MPI in Small Bites

PPCES 2024

HPC.NRW Competence Network

THE COMPETENCE NETWORK FOR HIGH PERFORMANCE COMPUTING IN NRW.

MPI & Threads – Hybrid Programming

HPC.NRW Competence Network

MPI in Small Bites

Hybrid Programming: Motivation

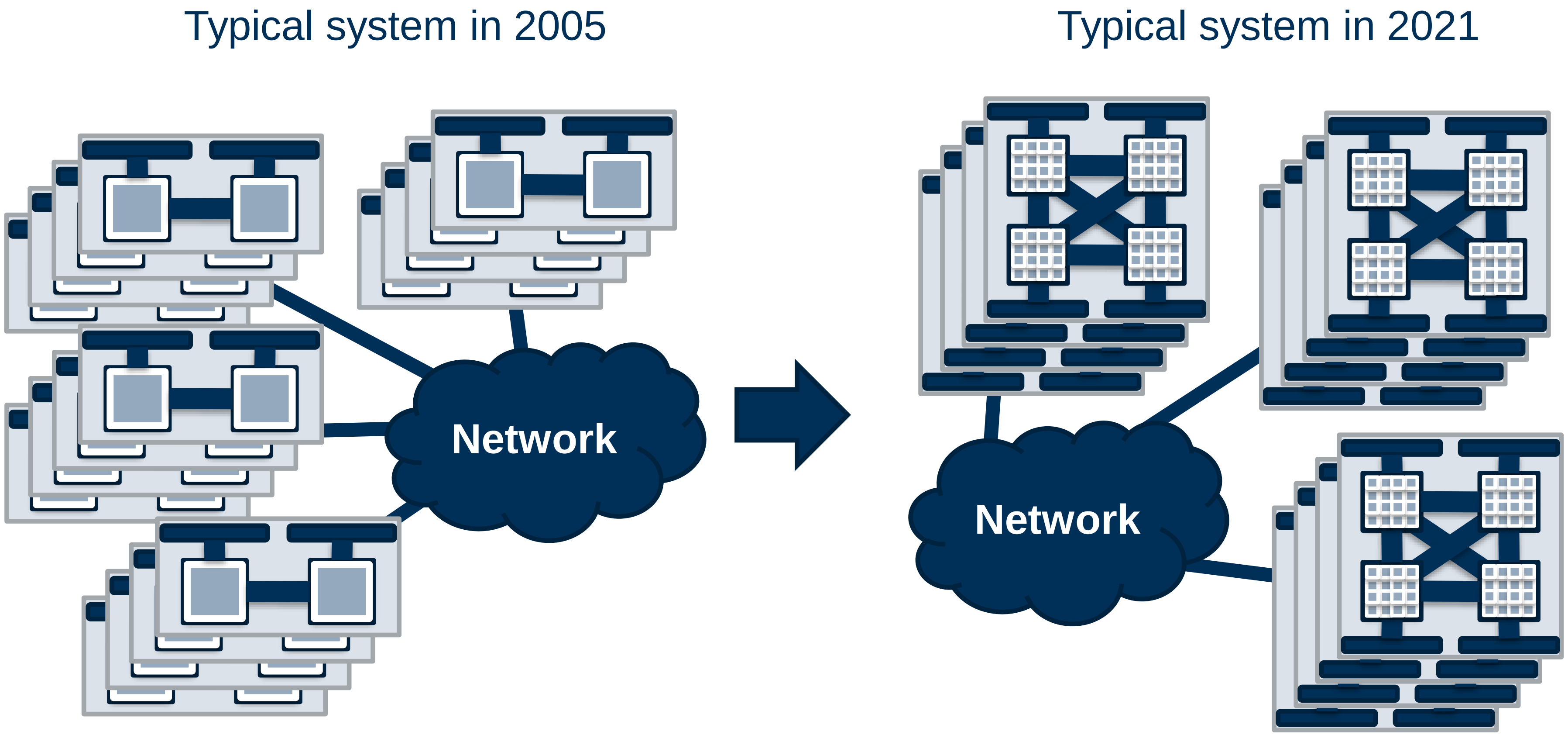
HPC.NRW

- MPI is sufficiently abstract so it runs perfectly fine on a single node:
  - it doesn’t care where processes are located as long as they can communicate
  - message passing implemented using shared memory and IPC
    - all details hidden by the MPI implementation;
    - usually faster than sending messages over the network;
  - but…
- … this is far from optimal:
  - MPI processes are separate (heavyweight) OS processes
  - portable data sharing is hard to achieve
  - lots of program control / data structures have to be duplicated (uses memory)
  - reusing cached data is practically impossible

