Contents

Message Passing Interface

- Standard message-passing library
  - Includes best of several previous libraries
- Versions for C/C++ and FORTRAN
- Available for free
- Can be installed on
  - Networks for workstations
  - Parallel computers
- There is no shared memory in MPI
- Each MPI process has its own separate address space
  - A pointer from one address space is useless in another
- Aimed at distributed memory machines

- Giant supercomputers made of clusters
- Machines in separate locations

- But it can be implemented on a physically shared-memory machine

Services

- Hide details of architecture
- Hide details of message passing, buffering
- Provides message management services
  - packaging
  - send, receive
  - broadcast, reduce, scatter, gather message modes

Program Organisation

- SPMD - Single Program, Multiple Data
  - Every processor runs the same program
  - Each processor computes with different data
  - Variation of computation on different processors through if or switch statements
- MIMD in a SPMD framework
  - Different processors can follow different computation paths
  - Branch on if or switch based on processor identity

Starting and Finishing

- Statement needed in every program before any other MPI code: `MPI_Init(&argc, &argv)`
- Last statement of MPI code must be: `MPI_Finalize()`
  - Program will not terminate without this statement

Messages

- Message content, a sequence of bytes
- Message needs wrapper
  - Analogous to an envelope for a letter

Letter	Message
Address	Destination
Return Address	Source
Type of Mailing (class)	Message type
Letter Weight	Size (count)
Country	Communicator
Magazine	Broadcast

Basics

Communicator

- Collection of processes
- Determines scope to which messages are relative
- Identity of process (rank) is relative to communicator
- Scope of global communications (broadcast, etc.)

Message Protocol

Send

- Message contents: block of memory
- Count: number of items in a message
- Message type: type of each item
- Destination: rank of processor to receive
- Tag: integer designator for message
- Communicator: the communicator within which the message is sent

Receive

- Message contents: buffer in memory to store received message
- Count: number of items in message
- Message type: type of each item
- Source: rank of processor sending
- Tag: integer designator for message
- Communicator: the communicator within which the message is sent
- Status: information about message received

Example

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"     /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
    int  myRank;        /* rank (identity) of process     */
    int  numProc;       /* number of processors           */
    int  source;        /* rank of sender                 */
    int  dest;          /* rank of destination            */
    int  tag = 0;       /* tag to distinguish messages    */
    char mess[MAXSIZE]; /* message (other types possible) */
    int  count;         /* number of items in message     */
    MPI_Status status;  /* status of message received     */

    MPI_Init(&argc, &argv);          /* start MPI   */

    /* get number of processes  */
    MPI_Comm_size(MPI_COMM_WORLD, &numProc);

    /* get rank of this process */
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if (myRank != 0) { /* all processes send to root */
        /* create message */
        sprintf(mess, "Hello from %d", myRank);
        dest  = 0;                  /* destination is root     */
        count = strlen(mess) + 1;   /* include '\0' in message */

        MPI_Send(mess, count,MPI_CHAR, 0,tag, MPI_COMM_WORLD);
    }
    else { /* root (0) process receives and prints messages  */
           /* from each processor in rank order              */
        for(source = 1; source < numProc; source++){
            MPI_Recv(mess, MAXSIZE, MPI_CHAR,
                     source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", mess);
        }
    }

