Loop Level Parallelism in Depth

```
!$omp parallel do [clause [,] [clause…]]
   do index = first, last [, stride]
      body of loop
   enddo
[!$omp end parallel do]
```

```
#pragma omp parallel for [clause [clause…..]]
  for (index = first; test_expr ; increment_expr ) {
    body of the loop
   }
```

Restrictions on Loop Parallelism

- Compiler needs to determine trip count i.e. # of times loop executed without actually executing the loop
- In Fortran **parallel do** must be followed by a do loop
- In C the statement following **parallel for** pragma must be a for loop
- Program must complete all iterations of loop – cannot exit loop before all iterations are complete – loop must be structured block
- Other OpenPM directive that require structured blocks are :
  - parallel
  - sections
  - single
  - master
  - critical
  - ordered

Nested Loop Parallelization

- Nested loop 1 :

```
!$omp parallel do
        do j = 1, n
            a(0,j) = 0
            do i = 1, m
                a(0,j) = a(0,j) + a(i,j)
            enddo
        enddo
        end
```

- Nested loop 2 :

```
        do j = 1, n
!$omp parallel do
            do i = 1,m
                a(i,j) = (a(i-1,j-1)+a(i,j-1)+a(i+1,j-1) )/3.0
            enddo
        enddo
        end
```

Shared Variables

- Shared variables are shared by all threads in the team for the duration of the parallel construct
- All the threads access the single instance of the variable in the shared memory
- All modification of shared variable is available to all threads

Private variables

- **private** clause requires that each thread create a private instance of the variable
- **private** variables get new storage for the duration of the parallel construct
  - They are undefined and uninitialized upon entry to the parallel region
  - They are also undefined upon exit of parallel region
- Private pointers
  - private clause makes the pointer variable private but says nothing about what it points to
  - It is undefined just like regular private variables upon entry and exit of a parallel construct

Private Variables

- By default in Fortran the loop index of a sequential nested loop is private but not in C/C++ (needs to be explicitly scoped)
- When a subroutine is called from within a parallel region, the local variables within the called subroutine are private to each thread
  - Fortran : if these variables are declared with the **save** attribute then they are shared
  - C/C++ : if these variables are declared **static** then they are shared

Changing Default Scoping

- Fortran :
  - default (shared | private | none)
  - One of the above three can be chosen as default (if nothing is chosen the default is shared in Fortran)
  - default (private) can be used to convert MPI code to OpenMP
  - default (none) can sometimes help catch scoping error
- C/C++:
  - default(shared | none)
  - No default (private) in C/C++ because many C standard libs are written using macros that reference global variables and scoping these as private can be incorrect

Reduction Operations

- In a reduction we repeatedly apply a binary operator to a variable and some other value and store the result back in the variable,
- Example 1 :

```
                sum = 0
!$omp           parallel do reduction(+ : sum)
                do j = 1, n
                        sum = sum + a(j)
                enddo
```

- Example 2 :

```
                x = a(1)
                do j = 2, n
                        x = max(x, a(j))
                enddo
```

Reduction Operations

- Reduction Operations :
  - At runtime each thread performs a portion of the additions that make up the final sum
  - At the end of the parallel loop, the threads combine their partial sum into a final sum
- To understand operation of reduction clause examine an OpenMP code that performs the same computation in parallel without using the reduction clause itself
- Check Fortran and C standard (provided in class webpage) for allowable operators

Private Variable Initialization and Finalization

- Default :
  - Each threads copy of a private variable on a parallel do/for and parallel region has undefined initial value
  - After the parallel do/for and parallel region the master thread's copy also has undefined value
- Sometimes we need to access (inside the parallel construct) the value that was master's copy just before entering the parallel construct
- Sometime we need to copy the "last" value written to a private variable back to the master's copy outside the parallel construct
  - The "last" value is the value assigned in the last iteration of a sequential execution of the loop

Private Variable Initialization and Finalization

- OpenMP provides **firstprivate** and **lastprivate** clauses as variations of private
- At the start of a parallel do/for and parallel regions, **firstprivate** initializes each thread's copy of a private variable to the value of the master's copy
  - firstprivate is initialized once per thread, rather than once per iteration
- At the end of parallel do/for and parallel regions, **lastprivate** writes back to the master's copy the value from the thread that executed the sequentially last iteration of the loop

- lastprivate example :

```
!$omp  parallel do lastprivate(temp)
        do j = 1, 100
                temp = f (j)
        enddo
        print*, ‘f(100) =‘ , temp
```

i.e. temp is equal to f(100) at the end of the loop

- firstprivate example :

```
        j = 2 ;
#pragma omp parallel firstprivate (j)
        for (k = 1 ; k < n ; k = k + 1 ) {
                a[k] = j*b[k] ; }
```

