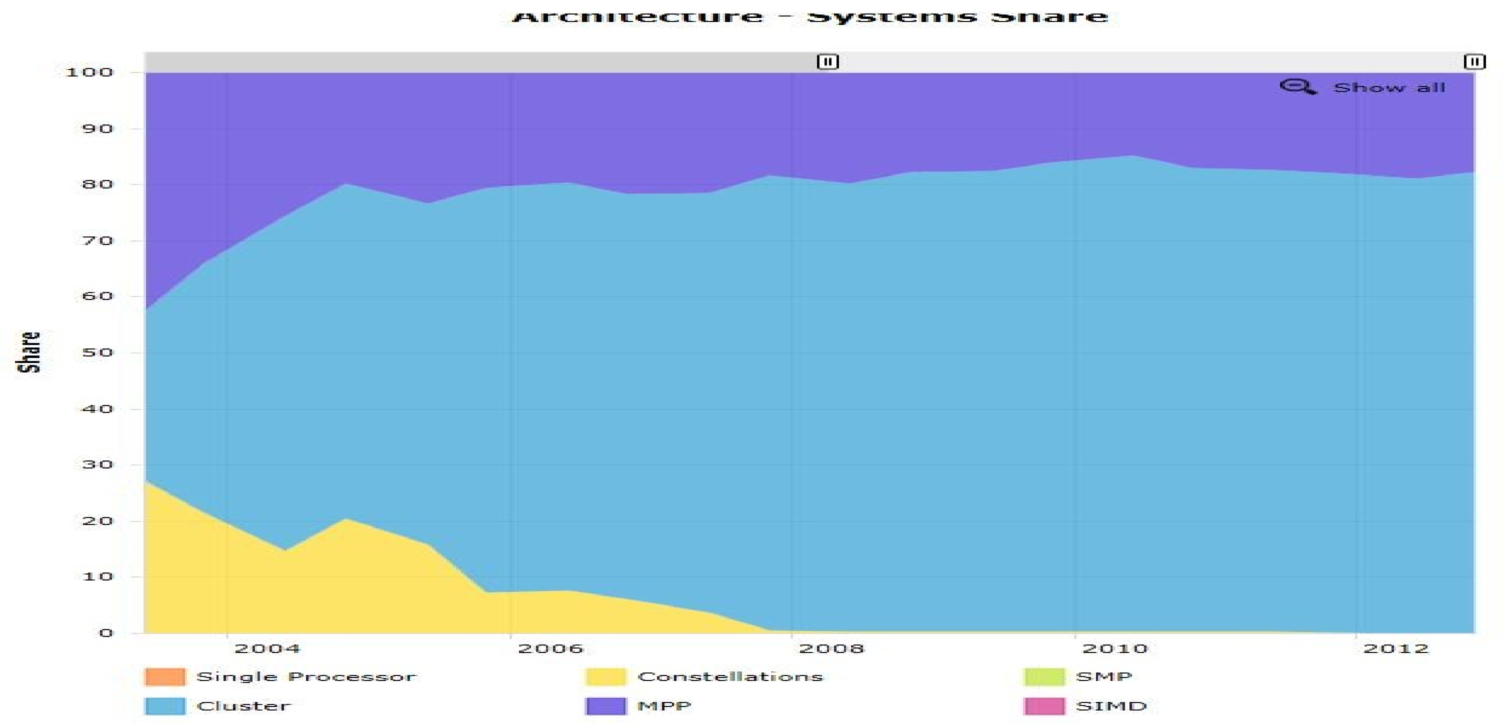
Summer School on PARALLEL COMPUTING

Introduction to MPI+OpenMP hybrid programming

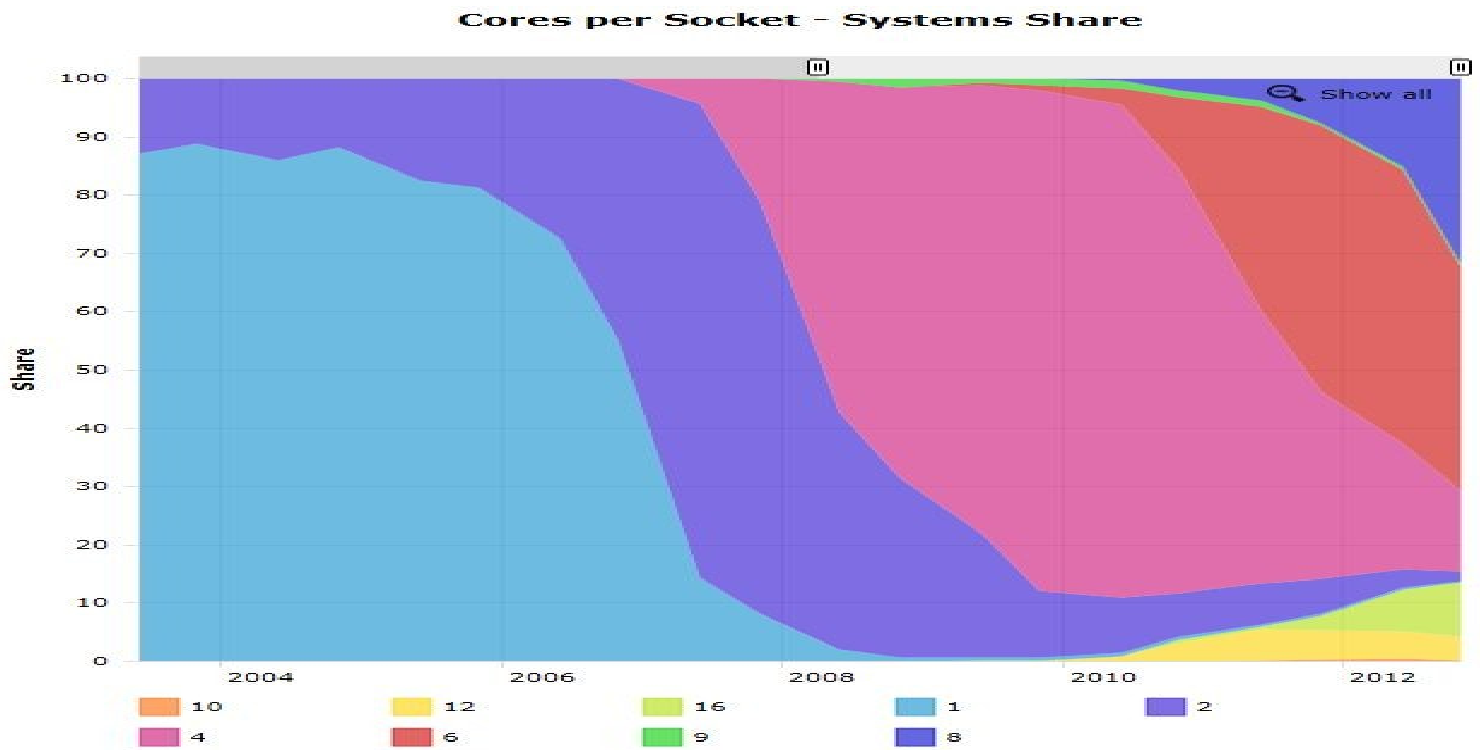
Fabio Affinito– f.affinito@cineca.it
SuperComputing Applications and Innovation Department

