OpenMP for Computational Scientists

1: Parallel worksharing

Dr Tom Deakin
University of Bristol

Tuesday 1 December, 2020

What is OpenMP?

A collection of compiler directives, library routines, and environment variables for parallelism for shared memory parallel programs.

- Create and manage parallel programs while permitting portability.
- User-directed parallelization.

A *specification* of annotations you can make to your program in order to make it parallel.

Syntax

University of BRISTOL

- OpenMP mostly formed of *compiler directives*

```
!$omp construct [clause [clause]...]
```

  These tell the compiler to insert some extra code on your behalf.

- Compiler directives usually apply to a *structured block* of statements. Limited scoping in Fortran means we often need to use *end* directives.

```
!$omp construct
... ! lines of Fortran code
!$omp end construct
```

- Library API calls

```
use omp_lib
call omp_...()
```

Building with OpenMP

University of BRISTOL

Turn on OpenMP in the compiler:

```
gfortran *.f90 -fopenmp # GNU
ifort *.f90 -qopenmp    # Intel
ftn *.f90 -homp         # Cray (now off by default)
pgf90 *.f90 -mp         # PGI
```

To also use the API calls within the code, use the library:

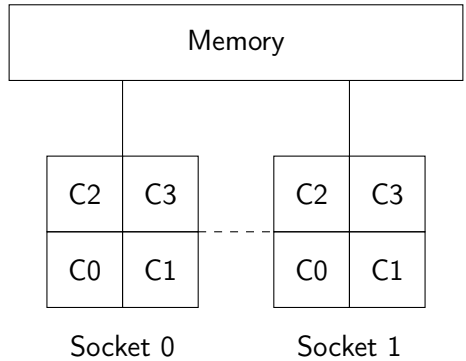
```
USE omp_lib
```

Note

No need to include the library if only using the compiler directives. The library only gets you the API calls.

Shared memory

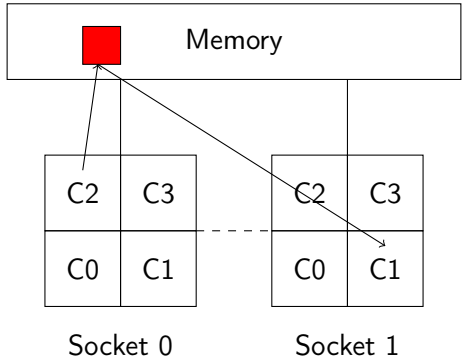
OpenMP is for shared memory programming: all threads have access to a shared address space.
A typical HPC node consisting of 2 multi-core CPUs.



All threads (each running on a core) can access the same memory.
Different to MPI, where one process cannot see the memory of another without explicit

Shared memory

OpenMP is for shared memory programming: all threads have access to a shared address space.
A typical HPC node consisting of 2 multi-core CPUs.



All threads (each running on a core) can access the same memory.
Different to MPI, where one process cannot see the memory of another without explicit

Fork-join model

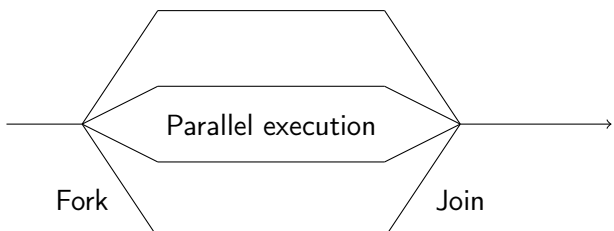
University of BRISTOL

Serial/sequential execution:

Fork-join model

Serial/sequential execution:

In a *fork-join* model, code starts serial, *forks* a *team* of threads then *joins* them back to serial execution.



Nested threads are allowed, where a thread forks its own team of threads.

Creating OpenMP threads

University of BRISTOL

```
1  program hello
2
3  !$omp parallel
4    print *, "Hello"
5  !$omp end parallel
6
7  end program hello
```

Threads *redundantly* execute code in the block.
Each thread will output `Hello`.
Threads are synchronised at the end of the parallel region.

Setting number of threads

University of BRISTOL

You might need to set the number of threads to launch (though typically you'll leave OpenMP to set the number of threads for you at run-time).
OpenMP has 3 ways to do this:

- Environment variables

```
OMP_NUM_THREADS=16
```

- API calls

```
call omp_set_num_threads(16)
```

- Clauses

```
!$omp parallel num_threads(16)
!$omp end parallel
```

In general it's better to use environment variables if you need to do this, as this approach gives you more flexibility at runtime.

