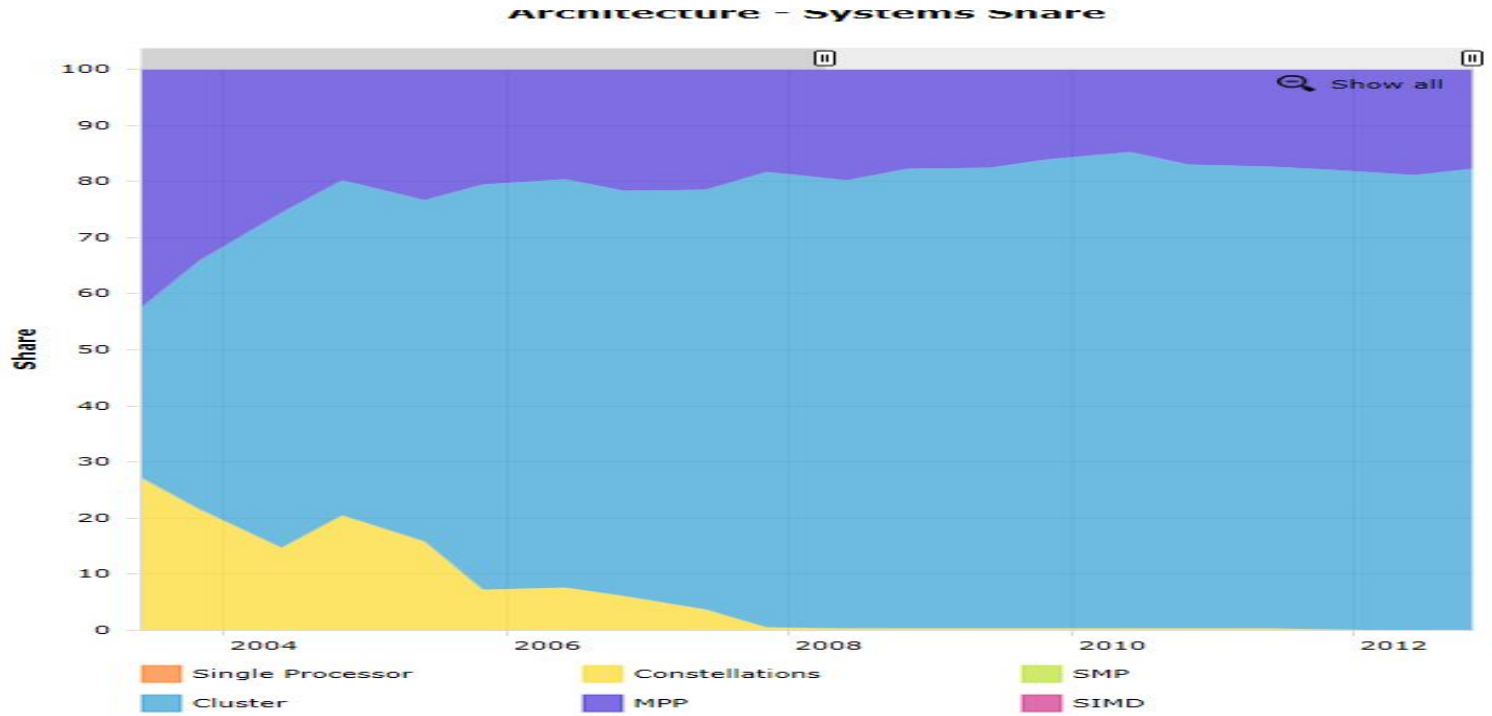
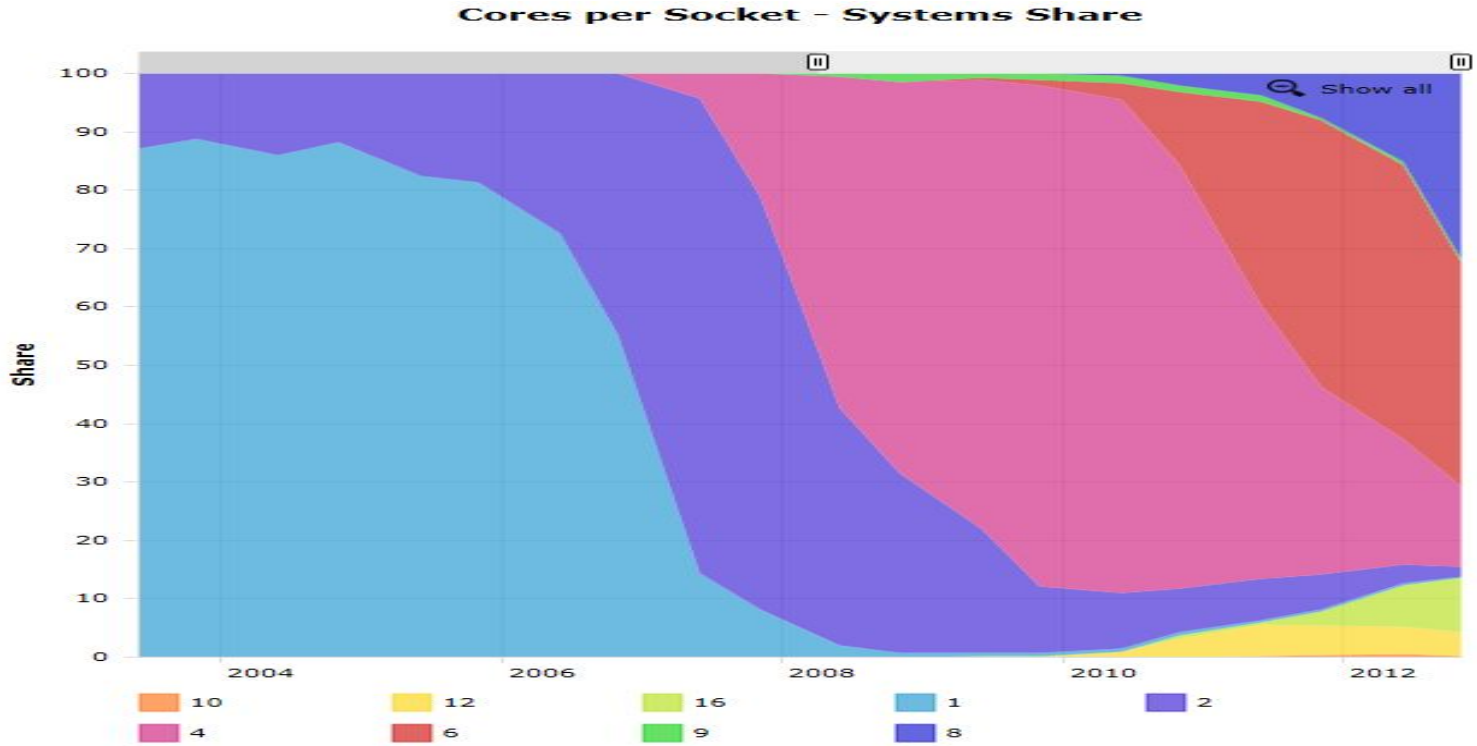
Introduction to MPI+OpenMP hybrid programming

