MPI: Part III

AQTIVATE Training Workshop I

Dirk Pleiter

CST | EECS | KTH

December 2023

Overview

Communicators

Derived Data Types

One-Sided Communication

Hybrid Parallelisation

Content

Communicator Creation: Duplication

- Create a new communicator based on an existing communicator

```
int MPI_Comm_dup(
  MPI_Comm comm,      /* Existing communicator */
  MPI_Comm *newcomm   /* New communicator */
);
```

- The new communicator
  - Refers to the same set of processes and leaves rank ordering and other associated values unchanged
  - Defines a new context for message communication (used, e.g., by parallel libraries)

Communicator Creation: Partition (1/2)

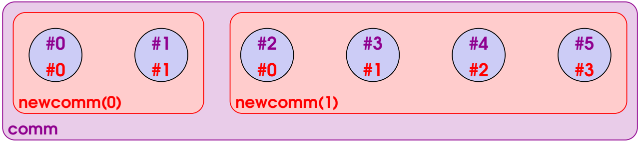
- MPI allows to partition the processes in disjoint subgroups in a flexible manner using

```c
int MPI_Comm_split(
 MPI_Comm comm,         /* Existing communicator */
 int color,             /* Control of subset assignment */
 int key,               /* Control of rank assigment */
 MPI_Comm *newcomm      /* New communicator */
);
```

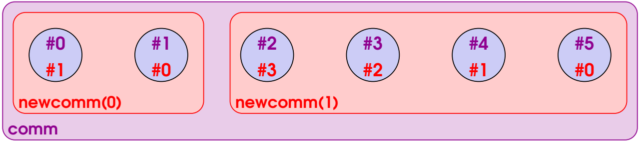
- For each value of `color` a different subgroup (and communicator) is created
- Within each subgroup, the processes are ranked in the order defined by `key`
  - The rank of the old group is used by tie breaking

Communicator Creation: Partition (2/2)

- Example #1: `color` = (`rank` < 2) ? 0 : 1; `key` = 0



- Example #2: `color` = (`rank` < 2) ? 0 : 1; `key` = `-rank`



Inter-Communicators (1/2)

- Communications between different subgroups (partitions) can be established using **inter-communicators**, e.g. using

```
int MPI_Intercomm_create(
 MPI_Comm local_comm,      /* Local communicator */
 int local_leader,         /* Rank of local leader */
 MPI_Comm peer_comm,       /* Peer communicator */
 int remote_leader,        /* Rank of remote leader */
 int tag,                  /* Tag */
 MPI_Comm *newintercomm    /* Inter-communicator */
);
```

- `local_comm` and `local_leader` identify lead process within local subgroup
- `peer_comm` must be a group to which both the local and remote leader process belong to
- The remote leader is identified within the `peer_comm` through `peer_leader`

Inter-Communicators (2/2)

- The syntax of point-to-point and collective communication is the same for both inter- and intra-communication
  - A target process is addressed by its rank in the remote group

Process Topologies

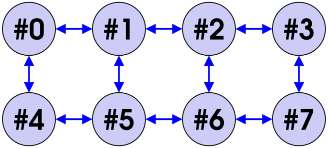
- Communicators can be used to define a **virtual network topology** that defines the neighbourship relations between the processes within the given group
- Benefits of such virtual topologies:
  - Allow to express communication patterns in a more natural way
  - Provide middleware with hints about target communication patterns
    - MPI may use this information to select a better mapping of processes to underlying **physical network topology** (if user allows for rank re-ordering)

Process Topologies: Cartesian (1/2)

- With a Cartesian communicator processes are assumed to be arranged on a one or higher-dimensional mesh or torus

```
int MPI_Cart_create(
 MPI_Comm comm,           /* Existing communicator */
 int ndims,               /* Number of dimensions of grid */
 const int dims[],        /* Grid size */
 const int periods[],     /* True (false) if grid is (not) periodic */
 int reorder,             /* Allow to reorder or not */
 MPI_Comm *comm_cart      /* New communicator */
);
```

- Example with 8 processes and `ndims` $=2$, `dims` $=[4,2]$, `periods` $=[0,0]$ and `reorder` $=0$:



Process Topologies: Cartesian (2/2)

