPI_F_SYNC_REG(X)
  MPI_Send
  MPI_Win_sync(win)
  MPI_F_SYNC_REG(X)
  MPI_Rececv
END DO
```

- A new value is written in X.
- Message telling that X is filled.
- Message telling that X is read out and can be refilled.
- At begin of next iteration: Next write of X.
- Is missing in MPI-3.1/MPI-4.0, pages 468f/626f, Exa. 11/12.21 (i.e., page 469/627, line 31/14).

**See Exercise 3**

- For MPI_WIN_SYNC, a passive target epoch is established with MPI_WIN_LOCK_ALL.
- Data exchange in this direction, therefore MPI_Win_sync is needed in both processes: Write-read-rule.
- MPI_WIN_SYNC acts only locally as a processor-memory-fence.
- 2nd pair of MPI_Win_sync is needed to guarantee the read-write-rule.

X is part of a shared memory window and should be the same memory location in both processes.

A new value is written in X.
Progress text / functionality update → delayed until MPI-5
What is progress

- To internally finish a started operation
  - the process that started the operation, and/or other related processes may need to make **progress** from the viewpoint of the underlying MPI system.
  - Example:
    - **Process 1**: Operation MPI receive, e.g., started with `MPI_Recv` or `MPI_Irecv`
    - **Process 0**: Is other related process
      - Called `MPI_Bsend`, already returned,
      - but data still buffered (from the viewpoint of the underlying MPI system)
    - That process 1 can internally finish the receive operation, process 0 needs to make progress, i.e., to really send the buffered data

![Diagram](attachment:image.png)

- Which rules apply that process 0 provides progress?

See next slide
Use cases for nonblocking operations

- Real overlapping of
  - several communications
  - communication and computation
General progress rule of MPI

- MPI is mainly defined in a way that **progress** on communication (and ...) is **required only during MPI procedure calls**.
- But then, progress is required
  - for **all** outstanding (incomplete/nonblocking) communications
  - together with operation of the current communication (…) procedure call.
- See, e.g., in MPI-4.0
  - Sect. 3.5, page 54, and 3.7.4, page 75; Paragraphs “Progress”, esp. progress of repeated MPI_Test, p.75, page 38-40
  - Sect. 3.8.1 and 3.8.2 about MPI_(I)(M)PROBE
  - Sect. 3.8.4 Cancel, esp. page 94 lines 8-16 & MPI_Finalize Example 11.6, page 496-48
  & MPI_Session_finalize, esp. page 502 and Example 11.8 on page 503
  - Sect. 4.2.2 MPI_Parrived: Same progress rule as for repeated MPI_Test, see page 111
  - Sect. 5.12: Nonblocking collectives: Same rules as for nonblocking pt-to-pt
  - Sect. 12.7.3: Progress with one-sided communication, especially the **rationale at the end**
  - Sect. 11.6: MPI and Threads
  - Sect. 14.6.3: Progress with MPI-I/O
- **Non** of these rules require progress outside of called MPI routines,
  - But MPI_Test and each MPI routine that blocks must do progress on any ongoing (i.e. nonblocking) communication
- **Additional progress**
  - By several calls to MPI_Test(), which enables progress
  - Use non-standard extensions to switch on asynchronous progress
    - E.g., with MPICH: export MPICH_ASYNC_PROGRESS=1
      Implies a helper thread and MPI_THREAD_MULTIPLE (?)

© 2000-2022 HLRS, Rolf Rabenseifner

Slide ~187 in the HLRS MPI course
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.

An MPI procedure is **local** if it is not *non-local*.

- **Local MPI procedures may be implemented as “weak local”:**
  - To complete its work locally, it may require an *unspecific* MPI call on another process

- **Examples** (always tested with **large** messages):
  - **Bsend** is local.
    - **Corresponding MPI_Receiv** may require progress in the sending process  may be blocked until the sending process calls another unspecified MPI procedure
  - **Rsend** is local, since the corresponding MPI_(I)Recv must already be called.