Fabio Affinito– f.affinito@cineca.it
SuperComputing Applications and Innovation Department

Architectural trend



Architectural trend



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

- Distributed parallel computers rely on MPI
  - strong
  - consolidated
  - standard
  - enforce 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: Bluegene/Q has 16GB per node and 16 cores. Can you imagine to put there more than 16MPI (tasks), i.e. less than 1GB per core?

Programming models

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

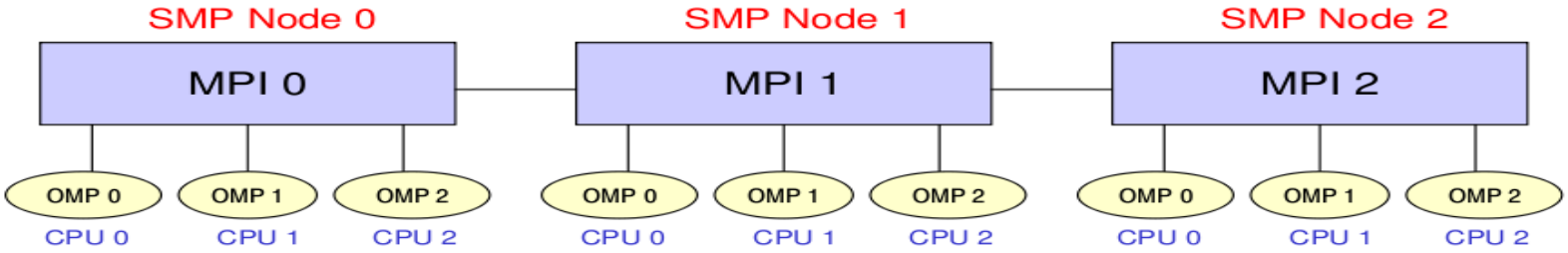
- OpenMP is robust, clear and sufficiently easy to implement but
  - depending on the implementation, typically the scaling on the number of threads is much less effective than the scaling on number of MPI tasks

- 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 a Bluegene/Q 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
  - No false sharing
  - Scalability out-of-node
- **Pure MPI Con:**
  - Hard to develop and debug.
  - Explicit communications
  - Coarse granularity
  - Hard to ensure load balancing

MPI vs OpenMP

- **Pure MPI Pro:**
  - High scalability
  - High portability
  - No false sharing
  - Scalability out-of-node
- **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

Increasing granularity

- OpenMP introduces fine granularity parallelism
- Loop-based parallelism
- Task construct (OpenMP 3.0): powerful and flexible
- Load balancing can be dynamic or scheduled
- All the work is in charge to the compiler
- No explicit data movement

Two level parallelism

- Using a hybrid approach means to balance the hierarchy between MPI tasks and thread.
- MPI in most cases (but not always) occupy the upper level respect to OpenMP
  - usually you assign n threads per MPI task, not m MPI tasks per thread
- The choice about the number of threads per MPI task strongly depends on the kind of application, algorithm or kernel. (this number can change inside the application)
- There's no a golden rule. More often this decision is taken a-posteriori after benchmarks on a given machine/architecture

Saving MPI tasks

- 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

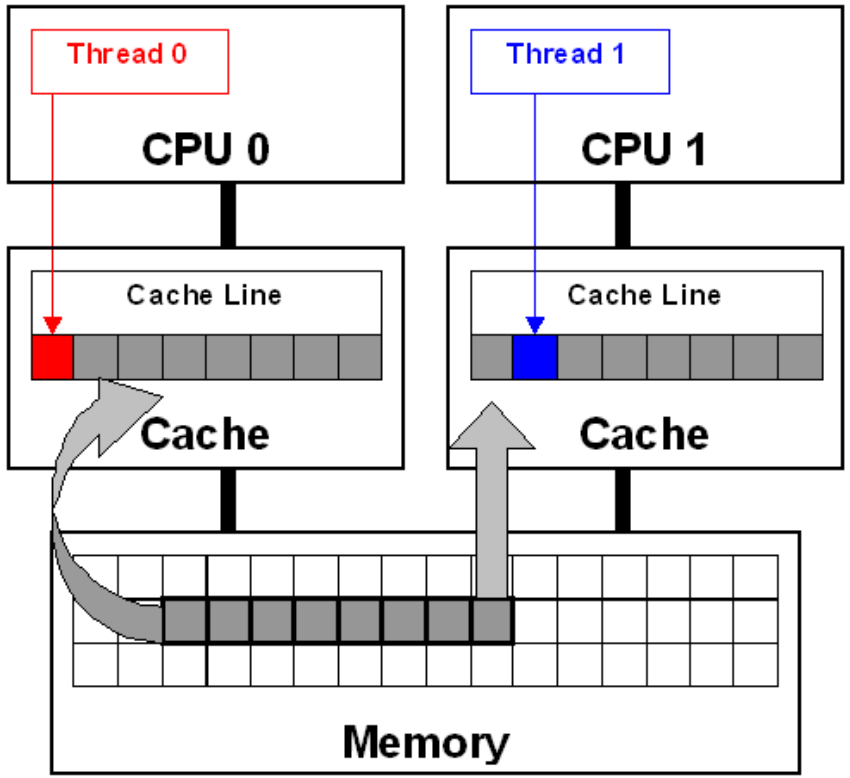
- In real practise, mixing MPI and OpenMP, sometimes, can make your code slower
  - If you exceed with the number of OpenMP threads you can encounter problems with locking of resources
  - Sometimes threads can stay in a idle state (spin) for a long time
  - Problems with cache coherency and false sharing
  - Difficulties in the management of variables scope

