MPI

Lorenzo ANARDU

02-03/03/2010

1 Overview and Goals

MPI (Message Passing Interface) is a message-passing library interface specification. MPI is a specification, not an implementation. Multiple implementations exist of MPI (such as MPICH, OpenMPI) that are often very efficient (they can be optimized for specific network hardware on which they run).

MPI was first released on 1994. It is a standard for a message-passing communication between processes which model a parallel program on a distributed memory system; the programming models can be adopted on clusters and multiproessors, but it can also be exploited on multicore computers and SMP machines. The standard has been defined through an open process by a community (MPI Forum) involving about 40 organizations.

The principal aim of MPI project was to define a widely adopted layer in the software architecture, providing message passing capabilities, on top of which applications, application libraries and languages could run.

Internally, we can think of MPI as structured in 2 sub-layers:

- a core MPI layer, which consists of a small number of functionalities (it has been shown that 6 unavoidable primitives are enough to build the whole MPI standard);
- the rest of the MPI library, which consists of about 200 primitives implemented on top of the core MPI primitives.

This abstract structure has been actually exploited, for instance in the MPICH implementation of MPI, in order to achive maximum portability. This structure allows implementors to port implementations to a lot of different architectures modifying a relatively small part of the rest of the standard.

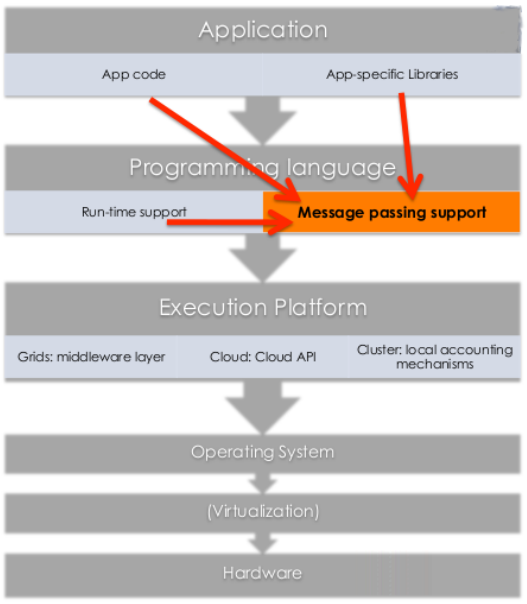
MPI is a standard designed to be language independent, and to provide interoperability with several languages. Specifications of the MPI API have been defined for ANSI C, C++ and Fortran.

This fact had a deep impact in the design of the standard and in its implementations. As a matter of fact in several cases MPI has to deal with problems related to languages interoperability (types representation, conversion, etc) or execution environment such as number representation (little/big endian, floating point formats, etc).
All primitives are expressed as functions, subroutines or methods, according to the appropriate language bindings.

Currently the standard has a couple of main versions (called MPI-1 and MPI-2). MPI-2 is obviously a superset of MPI-1, except for a few functions which have been deprecated. Implementations of MPI usually support both standards, so they can be seen a mix of the two versions. The main advantages of establishing a message-passing standard are portability, ease of use, high performance and scalability.

2 Architecture and functionalities

The MPI standard is intended for use by all those who want to write portable message-passing programs in the supported languages (individual programmers, software developers, creators of environments). Because of this the standard must provide an easy-to-use interface without precluding high-performance message-passing operations on advanced machines.
Programmers using MPI implementations are responsable for identifying parallelism and implementing algorithms through MPI primitives. The number of tasks dedicated to run in parallel is static, it is impossible adding new tasks dynamically. The programmer must also explicitly expresses the interaction between processes.
The processes that define a parallel application can run on different nodes or several processes can run on the same node (for maximum performance each CPU, or core in a multicore machine, will be assigned to a single process only) and the communication via MPI works alike. The possibility that the processes can run on different processors is hidden by the MPI implementations (that is one of the standard's aims).
Communication in multithreaded systems works in a different way: only one thread per process is explicitly chosen to perform the communication. Actually MPI implementations can be compatible with threads, but threads are programmed using other shared-memory mechanisms (pthread, OpenMP, etc). During the course we will assume to have a single thread per process.
An application which uses only MPI for inter-process communications should be easily portable to different OS and hardware architectures. This is guaranteed by the standard's structure: just MPI implementation exploits the hardware and the OS for performing communications, the user does not need to know anything about lower levels. MPI belongs in layers 5 and higher of OSI reference model.



For what concerns portability of parallel prorgrams, we may attempt to preserve both functional behaviour (meaning that a program written for a specific architecture will produce the same results if it is recompiled on different ones) and non-functional behaviour (meaning that also performance, efficiency and other features are preserved among different architectures). The second goal heavily depends on how the program was written and on the implementation of the parallel support (MPI in our case). Solutions to this problem may be to use approaches like performance tuning for obtaining about the same performances on different architectures, or performance debugging for inspecting and analyzing distributed memory parallel programs.