Thread API calls

University of BRISTOL

Parallel programs often written in a SPMD style:
Single **P**rogram, **M**ultiple **D**ata.

- MPI has a SPMD model.
- Threads run the same code, and use their ID to work out which data to operate on.

The OpenMP API gives you calls to determine thread information when *inside* a parallel region:

- Get number of threads

```
nthreads = omp_get_num_threads()
```

- Get thread ID

```
tid = omp_get_thread_num()
```

Vector add

Walkthrough parallelising vector addition using OpenMP.

```
1  program vecadd
2    integer :: N = 1024  ! Length of array
3    ! Arrays
4    real(kind=8), allocatable, dimension(:) :: A, B, C
5    integer :: i          ! Loop counter
6
7    ! Allocate and initialise vectors
8    allocate(A(N), B(N), C(N))
9    A = 1.0; B = 2.0; C = 0.0
10
11   ! Vector add
12   do i = 1, N
13     C(i) = A(i) + B(i)
14   end do
15
16   deallocate(A,B,C)
17 end program vecadd
```

Vector add: Step 1

University of BRISTOL

Add parallel region around work

```
!$omp parallel
do i = 1, N
  C(i) = A(i) + B(i)
end do
!$omp end parallel
```

Every thread will now do the entire vector addition — redundantly!

Vector add: Step 2

University of BRISTOL

Get thread IDs

```
integer :: tid, nthreads

!$omp parallel
tid = omp_get_thread_num()
nthreads = omp_get_num_threads()

do i = 1, N
  C(i) = A(i) + B(i)
end do
!$omp end parallel
```

Vector add: Step 2

University of BRISTOL

Get thread IDs

```
integer :: tid, nthreads

!$omp parallel
tid = omp_get_thread_num()
nthreads = omp_get_num_threads()

do i = 1, N
  C(i) = A(i) + B(i)
end do
!$omp end parallel
```

Incorrect behaviour at runtime

What's the problem here?

Vector add: Step 2, take 2

University of BRISTOL

- In OpenMP, all variables are *shared* between threads.
- But each thread needs its own copy of `tid`.
- Solution: use the `private` clause on the `parallel` region.
- This gives each thread its own unique copy in memory for the variable.

```
integer :: tid, nthreads

!$omp parallel private(tid)
tid = omp_get_thread_num()
nthreads = omp_get_num_threads()

do i = 1, N
  C(i) = A(i) + B(i)
end do
!$omp end parallel
```

Much more information about data sharing clauses in next session.

Vector add: Step 3

University of BRISTOL

Finally, distribute the iteration space across the threads.

```
integer :: tid, nthreads

!$omp parallel private(tid)
tid = omp_get_thread_num()
nthreads = omp_get_num_threads()

do i = 1+(tid*N/nthreads), (tid+1)*N/nthreads
  C(i) = A(i) + B(i)
end do
!$omp end parallel
```

Remember

Thread IDs are numbered from 0 in OpenMP. Be careful with your index calculation.

Barriers

University of BRISTOL

A barrier simply synchronises threads in a parallel region.

```
1   !$omp parallel private(tid)
2
3   tid = omp_get_thread_num()
4   A(tid) = big_work1(tid)
5
6   !$omp barrier
7
8   B(tid) = big_work2(A, tid)
9
10  !$omp end parallel
```

- Running in parallel, need to compute `A(:)` before computing `B(:)`.
- The barrier ensures all threads wait between these statements.
- Must ensure all threads encounter the barrier.

Worksharing

University of BRISTOL

- The SPMD approach requires lots of bookkeeping.
- Common pattern of splitting loop iterations between threads.
- OpenMP has worksharing constructs to help with this.
- Used within a parallel region.
- The loop iterator is made `private` by default: no need for data sharing clause.

```
!$omp parallel
!$omp do
do i = 1, N
  C(i) = A(i) + B(i)
end do
!$omp end do
!$omp end parallel
```

Implicit synchronisation point at the *!$omp end do*.

Combined worksharing directives

Generally it's convenient to combine the directives:

```
!$omp parallel do
do i = 1, N
  ... ! loop body
end do
!$omp end parallel do
```

- This starts a parallel region, forking some threads.
- Each thread then gets a portion of the iteration space and computes the loop body in parallel.
- Implicit synchronisation point at the `end do`.
- Threads finally join again; later code executes sequentially.

