9th Advanced School on PARALLEL COMPUTING

Advanced hybrid MPI+OpenMP programming

Carlo Cavazzoni– c.cavazzoni@cineca.it
SuperComputing Applications and Innovation Department

Architecture features

Floating point units (multiply and add, pipeline stages)
Floating point vector units (number of bits, concurrent FP operations)
Number of simultaneous threads
Cache: size, organization, prefetch
Main Memory: size, bandwidth, affinity
Network: topology, bandwidth, latency

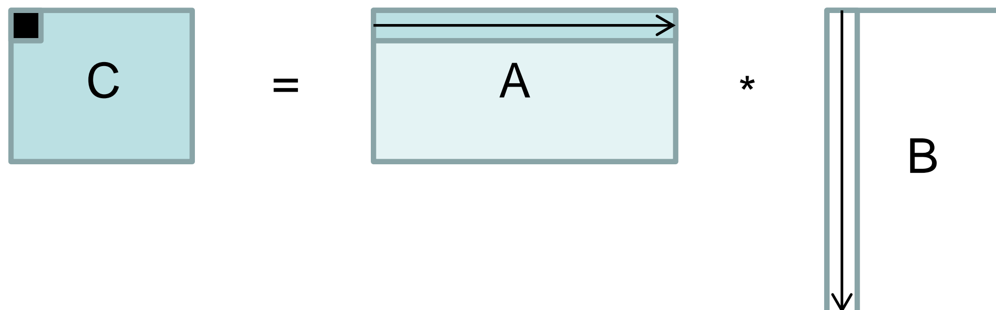
objectives

Maximize FP throughput
Maximize cache re-use
Minimize memory access
Minimize communication
Maximize FP/communication overlap

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
```

Serial – textbook algorithm



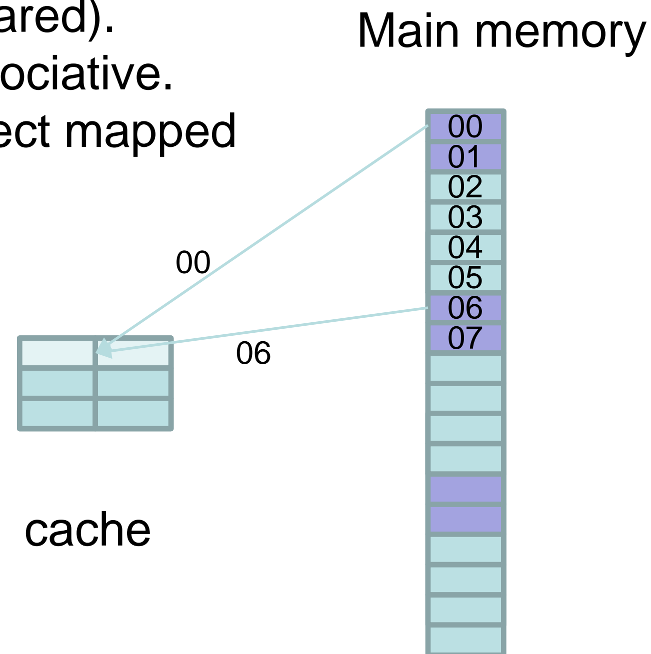
OpenMP parallelization

