OpenMP

A.Klypin

- Shared memory and OpenMP
- Simple Example
- Threads
- Dependencies
- Directives
- Handling Common blocks
- Synchronization
- Improving load balance with Dynamic schedule
- Data placement

Simple Example

Variables and common blocks can be either *shared* or *private*

Each thread (processor) has its own copy of *private* variables.

Shared variables are available for each thread and each thread can change them

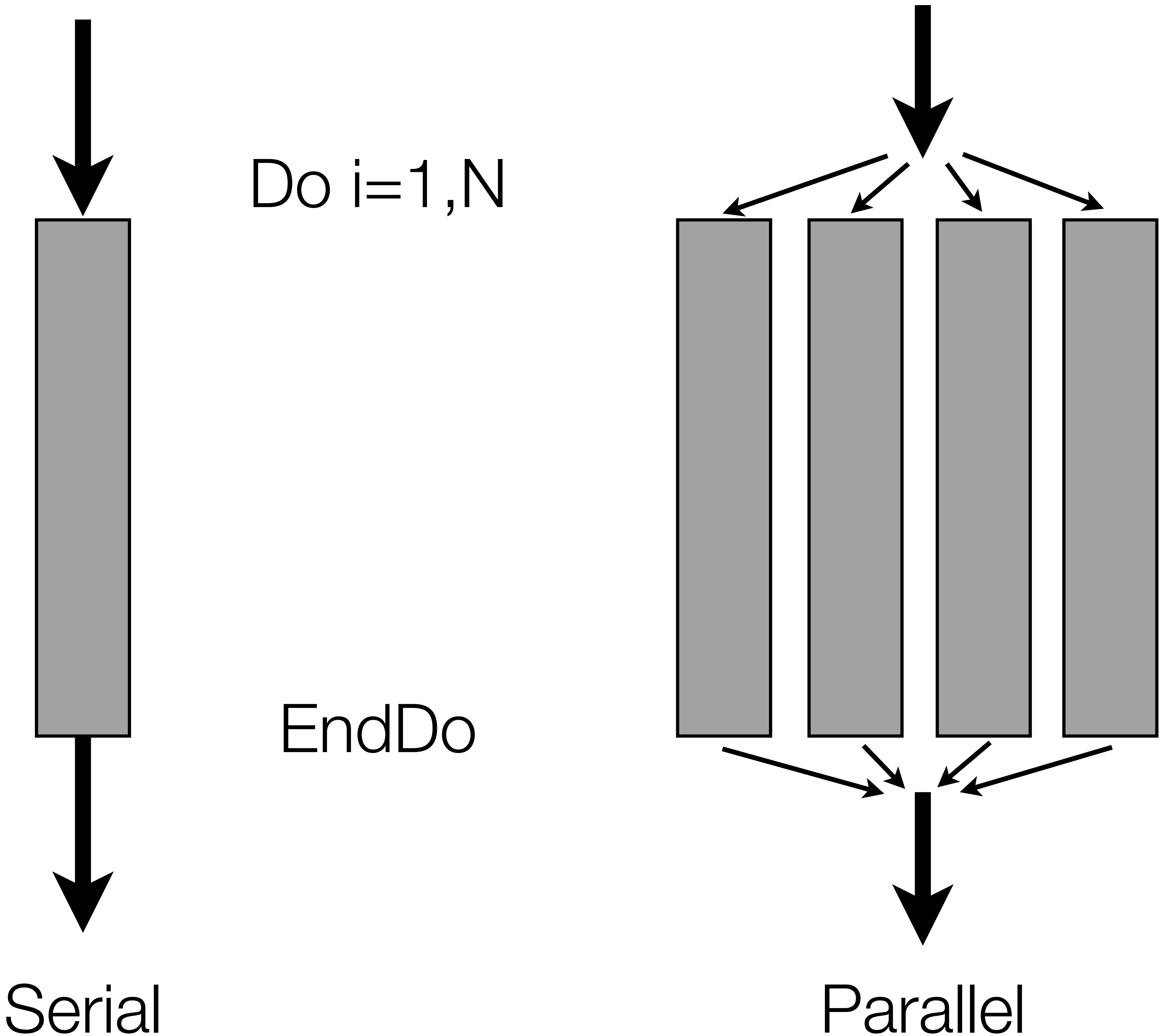
The most frequent bug: missed private variables

```
c-----------------------------------------------------------------
c              Test OMP parallelization
c-----------------------------------------------------------------
      parameter (N = 280 )
      COMMON /aa/ A(N,N,N)
      real*8    sum,sum1

      write (*,*)  ' Required Memory=',8.*N**3/1024.**2,'Mb'
      sum =0.
      rN    = N**2
C$OMP PARALLEL DO DEFAULT(SHARED)
C$OMP+PRIVATE (i,j,k)
C$OMP+REDUCTION(+:sum)
      Do i=1,N
      Do j=1,N
        Do k=1,N
          A(i,k,j) = exp(sin( (i**2+j**2+k**2)/rN))
          sum =sum + A(i,j,k)**2
        EndDo
      EndDo
      EndDo
```

