POLITECNICO
MILANO 1863

Message Passing Interface - MPI

Alessandro Margara
alessandro.margara@polimi.it
https://margara.faculty.polimi.it

History and motivations

Message Passing Interface

- MPI is a standard/specification for high-performance distributed computing (HPC) scenarios
  - It only standardizes an API
  - Several implementations exist
    - OpenMPI, MPICH, …
    - Mostly target C/C++ (and Fortran)

- MPI aims to develop a standard API that is
  - General / flexible
  - Easy to use
  - Portable
  - Efficient

History

- 1980s – early 1990s: parallel and distributed computing becomes a reality but lacks a standard
  - Different solutions …
  - … with different trade-offs between portability, performance, ease of use, …
- 1992: Workshop on Standards for Message Passing in a Distributed Memory Environment
  - Discussion of basic features required by a standard
  - Working group to continue the standardization process
- 1994: MPI 1.0 released
- …
- 2021: current version – MPI 4.0
  - Approved in June 2021

Use cases

- Compute-intensive tasks

- Typical example: computer simulations
  - Molecular modeling
    - E.g., for drug discovery
  - Computational fluid dynamics
    - E.g., climate modeling
  - Agent-based simulation
    - E.g., population dynamics

- Different from data-intensive tasks
  - The target of Apache Kafka, Apache Spark, distributed databases, …

Abstraction

- Single program, multiple processes

- The developer writes a single program
  - Executed in parallel by multiple processes
  - Each process has an associated id (its rank)
    - The developer can use the rank to differentiate the behavior of processes

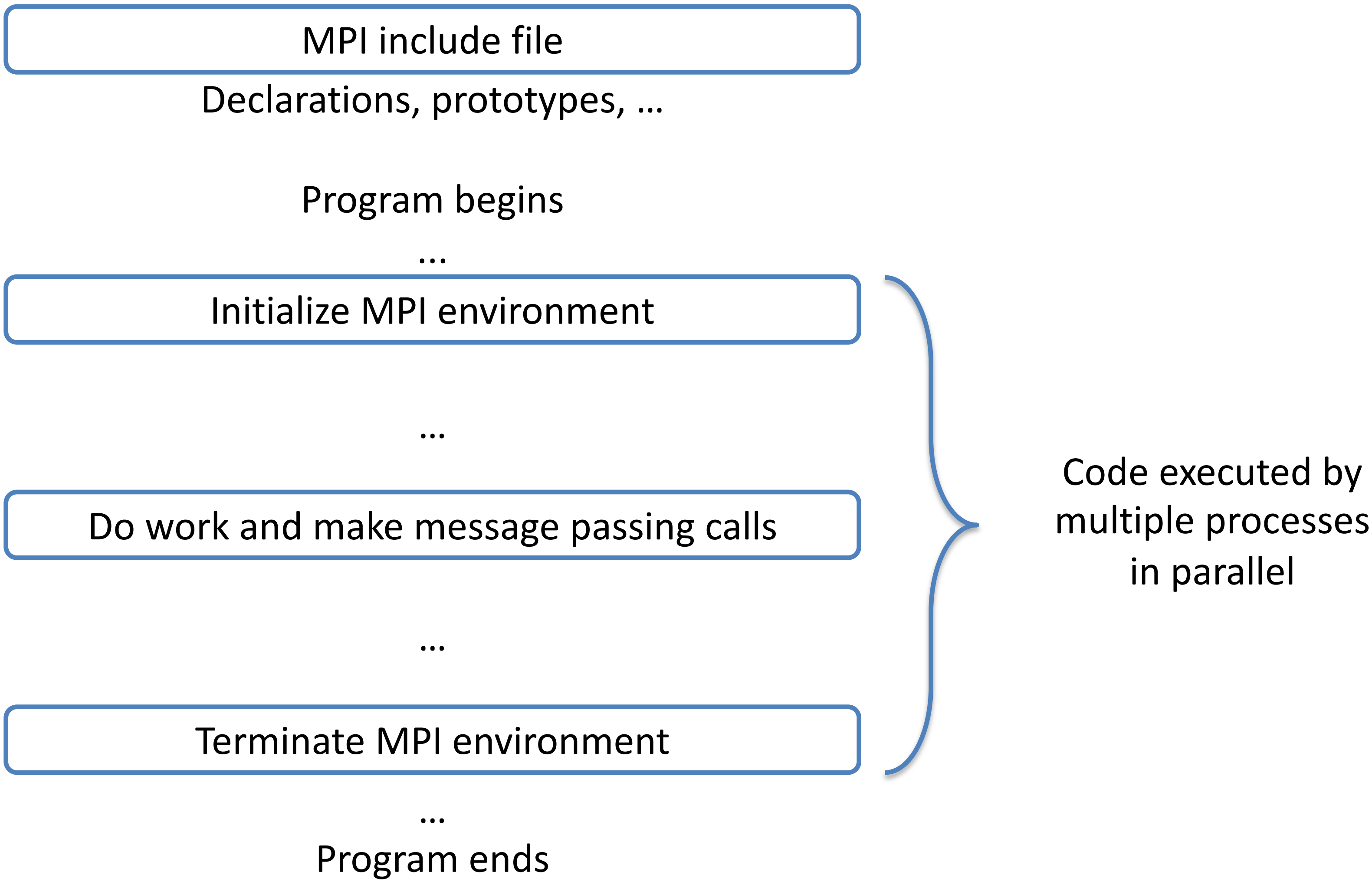
- Point-to-point and collective communication and synchronization primitives

