Computação em Larga Escala

Introduction to Message Passing Interface (MPI)

António Rui Borges

Summary

- *What is MPI?*
- *Anatomy of a MPI program*
  - *Error handling*
- *Matching of data types*
- *Communicator concept*
  - *Message format*
- *Point-to-point communication*
  - *Blocking primitives*
- *Suggested reading*

DETI

What is MPI?

MPI, or Message Passing Interface, is a library interface specification. It deals primarily with the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard.

The definition of a message-passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases for which they can provide hardware support, thereby enhancing scallability.

All MPI constants and functions are prefixed by the identifier MPI and for the C language binding are described in the interface file `mpi.h`, which should be always included in the source files.

DETI

Anatomy of a MPI program - 1

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


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

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

it should be the first instruction of the thread main

```
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

get the number of the process

```
   MPI_Comm_size (MPI_COMM_WORLD, &size);
```

get the number of the processes spawned

```
   printf ("Hello! I am %d of %d.\n", rank, size);

   if ((rank == 0) && (argc > 1))
      { for (i = 1; i < argc; i++)
          printf ("%s  ", argv[i]);
        printf ("\n");
      }

   MPI_Finalize ();
```

it should be the last instruction of the thread main before terminating

```
   return EXIT_SUCCESS;
}
```

DETI

Anatomy of a MPI program - 2

```
[ruib@ruib-laptop2 basic]$ mpicc -Wall hello.c -o hello
```

→ compilation

```
[ruib@ruib-laptop2 basic]$ mpiexec -n 4 ./hello
Hello! I am 0 of 4.
Hello! I am 2 of 4.
Hello! I am 1 of 4.
Hello! I am 3 of 4.
```

→ execution (4 processes are spawned – no command line arguments)

```
[ruib@ruib-laptop2 basic]$ mpiexec -n 8 ./hello one two three
Hello! I am 0 of 8.
one  two  three
Hello! I am 1 of 8.
Hello! I am 5 of 8.
Hello! I am 6 of 8.
Hello! I am 7 of 8.
Hello! I am 4 of 8.
Hello! I am 3 of 8.
Hello! I am 2 of 8.
[ruib@ruib-laptop2 basic]$
```

→ execution (8 processes are spawned – with command line arguments)

DETI

Anatomy of a MPI program - 3

For the C language binding, all MPI functions return the status of operation.

The programmer can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI function for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator `MPI_COMM_WORLD`. The attachment of error handlers to objects is purely local: each process may attach different error handlers to corresponding objects.

Two predefined error handlers are available in MPI

- `MPI_ERRORS_ARE_FATAL` – when called, this handler causes the program to abort on all executing processes; it has the same effect as if `MPI_ABORT` was called by the process that invoked the handler
- `MPI_ERRORS_RETURN` – when called, this handler has no effect other than returning the error code to the user.

The error handler `MPI_ERRORS_ARE_FATAL` is associated by default with `MPI_COMM_WORLD` after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal.

Anatomy of a MPI program - 4

With the default error handler (`MPI_ERRORS_ARE_FATAL`)

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

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

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   if (rank == 1)
      MPI_Init (&argc, &argv);            error! MPI_Init can be called only once.
   printf ("Hello! I am %d of %d.\n", rank, size);
   if ((rank == 0) && (argc > 1))
      { for (i = 1; i < argc; i++)
          printf ("%s  ", argv[i]);
        printf ("\n");
      }
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```

Anatomy of a MPI program - 5

```
[ruib@ruib-laptop2 basic]$ mpicc -Wall hello1.c -o hello1

[ruib@ruib-laptop2 basic]$ mpiexec -n 4 ./hello1
Hello! I am 3 of 4.
Hello! I am 0 of 4.
Hello! I am 2 of 4.
Fatal error in PMPI_Init: Other MPI error, error stack:
PMPI_Init(140): Cannot call MPI_INIT or MPI_INIT_THREAD more than once
[ruib@ruib-laptop2 basic]$
```

DETI

Anatomy of a MPI program - 6

With the error handler (`MPI_ERRORS_RETURN`)

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

int main (int argc, char *argv[])
{
   int stat, rank, size;
   char errMessage[100];
   int errMessLen;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   if (rank == 1)
      { MPI_Comm_set_errhandler (MPI_COMM_WORLD, MPI_ERRORS_RETURN);
        if ((stat = MPI_Init (&argc, &argv) & 0xFF) != MPI_SUCCESS)
           { switch (stat)
             { case MPI_ERR_COMM:  printf ("Invalid communicator!\n");
                                   break;
               case MPI_ERR_OTHER: MPI_Error_string (stat, errMessage, &errMessLen);
                                   printf ("%s: MPI_Init called more than once!\n", errMessage);
                                   break;
             }
             MPI_Abort (MPI_COMM_WORLD, EXIT_FAILURE);
           }
      }
   printf ("Hello! I am %d of %d.\n", rank, size);
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```

change the error handler

error! MPI_Init can be called only once.

Anatomy of a MPI program - 7