OpenMP and shared memory computers

- Programming with MPI is very difficult. We take some part of job, which should be handled by the system.
- It is easy to use OpenMP.
- Incremental parallelization
- Relatively easy to get speed up on modest sized computers
- Scalable to large processor counts: Altrix
- Multi core processors: you may have it
- Industry moves in the direction of shared memory systems.
- hybrid MPI+OpenMP minimizes communications

Formation of threads: master thread spawns a team of threads



Dependencies

In order for a loop to be parallelizable, its results should not depend on the order of its execution. For example, the following loop is not parallel:

```
do i=2,N
  a(i) =i*a(i-1)
enddo
```

Yet, this loop can be parallelized:

```
do i=2,N,2
  a(i) =i*a(i-1)
enddo
```

If results of execution change with the order of execution, then it is said that we have *race conditions*.

Dependencies

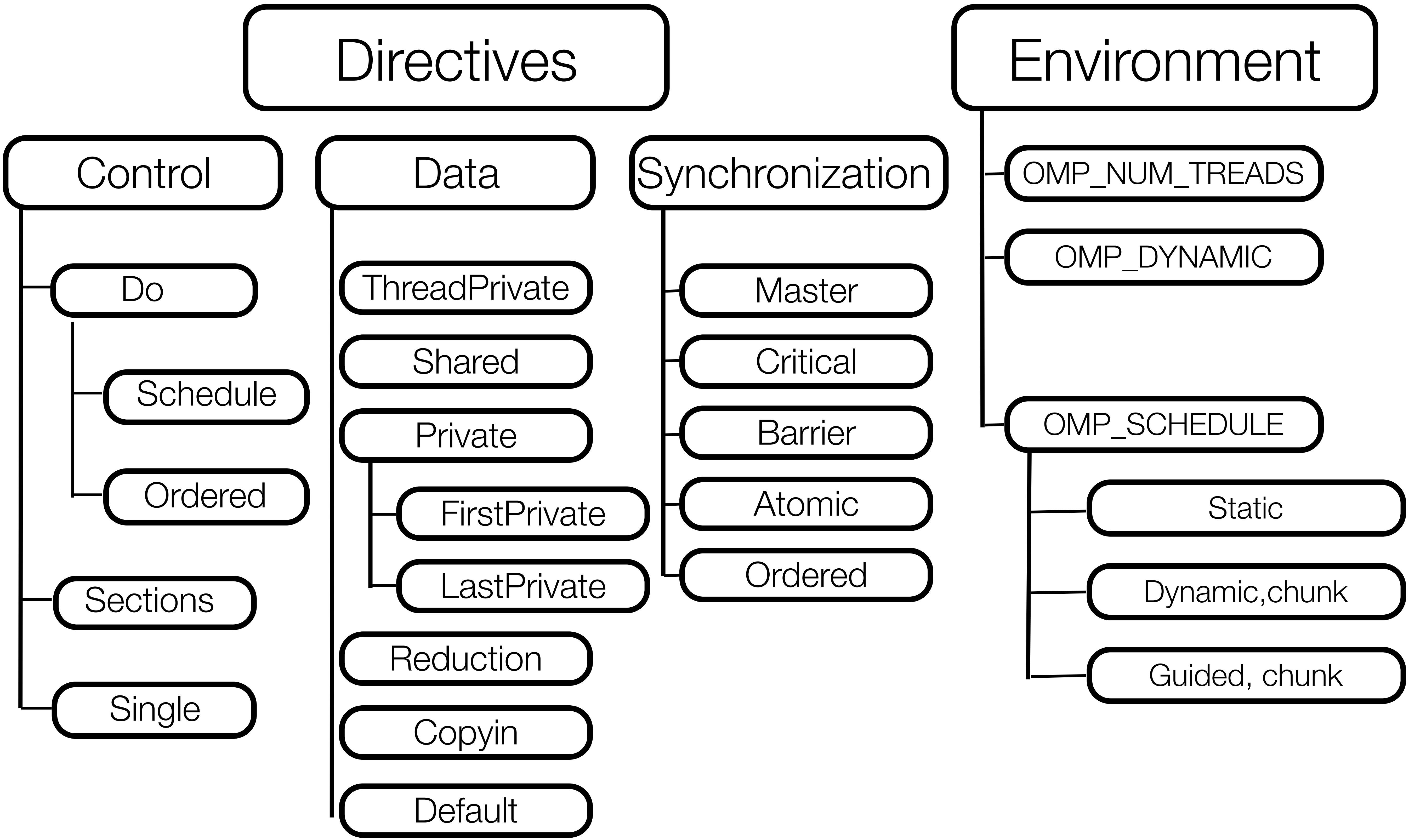
Different ways of removing race conditions:

- rewrite the algorithm
- split loop into two: one, which does not have dependencies (and makes most of computations) and another, which handles the dependancies
- introduce new arrays, which store results for each processor.

Example: assignment of density

```
C.... Open_MP
C$OMP PARALLEL DO DEFAULT(SHARED), SCHEDULE(DYNAMIC)
C$OMP+PRIVATE ( iproc,icount, ....)
          Do iproc =1,LProc
             icount = 0
             ......
             CALL Dens_Child(iproc,icount)    ! collect contributions
                                              ! store thm in array rBuffer(.,iproc)
            lnBuffer(iproc) =icount           ! number of contributions by processor iproc

          Enddo        ! end iproc

          Do iproc =1,LProc          ! non-parallel part of the algprithm
             Do iB =1,lnBuffer(iproc)
                iAcc =iBuffer(iB,iproc)
                ref(iAcc)   = ref(iAcc) + rBuffer(iB,iproc)
             enddo
          enddo
        enddo                 ! end iChunk
```

OpenMP constructs



Common blocks: *ThreadPrivate*

- Determine which commons are private and declare them in each subroutine.
- Use COPYIN(*list*) directive to assign the same values to threadprivate common blocks. List may contain names of common blocks and names of variables
- Be careful with large common blocks: you may run out of memory

```
SUBROUTINE Mine
Common /A/ x,y,z
Common /B/v,u,w
!$omp threadprivate(/A/,/B/)
!$omp  parallel do default (shared)
!omp& copyin(/A/,v,u)
      ....
End
SUBROUTINE MineTwo
!$omp threadprivate(/A/,/B/)
   Common /A/ x,y,z
   Common /B/v,u,w
      ....
End
```

Schedule: handling load balance

- Normally every thread receives equal amount of indexes to work on. For example, if you have 10 threads and the loop is *do i=1,10000*, then the first thread gets indexes (1-1000), the second (1001-2000), and so on. This works ok if there is equal amount of computations for each chunk of indexes. If this is not the case, we need to use *DYNAMIC* option in *SCHEDULE* clause.

