9th Advanced School on PARALLEL COMPUTING

Introduction to Hybrid MPI+OpenMP programming paradigm

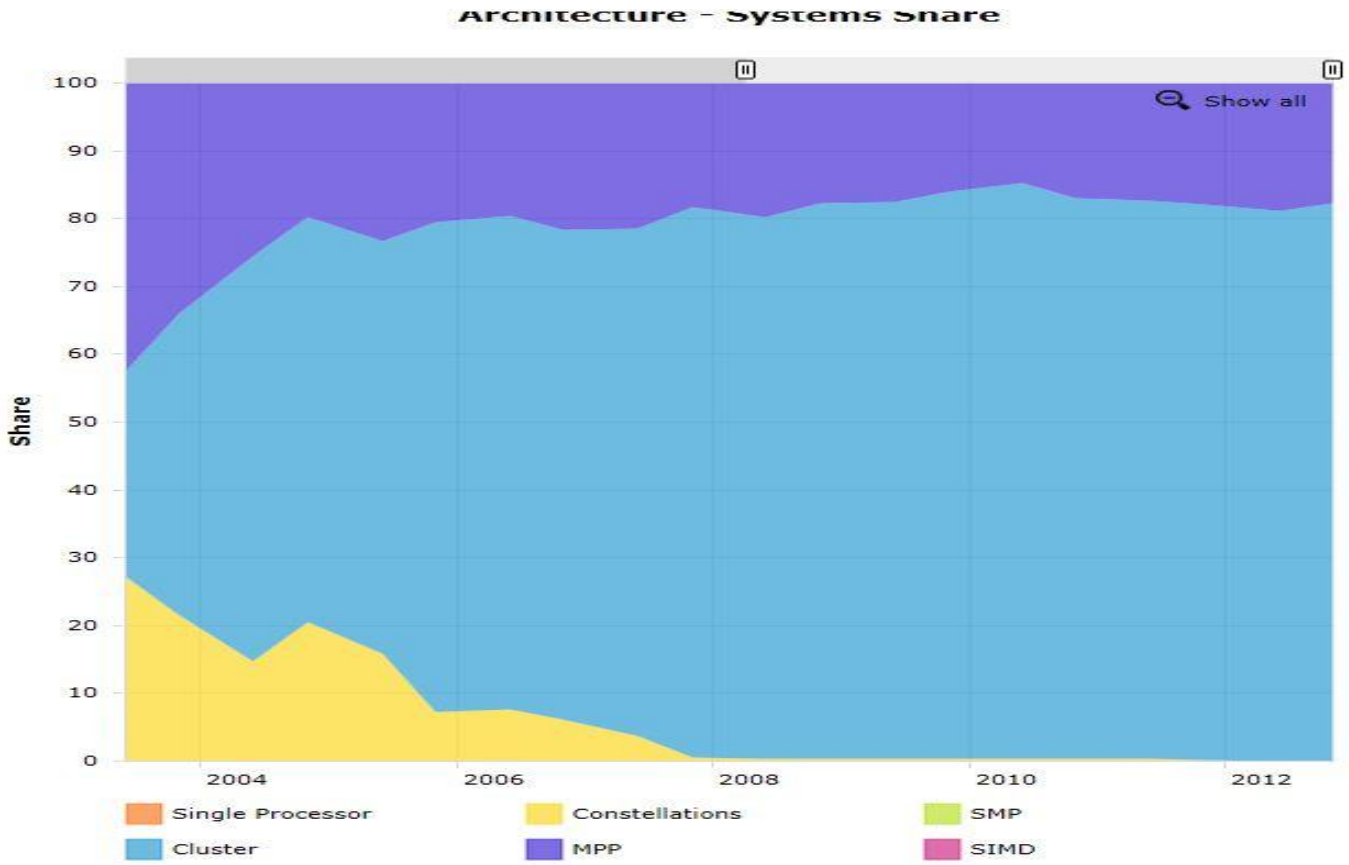
Piero Lanucara – p.lanucara@cineca.it
SuperComputing Applications and Innovation Department

