An Introduction to MPI

by A.N. Spyropoulos

MPI (Message-Passing Interface) is a message-passing library interface specification. The goal of the MPI is to establish a portable, efficient, and flexible standard for message passing. The MPI is the first library that has become the standard for writing message passing programs on HPC platforms. Thus, there is no need to modify your source code when you port your application to a different platform that supports the MPI standard.

The MPI is used on distributed and shared memory systems (SMP/NUMA), and on Hybrid architectures (SMP clusters, workstation clusters and heterogeneous networks)

The programmer is responsible for correctly identifying parallelism and implementing parallel algorithms.

1. Getting Started

A General MPI FORTRAN code has the following structure:

```
program main
implicit NONE
include 'mpif.h'
   <declarations>
   <Serial code>
MPI Start
   <Parallel code and communication>
MPI Finish
   <Serial code>
end
```

`include 'mpif.h'`

The header file mpif.h is required for all programs and routines which make MPI library calls. For example, the integer scalars MPI_SUCCESS and MPI_COMM_WORLD (mentioned below) are predefined into the header file mpif.h

MPI Start

`call MPI_INIT(error)`

Initializes the MPI execution environment. This function must be called in every MPI program, before any other MPI function and must be called only once in an MPI program.

Error code: Returned into the integer scalar **error**. MPI_SUCCESS if successful.

`call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)`

Determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application. Here the number of processes is stored into the scalar integer **numprocs**.

`call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)`

Determines a rank within the communicator MPI_COMM_WORLD. Initially, to each process is assigned a unique integer rank between 0 and (**numprocs** – 1) within the communicator. This rank is often referred to as a process ID. Here the process ID is stored into the scalar integer **`myid`**.

MPI Finish

`call MPI_FINALIZE(error)`

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program.

Example 1: My First MPI FORTRAN Code

```
program hello
implicit NONE
include 'mpif.h'

integer myid, numprocs, error

call MPI_INIT(error)
if(error/=MPI_SUCCESS) then
  print *, "Error starting MPI program"
  call MPI_FINALIZE(error)
endif

call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

print *, "My ID:", myid
print *, "Number of processes:", numprocs
if(myid==0) print *, "Hello world"

call MPI_FINALIZE(error)

end
```

2. Point to Point Communication Routines

MPI point-to-point operations always involve exactly two processes. One process is performing a send operation and the other process is performing a matching receive operation. To send a message, a source process makes an MPI call which specifies a destination process, in terms of its process ID, in the appropriate communicator (e.g. MPI_COMM_WORLD). The destination process also has to make an MPI call to receive the message.

There are four communications modes provided by MPI for the *send* operation (Table 1) and there is one mode for the *receive* operation

Table 1: MPI communication modes

	Completion Condition
Synchronous send	Only completes when the receive has completed
Buffered send	Always completes (unless an error occurs)
Standard send	Either synchronous or buffered
Ready send	Always completes (unless an error occurs)
Receive	Completes when a message has arrived

All four modes exists in both blocking and non-blocking forms (Table 2). In the blocking forms, return from the routine implies completion. In the non-blocking forms, all modes are tested for completion with special MPI routines (i.e. MPI_WAIT).

Table 2: MPI communication routines

	Blocking form	non-Blocking form
Synchronous send	MPI_SSEND	MPI_ISSEND
Buffered send	MPI_BSEND	MPI_IBSEND
Standard send	MPI_SEND	MPI_ISEND
Ready send	MPI_RSEND	MPI_IRSEND
Receive	MPI_RECV	MPI_IRECV

The argument lists for the *send* and the *receive* communication routines are shown in Table 3.

Table 3: MPI communication routines arguments.

Blocking *send*	buffer, count, type, dest, tag, comm, error
non-Blocking *send*	buffer, count, type, dest, tag, comm, request, error
Blocking *receive*	buffer, count, type, source, tag, comm, status, error
non-Blocking *receive*	buffer, count, type, source, tag, comm, request, error

buffer
the variable name that references the data to be sent or received.
count
the number of data elements to be sent or received.
type
the MPI data type

MPI data type	FORTRAN data type
MPI_CHARACTER	CHARACTER
MPI_INTEGER	INTEGER (KIND=4)
MPI_REAL	REAL (KIND=4)
MPI_DOUBLE_PRECISION	REAL (KIND=8)
MPI_COMPLEX	COMPLEX (KIND=4)
MPI_DOUBLE_COMPLEX	COMPLEX (KIND=8)
MPI_LOGICAL	LOGICAL

dest
is the destination process for the message. This is specified by the rank (process ID) of the destination process
source
is the process ID of the source of the message. **Wildcard:** MPI_ANY_SOURCE to receive a message from any process.
tag
arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag.
comm
indicates the communication context. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used.
status
indicates the source of the message and the tag of the message. In FORTRAN, it is an integer array of size MPI_STATUS_SIZE.
(i.e. integer status(MPI_STATUS_SIZE)
request

non-blocking communication is analogous to a form of delegation — the user makes a request to MPI for communication and checks that if the request will be completed satisfactorily only when it needs to know in order to proceed. The programmer uses this system assigned "handle" later (in a WAIT type routine). In FORTRAN, it is an integer.

