MPI: Part I

AQTIVATE Training Workshop I

Dirk Pleiter

CST | EECS | KTH

December 2023

Overview

Content

MPI Introduction

- **MPI** = Message-Passing library Interface specification
  - Model: Data is moved from the address space of one process to that of another process through cooperative operations on each process
- Standard defined by MPI Forum
  - Members of forum: academic and commercial institutions
  - History:

May 1994:	Release of first version MPI-1.0
September 2009:	Release of version MPI-2.2
September 2012:	Release of version MPI-3.0
June 2015:	Release of version MPI-3.1
November 2020:	Publication of MPI-4.0 Release Candidate
June 2021:	Release of version MPI-4.0
November 2023:	Release of version MPI-4.1

MPI Implementations

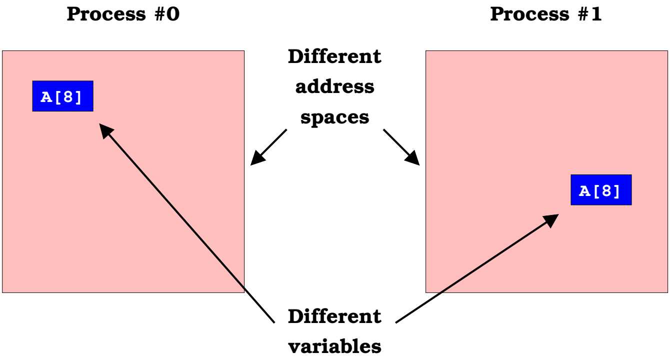
- MPI is only a standard, not an implementation
- Most popular implementations
  - OpenMPI (U Tennesse, LANL, Indiana U plus collaborators)
  - MPICH (ANL plus collaborators)
  - MVAPICH (Ohio State U plus collaborators)
- Different implementations have different advantages/disadvantages. Performance may differ on different platforms and depend on application.

Message Passing Model

- MPI-based parallel programs consist of a set of separate processes
  - A process is an active instance of a program with its own program counter and address space
  - The processes can run on the same node or on different nodes
- Typically all processes execute the same program, but operate on different data
  - SPMD = Single Program Multiple Data
- Data has to be communicated explicitly between processes
  - Different communication patterns are supported:
    - Point-to-point communication operations
    - Collective communication operations
- Programmer has to take care of data distribution, communication and synchronisation

Message Passing Model (2)

Beware: Variables of the same program with the same name in different processes are different variables stored at different memory addresses



Why Using MPI? (1/2)

- **Observation**: MPI has become a standard parallel programming model and is supported on essentially all HPC systems
  - MPI replaced various other message passing solutions
- **Portability**: An application developed in a standard compliant manner on one system will run also on other systems that support an MPI implementation
  - In practice, some system specific optimisations may be required
- **Functionality**: Increasingly rich set of features
- **Wide uptake**: MPI is used for the vast majority of parallel applications leading to
  - Many training opportunities and widely available expertise
  - Broad eco-system

Why Using MPI? (2/2)

- **Availability**: MPI runs on all Linux and most other systems
  - Open source implementations are available and distributed with popular Linux distributions
- **Performance opportunities**: MPI is co-developed with HPC solution providers that perform significant efforts optimising the software
  - Network technology providers provide optimised back-ends
  - HPC system providers provide there own implementations of MPI, e.g. Cray MPI, IBM Spectrum MPI

References (and credits)

- MPI Standard documents: `http://mpi-forum.org/docs/docs.html`
- Current MPI Standard (MPI-4.1): `https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf`
- W. Gropp, E. Lusk, A. Skjellum, “Using MPI: Portable Parallel Programming with the Message-passing Interface”, Volume 1, MIT Press, 1999
- W. Gropp, E. Lusk, R. Thakur, “Using MPI-2: Advanced Features of the Message-passing Interface”, Globe Pequot Press, 1999
- W. Gropp, T. Hoefler, R. Thakur, E. Lusk “Using Advanced MPI: Modern Features of the Message-Passing Interface”, MIT Press, 2014
- M. Hava, “Parallel Programming with MPI”, PRACE Autumn School 2016

Content

First MPI Program: Hello World in C

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

int main (int argc, char *argv[])
{
  int irank, isize;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &irank);
  MPI_Comm_size(MPI_COMM_WORLD, &isize);

  printf("Processor %d of %d: Hello World!\n",
         irank, isize);

  MPI_Finalize();

  return 0;
}
```

First MPI Program: Hello World in F90

```fortran
PROGRAM MPI

IMPLICIT NONE

INCLUDE 'mpif.h'

INTEGER irank, isize, ierr

CALL MPI_Init(ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD, irank, ierr)
CALL MPI_Comm_size(MPI_COMM_WORLD, isize, ierr)

WRITE(*,*) "Processor ", irank, " of ", &
           isize,": Hello World!"