3 Concepts

Before explaining MPI primitives meaning and utilization it is important to focus on the MPI fundamental concepts.

3.1 Communicators

Communicators are objects connecting groups of processes in the MPI session. A communicator specifies the communication context, that is it defines a group of processes which takes part to the communication. Each communicator is a separate virtual universe, there are no message interactions between different universes. Communicators are tools designed to let MPI users structuring their own applications in different processes groups executing distinct modules and communicating with other groups in a pipeline or a module-graph.
MPI provides some default communicator. MPI_COMM_WORLD is a global communicator including all the processes within the application, it is automatically created in the initialization routine. Another default communicator is MPI_COMM_SELF which, for every process, includes only the process itself.
Communicators can be partitioned in:

- intracommunicators: an intracommunicator is composed by a group of valid participants and allows message passing interactions between processes within the communicator;
- intercommunicators: an intercommunicator is composed by two groups A, B of processes and allows message passing between pairs of processes of the two groups;

Within a communicator each contained process has an independent identifier, called rank. In a group of N processes , ranks are consecutive integers between 0 and N-1. This means that absolute process identifiers in MPI doesn't exist: no process is guaranteed to have the same rank in different communicators. Ranks are practically used to specify the source and destination of a communication and to control program execution (e.g. if(rank==0) { do something } else { do something else }).

CommProc	A	B	C	D
WORLD	0	1	2	3
LITTLE		0	1	

Table 1: Ranks sample

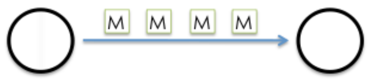
3.2 Point-to-point communication

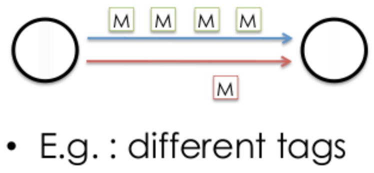
Sending and receiving messages by processes are the basic MPI communication mechanisms. The basic point-to-point operations are **send** and **receive**. Point-to-point communication in MPI relies on several different concepts.
The first one is the concept of envelope. The envelope qualifies all point-to-point communications. It can be defined as a tuple

Envelope = (source, destination, TAG, communicator)

- source and destination are process ranks related to the communicator;
- TAG is an additional ID that allows two processes to establish multiple communication channels (distinguishing different types of messages).

It is important to understand that two point-to-point operations match if their envelopes exactly match. MPI guarantees messages delivery order if they have the same envelope.





Another important concept relating point-to-point operations is the completion of an operation. A procedure is **local** if completion depends only on the local executing process such as forming a group of processes or the asynchronous send of a message ignoring results. Otherwise a procedure is **global** if completion of the operation requires interaction with other processes such as turning a group into a communicator or a synchronous sendreceive of a message.

Further classification relates to blocking and non-blocking operations. In a **blocking** operation the call returns only once the operation is complete, no special treatment is needed except error checking; this means that user is allowed to reuse resources specified in the call. In a **non-blocking** operation the call returns as soon as possible; this means that the procedure may return before the operation completes (even it is not still started). In this case the user is not allowed to reuse resources required by the operation and he has to verify the completion of the operation before using resources.

MPI standard provides four different communication modes:

- synchronous: it follows the common definition of synchronous, the first process waits for the second one to perform the matching send/receive;
- buffered: communication happens through a buffer, the call returns as soon as the data is in the buffer. Buffer allocation is onto the user, but it is hidden by MPI implementation;
- ready: it assumes that the other part is already waiting;
- standard: the most common one, it allows MPI implementation to chose one between the other three modes (often synchronous or buffered). The choice is made by the implementer: it is not necessary constant, this may cause portability problems.

3.3 Collective

Collective operations involve communication among all processes into a communicator. Therefore they act on a whole communicator: all processes in the group need to invoke the operation. Collective calls are serialized within a communicator: this means that a sequence of collective calls over the same communicator must be executed in the same order by all the members of the group. Distinct collective operations cannot overlap.

A collective operation may or may not be synchronizing. A non synchronizing collective operation may start or complete at different time for different processes. This does not happen for synchronizing operations such as MPI_Barrier.

3.4 Data Types

Data types are defined in MPI for two reasons:

- they let library implementation know how to handle the data (type conversion, packing/unpacking into buffers, etc);
- they allow the library to dynamically know which types are being used. MPI was designed as a static linked library. At the time it was developed it was not possible to infer data types at runtime (as it happens in VM languages).