```
!$omp parallel do default(none) &
!$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
```

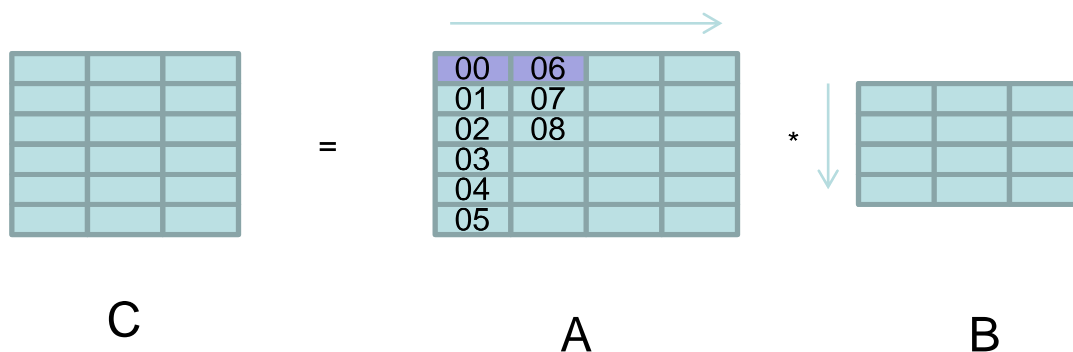
Not really efficient

The role of cache

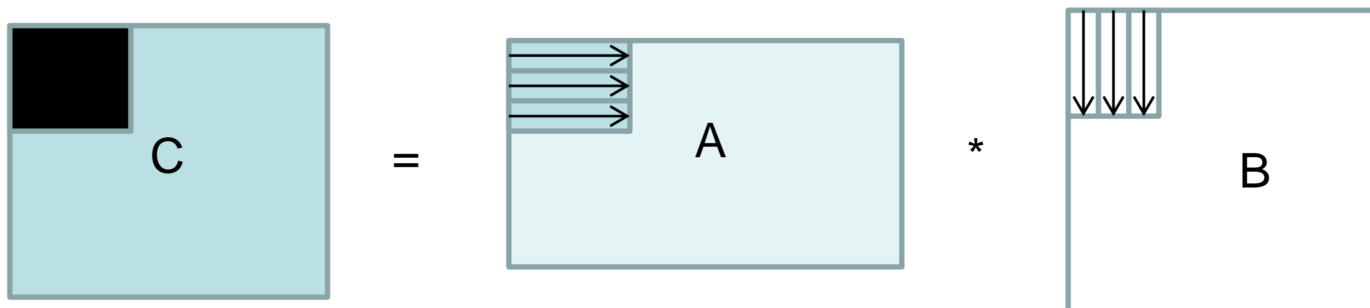
BGQ: 32MByte, 16slices (shared).
Slice: 2MByte 16way set associative.
Way: 128K, 128byte line, direct mapped



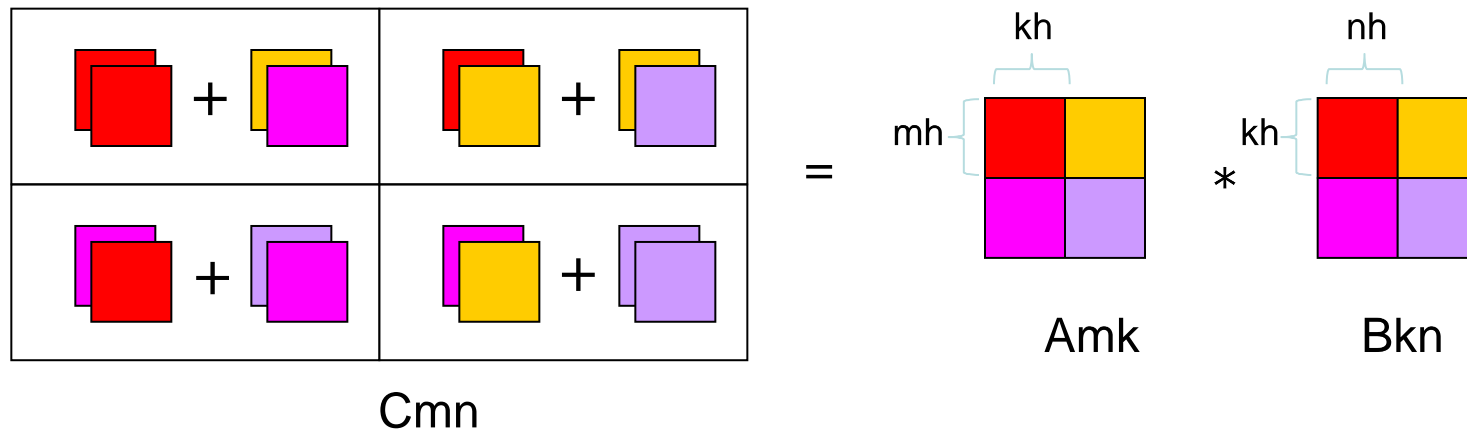
Cache trashing



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

Loops over Matrix bloks

Loops inside Matrix block