Cache coherency and false sharing

- It is a side effects of the cache-line granularity of cache coherence implemented in shared memory systems.
- The cache coherency implementation keep track of the status of cache lines by appending *state bits to* indicate whether data on cache line is still valid or outdated.
- Once the cache line is modified, cache coherence notifies other caches holding a copy of the same line that its line is invalid.
- If data from that line is needed, a new updated copy must to be fetched.

False sharing

```
#pragma omp parallel for
shared(a) schedule(static,1)
for (int i=0; i<n; i++)
      a[i] = i;
```



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 simple hybrid code

```
call MPI_INIT (ierr)
call MPI_COMM_RANK (…)
call MPI_COMM_SIZE (…)
 …  some computation and MPI communication
 call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL
!$OMP DO
   do i=1,n
        … computation
   enddo
 !$OMP END  DO
 !$OMP END  PARALLEL
 …  some computation and MPI communication
call MPI_FINALIZE (ierr)
```

Master-only approach

Advantages:

- Simplest hybrid parallelization (easy to understand and to manage)
- No message passing inside a SMP node

Disadvantages:

- All other threads are sleeping during MPI communications
- Thread-safe MPI is required

MPI_Init_thread support

- **MPI_INIT_THREAD** (required, provided, ierr)
  - IN: required, desired level of thread support (integer).
  - OUT: provided, provided level (integer).
    provided may be less than required.

- Four levels are supported:
  - **MPI_THREAD_SINGLE**: Only one thread will runs. Equals to MPI_INIT.
  - **MPI_THREAD_FUNNELED**: processes may be multithreaded, but only the main thread can make MPI calls (MPI calls are delegated to main thread)
  - **MPI_THREAD_SERIALIZED**: processes could be multithreaded. 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.

MPI_Init_thread

- The various implementations differs in levels of thread-safety
- If your application allow multiple threads to make MPI calls simultaneously, whitout MPI_THREAD_MULTIPLE, is not thread-safe
- Using OpenMPI, you have to use –enable-mpi-threads at configure time to activate all levels.
- Higher level corresponds higher thread-safety. Use the required safety needs.

