A brief introduction to OpenMP

1: Getting up to speed with OpenMP

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*

  ```
  #pragma 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.

  ```
  #pragma omp construct
  {
    ... // lines of C code
  }
  ```

- Library API calls

  ```
  #include <omp.h>
  omp_...();
  ```

Building with OpenMP

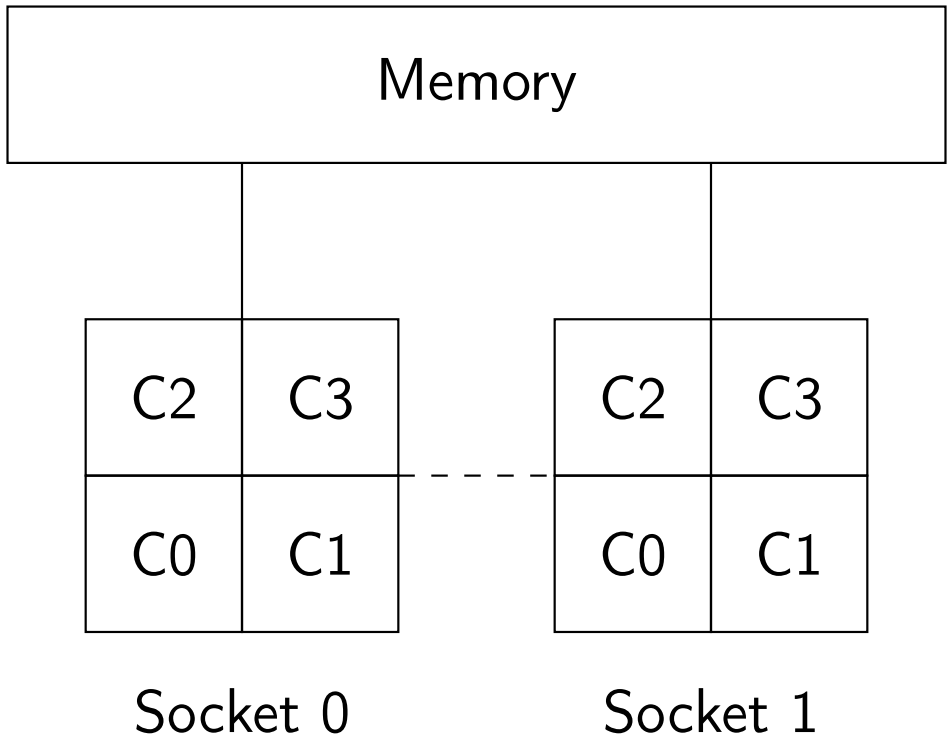
Turn on OpenMP in the compiler:

```
gcc *.c -fopenmp     # GNU
icc *.c -qopenmp     # Intel
```

Can also build MPI and OpenMP programs by passing flags to `mpicc`.

Shared memory

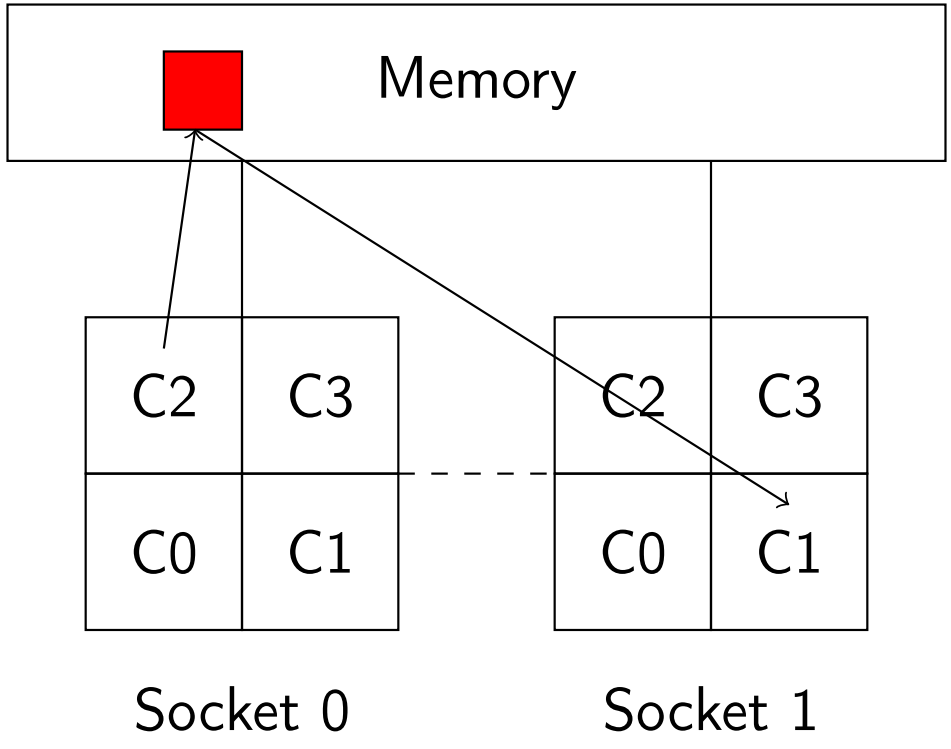
OpenMP is for shared memory programming: all threads have access to a shared address space.
A typical HPC node consisting of 2 multi-core CPUs.



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

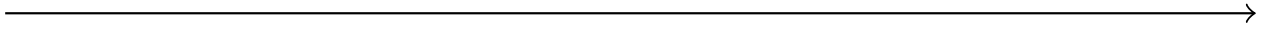
OpenMP is for shared memory programming: all threads have access to a shared address space.
A typical HPC node consisting of 2 multi-core CPUs.



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.

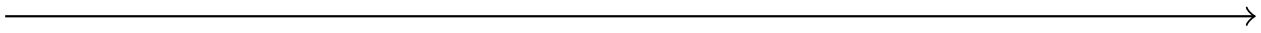
Fork-join model

Serial/sequential execution:

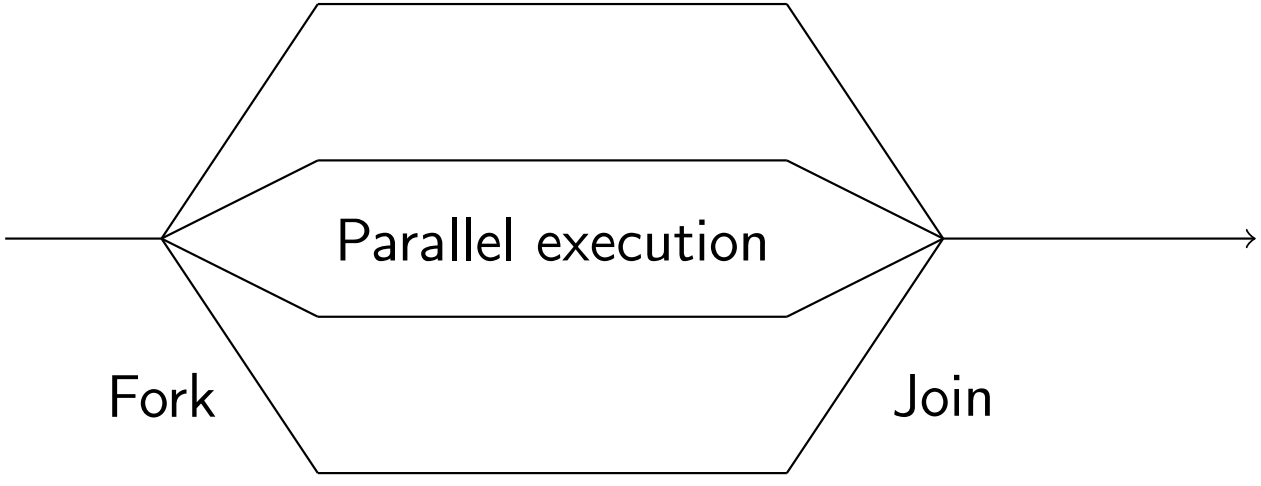


Fork-join model

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  #pragma omp parallel
2  {
3    printf("Hello\n");
4  }
```

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

```
omp_set_num_threads(16);
```

