Shared Memory Programming

Parallel Regions

Parallel region directive

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

Fortran:

```
!$OMP PARALLEL
        block
!$OMP END PARALLEL
```

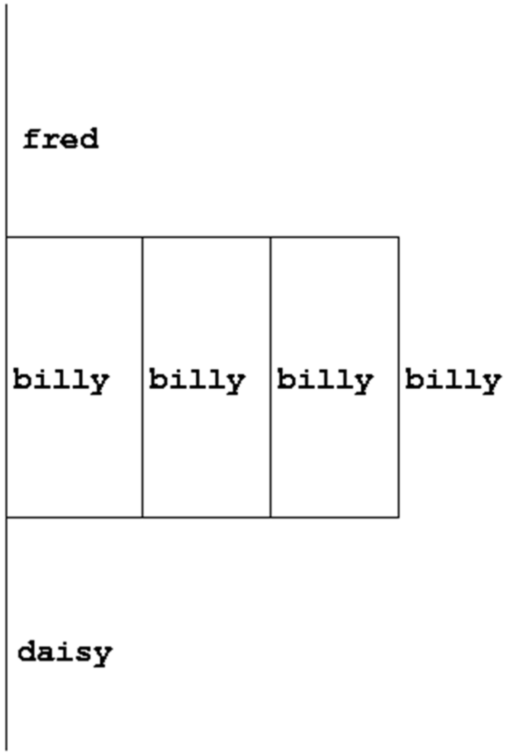
C/C++:

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

Parallel region directive (cont)

Example:

```
      call fred()
!$OMP PARALLEL
      call billy()
!$OMP END PARALLEL
      call daisy()
```



Clauses

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

Fortran : `!$OMP PARALLEL` *[clauses]*

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

- Clauses are comma or space separated in Fortran, space separated in C/C++.

Shared and private variables

- 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

Fortran: **SHARED(***list***)**

PRIVATE(*list***)**

DEFAULT(SHARED|PRIVATE|NONE)

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

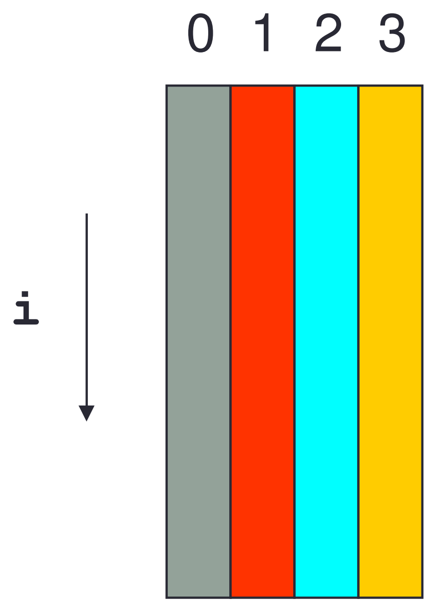
private(*list***)**

default(shared|none)

Shared and private (cont)

Example: each thread initialises its own column of a shared array:

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID),
!$OMP& SHARED(A,N)
      myid = omp_get_thread_num() + 1
      do i = 1,n
         a(i,myid) = 1.0
      end do
!$OMP END PARALLEL
```



Shared and private (cont)

- How do we decide which variables should be shared and which private?
  - Write-before-read scalars - usually private
    - e.g. loop indices and loop temporaries
  - Read-only variables - shared
  - Main arrays - shared
  - Sometimes either is semantically OK, but there may be performance implications in making the choice.
- N.B. can have private arrays as well as scalars
  - making large arrays private may cause the program to exhaust memory resources
- Making this decision is often the hardest part of writing OpenMP code

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:

Fortran: **`FIRSTPRIVATE(`***list***`)`**

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

Initialising private variables (cont)

Example:

```
      b = 23.0;
       . . . . .
#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:

Fortran: **`REDUCTION(`***op***`:`***list***`)`**

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

- Can have reduction arrays in Fortran, but not in C/C++

Reductions (cont.)

Example:

```
      b = 10
!$OMP PARALLEL REDUCTION(+:b),
!$OMP& PRIVATE(I,MYID)
      myid = omp_get_thread_num() + 1
      do i = 1,n
         b = b + c(i,myid)
      end do
!$OMP END PARALLEL
      a = b
```

- Value in original variable is saved (b = 10)
- Each thread gets a private copy of **b**, initialised to 0 (PARALLEL REDUCTION(+:b))
- All accesses inside the parallel region are to the private copies (b = b + c(i,myid))
- At the end of the parallel region, all the private copies are added into the original variable (a = b)

IF clause

- We can make the parallel region directive itself conditional.
- Can be useful if there is not always enough work to make parallelism worthwhile.

Fortran: **`IF`** **(***scalar logical expression***)**

`C/C++`: **`if`** **(***scalar expression***)**

IF clause (cont.)

Example:

```
#pragma omp parallel if (tasks > 1000)
{
  while(tasks > 0) donexttask();
}
```

Multi-line directives

- Fortran: fixed source form

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID),
!$OMP& SHARED(A,N)
```

- Fortran: free source form

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID), &
!$OMP SHARED(A,N)
```

- C/C++:

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

Useful functions

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

Fortran:

```
USE OMP_LIB
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

C/C++:

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

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

Useful functions (cont)

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

Fortran:

```
USE OMP_LIB
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

C/C++:

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

- Returns values between 0 and **OMP_GET_NUM_THREADS()** - 1

Practical Session

Area of the Mandelbrot set

- Aim: introduction to using parallel regions.
- Estimate the area of the Mandelbrot set by Monte Carlo sampling.
  - Generate a set of random complex numbers in a box surrounding the set
  - Test 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!