Advanced OpenMP

OpenMP Basics

Parallel region

- The *parallel region* is the basic parallel construct in OpenMP.
- A parallel region defines a section of a program.
- Program begins execution on a single thread (the master thread).
- When the first parallel region is encountered, the master thread creates a team of threads (fork/join model).
- Every thread executes the statements which are inside the parallel region
- At the end of the parallel region, the master thread waits for the other threads to finish, and continues executing the next statements

Parallel region

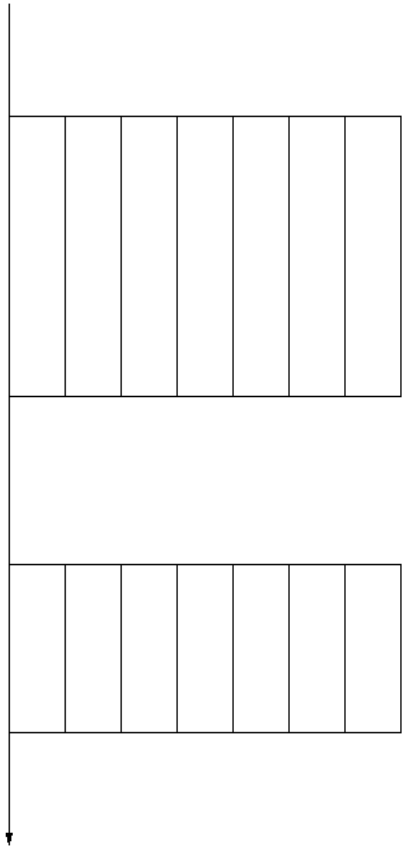
Sequential part

Parallel region

Sequential part

Parallel region

Sequential part



```
      PROGRAM FRED
        .
        .
!$OMP PARALLEL
        .
        .
        .
        .
        .
        .
        .
        .
        .
!$OMP END PARALLEL
        .
        .
        .
        .
        .
!$OMP PARALLEL
        .
        .
        .
        .
        .
!$OMP END PARALLEL
        .
        .
        .
```

```
int main(){
.
.
#pragma omp parallel
{
.
.
.
.
.
.
.
.
.
}
.
.
.
.
#pragma omp parallel
{
.
.
.
}
.
.
.
```

epcc

THE UNIVERSITY OF EDINBURGH

Parallel region directive

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

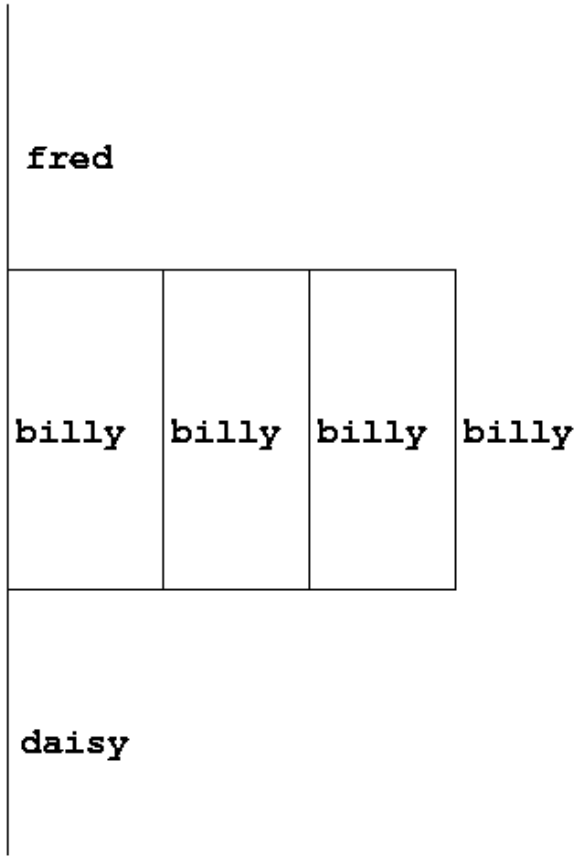
Fortran:
```
!$OMP PARALLEL
      block
!$OMP END PARALLEL
```