CINECA

February 11 - 15, 2013

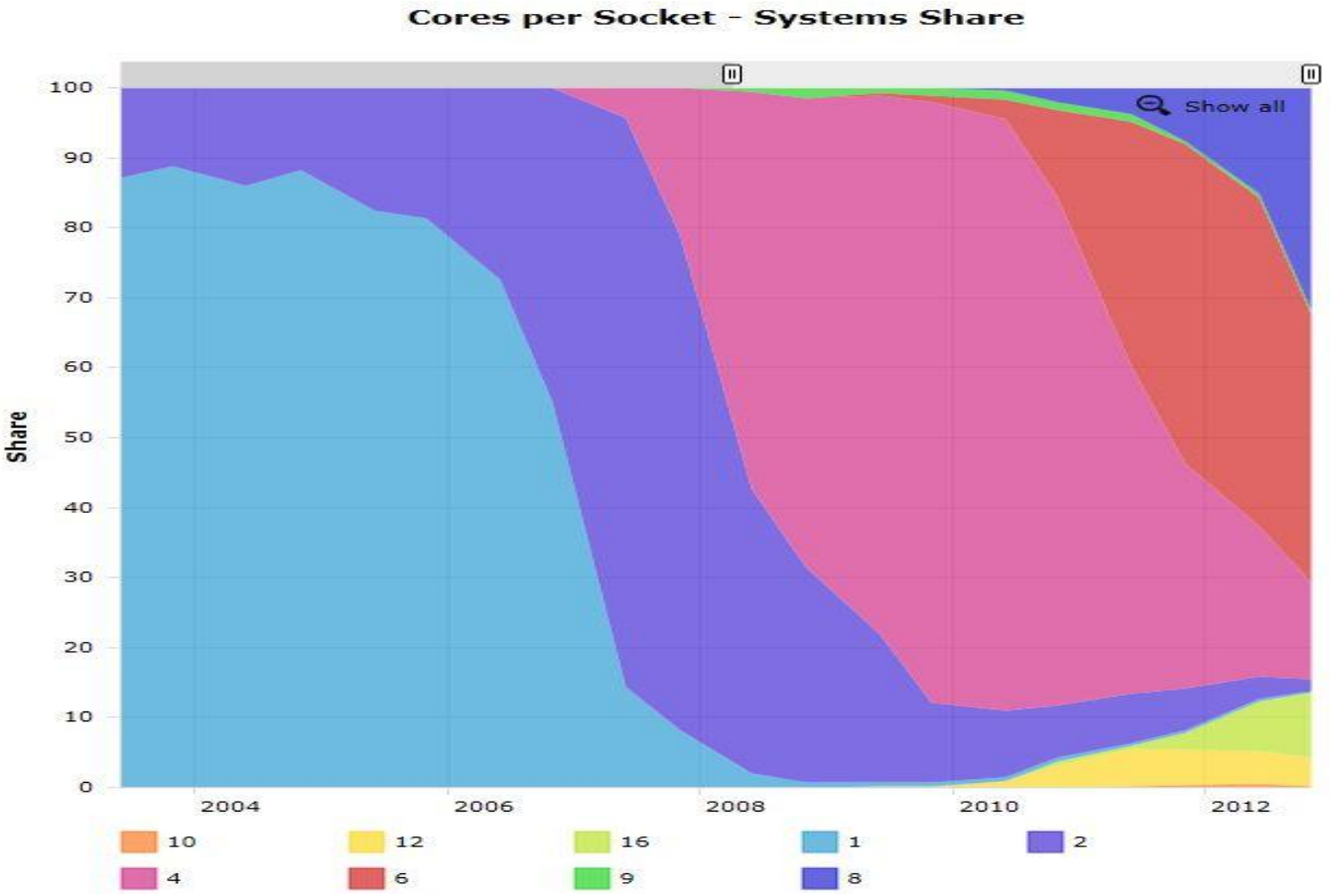
Architecture Trend

- Top 500 historical view: clusters (and MPP) dominates HPC arena



Architecture Trend (cont.)

- Top 500 historical view: the multicore age



Architecture Trend (cont.)

- Multi-socket nodes with rapidly increasing core counts.
- Memory per core decreases.
- Memory bandwidth per core decreases.
- Network bandwidth per core decreases.
- Deeper memory hierarchy.

Which programming model is the best choice for this architecture trend ?

Programming model

Which programming model is the best choice for this architecture?

- MPI is the de-facto standard for distributed memory architectures
- in principle, MPI library is supposed to scale up to 10k cores and over….
- …but the MPI model (*flat*) is not guaranteed to match with this architecture for any kind of application!

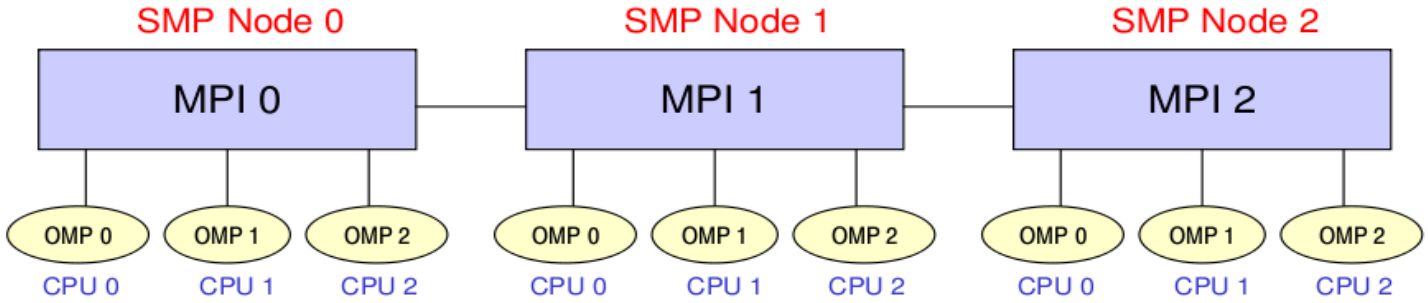
Programming model

Which programming model is the best choice for this architecture?

- OpenMP is the de-facto standard for shared-memory architectures (SMP and Multi-socket nodes).
- OpenMP standard is robust, clear and sufficently easy to implement but is supposed not to scale up to hundreds of cores!
- What about MPI+OpenMP?

