



**21st Summer
School of
PARALLEL
COMPUTING**

July 2 - 13, 2012 (Italian)

September 3 - 14, 2012 (English)

Hybrid programming MPI+OpenMP

Gabriele Fatigati - [g.fatigati@Cineca.it](mailto:g.fatigati@ Cineca.it)

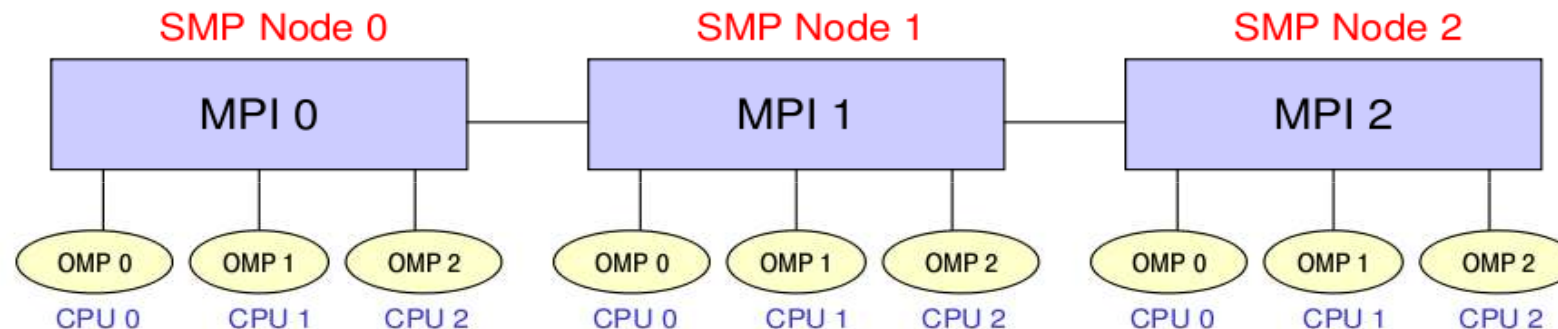
Supercomputing Group





The hybrid model

- ❖ Multi-node SMP (Symmetric Multiprocessor) connected by an interconnection network.
- ❖ Each node is mapped (at least) one process MPI and OpenMP threads more.





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 long waits for unlocking data
- 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.
- ❖ The two-level parallelism introduces other problems
- ❖ 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.



Optimizing the memory usage

Each MPI process needs to allocate some extra memory to manage communications and MPI environment.

Threads uses less memory than process. No extra memory => shared memory

Example: one node having 8 cores and 32 GB. Two ways:

Pure MPI: 8 MPI process, 4 GB for each

Pure MPI: 1 MPI process, 32 GB

Hybrid: 1 MPI process, 8 threads. 32 GB shared per process, 4 GB per thread.



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
- ❖ Overhead of thread creation
- ❖ Cache coherency and false sharing.
- ❖ Pure OpenMP code is generally slower than pure MPI code
- ❖ Few optimizations by OpenMP compilers compared to MPI



False sharing in OpenMP

```
#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]



- ❖ The cache uses the principle of data spatial proximity
- ❖ Concurrent updates to individual elements of the same threads from different cache line invalidate the entire cache line.
- ❖ Once the cache line is marked as invalid, subsequent threads are forced to fetch the data from main memory, to ensure cache coherency.



- ❖ This happens because the cache coherence is cache line based, not on individual item
- ❖ A cache that load a single element would not apply spatial locality, and therefore, any new data would require fetch from the main memory
- ❖ Read-only data does not have this problem



Pseudo 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)
```



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 run. 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_THREAD_SINGLE

Hot to implement:

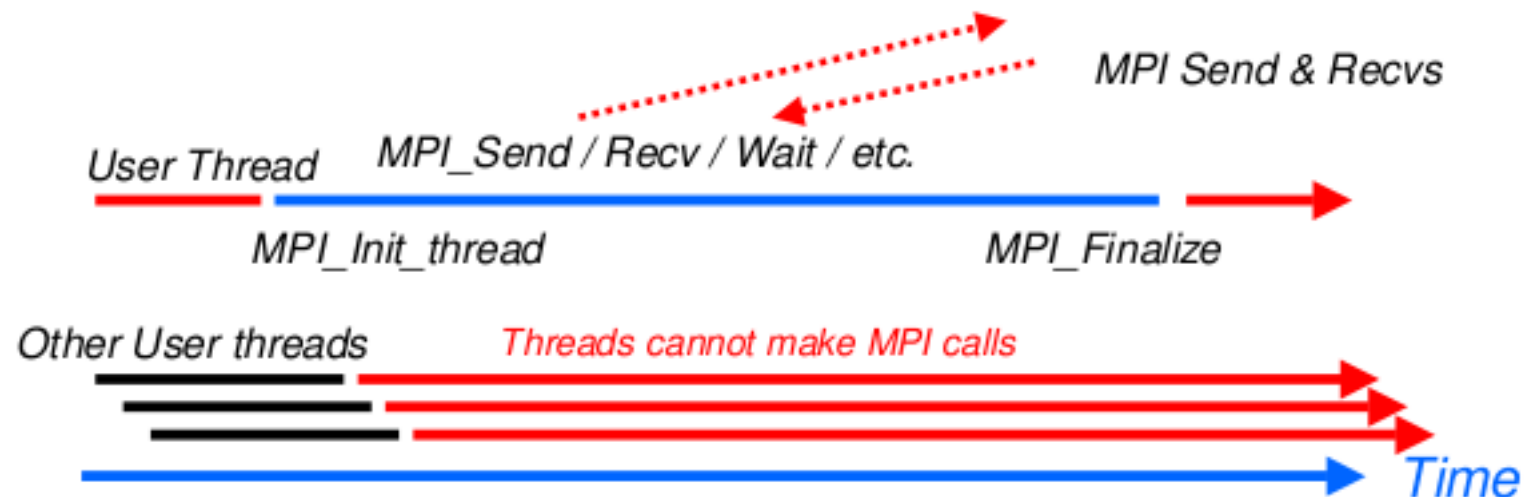
```
!$OMP PARALLEL DO
  do i=1,10000
    a(i)=b(i)+f*d(i)
  enddo
!$OMP END PARALLEL DO
  call MPI_Xxx(...)
!$OMP PARALLEL DO
  do i=1,10000
    x(i)=a(i)+f*b(i)
  enddo
!$OMP END PARALLEL DO
```

```
#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 main thread can do MPI communications. Obviously, there is a main thread for each node





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
```

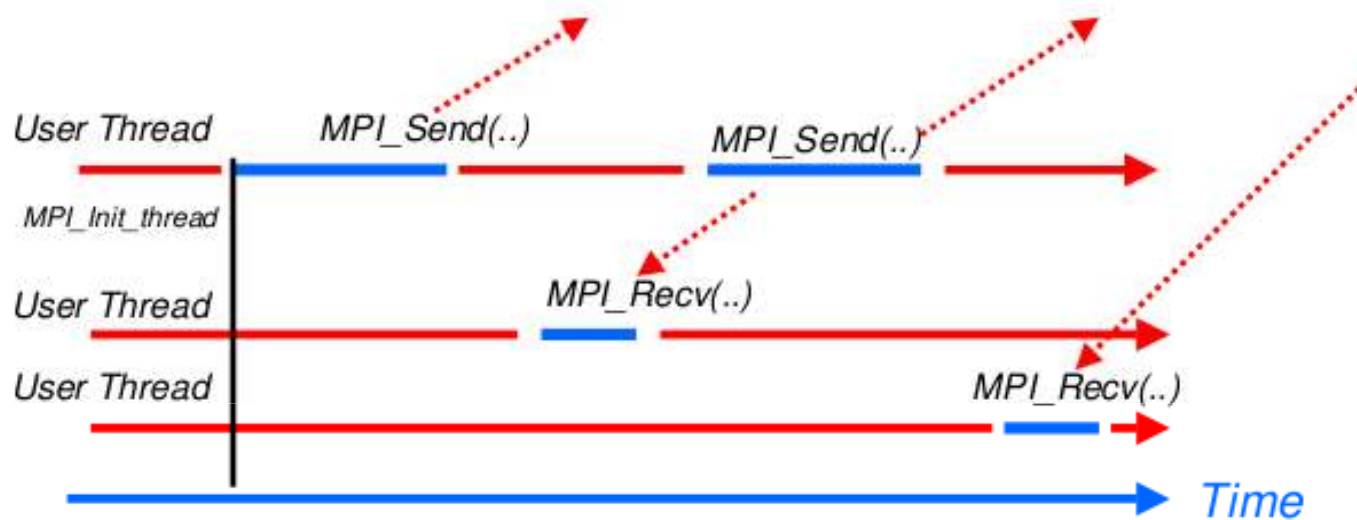
```
#pragma omp barrier  
#pragma omp master  
    MPI_Xxx(...);  
#pragma omp barrier
```