Scheduling Loops

- Loop Schedule : manner in which iterations are assigned to threads is called loop scheduling
- Default schedule allows each thread to execute same # of iterations as any other thread
- Balanced : if all the iterations do about same amount of work and take about the same time

- Imbalanced : if different iterations do different amount of work and hence take different amount of time
  - Amount of work per iteration can vary linearly (either go up or down) with iteration number
  - Amount of iterations can vary randomly with iteration number
- If default schedule is used faster threads may wait for slower threads to finish
  - Leads to load imbalance
  - Leads to increase in total execution time

- !$omp parallel do

```
do j = 1, n
        call work(j)
end do
```

Subroutine work may take more time with increasing j

- !$omp parallel do private(x)

```
do j = 1,n
  x = function(j)
  if (x .lt. 100) then
        call work100()
  else
        call work1000()
  endif
enddo
```

schedule clause

- A **schedule** is specified with parallel do/for directive
- Different options available for scheduling
  - Correctness of your code should not depend on scheduling option you choose
  - Only performance should depend on scheduling option chosen
- Syntax : **schedule(type [, chunk])**
- **type** can be:
  - **static**
  - **dynamic**
  - **guided**
  - **runtime**

The division of work among CPUs can be controlled with the SCHEDULE clause. For example

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
```
Iterations divided among the CPUs in contiguous chunks

```
!$OMP PARALLEL DO SCHEDULE(STATIC, N)
```
Iterations divided round-robin fashion in chunks of size N

```
!$OMP PARALLEL DO SCHEDULE(DYNAMIC,N)
```
Iterations handed out in chunks of size N as CPUs become available

static

- For **static** and without **chunk** specified each thread is statically assigned one chunk of iterations
  - The default chunks will be equal or nearly equal in size but depends on implementation of vendor company (how leftover iterations are assigned depends on implementation)
- For **static** with **chunk** present iterations are divided into size **chunk** to threads until fewer than chunk remain (how leftover iterations are assigned depends on implementation)
- Chunks are statically assigned to processors in a round-robin fashion ; the first proc gets the first chunk, the second proc gets the second chunk etc.

Example - SCHEDULE(STATIC)

```
CPU0: do i=1,32
    a(i)=b(i)+c(i)
   enddo

CPU1: do i=33,64
    a(i)=b(i)+ c(i)
   enddo

CPU2: do i=65,96
    a(i)=b(i)+c(i)
   enddo

CPU3: do i=97,128
    a(i)=b(i)+c(i)
   enddo
```

Example - SCHEDULE(STATIC,16)

```
CPU0: do i=1,16
       a(i)=b(i)+c(i)
      enddo
      do i=65,80
       a(i)=b(i)+c(i)
      enddo
CPU1: do i=17,32
       a(i)=b(i)+c(i)
      enddo
      do i=81,96
       a(i)=b(i)+c(i)
      enddo
```

```
CPU2: do i=33, 48
       a(i)=b(i)+c(i)
      enddo
      do i=97,112
       a(i)=b(i)+c(i)
      enddo
CPU3: do i=49,64
       a(i)=b(i)+c(i)
      enddo
      do i=113,128
       a(i)=b(i)+c(i)
      enddo
```

dynamic

- For **dynamic** iterations are divided into chunks of size **chunk** (if chunk is not present, the size of all chunks is 1)
- At runtime, chunks are assigned to threads dynamically
- As threads become available they pick up chunks
- Useful for **load balance** if different chunks do different amount of work

guided

- The first chunk size is implementation dependent
- Size of successive chunks decreases exponentially down to minimum size of chunk
- If chunk is not specified minimum chunk size is one
- A guided schedule typically picks an initial chunk size K0 of N/P (N = total iterations, P = # of threads)
  Then picks successive chunk sizes using :

$$K_i = (1 - 1/P)K_{i-1}$$

- guided produces fewer chunk than dynamic and hence less synchronization cost

runtime

- The schedule type is chosen at runtime based on the environment variable **omp_schedule**
- No chunk size allowed

Advise on scheduling options

After you write a correct parallel code you may experiment with different scheduling options to see which gives best performance

Summary of schedule clause

The division of work among CPUs can be controlled with the SCHEDULE clause. For example

- `!$OMP PARALLEL DO SCHEDULE(STATIC)`

Iterations divided among the CPUs in contiguous chunks

- `!$OMP PARALLEL DO SCHEDULE(STATIC, N)`

Iterations divided round-robin fashion in chunks of size N

- `!$OMP PARALLEL DO SCHEDULE(DYNAMIC,N)`

Iterations handed out in chunks of size N as CPUs become available

- `!$OMP PARALLEL DO SCHEDULE(GUIDED,N)`

Iterations handed out in chunks of exponentially decreasing sizes

Parallel Region

- Loop level parallelism is fine grained parallelism where unit of work done in parallel is small
- Next we look into coarse grained parallelism with OpenMP
  - Coarse grained parallelism allows SPMD type programming (which is used in MPI)

Parallel Region Clause

Fortran :