- *DYNAMIC* has a parameter, *chunk*, which defines the number of indexes assigned the each thread. The first thread to finish its job takes the next available chunk. Parameter *chunk* is a variable. It can be assigned inside the code.

```
SUBROUTINE Mine(N)
        ...
  Nchunk =N/100
!$omp parallel do default (shared)
!$omp+private(i)
!$omp+schedule(dynamic,Nchunk)
    Do i=1,N
        ...
     EndDo
```

Example: find maxima of density

assign threads dynamically

This condition makes cpu very different for different ‘i’

```
C$OMP PARALLEL DO DEFAULT(SHARED)
C$OMP+PRIVATE (i,Xc,Yc,Zc,a0,in,Radius)
C$OMP+PRIVATE (Xnew,Ynew,Znew,Amnew,Dr)
C$OMP+SCHEDULE(DYNAMIC,5000)
      Do i=1,Ncentr
         Xc =Xm(i)
         Yc =Ym(i)
         Zc =Zm(i)
         a0 =Amc(i)
         in =Lab(i)
         Radius =Rmc(i)
         If(Radius.lt.0.999*RadSR2)in =-Niter
         If(in.gt.0)Then         ! If not converged, keep iterating
            Call Neib(Xnew,Ynew,Znew,Amnew,Xc,Yc,Zc,Radius) ! new center of mass
            Dr =MAX(ABS(Xnew-Xc),ABS(Ynew-Yc),ABS(Znew-Zc))
            IF(Dr.LT.Riter.or.Amnew.lt.Amc(i))
     .            Lab(i)=-Niter     ! iterations converged
                  IF(Xnew.lt.0.)Xnew =Xnew+Box
                  IF(Ynew.lt.0.)Ynew =Ynew+Box
                  IF(Znew.lt.0.)Znew =Znew+Box
                  IF(Xnew.gt.Box)Xnew =Xnew-Box
                  IF(Ynew.gt.Box)Ynew =Ynew-Box
                  IF(Znew.gt.Box)Znew =Znew-Box
            Xm(i)  = Xnew
            Ym(i)  = Ynew
            Zm(i)  = Znew
            Amc(i) = Amnew
         EndIf
      EndDo
```

Synchronization

- *Critical* section defines section of the code, which is executed only by one thread at a time. It may dramatically slow down the code. If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.
- *Critical* section can be used to
  - ✳Sum up private contributions into shared variables
  - ✳Make I/O contiguous

```
SUBROUTINE Mine
      ...
!$omp parallel do default (shared)
        ...
!$omp critical
    Global(i,j,k) = Global(i,j,k) +dx
!$omp end critical
!$omp critical
    write(*,*) ‘ I’m here:’,i
!$omp end critical
```

Synchronization

- The BARRIER directive synchronizes all threads in the team.
- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

- The ATOMIC directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-CRITICAL section.

```
SUBROUTINE Mine
      ...
!$omp parallel do default (shared)
       ...
!$omp critical
    Global(i,j,k) = Global(i,j,k) +dx
!$omp end critical
!$omp critical
    write(*,*) ' I'm here:',i
!$omp end critical
```

REDUCTION clause

- The REDUCTION clause performs a reduction on the variables that appear in its list. A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.
- This is the way to get constructs such as scalar products or to find maximum of elements in an array.

- **`REDUCTION`** ***`(operator|intrinsic: list)`***

- Operators: +,*,-,Max,Min,IAND,IOR,AND,OR
- Examples:
  !$omp do reduction(+:x,y) reduction(max:xmax,ymax)

- **Environmental variables:**

OMP_NUM_THREADS

Sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

OMP_SCHEDULE

Applies only to DO, PARALLEL DO (Fortran) and `for, parallel for` (C/C++) directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors. For example:

```
setenv OMP_SCHEDULE "guided, 4"
setenv OMP_SCHEDULE "dynamic"
```

Memory allocation and access

- Shared memory is not always really the true shared memory. On dual and quad systems the memory is on the same board as the processors. As the result, the memory access is relatively fast.

- On large many-processors systems memory access is much more complicated and, as the result, it is typically much more slower. Memory access is fast if a processor requests memory, which is local (on the same board). The further the memory is from the processor, the larger is the cost of accessing it. Formally we have shared memory, but if we are not very careful, there will be no speed up.

- Improving locality is the main goal.