The hybrid model

- Logical view: multi-node SMP (Symmetric Multiprocessor).
- MPI between the nodes via node interconnect
- OpenMP (the standard for shared memory parallel programming) inside of the SMP nodes



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

Why hybrid?

- MPI+OpenMP hybrid paradigm is the trend for clusters with SMP architecture.
- Elegant in concept: use OpenMP within the node and MPI between nodes, in order to have a good use of shared resources.
- **Avoid additional communication within the MPI node.**
- OpenMP introduces fine-granularity.
- Two-level parallelism
- Some problems can be reduced by lowering MPI procs number
- If the problem is suitable, the hybrid approach can have better performance than pure MPI or OpenMP codes.

Avoid additional communication within the MPI node

In the pure MPI model each process needs to allocate some extra memory to manage communications and MPI environment
Supposing to use threads within node :

- Smaller number of MPI processes
- Fewer messages, larger message size

Example: one node having 8 cores and 32 GB. Pure MPI and Hybrid:

Pure MPI: 8 MPI process, 4 GB for each (parallelism is 8)
Pure MPI: 1 MPI process, 32 GB (serial)
Hybrid: 1 MPI process, 8 threads. 32 GB **shared** per process, 4 GB per thread. (parallelism is 8)

Why hybrid?

- MPI+OpenMP hybrid paradigm is the trend for clusters with SMP architecture.
- Elegant in concept: use OpenMP within the node and MPI between nodes, in order to have a good use of shared resources.
- Avoid additional communication within the MPI node.
- **OpenMP introduces fine-granularity**.
- Two-level parallelism
- Some problems can be reduced by lowering MPI procs number
- If the problem is suitable, the hybrid approach can have better performance than pure MPI or OpenMP codes.

OpenMP introduces fine-granularity

- Loop-based parallelism (just a set of directives in your code)
- Task construct (OpenMP 3.0): powerful and flexible
- Dynamic and guided scheduling (load balancing)
- Without additional software effort
- Without explicit data movement (MPI's drawback)

Why hybrid?

- MPI+OpenMP hybrid paradigm is the trend for clusters with SMP architecture.
- Elegant in concept: use OpenMP within the node and MPI between nodes, in order to have a good use of shared resources.
- Avoid additional communication within the MPI node.
- OpenMP introduces fine-granularity.
- **Two-level parallelism**
- Some problems can be reduced by lowering MPI procs number
- If the problem is suitable, the hybrid approach can have better performance than pure MPI or OpenMP codes.

Two level parallelism

- Parallelism across SMP nodes, single node equipped with m sockets and n cores per socket.
- To be assigned: the number of MPI process and the (optimal) number of threads per MPI process.
- Heuristics:
  - (often) n threads per MPI process
  - (sometimes) n/2 threads per MPI process
  - (seldom) 2n threads per MPI process
- No golden rule, application and hardware dependent

Why hybrid?

- MPI+OpenMP hybrid paradigm is the trend for clusters with SMP architecture.
- Elegant in concept: use OpenMP within the node and MPI between nodes, in order to have a good use of shared resources.
- Avoid additional communication within the MPI node.
- OpenMP introduces fine-granularity.
- Two-level parallelism
- **Some problems can be reduced by lowering MPI procs number**
- If the problem is suitable, the hybrid approach can have better performance than pure MPI or OpenMP codes.

Some problems can be reduced by lowering MPI procs number

- Memory consumption can be alleviated by a reduction of replicated data on MPI level
- Speedup limited due to algorithmic problem can be solved
- MPI scaling problems (expecially to high number of cores) can be significantly reduced
- MPI scaling problems can be solved by a reduced aggregated message size (compared to pure MPI)

Why mixing MPI and OpenMP code can be slower?

- OpenMP has lower scalability because of locking resources while MPI has not potential scalability limits.
- All threads are idle except ones during an MPI communication
  - Need overlap computation and communication to improve performance
  - Critical section for shared variables update
- Overhead of thread creation
- **Cache coherency and false sharing**.
- Pure OpenMP code is generally slower than pure MPI code
- Fewer optimizations by OpenMP compilers compared to MPI

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.

Cache coherency and False sharing

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

Suppose that each cache line consist of 4 elements and you are using 4 threads

Each thread store:

Thread ID	Stores
0	a[0]
1	a[1]
2	a[2]
3	a[3]
0	a[4]
...	...

Assuming that a[0] is the beginning of the cache line, we have 4 false sharing
The same for a[4]...,a[7]

Cache coherence and False sharing

- The problem is that *state bits* do not keep track of which part of the line is outdated, but indicates the **whole** line
- As a result, when two threads update different data elements in the same cache line, they interfer with each other
- Solving:
  - Using **private data** instead of shared data
  - **Padding**

Hybrid parallelization Roadmap

- From serial code decompose with MPI first and then add OpenMP
- From OpenMP code treat as serial and decompose with MPI
- From MPI code add OpenMP
- Simplest and least error-prone way is to use MPI outside parallel regions, and allow only master thread to communicate between MPI tasks (Hybrid Masteronly)
- Then, try to use MPI inside parallel regions with a tread-safe MPI

Pseudo hybrid code (Masteronly)

```
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)
```