    - But the MPI_Rsend may require progress in the receiving process  may be blocked until the receiving process calls another unspecified MPI routine

**Experiments, see**

MPI/tasks/C/Ch18/progress-test-bsend.c + progress-test-bsend-output.txt

**MPI/tasks/C/Ch18/progress-test-rsend.c + progress-test-rsend-output.txt**
Possible consequences with MPI_Bsend

```c
MPI_Buffer_attach(...)
MPI_Barrier(...); // Only for starting the experiment together
for (iter=1; iter <=3; iter++) {
    if (my_rank>0) MPI_Bsend(..., my_rank-1, ...);
    if (my_rank<numprocs-1) MPI_Bsend(..., my_rank+1, ...);
    sleep(...); // some small delay
    if (my_rank>0) MPI_Recv(..., my_rank-1, ...);
    if (my_rank<numprocs-1) MPI_Recv(..., my_rank+1, ...);
    sleep(20); // simulating 20 sec of numerical work
}
MPI_Barrier(...); // Not needed because the blocking non-collective
MPI_Buffer_detach(...) buffer detach would cause the same result
```

Expected behavior with independent progress

- **Process 0**: Delay, some numerics → MPI_Bsend
- **Process 1**: some numerics → delay, some numerics
- **Process 2**: some numerics → delay, some numerics → MPI_Recv

Real behavior without independent progress

- **Process 0**: delay, some numerics, delay, some numerics, delay, some numerics, delay
- **Process 1**: some numerics, delay, some numerics, delay, some numerics, delay, some numerics
- **Process 2**: some numerics, delay, some numerics, delay, some numerics, delay

**Caution**: 2nd message is sent before 1st message is delivered → double buffer space is needed

Solution (without independent progress): add buffer detach/attach before numerics

- **Process 0**: Delay, some numerics, delay, some numerics, delay, some numerics, delay, some numerics
- **Process 1**: some numerics, delay, some numerics, delay, some numerics, delay, some numerics
- **Process 2**: some numerics, delay, some numerics, delay, some numerics, delay

The programs and protocols contain also a 2nd experiment:
- It is without the “small delays” and reports 120 sec vs. 60 sec, i.e., **two times slower** without detaching + re-attaching the buffer after each comm. step

© 2000-2022 HLRS, Rolf Rabenseifner

MPI course → Chap. 18 Best Practice → Progress

7) The receive of the buffered message is delayed until another unspecified MPI call in the sending process can implement the data transfer: MPI_Recv or MPI_Buffer_detach (2nd example).
MPI Progress Rule

• MPI library must provide the following **minimal** progress:
  
  1. **Blocked MPI procedure calls** must provide progress on all enabled MPI operations.
  2. Test procedures will eventually return flag=true once the matching operation has been started:
     • MPI_Test, MPI_Iprobe, MPI_Iprobe,
     • MPI_Request_get_status, MPI_Win_test (specification is missing in MPI-3.1/MPI-4.0, may be clarified in MPI-4.1)
     • MPI_Parrived (new procedure in MPI-4.0)
  3. MPI finalization must guarantee that all required progress will be provided before the process exits.
  4. Further rules, e.g., on collectives, I/O, …

• **A blocked MPI procedure call** can be:

  – Non-local MPI procedure
    (e.g., MPI_Send, MPI_Recv, MPI_Wait for a receive/send request handle)
    **waits** for a specific semantically-related MPI call on another MPI process
    (e.g., MPI_(I)Recv, MPI_(I)Send, MPI_(I)Send / MPI_(I)Recv)
  
  – Local MPI procedure (see also references 3.)
    (e.g., MPI_Rsend)
    **waits** for some unspecific MPI call on another MPI process
    (e.g., any other MPI call that must do progress → see above 1. or 2. or 3 but it may be also a related routine, e.g., the MPI_Wait in the example).

Of course, more progress is always allowed! E.g., through a progress thread 😊

References in MPI-4.0:

1. Sect. 3.5, page 54, and 3.7.4, page 75. Paragraphs "Progress".
   Sect. 11.6: MPI and Threads.
   Sect. 12.7.3: Progress with one-sided communication, especially the rationale at the end.

