Introduction to OpenMP

Lecture 4: Work sharing directives
Work sharing directives

- Directives which appear inside a parallel region and indicate how work should be shared out between threads
  - Parallel do/for loops
  - Single directive
  - Master directive
  - Sections
  - Workshare
Parallel do loops

• Loops are the most common source of parallelism in most codes. Parallel loop directives are therefore very important!

• A parallel do/for loop divides up the iterations of the loop between threads.

• There is a synchronisation point at the end of the loop: all threads must finish their iterations before any thread can proceed
Parallel do/for loops (cont)

Syntax:

Fortran:

```fortran
!$OMP DO [clauses]
   do loop
[ !$OMP END DO ]
```

C/C++:

```c
#pragma omp for [clauses]
   for loop
```
Parallel do/for loops (cont)

• With no additional clauses, the DO/FOR directive will partition the iterations as equally as possible between the threads.

• However, this is implementation dependent, and there is still some ambiguity:
  
e.g. 7 iterations, 3 threads. Could partition as 3+3+1 or 3+2+2
Restrictions in C/C++

- Because the for loop in C is a general while loop, there are restrictions on the form it can take.

- It has to have determinable trip count - it must be of the form:

  ```
  for (var = a; var logical-op b; incr-exp)
  ```

  where `logical-op` is one of `<`, `<=`, `>`, `>=`
  
  and `incr-exp` is `var = var +/- incr` or semantic equivalents such as `var++`.
  
  Also cannot modify `var` within the loop body.
Parallel do/for loops (cont)

• How can you tell if a loop is parallel or not?
• Useful test: if the loop gives the same answers if it is run in reverse order, then it is almost certainly parallel
• Jumps out of the loop are not permitted.

e.g.

\[
\text{do } i=2, n \\
\quad a(i) = 2 \times a(i-1) \\
\text{end do}
\]
Parallel do/for loops (cont)

- How can you tell if a loop is parallel or not?
- Useful test: if the loop gives the same answers if it is run in reverse order, then it is almost certainly parallel
- Jumps out of the loop are not permitted.

e.g.

```plaintext
do i=2,n
   a(i)=2*a(i-1)
end do
```
2.

\[
ix = \text{base} \\
do \ i=1,n \\
\quad a(ix) = a(ix) \times b(i) \\
\quad ix = ix + \text{stride} \\
\end{do}

3.

\[
do \ i=1,n \\
\quad b(i) = (a(i) - a(i-1)) \times 0.5 \\
\end{do}
2.

\[
\text{ix} = \text{base} \\
\text{do } i=1,n \\
\quad a(\text{ix}) = a(\text{ix})*b(i) \\
\quad \text{ix} = \text{ix} + \text{stride} \\
\text{end do}
\]

3.

\[
\text{do } i=1,n \\
\quad b(i) = (a(i)-a(i-1))*0.5 \\
\text{end do}
\]
2.

\[
\text{ix} = \text{base} \\
\text{do i=1,n} \\
\quad \text{a(ix)} = \text{a(ix)}*\text{b(i)} \\
\quad \text{ix} = \text{ix} + \text{stride} \\
\text{end do}
\]

3.

\[
\text{do i=1,n} \\
\quad \text{b(i)} = (\text{a(i)} - \text{a(i-1)})*0.5 \\
\text{end do}
\]
Parallel do loops (example)

Example:

```c
!$OMP PARALLEL

!$OMP DO

  do i=1,n

    b(i) = (a(i)-a(i-1))*0.5

  end do

!$OMP END DO

!$OMP END PARALLEL
```
Parallel for loops (example)

Example:

```c
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i < n; i++)
    {
        b[i] = (a[i]-a[i-1])*0.5;
    }
} // omp parallel
```
• This construct is so common that there is a shorthand form which combines parallel region and DO/FOR directives:

Fortran:

```fortran
!$OMP PARALLEL DO [clauses]
   do loop
   [$ !$OMP END PARALLEL DO ]
```

C/C++:

```c
#pragma omp parallel for [clauses]
   for loop
```
Clauses

• DO/FOR directive can take PRIVATE, FIRSTPRIVATE and REDUCTION clauses which refer to the scope of the loop.