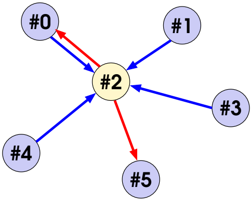
- MPI provides a (convenience) function to compute source and destination to communication with the grid:

```
int MPI_Cart_shift(
 MPI_Comm comm_cart, /* Cartesian communicator */
  int direction,      /* Communication direction */
  int disp,           /* Displacement (sign determins direction) */
  int *source,        /* Returned source rank */
  int *dest           /* Returned destination rank */
);
```

  - The returned `source` and `dest` ranks can be directly used as input for `MPI_Sendrecv`

Process Topologies: Graph (1/2)

- For graph-based algorithms, MPI allows defining communicator with each process specifying each of its incoming and outgoing (adjacent) edges in the logical communication graph
- Weights may be provided to influence the rank reordering
  - `MPI_UNWEIGHTED` may be used in case no weights are known
- Example:
  - `indegree` $=$ 4
    `sources` $= [0, 1, 3, 4]$
  - `outdegree` $=$ 2
    `sources` $= [0, 5]$



Process Topologies: Graph (2/2)

- Syntax:

```
int MPI_Dist_graph_create_adjacent(
 MPI_Comm comm,             /* Existing communicator */
 int indegree,              /* Size of sources/sourceweights arrays */
 const int sources[],       /* List of ranks where called is a
                               possible destinations */
 const int sourceweights[], /* Weight of the edges */
 int outdegree,             /* Size of destination/destinationweights
                               arrays */
 const int destinations[],  /* List of possible destinations */
 const int destweights[],   /* Weight of the edges */
 MPI_Info info,             /* Information object */
 int reorder,               /* Allow to reorder or not */
 MPI_Comm *comm_dist_graph  /* New communicator */
);
```

Content

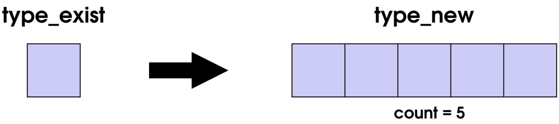
Creating Derived Data Types

- A **derived data** type (also called general data type) is an opaque object that specifies two things:
  - A sequence of basic data types
  - A sequence of integer (byte) displacements
- Creating derived data types involves the following steps:
  - Definition of derived data types through type constructor functions
  - Commit of the new type
  - Release of all types

Derived Data Types: Contiguous Type

- Constructor allows replication of a data type into contiguous locations:

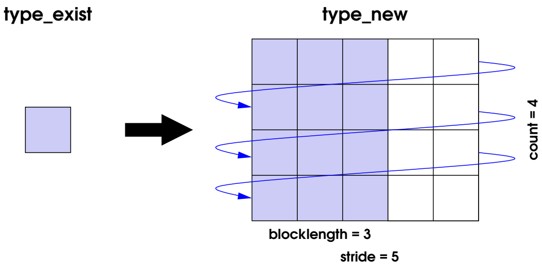
```c
int MPI_Type_contiguous(
 int count,                    /* Replication count */
 MPI_Datatype type_exist,      /* Existing data type */
 MPI_Datatype *type_new        /* New data type */
);
```



Derived Data Types: Vector Type

- `MPI_Type_vector` is a more general constructor that allows replication of a data type into locations that consist of equally spaced blocks:

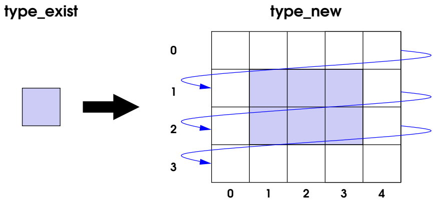
```c
int MPI_Type_vector(
  int count,                   /* Number of blocks */
  int blocklength,             /* Number of elements in each block */
  int stride,                  /* Number of elements between start of each block */
  MPI_Datatype type_exist,     /* Existing data type */
  MPI_Datatype *type_new       /* New data type */
);
```



Derived Data Types: Subarray Type

- To describe an n-dimensional sub-array of an n-dimensional array use

```
int MPI_Type_create_subarray(
  int ndims,
  const int size[],
  const int subsize[],
  const int start[],
  int order,
  MPI_Datatype type_exist,
  MPI_Datatype *type_new
);
```