```
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
```

```
 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
```

```
 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
```

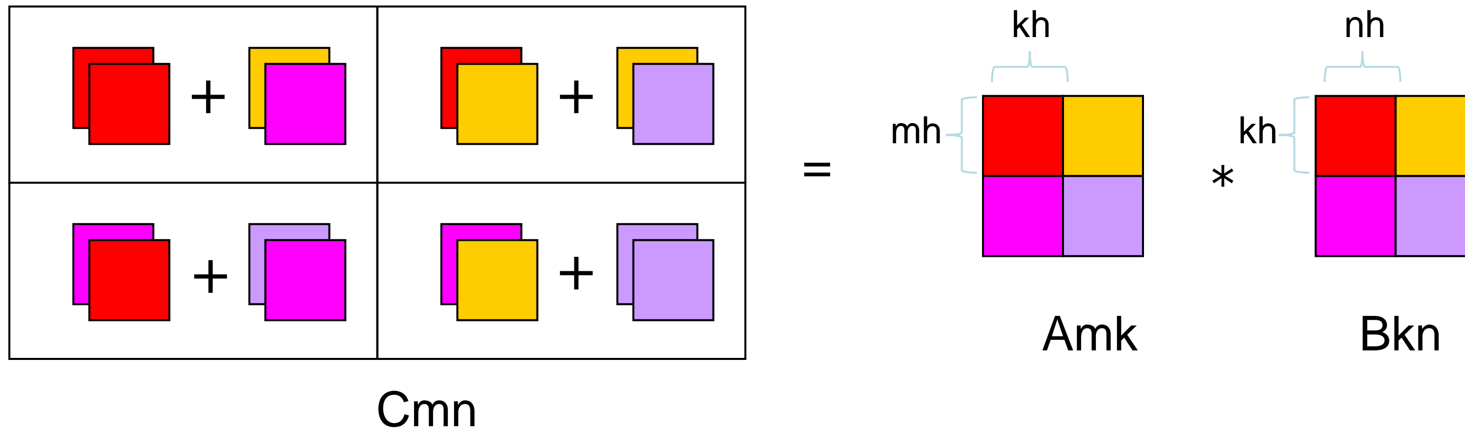
MPI parallelization

Matrix connot be stored in single node memory.

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

Blocks again!

Assign blocks to tasks



m, k, n: matrixes sizes

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

mb, kb, nb: number of blocks

I need to minimize communications

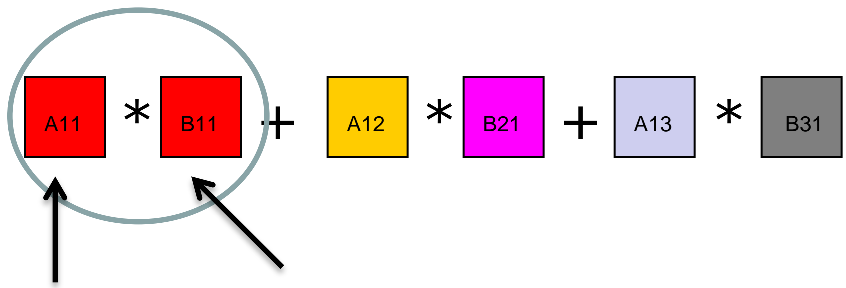
In details

1) Distribute processors on a 2D mesh (2D-Torus is good as well)
2) Processor grid has dimension PxQ
3) Each processor has his coordinates (p,q)