• Note that the parallel loop index variable is PRIVATE by default
  – other loop indices are private by default in Fortran, but not in C.

• PARALLEL DO/FOR directive can take all clauses available for PARALLEL directive.
The SCHEDULE clause gives a variety of options for specifying which loops iterations are executed by which thread.

Syntax:

**Fortran:** SCHEDULE \((kind[, \text{chunksize}])\)

**C/C++:** `schedule (kind[, chunksize])`

where `kind` is one of **STATIC, DYNAMIC, GUIDED, AUTO or RUNTIME**

and `chunksize` is an integer expression with positive value.

E.g. ```$OMP DO SCHEDULE (DYNAMIC, 4)```
STATIC schedule

• With no chunksize specified, the iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread in order (block schedule).

• If chunksize is specified, the iteration space is divided into chunks, each of chunksize iterations, and the chunks are assigned cyclically to each thread in order (block cyclic schedule)
STATIC schedule

\[ \text{SCHEDULE (STATIC)} \]

\[ \text{SCHEDULE (STATIC, 4)} \]
DYNAMIC schedule

- DYNAMIC schedule divides the iteration space up into chunks of size \textit{chunksize}, and assigns them to threads on a first-come-first-served basis.

- i.e. as a thread finish a chunk, it is assigned the next chunk in the list.

- When no \textit{chunksize} is specified, it defaults to 1.
GUIDED schedule

• GUIDED schedule is similar to DYNAMIC, but the chunks start off large and get smaller exponentially.

• The size of the next chunk is proportional to the number of remaining iterations divided by the number of threads.

• The chunksize specifies the minimum size of the chunks.

• When no chunksize is specified it defaults to 1.
DYNAMIC and GUIDED schedules

1

SCHEDULE (DYNAMIC, 3) 46

1

SCHEDULE (GUIDED, 3) 46
AUTO schedule

• Lets the runtime have full freedom to choose its own assignment of iterations to threads
• If the parallel loop is executed many times, the runtime can evolve a good schedule which has good load balance and low overheads.
Choosing a schedule

When to use which schedule?

• STATIC best for load balanced loops - least overhead.
• STATIC,$n$ good for loops with mild or smooth load imbalance, but can induce overheads.
• DYNAMIC useful if iterations have widely varying loads, but ruins data locality.
• GUIDED often less expensive than DYNAMIC, but beware of loops where the first iterations are the most expensive!
• AUTO may be useful if the loop is executed many times over
RUNTIME schedule

- The RUNTIME schedule defers the choice of schedule to run time, when it is determined by the value of the environment variable `OMP_SCHEDULE`.

- e.g. `export OMP_SCHEDULE="guided,4"`

- It is illegal to specify a chunksize in the code with the RUNTIME schedule.
Nested loops

• For perfectly nested rectangular loops we can parallelise multiple loops in the nest with the `collapse` clause:

```c
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        ....
    }
}
```

• Argument is number of loops to collapse starting from the outside
• Will form a single loop of length NxM and then parallelise that.
• Useful if N is O(no. of threads) so parallelising the outer loop may not have good load balance
SINGLE directive

- Indicates that a block of code is to be executed by a single thread only.
- The first thread to reach the SINGLE directive will execute the block.
- There is a synchronisation point at the end of the block: all the other threads wait until block has been executed.
SINGLE directive (cont)

Syntax:

Fortran:

    !$OMP SINGLE [clauses]

    block

    !$OMP END SINGLE

C/C++:

    #pragma omp single [clauses]

    structured block
SINGLE directive (cont)

Example:

```c
#pragma omp parallel
{
    setup(x);

#pragma omp single
{
    input(y);
}

    work(x,y);
}
```
SINGLE directive (cont)

- SINGLE directive can take PRIVATE and FIRSTPRIVATE clauses.