```
[ruib@ruib-laptop2 basic]$ mpicc -Wall hello2.c -o hello2

[ruib@ruib-laptop2 basic]$ mpiexec -n 4 ./hello2
Hello! I am 0 of 4.
Hello! I am 2 of 4.
Hello! I am 3 of 4.
Other MPI error: MPI_Init called more than once!
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 1
[ruib@ruib-laptop2 basic]$
```

DETI

Matching of data types - 1

MPI, being a library interface specification with bindings to several programming languages, namely C and Fortran, has to specify how its predefined data types used to store and retrieve data during message exchange operations, match the data types of the programming language.

MPI data types can then be thought of as *formal data types* used to describe the *formal parameters* of the MPI functions and the corresponding programming language data types as *actual data types* used to represent the actual values of the parameters.

DETI

Matching of data types - 2

Matching between predefined MPI data types and basic C data types

MPI data type	C data type
`MPI_CHAR`	`char`
`MPI_SHORT`	`short`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`

DETI

Matching of data types - 3

Matching between predefined MPI data types and basic C data types
(`stdbool.h` and `stdint.h`)

MPI data type	C data type
MPI_C_BOOL	bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t

DETI

Matching of data types - 4

There are two special MPI data types which have no match to programming language data types. They are

- `MPI_BYTE` – a value of this data type is simply an 8-bit byte; in C language, one may think of it as an `unsigned char`
- `MPI_PACKED` – a value of this data type, perceived as an array of bytes, maps in a contiguous buffer copies of the contents of non-contiguous regions of data memory; it becomes useful for composing the information content of messages consisting of the values of variables dispersed in memory.

DETI

Communicator concept - 1

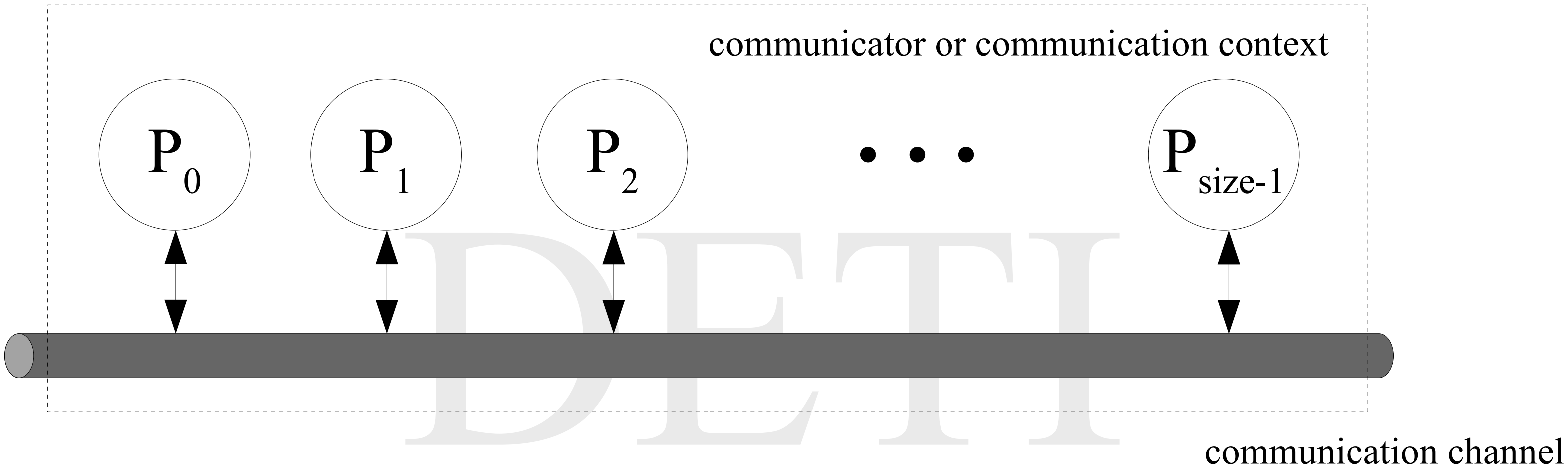
A *communicator* specifies the communication context for a communication operation. Each communication context provides a separate communication universe: messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share the communication context. This set or *process group* is ordered and processes are identified by their *rank* within the group. Thus, the range of valid values for process identification is $0, \ldots, size - 1$, where *size* is the number of processes in the group.

A predefined communicator, called `MPI_COMM_WORLD`, is provided by MPI. It allows communication with all processes that are accessible after MPI initialization. The processes are identified by their rank in the group of `MPI_COMM_WORLD`.

DETI

Communicator concept - 2



- the *size* of the communication context is set, after MPI initialization, by the number of the spawned processes specified on the execution command
- each process within the communication context is identified by its *rank*

Message format - 1

header *or* ***envelope***

- communication context
- *source identification*
- *destination identification*
- *tag*

DETI

information content

- *data type*
- *reference to buffer where data are stored or will be stored*
- *number of data elements*

Message format - 2

Header or **envelope**

- *communication context* – the process group within which the communication operation takes place
- *source identification* – the identification of the process which sends the message, that is, its rank; it is implicitly determined for sending operations
