Chapter 5

Distributed Memory Systems: Part IV

Message Passing Interface API

The Message Passing Interface is a standard for creation of parallel programs using the message passing programming model. It describes

- functionality,
- behavior,
- API syntax

of the required routines. It does, however, not prescribe any implementation details. It is, e.g., completely open by what means a message is transferred.

Message Passing Interface API

The MPI uses a specialized execution environment that spawn and administrates the instances of a process. Relevant functions for

- setup and destruction of the working environments context
- grouping processes
- actual message transmission
- ...

are collected in the `mpi.h` header file. We will see later for the case of the Open MPI implementation of the standard how we can compile and run a program using the MPI features.

Message Passing Interface API

MPI Context Initialization and Finalization

The most basic components of the MPI program are

```
#include <mpi.h>
```

to make the standard available. Then before we can use any message passing routines we need to initialize the execution context via

```
int MPI_Init(int *argc, char ***argv)
```

passing on the usual arguments of the `main()` function of our C program. After we have finished our MPI related work the execution context is destroyed using

```
int MPI_Finalize()
```

Processes may continue performing local work after the finalization, but with a very few exceptions none of the MPI function work anymore. It is mandatory to make sure all MPI operations have finished before calling `MPI_Finalize()`.

Message Passing Interface API

Process Groups and Communicators: Process Groups

Definition

Process group Processes in MPI may be clustered in so called process groups. These are ordered sets of instances of the program numbered from 0 to $n-1$. The local numbers of the processes are called `rank`.

From the programmers view an MPI group is an object of type `MPI_Group`, which can be accessed via a handle. There exists one predefined group constant `MPI_GROUP_EMPTY`, denoting the empty group.

MPI process groups are useful to implement task parallel applications.

MPI supports communication inside a group and point to point type communication between groups.

Message Passing Interface API

Process Groups and Communicators: Process Group Functions

```
int MPI_Group_union(MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *newgroup)
```

Generates the union of two existing groups by including all elements of the first group, followed by all elements of second group that are not in the first group.

- `group1`, `group2` groups to include
- `*newgroup` handle of the group to create. This may be equal to the empty group `MPI_GROUP_EMPTY`.

The operation is not commutative but associative.

Message Passing Interface API

Process Groups and Communicators: Process Group Functions

```
int MPI_Group_intersection(MPI_Group group1,
                           MPI_Group group2,
                           MPI_Group *newgroup)
```

Produces a group at the intersection of two existing groups by including all elements of the first group that are also in the second group, ordered as in first group.

- `group1`, `group2` groups to intersect,
- `*newgroup` handle of the group to create. This may be equal to the empty group `MPI_GROUP_EMPTY`.

The operation is not commutative but associative.

Message Passing Interface API

Process Groups and Communicators: Process Group Functions

```
int MPI_Group_difference(MPI_Group group1,
                         MPI_Group group2,
                         MPI_Group *newgroup)
```

Generates teh new group from the difference of the existing groups by including all elements of the first group that are not in the second group, ordered as in the first group.

- `group1`, `group2` groups to determine the difference from
- `*newgroup` handle of the group to create. This may be equal to the empty group `MPI_GROUP_EMPTY`.

Message Passing Interface API

Process Groups and Communicators: Process Group Functions

```
int MPI_Group_incl(MPI_Group group,
                   int n,
                   int *ranks,
                   MPI_Group *newgroup)
```

Create a new group from an existing group by including a possibly reordered subset of the processes.

- `group` the existing group
- `n` number of ranks used in the new group
- `ranks` ordered list of members for the new group
- `*newgroup` handle of the group to create.

Message Passing Interface API

Process Groups and Communicators: Process Group Functions

```
int MPI_Group_excl(MPI_Group group,
                   int n,
                   int *ranks,
                   MPI_Group *newgroup)
```

Create a new group from an existing group by excluding a possibly reordered subset of the processes.

- `group` the existing group
- `n` number of ranks used in the new group
- `ranks` ordered list of members to exclude from the new group
- `*newgroup` handle of the group to create.

Message Passing Interface API

Process Groups and Communicators: Process Group Functions

```
int MPI_Group_size(MPI_Group group, int *size)
```

Determines the number of members of a group, returned in `size`.

