Parallel Programming with MPI

Part VI - Introduction to MPI+OpenMP hybrid programming

HPC SCHOOL
COSTA RICA

CINECA

RISC2

CeNAT
Centro Nacional de Alta Tecnología

ALESSANDRO MARANI

a.marani@cineca.it || San José 2023

AGENDA

WHY MPI+OPENMP?
Architectural trends, pro and cons of pure parallel models, pro and cons of hybridization

HYBRID PROGRAMMING
MPI_Init_thread support, thread support levels, dealing with MPI_THREAD_MULTIPLE

IMPLEMENTATION NOTES
Benefitting applications, implementation and thread support, resource managing and thread affinity

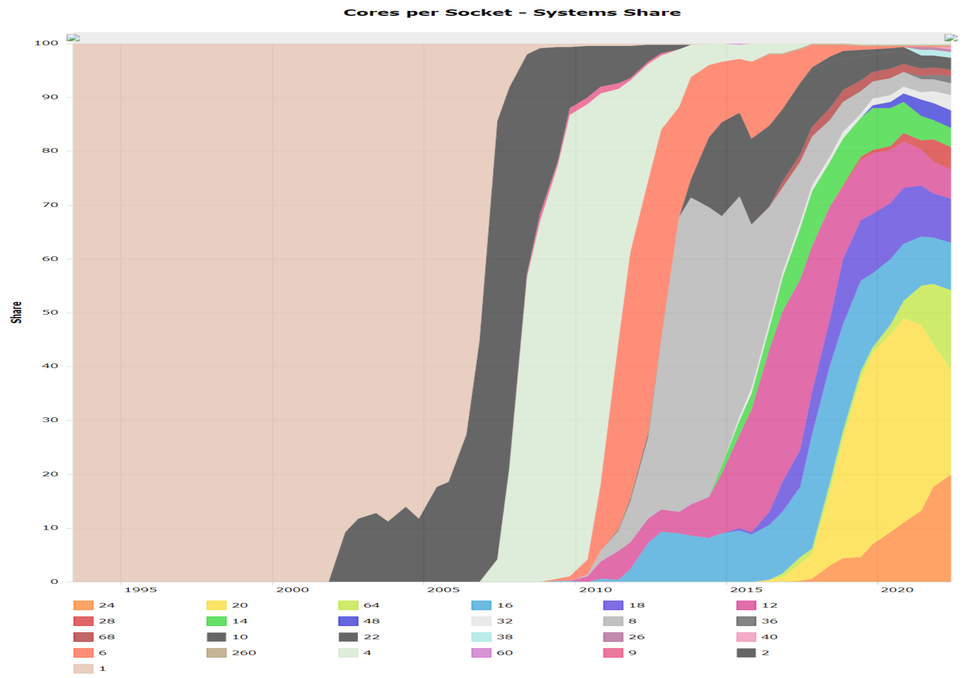
KABRÉ
SUPERCOMPUTADORA

Why MPI+OpenMP?

Architectural trend

- In a nutshell:

  - memory per core decreases
  - memory bandwidth per core decreases
  - number of cores per socket increases
  - single core clock frequency decreases



- Programming model should follow the new kind of architectures available on the market: what is the most suitable model for this kind of machines?

Programming models: MPI

Distributed parallel computers rely on MPI:

- strong
- consolidated
- standard
- enforces the scalability (depending on the algorithm) up to a very large number of tasks

But... is it enough when memory is such small amount on each node?

Example: a machine equipped with 16GB per node and 16 cores. Can you imagine to put there more than 16 MPI (tasks), i.e. less than 1GB per core?