    MPI_Finalize();               /* shut down MPI */
}
```

- Send - Receive is point-to-point, destination processor is specified by fourth parameter (`dest`) in `MPI_Send`
- Messages can be tagged by integer to distinguish messages with different purposes by the fifth argument in `MPI_Send` and `MPI_Recv`
- `MPI_Recv` can specify a specific source from which to receive (fourth parameter)
- `MPI_Recv` can receive from any source or with any tag using `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
- Communicator, switch parameter in `MPI_Send` and `MPI_Recv` determines context for destination and source ranks
- `MPI_COMM_WORLD` is automatically supplied communicator which includes all processes created at start-up
- Other communicators can be defined by use to group processes and to create virtual topologies
- Status of message received by `MPI_Recv` is returned in the seventh (status) parameter
- Number of items actually received can be determined from status by using function `MPI_Get_count`
- The following call inserted into the previous code would return the number of characters sent in the integer variable `count`: `MPI_Get_count(&status, MPI_CHAR, &count);`

Send and Receive Synchronisation

- Fully Synchronised (Rendezvous)
  - Send and Receive complete simultaneously
    - Whichever code reaches the send/receive first waits
  - Provides synchronisation point (up to network delays)
- Buffered
  - Receive must wait until message is received
  - Send completes when message is moved to buffer clearing memory of message for reuse
- Asynchronous (different API call)
  - Sending process may proceed immediately
    - Does not need to wait until message is copied to buffer
    - Must check for completion before using message memory
  - Receiving process may proceed immediately
    - Will not have message to use until it is received
    - Must check for completion before using message

`MPI_Send` may be fully synchronous or may be buffered

`MPI_Recv` suspends until message is received

`MPI_Isend`/`MPI_Irecv()` are non-blocked. Control returns to program after call is made. (Syntax is the same except `MPI_Request` replaced `MPI_Status`).

Detecting completion

- `MPI_Wait(&request, &status)`
  - `request`: matches request on `Isend` or `Irecv`
  - `status`: returns status equivalent to status for `Recv` when complete
  - Blocks for send until message is buffered or sent so message variable is free
  - Blocks for receive until message is received and ready
- `MPI_Test(&request, flag, &status)`
  - `request`, `status` as for `MPI_Wait`
  - Does not block
  - Flag indicates whether message is sent/received
  - Enables code which can repeatedly check for communication completion

Broadcasting a Message

- Broadcast: one sender, many receivers
- Includes all processes in communicator, all processes must make an equivalent call to `MPI_Bcast`
- Any processor may be sender (root), as determined by the fourth parameter
- First three parameters specify message as for `MPI_Send` and `MPI_Recv`, fifth parameter specifies communicator
- Broadcast serves as a global synchronisation

```
MPI_Bcast(mess, count, MPI_INT, root, MPI_COMM_WORLD);
```

- `mess`: pointer to message buffer
- `count`: number of items sent
- `MPI_INT`: type of item sent
- `root`: sending processor
- `MPI_COMM_WORLD`: communicator within which broadcast takes place

```
MPI_Barrier(MPI_COMM_WORLD)
```

- Provides for barrier synchronisation without message of broadcast
- A barrier is a point in the code where all processes must stop and wait until every process has reached the barrier
- Once all processes have executed the `MPI_Barrier` call then all processes can continue

Reduce

- All processors send to a single processor, the reverse of broadcast
- Information must be combined at receiver
- Several combining functions available
- Data are result may be arrays - combining operation applied element-by-element
- Illegal to alias `dataIn` and `result`

```
MPI_Reduce(&dataIn, &result, count, MPI_DOUBLE, MPI_SUM, root,
MPI_COMM_WORLD)
```

- `dataIn`: data send from each processor
- `result`: store result of combining operation
- `count`: number of items in each dataIn result
- `MPI_DOUBLE`: data type for `dataIn`
- `MPI_SUM`: combining operation
- `root`: rank of processor receiving data
- `MPI_COMM_WORLD`: communicator

Scatter

`MPI_Scatter()`

- Spreads array to all processors
- Source is an array on the sending processor
- Each receiver, including sender, gets a piece of the array corresponding to their rank in the communicator

Gather

- `MPI_Gather()`
- Opposite of scatter
- Values on all processors (in the communicator) are collected into an array on the receiver
- Array locations correspond to ranks of processors

Data Packaging

- Needed to combine irregular, non-contiguous data into single message
- pack/unpack, explicitly pack data into a buffer, send, unpack data from buffer
- Derived data types, MPI heterogeneous data types which can be sent as a message

```
MPI_Pack(Aptr, count, MPI_DOUBLE, buffer, size, &pos, MPI_COMM_WORLD)
```

- `Aptr`: pointer to data to pack
- `count`: number of items to pack
- `buffer`: buffer being packed
- `size`: size of buffer (in bytes)
- `pos`: position in buffer (in bytes), updated communicator

```
MPI_Unpack(buffer, size, &pos, Aptr, count, MPI_DOUBLE, MPI_COMM_WORLD)
```

Timing Programs

- `MPI_Wtime()`
  - returns a double giving time in seconds from a fixed time in the past
  - To time a program, record `MPI_Wtime()` in a variable at start, then again at finish, difference is elapsed time

Compiling and Running MPI

- MPI programs are compiled with a standard compiler such as `gcc`
- However, the compiler takes a lot of command line parameters to include the correct files and pass the right flags
- MPI implementations usually provide a special version of the compiler just for MPI programs
- In OpenMPI this is `mpicc`
  - `mpicc program.c -o program`
  - Internally this works by calling `gcc` with all the necessary flags and includes
- In OpenMPI you need to provide a command-line parameter to say how many copies of your program you want to run
  - `mpirun -np 10 ./program`

- Running MPI programs also requires a lot of flags and environment variables
  - Dynamic linking with MPI run-time libraries
- In OpenMPI a special program is used to load, dynamically link, and run your compiled MPI program