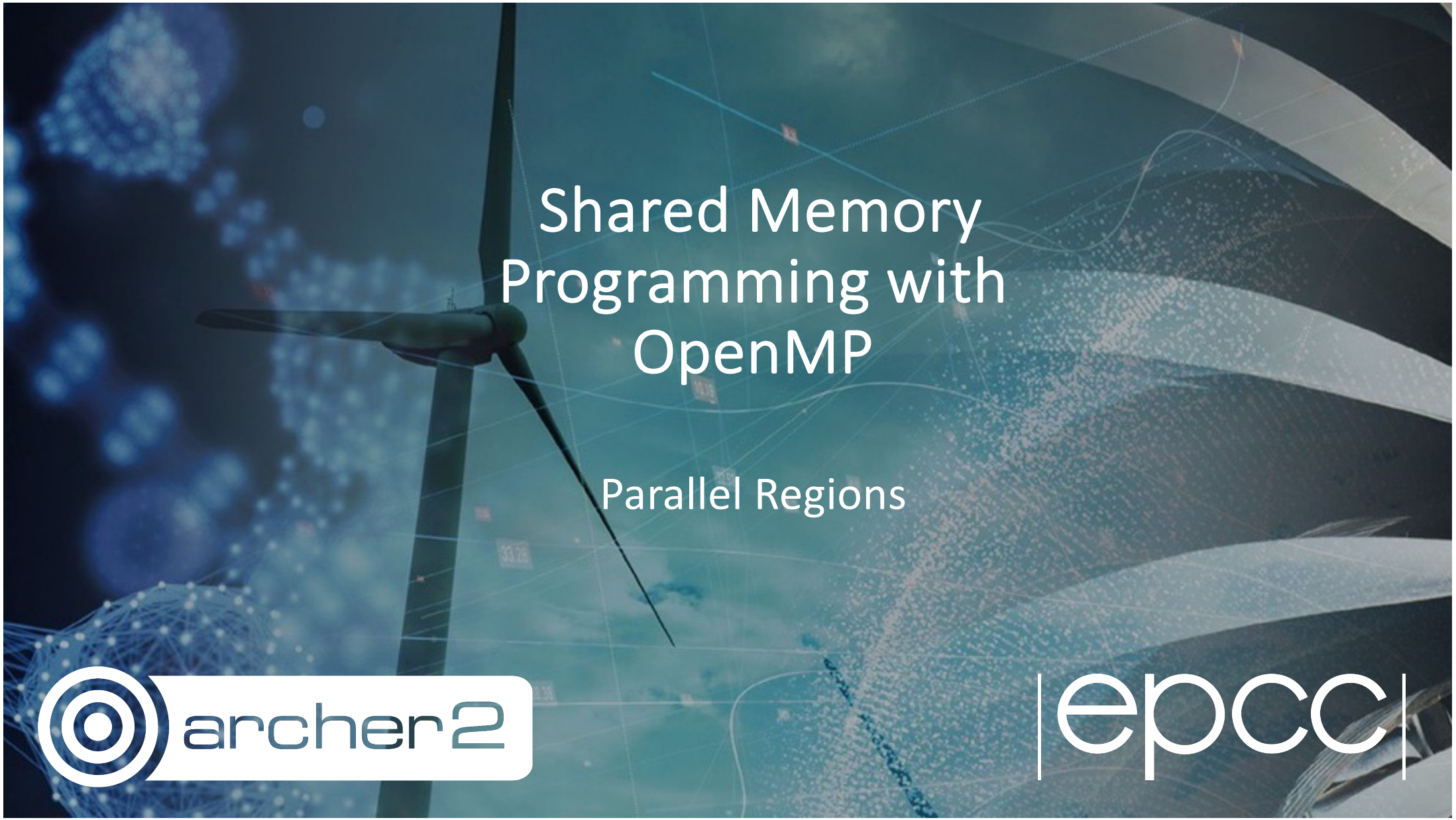

Shared Memory Programming with OpenMP
Parallel Regions
archer2
epcc

Parallel region directive

epcc

- Code within a parallel region is executed by all threads.
- Syntax:

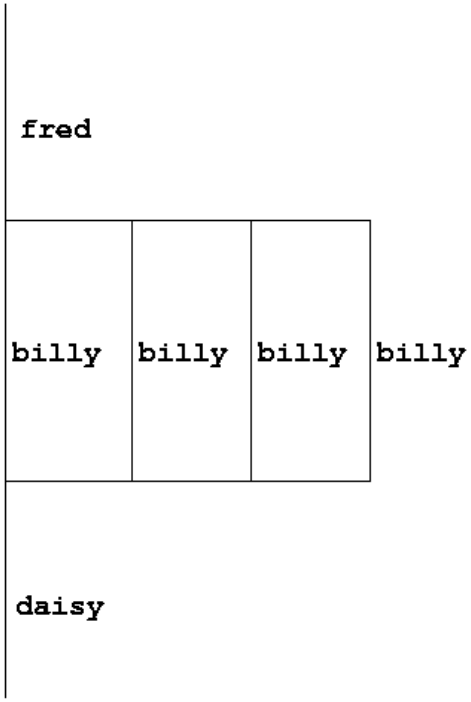
C/C++:

```
#pragma omp parallel
{
    block
}
```

Parallel region directive (cont)

Example:

```
fred();
#pragma omp parallel
{
   billy();
}
daisy();
```



Useful functions

epcc

- Often useful to find out number of threads being used.

C/C++:

```
#include <omp.h>
    int omp_get_num_threads(void);
```

- Important note: returns 1 if called outside parallel region!

Useful functions (cont)

epcc

- Also useful to find out number of the executing thread.

C/C++:

```
#include <omp.h>
    int omp_get_thread_num(void)
```

- Takes values between 0 and `omp_get_num_threads()-1`

Clauses

epcc

- Specify additional information in the parallel region directive through clauses:

- C/C++: `#pragma omp parallel [clauses]`

- Clauses are comma or space separated.

Shared and private variables

epcc

- Inside a parallel region, variables can be either shared (all threads see same copy) or private (each thread has its own copy).
- Shared, private and default clauses

C/C++: `shared(`*list*`)`

`private(`*list*`)`

`default(shared|none)`

In C/C++, variables declared inside the scope of the parallel region are private.

Shared and private (cont.)

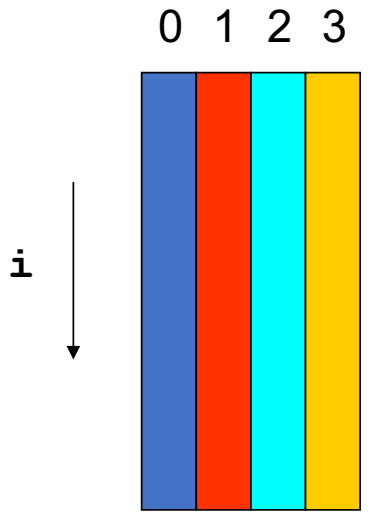
- On entry to a parallel region, private variables are uninitialised.
- Variables declared inside the scope of the parallel region are automatically private.
- After the parallel region ends the original variable is unaffected by any changes to private copies.
- In C++ private objects are created using the default constructor
- Not specifying a `default` clause is the same as specifying `default(shared)`
- Danger!
  - Always use `default(none)`

Shared and private (cont)

Example - each thread initialises its own part of a shared array:

```
int i,myid,n=1000;
float a[4][1000];

#pragma omp parallel default(none) private(i,myid) shared(a,n)
{
    myid = omp_get_thread_num();
    for (i=0; i<n; i++){
      a[myid][i] = 1.0;
    }
}
```