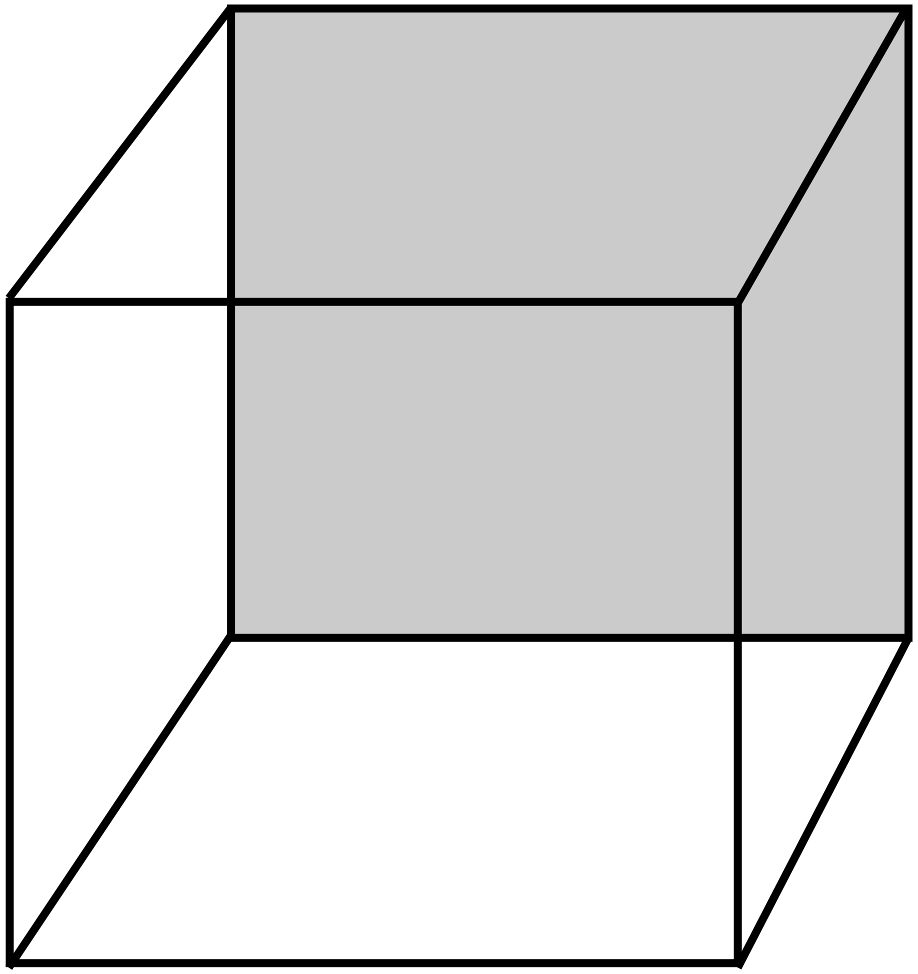
Improving data locality

- **Cash misses, TLB misses:** data are retrieved from memory in blocks, which size depends on particular system. Data should be organized in such a way that cash is reused many times.