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

Find the `rank` (local number) of the current process in `group`.

```
int MPI_Group_compare(MPI_Group group1,
                      MPI_Group group2,
                      int *result)
```

Find out how different `group1` and `group2` are. The `result` is `MPI_IDENT` if they are the same, `MPI_SIMILAR` in case they only differ in the order of the processes and `MPI_UNEQUAL` otherwise.

Unused groups can be released by calling

```
int MPI_Group_free(MPI_Group *group)
```

On successful return `group` is set to `MPI_GROUP_NULL`

Message Passing Interface API

Process Groups and Communicators: Communicators

Definition

Communicators The participants in a communication operation in MPI are usually determined via so called communicators. MPI distinguishes two types of communicators

- intra-communicators for the collective communication inside a process group
- inter-communicators for the point-to-point like communication between two process groups.

If we are following the SPMD programming model and do not want to have task-parallelism in our code, we are usually fine with the predefined default communicator `MPI_COMM_WORLD`. When people simply speak of a communicator they usually refer to an intra-communicator.

Communicators are objects of type `MPI_Comm`

Message Passing Interface API

Process Groups and Communicators: Communicator Functions

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

Create a new communicator for a subset of the processes.

- `comm` base communicator
- `group` process group the new communicator will be associated with. Must be a subgroup of the group associated to `comm`.
- `*newcomm` handle to the newly created communicator.

Message Passing Interface API

Process Groups and Communicators: Communicator Functions

```
int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```

are the communicator equivalents of the equally called group functions. For `comm` equal to `MPI_COMM_WORLD` the total number of processes and the global `rank`s are returned. Otherwise those of the associated group are given.

For the `MPI_Comm_compare` function the value `MPI_IDENT` here means that the underlying groups are in fact the same. `MPI_CONGRUENT` is returned if the groups are equal (including the order of the `rank`s) but not the same one group. If only the order differs the result is `MPI_SIMILAR` again and `MPI_UNEQUAL` otherwise.

Message Passing Interface API

Point-to-Point Communication

```
int MPI_Send(void *buf,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm)
```

Perform a blocking send operation.

- `buf` address of the sendbuffer
- `count` number of elements to send
- `datatype` type of send buffer elements
- `dest` the `rank` of the destination process inside `comm`
- `tag` a message identifier
- `comm` the communicator to use for the transmission

Message Passing Interface API

Point-to-Point Communication

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

Performs a standard-mode blocking receive.

- `buf` address of the send buffer
- `count` number of elements to send
- `datatype` type of send buffer elements
- `source` the `rank` of the sending process inside `comm`
- `tag` a message identifier
- `comm` the communicator to use for the transmission
- `status` a status object containing information about the sender, the message tag, and possible errors. Also the length of the message received can be retrieved from it using the `MPI_Get_count` function. This can be set to the constant `MPI_STATUS_IGNORE` to save resources if not needed by the application.

Message Passing Interface API

Point-to-Point Communication

Variants of these functions performing the send and receive in a single call or that are non-blocking, exist, for the details see the Standard and the man pages of `MPI_Sendrcv()`, `MPI_Isend()`, `MPI_Irecv()`.

For the non-blocking communication operations the function `MPI_Test()` can be used to check whether a certain message has been transferred.

Message Passing Interface API

Single-Collective Communication

```
int MPI_Barrier(MPI_Comm comm)
```

Actually not performing a real communication this function makes sure that process flow stops until all processes in the group associated to `comm` have reached this point.

- `comm` the communicator to use the barrier for

Message Passing Interface API

Single-Collective Communication

```
int MPI_Bcast(void *buffer,
              int count,
              MPI_Datatype datatype,
              int root,
              MPI_Comm comm)
```

Broadcasts a message from one process to all other processes of the communicator.

- `*buffer` address of the send/receive buffer
- `count` number of elements to send
- `datatype` type of send buffer elements
- `root` the `rank` of the sending process
- `comm` the communicator to be use

Message Passing Interface API

Single-Collective Communication

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

Reduces values on all processes within a group associated to a communicator

- `*sendbuf` address of the send buffer
- `*recvbuf` address of the receive buffer (only relevant on `root`)
- `count` number of elements to send
- `datatype` type of buffer elements
- `op` the arithmetic operation to use in the reduce
- `root` the `rank` of the root/receiving process
- `comm` the communicator to be use

Message Passing Interface API

Single-Collective Communication