Comparison

Big data platforms	HPC / MPI
Implicit parallelism, synchronization, communication	Explicit parallelism, synchronization, communication
High-level, specialized primitives (relational data, graphs, …)	Low-level, general primitives (send/receive, synchronization, …)
Focus on ease of use	Focus on performance
Fast prototyping	Slow-evolving libraries
Minimize use of resources • Overcommit of resources • Dynamic tasks	Maximize use of resources • No overcommit of resources • Static tasks
Fault-tolerance	No fault-tolerance
Heterogeneous hardware • Stragglers	Homogeneous hardware • Synchronous computation
Overlapping of computation and communication	Bulk synchronous computations

Overview

Structure of an MPI program



Format of MPI calls

- General format

```
rc = MPI_Xxxxx(parameter, ... );
```

- Example

```
rc = MPI_send(&buf, count, type, dest, tag, comm);
```

- Error code is returned as `rc`

```
MPI_SUCCESS if successful
```

Hello MPI

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

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    printf("Hello, MPI!\n");
    MPI_Finalize();
    return 0;
}
```

Compile and run MPI programs

- How to compile and run MPI programs depends on the specific implementation used
- In OpenMPI and MPICH
  - Compile: `mpicc -o hello hello.c`
  - Run: `mpirun [-np <n>][-H <h1,h2,..>] hello`
    - Instantiates n processes on the hosts h1, h2, …
    - All hosts must be reachable by the user with ssh with public key authentication
    - All hosts must have the same version of MPI and the same executable in the same path

Communicators, groups, and ranks

- MPI organizes processes into *groups* that exchange messages over a *communicator*
  - The communicator defines the scope of communication
  - Most MPI routines require the developer to specify a communicator as an argument
  - The `MPI_COMM_WORLD` is the predefined communicator that includes all MPI processes

- Within a communicator, every process has its own *rank*
  - Integer ID assigned by the system
  - Ranks are contiguous and begin at zero
  - Used to specify the source and destination of messages
  - Often also used to control program execution
    - if rank==0 do this / if rank==1 do that

Discovering the MPI environment

- MPI provides several functions to manage and query the environment

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

  - Returns the total number of MPI processes in the specified communicator

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

  - Returns the rank of the calling MPI process within the specified communicator

```
int MPI_Get_processor_name(char *name, int *len)
```

  - Returns the processor name
  - Implementation dependent: might not be the same as the output of the hostname shell command

Hello MPI (again)

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

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

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello, MPI! ");
  printf("I am process %d out of %d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```