CINECA

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

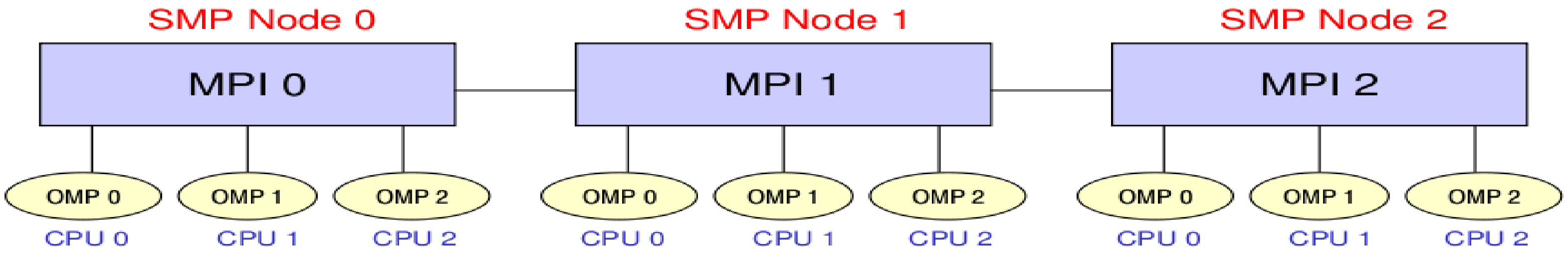
- OpenMP is robust, clear and sufficiently easy to implement but
  - depending on the implementation, typically the scaling on the number of threads is much less effective than the scaling on number of MPI tasks

- 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

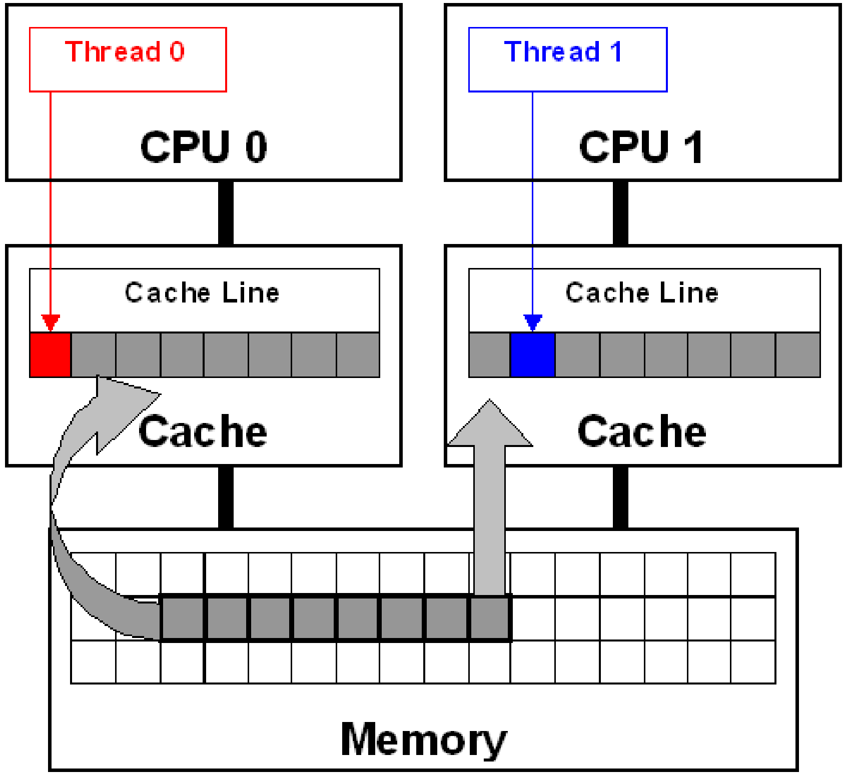
- In real practise, mixing MPI and OpenMP, sometimes, can make your code slower
  - If you exceed with the number of OpenMP threads you can encounter problems with locking of resources
  - Sometimes threads can stay in a idle state (spin) for a long time
  - Problems with cache coherency and false sharing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