```
int MPI_Scatter(void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm)
```

Distributes data from one process among all processes in the communicator

- `*sendbuf` address of the send buffer
- `sendcount` number of elements to send
- `sendtype` type of the send buffer elements
- `*recvbuf` address of the receive buffer
- `recvcount` number of elements to receive
- `recvtype` type of the receive buffer elements
- `root` the `rank` of the root/sending process
- `comm` the communicator to be use

Message Passing Interface API

Single-Collective Communication

```
int MPI_Gather(void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm)
```

Collects data from all processes on a single process.

- `*sendbuf` address of the send buffer
- `sendcount` number of elements to send
- `sendtype` type of the send buffer elements
- `*recvbuf` address of the receive buffer
- `recvcount` number of elements to receive
- `recvtype` type of the receive buffer elements
- `root` the `rank` of the root/receiving process
- `comm` the communicator to be use

Message Passing Interface API

Multi-Collective Communication

```
int MPI_Allgather(void *sendbuf,
                  int  sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm)
```

Collects and redistributes data from all processes to all processes.

- `*sendbuf` address of the send buffer
- `sendcount` number of elements to send
- `sendtype` type of the send buffer elements
- `*recvbuf` address of the receive buffer
- `recvcount` number of elements to receive
- `recvtype` type of the receive buffer elements
- `comm` the communicator to be use

Message Passing Interface API

Multi-Collective Communication

```
int MPI_Allreduce(void *sendbuf,
                  void *recvbuf,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Op op,
                  MPI_Comm comm)
```

Similar to the `MPI_Reduce()` function it combines values from all processes, but in addition it distributes the result back to all processes.

- `*sendbuf` address of the send buffer
- `*recvbuf` address of the receive buffer
- `count` number of elements to send
- `datatype` type of buffer elements
- `op` the arithmetic operation to use in the reduce
- `comm` the communicator to be use

Message Passing Interface API

Multi-Collective Communication

```
int MPI_Alltoall(void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 MPI_Comm comm)
```

The total exchange operation, i.e., every process sends to all other processes.

- `*sendbuf` address of the send buffer
- `sendcount` number of elements to send
- `sendtype` type of the send buffer elements
- `*recvbuf` address of the receive buffer
- `recvcount` number of elements to receive
- `recvtype` type of the receive buffer elements
- `comm` the communicator to be use

Message Passing using Open MPI

Multi-Collective Communication: Hello World

The obligatory "hello world!" program does no more than initializing the MPI context, printing the obligatory text from all instances and destroying the context again:

```
#include <stdio.h>
#include <mpi.h>


int main (int argc, char** argv){

  /* start MPI context*/
  MPI_Init(&argc, &argv);

  /*Do something*/
  printf("Hello world\n");

  /* Stop MPI context*/
  MPI_Finalize();
  return 0;
}
```

Message Passing using Open MPI

Multi-Collective Communication

In Open MPI[8] a C wrapper compiler called `mpicc` is provided. Its sole purpose is to transparently

- add relevant compiler and linker flags to the user's compiler command line
- and then call the underlying compiler to perform the actual compilation.

Especially we do not need to care where exactly the necessary MPI libraries are located and which additional flags are required. If we have specified additional parameters (e.g. for code optimization, or debugging), `mpicc` passes them on to the underlying compiler.

Example

Thus, to compile the "hello world" code, we simply use:

```
mpicc hello_world.c -o hello_world -O2
```

[8] `http://www.open-mpi.org/`

Message Passing using Open MPI

Multi-Collective Communication

The drawback of the MPI framework is that processes need to be started within a special runtime environment. In the case of Open MPI this is invoked using the `mpirun` tool:

```
mpirun [ options ] <program> [ <args> ]
```

The tool takes a couple of options that allow to steer the number of processes spanned, including where they are spanned, control their working environment (path, working directory, environment variables, . . .) and the redirection of standard input and output and many details more.

Message Passing using Open MPI

Multi-Collective Communication

The most important options of `mpirun` for beginners are:

- `-n <#>` run this many copies, if unset Open MPI spans one copy per processor (aliases are `-c`, `--n`, `-np`).
- `-H` List of hosts (comma separate) to spawn the processes on (aliases `-host`, `--host`)
- `-hostfile` Provide a hostfile to use instead of the list above. (aliases and synonyms `--hostfile`, `-machinefile`, `--machinefile`)

Example

To run 1 copy of `hello_world` (from the local directory) each on the two hosts `alpha`, `beta` we may use

```
mpirun -np 2 -H alpha,beta ./hello_world
```