MPI_THREAD_SINGLE

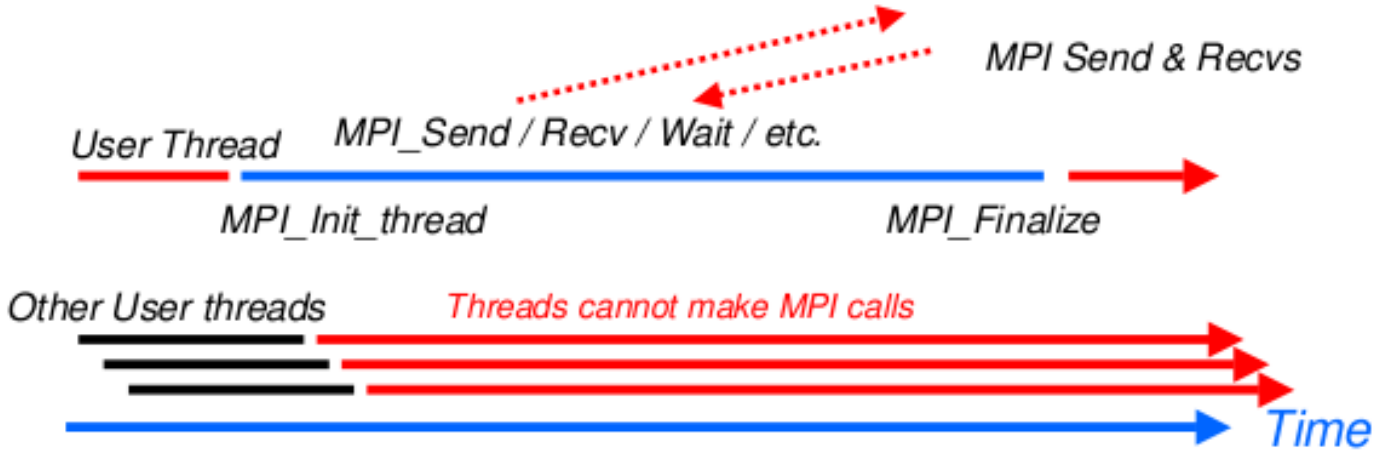
- It is fully equivalent to the master-only approach

```
!$OMP PARALLEL DO
   do i=1,10000
      a(i)=b(i)+f*d(i)
  enddo
!$OMP END PARALLEL DO
   call MPI_Xxx(...)
!$OMP PARALLEL DO
   do i=1,10000
      x(i)=a(i)+f*b(i)
   enddo
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for
      for (i=0; i<10000; i++)
      { a[i]=b[i]+f*d[i];
      }
/* end omp parallel for */
      MPI_Xxx(...);
#pragma omp parallel for
      for (i=0; i<10000; i++)
      { x[i]=a[i]+f*b[i];
      }
/* end omp parallel for */
```

MPI_THREAD_FUNNELED

- It adds the possibility to make MPI calls inside a parallel region, but only the master thread is allowed to do so



MPI_THREAD_FUNNELED

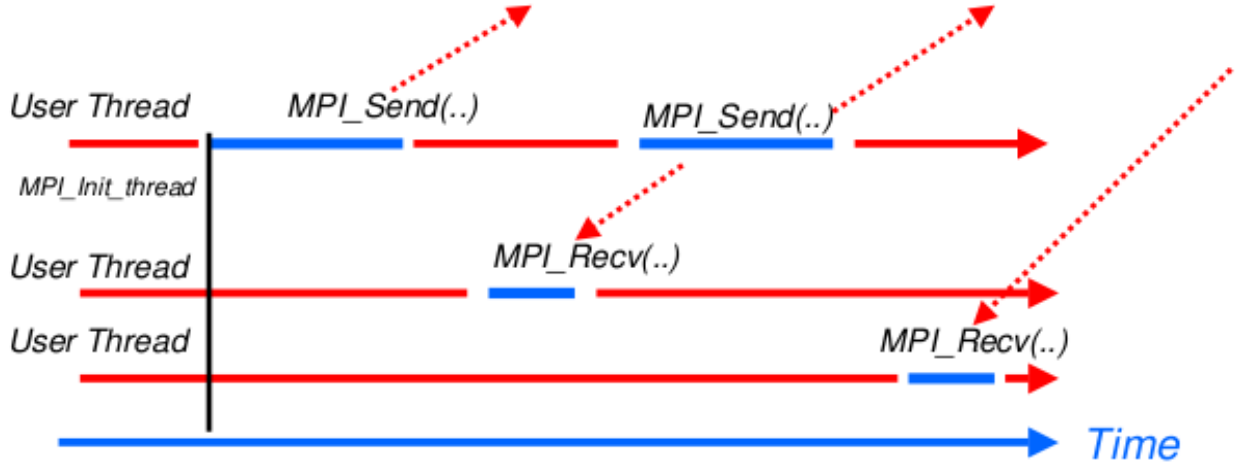
