Message Passing Interface (MPI)

In a typical Unix installation a MPI Fortran 77 program will compile with

$$\ldots\texttt{pathname/mpif77 -O fln.f} \tag{1}$$

and generates an executable file, `a.out`, which runs with something like

$$\ldots\texttt{pathname/mpirun -nolocal -np n a.out} \tag{2}$$

or ... `./a.out` in some installations. Here the option `-np` defines the number n of processes, $n = 1$ if the `-np` option is omitted. In our installation (2) reads

$$\ldots\texttt{pathname/mpirun --hostfile hostfile.txt -nolocal -np n a.out} \tag{3}$$

where the file `hostfile.txt` contain the names (addresses) of the participating non-local processors. The number of processors (CPUs) can be smaller than the number of processes. MPI will then run several, or even all, processes on one CPU.

As in our previous Fortran programs, we include the implicit declaration for our real and logical (I) variables and use the Fortran default for the remaining integers. This is now followed by the MPI preprocessor directive, which is brought into the routines through

```
include 'mpif.h' .
```

In a properly implemented MPI environment the compiler will find the file `mpif.h` which contains definitions, macros and function prototypes necessary for compiling the MPI program. The order of the statements matters, because `mpif.h` overwrites some of our implicit declarations with explicit ones. All definitions which MPI makes have names of the form

```
MPI_ANY_NAME .
```

Therefore, **do not to introduce any own variables with names of the form** `MPI_...`. In our code we use instead `ANY_NAME_MPI...`, when we like to emphasize a

relationship to MPI. Our usual include statements for parameter files and common blocks follow `include 'mpif.h'`.

Before calling MPI communication routines, MPI needs to be initialized and it is mandatory to call

```
MPI_INIT(IERROR) .
```

`MPI_INIT` must be called only once. The integer variable `IERROR` can be ignored as long as everything proceeds smoothly. It returns MPI error codes, which can be looked up in a MPI manual. Next, a process may call

```
MPI_COMM_RANK(COMM, RANK, IERROR)
```

to find out its rank. The rank is an integer, called `MY_ID` (for my identity) in our program. The first argument of `MPI_COMM_RANK` is a **communicator**. A communicator is a collection of processes that can send messages to each other. For basic programs the only communicator needed is the one used here

```
MPI_COMM_WORLD .
```

It is predefined in MPI and consists of all processes running when the program execution begins. The second argument is the integer rank of the process, which takes the values $0, \ldots, n-1$ if there are n processes. In a program the integer variable `MY_ID` (which denotes the rank) can be used to branch the program by assigning different random numbers and temperatures to the processes.

Some constructs depend on the total number of processes executing the program. MPI provides the routine

`MPI_COMM_SIZE(COMM, ISIZE, IERROR)`

which returns the number of processes of the communicator `COMM` in its integer argument `ISIZE`. In our programs we use the integer variable `N_PROC` (number of processes) to denote the size.

Two important MPI routines achieve point to point communication between

processes:

```
MPI_SEND(BUF, ICOUNT, DATATYPE, IDEST, ITAG, COMM, IERROR)
```

and

```
MPI_RECV(BUF, ICOUNT, DATATYPE, ISOURCE, ITAG, COMM, ISTATUS, IERROR) .
```

Differences in the arguments are `DEST` in `MPI_SEND` versus `SOURCE` in `MPI_RECV` and the additional `STATUS` array in `MPI_RECV`. All arguments are explained in the following.

1. `BUF`: The initial address of the send or receive buffer.

2. `ICOUNT`: An integer variable which gives the number of entries to be send.

3. `DATATYPE`: The datatype of each entry. MPI datatypes which agree with the corresponding Fortran datatypes are `MPI_INTEGER`, `MPI_REAL`,

MPI_DOUBLE_PRECISION, MPI_COMPLEX, MPI_LOGICAL and MPI_CHARACTER. Additional MPI data types (MPI_BYTE and MPI_PACKED) exist.

4. IDEST: The rank of the destination process.

5. ISOURCE: The rank of the source. Note that the sender and receiver can be identical.

6. ITAG: The message tag, an integer in the range $0, \ldots, 32767$ (many MPI implementations allow for even larger numbers than 32767), which has to be identical for sender and receiver. The purpose of the tag is that the receiver can figure out the message order when one process sends two or more messages (the receive can be delayed due to other processing).

7. COMM: The communicator. Most often this is MPI_COMM_WORLD.

8. ISTATUS: The return status of MPI_RECV. This array is properly dimensiones by MPI_STATUS_SIZE. Strange errors can result if the corresponding dimension

statement is missing (consider to have ISTATUS(MPI_STATUS_SIZE) in a common block, which is propagated to all routines that need it). Besides that ISTATUS can be more or less ignored. Consult the MPI manual, if peculiar completions codes are encountered.

9. IERROR: The error code, an integer with which we deal in the same way as with the return status.

The fields **source, destination, tag** and **communicator** are collectively called the **MPI message envelope**. The receiver may specify the wildcard value MPI_ANY_SOURCE for SOURCE, and/or the wildcard value MPI_ANY_TAG for TAG, to indicate that any source and/or tag value is acceptable. However, there is then a danger to loose control about what it transferred by whom to who.

MPI code should conclude with a call to

```
MPI_FINALIZE(IERROR)
```

which tells its process that no further MPI instructions follow. This call is mandatory for an error free termination of MPI.

More usefull instruction are listed in the following.

```
MPI_BARRIER(COMM, IERROR)
```

which is normally called with the MPI_COMM_WORLD communicator. Execution of the code proceeds only when all processes have reached the barrier, so that synchronization is enforced at this point. In particular for debugging a code that can be of value.

```
MPI_ALLGATHER (SENDBUF, ISENDCOUNT, SENDTYPE, RECVBUF,
               IRECVCOUNT, RECVTYPE, COMM, IERROR)
```

gathers on each process information from all processes, with the range defined by a communicator. The arguments are: SENDBUF, the starting element of the send buffer; ISENDCOUNT (integer), the number of elements in the send buffer; SENDTYPE, the data type of the send buffer elements; RECVBUF, the first element of the receive buffer; IRECVCOUNT (integer), the number of elements in the receive buffer; RECVTYPE, the data type of the receive buffer; COMM, the communicator; IERROR, the MPI error code.

Finally a usefull instruction, in particular for debugging purposes, is

MPI_BCAST(BUF, ICOUNT, DATATYPE, ISOURCE, COMM, IERROR)

which broadcasts the buffer BUF from the process with rank ISOURCE to all processes specified by the communicator COMM. The number of elements in the buffer is given by ICOUNT, the datatype by DATATYPE, and IERROR is the MPI error code.