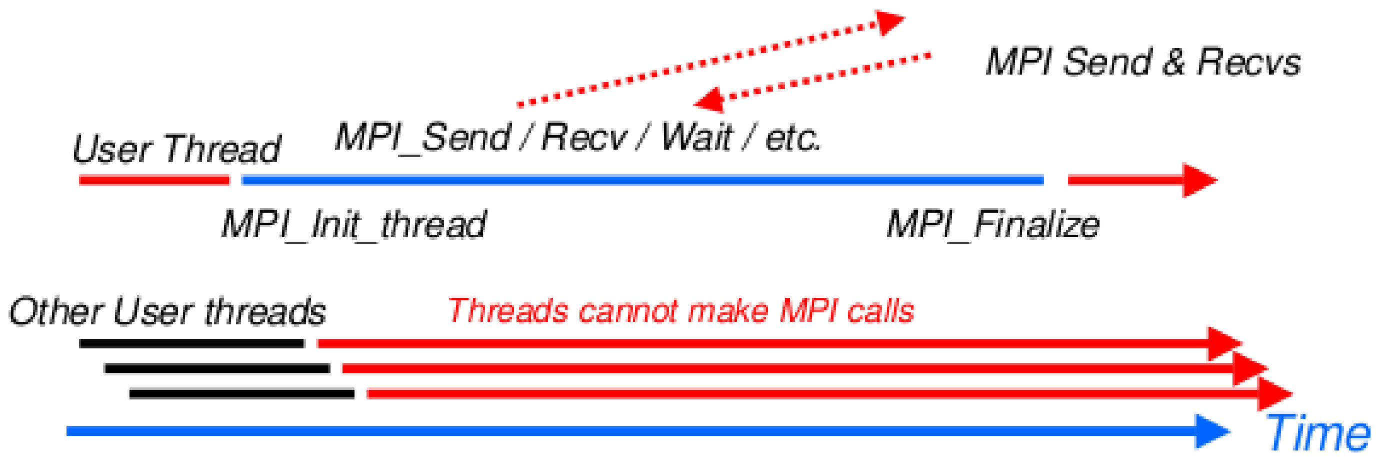
- It is fully equivalent to the master-only approach

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

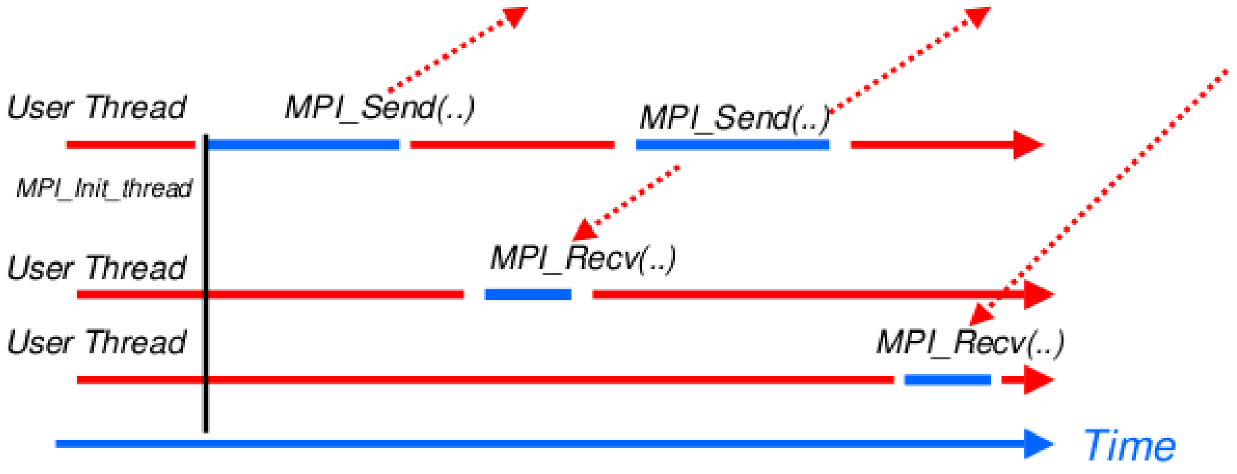
- MPI function calls can be: outside a parallel region or in a parallel region, enclosed in “omp master” clause
- There's no synchronization at the end of a “omp master” region, so a barrier is needed before and after to ensure that data buffers are available before/after the MPI communication

```
!$OMP BARRIER
!$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

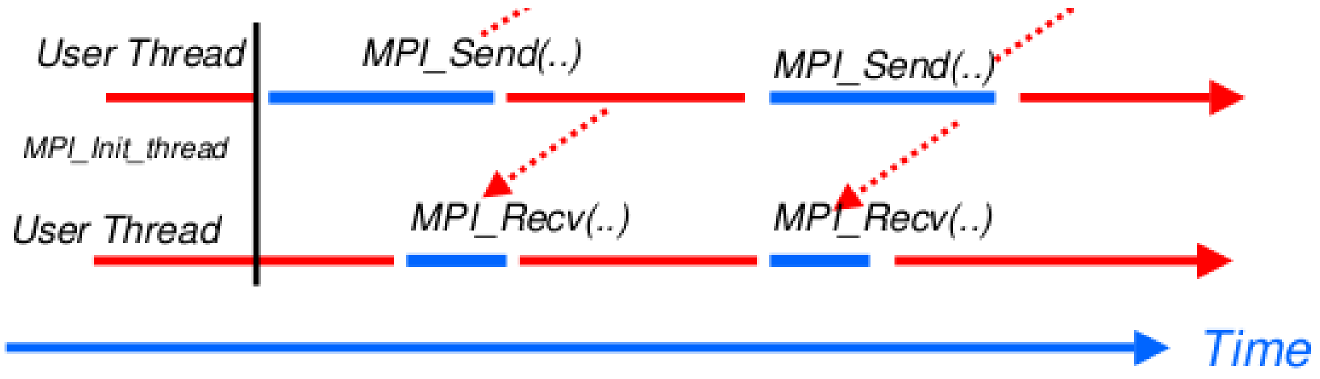
- MPI calls can be outside parallel regions, or inside, but enclosed in a “omp single” region (it enforces the serialization)
- Again, a barrier should ensure data consistency