- MPI function calls can be: outside a parallel region or in a parallel region, enclosed in “omp master” clause
- There's no synchronization at the end of a “omp master” region, so a barrier is needed before and after to ensure that data buffers are available before/after the MPI communication

```
!$OMP BARRIER
!$OMP MASTER
      call MPI_Xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

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

MPI_THREAD_SERIALIZED

- MPI calls are mad concurrently by two or more different threads. All the MPI communications are serialized.



MPI_THREAD_SERIALIZED

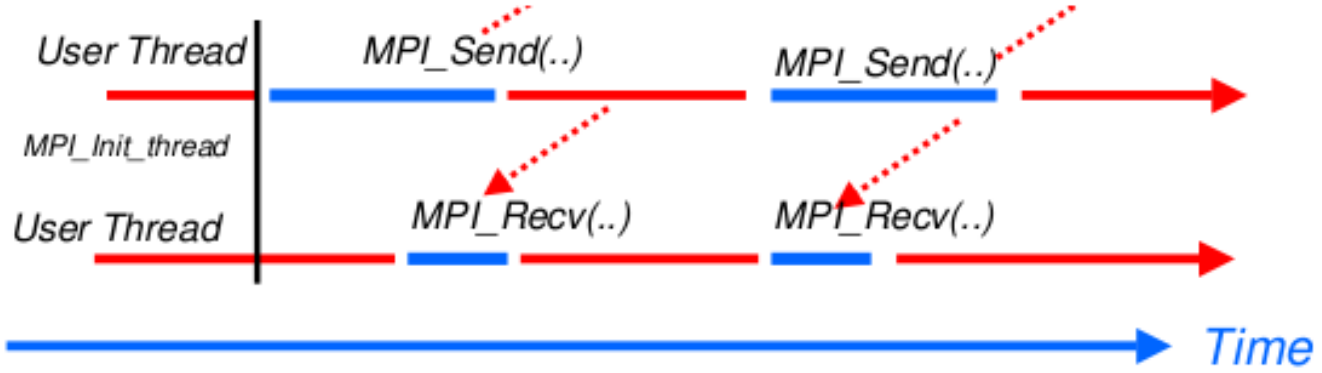
- MPI calls can be outside parallel regions, or inside, but enclosed in a “omp single” region (it enforces the serialization)
- Again, a barrier should ensure data consistency

