OpenMP for Computational Scientists
1: Getting up to speed with OpenMP

Tom Deakin
University of Bristol

Thanks

Thanks go to the following authors, whose own OpenMP tutorials have inspired this one:

- ▶ Tim Mattson (Intel)
- ▶ Alice Koniges (Berkeley Lab/NERSC)
- ▶ Simon McIntosh-Smith and the HPC team (UoBristol)
- ▶ Gethin Williams (UoBristol)
- ▶ and many others

Exercises

- ▶ This is a hands-on course!
- ▶ Exercises will be set for you to try programming OpenMP yourselves.
- ▶ Sample solutions also provided.
- ▶ All the exercises will be in Fortran.

Download code (and slides) from:
https://github.com/UoB-HPC/openmp-for-cs

Course Outline

Organised as 6 sessions teaching OpenMP plus top-tips for getting good performance.

1. OpenMP overview
2. Data sharing and reductions
3. Vectorisation and code optimisations
4. NUMA and MPI interoperability
5. GPU programming with OpenMP
6. Tasks and Tools

The first exercise

- At the end of this session, you will be able to parallelise a (simple) 5-point stencil code using OpenMP!
- The other sessions provide you with details you might need for real world codes.

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

► 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

Turn on OpenMP in the compiler:

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

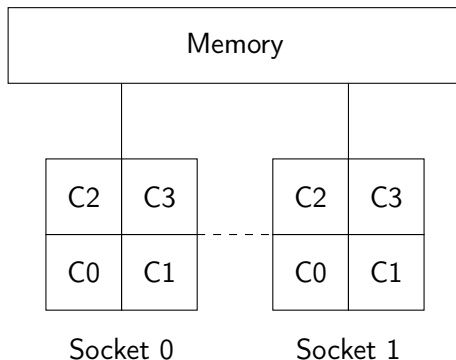To also use the API calls within the code, use the library:

```
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.
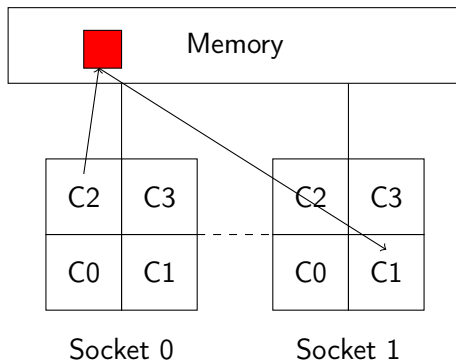
A typical HPC node consisting of 2 multi-core CPUs.

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 communication.

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 communication.
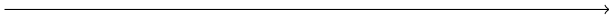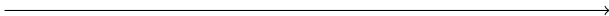
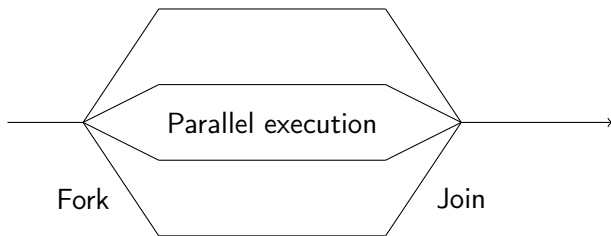University of BRISTOL

Serial/sequential execution:

$\longrightarrow$

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

```fortran
program hello

!$omp parallel
  print *, "Hello"
!$omp end parallel

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.

Pthreads

```fortran
1  program hello
2    use fpthread
3    integer :: i, err
4    integer :: N = 4
5    type(fpthread_t) :: Tide(N)
6
7    do i = 1, N
8      call fpthread_create(tid(i), NULL, run, NULL, err)
9    end do
10   do i = 1, N
11     call fpthread_join(tid(i), NULL, err)
12   end do
13
14   subroutine run
15     print *, "Hello"
16   end subroutine run
17 end program hello
```

OpenMP and Pthreads

- ▶ Pthreads is very error prone and verbose.
- ▶ The OpenMP `!$omp parallel` abstracts this away.
- ▶ The compiler directive inserts this extra code on your behalf.
- ▶ Pthreads requires wrapping up your parallel work in subroutines.
  - ▶ Kernels are a useful abstraction used in many programming models.
- ▶ OpenMP much more convenient for *incrementally* adding parallelism to your code.