Hybrid Masteronly

- The various MPI implementations differs in levels of thread-safety
- Advantages of Masteronly:
  - No message passing inside of SMP nodes
  - Simplest hybrid parallelization (easy to implement, debug, …)
- Major problems:
  - **All other threads are sleeping while master thread communicates**
  - Use of internode bandwidth satisfactory?
  - Thread-safe MPI is required

MPI_INIT_Thread support (MPI-2)

- **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 support (MPI-2)

- 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

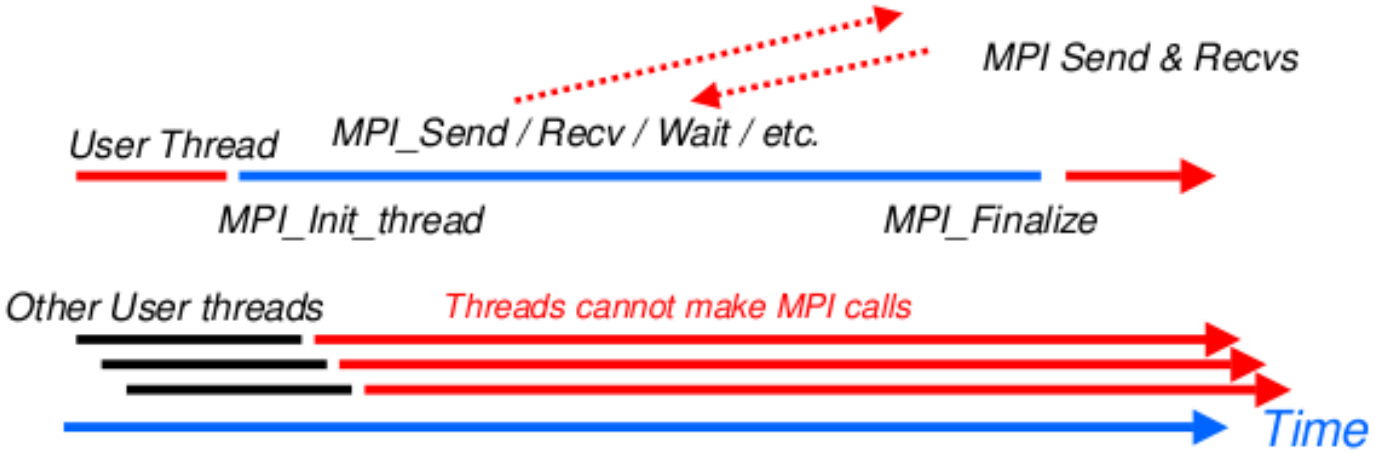
- Equivalent to Hybrid Masteronly:

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

- Only the master thread can do MPI communications.



MPI_THREAD_FUNNELED

- MPI calls:
  - outside the parallel region.
  - inside the parallel region with “omp master”.