```
!$OMP BARRIER
!$OMP SINGLE
      call MPI_Xxx(...)
!$OMP END SINGLE
```

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

MPI_THREAD_MULTIPLE

- It is the most flexible mode, but also the most complicate one
- Any thread is allowed to perform MPI communications, without any restrictions.



Comparison to pure MPI

Funneled/serialized

- All threads but the master are sleeping during MPI communications
- Only one threads may not be able to lead up to max inter-node bandwith

Pure MPI

- Each CPU can lead up max inter-node bandwidth

Hints: overlap as much as possible communications and computations

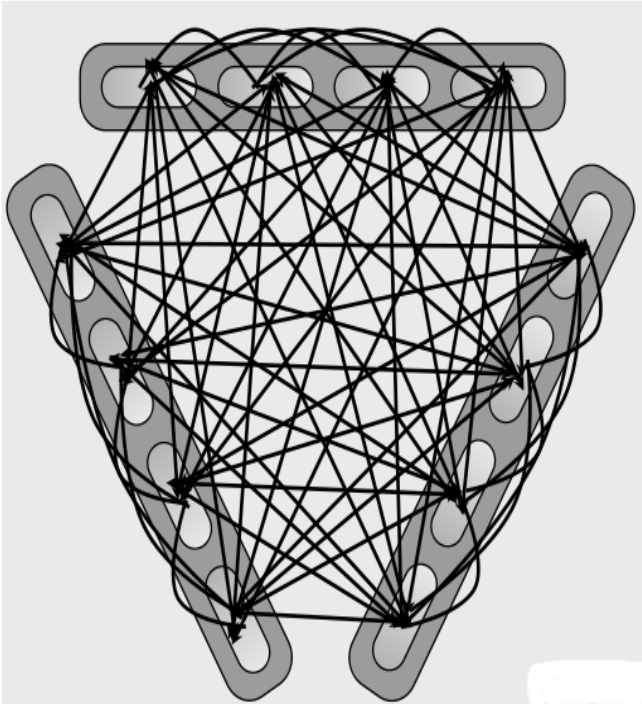
Overlap communications and computations

- In order to overlap communications with computations, you require at least the MPI_THREAD_FUNNELED mode
- While the master thread is exchanging data, the other threads performs computation
- It is difficult to separate code that can run before or after the data exchanged are available