```
!$OMP BARRIER
!$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

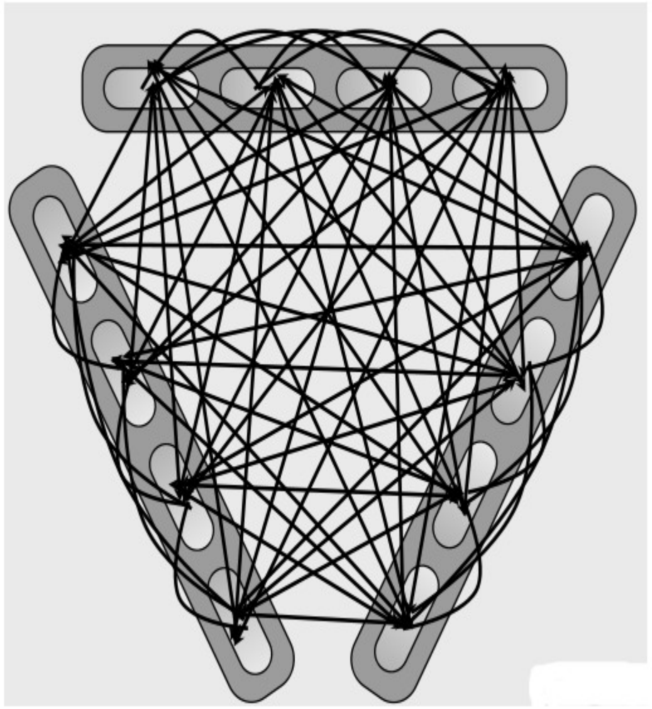
Overlap communications and computations

- 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
```

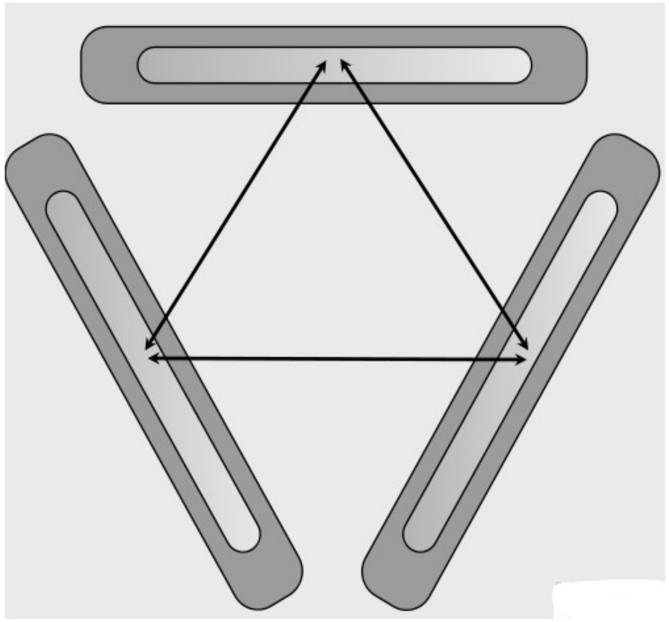
MPI collective hybridization

- 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

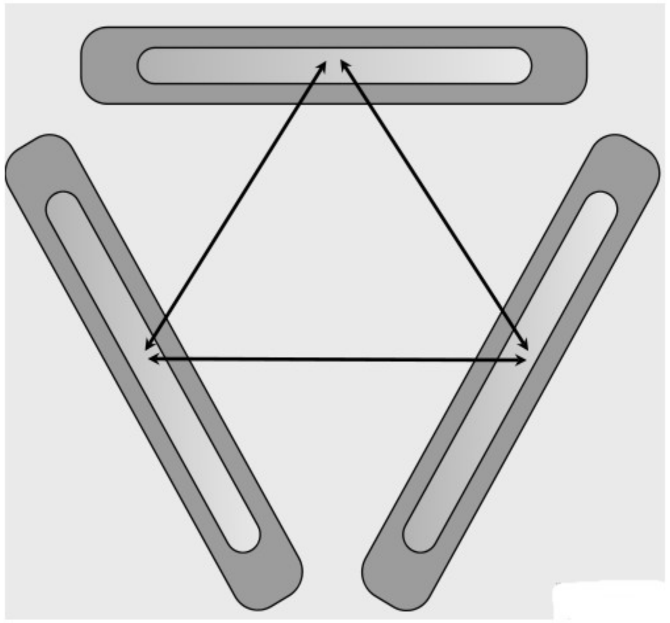
- Better scalability by a reduction of both the number of MPI messages and the number of process. Tipically:
- for all-to-all communications, the number of transfers decrease by a factor #threads^2
- the length of messages increases by a factor #threads
- Allow to overlap communication and computation.



MPI collective hybridization

Restrictions:

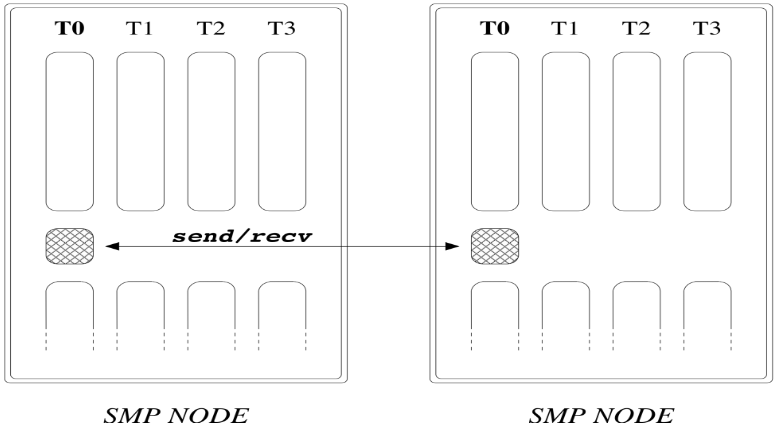
- In MPI_THREAD_MULTIPLE mode is forbidden at any given time two threads each do a collective call on the same communicator (MPI_COMM_WORLD)
- 2 threads calling each a MPI_Allreduce may produce wrong results
- **Use different communicators for each collective call**
- **Do collective calls only on 1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



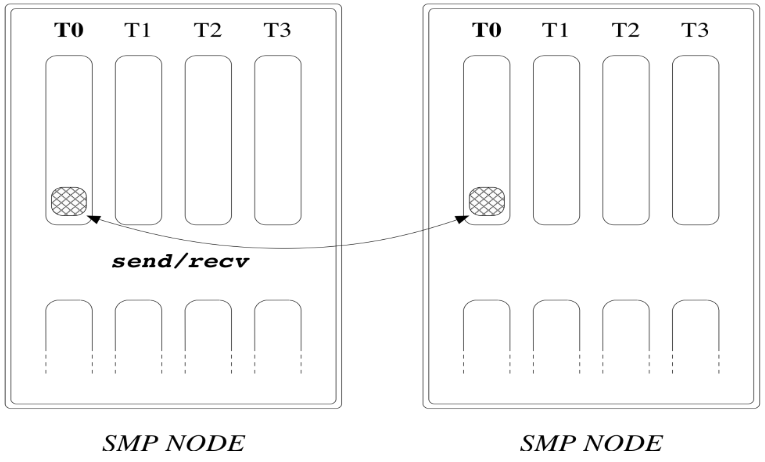
Only the master thread can do MPI communications (Pseudo QE code)