4) Partition the matrixes into PxQ blocks
5) Distribute blocks to processors
6) Perform block by block operations, local to each processor
7) Communicate blocks between processors

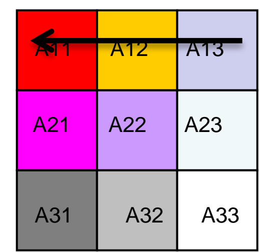
Cannon's algorithm

Consider 3x3 processor grid

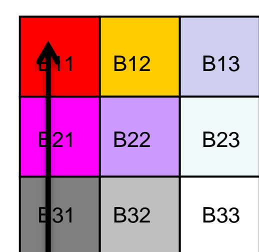
Block C11
Processor (0,0)



=



*



Already on
Proc (1,0)

Already on
Proc (1,0)

Amk

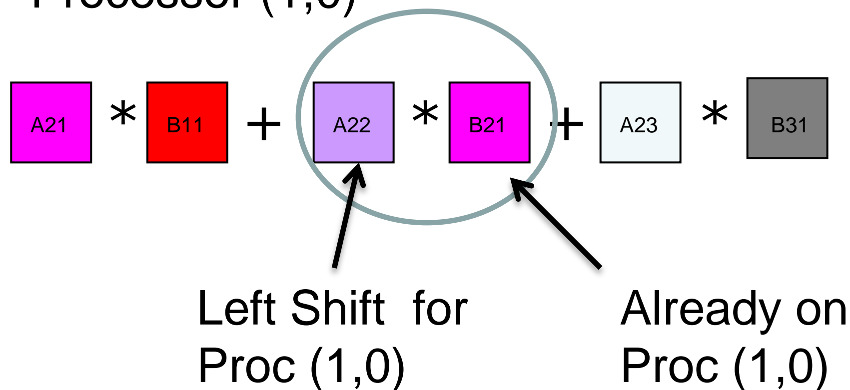
Bkn

Cannon’s algorithm

Consider 3x3 processor grid

Block C21

Processor (1,0)



=

A11	A12	A13
A21	A22	A23
A31	A32	A33

Amk

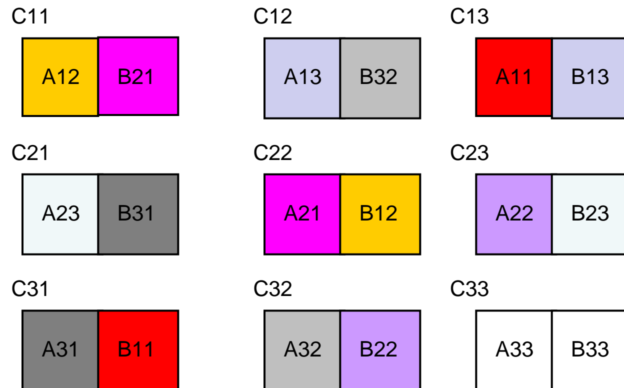
*

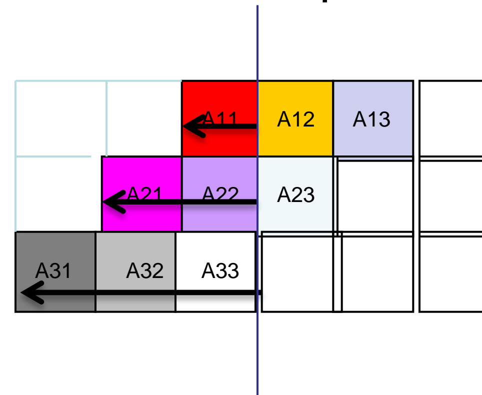
B11	B12	B13
B21	B22	B23
B31	B32	B33

Bkn

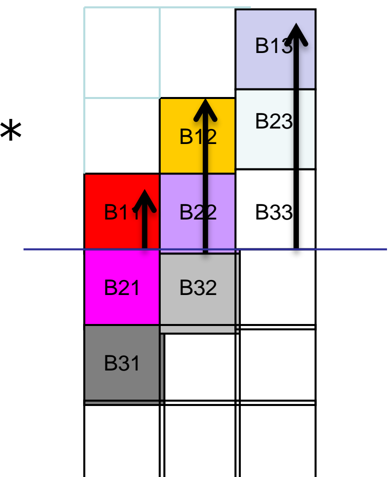
Cannon’s algorithm