Programming models: OpenMP

On the other side, OpenMP is a standard for shared memory systems

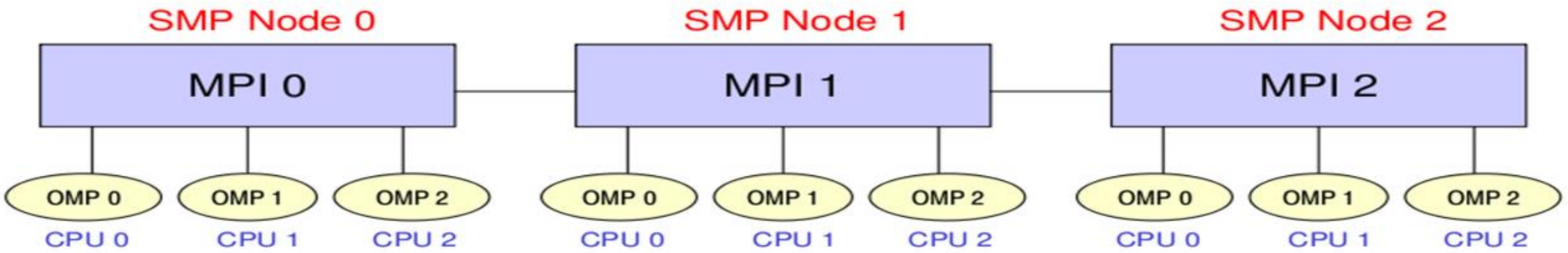
OpenMP is robust, clear and sufficiently easy to implement, but depending on the implementation, typically the scaling on the number of threads is much less effective than the scaling on number of MPI tasks

Putting together MPI with OpenMP could permit to exploit the features of the new architectures, mixing these paradigms

Hybrid model: MPI + OpenMP

- In a single node you can exploit a shared memory parallelism using OpenMP
- Across the nodes you can use MPI to scale up

Example: on the previous machine you can put 1 MPI task on each node and 16 OpenMP threads. If the scalability on threads is good enough, you can use all the node memory.



MPI vs. OpenMP

- **Pure MPI Pro**:
  - High scalability
  - High portability
  - Scalability out-of-node
  - Possible to overlap inter and intranode communications

- **Pure MPI Con**:
  - Hard to develop and debug
  - Explicit communications
  - Coarse granularity
  - Hard to ensure load balancing

- **Pure OpenMP Pro:**
  - Easy to deploy (often)
  - Low latency
  - Implicit communications
  - Coarse and fine granularity
  - Dynamic Load balancing

- **Pure OpenMP Con:**
  - Only on shared memory machines
  - Intranode scalability
  - Possible data placement problem
  - Undefined thread ordering

MPI + OpenMP

- Conceptually simple and elegant
- Suitable for multicore/multinodes architectures
- Two-level hierarchical parallelism
- In principle, you can alleviate problems related to the scalability of MPI, reducing the number of tasks and network flooding
- Using a hybrid approach MPI+OpenMP can lower the number of MPI tasks used by the application.
- Memory footprint can be alleviated by a reduction of replicated data on MPI level
- Speed-up limited due algorithmic issues can be solved (because you're reducing the amount of communication)

Reality is bitter…

In real scenarios, mixing MPI and OpenMP, sometimes, can make your code slower:

- If you exceed with the number of OpenMP threads you can encounter problems with locking of resources
- Sometimes threads can stay in a idle state (spin) for a long time
- Difficulties in the management of variables scope

Hybrid programming

Let’s start!!

The most simple recipe is:

- start from a serial code and make it a MPI-parallel code
- implement for each of the MPI task a OpenMP-based parallelization

Nothing prevents to implement a MPI parallelization inside a OpenMP parallel region

- in this case, you should take care of the thread-safety

To start, we will assume that only the master thread is allowed to communicate with others MPI tasks

A very first simple hybrid code

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     …some computation and MPI communication

