Overview

Introduction to MPI-2

- MPI-2 is an **extension** of MPI.
- Each valid MPI program is also a valid MPI-2 program.
- Additional functionality resides in the areas of **dynamic** process management, **one-way** communication and **parallel in- / output**.

Overview

Process creation and management

- MPI-2 allows **creation** of new and **removal** of existing processes.
- Many MPI-2 functions use data structure of type `MPI_Info`, which enables interaction between the respective MPI-2 implementation and the underlying **operating system**.
- Use of `MPI_Info` leads to **limited portability**.
- Data structure of type `MPI_Info`, basically, contains **pairs** of the form `(key, value)` (which in C consist of character strings of type `char *` terminated by `'\0'`).

MPI_Info Data Structure

Access of the MPI_Info data structure is supported by multiple functions. By

```
int MPI_Info_create(MPI_Info *info)
```

a **new** structure of this type is created. The function

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

adds a **new** pair (key, value) to info, or **overwrites** an already existing pair by with the same content of key. Call of

```
int MPI_Info_get(MPI_Info info, char *key,
                 int valuelen, char *value, int flag)
```

searches in info for a pair with the provided key and writes in value the respective value with a max. length of valuelen. Value of flag is set to false if no matching pair was found, otherwise it is set to true. Pair (key, value) can be **removed** by the following function:

```
int MPI_Info_delete(MPI_Info info, char *key)
```

Creation of New Processes

New processes can be created in MPI-2 by the following function:

```
int MPI_Comm_spawn(char *command, char *argv[],
                   int maxprocs, MPI_Info info,
                   int root, MPI_Comm comm,
                   MPI_Comm *intercomm, int errcodes[])
```

- `command` denotes the name of the **program** to be started.
- `argv` contains **command line arguments** (where `argv[0]` not, as usual in C contains the program name). With `MPI_ARGV_NULL` an empty parameter list can be displayed.
- `maxprocs` indicates the **number** of processes to be started.
- `info` can be used to specify, among other things, the path, name and parameters of the program to be invoked.
  To preserve **portability** `MPI_INFO_NULL` should be passed.
- The above arguments are only evaluated by the **root process** `root`. It also splits new processes.

Creation of New Processes

- The function must be called by **all** processes of communicator `comm`.
- **Inter-communicator** `intercomm` serves for communication between the communicator `comm` and new processes.
- Newly split processes are located in a different group from the one the root process is in. All of them are contained in their **own** `MPI_COMM_WORLD` communicator.
- **Inter-communicator**, which is returned to the calling processes, can be determined by the **new** processors by means of the following function: `MPI_Comm_get_parent(MPI_Comm *parent)`
- `errcodes` is a field of min. length `maxprocs`. After the call each element contains either an **error code** or `MPI_SUCCESS` if the respective process was successfully started.

Creation of Multiple New Processes

Several **different** MPI programs with possibly different command line arguments can be split off as new processes by the following function:

```
int MPI_Comm_spawn _multiple(
                  int count, char *commands[] , char **argv[],
                  int maxprocs[], MPI_Info infos[],
                  int root, MPI_Comm comm,
                  MPI_Comm *intercomm, int errcodes[])
```

- count indicates the **number** of different calls.
- commands, argv, maxprocs and infos are fields of length count, the elements of which have the same meaning as the corresponding arguments of MPI_Comm_Spawn().
- Other arguments directly correspond to those of MPI_Comm_Spawn().
- errcodes contains $\sum_{i=0}^{\texttt{count}-1} \texttt{maxprocs}[i]$ error codes, where the order of those codes corresponds to the respective calls in commands field.

- By appropriate multiple calls of the function `MPI_Comm_spawn()` the same programs can be started similarly to a single call of the `MPI_Comm_spawn_multiple()` function. Though, in case of a single `MPI_Comm_spawn_multiple()` call only one new **shared** communicator `MPI_COMM_WORLD` is created for all new processes, while a single call of `MPI_Comm_spawn()` always creates another **different** communicator `MPI_COMM_WORLD`.

- The **maximum number of processes**, which can be active concurrently is provided at program launch by `MPI_Init()` in the `MPI_UNIVERSE_SIZE` attribute of the communicator `MPI_COMM_WORLD`. It, typically, corresponds to the number of available processors.