Two typical MPI *waits* routines are:

MPI_WAIT (request, status, error)

MPI_WAITALL(count, array_of_requests, array_of_statuses, error)

Example 2: Blocking Message Passing

```
program ping
implicit NONE
include 'mpif.h'

integer numprocs, myid, error
integer dest, source, tag
integer status(MPI_STATUS_SIZE)
integer, parameter :: count=17
character(LEN=count) send_msg , recv_msg

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

tag=1

if (myid==0) send_msg='Message from ID 0'
if (myid==1) send_msg='Message from ID 1'

if (myid==0) then

  dest = 1
  source = 1

  call MPI_SSEND(send_msg, count, MPI_CHARACTER, &
                 dest, tag, MPI_COMM_WORLD, error)

  call MPI_RECV(recv_msg, count, MPI_CHARACTER, &
                 source, tag, MPI_COMM_WORLD,    &
```

```
                status, error)
endif

if (myid==1) then
  dest = 0
  source = 0

  call MPI_RECV(recv_msg, count, MPI_CHARACTER, &
                source, tag, MPI_COMM_WORLD,    &
                status, error)

  call MPI_SSEND(send_msg, count, MPI_CHARACTER, &
                dest, tag, MPI_COMM_WORLD, error)
endif

print *, myid,'recv_mesg: ',recv_msg

call MPI_FINALIZE(error)

end
```

Example 3: non-Blocking Message Passing – Exchange messages on ring topology

```
program ring
implicit NONE
include 'mpif.h'

integer myid, numprocs, error
integer next, prev
integer, dimension(2) :: buf
integer tag1, tag2
integer statuses(MPI_STATUS_SIZE,4), requests(4)
tag1 = 1
tag2 = 2

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

prev = myid - 1
next = myid + 1
```

```
if (myid==0) prev = numprocs - 1

if (myid==(numprocs – 1)) next = 0

call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1, &
               MPI_COMM_WORLD, requests(1), error)
call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2, &
               MPI_COMM_WORLD, requests(2), error)

call MPI_ISEND(myid, 1, MPI_INTEGER, prev, tag2,   &
               MPI_COMM_WORLD, requests(3), error)
call MPI_ISEND(myid, 1, MPI_INTEGER, next, tag1,   &
               MPI_COMM_WORLD, requests(4), error)

!        do some work

call MPI_WAITALL(4, requests, statuses, error)

print *, buf(1), myid, buf(2)

call MPI_FINALIZE(error)

end
```

3. Collective Communication Routines

Collective operations involve all processes in the scope of a communicator.

`MPI_BARRIER(comm, error)`
blocks the calling process until all other processes of the communicator *comm* have called it.

`MPI_BCAST(buffer, count, type, root, comm, error)`
the process with process ID *root* sends the variable *buffer* to all other processes

```
MPI_SCATTER(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype,
            root, comm, error)
```
the process with process ID *root* distributes the array *sendbuf* to all other processes

```
MPI_GATHER(sendbuf, sendcount, sendtype,
```

```
            recvbuf, recvcount, recvtype,
            root, comm, error)
```

the process with process ID *root* collects the array *recvbuf* from all other processes

```
MPI_REDUCE(sendbuf, recvbuf, count, type,
           op, root, comm, error)
```

applies the reduction operation *op* on all processes and places the result *recvbuf* in process with process ID *root*. Some common operations *op* are: MPI_MAX, MPI_MIN, MPI_SUM

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, type,
              op, comm, error)
```

applies the reduction operation *op* on all processs and places the result *recvbuf* on all processes.

Example 4: Collective Comunication – Scatter the columns of an array

```
program scatter
implicit NONE
include 'mpif.h'

integer, parameter :: size=4
integer myid, numprocs, error
integer sendcount, recvcount, source
real, dimension(size,size) :: sendbuf
real, dimension(size)      :: recvbuf

data sendbuf / 1.0,  2.0,  3.0,  4.0, &
               5.0,  6.0,  7.0,  8.0, &
               9.0, 10.0, 11.0, 12.0, &
              13.0, 14.0, 15.0, 16.0 /

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, error)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, error)

if (numprocs==size) then
  source = 1
  sendcount = size
  recvcount = size
  call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, &
                   recvbuf, recvcount, MPI_REAL, &
```

```
                    source, MPI_COMM_WORLD, error)
  print *, 'Process ID = ',myid,' recvbuf: ',recvbuf
else
  if (myid==0) print *, 'Error: numprocs/=4'
endif

call MPI_FINALIZE(error)

end
```