     #pragma omp parallel for
     for (i=0; i<n; i++)
     …computation

     …some computation and MPI communication
     MPI_Finalize();
     return 0;
}
```

Not of particular interest: MPI and Openmp don’t interact with each other at all!!

MPI_Init_thread support

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

required desired level of thread support
provided provided level (may be less than required)

Four levels are supported:

- **MPI_THREAD_SINGLE**: Only one thread will run (no MPI in parallel regions)
- **MPI_THREAD_FUNNELED**: processes may be multithreaded, but only the master thread can make MPI calls (MPI calls are delegated to master thread)
- **MPI_THREAD_SERIALIZED**: processes could be multi-threaded More than one thread can make MPI calls, but only one at a time.
- **MPI_THREAD_MULTIPLE**: multiple threads can make MPI calls, with no restrictions.

The various implementations differ in levels of thread-safety

MPI_THREAD_SINGLE

No MPI calls inside parallel regions

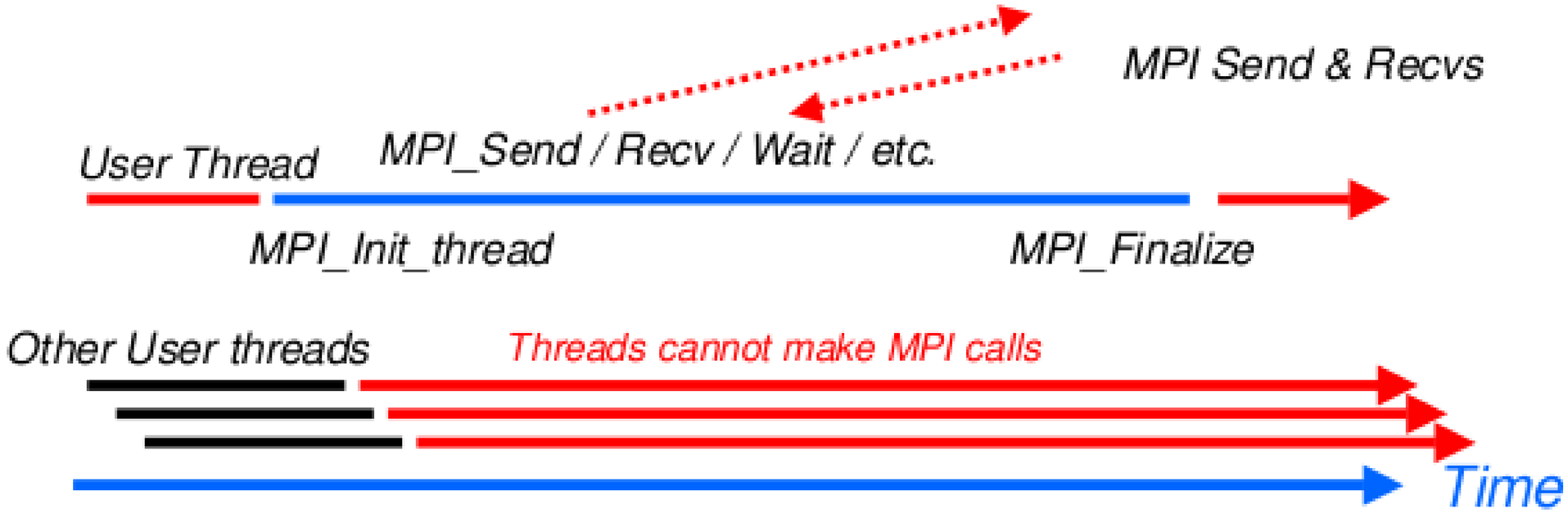
- A particular case when there are no OpenMP parallel regions
- *MPI_Init_thread* with MPI_THREAD_SINGLE is fully equivalent to *MPI_Init*

```
int main(int argc, char ** argv)
{
    int buf[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (i = 0; i < 100; i++)
       compute(buf[i]);
    /* Do MPI stuff */
    MPI_Finalize();
    return 0;
}
```

MPI_THREAD_FUNNELED

All MPI calls are made by the master, but only by the master thread

- It adds the possibility to make MPI calls inside a parallel
- The programmer must guarantee that only the master thread makes the call!



MPI_THREAD_FUNNELED