Vector add code

The vector add codes are available in the repository for you to look at:

- Serial: `vadd.f90`
- SPMD: `vadd_spmd.f90`
- Worksharing: `vadd_paralleldo.f90`

Nested loops

University of BRISTOL

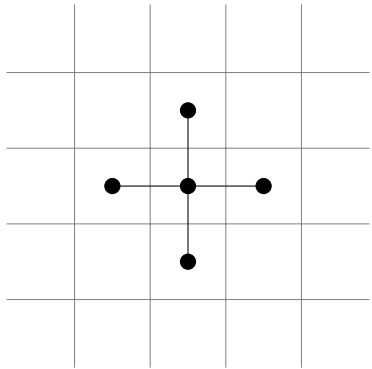
- Often have tightly nested loops in your code.
- E.g. 2D grid code, every cell is independent.
- OpenMP worksharing would only parallelise over first loop with each thread performing inner loop serially.
- Use the `collapse(...)` clause to combine iteration spaces.
- OpenMP then workshares the combined iteration space.

```
!$omp parallel do collapse(2)
do i = 1, N
  do j = 1, N
    ... ! loop body
  end do
end do
!$omp end parallel do
```

All N^2 iterations are distributed across threads, rather than just the N of the outer loop.

5-point stencil exercise

First exercise: parallelise a simple 5-point stencil code using OpenMP.



Value in every cell is set to the average of its neighbours.

5-point stencil exercise

University of BRISTOL

Take `stencil.f90` and parallelise it using OpenMP:

1. Using a SPMD style.
2. Using the OpenMP worksharing clauses.
3. Vary the number of threads using `OMP_NUM_THREADS`.

Focus on parallelising the main loop(s):

```
do i = 1, nx
  do j = 1, ny
    Anew(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j) + A(i,j-1) + A(i,j+1))
    ↪  / 5.0
  end do
end do
```

Sample solutions are provided, but do try it yourself first.

The Schedule clause

University of BRISTOL

- The worksharing clauses use default rules for assigning iterations to threads.
- Can use the `schedule` clause to specify the distribution.
- General format:

```
!$omp parallel do schedule(...)
```

Next slides go through the options, using the following loop as an example:

```
!$omp parallel do num_threads(4)
do i = 1, 100
  ... ! loop body
end do
!$omp end parallel do
```

Static schedule

University of BRISTOL

```
schedule(static)
schedule(static,16)
```

- Static schedule divides iterations into chunks and assigns chunks to threads in round-robin.
- If no chunk size specified, iteration space divided roughly equally.

For our example loop:

`schedule(static)`

Thread ID	Iterations
0	1–25
1	26–50
2	51–75
3	76–100

`schedule(static,16)`

Thread ID	Iterations
0	1–16, 65–80
1	17–32, 81–96
2	33–48, 97–100
3	49–64

Dynamic schedule

University of BRISTOL

```
schedule(dynamic)
schedule(dynamic,16)
```

- Iteration space is divided into chunks according to chunk size.
- If no chunk size specified, default size is one.
- Each thread requests and executes a chunk, until no more chunks remain.
- Useful for unbalanced work-loads if some threads complete work faster.

For our example with a chunk size of 16:

- The iteration space is split into chunk of 16 (the last chunk may be smaller).
- Each threads gets one chunk, then requests a new chunk to work on.

Guided schedule

University of BRISTOL

```
schedule(guided)
schedule(guided,16)
```

- Similar to assignment to dynamic, except the chunk size decreases over time.
- Granularity of work chunks gets finer over time.
- If no chunk size is specified, the default size is one.
- Useful to try to mitigate overheads of a `dynamic` schedule by starting with large chunks of work.

For our example with a chunk size of 16:

- Each thread gets a chunk of 16 to work on.
- Each thread requests a new chunk, which might be smaller than 16.

Other schedules

University of BRISTOL

`schedule(auto)`

- Let the compiler or runtime choose the schedule.

`schedule(runtime)`

- Get the schedule from the `OMP_SCHEDULE` environment variable.

Recommendation

Just use a `static` schedule unless there is a good reason not to! `static` is usually the fastest of all the options. The choice of schedules is an advanced tuning option.

The nowait clause

University of BRISTOL

- May have series of loops in your code which are independent.
- Threads must wait/synchronise at the end of the loop.
- But it might be possible to delay this synchronisation using the `nowait` clause.
- When a thread finishes the first loop, it starts on the next loop.

```
1  !$omp parallel
2  !$omp do nowait
3  do i = 1, N
4    A(i) = i
5  end do
6  !$omp end do        ! No barrier!
7  !$omp do
8  do i = 1, N
9    B(i) = i
10 end do
11 !$omp end do        ! Implicit barrier
12 !$omp end parallel ! Implicit barrier
```