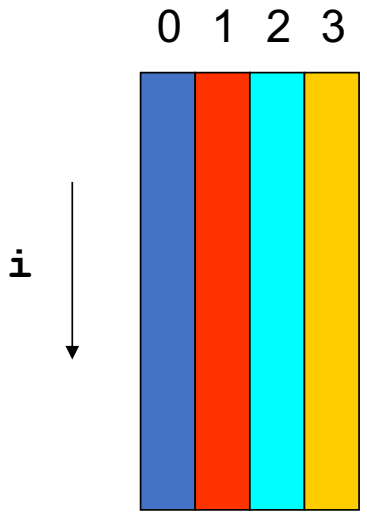


Shared and private (cont)

Using the C language scoping:

```
int n=1000;
float a[4][1000];

#pragma omp parallel default(none) shared(a,n)
{
    int myid = omp_get_thread_num();
    for (int i=0; i<n; i++){
      a[myid][i] = 1.0;
    }
}
```



Multi-line directives

epcc

- C/C++:

```
#pragma omp parallel default(none) \
private(i,myid) shared(a,n)
```

Note: no white space after the \

Initialising private variables

- Private variables are uninitialised at the start of the parallel region.
- If we wish to initialise them, we use the **`firstprivate`** clause instead of **`private`** :

C/C++: **`firstprivate`**`(`*list*`)`

- Private copies are initialised with the value in the original variable at the start of the parallel region
- Note: use cases for this are surprisingly uncommon!
- In C++ the default copy constructor is called to create and initialise the new object

Initialising private variables (cont)

Example:

```
      float b = 23.0;
      float c[4][1000];
      . . . . .
#pragma omp parallel firstprivate(b), private(i,myid)
   {
      myid = omp_get_thread_num();
      for (i=0; i<n; i++){
         b += c[myid][i];
      }
      c[myid][n] = b;
   }
```

Reductions

- A *reduction* produces a single value from associative operations such as addition, multiplication, max, min, and, or.
- Would like each thread to reduce into a private copy, then reduce all these to give final result.
- Use **`reduction`** clause:

C/C++: **`reduction`**`(`*op* `:` *list*`)`

- Can have reduction arrays – reduction is done on each element
  - In C/C++, need to use a special OpenMP syntax for array sections
  - **`array[firstelement:numofelements]`**

Reductions (cont.)

Example:

```
  int b = 10;
#pragma omp parallel reduction(+:b)
  {
      int myid = omp_get_thread_num();
      for (int i=0; i<n; i++){
         b += c[myid][i];
      }
  }
  a = b;
```

Value in original variable is saved

Each thread gets a private copy of **b**, initialised to 0

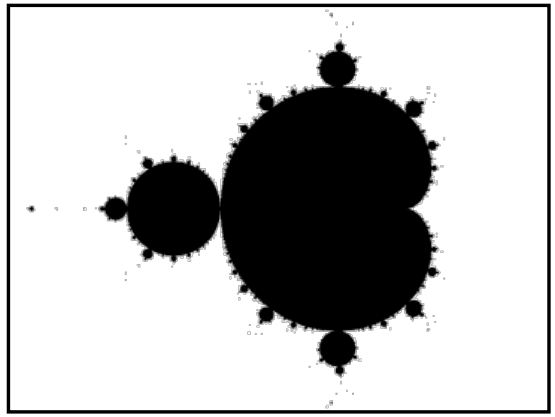
All accesses inside the parallel region are to the private copies

At the end of the parallel region, all the private copies are added into the original variable

Exercise

Area of the Mandelbrot set

- Aim: introduction to using parallel regions.
- Estimate the area of the Mandelbrot set.
  - Sequential code is provided as a starting point
  - Generates a grid of complex numbers in a box surrounding the set
  - Tests each number to see if it is in the set or not.
  - Ratio of points inside to total number of points gives an estimate of the area.
  - Testing of points is independent - parallelise with a parallel region!
  - Using the thread ID and number of threads, calculate which loop iterations each thread should execute



Reusing this material