CALL MPI_Finalize(ierr)

END PROGRAM
```

First MPI Program: Comments

- Need to include MPI header files or declare use of MPI module to make MPI functions, variable types and constants declared or defined
  - C or C++: `#include <mpi.h>`
  - Fortran 2008 or later: `USE mpi_f08`
  - Earlier Fortran: `USE mpi` or `INCLUDE 'mpif.h'`
- `MPI_Init` must be the first function that is called (few exceptions)
- `MPI_Finalize` is required to clean-up all MPI states
- C-language binding
  - MPI used through function calls
  - All functions return an error code of type `int`
- Fortran binding
  - MPI used mainly through subroutine but also function calls
  - Last argument is the error code

MPI Initialisation (1/2)

- Single-threaded case

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

  ```
  MPI_Init(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
  ```

- Instead of the pointers `argc` and `argv` one may also provide `NULL`

  ```
  int MPI_Init(NULL, NULL)
  ```

MPI Initialisation (2/2)

- Multi-threaded case

```
int MPI_Init_thread(int *argc, char ***argv,
                    int required, int *provided)

MPI_Init_thread(required, provided, ierror)
  INTEGER, INTENT(IN)                 :: required
  INTEGER, INTENT(OUT)                :: provided
  INTEGER, OPTIONAL, INTENT(OUT)      :: ierror
```

with `required` taking the following values

  - `MPI_THREAD_SINGLE`: Only one thread will execute
  - `MPI_THREAD_FUNNELED`: Only main thread makes MPI calls
  - `MPI_THREAD_SERIALIZED`: Only one threat at a time makes MPI calls
  - `MPI_THREAD_MULTIPLE`: No restrictions

MPI Finalisation

- Routine to clean up all MPI state

```
int MPI_Finalize(void)
```

```
MPI_Finalize(ierror)
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Before calling for finalisation, a process must perform all MPI calls to complete its involvement in MPI communications

MPI Communicators

- MPI **communicators** (more precisely: intra-communicators) allow to group MPI processes
  - Processes within a communicator can communicate among each other
- MPI communicators define a communication context
  - Communications within different communicators will never be mixed up
- Predefined Communicators
  - `MPI_COMM_WORLD`: Group including all processes
  - `MPI_COMM_SELF`: Group containing only the current process

MPI Process Identification: Rank

- All processes within a communicator are assigned a unique identify called **rank**
  - Ranks are integer values starting from zero
- An MPI process may be part of multiple communicators and therefore have multiple ranks
  - Typically, rank refers to the identity of a process within the communicator `MPI_COMM_WORLD`

Compiling MPI Programs

- MPI programs should be compiled using the wrapper compilers provided by the different MPI implementations, e.g.
  - C:
    - mpicc, mpiicc
  - C++:
    - mpicxx, mpiCC, mpic++, mpiicpc
  - Fortran:
    - mpifc, mpif77, mpif90, mpiifort
- Most wrappers allow to show what they are doing
  - Example OpenMPI: `mpicc --showme`

Running MPI Programs

- Starting an MPI program requires a tool called `mpiexec`
  - Not standard compliant alternative: `mpirun`
- Typically, a single MPI is executed:

  ```
  mpiexec -n 8 ./hello_mpi.x
  ```

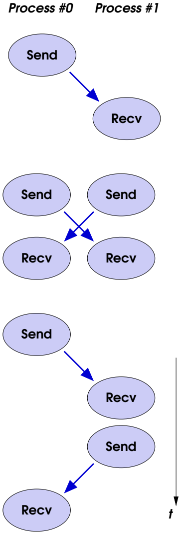
  - The option `-n <numprocs>` defines the number of processes that are started
  - Other options are MPI implementation specific
- MPI also allows to execute multiple programs, e.g.

  ```
  mpiexec -n 1 ./master.x : -n 23 ./worker.x
  ```

Content

Digression: Communication Patterns

- Communication patterns involving a pair of processes
- **Ping**:
  - Process #0 sends a message to process #1 (Ping)
- **PingPing**:
  - Each process sends a message to the other process
- **PingPong**:
  - Process #0 sends a message to process #1 (Ping)
  - After receiving the message process #1 returns it to process #0 (Pong)