- Clauses

```
#pragma omp parallel num_threads(16)
{
}
```

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

```
int nthreads = omp_get_num_threads();
```

- Get thread ID

```
int tid = omp_get_thread_num();
```

Vector add

Walkthrough parallelising vector addition using OpenMP.

```
1  int main(void) {
2    int N = 1024; // Length of array
3    double *A, *B, *C; // Arrays
4
5    // Allocate and initialise vectors
6    A = malloc(sizeof(double)*N);
7    B = malloc(sizeof(double)*N);
8    C = malloc(sizeof(double)*N);
9    for (int i = 0; i < N; ++i) {
10     A[i] = 1.0; B[i] = 2.0; C[i] = 0.0;
11   }
12
13   // Vector add
14   for (int i = 0; i < N; ++i)
15     C[i] = A[i] + B[i];
16
17   free(A); free(B); free(C);
18 }
```

Vector add: Step 1

Add parallel region around work

```
#pragma omp parallel
{
  for (int i = 0; i < N; ++i)
    C[i] = A[i] + B[i];
}
```

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

Vector add: Step 2

Get thread IDs

```
int tid, nthreads;

#pragma omp parallel
{
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  for (int i = 0; i < N; ++i)
    C[i] = A[i] + B[i];
}
```

Vector add: Step 2

University of BRISTOL

Get thread IDs

```
int tid, nthreads;

#pragma omp parallel
{
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  for (int i = 0; i < N; ++i)
    C[i] = A[i] + B[i];
}
```

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 or declare variables inline.
- This gives each thread its own unique copy in memory for the variable.

```
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int nthreads = omp_get_num_threads();

  for (int i = 0; i < N; ++i)
    C[i] = A[i] + B[i];
}
```

Much more information about data sharing clauses in next session.

Vector add: Step 3

Finally, distribute the iteration space across the threads.

```
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int nthreads = omp_get_num_threads();

  int i;
  for (i = tid*N/nthreads; i < (tid+1)*N/nthreads; ++i)
    C[i] = A[i] + B[i];
}
```

Remember

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

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.

```
int i;
#pragma omp parallel
{
  #pragma omp for
  for (i = 0; i < N; ++i) {
    C[i] = A[i] + B[i];
  } // End of for loop
} // End of parallel region
```

Implicit synchronisation point at the `} // End of for loop`.

Combined worksharing directives

Generally it's convenient to combine the directives:

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
  ... // Loop body
}
```

- 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 }.
- Threads finally join again; later code executes sequentially.

Taking stock

University of BRISTOL

- 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.

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:

```
#pragma omp parallel for schedule(...)
```

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

```
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 100; ++i) {
  ... // loop body
}
```

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 chunks of 16 (the last chunk may be smaller).
- Each thread gets one chunk, then requests a new chunk to work on.

Guided schedule

University of BRISTOL

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

- Similar 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

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.

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

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

Nested loops

University of BRISTOL

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

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  #pragma omp parallel
2  {
3    #pragma omp for nowait
4    for (int i = 0; i < N; ++i) {
5      A[i] = i;
6    } // No barrier!
7
8    #pragma omp for
9    for (int i = 0; i < N; ++i) {
10     B[i] = i;
11   } // Implicit barrier
12 } // Implicit barrier
```

Barriers

A barrier simply synchronises threads in a parallel region.

```
1  #pragma omp parallel
2  {
3
4  int tid = omp_get_thread_num();
5  A[tid] = big_work1(tid);
6
7  #pragma omp barrier
8
9  B[tid] = big_work2(A, tid);
10
11 }
```

- 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.

Multi-line directives

University of BRISTOL

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

```
#pragma omp construct \
clause \
clause
```

Summary

This section introduced the OpenMP programming model:

- Creating parallel regions: `pragma omp parallel`
- Getting thread IDs: `omp_get_thread_num()`/`omp_get_num_threads()`
- Worksharing constructs: `pragma omp for`
- The `schedule` and `nowait` clauses
- Synchronising threads with barriers: `pragma omp barrier`

Resources

University of BRISTOL

- 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/`