```
# 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)



```
# 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
```

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

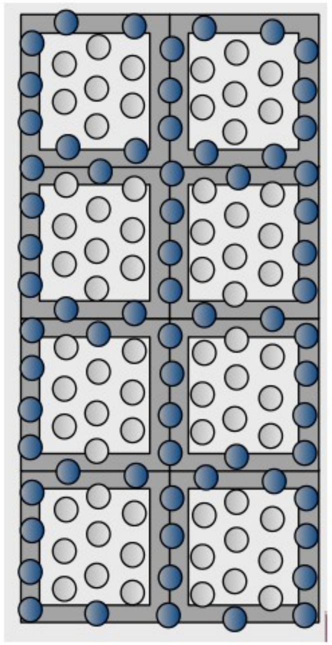
Domain decomposition

- 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

```
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

- The hybrid approach allows you to share the memory area where ghost-cells are stored
- In the master-only approach, 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.

Master-only 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)
```

Thread funneled domain dec.

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
```

Thread funneled domain dec.

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
```

Thread serialized domain dec.

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
….
```

Thread serialized domain dec.

***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
```

Thread multiple domain dec.

- 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

Thread multiple domain dec.

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
```

Thread multiple domain dec

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
```

BGQ benchmarks

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

Number of threads / processes	MPI+OpenMP (TOT= 64 MPI, 1PPN) MPI_THREAD_MULTIPLE version Elapsed time (sec.)	MPI ONLY (TOT= 1024 MPI, 16,32,64 ppn) 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

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

Conclusions

- 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).

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.

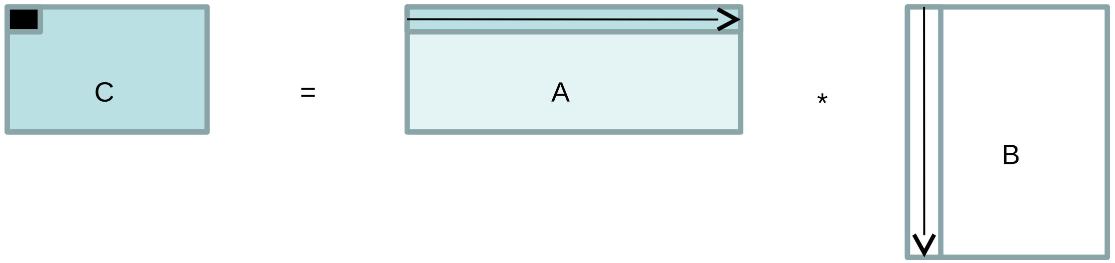
Conclusions

- 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).

Backup slides

Case-Study: Matrix Multiplication

```
do i = ioff, iend
  do j = joff, jend
    do l = loff, lend
      c( i, j ) = c( i, j ) + a( i, l ) * b( l, j )
    end do
  end do
end do
```