Setting number of threads

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

Most parallel programs are 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.

```fortran
program vecadd
  integer :: N = 1024   ! Length of array
  ! Arrays
  real(kind=8), allocatable, dimension(:) :: A, B, C
  integer :: i            ! Loop counter

  ! Allocate and initialise vectors
  allocate(A(N), B(N), C(N))
  A = 1.0; B = 2.0; C = 0.0

  ! Vector add
  do i = 1, N
    C(i) = A(i) + B(i)
  end do

  deallocate(A,B,C)
end program vecadd
```

Add parallel region around work

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

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

Get thread IDs

```fortran
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
```

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

- ▶ 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.

```fortran
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.

University of BRISTOL

Finally, distribute the iteration space across the threads.

```fortran
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.

Worksharing

- ▶ 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.

```fortran
!$omp parallel
!$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

University of BRISTOL

Generally it's convenient to combine the directives:

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

Taking stock

- ▶ We've seen how to parallelise a simple program using OpenMP.
- ▶ Shown the MPI-style SPMD approach for dividing work.
- ▶ OpenMP worksharing constructs make this easier.

The rest of this session:

- ▶ Expands on the worksharing constructs.
- ▶ The first example for you to try.

Next time: the rest of the OpenMP common core.

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

```
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

```
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.

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

`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.

Nested loops

- ▶ 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.

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

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

Nested loops

Performance note

Collapsing loops may subtly effect the compiler's knowledge about alignment and could affect vectorisation. More on this when we talk about vectorisation in a later session.

The nowait clause

- ▶ 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.

```fortran
!$omp parallel
!$omp do nowait
do i = 1, N
  A(i) = i
end do
!$omp end do        ! No barrier!
!$omp do
do i = 1, N
  B(i) = i
end do
!$omp end do        ! Implicit barrier
!$omp end parallel  ! Implicit barrier
```

Synchronisation

A number of ways to synchronise the threads in OpenMP:

- Barriers
- Critical
- Atomics
- Locks

- Ordered
- Single
- Master
- Flush

- Will look at Critical and Atomic in Session 2.
- Ordered, Single and Master in Session 6.
- Won't formally cover Flush and Locks — advanced stuff with esoteric use cases.

Quickly cover barriers now.

A barrier simply synchronises threads in a parallel region.

```fortran
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.

Nested threads

- ▶ Turn on support with by setting the environment variable OMP_NESTED=true, otherwise inner region is default serial.
- ▶ Every thread in the (outer) parallel region then spawns threads.
- ▶ Control the number of threads with clauses or environment variable: OMP_NUM_THREADS=4,2.

```
!$omp parallel num_threads(4)
... ! A parallel region
  !$omp parallel num_threads(4)
  ... ! Inner parallel region
  !$omp end parallel
!$omp end parallel
```

Nested threads

Warning!

Be careful how you use nesting threads. It's very easy to oversubscribe threads. Thread affinity can be tricky. You probably don't need to use nested threads!

Multi-line directives

- Sometimes OpenMP directives can be quite long.
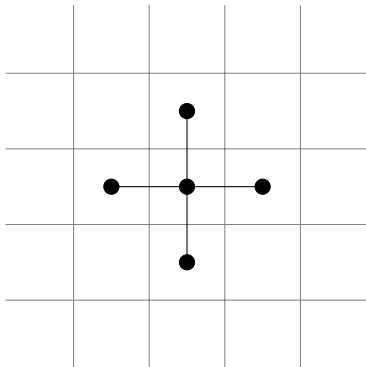- Nicer to split up the directive across lines in the source file using line continuation character &:

```
!$omp construct &
!$omp& clause
!$omp& clause
```

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

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):

```fortran
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.

Summary

This section introduced the OpenMP programming model:

- ▶ Creating parallel regions:
  !$omp parallel / *!$omp end parallel*
- ▶ Getting thread IDs:
  omp_get_thread_num()/omp_get_num_threads()
- ▶ Worksharing constructs: *!$omp do* / *!$omp end do*
- ▶ The schedule and nowait clauses
- ▶ Synchronising threads with barriers: *!$omp barrier*

Resources

- ▶ OpenMP website: `https://www.openmp.org`
  - ▶ The specification (not for the faint hearted).
  - ▶ Download summary cards.
  - ▶ List of compiler support.
  - ▶ Example code for all the directives.
  - ▶ List of books:
    `https://www.openmp.org/resources/openmp-books/`
- ▶ cOMPunity
  - ▶ `http://www.compunity.org`
- ▶ Online tutorials:
  - ▶ Tim Mattson's YouTube tutorial:
    `https://youtu.be/nE-xN4Bf8XI`
  - ▶ SC'08 tutorial from Tim Mattson and Larry Meadows: `https://openmp.org/mp-documents/omp-hands-on-SC08.pdf`
  - ▶ From Lawrence Livermore National Lab:
    `https://computing.llnl.gov/tutorials/openMP/`