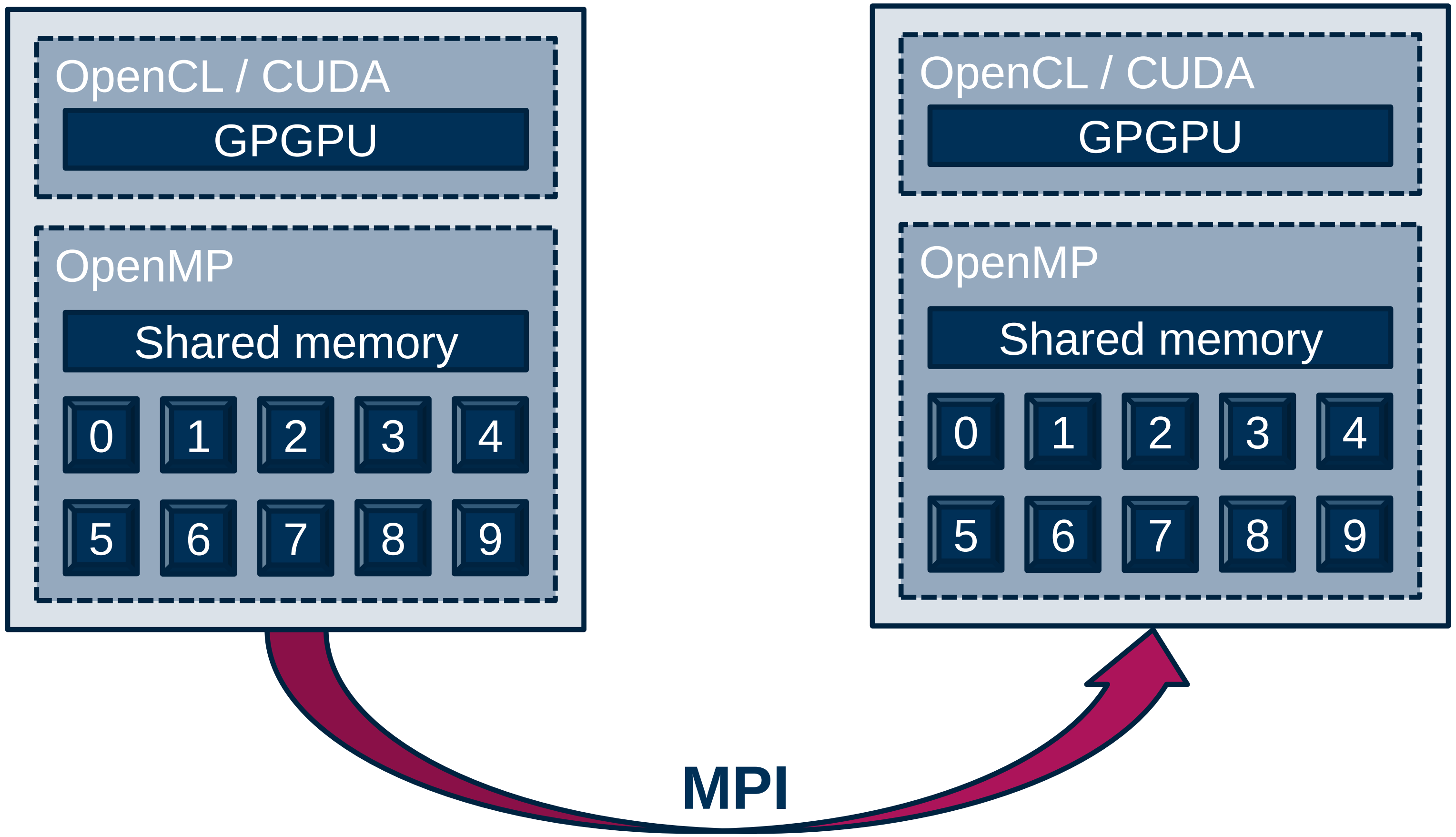
Hybrid Programming: Motivation

HPC.NRW

- Increasing number of cores per node
  - Increasingly complex nodes – many cores, GPUs, Intel® Xeon Phi™, etc.