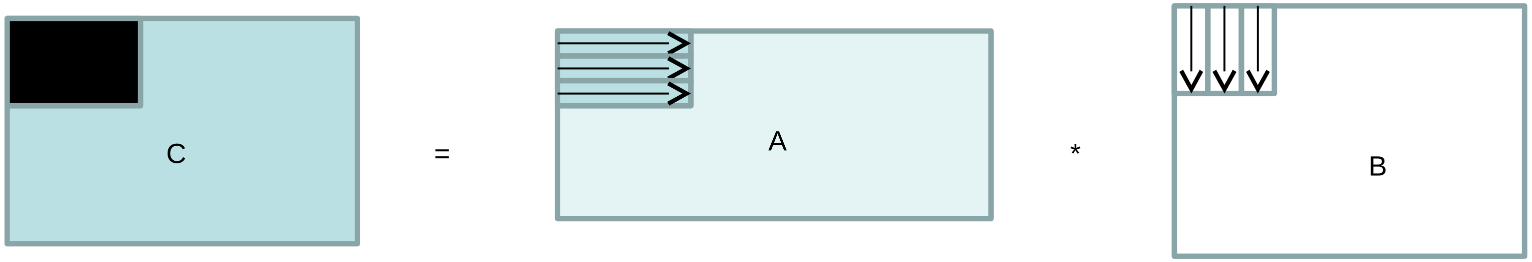


OpenMP parallelization

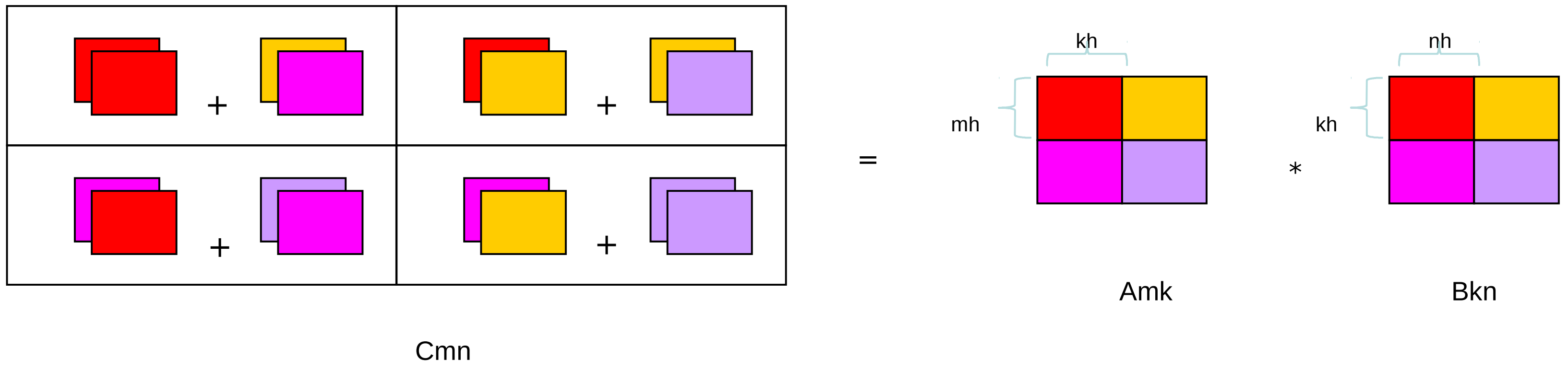
```
!$omp parallel do default(none) &
!$omp          shared(a,b,c,ioff,joff,loff,iend,jend,lend) &
!$omp          private(i,j,l)
do i = ioff, iend
  do j = joff, jend
    do l = loff, lend
      c( i, j ) = c( i, j ) + a( i, l ) * b( l, j )
    end do
  end do
end do
!$omp end parallel do
```

Not really efficient

Cache blocking



Compute blocks



m, k, n: matrixes sizes

mh, kh, nh: block sizes, “Free” parameters

mb, kb, nb: number of blocks

Cache blocking algorithm

```
do ib = 0, mb-1
     ioff = 1 + ib * mh
     iend = MIN( m, ioff+mh-1)
        do jb = 0, nb-1
           joff = 1 + jb * nh
           jend = MIN( n, joff+nh-1 )
           do lb = 0, kb-1
              loff = 1 + lb * kh
              lend = MIN( k, loff+kh-1 )
         ! Cij = Aik * Bkj
              do i = ioff, iend
                  do j = joff, jend
                      do l = loff, lend
                         c( i, j ) = c( i, j ) + a( i, l ) * b( l, j )
                      end do
                  end do
              end do
           end do
        end do
 end do
```

Cache friendly OpenMP

```
!$omp parallel do default(none) &
!$omp          shared(a,b,c,mb,nb,kb,m,n,k,mh,nh,kh) &
!$omp          private(ib,jb,lb,i,j,l,ioff,joff,loff,iend,jend,lend)
   do ib = 0, mb-1
      ioff = 1 + ib * mh
      iend = MIN( m, ioff+mh-1)
      do jb = 0, nb-1
         joff = 1 + jb * nh
         jend = MIN( n, joff+nh-1 )
         do lb = 0, kb-1
            loff = 1 + lb * kh
            lend = MIN( k, loff+kh-1 )
            ! Cij = Aik * Bkj
            do i = ioff, iend
               do j = joff, jend
                  do l = loff, lend
                     c( i, j ) = c( i, j ) + a( i, l ) * b( l, j )
                  end do
               end do
            end do
         end do
      end do
   end do
!$omp end parallel do
```

Using blas library

```
!$omp parallel default(none) &
!$omp private( mytid, ntids, ntids_row, ntids_col, myrow, mycol, mb, nb, m_off, n_off) &
!$omp shared( m, n, k, lda, ldb, ldc, a, b, c )

   mytid = omp_get_thread_num()  ! get the thread ID
   ntids = omp_get_num_threads() ! get the number of threads

   ! define a grid of threads as square as possible
   CALL gridsetup( ntids, ntids_row, ntids_col )

   ! Find row and column thread id (row mayour order)
   myrow = MOD( mytid, ntids_row )
   mycol = mytid / ntids_row

   ! find my block size
   mb = ldim_block( m, ntids_row, myrow )
   nb = ldim_block( n, ntids_col, mycol )

   ! find the offset
   m_off = gind_block(1, m, ntids_row, myrow)
   n_off = gind_block(1, n, ntids_col, mycol)

   CALL dgemm('N','N', mb, nb, k, 1.0d0, a(m_off,1), lda, b(1,n_off), ldb, 0.0d0, c(m_off,n_off), ldc )

!$omp end parallel
```

Computing grid and blocks sizes

```
SUBROUTINE gridsetup( nproc, nprow, npcol )
! This subroutine factorizes the number of processors (NPROC)
! into NPROW and NPCOL,  that are the sizes of the 2D processors mesh.
  IMPLICIT NONE
  integer nproc,nprow,npcol
  integer sqrtnp,i
  sqrtnp = int( sqrt( dble(nproc) ) + 1 )
  do i=1,sqrtnp
    if(mod(nproc,i).eq.0) nprow = i
  end do
  npcol = nproc/nprow
  return
END SUBROUTINE
```

Computing grid and blocks sizes

```
SUBROUTINE gridsetup( nproc, nprow, npcol )
! This subroutine factorizes the number of processors (NPROC)
! into NPROW and NPCOL,  that are the sizes of the 2D processors mesh.
  IMPLICIT NONE
  integer nproc,nprow,npcol
  integer sqrtnp,i
  sqrtnp = int( sqrt( dble(nproc) ) + 1 )
```

```
 INTEGER FUNCTION ldim_block(gdim, np, me)
! This function compute the local block size of a distributed array
!   gdim = global dimension of distributed array
!   np   = number of processors
!   me   = index of the calling processor (starting from 0)
 IMPLICIT NONE
 INTEGER :: gdim, np, me, r, q
 q = INT(gdim / np)
 r = MOD(gdim, np)
 IF( me .LT. r ) THEN
    ldim_block = q+1
 ELSE
    ldim_block = q
 END IF
 RETURN
END FUNCTION ldim_block
```

