The MPI Message-passing Standard Practical use and implementation (I)

SPD Course

23/02/2015

Massimo Coppola

UNIVERSITÀ DI PISA

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

References

- Standard MPI 2.2
  – Only those parts that we will cover during the lessons
  – They will be specified in the slides/web site.
  – Available online : http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
- B. Wilkinson, M. Allen Parallel Programming, 2nd edition. 2005, Prentice-Hall.
  – This book will be also used; the 1st edition can as well do, and it is available in the University Library of the Science Faculty, [C.1.2 w74 INF]

What is MPI

- MPI: Message Passing Interface
  - a standard defining a *communication library* that allows message passing applications, languages and tools to be written in a portable way
- MPI 1.0 released in 1994
- Standard by the MPI Forum
  - aims at wide adoption
- Goals
  - Portability of programs, flexibility, portability and efficiency of the MPI library implementation
  - Enable portable exploitation of shortcuts and hardware acceleration
- Approach
  - Implemented as a library, static linking
- Intended use of the implemented standard
  - Support **Parallel Programming Languages** and **Application-specific Libraries**, not only parallel programs

Standard history

- 1994 - 1.0 core MPI
  - 40 organizations aim at a widely used standard
- 1995 - 1.1 corrections & clarifications
- 1997 - 1.2
  - small changes to 1.1 allow extensions to MPI 2.0
- 1997 - 2.0
  - large additions: process creation/management, one-sided communications, extended collective communications, external interfaces, parallel I/O
- 2008 - 1.3 combines MPI 1.1 and 1.2 + errata
- 2008 - 2.1 merges 1.3 and 2.0 + errata
- 2009 - 2.2 few extensions to 2.1 + errata
- MPI3.0

What do we mean with message passing?

- An MPI program is composed of multiple processes with **separate memory spaces & environments**
- Processes are possibly on separate computing resources
- Interaction happens via **explicit message exchanges**
- Support code provides primitives for communication and synchronization
- The M.P.I., i.e. the kind of primitives and the overall communication structure they provide, constrain the kind of applications that can be expressed
- Different implementation levels will be involved in managing the MPI support

SPMD: single-program multiple-data

- A *basic* MPI program is a **single executable** that is started in multiple parallel instances (possibly on separate hardware resources)
- As already stated, an MPI program is composed of multiple processes with **separate memory spaces & environments**
- Each process has its own execution environment, status and control-flow
- In SPMD C/C++/Fortran programs, sequential data types are likely common to all process instances
- However, variable and buffer allocation as well as MPI runtime status (e.g. MPI data types, buffers) are entirely local
- Understanding (and debugging) the interaction of multiple program flows within the same code requires proper program structuring
- **Changes were introduced with MPI2.0 and over, with dynamic process spawn allowing a full MPMD (multiple-program, multiple data) execution model**

On the meaning of Portability

- Preserve software *functional* behaviour across systems :
  - (recompiled) programs return correct results
- Preserve *non-functional* behaviour :
  - You expect also performance, efficiency, robustness and other features to be preserved

In the "parallel world", the big issue is to safekeep parallel performance and scalability

- Performance Tuning
  - Fiddling with program and deployment parameters to enhance performance
- Performance Debugging
  - Correct results, but awful performance: what happened?
  - Mismatched assumptions among SW/HW layers

What do we do with MPI?

MPI is a tool to develop:
- Applications
- Programming Languages
- Libraries

Much more than the typical usage patterns you can find around on the web!