```
!$OMP PARALLEL
    if (thread_id==0) then
        call MPI_xxx(…)
    else
        do some computation
    endif
!$OMP END PARALLEL
```

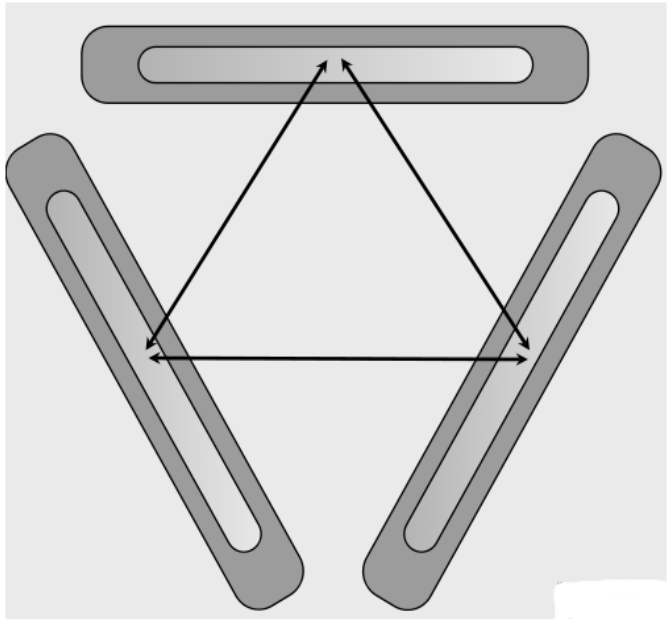
MPI collective hybridization

- MPI collectives are highly optimized
- Several point-to-point communication in one operations
- They can hide from the programmer a huge volume of transfer (MPI_Alltoall generates almost 1 million point-to-point messages using 1024 cores)
- There is no non-blocking (no longer the case in MPI 3.0)



MPI collective hybridization

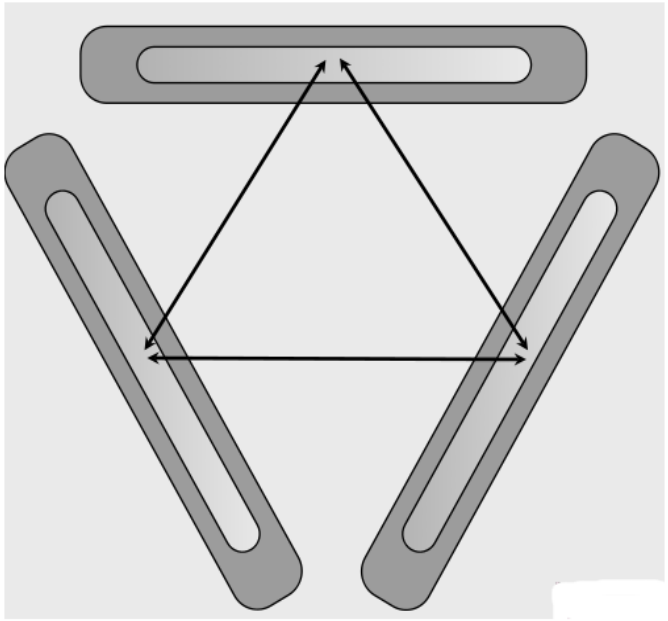
- Better scalability by a reduction of both the number of MPI messages and the number of process. Tipically:
- for all-to-all communications, the number of transfers decrease by a factor #threads^2
- the length of messages increases by a factor #threads
- Allow to overlap communication and computation.



MPI collective hybridization

Restrictions:

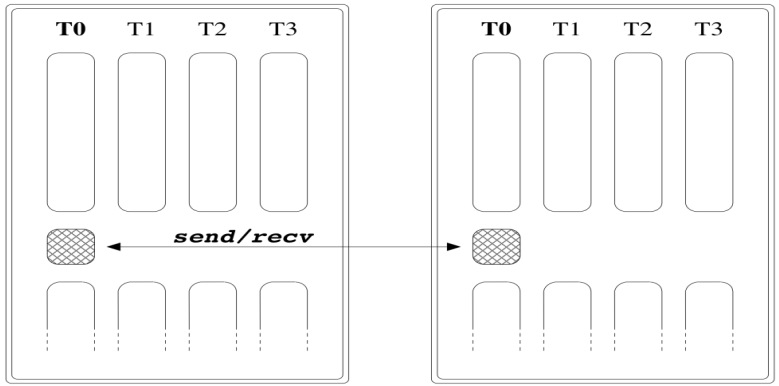
- In MPI_THREAD_MULTIPLE mode is forbidden at any given time two threads each do a collective call on the same communicator (MPI_COMM_WORLD)
- 2 threads calling each a MPI_Allreduce may produce wrong results
- **Use different communicators for each collective call**
- **Do collective calls only on 1 thread per process (MPI_THREAD_SERIALIZED mode should be fine)**



Multithreaded libraries

- Introduction of OpenMP into existing MPI codes includes OpenMP drawbacks (synchronization, overhead, quality of compiler and runtime…)
- A good choice (whenever possible) is to include into the MPI code a **multithreaded, optimized library suitable for the application**.
- **BLAS, LAPACK, MKL (Intel), FFTW** are well known multithreaded libraries available in the HPC ecosystem.
- MPI_THREAD_FUNNELED (almost) must be supported.

Multithreaded FFT (QE)



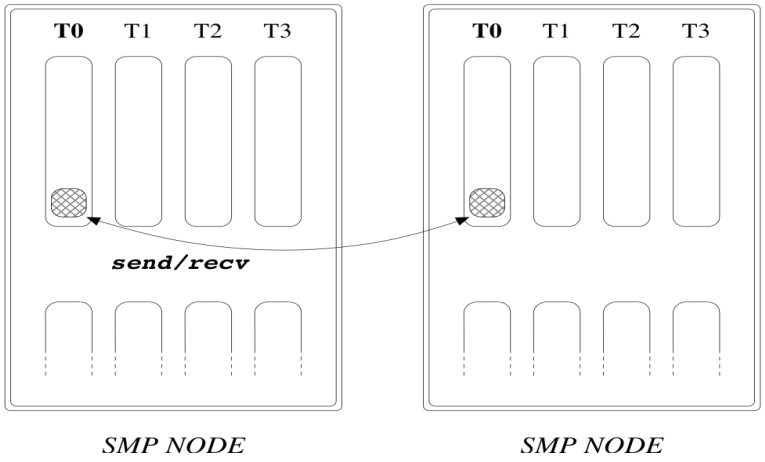
Only the master thread can do MPI communications (Pseudo QE code)

