MPI + OpenMP

Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING

2021-2022 @ Università di Trieste

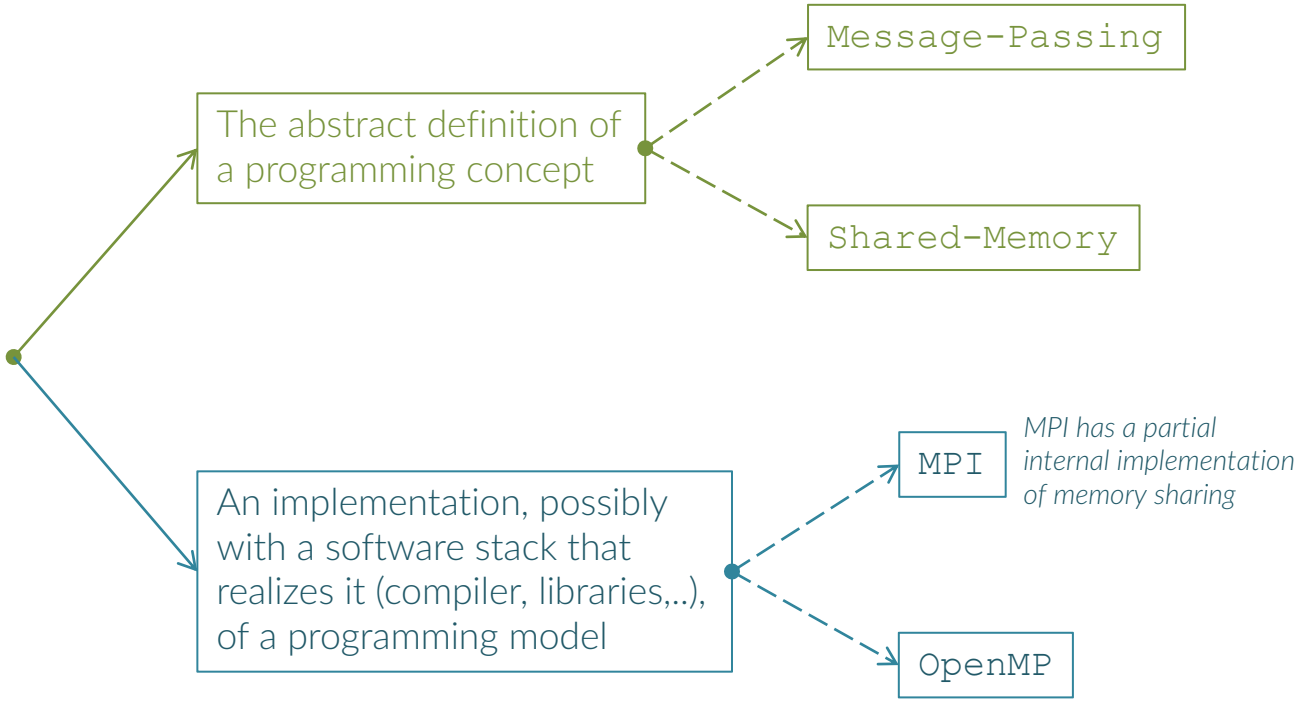
Advanced Parallelism Outline

Hybrid codes
MPI + OpenMP

A handy definition of hybrid

Hybrid programming:

Having more than one *programming models (paradigms)* or *programming systems* in the same code



A handy definition of hybrid

At the time MPI was designed, threads were of course already existent and used.
At odds with other message-passing implementation, MPI was conceived to be *thread-safe*(*) by design, purposely to encourage hybrid programming.
For instance, that includes not to have a concept like the “current” message, but considering each message as an object itself.

However, the programmer still have to ensure that accesses to data that are shared must be properly implemented/protected

Even if the threads model inside message-passing enables the concurrency within the distributed parallelism, the messages are always exchanged at process-level and not at thread-level: in other words, a thread can perform an MPI call on behalf of its father process, but it will always address another MPI process and not a precise thread inside that process.