Overview

One Sided Communication

A source process can access the address space of a destination process without an active involvement of the destination process.
$\rightarrow$ RMA (remote memory access)

- flexible and dynamic distribution of program data on the memories of the participating processors
- flexible memory access of the participating processors
- coordination of the memory accesses through the programmer, i.e. at a time different processes should not modify memory addresses of an address space concurrently because then race conditions are possible

Overview

Window Objects I

Situation:

- Process A should access the local memory region from a process B Windows creation for the external access.
- Therefor the Process B has open its memory region for the external access.

```
int MPI_Win_create(void *base,
                   MPI_Aint size,
                   int displ_unit,
                   MPI_Info info,
                   MPI_Comm comm,
                   MPI_Win *win)
```

- each process from the communicator comm has to execute that operation
- base is the start address of the memory region

Window Objects II

- size is the size in bytes
  (MPI type `MPI_Aint` is used for representing the size of memory regions; larger than 2^{32})

Window Objects III

- `displ_unit` specifies the displacement between neighboring entries of the window
- info data structure used to give the runtime system additional hints (normally `info = MPI_INFO_NULL`)
- `comm` is the communicator of the processes taking part in the window creation with `MPI_Win_create()`
- returns: window object of type `MPI_Win`

Releasing a window for external access:

```
int MPI_Win_free (MPI_Win *win)
```

All operations of a participating processes have to be finished.

Overview

RMA-Operation I

Data transfer with three different non-blocking RMA operations:

- `MPI_Put()` transfers data from the memory of the calling process into the window of another process
- `MPI_Get()` transfers data from the window of the destination process into the memory of the calling process
- `MPI_Accumulate()` accumulates the data into the window of the destination process

Test of completeness of a non blocking RMA-operation by using: Synchronization Operations
→ Forwarding of a local buffer of a communication operation should only be done after using one of the following synchronization operations

Put-Operation I

Storing a data block into the memory of another process:

```
int MPI_Put(void *origin_addr,
            int origin_count,
            MPI_Datatype origin_type,
            int target_rank,
            MPI_Aint target_displ,
            int target_count,
            MPI_Datatype target_type,
            MPI_Win win)
```

- `origin_addr` starting address of the data block provided by the calling process,
- `origin_count` is the number of elements in this data block, `origin_type` is the data type of the elements.
- `target_rank` specifies the rank of the destination process. (This destination process has to have created a window object `win` by a previous call to `MPI_Win_create()`

Put-Operation II

- `target_displ` specifies the distance between the start of the window and the start of the destination buffer within the target process
- `target_count` specifies the number of elements to be received in the buffer at the target process
- `target_type` represent the data type at the target process.

The data block is stored into the memory of the target process starting at position:

`target_addr := window_base + target_displ * displ_unit`

mit

`window_base` = Starting address of the window of the target process and
`displ_unit` = stride between consecutive elements of a window

Corresponds to the semantics of two sided communication: Source process with send operation

```
int MPI_Isend (origin_addr, origin_count, origin_type, target_rank,
tag, comm)
```

Target process with receive operation

```
int MPI_Recv (target_addr, target_count, target_type, source, tag,
comm, &status)
```

Put-Operation III

`comm` is communicator belonging to the group for window win.

Get-Operation I

Reading a data block from the memory of another process

```
int MPI_Get(void *origin_addr,
            int origin_count,
            MPI_Datatype origin_type,
            int target_rank,
            MPI_Aint target_displ,
            int target_count,
            MPI_Datatype target_type,
            MPI_Win win)
```

- `origin_addr` is the starting address of the receive buffer in the local memory of the calling process,
- `origin_count` specifies the number of elements from type origin_type, transferred to the receiving buffer.
- `target_rank` is the rank of the target process, i.e. the process to be read from
- `win` is the window object

Get-Operation II

- Analogous to the `MPI_Put()` operation the remaining parameters specify the position and the number of elements of the data elements read from the data block of the destination window
  Its starting address can be calculated by
  `target_addr := window_base + target_displ * displ_unit.`

Accumulate-Operation I

Accumulation of data in the memory of another process

```
int MPI_Accumulate(void *origin_addr,
                   int origin_count,
                   MPI_Datatype origin_type,
                   int target_rank,
                   MPI_Aint target_displ,
                   int target_count,
                   MPI_Datatype target_type,
                   MPI_Op op,
                   MPI_Win win)
```

- The parameters have the same meaning as the parameters of the `MPI_Put()`-operation.
- additional parameter op specifies reduction operation to be used, compare with reduction operations for `MPI_Reduce()` **no** user definable reduction operations

Restrictions of one sided communication I

- Each memory operation in a window is only allowed to be the target of one one sided communication operation at any time in program execution $\longrightarrow$
  i.e. a concurrent access of a memory location by different processes is not allowed
- Exception: several `MPI_Accumulate`-operations could be active for the same memory location at the same time
  $\longrightarrow$
  Results of the operation = any order of the executed operations (commutative reduction operations guarantee always the same result)
- A window of a process P is not allowed to be accessed by `MPI_Put()`- or `MPI_Accumulate()`-operations of a different process and by a modification of local write-operation from P at the same time (also not if different memory locations of the same window are accessed).