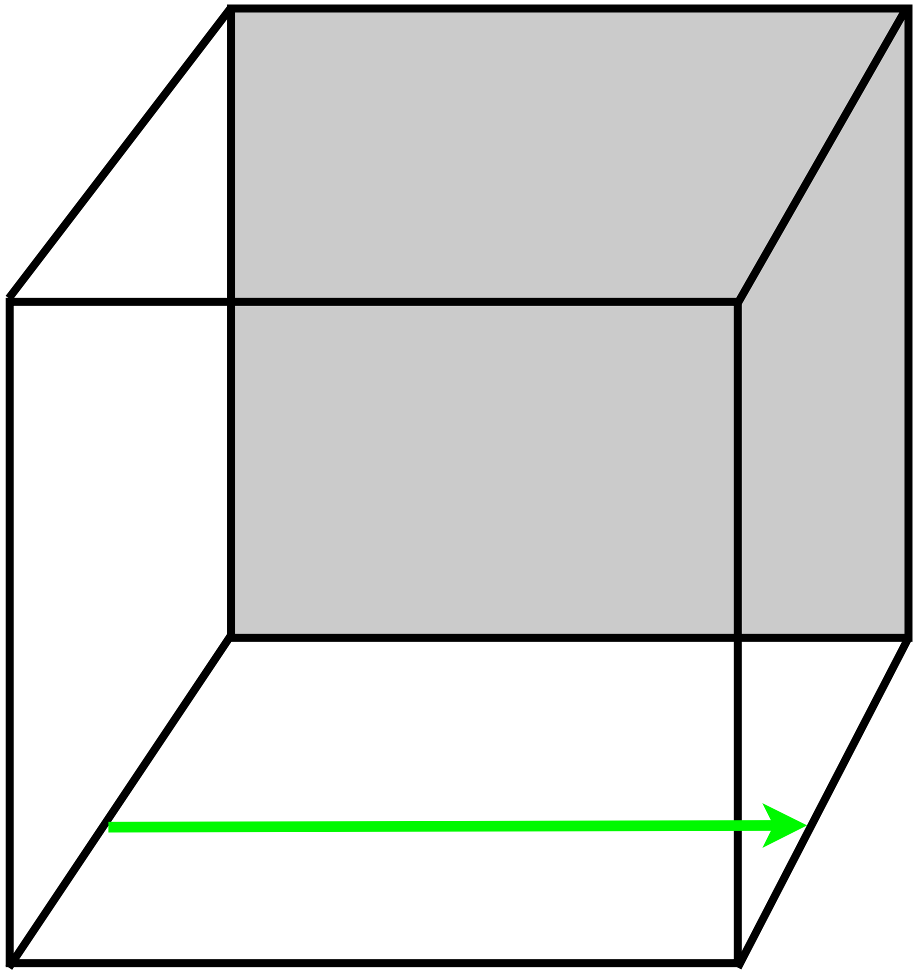
- Re-odering or sorting

- Place data in local arrays

Re-ordering data: 3d FFT, pass in z direction

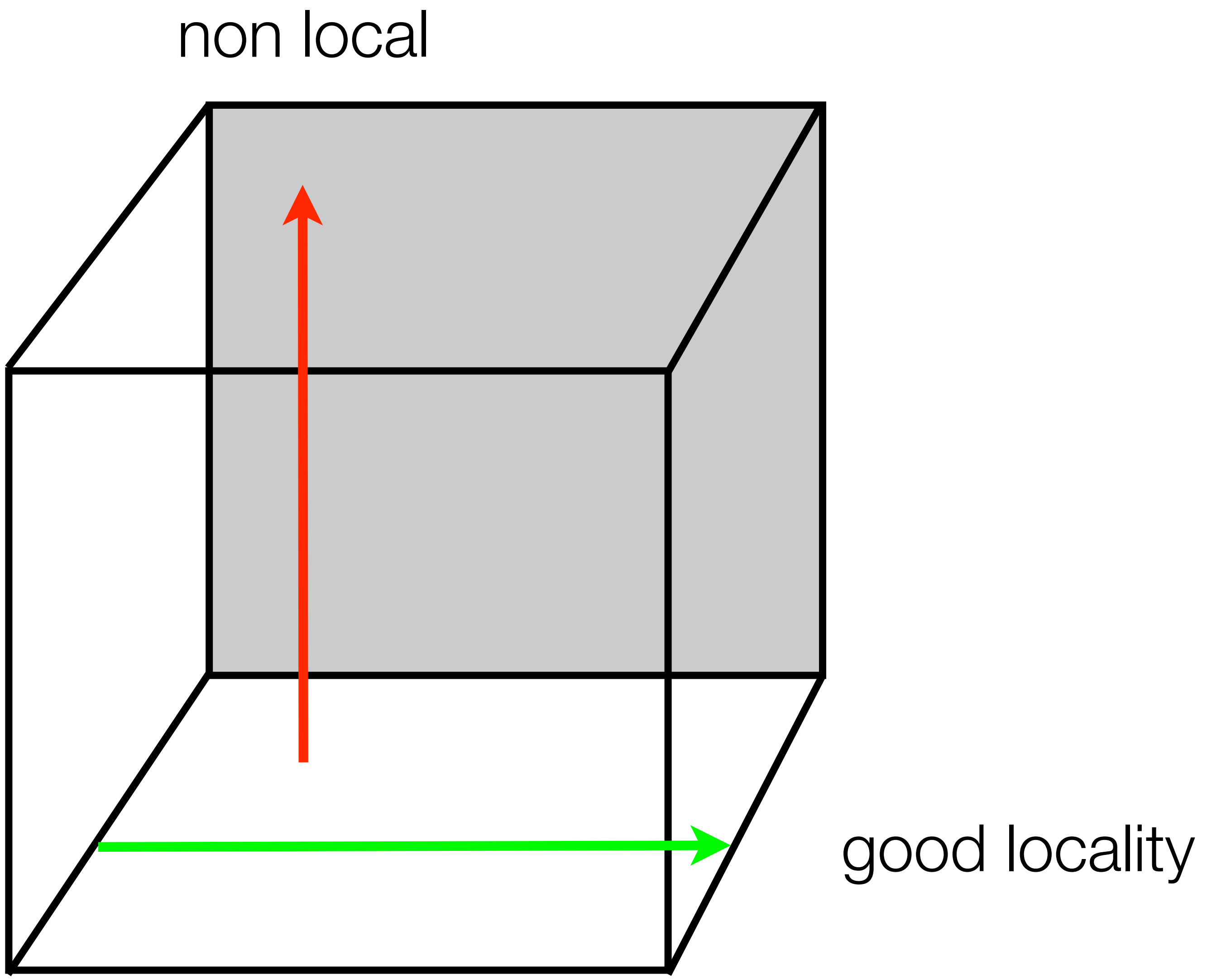


Re-ordering data: 3d FFT, pass in z direction



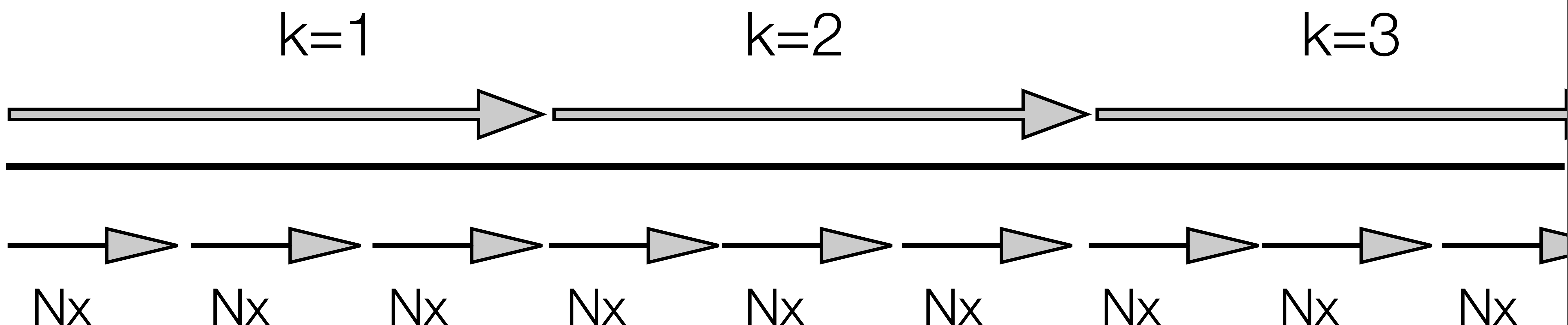
