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 use the MPI namespace.

```
#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

The C++ bindings typically pass arguments by reference and return the value of interest, if it is a single variable. The function prototypes indicate how the parameters are to be declared, but if a parameter specified as a pointer but is declared as a variable, an ampersand must be prepended in the method call.

MPI::Init

This must be the first MPI routine invoked.

```
void MPI::Init(int& argc, char**& argv)
```

```
void MPI::Init()
```

example

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

MPI_Comm_rank

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

```
int MPI::Comm::Get_rank() const
```

example

```
my_rank = MPI::COMM_WORLD.Get_rank();
```

MPI_Comm_size

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

```
int MPI::Comm::Get_size() const
```

example

```
np = MPI::COMM_WORLD.Get_size();
```

MPI_Finalize

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

```
void MPI::Finalize()
```

```
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.`

```
void MPI::Comm::Abort(int errorcode)
```

example

```
MPI::COMM_WORLD.Abort(errcode);
```

MPI_Bcast

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

```
MPI::Comm::Bcast(void* buff, int count,
                    const MPI::Datatype& datatype, int root) const=0
```

example

```
int myval;

MPI::COMM_WORLD.Bcast(&myval, 1, MPI_DOUBLE, 0);
```

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.

```
void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
          const MPI::Datatype& datatype, const MPI::Op& op, int root)
                    const=0
```

example

```
MPI::COMM_WORLD.Reduce(&myval, &val, 1, MPI::FLOAT, MPI::SUM, 0);
```

`MPI::Comm::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.

```
void MPI::Comm::Barrier() const=0
```

example

```
MPI::COMM_WORLD.Barrier();
```

MPI_Send

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

```
MPI::Comm::Send(const void* buf, int count, MPI::Datatype& datatype,
                                 int dest, int tag) const
```

example

```
MPI::COMM_WORLD.Send(&myval, 1, MPI::INT, my_rank+1, 0);
```

or if mybuf is an array mybuf[100],

```
MPI::COMM_WORLD.Send(mybuf, 100, MPI::INT, my_rank+1, 0);
```

MPI_Recv

`MPI_Recv` receives a buffer from a single sender.

```
void MPI::Comm::Recv(void* buf, int count, MPI::Datatype& datatype,
               int source, int tag, MPI::Status* status) const
```

`or`

```
void MPI::Comm::Recv(void* buf, int count, MPI::Datatype& datatype,
               int source, int tag) const
```

examples

```
MPI::COMM_WORLD.Recv(&myval, 1, MPI::INT, my_rank-1, 0, status);
```

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

```
MPI::COMM_WORLD.Recv(mybuf, 100, MPI::INT, my_rank-1, 0, 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.

```
void MPI::Comm::Sendrecv(const void* sendbuf, int sendcount,
                   MPI::Datatype& sendtype, int dest, int sendtag,
                   void* recvbuf, int recvcount,
                   MPI::Datatype recvtype, int source, int recvtag,
                   MPI::Status* status) const
```

or

```
void MPI::Comm::Sendrecv(const void* sendbuf, int sendcount,
                   MPI::Datatype& sendtype, int dest, int sendtag,
                   void* recvbuf, int recvcount, MPI::Datatype
                   recvtype, int source, int recvtag)  const
```

example

```
MPI::COMM_WORLD.Sendrecv(halobuf, 100, MPI::FLOAT, myrank+1, 0,
               bcbuf, 100, MPI::FLOAT, myrank-1, 0, 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.

```
void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
          MPI::Datatype& sendtype, void* recvbuf, int recvcount,
          const MPI::Datatype& recvtype, int root) const
```

```
example

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

const int nprocs=comm.Get_size();
recvbuf=new int [nprocs*100*sizeof(int)]
MPI::COMM_WORLD.Gather(sendarr, 100, MPI::INT, recvbuf, 100,
                       MPI::INT, root,);
```

`MPI::Comm::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.

```
void MPI::Comm::Allgather(const void *sendbuf, int sendcount,
                         MPI::Datatype& sendtype, void *recvbuf,
                         int recvcount,MPI::Datatype& recvtype) const
```

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::Comm::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.

```
void MPI::Comm::Scatter(const void* sendbuf, int sendcount,
                         MPI::Datatype& sendtype, void* recvbuf,
                         int recvcount, MPI::Datatype& recvtype,
                         int root) const
```

```
example

int recvarr[100];
int root=0;
int *sendbuf;

sendbuf=new int [nprocs*100*sizeof(int)]

MPI::COMM_WORLD.Scatter(sendbuf, 100, MPI::INT, recvarr, 100,
                        MPI::INT, root);
```

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

Hello, World!

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

int main(int argc, char *argv[])
{
  MPI::Init(argc, argv);

  int rank = MPI::COMM_WORLD.Get_rank();
  int npes = MPI::COMM_WORLD.Get_size();

  if ( rank == 0 ) {
     cout << "Running on "<< npes << " Processes "<< endl;
  }

  cout << "Greetings from process " << rank << endl;

  MPI::Finalize();
}
```