Before explaining MPI data types it is useful to introduce the concept of opaque object. Opaque objects are objects which are defined in (and accessible by) the implementation of the library. The library users don't know their size and shape and either can't directly access the memory in which they are allocated. Opaque objects are accessed via handles, which are allocated in the user's space. The allocation and setting-free of these objects is performed only by the MPI code. Safeness of deallocation operations is guaranteed by MPI: user can only request to deallocate an object and the request makes the object inaccessible. The operation will be performed only when all the pending operations on the object will be completed.
MPI provides primitive data types corresponding to basic types of most programming languages.
MPI also allows user to define very complex data types such as structures, multi-dimensional arrays, and much more. Derived data types can manage packing and unpacking of data structures for communication, and allow semantically correct parallel operations on partitioned data structures.

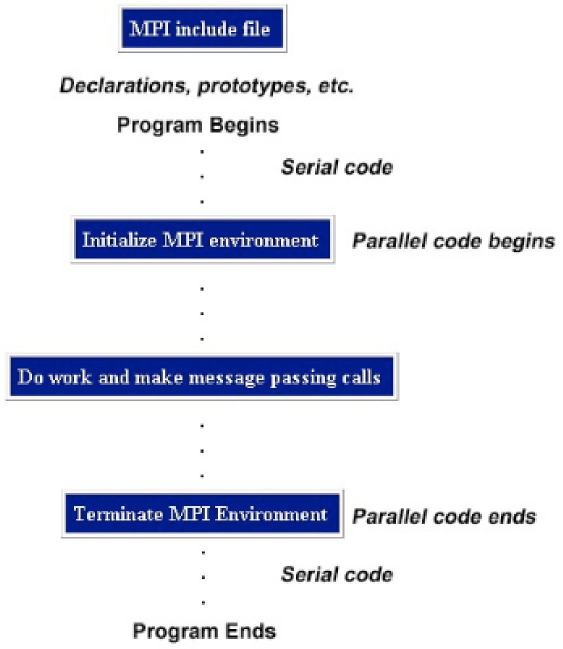
4 Practical use and implementation

MPI standard does not guarantee fairness, it is up to the programmer to prevent operation starvation.
MPI programs have a similar structure, as indicated by the following figure:

After MPI_Initialize call, which initializes MPI environment, the parallel code begins. Parallel processes must not have the same behaviour: generally the processes' semantic is controlled through process ranks. For complex problems solving, it is possible to have several phases using different forms of parallelism (farms, pipelines, graphs, etc), in these cases conceptual modules are separated through communicators. The parallel code is terminated by the MPI_Finalize routine, which frees all the space occupied by opaque objects that have not been deallocated yet.

4.1 Communication primitives

Sending and receiving of messages by processes are the basic MPI communication mechanisms. The basic point-to-point communication primitives are send and receive.
Generally communication can be performed using different formalisms, or their combinations:


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work and make message passing calls
Terminate MPI Environment
Parallel code ends
Serial code
Program Ends

- synchronous vs asynchronous send and receive;
- symmetric vs asymmetric **send** and **receive**.

In MPI, users can choose between a synchronous/asynchronous send and a symmetricasymmetric receive.
The standard specifies the routine's interfaces for all the supported languages. In these notes we will only use the pseudo-language notation.

4.1.1 Send

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

This is the standard blocking send. This means that MPI implementation will choose between the other sending modes. The location, size and type of the message are defined by the first three parameters of the call. In addition, the send operation associates an envelope with the message. The last three parameters specify the envelope for the sent message. This envelope specifies the message destination and supplies distinguishing informations for the receiver.

4.1.2 Receive

MPI_RECV (buf, count, datatype, source, tag, comm, status)

This is the standard blocking receive. The parameters meaning is the same as the send operation but in this case there is an additional parameter which contains the status of the operation. The type of the status is MPI-defined, its implementation depends on the language (e.g. in C it is a struct, in Fortran it is an array of INTEGER). Status variables need to be explicitly allocated by the user: this means they are not system objects.
The receive buffer consists of a set of count locations of the type specified by datatype starting at the

address buf. The length of the received message must be less than or equal to the length of the receive buffer, otherwise an overflow error occurs. If the message is shorter than the buffer only the location corresponding to the message is modified.
The receiver can specify a wildcard MPI_ANY_SOURCE value for the source, and/or a wildcard MPI_ANY_TAG for tag value. No wildcards for communicators exist, thus a process can receive a message only if the message is addressed to it.
Note the asymmetry between send and receive: a receive operation may accept messages from an arbitrary sender, on the other hand a send operation must specify a unique receiver.