```
# begin OpenMP region
   do i = 1, nsl    in parallel
      call 1D-FFT along z ( f[offset] )
   end do
# end OpenMP region

   call fw_scatter( ... )

# begin OpenMP region
   do i = 1, nzl    in parallel
      do j = 1, Nx
            if ( dofft[j] ) then
               call 1D-FFT along y  ( f[offset] )
      end do
      call 1D-FFT along x ( f[offset] )  Ny-times
   end do
# end OpenMP region
```

Multithreaded FFT (QE)



Funneled: master thread do MPI communications within parallel region (Pseudo QE code)

```
# begin OpenMP region
   do i = 1, nsl    in parallel
      call 1D-FFT along z ( f[offset] )
   end do

# begin of OpenMP MASTER section
   call fw_scatter( ... )
# end of OpenMP MASTER section
# force synchronization with OpenMP barrier

   do i = 1, nzl    in parallel
      do j = 1, Nx
            if ( dofft[j] ) then
               call 1D-FFT along y ( f[offset] )
      end do
      call 1D-FFT along x ( f[offset] )  Ny-times
   end do
# end OpenMP region
```

Conclusions

Applications that can benefit from hybrid approach:

- Codes having limited MPI scalability (through the use of 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.

Conclusions

- Hybrid programming is complex and requires high level of expertise.
- Both MPI and OpenMP performances are needed (Amdhal's law apply separately to the two approaches).
- Savings in performances are not guaranteed (extra additional costs).