- *destination identification* – the identification of the process which receives the message, that is, its rank; it is implicitly determined for receiving operations
- *tag* – an integer value which may be used by the application programmer to distinguish different types of messages; the range of valid tag values is 0, . . . ,UB, where the value of UB is implementation dependent (`MPI_TAG_UB`)

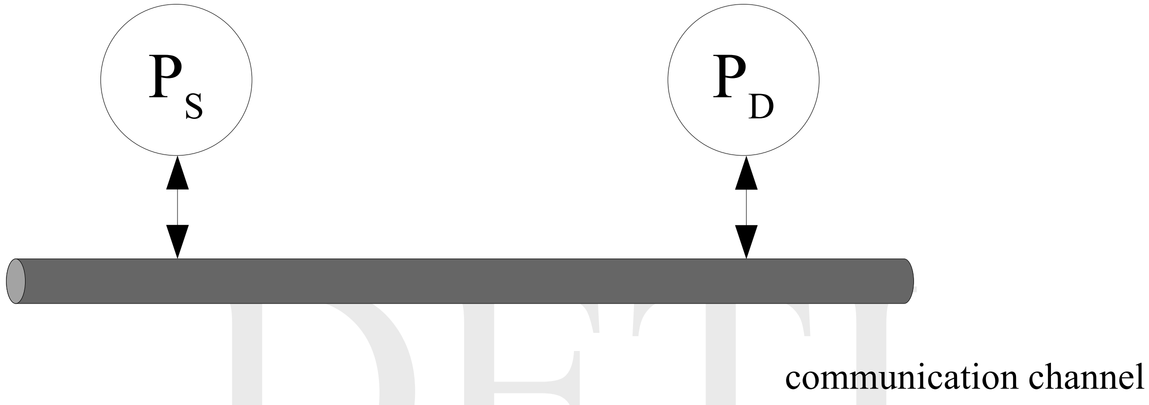
Message format - 3

Information content

- *data type* – the information data type, seen as an array of elements
- *reference to storage buffer* – pointer to the location in memory where the array is stored, or will be stored
- *count* – the number of array elements to be transferred; *count* may be zero, in which case the message content is empty

DETI

Point-to-point communication - 1



- one means a *point-to-point communication* is taking place when a process, called the *source*, sends a message to another process, called the *destination*
- in the standard case, *send* and *receive* are blocking operations, that is, they are synchronized: *send* forwards the message and blocks until the message is effectively received; *receive*, in turn, only returns upon message reception

Point-to-point communication - 2

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

buf – pointer to the memory region where the information content of the message is, or will be, stored

count – number of elements of the array which represents the information content of the message

datatype – MPI information data type

dest – rank of the destination process (send primitive)

source - rank of the source process (receive primitive)
it may be MPI_ANY_SOURCE so that a message from any source is received

tag – message tag, it may be used by the application programmer to distinguish different types of messages
it may be MPI_ANY_TAG so that a message with any tag is received

comm – identification of the communication context (process group)

status - pointer to a structure which defines the receive operation status
it contains at least the following fields: MPI_SOURCE, MPI_TAG, and MPI_ERROR; if the status is not important, the pointer constant MPI_STATUS_IGNORE may be used

DETI

Point-to-point communication - 3

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char *argv[])
{
   int rank;
   char data[] = "I am here!",
        *recData;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   if (rank == 0)
      { printf ("Transmitted message: %s \n", data);
        MPI_Send (data, strlen (data), MPI_CHAR, 1, 0, MPI_COMM_WORLD);
      }
      else if (rank == 1)
              { int i;
                recData = malloc (100);
                for (i = 0; i < 100; i++)
                  recData[i] = '\0';
                MPI_Recv (recData, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
                          MPI_STATUS_IGNORE);
                printf ("Received message: %s \n", recData);
              }
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```


DETI

Point-to-point communication - 4

```
[ruib@ruib-laptop2 basic]$ mpicc -Wall -o sendRecData sendRecData.c

[ruib@ruib-laptop2 basic]$ mpiexec -n 2 ./sendRecData
Transmitted message: I am here!
Received message: I am here!
[ruib@ruib-laptop2 basic]$
```

DETI

Suggested reading

- *Introduction to HPC with MPI for Data Science*, Nielsson F., Springer International, 2016
  - Chapter 2: *Introduction to MPI: The Message Passing Interface*
- *MPI: A Message-Passing Interface Standard* (*Version 3.1*), Message Passing Interface Forum, 2015