Computing grid and blocks sizes

Advanced School on PARALLEL COMPUTING

```
SUBROUTINE gridsetup( nproc, nprow, npcol )
! This subroutine factorizes the number of processors (NPROC)
! into NPROW and NPCOL,  that are the sizes of the 2D processors mesh.
  IMPLICIT NONE
  integer nproc,nprow,npcol
  integer sqrtnp,i
  sqrtnp = int( sqrt( db
  do i=1,sqrtnp
    if(mod(nproc,i).eq.0
  end do
```

```
 INTEGER FUNCTION ldim_block(gd
! This function compute the loc
!   gdim = global dimension of
!   np   = number of processors
!   me   = index of the calling
 IMPLICIT NONE
 INTEGER :: gdim, np, me, r, q
 q = INT(gdim / np)
 r = MOD(gdim, np)
 IF( me .LT. r ) THEN
    ldim_block = q+1
 ELSE
    ldim_block = q
 END IF
 RETURN
END FUNCTION ldim_block
```

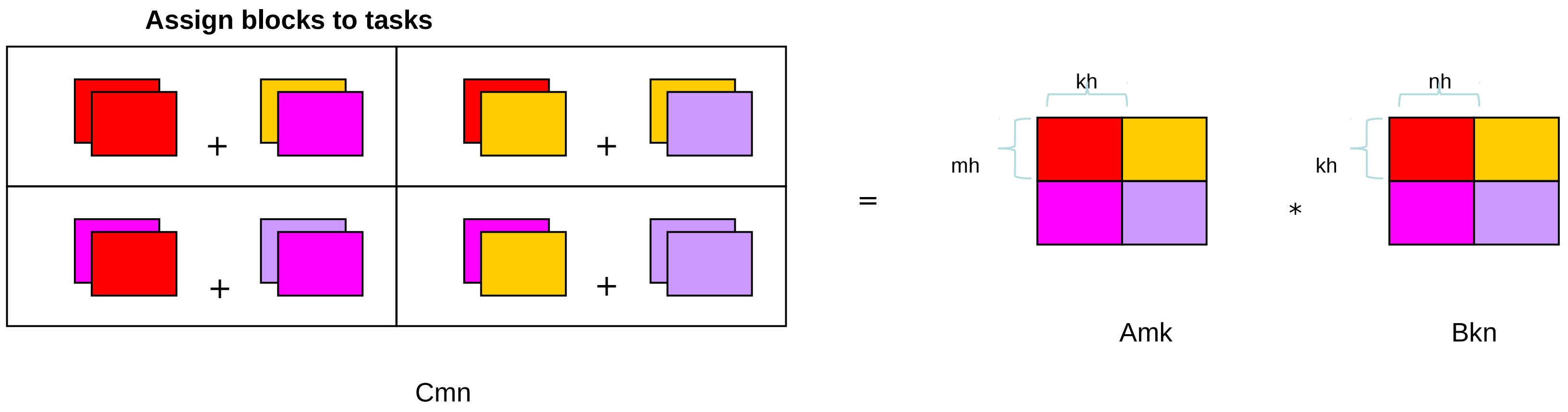
```
INTEGER FUNCTION gind_block( lind, n, np, me )
!  This function computes the global index of a distributed array element
!  pointed to by the local index lind of the process indicated by me.
!  lind      local index of the distributed matrix entry.
!  N         is the size of the global array.
!  me        The coordinate of the process whose local array row or
!            column is to be determined.
!  np        The total number processes over which the distributed
!            matrix is distributed.
   INTEGER, INTENT(IN) :: lind, n, me, np
   INTEGER :: r, q
   q = INT(n/np)
   r = MOD(n,np)
   IF( me < r ) THEN
      gind_block = (Q+1)*me + lind
   ELSE
      gind_block = Q*me + R + lind
   END IF
   RETURN
END FUNCTION gind_block
```

MPI parallelization

Matrix connot be stored in single node memory.

Multiplication takes too long.
(Matrix multiplication scale as cubic power of matrix linear dimension)

Blocks again!



m, k, n: matrixes sizes

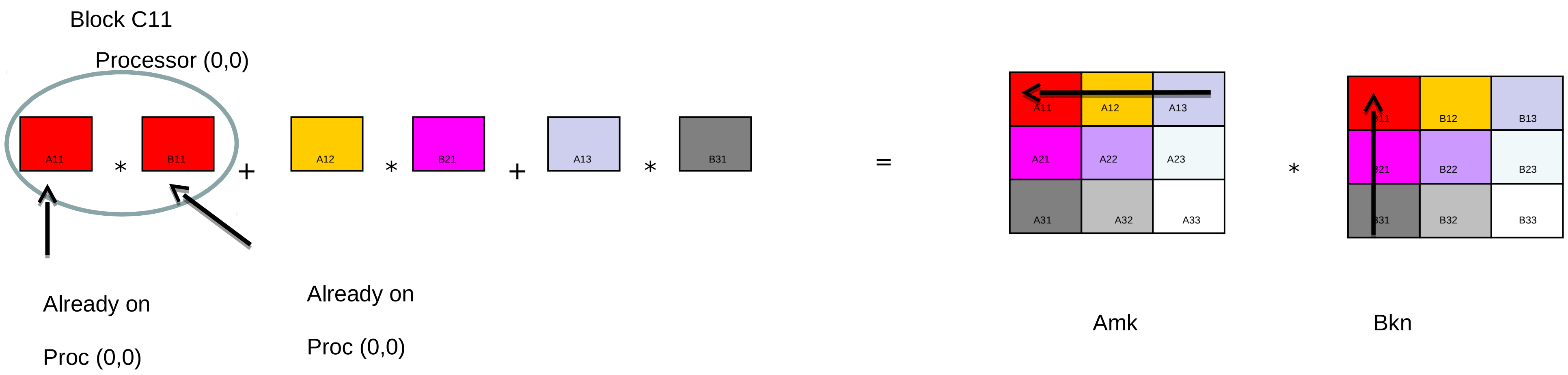
mh, kh, nh: block sizes, “Free” parameters