Overview

Global Synchronization I

Global synchronization of a process group of a window

- Suitable for regular applications with alternating
  - global computation phases and
  - global communication phases

Global Synchronization Operation:

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

- Has to be called by all processes of the process group of the window `win`.
- A calling process continues only then with the following instruction, if **all** from this processes issued and on the window win working **one sided communication operations** have finished
- The parameter `assert` can be used to specify the context of the call to `MPI_Win_fence()`, which may be used by the runtime system to do optimizations. (Normal case: no additional information, i.e. `assert = 0`)

Example I

- Iterative computation using a distributed data structure `A`.
- Per iteration step: Each participating process
  - updates its local part of the data structure (`update()`)
  - copies parts of data in continuous buffer(`update_buffer()`)
  - provides parts of its data structure to the neighbor processes (`MPI_Put()`)
  - `MPI_Win_fence()` used before and after communication phases

```
  while (!converged (A)) {
  update(A);
  update_buffer(A, from_buf);
  MPI_Win_fence (0, win);
  for (i=0; i<num_neighbors; i++)
      MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i],
                    to_disp[i], size[i], MPI_INT, win);
  MPI_Win_fence (0, win);
}
```

Loose Synchronization I

Synchronization for the accessing processes and the process whose window is accessed.

- Accessing process specifies the begin and end of an access epoch to the windows of other processes of the process group the accessing process is also part of by using `MPI_Win_start()` or `MPI_Win_complete()`
- Processes whose window is accessed specify the **begin** and the **end** of the **exposure epoch** by using `MPI_Win_post()` and `MPI_Win_wait()`
- Synchronization between `MPI_Win_start()` and `MPI_Win_post()`
  $\longrightarrow$
  RMA-operation of the accessing process (after `MPI_Win_start()`) only after the target process has finished its `MPI_Win_post()`-operation.
- Synchronization between `MPI_Win_complete()` and `MPI_Win_wait()`
  $\rightarrow$
  RMA-operation of the accessing process are finished before the target process has finish the `MPI_Win_wait()`-operation

Operations for Loose Synchronization I

-
  ```
  int MPI_Win_start(MPI_Group group,
                    int assert,
                    MPI_Win win)
  ```
- `int MPI_Win_complete (MPI_Win win)`

-
  ```
  int MPI_Win_post(MPI_Group group,
                   int assert,
                   MPI_Win win)
  ```
- `int MPI_Win_wait (MPI_Win win)`
- `int MPI_Win_test (MPI_Win win, int *flag)`
  `flag = 1` if all RMA-operations have finished on the window win
  `flag = 0` if not all RMA-operations have finished on the window win

Example: Loose Synchronization

```
while (!converged (A)) {
  update (A);
  update_buffer(A, from_buf);
  MPI_Win_start (target_group, 0, win);
  MPI_Win_post (source_group, 0, win);
  for (i=0; i<num_neighbors; i++)
       MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i],
                    to_disp[i], size[i], MPI_INT, win);
  MPI_Win_complete (win);
  MPI_Win_wait (win);
}
```

- `source_group` are the neighbor processes accessing the local window of the actual process
- `target_group` are the neighbor processes whose window is accessed by the actual process
- Each process has different neighbors $\longrightarrow$ weaker synchronization

Lock Synchronization

Synchronization mechanism where only the accessing processes participate actively on the access (Shared Memory Model)

- two processes can communicate using RMA-accesses to the window of a third process,
  without active involvement of the third process, e.g. by actively giving access to the window
- **Methods**:
  Setting a lock before accessing
  and releasing a lock after access
  `MPI_Win_lock()` and `MPI_Win_unlock()`
- Two different lock-mechanism (parameter `lock_type`):
  - exclusive lock by using `lock_type = MPI_LOCK_EXCLUSIVE`
    $\longrightarrow$ suitable for modifications with `MPI_Put`
  - shared lock, by using `lock_type = MPI_LOCK_SHARED` $\longrightarrow$ good for accesses with `MPI_Get()` or `MPI_Accumulate()`

Lock-Operations

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

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

Example: Lock-Synchronization

Each access to the window of another process is protected by an **exclusive lock**

```
while (!converged (A)) {
  update (A);
  update_buffer(A, from_buf);
  MPI_Win_start (target_group, 0, win);
  for (i=0; i<num_neighbors; i++) {
    MPI_Win_lock (MPI_LOCK_EXCLUSIVE, neighbor[i], 0, win);
    MPI_Put (&from_buf[i], size[i], MPI_INT, neighbor[i],
    to_disp[i], size[i], MPI_INT, win);
    MPI_Win_unlock(neighbor[i], win);
  }
}
```