good locality

Re-ordering data: 3d FFT, pass in z direction



Mapping multi dimensional array into 1d memory

- A(Nx,Ny,Nz) : A(i +(j-1)*Nx +(k-1)*Nx*Ny)



Transposition of matrix

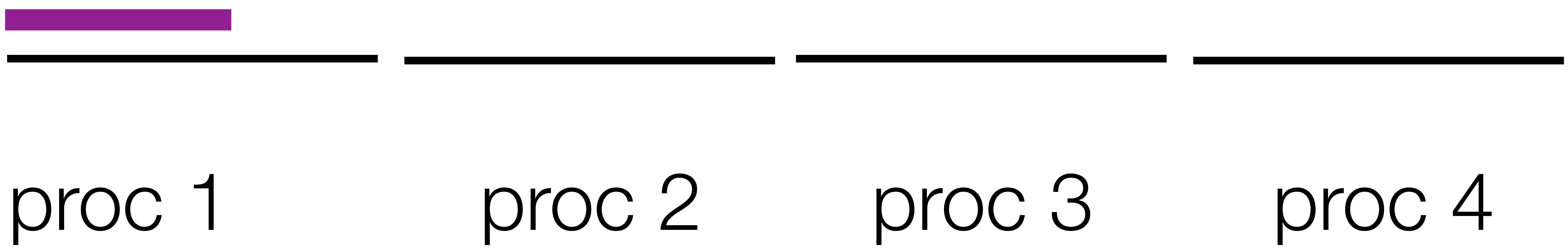
A(i,j,k) → A(k,j,i)