Interoperation of

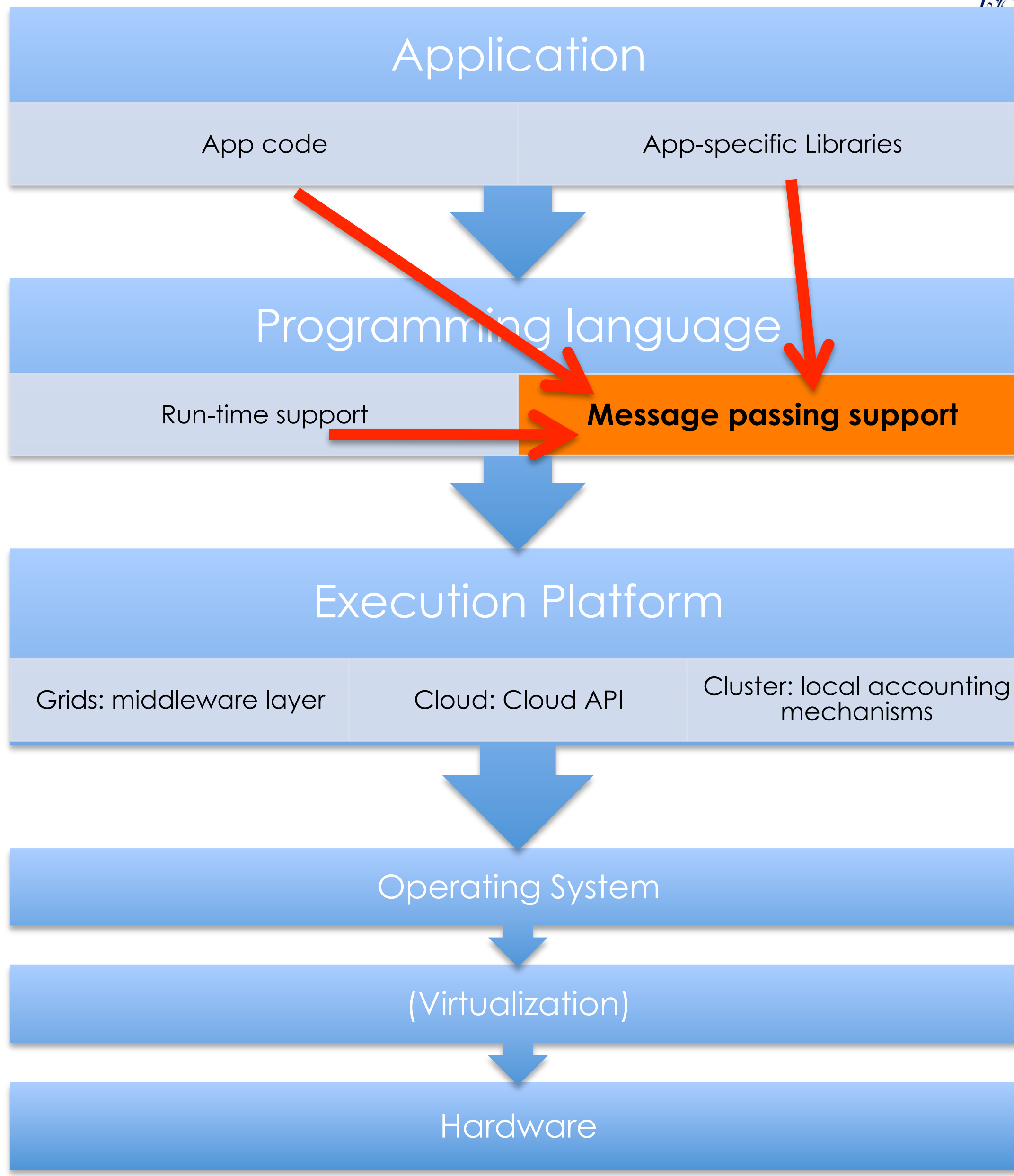
Programming languages (Fortran, C, C++ …)

Heterogeneous resources

Big/little endianness

FP formats

…



MPI functionalities

- MPI lets **processes** in a **distributed/parallel** execution environment **coordinate** and **communicate**
  - Possibly processes on different machines
  - We won't care about threads
    - MPI implementations can be compatible with threads, but you program the threads using some other shared-memory mechanism: pthreads, OpenMP …
- Same MPI library instance can be called by multiple high-level languages
  - Interoperability, multiple language bindings
  - impact on standard definition and its implementation
  - The MPI Library is eventually linked to the program, its support libraries and its language runtime
  - Some functionalities essential for programming language development

Key MPI Concepts

- Communicators
- Point to point communication
- Collective Communication
- Data Types

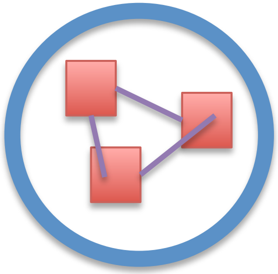
Key MPI Concepts: Communicators

- Communicators
  - Process groups + communication state
  - Inter-communicators vs Intra-communicators
  - **Rank** of a process
- Point to point communication
- Collective Communication
- Data Types

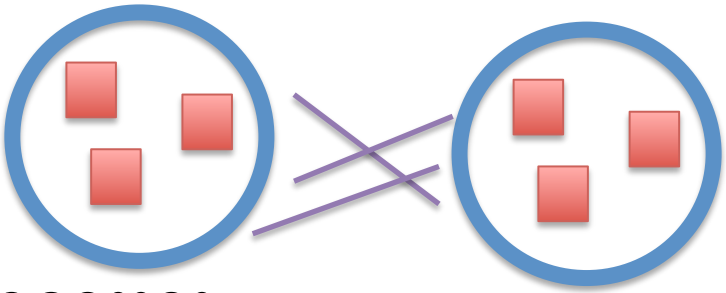
Communicators

- Specify the communication context
  - Each communicator is a separate "universe", no message interaction between different communicators
- A group of processes AND a global communication state
  - Forming a communicator implies some agreement among the communication support of the composing processes
  - A few essential communicators are created by the MPI initialization routine (e.g. MPI_COMM_WORLD)
  - More communicator features later in the course

