



Shared Memory Programming with OpenMP

Further topics in OpenMP



Overview

- Nested parallelism
- Orphaned constructs
- Thread-private globals
- Timing routines

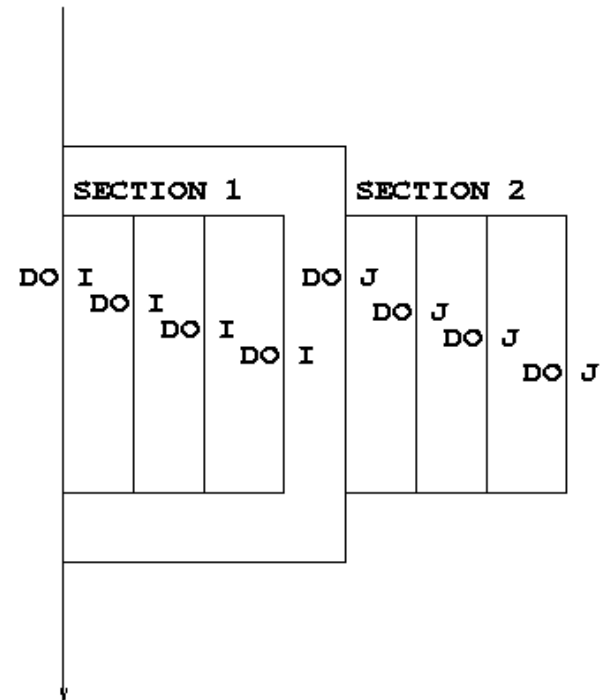
Nested parallelism

- Nested parallelism is supported in OpenMP.
- If a PARALLEL construct is encountered within another PARALLEL construct, a new team of threads will be created.
- This is enabled with the **OMP_NESTED** environment variable or the **OMP_SET_NESTED** routine.
- Not often needed, but can be useful if the outer level does not contain enough parallelism
- Usually disabled by default
 - don't enable nested parallelism unless you are using it!
- If nested parallelism is disabled, code with nested parallel regions will still execute, but the inner teams will contain only one thread.

Nested parallelism (cont)

Example:

```
!$OMP PARALLEL PRIVATE(myid)
myid = omp_get_thread_num()
if (myid .eq. 0) then
!$OMP PARALLEL DO
    do i = 1,n
        x(i) = 1.0
    end do
elseif (myid .eq.1) then
!$OMP PARALLEL DO
    do j = 1,n
        y(j) = 2.0
    end do
endif
!$OMP END PARALLEL
```



Controlling the number of threads



- Can use the environment variable

```
export OMP_NUM_THREADS=2,4
```

- Will use 2 threads at the outer level and 4 threads for each of the inner teams.
- Can use `omp_set_num_threads()` or the `num_threads` clause on the parallel region.

omp_set_num_threads()



- Useful if you want inner regions to use different numbers of threads:

```
omp_set_num_threads(2);  
#pragma omp parallel for  
    for (int i=0; i<4; i++) {  
        omp_set_num_threads(innerthreads[i]);  
#pragma omp parallel for  
        for (int j=0; j<N; j++) {  
            a[j][i] = b[j][i] * 17;  
        }  
    }
```

- The value set overrides the value(s) in the environment variable OMP_NUM_THREADS

num_threads clause

- Another way to control the number of threads used at each level is with the **num_threads** clause:

```
#pragma omp parallel for num_threads(2)
  for (int i=0; i<4; i++) {
#pragma omp parallel for num_threads(innerthreads[i])
    for (int j=0; j<N; j++) {
      a[j][i] = b[j][i] * 17;
    }
  }
```

- The value set in the clause overrides the value in the environment variable **OMP_NUM_THREADS** and that set by **omp_set_num_threads()**

More control....