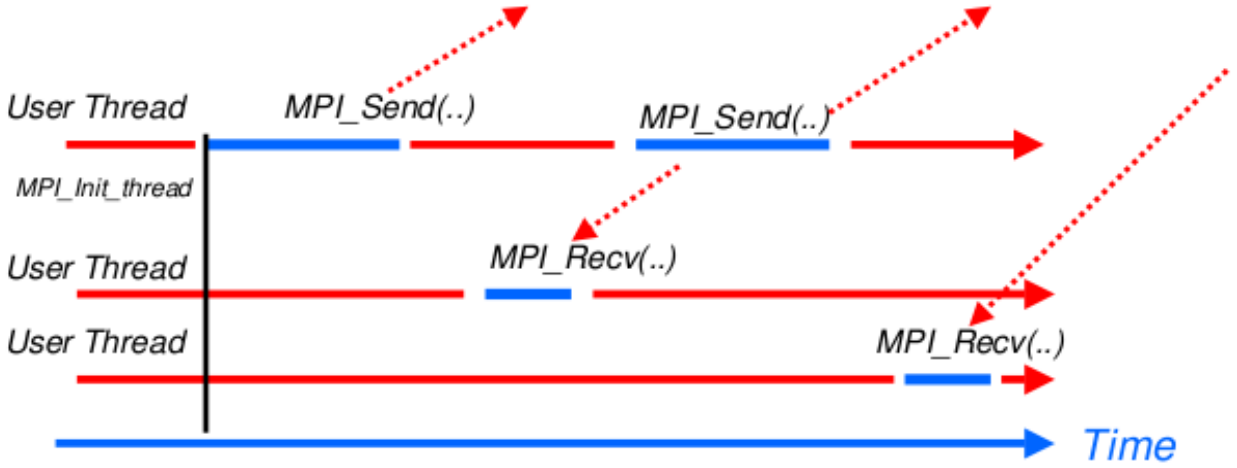
```
!$OMP BARRIER
!$OMP MASTER
      call MPI_Xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

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

There are no synchronizations with “omp master”, thus needs a barrier before and after, to ensure that data and buffers are available before and/or after MPI calls

MPI_THREAD_SERIALIZED

- MPI calls are made "concurrently" by two (or more) different threads (all MPI calls are serialized)



MPI_THREAD_SERIALIZED

- MPI calls:
- Outside the parallel region
- Inside the parallel region with "omp single"

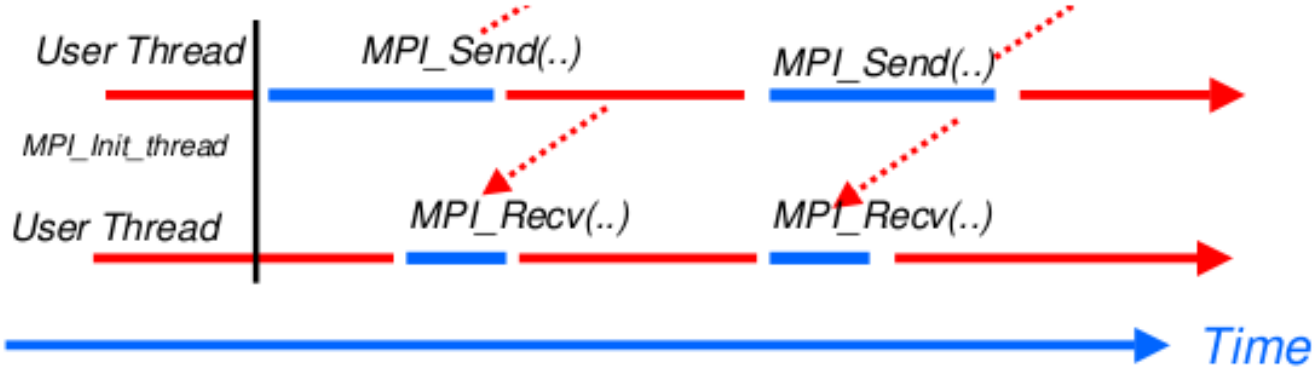
```
!$OMP BARRIER
!$OMP SINGLE
      call MPI_Xxx(...)
!$OMP END SINGLE
```

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

OMP_BARRIER is needed since OMP_SINGLE only guarantees synchronization at the end

MPI_THREAD_MULTIPLE

- Each thread can make communications at any times. Less restrictive and very flexible, but the application becomes very hard to manage



THREAD FUNNELED/SERIALIZED *vs.* Pure MPI

- FUNNELED/SERIALIZED:
  - All other threads are sleeping while just one thread is communicating.
  - Only one thread may not be able to lead up max internode bandwidth
- Pure MPI:
  - Each CPU communication can lead up max internode bandwidth
- **Hints: Overlap communications and computations**.

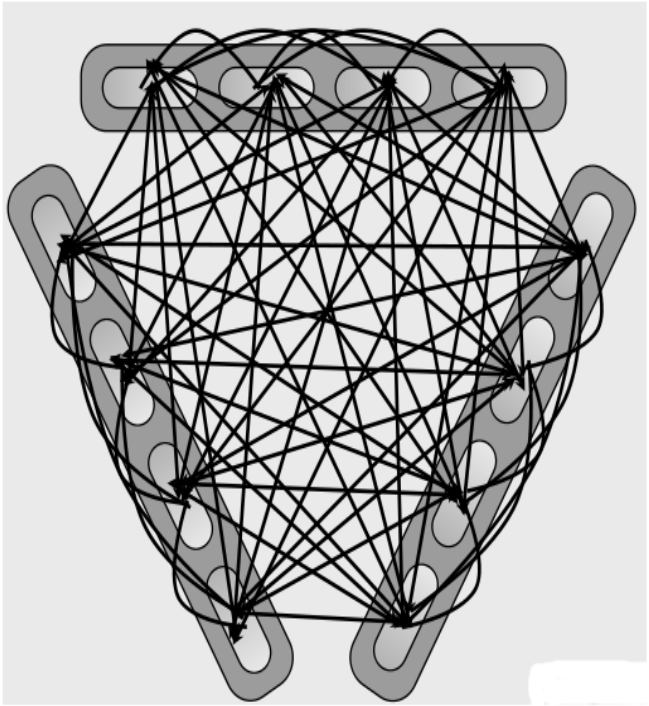