First Step:
shift left A blocks of rowid+1
shift Up B blocks of colid+1
Each proc (p,q) performs a local MatrixMatrix multiplication





*

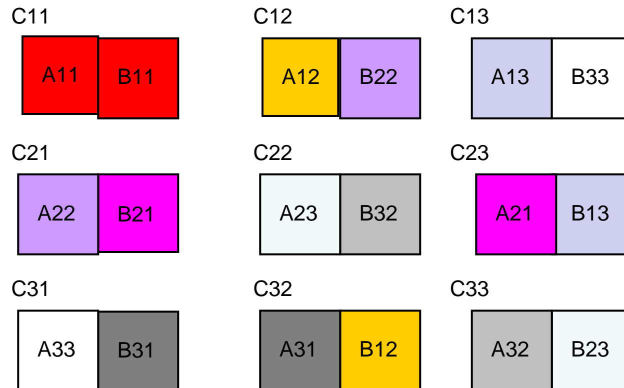


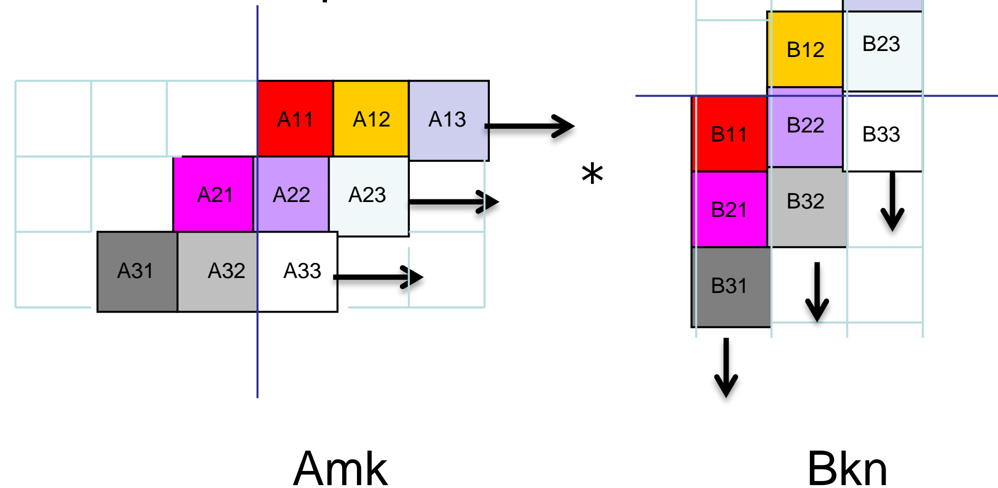
Amk

Bkn

Cannon's algorithm

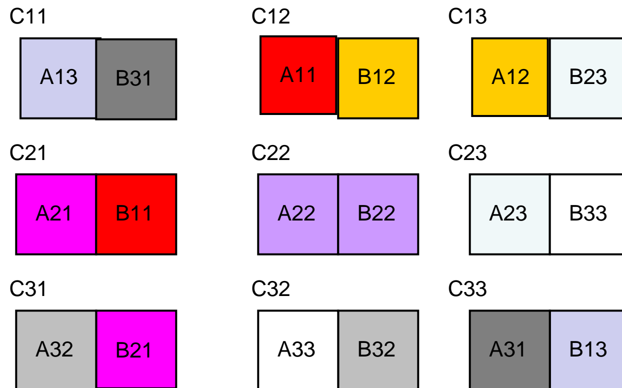
Second Step:
shift right A blocks of 1
shift down B blocks of 1
Each proc (p,q) performs a local MatrixMatrix multiplication

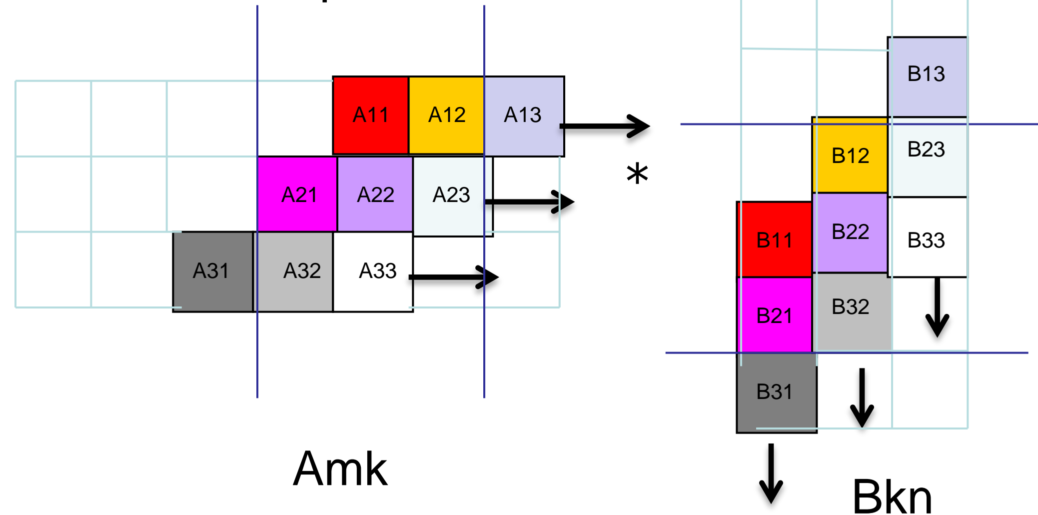




Cannon’s algorithm

Third Step:
shift right A blocks of 1
shift down B blocks of 1
Each proc (p,q) performs a local MatrixMatrix multiplication





```
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

```
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 )
```