mb, kb, nb: number of blocks

I need to minimize communications

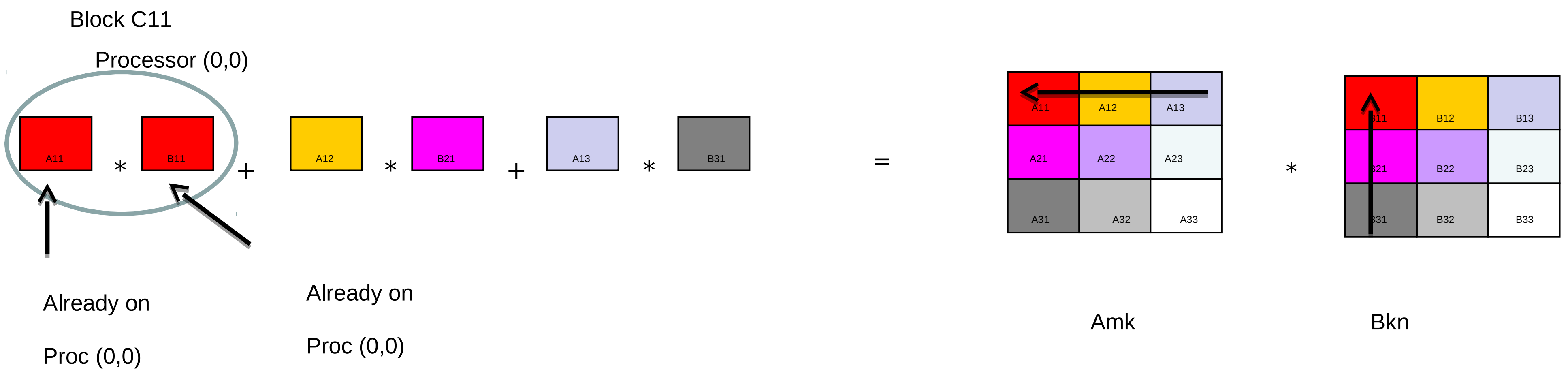
Cannon’s algorithm

Consider 3x3 processor grid



Cannon’s algorithm

Consider 3x3 processor grid



```
SUBROUTINE shift_block( blk, dir, ln, tag )
      IMPLICIT NONE
      REAL(DP) :: blk( :, : )
      CHARACTER(LEN=1), INTENT(IN) :: dir      ! shift direction
      INTEGER,          INTENT(IN) :: ln       ! shift length
      INTEGER,          INTENT(IN) :: tag      ! communication tag
      !
      INTEGER :: icdst, irdst, icsrc, irsrc, idest, isour
      !
      IF( dir == 'W' ) THEN
         irdst = rowid
         irsrc = rowid
         icdst = MOD( colid - ln + np, np )
         icsrc = MOD( colid + ln + np, np )
      ELSE IF( dir == 'E' ) THEN
         irdst = rowid
         irsrc = rowid
         icdst = MOD( colid + ln + np, np )
         icsrc = MOD( colid - ln + np, np )
      ELSE IF( dir == 'N' ) THEN
         irdst = MOD( rowid - ln + np, np )
         irsrc = MOD( rowid + ln + np, np )
         icdst = colid
         icsrc = colid
      ELSE IF( dir == 'S' ) THEN
         irdst = MOD( rowid + ln + np, np )
         irsrc = MOD( rowid - ln + np, np )
         icdst = colid
         icsrc = colid
      ELSE
         CALL errore( ' sqr_mm_cannon ', ' unknown shift direction ', 1 )
      END IF
      !
      CALL GRID2D_RANK( 'R', np, np, irdst, icdst, idest )
      CALL GRID2D_RANK( 'R', np, np, irsrc, icsrc, isour )
      !
      CALL MPI_SENDRECV_REPLACE( blk, nb*nb, MPI_DOUBLE_PRECISION, &
           idest, tag, isour, tag, comm, istatus, ierr)

      RETURN
   END SUBROUTINE shift_block
```

```
SUBROUTINE GRID2D_RANK( order, nprow, npcol, row, col, rank )
   !
   !  this subroutine compute the processor MPI task id "rank" of the processor
   !  whose cartesian coordinate are "row" and "col".
   !  Note that the subroutine assume cyclic indexing ( 0 + nprow = 0 )
   !
   IMPLICIT NONE
   CHARACTER, INTENT(IN) :: order
   INTEGER, INTENT(OUT) ::  rank           ! process index starting from 0
   INTEGER, INTENT(IN)  ::  nprow, npcol   ! dimensions of the processor grid
   INTEGER, INTENT(IN)  ::  row, col

   IF( order == 'C' .OR. order == 'c' ) THEN
     !  grid in COLUMN MAJOR ORDER
     rank = MOD( row + nprow, nprow ) + MOD( col + npcol, npcol ) * nprow
   ELSE
     !  grid in ROW MAJOR ORDER
     rank = MOD( col + npcol, npcol ) + MOD( row + nprow, nprow ) * npcol
   END IF
   !
   RETURN
END SUBROUTINE
```

Hybrid Parallel Matrix Multiplication Algorithm

Advanced School on PARALLEL COMPUTING

```
allocate( ablk( nb, nb ) )
DO j = 1, nc
   DO i = 1, nr
      ablk( i, j ) = a( i, j )
   END DO
END DO
!
allocate( bblk( nb, nb ) )
DO j = 1, nc
   DO i = 1, nr
      bblk( i, j ) = b( i, j )
   END DO
END DO

CALL shift_block( ablk, 'W', rowid+1, 1 )     !  Shift A rowid+1 places to the west
CALL shift_block( bblk, 'N', colid+1, np+1 )  !  Shift B colid+1 places to the north

CALL “serial or multithread - Matrix Multiplication”   ! Set C
!
DO iter = 2, np
   !
   CALL shift_block( ablk, 'E', 1, iter )    !  Shift A 1 places to the east
   CALL shift_block( bblk, 'S', 1, np+iter ) !  Shift B 1 places to the south
   !
   CALL “serial or multithread - Matrix Multiplication” ! Accumulate on C
   !
END DO

deallocate( ablk, bblk )
```

CINECA