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

- ❖ Outside the parallel region
- ❖ Inside the parallel region with "omp master"
- ❖ Inside the parallel region with "omp single"

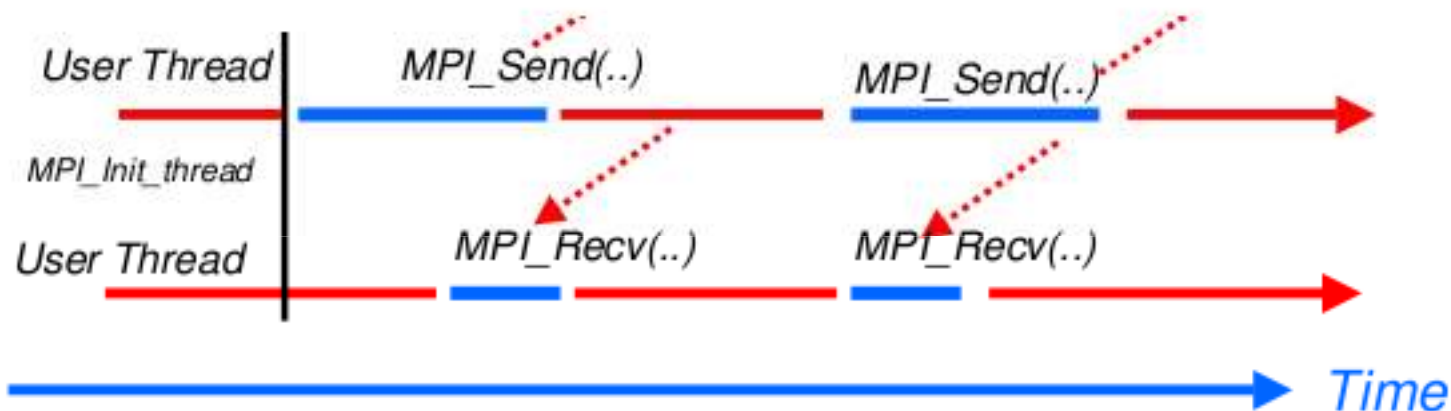
```
!$OMP BARRIER  
!$OMP SINGLE  
    call MPI_Xxx(...)  
!$OMP END SINGLE
```

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




MPI_THREAD_MULTIPLE

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





A little example

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
  int rank,omp_rank,mpisupport;
  MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED, &mpisupport);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  omp_set_num_threads(atoi(argv[1]));
  #pragma omp parallel private(omp_rank)
  {
    omp_rank=omp_get_thread_num();
    printf("%d %d \n",rank,omp_rank);
  }
  MPI_Finalize();
}
```

```
Output--> 0 0
           0 2
           0 1
           0 3
           1 0
           1 2
           1 1
           1 3
```



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
```