```
!$OMP parallel [clause…..]
      structured block of code
!$OMP end parallel
```

C/C++ :

```
#pragma omp parallel [clause….]
      structured block of code
```

```
real*8 x, y, x1, y1
integer i, j, m, n
integer dist(*,*), av_dist(*,*)
integer function
……..
x1 = 10.
y1 = 20.
do i = 1, m
   do j = 1, n
            x = i / real(m)
            y = j / real(n)
            distance(i, j) = function(x, y, x1, y1)
   enddo
enddo
do i = 1, m
    do j = 1, n
            av_dist(j,i) = 0.33*(dist(j-1,i) + dist(j,i) + dist(j+1,i) )
    enddo
enddo
```

- Using parallel do/for we would parallelize each loop nest individually
  - This requires threads to sync, join and fork at the end of first loop – is this necessary ?
  - Threads can move on to calculate average since average is over j direction only
- OpenMP supports this concept of parallel region using **parallel/end parallel** directives
- Block of code enclosed by the parallel/end parallel is done in parallel by a team of threads initially forked by the parallel directive
- **parallel/end parallel** can enclose any structured block of statements

```
                    ……..
                    x1 = 10.
                    y1 = 20.
!$omp parallel
!$omp& private(i, j, x, y)
!$omp& private(my_part,my_thr, my_i_start, my_i_end)
                    my_part = m/(omp_get_num_threads() )
                    my_thr = omp_get_thread_num()
                    my_i_start = my_part*my_thr +1
                    my_i_end = my_part*(my_thr+1)
                    do i = my_i_start, my_i_end
                      do j = 1, n
                              x = i / real(m)
                              y = j / real(n)
                              dist(i, j) = function(x, y, x1, y1)
                      enddo
                    enddo

                    do i = my_i_start, my_i_end
                      do j = 1, n
                              av_dist(j,i) = 0.33*(dist(j-1,i) + dist(j,i) + dist(j+1,i) )
                      enddo
                    enddo
!$omp end parallel
```

Runtime Model for Parallel Region

- The **parallel/end parallel** forks a team of parallel threads with individual data environments for enclosed code
- The enclosed code is executed concurrently
- Each thread executes the same code asynchronously
- The threads and their data environment disappears at the **end parallel** directive
- We now manage our own loop extents as a part of coarse grained parallelism
- For parallel region concept we decompose the problem in terms of underlying data structures and map to threads

Fine Grained and Coarse Grained Parallelism

- Fine grained or loop level : simple but may have limited scalability and performance

- Coarse grained or parallel region : requires more analysis of code and extra programming (than fine grained) but may provide higher scalability and performance

Parallel Region

The **!$omp parallel** and **#pragma omp parallel** directives can be used to mark entire regions as parallel. The following two examples are equivalent

Fortran:

```
!$OMP PARALLEL DO
do i=1,n
  a(i)=b(i)+c(i)
enddo
!$OMP PARALLEL DO
do i=1,n
  x(i)=y(i)+z(i)
enddo
```

```
!$OMP PARALLEL
!$OMP DO
do i=1,n
  a(i)=b(i)+c(i)
enddo
!$OMP DO
do i=1,n
  x(i)=y(i)+z(i)
enddo
!$OMP END PARALLEL
```

In C :

```
#pragma omp parallel for
    loop1
#pragma omp parallel for
    loop2
```

```
#pragma omp parallel
#pragma omp for
   loop1
#pragma omp for
   loop2
#pragma omp end parallel
```

NOWAIT Clause

When a parallel region is exited, a barrier is implied - all threads must reach the barrier before any can proceed. By using the **NOWAIT** clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
!$OMP PARALLEL
!$OMP DO SCHEDULE(dynamic)
do i=1,n
  call unbalanced_workA(i)
enddo
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(dynamic)
do i=1,n
 call unbalanced_workB(i)
enddo
!$OMP END DO
!$OMP END PARALLEL
```

Work Replication with Parallel Region

```
!$omp parallel
      do j = 1, 10
       print*,'hello',j
      enddo
!$omp end parallel
```

On 5 threads we get

50 print out of hello since each thread executes 10 iterations concurrently with other 10 threads

```
!$omp parallel do
      do j = 1, 10
       print*,'hello',j
      enddo
```

Regardless of # of threads we get 10 print out of hello since do loop iterations are executed in parallel by team of threads