2. Sect. 3.7.4 on MPI_Test, esp. p.75
   Sect. 3.8.1 & 3.8.2: MPI_(I)(M)PROBE, 
   Sect. 4.2.2 MPI_Parrived p. 111

3. Sect. 3.8.4 Cancel, p. 94 lines 8-16.
   MPI_Finalize Example 11.6, p. 496
   MPI_Session_finalize, esp. p. 502
   and Example 11.8 , p. 804

   Sect. 14.6.3: MPI-I/O

© 2000-2022 HLRS, Rolf Rabenseifner
MPI course → Chap. 18 Best Practice → Progress

Slide ~592 in the HLRS MPI course

Slide 61 / 68
Progress / weak local – summary

→ In principle, program as if your MPI library provides independent progress
→ But weak progress can lead to very unexpected performance behavior
→ Hopefully fixed in many MPI libraries
→ MPI_THREAD_MULTIPLE instead of …_SINGLE usually makes no difference
  – Test with progress-test-bsend_init.c & progress-test-bsend_init-thread-multiple.c
→ Nevertheless, make sure that your programs are correct & portable, e.g.:

Back to our loop( bsend left+right; recv left+right ) example:
Only by receiving this (response) message, process 2 logically knows now (and not earlier) that its 1st message is received. Therefore here (still without this knowledge), process 2 must have attached enough buffer space for both the 1st and 2nd message together. This logical consideration is independent of weak or strong progress.
Weighted Cartesian Topologies
The problems

1. All MPI libraries provide the necessary interfaces 😊😊😊, but **without** re-numbering in nearly all MPI-libraries 😞😞😞

   - **You may substitute MPI_Cart_create() by Bill Gropp’s solution**

2. The existing MPI-3.1 and MPI-4.0 interfaces are not optimal
   – for cluster of ccNUMA node hardware,
     - **We substitute MPI_Dims_create() + MPI_Cart_create()**
       by **MPIX_Cart_weighted_create(… MPIX_WEIGHTS_EQUAL …)**
   – nor for application specific data mesh sizes or direction-dependent bandwidth
     - by **MPIX_Cart_weighted_create( … weights ….)**

3. Caution: The application must be prepared for rank re-numbering
   - All communication through the newly created Cartesian communicator with re-numbered ranks!
   - One must not load data based on MPI_COMM_WORLD ranks!
Examples

• Application topology awareness
  – 2-D example with 12 MPI processes and data mesh size 1800x580
    • MPI_Dims_create \rightarrow 4x3
    • data mesh aware \rightarrow 6x2 processes

• Hardware topology awareness
  – 2-D example with 25 nodes x 24 cores and data mesh size 3000x3000
    • MPI_Dims_create \rightarrow 25 x 24
    • Hardware aware \rightarrow 30 x 20
      = (5 nodes x 6 cores) \times (5 nodes x 4 cores)
Other small functionality / changes
### Environmental inquiries

- **C:** `MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &p, &flag)`
  - Will return in `p` a pointer to an int containing the `attribute_val`

- **Fortran:** `MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, attribute_val, flag, ierror)`

- **Python:** `attribute_val = MPI.COMM_WORLD.Get_attr(keyval)`

  - with `keyval =`  
    - `MPI_TAG_UB`
      - returns upper bound for tag values in `attribute_val`
      - must be at least 32767
    - `MPI_HOST`
      - returns host-rank (if exists) or `MPI_PROC_NULL` (if there is no host)
    - `MPI_IO`
      - returns `MPI_ANY_SOURCE` in `attribute_val` (if every process can provide I/O)
    - `MPI_WTIME_IS_GLOBAL`
      - returns 1 in `attribute_val` (if clocks are synchronized), otherwise, 0

**Examples:** see MPI-3.1, Sect. 17.2.7, page 664, line 43 – page 665, line 13 or MPI-4.0, Sect. 19.3.7, page 852, line 29-47
Summary

MPI-4.0 has a lot for better service / better performance

- Large counts
- Sessions Model
- Better error handling
- More consistent standard:
  - Revisited terms & semantics
  - New introduction for nonblocking operations
  - Removed / Semantic changes & warnings / Errata

Outlook on MPI-4.1 / 5.0