THREAD FUNNELED/SERIALIZED vs. Pure MPI

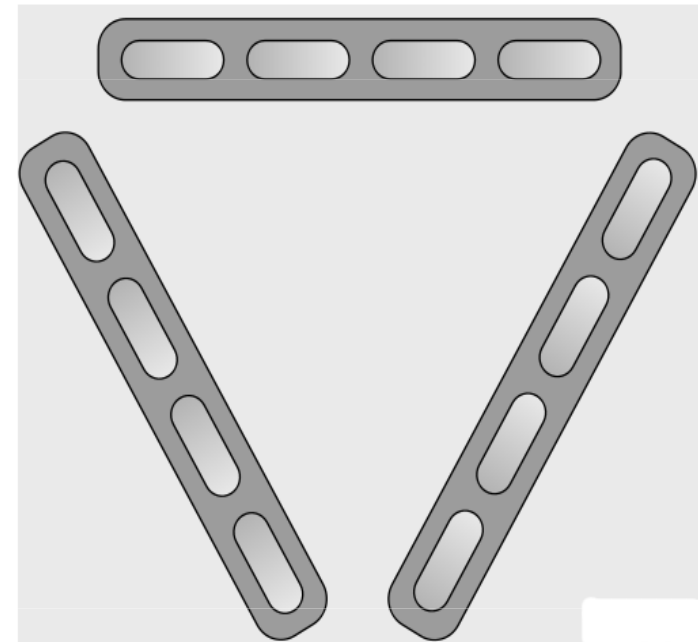
- ❖ FUNNELED/SERIALIZED:
 - ❖ All other threads are sleeping while just one thread is communicating.
 - ❖ Only one thread may not be able to lead up max internode bandwidth
- ❖ Pure MPI:
 - ❖ Each CPU communication can lead up max internode bandwidth
- ❖ Overlap communications and computations.



- ❖ 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
- ❖ In OpenMPI, you have to use `-enable-mpi-threads` at compile time to activate all levels.
- ❖ Higher level corresponds higher thread-safety. Use the required safety needs.

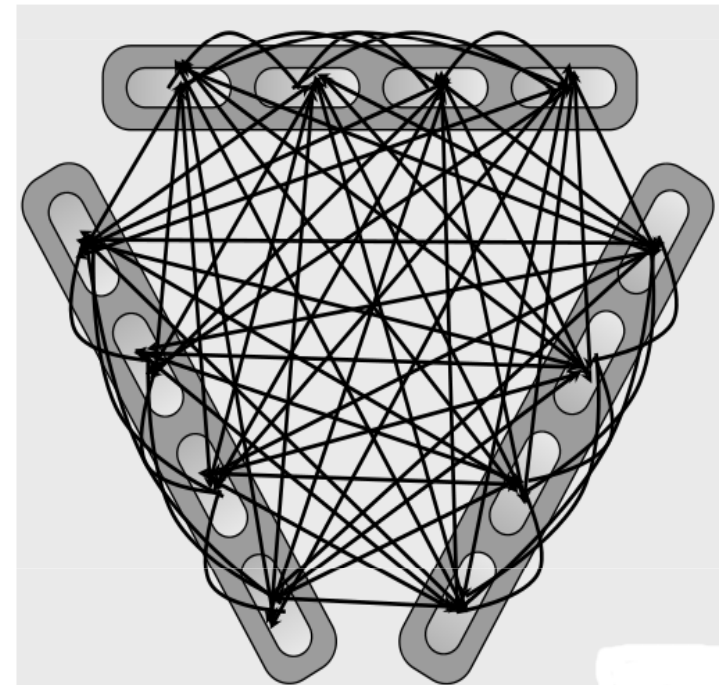


- ❖ Collective operations are often bottlenecks
 - ❖ All-to-all communications
 - ❖ Point-to-point can be faster
- ❖ Hybrid implementation:
 - ❖ For all-to-all communications, the number of transfers decrease by a factor $\#threads^2$
 - ❖ The length of messages increases by a factor $\#threads$
 - ❖ Allow to overlap communication and computation.