Now do FFT along x

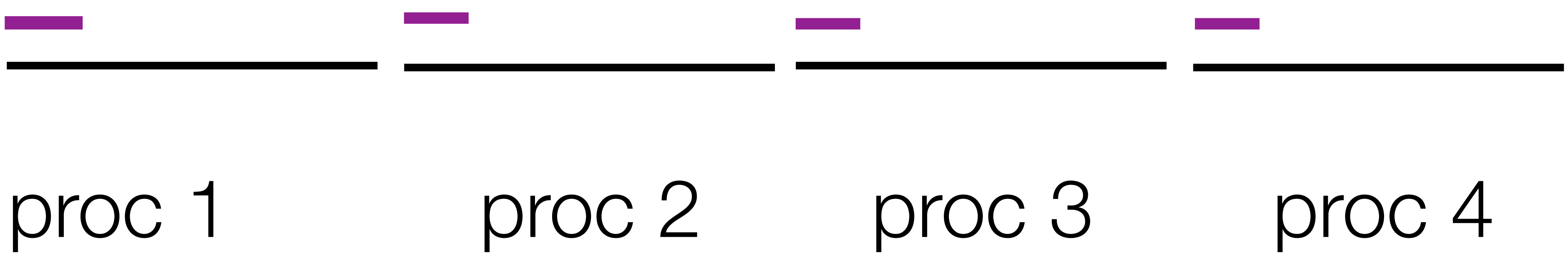
```
C.... Open_MP
C$OMP PARALLEL DO DEFAULT(SHARED)
C$OMP+PRIVATE ( k,j,i,aa)
      DO J=1,NGRID
      DO K=1,NGRID-1
         DO I=K+1,NGRID
           aa = FI(I,J,K)
           FI(I,J,K) =FI(K,J,I)
           FI(K,J,I) =aa
         ENDDO
        ENDDO
      ENDDO

      write (*,*) '      ....z'
C.... Open_MP
C$OMP PARALLEL DO DEFAULT(SHARED)
C$OMP+PRIVATE ( k,j,i,zf,yf,ip,isl,l1,n2,n4 )
      DO K=1,NGRID
        DO J=1,NGRID
         DO I=1,NGRID
           Zf(I) =FI(I,J,K)
         ENDDO
         call four67(ib1,iq,ip,isl,l1,n2,n7,si,indx,
     &                     Zf,Yf)
         DO I=1,NGRID
             FI(I,J,K) =  Yf(I)**2/FT
         ENDDO
        ENDDO
      ENDDO
```

Memory distribution: multi processor system

- How an array A(N) is allocated?



Or this way:



First touch rule

- The way how an array is accessed the first time in the code defines how the array is distributed: on one processor (for serial access) or on many processors (for parallel access)
- Note in that in HDF (now almost extinct) one can decide how to allocate an array.
- On Altrix a parallel distribution seems to be a default for common blocks.
- To improve locality, instead of

  ```
  COMMON/DATA/X(N),Y(N),Z(N)
  ```

  write:

  ```
  COMMON/D1/X(N)
  COMMON/D2/Y(N)
  COMMON/D3/Z(N)
  ```

Memory access on multi processors shared memory computers

- Normally we do not parallelize simple loops such as

```
Do i=1,N
   s=s +a(i)
EndDo
```

- On large computers every access of non-local memory is so expensive that every effort should be made to parallelize every loop.

- Once everything is done, codes can be very efficient. Halo finder, which before was taking 15hrs, now works for 15min with 24 procs.