MPI Summary for Fortran

Header File

All program units that make MPI calls must either include the `mpif.h` header file or must use the mpi module. This file defines a number of MPI constants as well as providing the MPI function prototypes. All MPI constants and procedures have the `MPI_` prefix.

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

All new code should be written in Fortran 90+ and the use of the module is strongly encouraged, as it will provide for some degree of checking of subroutine parameters and types.

```
use mpi
```

Choose one of the above statements; do not use both.

Important Predefined MPI Constants

```
MPI_COMM_WORLD
MPI_PROC_NULL
MPI_ANY_SOURCE
MPI_ANY_TAG
MPI_IN_PLACE
```

Widely-Used Predefined MPI Types

Corresponding to standard Fortran types:

```
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_CHARACTER
MPI_LOGICAL
MPI_COMPLEX
```

Nonstandard but supported in most distributions:

MPI_REAL*4
MPI_REAL*8
MPI_DOUBLE_COMPLEX

No corresponding Fortran types:

```
MPI_BYTE
```

```
MPI_PACKED
```

The Essential MPI Procedures

All subroutines have an integer as the last parameter unless otherwise noted. This integer represents a success or failure code. Here we will write the names of the subroutines in all capitals, but this is a convention since Fortran is not case sensitive.

MPI_INIT

This must be the first MPI routine invoked.

```
MPI_INIT(ierr)
integer ierr
```

example

```
call MPI_INIT(ierr)
```

MPI_COMM_RANK

This routine obtains the rank of the calling process within the specified communicator group.

```
MPI_COMM_RANK(comm, rank, ierr)
integer comm, rank, ierr
```

example

```
call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
```

MPI_COMM_SIZE

This procedure obtains the number of processes in the specified communicator group.

```
MPI_COMM_SIZE(comm, np, ierr)
integer comm, np, ierr
```

example

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
```

MPI_FINALIZE

The MPI_FINALIZE routine cleans up the MPI state in preparation for the processes to exit.

```
MPI_FINALIZE(ierr)
integer ierr

example

call MPI_FINALIZE(ierr)
```

MPI_ABORT

This routine shuts down MPI and forces an abnormal termination. It should be called when an error condition is detected, and in general the communicator should always be `MPI_COMM_WORLD`.

```
MPI_ABORT(comm, errorcode, ierr)
integer comm, errorcode, ierr
```

example

```
call MPI_ABORT(MPI_COMM_WORLD, errcode, ierr)
```

MPI_BCAST

This procedure broadcasts a buffer from a sending process to all other processes.

```
MPI_BCAST(buffer, count, datatype, root, comm, ierr)
integer count, datatype, root, comm, ierr
<type> buffer(<length>)

example

call MPI_BCAST(myval,1,MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
```

MPI_REDUCE

The MPI_REDUCE function sends the local value(s) to a specified root node and applies an operator on all data in order to produce a global result, e.g. the sum of all the values on all processes.

```
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
integer count, datatype, op, root, comm, ierr
<type> sendbuf(<length>), recvbuf(<length>)
```

example

```
call MPI_REDUCE(myval,val,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
```

If all processes need the data, it is usually more efficient to use the routine MPI_ALLREDUCE rather than to perform a reduction followed by a broadcast. The syntax of MPI_ALLREDUCE is identical to that of MPI_Reduce except that the parameter for the root process is omitted.

```
call MPI_ALLREDUCE(myval,val,1,MPI_REAL,MPI_SUM,MPI_COMM_WORLD,ierr)
```

MPI_REDUCE operators

```
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_MAXLOC
MPI_MINLOC
MPI_LAND
MPI_BAND
MPI_LOR
MPI_BOR
MPI_LXOR
MPI_BXOR
```

MPI_BARRIER

The MPI_BARRIER function causes all processes to pause until all members of the specified communicator group have called the procedure.