Hierarchical mixing of different programming paradigms



MPI – Threads Interaction

HPC.NRW

- Most MPI implementation are threaded (e.g., for non-blocking requests) but not thread-safe.

- Four levels of threading support in increasing order:

Level identifier	Description
`MPI_THREAD_SINGLE`	Only one thread may execute
`MPI_THREAD_FUNNELED`	Only the main thread may make MPI calls
`MPI_THREAD_SERIALIZED`	Only one thread may make MPI calls at a time
`MPI_THREAD_MULTIPLE`	Multiple threads may call MPI at once with no restrictions

- All implementations support MPI_THREAD_SINGLE, but some do not support MPI_THREAD_MULTIPLE.

Initialization MPI when using Threads

HPC.NRW

- Initialise MPI with thread support:

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

MPI_INIT_THREAD (required, provided, ierr)
```

  - **required** specifies what thread level support one requires from MPI
  - **provided** is set to the actual thread level support provided
    - could be lower or higher than the required level – **always check**!
  - **MPI_Init** – equivalent to **required = MPI_THREAD_SINGLE**

- The level of thread support cannot be changed later
- The thread that calls MPI_Init_thread becomes the main thread

Query Functions

HPC.NRW

– Obtain the provided level of thread support:

```
MPI_Query_thread (int *provided)
```

– If MPI was initialised by **MPI_Init_thread**, then **provided** is set to the same value as the one returned by the initialisation call
– If MPI was initialised by **MPI_Init**, then **provided** is set to an implementation specific default value

– Find out

```
MPI_Is_thread_main (int *flag)
```

– **flag** set to **true** if the current thread is the main thread

MPI + OpenMP

- The most common approach to hybrid programming
  - Coarse-grained parallelisation with MPI
  - Fine-grained loop or task parallelisation with OpenMP

- Different MPI implementations provide varying degree of support for threaded programs
  - MPI_THREAD_MULTIPLE often not implemented completely for all transports
  - Performance decrease due to locking overhead

- Safest and most portable approach: Call MPI from the main thread only (and outside any OpenMP parallel region) → MPI_THREAD_FUNNELED

Simple: Iterative processing with MPI only

HPC.NRW

```
double data[], localData[];

for (int iter = 0; iter < maxIters; iter++) {

   MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);


   for (int i = 0; i < count; i++)
      localData[i] = exp(localData[i]);

   MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

}
```

Safe: MPI called outside any OpenMP parallel region

HPC.NRW

```
double data[], localData[];

for (int iter = 0; iter < maxIters; iter++) {

   MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

   #pragma omp parallel for
   for (int i = 0; i < count; i++)
      localData[i] = exp(localData[i]);

   MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

}
```

Advanced: MPI called by the master OpenMP thread only

```
double data[], localData[];
#pragma omp parallel
for (int iter = 0; iter < maxIters; iter++) {
   #pragma omp master
   MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);
   #pragma omp barrier
   #pragma omp for
   for (int i = 0; i < count; i++)
      localData[i] = exp(localData[i]);
   #pragma omp master
   MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);
   #pragma omp barrier
}
```

Adventurous: MPI called by a single OpenMP thread at a time

HPC.NRW

```
MPI_Init_thread(&argc, &argc, MPI_THREAD_SERIALIZED, &provided);