Types of communicators



- Intracommunicator
  - Formed by a single group of processes
  - Allows message passing interaction among the processes within the communicator



- Intercommunicators
  - Formed by two groups A, B of processes
  - Allows message passing between pairs of processes of the two different groups
    (x,y) can communicate *if-and-only-if*
    x belongs to group A and y belongs to B

Communicators and Ranks

- No absolute process identifiers in MPI
- The **Rank** of a process is always relative to a specific communicator
- In a group or communicator with N processes, ranks are consecutive integers 0…N-1
- No process is guaranteed to have the same rank in different communicators,
    - unless the communicator is specially built by the user

Key MPI Concepts : point to point

- Communicators
- Point to point communication
  - **Envelope**
  - **Local** vs **global completion**
  - **Blocking** vs **non-blocking** communication
  - Communication **modes**
- Collective Communication
- Data Types

Envelopes

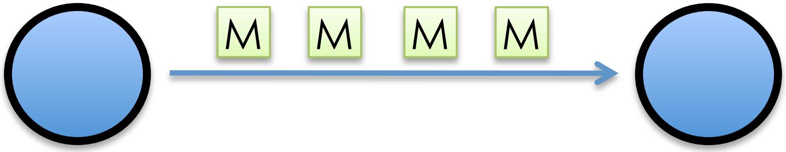
Envelope =

(source, destination, TAG, communicator)

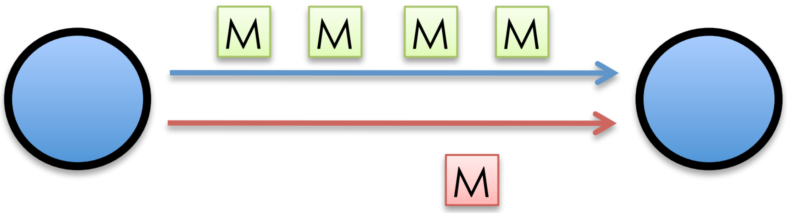
- Qualifies all point to point communications
- Source and dest are ***related*** to the communicator
- Two point-to-point operations (send+receive) match if their envelopes match **exactly**
- **TAG** meaning is user-defined → play with tags to assign semantics to a communication
  - TAG provide communication insulation within a communicator, for semantic purposes
  - Allow any two processes to establish multiple communication "*Channels*" *(in a non-technical meaning)*

Envelopes and comunication semantics

- Messages with the same envelope never overtake each other



- No guarantee on messages with different envelope!



- E.g. : different tags

A first look at the SEND primitive

MPI_SEND(buf, count, datatype, dest, tag, comm)

• IN buf	initial address of send buffer
• IN count	number of elements in send buffer (non-negative integer, **in datatypes**)
• IN datatype	datatype of each send buffer element (handle)
• IN dest	rank of destination
• IN tag	message tag
• IN comm	communicator (handle)

Local and global completion

- **Local completion** : a primitive does not need to interact with other processes to complete
  - Forming a group of processes
  - Asynchronous send of a message while ignoring the communication status
- **Global completion** : interaction with other processes is needed to complete the primitive
  - Turning a group into a communicator
  - Synchronous send/receive : semantics mandates that parties interact before communication happens

Blocking vs non-blocking operations

- Blocking operation
  - The call returns only once the operation is complete
  - No special treatment is needed, only error checking
- non blocking operation
  - The call returns as soon as possible
  - Operation may be in progress or haven't started yet
  - Resources required by the operation cannot be reused (e.g. message buffer is not to be modified)
  - User need to subsequently check the operation completion and its results
- Tricky question: do we mean local or global completion?

Communication MODES

- Synchronous
  - Follows the common definition of synchronous communication, first process waits for the second one to reach the matching send/receive
- Buffered
  - Communication happens through a buffer, operation completes as soon as the data is in the buffer
  - Buffer allocation is onto the user AND the MPI implementation
- Ready
  - Assumes that the other side is already waiting (can be used if we know the communication party already issued a matching send/receive)
- **Standard**
  - The most common, and less informative
  - MPI implementation is free to use any available mode, i.e. almost always Synchronous or Buffered

Example: portability and modes

- Standard sends are implementer's choice
  - Choice is never said to remain constant…
- A user program exploit standard sends, implicitly relying on *buffered* sends
  - Implementation actually chooses them, so program works
- What if
  - Implementation has to momentarily switch to synchronous sends due to insufficient buffer space?
  - Program is recompiled on a different MPI implementation, which does not use buffered mode by default?