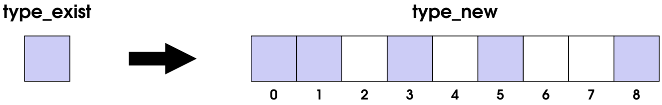
- Example:
  - `ndims = 2`
  - `size[] =` $[5, 4]$
  - `subsize[] =` $[3, 2]$
  - `start[] =` $[1, 1]$
  - `order = MPI_ORDER_FORTRAN`

Derived Data Types: Indexed Type (1/2)

- Even more general: Replicate an existing data type into a sequence of blocks, where each block can contain a different number of copies and have a different displacement:

```
int MPI_Type_indexed(
  int count,                    /* Array length */
  const int blocklenght[],      /* Number of elements per block */
  const int displacement[],     /* Displacement for each block */
  MPI_Datatype type_exist,      /* Existing data type */
  MPI_Datatype *type_new        /* New data type */
);
```

- Example: `count` $= 4$, `blocklength[]` $= [2, 1, 1, 1]$, `displacement[]` $= [0, 3, 5, 8]$

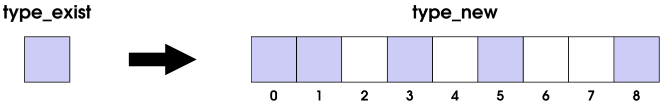


Derived Data Types: Indexed Type (2/2)

- Slightly less general: Assume all blocks to be of the same length:

```
int MPI_Type_create_indexed_block(
  int count,
  int blocklength,
  const int displacement[],
  MPI_Datatype type_exist,
  MPI_Datatype *newtype
);
```

- Example: `count` $= 5$, `blocklength` $= 1$, `displacement[]` $= [0, 1, 3, 5, 8]$

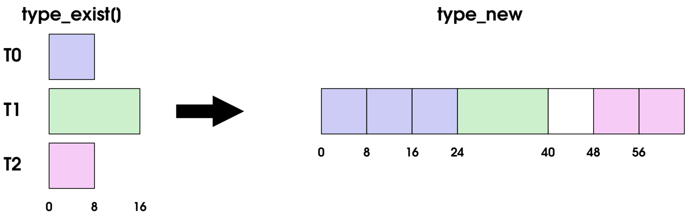


Derived Data Types: Structured Type

- Finally, the most general constructor:

```c
int MPI_Type_create_struct(
  int count,                          /* Array length */
  const int blocklength[],            /* List of number of elements per block */
  const MPI_Aint displacement[],      /* List of displacements in Bytes */
  const MPI_Datatype type[],          /* List of data types */
  MPI_Datatype *type_new              /* New data type */
);
```

- Example: `count` $= 3$, `blocklength` $= [3, 1, 2]$, `displacement` $= [0, 24, 48]$, `type` $=$ [T0,T1,T2],



Derived Data Types: Caveats

- The data type constructors typically assume that objects and arrays are stored linearly in memory
  - For dynamically allocated multi-dimensional arrays in C or C++ this may not be the case
- Performance depends on the data types (and the MPI implementation)
  - Generally expect more general data types to be slower

Content

MPI One-Sided Communication

- Send-receive communication assumes that both end-points of the communication take action (**two-sided communication**)
- MPI since version 2.0 supports also communication based on a get/put semantics with only one end-point being active (**one-sided communication**)
  - Also called **remote memory access (RMA)**
- Challenges:
  - Need for a common address space
  - Suitable synchronisation to ensure RAW and WAR dependencies to be respected

One-Sided Communication: Basic Operations

- Create a memory window for remote access
- Communicate using, e.g.,
  - `MPI_Put`: Write to remote memory
  - `MPI_Get`: Read from remote memory
- Synchronise using `MPI_Win_fence`

Memory Window Creation

- Specify a window of existing memory that processes exposes to RMA accesses:

```
int MPI_Win_create(
  void *base,        /* Initial address of window */
  MPI_Aint size,     /* Size of window [Bytes] */
  int disp_unit,     /* Local unit size for displacements [Bytes] */
  MPI_Info info,     /* Info object */
  MPI_Comm comm,     /* Communicator */
  MPI_Win *win       /* Window handle (out) */
);
```

- Operation is a collective call, all processes must participate
- All address computations are scaled by the `disp_unit`
  - Choose `disp_unit` = 1 for no scaling
- The `MPI_Info` object allows to provide optimisation hints (not covered here)
  - `MPI_Info` allows to store an unordered set of (key,value) pairs
  - May use `info` = `MPI_INFO_NULL`