- Can also control the maximum number of threads running at any one time.

```
export OMP_THREAD_LIMIT=64
```

- ...and the maximum depth of nesting

```
export OMP_MAX_ACTIVE_LEVELS=2
```

or call

```
omp_set_max_active_levels()
```


Utility routines for nested parallelism



- `omp_get_level()`
 - returns the level of parallelism of the calling thread
 - returns 0 in the sequential part
- `omp_get_active_level()`
 - returns the level of parallelism of the calling thread, ignoring levels which are inactive (teams only contain one thread)
- `omp_get_ancestor_thread_num(level)`
 - returns the thread ID of this thread's ancestor at a given level
 - ID of my parent:
`omp_get_ancestor_thread_num(omp_get_level() - 1)`
- `omp_get_team_size(level)`
 - returns the number of threads in this thread's ancestor team at a given level

Nested loops



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

```
#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 $N \times M$ and then parallelise and schedule that.
- Useful if N is $O(\text{no. of threads})$ so parallelising the outer loop may not have good load balance
- More efficient than using nested teams

Synchronisation in nested parallelism



- Note that barriers (explicit or implicit) only affect the innermost enclosing parallel region.
- No way to have a barrier across multiple teams
- In contrast, critical regions, atomics and locks affect all the threads in the program
- If you want mutual exclusion within teams but not between them, need to use locks (or atomics).

Orphaned directives

- Directives can be present in functions called from inside parallel regions
- Example:

<pre>!\$OMP PARALLEL call fred() !\$OMP END PARALLEL subroutine fred() !\$OMP DO do i = 1,n a(i) = a(i) + 23.5 end do return end</pre>	<pre>#pragma omp parallel { fred(); } void fred() { #pragma omp for for (int i=0; i<N; i++) { a[i] += 23.5; } }</pre>
---	---

Orphaned directives (cont)



- This is very useful, as it allows a modular programming style....
- But it can also be rather confusing if the call tree is complicated (what happens if **fred** is also called from outside a parallel region? - the worksharing loop is all executed by the master thread)
- There are some extra rules about data scope attributes....

Data scoping rules

When we call a subroutine from inside a parallel region:

- Variables in the argument list inherit their data scope attribute from the calling routine.
- Global variables in C/C++, and COMMON blocks or module variables in Fortran are shared, unless declared THREADPRIVATE (see later).
- **static** local variables in C/C++ and **SAVE** variables in Fortran are shared.
- All other local variables are private.

Thread private global variables



- It can be convenient for each thread to have its own copy of variables with global scope (e.g. COMMON blocks and module data in Fortran, or file-scope and namespace-scope variables in C/C++).
- Outside parallel regions and in MASTER directives, accesses to these variables refer to the master thread's copy.

Thread private globals (cont)



Fortran: **!\$OMP THREADPRIVATE** (*list*)

where *list* contains named common blocks (enclosed in slashes), module variables and SAVEd variables..

This directive must come after all the declarations for the common blocks or variables.

C/C++: **#pragma omp threadprivate** (*list*)

This directive must be at file or namespace scope, after all declarations of variables in *list* and before any references to variables in *list*. See standard document for other restrictions.

The **COPYIN** clause allows the values of the master thread's THREADPRIVATE data to be copied to all other threads at the start of a parallel region.

Timing routines

OpenMP supports a portable timer:

- return current wall clock time (relative to arbitrary origin) with:

```
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
```

```
double omp_get_wtime(void);
```

- return clock precision with

```
DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
```

```
double omp_get_wtick(void);
```

Using timers

```
DOUBLE PRECISION STARTTIME, TIME

STARTTIME = OMP_GET_WTIME()
.....(work to be timed)
TIME = OMP_GET_WTIME() - STARTTIME
```

Note: timers are local to a thread: must make both calls on the same thread.

Also note: no guarantees about resolution, but you can query it!

Exercise



Molecular dynamics again

- Aim: use of orphaned directives.
- Modify the molecular dynamics code so by placing a parallel region directive around the iteration loop in the main program, and making *all* code within this sequential except for the forces loop.
- Don't expect this to change the performance much: the idea is to see how it affects data-attribute scoping
- Be careful about harmless-looking race conditions!

Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.