```
int main(int argc, char ** argv)
{
    int buf[100], provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    #pragma omp parallel for
    for (i = 0; i < 100; i++)
       compute(buf[i]);
    /* Do MPI stuff */
    MPI_Finalize();
    return 0;
}
```

MPI function calls can also be in a parallel region, enclosed in “omp master” clause

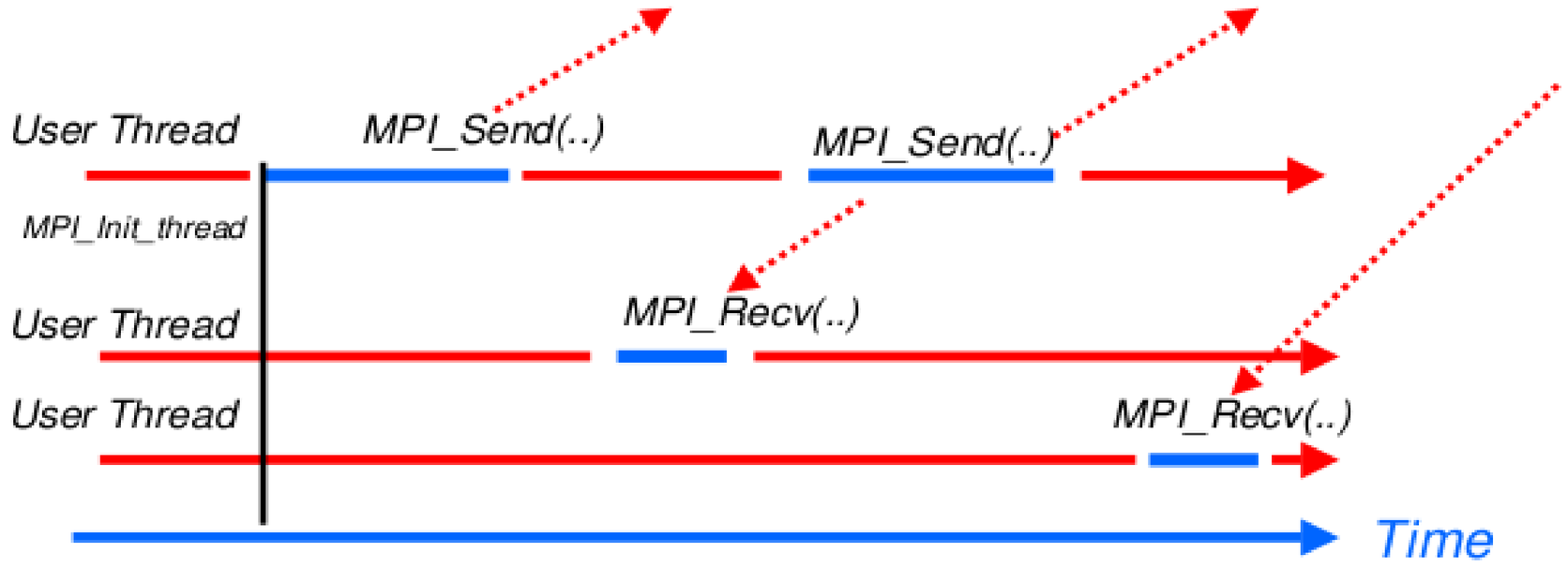
There is no synchronization at the end of a “omp master” region, so a barrier is needed before and after to ensure that data buffers are available before/after the MPI communication

```
#pragma omp barrier
#pragma omp master
     MPI_Xxx(...);
#pragma omp barrier
```

MPI_THREAD_SERIALIZED

Multiple threads may make MPI calls, but only one at a time

- MPI calls are not made concurrently from two distinct threads. MPI calls are "serialized"
- The programmer must guarantee that!



MPI_THREAD_SERIALIZED

```
int main(int argc, char ** argv)
{
    int buf[100], provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
       compute(buf[i]);
       #pragma omp critical
       /* Do MPI stuff */
    }
    MPI_Finalize();
    return 0;
}
```

MPI calls can be inside a parallel region, but enclosed in a “omp single” region or “omp critical” or similar regions.

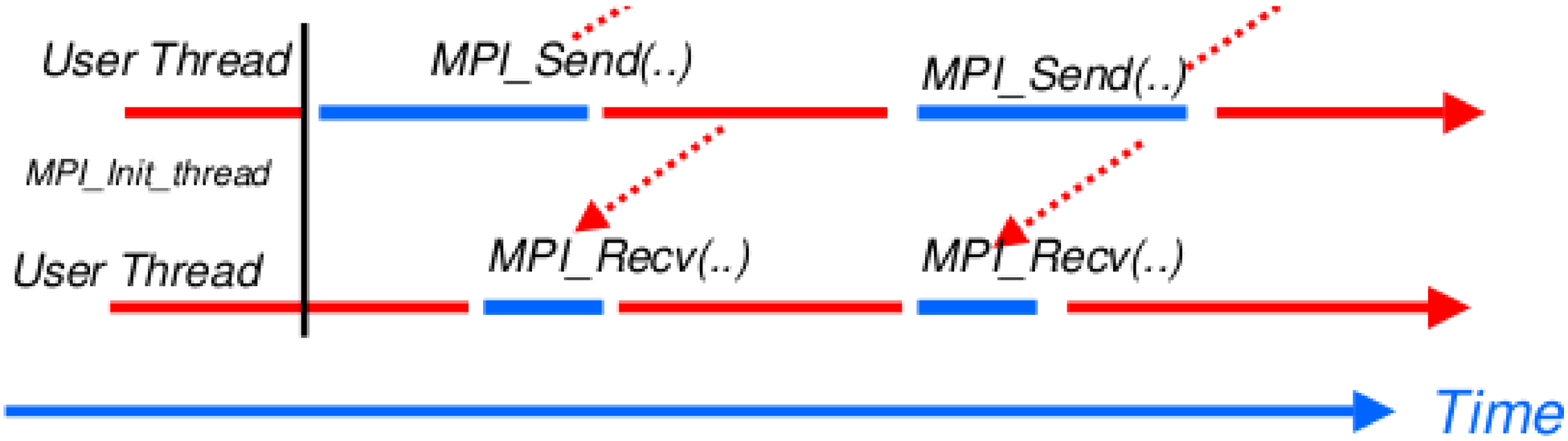
Again, a starting barrier may be needed to ensure data consistency, but at the end of omp single there is an automatic barrier (unless nowait is specified)