Synchronisation

- All RMA calls related to a particular window can be synchronised using

```
int MPI_Win_fence(
 int assert,   /* Program assertion */
 MPI_Win win   /* Window handle */
);
```

- The application may make assertions, which may allow for optimisations but `assert` $= 0$ is always valid
- `MPI_Win_fence` separates different **access epochs**
  - Epoch starts with an RMA synchronisation call on `win`
  - Next, a number of communication calls may happen
  - Epoch stops with another synchronisation call
- Other synchronisation mechanisms available (not considered here)

MPI_Put

- The analogue of a send operation is the following operation that puts data into remote memory:

```c
int MPI_Put(
  void *oaddr,                /* Address of origin buffer */
  int ocount,                 /* Number of entries in origin buffer */
  MPI_Datatype odatatype,     /* Data type of entries in origin buffer */
  int trank,                  /* Target rank */
  MPI_Aint tdisp,             /* Displacement from window start to target buffer */
  int tcount,                 /* Number of entries in target buffer */
  MPI_Datatype tdatatype,     /* Data type of entries in origin buffer */
  MPI_Win win                 /* Window handle */
);
```

- The data is
  - Read from the local memory at address `oaddr`
  - written to the remote memory at address $\mathtt{taddr} = \mathtt{window\_base} + \mathtt{tdisp} \times \mathtt{disp\_unit}$
- Need synchronisation to avoid
  - Remote data is overwritten before read of old data (WAR)
  - Remote data is used before new data is written (RAW)

MPI_Get

- To copy data from the target memory to the origin use

```
int MPI_Get(
  void *oaddr,                 /* Address of origin buffer */
  int ocount,                  /* Number of entries in origin buffer */
  MPI_Datatype odatatype,      /* Data type of entries in origin buffer */
  int trank,                   /* Target rank */
  MPI_Aint tdisp,              /* Displacement from window start to target buffer */
  int tcount,                  /* Number of entries in target buffer */
  MPI_Datatype tdatatype,      /* Data type of entries in origin buffer */
  MPI_Win win                  /* Window handle */
);
```

- Need synchronisation to avoid
  - Remote data is not updated before get is executed (RAW)

One-Sided Communication: Ping (1/2)

```c
int main(int argc ,char *argv[])
{
  MPI_Init(NULL, NULL);

  int irank;
  MPI_Comm_rank(MPI_COMM_WORLD, &irank);

  int buf = -1; MPI_Win win;
  MPI_Win_create(&buf, sizeof(int), sizeof(int),
                 MPI_INFO_NULL, MPI_COMM_WORLD, &win);

  int src = 1; int dst = 0;
  MPI_Win_fence(0, win);
  if (irank == src) {
    int x = 100 + irank;
    MPI_Put(&x, 1, MPI_INT, dst, 0, 1, MPI_INT, win);
    printf("#%d: have put x=%d\n", irank, x);
  }
  MPI_Win_fence(0, win);

  printf("#%d: buf=%d\n", irank, buf);

  MPI_Finalize();

  return 0;
}
```

What is the expected output?

One-Sided Communication: Ping (2/2)

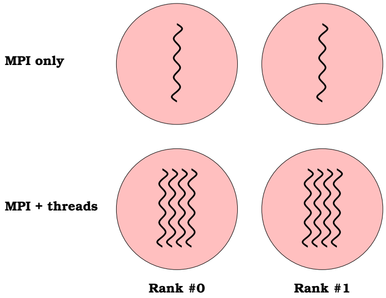
- Answer when using 4 processes (order is not deterministic):

```
#1: have put x=101
#1: buf=-1
#3: buf=-1
#0: buf=101
#2: buf=-1
```

Content

Hybrid Parallelisation Introduction (1/2)

- MPI parallelism
  - processes possibly running on different nodes
  - Separate memory address spaces
- Thread parallelism (e.g., OpenMP)
  - Multiple threads running within a single process
  - Access to a shared memory address space
- **Hybrid parallelisation** $=$ MPI $+$ threads



Hybrid Parallelisation Introduction (2/2)

- All threads within an MPI process share all MPI objects
  - Example: Communicators
- Benefits of hybrid parallelisation
  - Opportunity for more parallelism
  - Reduced memory footprint
- Possible need for tuning
  - Using less MPI ranks per node may reduce overheads
  - Parallel efficiency reduces for a larger number of threads