double data[], localData[];
#pragma omp parallel
for (int iter = 0; iter < maxIters; iter++) {
   #pragma omp single
   MPI_Scatter(data, count, MPI_DOUBLE,
               localData, count, MPI_DOUBLE,
               0, MPI_COMM_WORLD);
   #pragma omp for
   for (int i = 0; i < count; i++)
      localData[i] = exp(localData[i]);
   #pragma omp single
   MPI_Gather(localData, count, MPI_DOUBLE,
              data, count, MPI_DOUBLE,
              0, MPI_COMM_WORLD);
}
```

Addressing in Hybrid Programs

HPC.NRW

- MPI was not designed initially with multithreading in mind
  - Single rank (end-point) per process per communicator
  - Addressing individual threads is tricky (and mostly hacky)
- MPI and OpenMP IDs live in orthogonal spaces
  - MPI rank $\in$ [0, #procs-1] **MPI_Comm_rank()**
  - OpenMP thread ID $\in$ [0, #threads-1] **omp_get_thread_num()**
  - Hybrid

Field	Value source	Remark
source rank	Sender process rank	Automatically copied, no control over it
destination rank	user-supplied	Only one rank per process
tag	user-supplied	Free to choose
communicator	user-supplied	Multiple communicators possible

Addressing in Hybrid Programs

HPC.NRW

- Tags as thread IDs
  - Each MPI message carries a tag with at least 15 bits of user-supplied data

- Simple idea: use tag value to address individual threads
  - (+) straightforward to implement
  - (+) very large number of threads per process addressable
  - (-) not possible to further differentiate the messages
  - (-) no information about the sending thread retained

Addressing in Hybrid Programs

- Tags as thread IDs
  - Each MPI message carries a tag with at least 15 bits of user-supplied data

- Better idea: multiplex destination thread ID with tag value
  - e.g., 7 bits for tag value (0..127) and 8 bits for thread ID (0..255)
  - (+) still possible to differentiate the messages
  - (-) wildcard receives not trivial to implement
  - (-) no information about the sending thread retained

Addressing in Hybrid Programs

- Tags as thread IDs
  - Each MPI message carries a tag with at least 15 bits of user-supplied data
- Even better idea: multiplex source and destination thread IDs with tag value
  - suitable for MPI implementations that allow more than 15 bits for tag value
    - Open MPI and Intel MPI both allow tag values from 0 to 2^{31}-1
  - (+) still possible to differentiate the messages
  - (+) information about the sending thread retained
  - (-) wildcard receives not trivial to implement
  - (-) not portable to MPI implementations with smaller tag space

Multiplex source and destination thread IDs with tag value

HPC.NRW

```
#define MAKE_TAG (tag,stid,dtid) \
    (((tag) << 16) | ((stid) << 8) | (dtid))

// Send data to drank:dtid with tag mytag

MPI_Send(data, count, MPI_FLOAT, drank,
         MAKE_TAG(mytag, omp_get_thread_num(), dtid),
         MPI_COMM_WORLD);

-----------------------------------------------------------

// Receive data from srank:stid with a specific tag mytag

MPI_Recv(data, count, MPI_FLOAT, srank,
         MAKE_TAG(mytag, stid, omp_get_thread_num()),
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Multiplex source and destination thread IDs with tag value