```
#pragma omp barrier
#pragma omp single
     MPI_Xxx(...);
```

MPI_THREAD_MULTIPLE

Any thread is allowed to perform MPI communications

It is the most flexible mode, but also the most complicated



MPI_THREAD_MULTIPLE

```
int main(int argc, char ** argv)
{
    int buf[100], provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
       compute(buf[i]);
       /* Do MPI stuff */
    }
    MPI_Finalize();
    return 0;
}
```

Takeaway: comparison to pure MPI

Master-only (MPI_THREAD_SINGLE)

- Simple to write and maintain
- Clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
- Threads other than the master thread are idle during MPI calls
- Inter-process and inter-threads communication do not overlap

Funneled (MPI_THREAD_FUNNELED)

- Relatively simple to write and maintain
- Possible for other threads to compute while master is in an MPI call
- Less clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
- Inter-process and inter-threads communication still do not overlap

Hint: overlap as much as possible communication and computation

Takeaway: comparison to pure MPI

Serialized (MPI_THREAD_SERIALIZED)

- ❑ Easier for other threads to compute while one is in an MPI call
- ❑ Can arrange threads to communicate only their own data (i.e. the data they read and write)
- ❑ Getting harder to write/maintain
- ❑ More, smaller messages are sent (possible additional latency overheads)
- ❑ Need to use tags or communicators to distinguish between messages from or to different threads in the same MPI process

Multiple (MPI_THREAD_MULTIPLE)

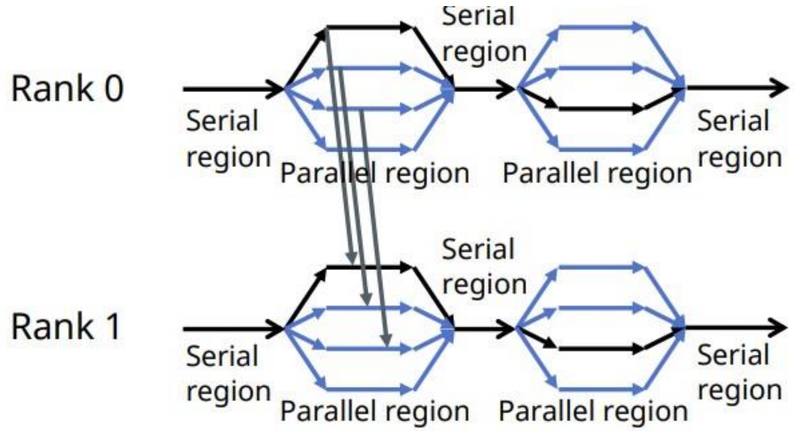
- ❑ Messages from different threads can in theory overlap (many MPI implementations serialize them internally)
- ❑ Natural for threads to communicate only their own data
- ❑ Hard to write/maintain
- ❑ Not all MPI implementation support this – loss of portability!
- ❑ Most MPI implementations don’t perform well with this!

Dealing with MPI_THREAD_MULTIPLE

With the so called multiple mode all threads can make MPI calls **independently**

When multiple thread communicate, the sending and receiving threads normally need to match

- **Use thread specific tags**
- **Use thread specific communicators**




```
#pragma omp parallel private (tid,tidtag)
{
  tid=omp_thread_get_num();
  tidtag=1024 + tid