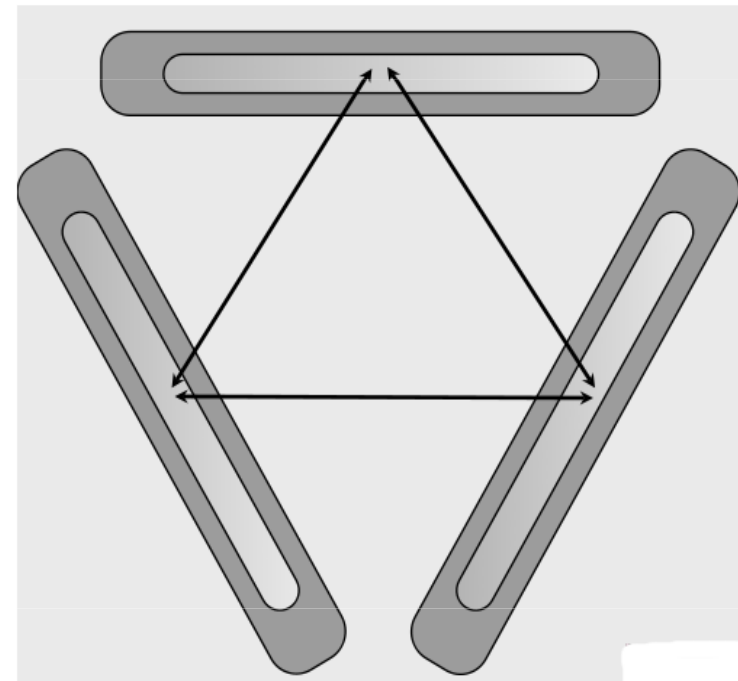


- ❖ Collective operations are often bottlenecks
 - ❖ All-to-all communications
 - ❖ Point-to-point can be faster
- ❖ Hybrid implementation:
 - ❖ For all-to-all communications, the number of transfers decrease by a factor $\#threads^2$
 - ❖ The length of messages increases by a factor $\#threads$
 - ❖ Allow to overlap communication and computation.





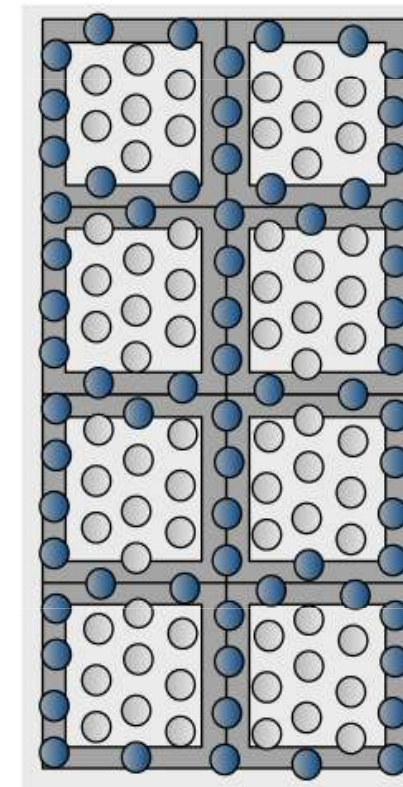
- ❖ Collective operations are often bottlenecks
 - ❖ All-to-all communications
 - ❖ Point-to-point can be faster
- ❖ Hybrid implementation:
 - ❖ For all-to-all communications, the number of transfers decrease by a factor $\#threads^2$
 - ❖ The length of messages increases by a factor $\#threads$
 - ❖ Allow to overlap communication and computation.





Domain decomposition

- ❖ In MPI implementation, each process has to exchange ghost-cell
- ❖ This even two different processes are within the same node. This is because two different process do not share the same memory





Domain decomposition

- ❖ The hybrid approach allows you to share the memory area where ghost-cells are stored
- ❖ Each thread has not to do communication within the node, since it already has available data.
- ❖ Communication decreases, and as in the previous case, increases MPI message size.