Overlap communications and computation

- Need at least MPI_THREAD_FUNNELED.
- While the master or the single thread is making MPI calls, other threads are doing computations.
- It's difficult to separate code that can run before or after the exchanged data are available

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

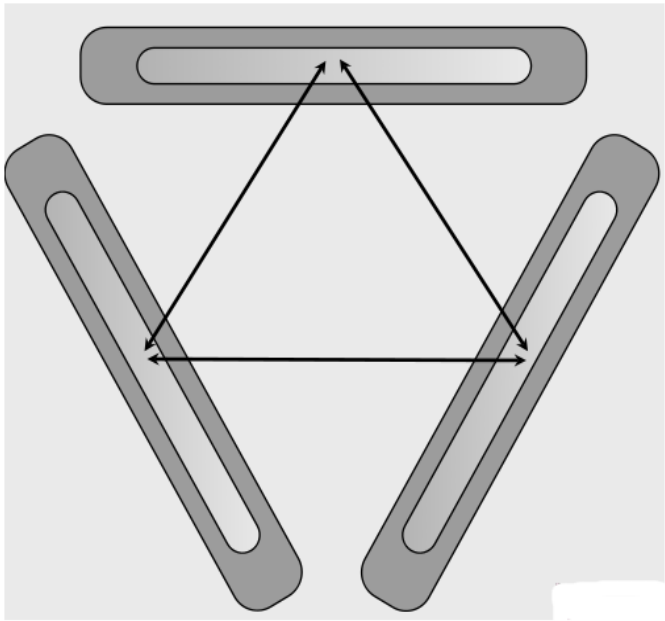
MPI collective hybridization

- 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)



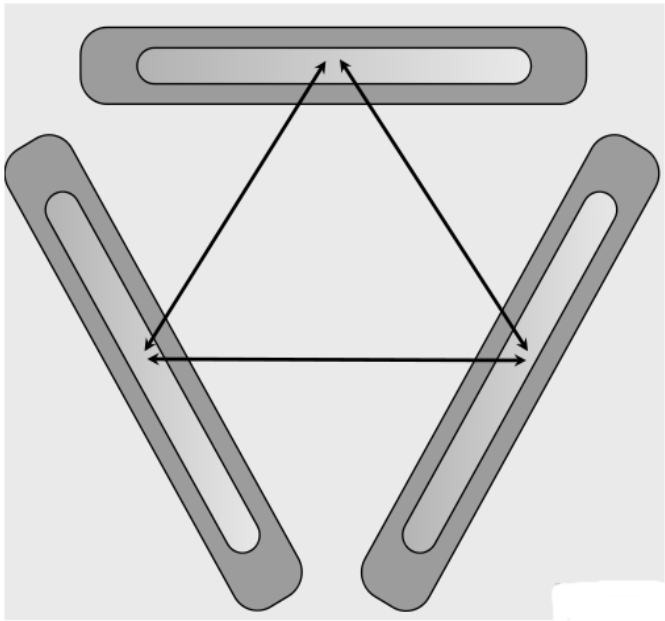
MPI collective hybridization

- Hybrid implementation:
- 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

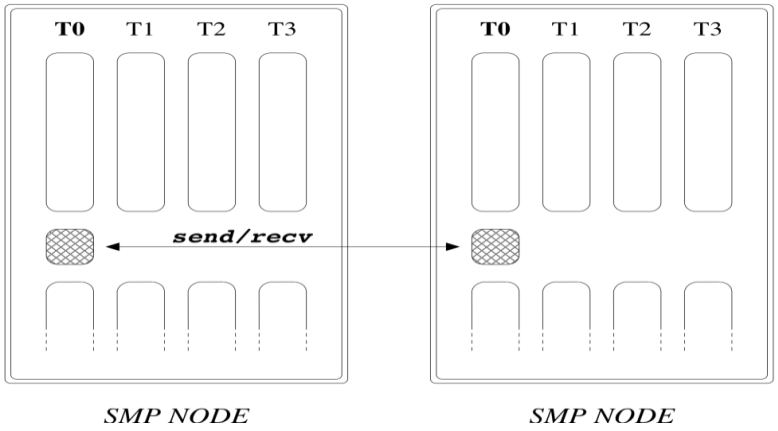
- Restrictions:
- In MPI_THREAD_MULTIPLE mode is forbidden at any given time two threads each do a collective call on the same communicator (MPI_COMM_WORLD)
- 2 threads calling each a MPI_Allreduce may produce wrong results
- **Use different communicators for each collective call**
- **Do collective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



Hybrid programming via multithreaded library

- 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, NAG (vendor), FFTW** are well known multithreaded libraries available in the HPC arena.
- MPI_THREAD_FUNNELED (almost) must be supported.

Hybrid programming via multithreaded library



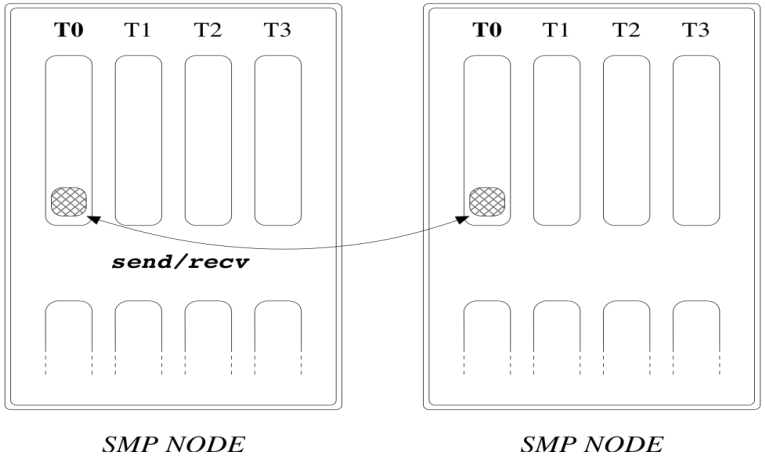
Only the master thread can do MPI communications (Pseudo QE code)