HPC.NRW

```
#define GET_TAG(val) \
    ((val) >> 16)
#define GET_SRC_TID(val) \
    (((val) >> 8) & 0xff)
#define GET_DST_TID(val) \
    ((val) & 0xff)

// Wildcard receive from srank:stid with any tag

MPI_Probe(srank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
if (GET_SRC_TID(status.MPI_TAG) == stid &&
    GET_DST_TID(status.MPI_TAG) == omp_get_thread_num())
{
    MPI_Recv(data, count, MPI_FLOAT, srank, status.MPI_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

MPI_Probe and multi-threading

HPC.NRW

- Beware of possible data races:
  - messages, matched by **MPI_Probe** in one thread, can be received by a matching receive in another thread, stealing the message from the first one
    - Needs very good care on the side of the thread handling
  - Problem solved in MPI-3 with **MPI_Mprobe** and **MPI_Mrecv**

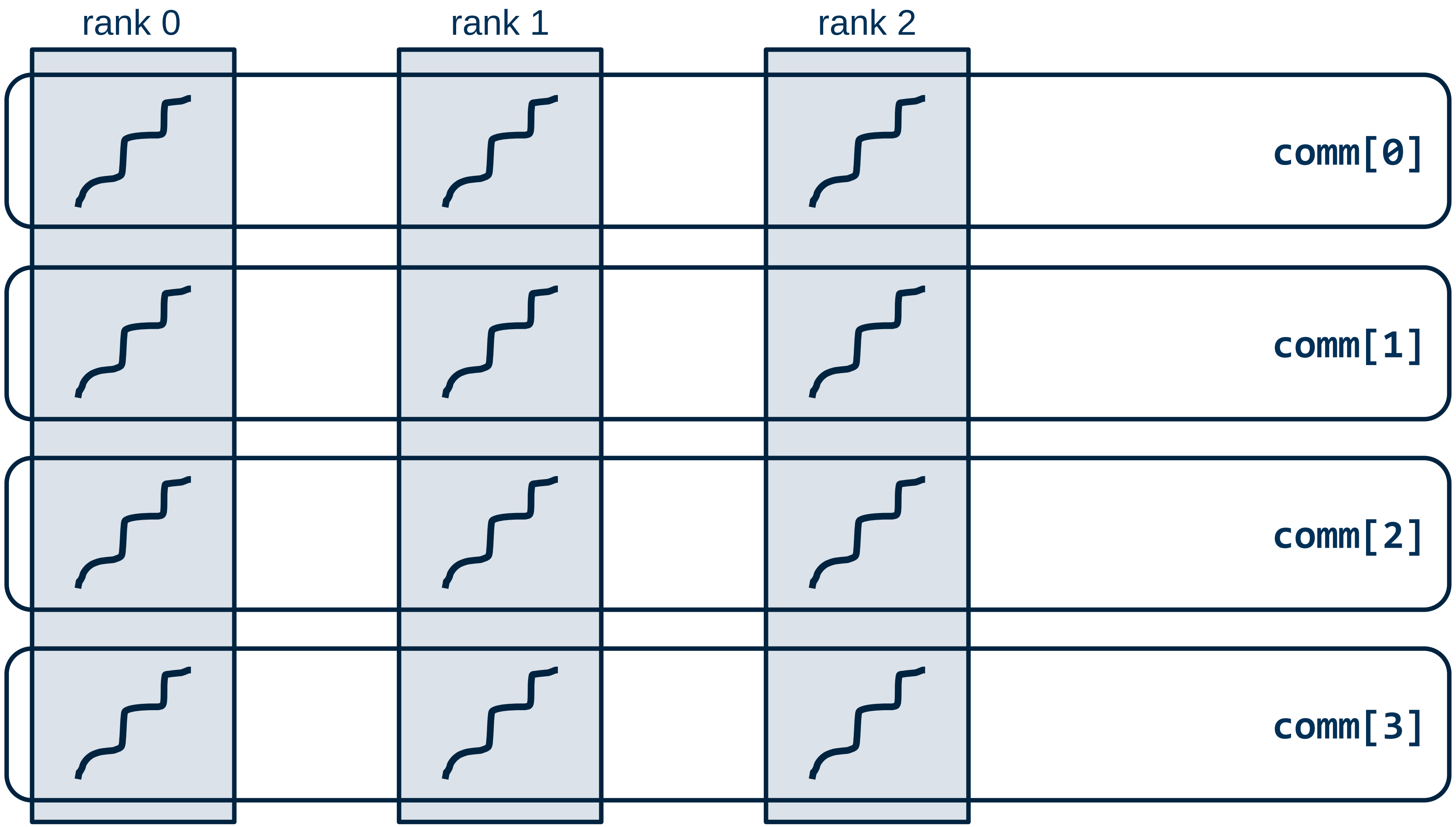
```
MPI_Mprobe (int source, int tag, MPI_Comm comm, MPI_Message *message,
            MPI_Status *status)

MPI_Mrecv (void* buf, int count, MPI_Datatype datatype,
           MPI_Message *message, MPI_Status *status)
```

- **MPI_Mprobe** removes the matched message from the matching process
  - Returns a **message handle** to reference the matched message in future receives
- **MPI_Mprobe** (or **MPI_Improbe**) used to received a message via message handle

Use of multiple communicators

HPC.NRW



Use of multiple communicators

HPC.NRW

```
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
MPI_Comm comm[nthreads], tcomm;

#pragma omp parallel private(tcomm) num_threads(nthreads)
{
   MPI_Comm_dup(MPI_COMM_WORLD, &comm[omp_get_thread_num()]);
   tcomm = comm[omp_get_thread_num()];
--------------------------------------------------------------
   // Sender
   MPI_Send(data, count, MPI_FLOAT, omp_get_thread_num(),
            drank, comms[dtid]);
--------------------------------------------------------------
   // Receiver
   MPI_Recv(data, count, MPI_FLOAT, stid, srank, tcomm,
            &status);
--------------------------------------------------------------
   MPI_Comm_free(&comm[omp_get_thread_num()]);
}
```

Summary of Caveats

HPC.NRW

- Race-condition possible between MPI_Probe and corresponding MPI_Recv
  - Use of “Matched Probe and Receive”
- MPI provides no way to address specific threads in a process
  - clever use of message tags
  - clever use of many communicators
- Thread-safe MPI implementations often perform worse than non-thread-safe
  - Additional synchronisation overhead