Ping in C using MPI

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

int main(int argc ,char *argv[])
{
  int irank;
  int x, y;
  int src, dst;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &irank);

  x = 100 + irank; y = -1;
  src = 1; dst = 0;

  if (irank == src) {
    MPI_Send(&x, 1, MPI_INT, dst, 99, MPI_COMM_WORLD);
    printf("#%d: have sent x=%d\n", irank, x);
  }
  else if (irank == dst) {
    MPI_Recv(&y, 1, MPI_INT, src, 99, MPI_COMM_WORLD, &status);
    printf("#%d: have received y=%d\n", irank, y);
  }

  MPI_Finalize();

  return 0;
}
```

Ping in Fortran 2008 using MPI

```fortran
PROGRAM PING

USE mpi_f08

IMPLICIT NONE

INTEGER          :: irank, ierr
INTEGER          :: x, y
INTEGER          :: src, dst
TYPE(MPI_Status) :: xstat

CALL MPI_Init(ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD, irank);

x = 100 + irank; y = -1;
src = 1; dst = 0;

IF (irank == src) THEN
  CALL MPI_Send(x, 1, MPI_INT, dst, 99, MPI_COMM_WORLD, ierr)
  WRITE(*,*) "#", irank, ": have sent x=", x
ELSE IF (irank == dst) THEN
  CALL MPI_Recv(y, 1, MPI_INT, src, 99, MPI_COMM_WORLD, xstat, ierr)
  WRITE(*,*) "#", irank, ": have received y=", y
END IF

CALL MPI_Finalize();

END PROGRAM
```

Standard `MPI_Send`

- Syntax of a blocking send operation:

```c
int MPI_Send(
  const void* buf,        /* Pointer to send buffer */
  int count,              /* Number of elements */
  MPI_Datatype datatype,  /* Data type */
  int dest,               /* Rank of destination */
  int tag,                /* Message tag */
  MPI_Comm comm           /* Communicator */
);
```

- The **message buffer** is described by `buf`, `count` and `datatype`
- The **target** is identified by `dest` and `comm`
- Messages are sent with an accompanying user-defined integer **tag** that is used by receiver for identification
  - Allow send-receiver pairs to perform concurrent communications

MPI Data Types

- MPI is aware of intrinsic data types in the supported languages (C, Fortran, C++)
- Benefits
  - No need for buffer length in Bytes to be computed by programmer
  - Facilitate some support of type checking
  - Automatic data conversion, e.g.
    - Little versus big endian
    - Interfacing C and Fortran routines

MPI Predefined Data Types

C:

MPI data type	C data type
MPI_CHAR	char
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double

Fortran:

MPI data type	Fortran data type
MPI_CHAR	CHARACTER(1)
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION

Standard `MPI_Receive`

- Syntax of a blocking receive operation:

```c
int MPI_Receive(
  const void* buf,        /* Pointer to receive buffer */
  int count,              /* Number of elements */
  MPI_Datatype datatype,  /* Data type */
  int source,             /* Rank of source */
  int tag,                /* Message tag */
  MPI_Comm comm,          /* Communicator */
  MPI_Status *status      /* Status object */
);
```

- Waits until a matching (on source and tag) message is received
  - After function returned, receive buffer can be used
- Instead of providing a specific source or tag, `MPI_SOURCE_ANY` or `MPI_TAG_ANY` may be used
- The `status` object on return contains further information
- Receiving fewer than `count` occurrences of `datatype` is OK, but receiving more is an error

Return Status Object

- Returned information
  - `MPI_SOURCE`: Source of received message
  - `MPI_TAG`: Tag of received message
  - `MPI_ERROR`: Error code
- Language binding
  - C: Structure with 3 elements, e.g. `status.MPI_SOURCE`
  - Fortran with `USE mpi` or `INCLUDE 'mpif.h'`: Array of length 3, e.g. `status(MPI_TAG)`
  - Fortran with `USE mpi_f08`: Structure with 3 elements, e.g. `status%MPI_ERROR`

Error Codes

- The MPI standard leaves it to the implementations to define specific error codes, which must comply to the following rules
  - Predefined code 0 = `MPI_SUCCESS`
  - Other codes must be in the range [1, `MPI_ERR_LASTCODE`]
- The standard defines a set of error classes and a function to translate error code to error class:

```
int MPI_Error_class(int errorcode, int *errorclass)
```

```
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
  INTEGER ERRORCODE, ERRORCLASS, IERROR
```

- Error class examples:

`MPI_ERR_BUFFER`	Invalid buffer pointer
`MPI_ERR_RANK`	Invalid rank

Standard MPI_Sendrecv

- If all processes send and receive messages (PingPong pattern) send and receive can be initiated using a single MPI call:

```
int MPI_Sendrecv(
 const void *sendbuf,     /* Send buffer */
 int sendcount,           /* Number of elements */
 MPI_Datatype sendtype,   /* Data type */
 int dest,                /* Rank of destination */
 int sendtag,             /* Send tag */
 void *recvbuf,           /* Receive buffer */
 int recvcount,           /* Number of elements */
 MPI_Datatype recvtype,   /* Data type */
 int source,              /* Rank of source */
 int recvtag,             /* Receive tag */
 MPI_Comm comm,           /* Communicator */
 MPI_Status *status       /* Status object */
);
```

Probe Incoming Messages

- MPI allows incoming messages to be checked for, without actually receiving them, using the blocking function `MPI_Mprobe` (and similar variants)

```c
int MPI_Mprobe(
  int source,                /* Source */
  int tag,                   /* Tag */
  MPI_Comm comm,             /* Communicator */
  MPI_Message *message,      /* Message handler */
  MPI_Status *status         /* Status object */
);
```

- Instead of a specific source and/or tag, `MPI_SOURCE_ANY` or `MPI_TAG_ANY` may be used
- Information about the message length can be obtained using `MPI_Get_Count`

```c
int MPI_Get_count(
  const MPI_Status *status,  /* Status object */
  MPI_Datatype datatype,     /* Data type */
  int *count                 /* Message length */
);
```

Probe Incoming Messages: Example

Function that allows to receive messages with an arbitrary number of integers:

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

int recv_int(int* buf) {
    int len;
    MPI_Status status;
    MPI_Message msg;

    /* Blocking check for message, do not yet receive */
    MPI_Mprobe(0, 0, MPI_COMM_WORLD, &msg, &status);

    /* Get message length, allocate buffer and receive */
    MPI_Get_count(&status, MPI_INT, &len);

    buf = (int *) calloc(len, sizeof(int));
    MPI_Mrecv(buf, len, MPI_INT, &msg, &status);

    printf("Recieved %d integers from process 0.\n", len);

    return len;
}
```

Timing Measurement

- MPI defines a timer that returns time in units of seconds:

  `double MPI_Wtime(void)`

  `DOUBLE PRECISION MPI_Wtime()`

- Example program in C:

```c
double tstart = MPI_Wtime();
/* stuff to be timed */
double tend = MPI_Wtime();
printf("Execution time: %.2e seconds\n",tend-tstart);
```

- Time is measured as difference to some time in the past that is guaranteed not to change during process execution
- The times returned are local to the node that called them
- Time resolution can be checked using `MPI_Wtick()`

Time Measurement: Pitfall (1/3)

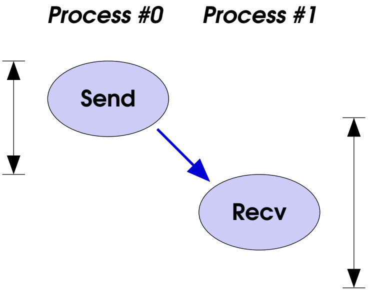
- Will the modified version of the MPI Ping program correctly measure the time to communicate one integer?

```c
src = 1; dst = 0;

double tstart = MPI_Wtime();
if (id == src) {
  MPI_Send(&x, 1, MPI_INT, dst, 99, MPI_COMM_WORLD);
  printf("#%d: have sent x=%d\n", id, x);
}
else if (id == dst) {
  MPI_Recv(&y, 1, MPI_INT, src, 99, MPI_COMM_WORLD, &status);
  printf("#%d: have received y=%d\n", id, y);
}
double tend = MPI_Wtime();
printf("Time for communication: %.2es\n", tend - tstart);
```

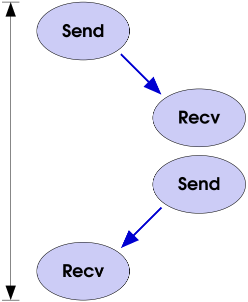
Time Measurement: Pitfall (2/3)

- Answer: No
- Process #0 measures the time it takes to complete the send operation (plus printing a string)
- Process #1 measures the time from an arbitrary starting point until the receive operation completed



Time Measurement: Pitfall (3/3)

- Alternative: Implement a PingPong communication pattern to measure **round-trip time**



Finish with Past Networking Technology

[Wikipedia Commons]

Morse code receiver with paper tape recorder