```
# 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
```

Hybrid programming via multithreaded library



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
```

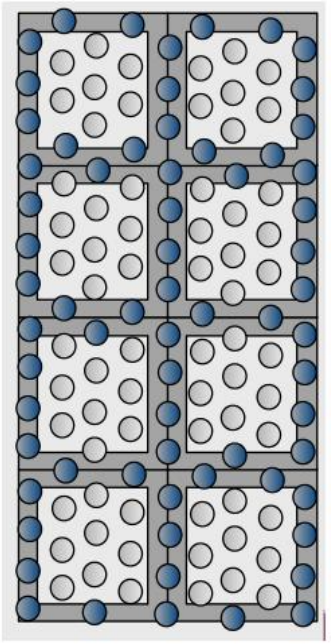
Hybrid programming via domain decomposition

- Starting point is a well known MPI parallel code that solve Helmoltz Partial Differential Equation on a square domain.
- Standard domain decomposition (into slices for simplicity).
- No huge I/O
- The benchmark collect the timing of the main computational routine (Jacobi), GFLOPS rate, the number of iterations to reach fixed error and the error with respect to known analytical solution

Domain decomposition

- In the MPI basic implementation, each process has to **exchange ghost-cells at every iteration** (also on the same node)

```
 reqcnt = 0
        if ( me .ne. 0 ) then
!           receive stripe mlo from left neighbour blocking
            reqcnt = reqcnt + 1
            call MPI_IRECV( uold(1,mlo), n, MPI_DOUBLE_PRECISION,
me,1, 11, MPI_COMM_WORLD,reqary(reqcnt),ierr)
        end if
        if ( me .ne. np-1 ) then
!           receive stripe mhi from right neighbour blocking
            reqcnt = reqcnt + 1
…
        if ( me .ne. 0 ) then
!           send stripe mlo+1 to left neighbour async
            reqcnt = reqcnt + 1
            call MPI_ISEND ( u(1,mlo+1), n, MPI_DOUBLE_PRECISION,
               me-1, 12, MPI_COMM_WORLD,reqary(reqcnt),ierr)
 end if
```



Domain decomposition

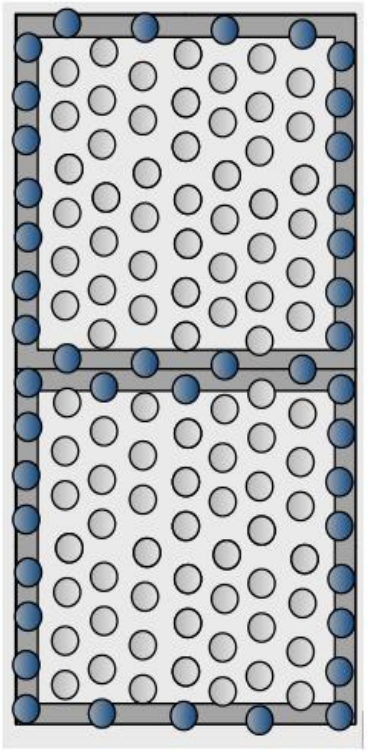
- The pseudo code for the rest of the Jacobi routines:

```
do j=mlo+1,mhi-1
    do i=1,n
       uold(i,j) = u(i,j)
    enddo
  enddo
  call MPI_WAITALL ( reqcnt, reqary, reqstat, ierr)
```

```
do j = mlo+1,mhi-1
            do i = 2,n-1
!     Evaluate residual
               resid = (ax*(uold(i-1,j) + uold(i+1,j)) +…
     &                  + b * uold(i,j) - f(i,j))/b
               u(i,j) = uold(i,j) - omega * resid
! Accumulate residual error
               error = error + resid*resid
            end do
          enddo
          error_local = error
          call MPI_ALLREDUCE ( error_local,….,error,…)
```

Domain decomposition hybridization

- The hybrid approach allows you to share the memory area where ghost-cells are stored
- In the Hybrid Masteronly, each thread has not to do MPI communication within the node, since it already has available data (via shared memory).
- Communication decreases as the number of MPI process, but increases MPI message size for Jacobi routine.



Hybrid Masteronly Domain decomposition

- Advantages:
- No message passing inside SMP nodes
- Simplest hybrid parallelization (easy to implement)
- Major problems:
- All other threads are sleeping while master thread communicate

```
!$omp parallel
!$omp do
        do j=mlo+1,mhi-1
          do i=1,n
            uold(i,j) = u(i,j)
          enddo
        enddo
!$omp end do
!$omp end parallel
 call MPI_WAITALL ( reqcnt, reqary, reqstat, ierr)
```

MPI_THREAD_FUNNELED domain decomposition

Only the master thread can do MPI communications.

The other threads are sleeping as in the previous case

```
!$omp parallel  default(shared)
!$omp master
        error = 0.0
        …
        if ( me .ne. 0 ) then
!           receive stripe mlo from left neighbour blocking
           reqcnt = reqcnt + 1
           call MPI_IRECV( uold(1,mlo), n, MPI_DOUBLE_PRECISION, &
     &        me-1, 11, MPI_COMM_WORLD,reqary(reqcnt),ierr)
        end if
        ….
!$omp end master
!$omp do
        do j=mlo+1,mhi-1
           do i=1,n
              uold(i,j) = u(i,j)
           enddo
        enddo
!$omp end do
```

MPI_THREAD_FUNNELED domain decomposition

The barrier is needed after *omp_ master* directive in order to ensure correctness of results.

```
!$omp master
 call MPI_WAITALL ( reqcnt, reqary, reqstat, ierr)
!$omp end master
!$omp barrier
! Compute stencil, residual, & update
!$omp do private(resid) reduction(+:error)
        do j = mlo+1,mhi-1
           do i = 2,n-1
                ….
 error = error + resid*resid
           end do
        enddo
!$omp end do
!$omp master
        …
        call MPI_ALLREDUCE ( error_local, error,1, &
    &      MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,ierr)
!$omp end master
!$omp end parallel
```