- Directive must contain a structured block: cannot branch into or out of it.
• Indicates that a block of code should be executed by the master thread (thread 0) only.

• There is no synchronisation at the end of the block: other threads skip the block and continue executing: N.B. different from SINGLE in this respect.
MASTER directive (cont)

Syntax:

Fortran:

```fortran
!$OMP MASTER

block

!$OMP END MASTER
```

C/C++:

```c
#pragma omp master

structured block
```
Parallel sections

- Allows separate blocks of code to be executed in parallel (e.g. several independent subroutines)

- There is a synchronisation point at the end of the blocks: all threads must finish their blocks before any thread can proceed

- Not scalable: the source code determines the amount of parallelism available.

- Rarely used, except with nested parallelism - see later!
Parallel sections (cont)

Syntax:

Fortran:

```fortran
!$OMP SECTIONS [clauses]
[ !$OMP SECTION ]
  block
[ !$OMP SECTION
    block ]
  ...
!$OMP END SECTIONS
```
Parallel sections (cont)

C/C++:

```c
#pragma omp sections [clauses]
{
    [#pragma omp section ]
    structured-block
    [#pragma omp section
        structured-block
        . . .]
} 
```

Example:

```c
 !$OMP PARALLEL
 !$OMP SECTIONS
 !$OMP SECTION
   call init(x)
 !$OMP SECTION
   call init(y)
 !$OMP SECTION
   call init(z)
 !$OMP END SECTIONS
 !$OMP END PARALLEL
```

Diagram:
```
    init(x)    init(y)    init(z)  idle
    |___________|___________|_________
```
Parallel sections (cont)

• SECTIONS directive can take PRIVATE, FIRSTPRIVATE, LASTPRIVATE (see later) and clauses.

• Each section must contain a structured block: cannot branch into or out of a section.
Parallel section (cont)

Shorthand form:

Fortran:

\[
\texttt{!$OMP PARALLEL SECTIONS [clauses]}
\]
\[
\ldots
\]
\[
\texttt{!$OMP END PARALLEL SECTIONS}
\]

C/C++:

\[
\texttt{#pragma omp parallel sections [clauses]}
\]
\[
\{
\ldots
\}
\]
A worksharing directive (!) which allows parallelisation of Fortran 90 array operations, WHERE and FORALL constructs.

Syntax:

```
! $OMP WORKSHARE
    block
! $OMP END WORKSHARE
```
Workshare directive (cont.)

• Simple example

REAL A(100,200), B(100,200), C(100,200)

...$
!$OMP PARALLEL
!$OMP WORKSHARE
   A=B+C
!$OMP END WORKSHARE
!$OMP END PARALLEL

• N.B. No schedule clause: distribution of work units to threads is entirely up to the compiler!

• There is a synchronisation point at the end of the workshare: all threads must finish their work before any thread can proceed
Workshare directive (cont.)

• Can also contain array intrinsic functions, WHERE and FORALL constructs, scalar assignment to shared variables, ATOMIC and CRITICAL directives.
• No branches in or out of block.
• No function calls except array intrinsics and those declared ELEMENTAL.
• Combined directive:

```!$OMP PARALLEL WORKSHARE
      block
!$OMP END PARALLEL WORKSHARE```

Workshare directive (cont.)

- Example:

```c
!$OMP PARALLEL WORKSHARE REDUCTION(+:t)
  A = B + C
  WHERE (D .ne. 0) E = 1/D
  t = t + SUM(F)
  FORALL (i=1:n, X(i)=0) X(i)= 1
!$OMP END PARALLEL WORKSHARE
```
Exercise

• Redo the Mandelbrot example using a worksharing do/for directive.