MPI Summary for C

Header File

All program units that make MPI calls must include the `mpi.h` header file. 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 “mpi.h”
```

Important Predefined MPI Constants

```
MPI_COMM_WORLD
MPI_PROC_NULL
MPI_ANY_SOURCE
MPI_ANY_TAG
```

Widely-Used Predefined MPI Types

Corresponding to standard C types:

```
MPI_INT
MPI_SHORT
MPI_LONG
MPI_LONG_LONG_INT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_SHORT
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_CHAR
MPI_UNSIGNED_CHAR
```

No corresponding standard C types:

```
MPI_BYTE
MPI_PACKED
```

The Essential MPI Procedures

All procedures return `int` unless otherwise noted. This integer represents a success or failure code. Important: note that the function prototype illustrates how the parameters should be declared, but if a parameter is specified as as pointer but it has been declared as a variable, then the ampersand must be prepended when the variable is sent.

MPI_Init

This must be the first MPI routine invoked.

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

example

```
MPI_Init(&argc, &argv);
```

MPI_Comm_rank

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

```
MPI_Comm_rank(MPI_Comm comm, int* rank)
```

example

```
int my_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

MPI_Comm_size

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

```
MPI_Comm_size(MPI_Comm comm, int* np)
```

example

```
int np;
MPI_Comm_size(MPI_COMM_WORLD, &np);
```

MPI_Finalize

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

```
MPI_Finalize(void)
```

example

```
MPI_Finalize();
```

MPI_Abort

This routine shuts down MPI in the event of 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(MPI_Comm comm, int errorcode)
```

example

```
MPI_Abort(MPI_COMM_WORLD, errcode);
```

MPI_Bcast

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

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

example

```
MPI_Bcast(&myval, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

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

example

```
float myval, val;

MPI_Reduce(&myval, &val, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

`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(MPI_Comm comm)
```

example

```
MPI_Barrier(MPI_COMM_WORLD);
```

MPI_Send

`MPI_Send` sends a buffer from a single sender to a single receiver.

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

example

```
MPI_Send(&myval, 1, MPI_INT, my_rank+1, 0, MPI_COMM_WORLD);
```

or if mybuf is an array mybuf[100],

```
MPI_Send(mybuf, 100, MPI_INT, my_rank+1, 0, MPI_COMM_WORLD);
```

MPI_Recv

`MPI_Recv` receives a buffer from a single sender.

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

example

```
MPI_Status status;
MPI_Recv(&myval, 1, MPI_INT, my_rank-1, 0, MPI_COMM_WORLD, &status);
```

or if `mybuf` is an array `mybuf[100]`,

```
MPI_Recv(mybuf, 100, MPI_INT, my_rank-1, 0, MPI_COMM_WORLD, &status);
```

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(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void* recvbuf, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag,
              MPI_Comm comm, MPI_Status* status)
```

example

```
MPI_Status status;
MPI_Sendrecv(halobuf, 100, MPI_FLOAT, myrank+1, 0, bcbuf, 100,
              MPI_FLOAT, myrank-1, 0, MPI_COMM_WORLD, &status);
```

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.

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

```
example

int nprocs, sendarr[100];
int root=0;
int *recvbuf;

recvbuf=(int *)malloc(nprocs*100*sizeof(int))
MPI_Gather(sendarr, 100, MPI_INT, recvbuf, 100, MPI_INT, root,
           MPI_COMM_WORLD);
```

`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.

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

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.

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

```
example
```

```
int nprocs, recvarr[100];
int root=0;
int *sendbuf;

sendbuf=(int *)malloc(nprocs*100*sizeof(int))
MPI_Scatter(sendbuf, 100, MPI_INT, recvarr, 100, MPI_INT, root,
           MPI_COMM_WORLD);
```

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

Hello, World!

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

int main(int argc, char *argv[])
{
  int rank, npes;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);

  if ( rank == 0 ) {
     printf("Running on %d Processes\n",npes);
  }

  printf("Greetings from process %d\n",rank);

  MPI_Finalize();
}
```