• Shared memory and OpenMP
• Simple Example
• Threads
• Dependencies
• Directives
• Handling Common blocks
• Synchronization
• Improving load balance with Dynamic schedule
• Data placement
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
- Multi core processors: we have it
- Industry moves in the direction of shared memory systems.
- Hybrid MPI+OpenMP minimizes communications
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
--- Test OMP parallelization ---

parameter (N = 280 )
COMMON /aaa/ 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
```
Formation of threads: master thread spawns a team of threads

Serial

Parallel

Do i=1,N

EndDo
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:

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

Yet, this loop can be parallelized:

```plaintext
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*. 
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 dependencies
- introduce new arrays, which store results for each processor.
Example: assignment of density

```c
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)
      InBuffer(iproc) =icount ! number of contributions by processor iproc
   Enddo ! end iproc

   Do iproc =1,LProc ! non-parallel part of the algorithm
      Do iB =1,InBuffer(iproc)
         iAcc =iBuffer(iB,iproc)
         ref(iAcc) = ref(iAcc) + rBuffer(iB,iproc)
      enddo
   enddo
Enddo ! end iChunk
```
<table>
<thead>
<tr>
<th>Clause</th>
<th>PARALLEL</th>
<th>DO/for</th>
<th>SECTIONS</th>
<th>SINGLE</th>
<th>PARALLEL DO/for</th>
<th>PARALLEL SECTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIVATE</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRSTPRIVATE</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASTPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REDUCTION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPYIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCHEDULE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDERED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOWAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
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 \( \text{do } i=1,10000 \), 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 \textit{DYNAMIC} option in \textit{SCHEDULE} clause.

- \textit{DYNAMIC} has a parameter, \textit{chunk}, which defines the number of indexes assigned to each thread. The first thread to finish its job takes the next available chunk. Parameter \textit{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
  ...

**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

```fortran
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

    !$omp critical
    !$omp critical
        write(*,*) ' I’m here:’,i
    !$omp end critical
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"
```
Re-ordering data: 3d FFT, pass in z direction

non local

good locality
Mapping multi dimensional array into 1d memory

- \( A(Nx,Ny,Nz) : A(i +(j-1)^*Nx +(k-1)^*Nx*Ny) \)
Transposition of matrix

\[ A(i,j,k) \rightarrow A(k,j,i) \]

Now do FFT along \( x \)