4.2 Data Types primitives

MPI provides a large range of native data types, corresponding to almost all languages' supported types: characters, integral numbers (short, long, signed, unsigned, different sizes), floating point numbers and also complex numbers. Two particular native types are MPI_BYTE and MPI_PACKED.
Conversion of type representations among different architectures or languages is automatically and implicitly done by MPI implementations (excepted MPI_BYTE and MPI_PACKED types). In this sense it is important to have some type-matching rule. MPI defines type-matching rules in two directions:

- matching of types of a send-and-receive operation: types match if both operations use identical names (e.g. MPI_INTEGER matches MPI_INTEGER). An exception is that MPI_PACKED can match any other type;
- matching of types of the host language with types specified in communication operations: a type matches if the type used in the data type name corresponds to the basic type of the host program variable (e.g. MPI_INTEGER matches a C variable of type int). Exceptions to this rule are represented by MPI_BYTE and MPI_PACKED which can be used to match any byte of storage (on a byte-addressable machine).

To summarize, the type matching rules distinguish communications in three fields:

- communication of typed values (with type different from MPI_BYTE) where the types used in the send and in the receive call must match;
- communication of untyped values where both sender and receiver use type MPI_BYTE. In this case the data types match is not required. This may cause problems related to architectural features, such as little or big endian representations of integers;
- communications involving packed data, where MPI_PACKED is used. Because of this might seem that there are no differences between MPI_BYTE and MPI_PACKED. The principal differences between the two is that MPI_PACKED is subject to conversion among different architectures, although it is explicitly performed by MPI_PACK and MPI_UNPACK routines; on the other hand MPI_BYTE is never converted.

4.2.1 Derived data types

Up to here, all point-to-point communications have involved only buffers containing a sequence of identical basic data types. MPI provides also mechanisms to specify mixed and noncontiguous data structures, which are automatically packed into communication buffers when communication is performed. The general mechanisms provided allows to transfer directly, without copying, objects of various shape and size.

It is not assumed that MPI is cognizant of the objects declared in the sequential host languages. Thus, if the user wants to transfer a derived data type (a structure, or an array section) he will have to provide to MPI a definition of the data structure (as a derived MPI datatype) that mimics the definition of the language specific data type in question. An **general data type** is an opaque object, which encapsulate the type definition and properties; the opaque object internally specifies at least **type map** information

- a sequence of basic data types;
- a sequence of integer (byte) displacement.

It is important to understand from the beginning that the displacements are not needed to be positive, distinct, or in increasing order. Therefore, it is also important to understand that the order of items need not to coincide with their order in the store, and an item may appear more than once. We call such sequence of pairs a "type map", and the sequence of data types (ignoring displacements) "type signature":

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$$

$$Typesig = \{type_0, ..., type_{n-1}\}$$

This typemap, associated with a base address buf, specifies a the data to be communicated. Usually this data is to be packed into a communication buffer. Such a buffer consists of n entries , where the i-th entry is at address $buf + disp_i$, and has type $type_i$.
Most data type constructors have replication count or block length arguments. Allowed values are non-negative integers: if the value is zero, no elements are generated in the type map and there is no effect on data type bounds or extent.

Handles to general data types can be used in all send and receive operations. In this view basic data types are predefined general data types (e.g. MPI_INT is a predefined handle to a data type with typemap = {(int,0)}). Another important concept related to general data types is the concept of extent. The extent of a data type is defined to be the span between the first byte to the last byte occupied by entries in the data type (eventually rounded for satisfy alignment requirements).

extent(Typemap) = lstByte(TypeMap) fstByte(TypeMap)

Each process in a program must define by itself the derived data types it want to use, because each process must communicate (through the type definition) to its implementation of the library the shape of the derived data type. The communication of a type to the library implementation is performed through the MPI_TYPE_COMMIT routine. In communications signatures of the data types must coincide whereas implementations may differ.
Let's see some data type constructor:

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

This is the simplest data type constructor: it allows replication of a data type into contiguous locations. newtype is the data type obtained concatenating count copies of oldtype. Concatenation is defined using extent as the sizes of the concatenated copies.

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

This is a more general constructor that allows replication of a data type into locations that consist of equally spaced blocks. Each block is obtained by concatenating blocklength copies of the oldtype. The stride parameter is an integer which indicates the number of elements between the start of each block. It can also be less or equal than zero.
This parameter allows the programmer to define types in a very flexible way: types for define parts of matrices (diagonals or arbitrary elements), matrices with replicated data, etc. can be defined.
A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), n arbitrary.
It exists an h-version for this function: MPI_TYPE_HVECTOR with the only difference that stride parameter expresses the distance in bytes instead of in number of elements.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

This constructor allows replication of an old data type into a sequence of blocks (each block is a concatenation of oldtype), where each block can contain a different number of copies and has a different displacement. This may be useful to define particular matrices, such as triangular matrices.
For this constructor exists an h-version too: MPI_TYPE_HINDEXED, with the only difference that the array of displacements is specified in bytes instead of in number of elements.

In general the use of the h-version of these methods is unsafe because there is the risk of producing errors during the type conversion in send and receive operations.