  MPI_Sendrecv(buf_s,n,MPI_INT,pid,tidtag,buf_r,n,
  MPI_INT,pid,tidtag,MPI_COMM_WORLD,status);
}
```

Dealing with MPI_THREAD_MULTIPLE

Collective operations in the multiple mode:

- MPI standard allows multiple threads to call collectives simultaneously
- Programmer must ensure that the same communicator is not being concurrently used by two different collective communication calls at the same process
- In most cases, even with MPI_THREAD_MULTIPLE, it’s safe to perform the collective communication from a single thread (usually the master thread)

Alternative: generate thread-specific communicators

```
nthr=omp_get_max_threads();
MPI_Comm tcomm[nhtr]
for (thr_id=1,thr_id<nthr,thr_id++) {
    MPI_Comm_split(MPI_COMM_WORLD,thr_id,proc_id,
    &tcomm[thr_id]);

#pragma omp parallel private(tid)
{
  tid=omp_thread_get_num();
  MPI_Bcast(…,tcomm[tid]);
}
```

Implementation notes

When to hybrid program

Applications that can benefit from hybrid approach:

- Codes having limited MPI scalability (use of heavy *MPI_Alltoall* for example).
- Codes requiring dynamic load balancing.
- Codes limited by memory size and having many replicated data between MPI processes or having data structures that depends on the number of processes.
- Inefficient MPI implementation library for intra-node communication.
- Codes working on problems of fine-grained parallelism or on a mixture of fine and coarse-grain parallelism.
- Codes limited by the scalability of their algorithms.

MPI Implementations and thread support

Most MPI implementations support now a good default thread support

- Standards are scarcely equipped with minimal thread support (MPI_THREAD_SINGLE)
- And probably MPI_THREAD_FUNNELED (even if not explicitly stated)
- Which (usually) is ok for cases where MPI communications are called outside of OMP parallel regions
- Implementations with MPI_THREAD_MULTIPLE thread support can be more complicated, error-prone and sometimes slower

Checking the MPI_Init_thread provided support is a must for the developer

```
required = MPI_THREAD_MULTIPLE;
MPI_Init_thread(&argc, &argv, required, &provided);
if(required != provided) {
    if(rank == 0) {
       fprintf(stderr,"incompatible MPI thread support\n");
       fprintf(stderr,"required,provided: %d %d\n", required, provided);
       }
    ierr = MPI_Finalize();
    exit(-1);
}
```

OpenMPI (1.8.3): there is a configure option to specify:
--enable-mpi-thread-multiple
[Enable MPI_THREAD_MULTIPLE support (**default: disabled**)]

Resource allocation

The problem: how to distribute and control the resource allocation and usage (MPI processes/OMP threads) within a resource manager

- How to use all the allocated resources: if n cores per node have been allocated, how to run my program using all of that cores
  - Not less, maximize performance
  - Not more, do not interact with jobs of other users
- How to use at its best the resource
  - optimal mapping between MPI processes/OMP threads and physical cores

Most MPI implementations allow a tight integration with resource managers in order to ease the usage of the requested resources. The integration may enforce some constraints or just give hints to the programmer

OpenMP thread assignment

OpenMP v4.5 provides **OMP_PLACES** and **OMP_PROC_BIND** (it's standard!)

- **OMP_PLACES**: Specifies on which “place” the threads should be placed. The thread placement can be either specified using an abstract name or by an explicit list of the places. Allowed abstract names: threads, cores and sockets
- **OMP_PROC_BIND**: specifies whether threads may be distributed between physical CPUs. If set to TRUE, OpenMP threads should not be distributed; if set to FALSE they may be. Use a comma separated list with the values MASTER, CLOSE and SPREAD to specify the thread affinity policy for the corresponding nesting level.

Thread assignment can be crucial for performances!!

OpenMP thread assignment

Example (on a machine with 2 10-core sockets):

export OMP_PLACES=cores;

• export OMP_PROC_BIND= master	• export OMP_PROC_BIND= close	• export OMP_PROC_BIND= spread
• t0 -> c0	• t0 -> c0	• t0 -> c0
• t1 -> c0	• t1 -> c1	• t1 -> c10
• t2 -> c0	• t2 -> c2	• t2 -> c1
• t3 -> c0	• t3 -> c3	• t3 -> c11

Current MPI Standard to be found at: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
Credits to P. Lanucara and many other colleagues from CINECA for the slides