```
MPI_BARRIER(comm, ierr)
integer comm, ierr
```

example

```
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
```

MPI_SEND

MPI_SEND sends a buffer from a single sender to a single receiver.

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)
integer count, datatype, dest, tag, comm, ierr
<type> buf(<length>)
```

example

```
call MPI_SEND(myval,1,MPI_INTEGER,my_rank+1,0,MPI_COMM_WORLD,ierr)
```

or if mybuf is an array mybuf(100),

```
call MPI_SEND(mybuf,100,MPI_INTEGER,my_rank+1,0,MPI_COMM_WORLD,ierr)
```

MPI_RECV

`MPI_RECV` receives a buffer from a single sender.

```
MPI_RECV(buf, count, datatype, source, tag, comm, status, ierr)
integer count, datatype, source, tag, comm
integer status(MPI_STATUS_SIZE)
<type> buf(<length>)
```

example

```
call MPI_RECV(myval, 1, MPI_INTEGER, my_rank-1, 0,                &
              MPI_COMM_WORLD, status, ierr)
```

or if `mybuf` is an array `mybuf(100)`,

```
MPI_RECV(mybuf, 100, MPI_INTEGER, my_rank-1, 0,                   &
              MPI_COMM_WORLD, status, ierr)
```

MPI_SENDRECV

The pattern of exchanging data between two processes simultaneously is so common that a routine has been provided to handle the exchange directly.

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
          recvcount, recvtype, source, recvtag, comm, status, ierr)
integer sendcount, sendtype, dest, sendtag
integer recvcount, recvtype, source, recvtag
integer comm, ierr
integer status(MPI_STATUS_SIZE)
<type> sendbuf(<length>), recvbuf(<length>)
```

example

```
call MPI_SENDRECV(halobuf, 100, MPI_REAL, myrank+1, 0, bcbuf, 100, &
               MPI_REAL, myrank-1, 0, MPI_COMM_WORLD, status, ierr)
```

MPI_GATHER

This routine collects data from each processor onto a root process, with the final result stored in rank order. The same number of items is sent from each process. The count of items received is the count sent by a single process, not the aggregate size, but the receive buffer must be declared to be of a size to contain all the data.

```
integer sendcount, sendtype, recvcount, recvtype
integer root
integer comm, ierr
<type> sendbuf(<length>), recvbuf(<length>)
```

```
example

real, dimension(100)  :: sendbuf
real, allocatable, dimension(:) :: recvbuf

call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
allocate(recvbuf(nprocs*100))
call MPI_GATHER(sendbuf, 100, MPI_REAL, recvbuf, 100,   &
```

```
                    MPI_REAL, 0, MPI_COMM_WORLD, ierr)
```

MPI_GATHER is limited to receiving the same count of items from each process, and only the root process has all the data. If all processes need the aggregate data, MPI_ALLGATHER should be used.

```
call MPI_ALLGATHER(sendbuf, sendcount, MPI_REAL, recvbuf, recvcount,&
                   MPI_REAL, MPI_COMM_WORLD, ierr)
```

If a different count must be sent from each process, the routine is MPI_GATHERV. This has a more complex syntax and the reader is referred to MPI reference books. Similar to GATHER/ALLGATHER, there is also an MPI_ALLGATHERV.

MPI_SCATTER

This routine distributes data from a root process to the processes in a communicator group. The same count of items is sent to each process.

```
integer sendcount, sendtype, recvcount, recvtype
integer root
integer comm, ierr
<type> sendbuf(<length>), recvbuf(<length>)
```

example

```
allocate(sendbuf(nprocs*100))
call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf, recvcount,  &
                 MPI_REAL, 0, MPI_COMM_WORLD, ierr)
```

There is also an MPI_SCATTERV that distributes an unequal count to different processes.

Hello, World!

```
program hello
use mpi

integer ::  myrank, nprocs
integer ::  err

call MPI_INIT(err)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, err)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, err)

if ( myrank .eq. 0 ) then
   print *, 'Running on ',nprocs,' Processes'
endif

print *, 'Greetings from process ', myrank

call MPI_FINALIZE(err)

end
```