- Non-blocking communications
  - Overlapping of computation and communication
  - Hiding of communication latency

- Enables exploiting SMP

A handy definition of hybrid

(*) *thread-safe* by design means that multiple threads from an MPI process can perform MPI calls without interfering with each other; that is possible because MPI is designed so that all the information relative to a message is encapsulated in that same message objects and does not reside anywhere in the library common space.

However, that is not enough.
The thread library (*) must have the ability to yield the execution from one thread to another (for instance when a thread is executing a blocking operation of any kind – MPI, system call, I/O,…).
The programmer must be aware of this and correctly implement the use of threads.

Beware: all the caveats that we encountered about OpenMP programming, still hold in hybrid programming (memory races, false sharing, threads overhead, threads placing, …)

(*)*POSIX pthread, the most widely used in *nix systems; it is not the only possibility (Java, C++11 also have their native implementation)*

Few things to remember

- System calls

  Remember that explicitly calling system calls may lead to portability issues due to different behaviour of user and kernel threads on different systems. Using only MPI calls is instead portable.

- Competing for closing the same call

  MPI does not allow two threads to compete for closing the same non-blocking MPI call. It is instead permitted that a thread inits a call that is subsequently closed by a different one.

Initialize MPI for multithreading


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

This function initializes the MPI library with the required level of support, and give back the granted support level.
The allowed levels are the following:

MPI_THREAD_SINGLE	Only the main thread will be running
MPI_THREAD_FUNNELED	Many user threads, only the main one calls MPI
MPI_THREAD_SERIALIZED	Many user threads, only one at a time makes MPI calls
MPI_THREAD_MULTIPLE	Many user threads, any number of threads can make MPI calls at a time

By checking the returned support, you may choose the right MPI library to link with.
`MPI_Init()` actually is a shortcut for `MPI_Init_thread()` with `MPI_THREAD_SINGLE`; hence, whichever you use, you call `MPI_Finalize()`.

Who calls who

If a routine is called, it may need to understand what is

1. The threading support level
   that is achieved by calling `MPI_Query_thread(int *provided)`, which returns the provided level

2. Whether the calling thread is the main thread
   (i.e. the thread that `called MPI_Init_thread()`)
   that is achieved by calling `MPI_Is_thread_main(int*flag)`

A very important clarification

The support level that you require, which may perhaps change at run-time in different runs of your code, IS NOT a requirement neither on the MPI nor on the OpenMP standards.

Then, if you require `MPI_THREAD_FUNNELED` or `MPI_THREAD_SERIALIZED`, that does not mean in any way that **neither MPI nor OpenMP are instrumenting your code so that the MPI calls are made accordingly to the required level**.
That requirement is only a notice to the MPI library which will optimize its internal behaviour accordingly.

You are still in charge of ensuring the correctness of your hybrid code.

MPI and OpenMP

TIPS

The MPI standard does NOT require the environmental variables to be propagated to every process by the MPI itself (although it is a quite common case that a specific implementation does it anyway).

If you use a threading library that allows to specify the number of threads to be created by usage of env vars, like OpenMP, you should explicitly take care of this by retrieving the env vars values with Process 0 and then propagating them to the other Processes (alternatively, you do not use env vars and require a given number of threads in a different way).

Probing messages

If you need to probe a message by using `MPI_Prob/MPI_Iprobe` routines with a support level ≥ `MPI_THREAD_SERIALIZED`, you must use instead the thread-safe routine that has been introduced starting from MPI 3.0, and the related "m" routines:

```
MPI_Mprobe( int source, in tag, MPI_Comm comm,
            MPI_Message *message, MPI_Status *status )

MPI_Improbe( int source, in tag, MPI_Comm comm, int *flag,
            MPI_Message *message,MPI_Status *status )
```

Once a message has been m-probed, it can not be matched by other probe or receive operation; it must instead be matched by either `MPI_Mrecv()` or `MPI_Imrecv()`.

that’s all, have fun