Combining concepts

- Point to point concepts of ***communication mode*** and ***non-blocking*** are completely orthogonal : you can have all combinations

- local / global completion depends on
  - The primitive (some inherently local/global)
  - The combination of mode and blocking behavior
  - The MPI implementation and the hardware always have the last word

- We will be back to this later on in the course

Key MPI Concepts : Collective op.s

- Communicators
- Point to point communication
- Collective Communication
  - A **whole** communicator is involved
  - Always locally blocking *
    - *No longer true since MPI 3.0, but we will ignore this in this course*
  - No modes: collectives in a same communicator are **serialized**
- Data Types

Collective operations - I

- Basically a different model of parallelism in the same library
- Collectives act on a **whole** communicator
  - All processes in the communicator must call the collective operation
  - With compatible parameters
  - *Locally the collectives are always blocking (no longer true since MPI 3, but outside course scope)*
- Collective operations are **serialized** within a communicator
  - By contrast, point to point message passing is intrinsically concurrent
  - No communication modes or non-blocking behaviour apply to collective operations

Collective operations - II

- Much detail is left to the implementation
  - The standards makes minimal assumptions
  - Leaves room for machine specific optimization
- Still **No guarantee** that all processes are actually within the collective at the same time
  - Freedom for MPI developers to choose the implementation algorithms: collective may start or complete at different moments for different processes
  - MPI_Barrier is of course an exception

Key MPI Concepts : Datatypes

- Communicators
- Point to point communication
- Collective Communication
- Data Types
  - A particular kind of **Opaque objects**
  - MPI **primitive** datatypes
  - MPI **derived** datatypes

Opaque objects

- Data structures whose exact definition is hidden
  - Obj. internals depend on the MPI implementation
  - Some fields may be explicitly documented and made accessible to the MPI programmer
  - Other fields are only accessed through dedicated MPI primitives and object **handles**
  - Allocated and freed (directly or indirectly) only by the MPI library code
    - If the user is required to do so, it has to call an MPI function which is specific to the kind of opaque object
  - Example:
    Communicators and datatypes are Opaque Obj.

Primitive Datatypes

- MPI Datatypes are needed to let the MPI implementation know how to handle data
  – Data conversion
  – Packing data into buffers for communication, and unpacking afterwards
  – Also used for MPI I/O functionalities
- Primitive datatypes
  – Correspond to basic types of most programming languages: integers, floats, chars…
  – Have bindings for MPI supported languages
  – Enough for simple communication

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

MPI derived datatypes

- Derivate datatypes correspond to composite types of modern programming languages
  - Set of MPI constructors corresponding to various kinds of arrays, structures, unions
  - Memory organization of the data is highly relevant, and can be explicitly considered
  - Derived datatypes can automate packing and unpacking of complex data structures for communications, and allow semantically correct parallel operation on partitioned data structures

FILLING IN THE GAPS

Beware

- MPI uses a different abstraction than physical / logic channels, the one you know from previous courses
- When we speak of "channels" in MPI we mean the set of messages sharing the same envelope and some ordering constraint
- There is not such thing as an implementation of the channel defined or referenced in the MPI standard
- The two abstractions have different goals, but the implementation issues are the same: HW features, coprocessors, zero copy…
- You are expected to understand both and not confuse them

From Send and Recv to programs

- Simplest programs do not need much beyond Send and Recv
- Keep in mind that each process lives in a separate memory space
  - Need to initialize all your data structures
  - Need to initialize your instance of the MPI library
  - Should you make assumptions on process number?
  - How portable will your program be?

Running, Initializing the runtime

- Basic process spawning is done by the MPI launcher:
  `mpirun` [*mpi options*] <**`program _name`**>[*arguments*]
  - Check the mpirun man page of your MPI implementation

Each MPI process calls AT LEAST

- MPI_Init(int *argc, char ***argv)
  - Shall be called before using any MPI calls (very few exceptions)
  - Initializes the MPI runtime for all processes in the running program, some kind of handshaking implied
    - e.g. creates **MPI_COMM_WORLD**
- MPI_Finalize()
  - Frees all MPI resources and cleans up the MPI runtime, taking care of any operation pending
  - Any further call to MPI is forbidden
  - some runtime errors can be detected at finalize
    - e.g. calling finalize with communications still pending and unmatched

References

- MPI 2.2 standard (see http://www.mpi-forum.org/)
  - Only some parts
- Parallel Programming, B. Wilkinson & M. Allen. Prentice-Hall (2nd ed., 2005)
  - Only some references, 1st edition is ok too.

- Relevant Material for 1st lesson, MPI standard
  - Chapter 1: have a look at it.
  - Chapter 2:
    sec. 2.3, 2.4, 2.5.1, 2.5.4, 2.5.6, 2.6.3, 2.6.4, 2.7, 2.8
  - Chapter 3:
    sec. 3.1, 3.2.3, 3.4, 3.5, 3.7