Thread-safe MPI: Introduction

- MPI defines four levels of thread safety:
  - `MPI_THREAD_FUNNELED`
    - Multi-threaded, but only the main thread makes MPI calls (the one that calls `MPI_Init_thread`)
  - `MPI_THREAD_SERIALIZED`
    - Multi-threaded, but only one thread at a time makes MPI calls
  - `MPI_THREAD_MULTIPLE`
    - Multi-threaded and any thread can make MPI calls at any time (with some restrictions)
  - `MPI_THREAD_SINGLE`
    - Only one thread exists in the application
- **Note**: Choosing for a particular level means making commitments to MPI

Thread-safe MPI: `MPI_THREAD_SINGLE`

- Only one thread is used
  - There are no OpenMP parallel regions

```c
int main(int argc, char ** argv)
{
  double x[100];
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for (int i = 0; i < 100; i++)
    f(x[i]);

  /* MPI communication */

  MPI_Finalize();

  return 0;
}
```

Thread-safe MPI: `MPI_THREAD_FUNNELED`

- All MPI calls are made by the master thread
  - All MPI calls are outside of OpenMP parallel regions

```c
int main(int argc, char **argv)
{
  double x[100];
  int rank, provided;

  MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
  for (int i = 0; i < 100; i++)
    f(x[i]);

  /* MPI communication */

  MPI_Finalize();

  return 0;
}
```

Thread-safe MPI: `MPI_THREAD_SERIALIZED`

- Only one thread can make MPI calls at a time
  - Use OpenMP `critical` clause

```
int main(int argc, char ** argv)
{
  double x[100];
  int rank, provided;

  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
  for (int i = 0; i < 100; i++) {
    f(x[i]);

#pragma omp critical
    /* MPI communication */
  }

  MPI_Finalize();

  return 0;
}
```

Thread-safe MPI: `MPI_THREAD_MULTIPLE`

- Any thread can make MPI calls any time, but there are caveats

```c
int main(int argc, char ** argv)
{
  double x[100];
  int provided;

  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
  for (i = 0; i < 100; i++) {
    f(x[i]);

    /* MPI communication */
  }

  MPI_Finalize();

  return 0;
}
```

MPI_THREAD_MULTIPLE: Caveats

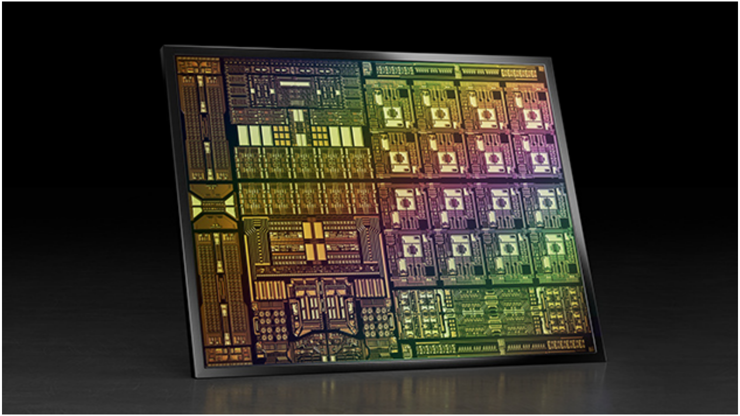
- **Ordering**
  - MPI calls within a thread are executed in order
  - MPI calls in different threads are executed in no particular order
  - Caveat: User is responsible to prevent different threads making conflicting MPI calls
  - Example: Ordering of `MPI_Send` and `MPI_Recv` calls
- **Blocking**
  - MPI calls will block only the calling thread
  - Caveat: User must ensure that all relevant threads are blocked
  - Example: Not all threads calling `MPI_Barrier`

Thread-safe MPI: Implementation Status

- All MPI implementations support `MPI_THREAD_SINGLE`
- `MPI_THREAD_FUNNELED` is typically supported
  - Situation on Dardel: Current CPE supports this levels
- `MPI_THREAD_SERIALIZED` and `MPI_THREAD_MULTIPLE` is possibly not supported
  - Situation on Dardel: Current CPE supports both levels

Finish with Upcoming Networking Technology

NVIDIA BlueField-3 with in-network processing capabilities:



[NVIDIA]