MPI_THREAD_SERIALIZED domain decomposition

omp_single guarantee serialized threads access . Note that no **barrier** is needed because ***omp_single*** guarantee **synchronization** at the end

```
!$omp parallel  default(shared)
!$omp single
        error = 0.0
        reqcnt = 0
        if ( me .ne. 0 ) then
!           receive stripe mlo from left neighbour blocking
           reqcnt = reqcnt + 1
           call MPI_IRECV( uold(1,mlo), n, MPI_DOUBLE_PRECISION, &
    &        me-1, 11, MPI_COMM_WORLD,reqary(reqcnt),ierr)
 end if
!$omp end single
!$omp single
        if ( me .ne. np-1 ) then
!           receive stripe mhi from right neighbour blocking
           reqcnt = reqcnt + 1
           call MPI_IRECV( uold(1,mhi), n, MPI_DOUBLE_PRECISION, &
    &        me+1, 12, MPI_COMM_WORLD,reqary(reqcnt),ierr)
 end if
!$omp end single
….
```

MPI_THREAD_SERIALIZED

omp_single **guarantee only one threads access to the MPI_Allreduce collective.**

```
…
!$omp do private(resid) reduction(+:error)
        do j = mlo+1,mhi-1
          do i = 2,n-1
!     Evaluate residual
              resid = (ax*(uold(i-1,j) + uold(i+1,j)) &
    &               + ay*(uold(i,j-1) + uold(i,j+1)) &
    &                + b * uold(i,j) - f(i,j))/b
! Update solution
              u(i,j) = uold(i,j) - omega * resid
! Accumulate residual error
              error = error + resid*resid
          end do
        enddo
!$omp end do
!$omp single
        error_local = error
        call MPI_ALLREDUCE ( error_local, error,1, …)
!$omp end single
!$omp end parallel
```

MPI_THREAD_MULTIPLE domain decomposition

- Each thread can make communications at any times (in principle)
- Some little change in the Jacobi routine
- Use of *omp sections* construct (it ensures that each thread is allowed a different MPI call at the same time)
- Use of *omp single* for MPI_Waitall and collectives

MPI_THREAD_MULTIPLE

*leftr, rightr,lefts and rights must to be **private** to ensure correct MPI calls.*

```
!$omp parallel default(shared) private(leftr,rightr,lefts,rights)
        error = 0.0
!$omp sections
!$omp section
        if ( me .ne. 0 ) then
!           receive stripe mlo from left neighbour blocking
          leftr=me-1
         else
          leftr=MPI_PROC_NULL
        endif
          call MPI_IRECV( uold(1,mlo), n, MPI_DOUBLE_PRECISION, &
    &         leftr, 11, MPI_COMM_WORLD,reqary(1),ierr)
!$omp section
….
!$omp end sections
!$omp do
        do j=mlo+1,mhi-1
          do i=1,n
            uold(i,j) = u(i,j)
          enddo
        enddo
!$omp end  do
```

MPI_THREAD_MULTIPLE

omp single is used both for MPI_Waitall call that for MPI_Allreduce collective.

```
!$omp single
 call MPI_WAITALL ( 4, reqary, reqstat, ierr)
!$omp end single
! Compute stencil, residual, & update
!$omp do private(resid) reduction(+:error)
        do j = mlo+1,mhi-1
           …
!     Evaluate residual
              resid = (ax*(uold(i-1,j) + uold(i+1,j)) …
          ….
! Update solution
              u(i,j) = uold(i,j) - omega * resid
! Accumulate residual error
              error = error + resid*resid
           …
!$omp end do
!$omp single
        …
        call MPI_ALLREDUCE ( error_local, error,1,…)
        error = sqrt(error)/dble(n*m)
!$omp end single
!$omp end parallel
```

Some results on bgq@CINECA

Up to 64 hardware threads per process are available on bgq (SMT)

Huge simulation, 30000x30000 points. Stopped after 100 iterations only for timing purposes.

Number of threads (process for MPI only) per node	MPI+OpenMP (64 MPI, 1 process per node) MPI_THREAD_MULTIPLE version Elapsed time (sec.)	MPI (1024 MPI, 16,32,64 processes per node) Elapsed time (sec.)
1	78.84	N.A
4	19.89	N.A
8	10.33	N.A
16	5.65	5.98
32	3.39	7.12
64	2.70	12.07

Lesson learned

- Better scalability by a reduction of both the number of MPI messages and the number of processes involved in collective communications and by a better load balancing.
- Better adeguacy to the architecture of modern supercomputers while MPI is only a flat approach.
- Optimization of the total memory consumption (through the OpenMP shared-memory approach, savings in replicated data in the MPI processes and in the used memory by the MPI library itself).
- Reduction of the footprint memory when the size of some data structures depends directly on the number of MPI processes.
- It can remove algorithmic limitations (maximum decomposition in one direction for example).

Applications that can benefit from it

- 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: there is no golden rule….

- 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).