C/C++:
```
#pragma omp parallel
{
   block
}
```

Parallel region directive (cont)

Example:

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



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)
```

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

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.

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)`

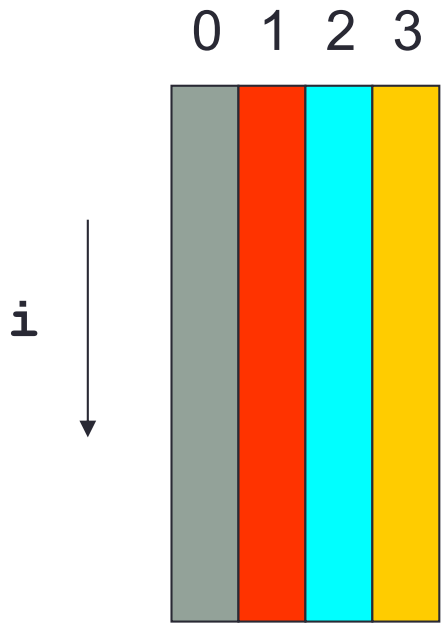
Shared and private (cont.)

- 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 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
```



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)
```

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*`)`

- Note: use cases for this are uncommon!
- In C++ the default copy constructor is called to create and initialise the new object

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
- In C/C++, can use special OpenMP syntax for array sections

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

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

Work sharing directives

- Directives which appear inside a parallel region and indicate how work should be shared out between threads
  - Parallel do/for loops
  - Single directive
  - Master directive

Parallel do loops

- Loops are the most common source of parallelism in most codes. Parallel loop directives are therefore very important!
- A parallel do/for loop divides up the iterations of the loop between threads.
- The loop directive appears inside a parallel region and indicates that the work should be shared out between threads, instead of replicated
- There is a synchronisation point at the end of the loop: all threads must finish their iterations before any thread can proceed

Parallel do/for loops (cont)

Syntax:

Fortran:

```
!$OMP DO [clauses]
      do loop
[ !$OMP END DO ]
```

C/C++:

```
#pragma omp for [clauses]
      for loop
```

Restrictions in C/C++

- Because the for loop in C is a general while loop, there are restrictions on the form it can take.

- It has to have determinable trip count - it must be of the form:

  `for (var = a; var` *logical-op* **`b;`** *incr-exp*`)`

  where *logical-op* is one of `<, <=, >, >=`

  and *incr-exp* is **`var = var +/- incr`** or semantic equivalents such as **`var++`**.

  Also cannot modify **`var`** within the loop body.

Parallel loops (example)

Example:

```
!$OMP PARALLEL
!$OMP DO
   do i=1,n
      b(i) = (a(i)-a(i-1))*0.5
   end do
!$OMP END DO
!$OMP END PARALLEL
```

```
#pragma omp parallel
{
#pragma omp for
  for (int i=0;i<n;i++){
    b[i] = (a[i]*a[i-1])*0.5;
  }
}
```

Parallel DO/FOR directive

- This construct is so common that there is a shorthand form which combines parallel region and DO/FOR directives:

Fortran:

```
!$OMP PARALLEL DO [clauses]
   do loop
[ !$OMP END PARALLEL DO ]
```

C/C++:

```
#pragma omp parallel for [clauses]
   for loop
```

Clauses

- DO/FOR directive can take PRIVATE , FIRSTPRIVATE and REDUCTION clauses which refer to the scope of the loop.

- Note that the parallel loop index variable is PRIVATE by default
  - other loop indices are private by default in Fortran, but not in C.

- PARALLEL DO/FOR directive can take all clauses available for PARALLEL directive.

- Beware! PARALLEL DO/FOR is not the same as DO/FOR or the same as PARALLEL

Parallel do/for loops (cont)

- With no additional clauses, the DO/FOR directive will partition the iterations as equally as possible between the threads.

- However, this is implementation dependent, and there is still some ambiguity:

e.g. 7 iterations, 3 threads. Could partition as 3+3+1 or 3+2+2

SCHEDULE clause

- The SCHEDULE clause gives a variety of options for specifying which loops iterations are executed by which thread.
- Syntax:

Fortran: **`SCHEDULE (`***kind[, chunksize]*`)`**

C/C++: **`schedule (`***kind[, chunksize]*`)`**

where *kind* is one of

`STATIC`, **`DYNAMIC`**, **`GUIDED`**, **`AUTO`** or **`RUNTIME`**

and *chunksize* is an integer expression with positive value.

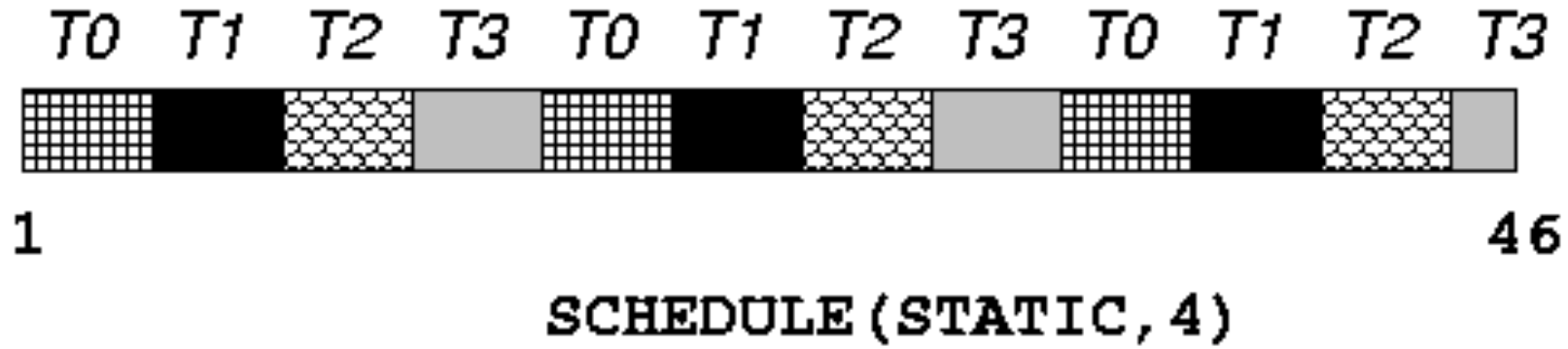
- E.g. **`!$OMP DO SCHEDULE(DYNAMIC,4)`**

STATIC schedule

- With no *chunksize* specified, the iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread in order (**block** schedule).

- If *chunksize* is specified, the iteration space is divided into chunks, each of *chunksize* iterations, and the chunks are assigned cyclically to each thread in order (**block cyclic** schedule)

STATIC schedule



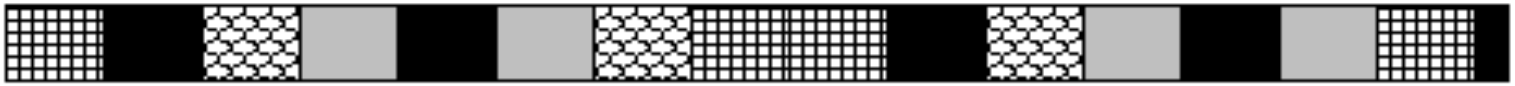
DYNAMIC schedule

- DYNAMIC schedule divides the iteration space up into chunks of size *chunksize*, and assigns them to threads on a first-come-first-served basis.
- i.e. as a thread finish a chunk, it is assigned the next chunk in the list.
- When no *chunksize* is specified, it defaults to 1.

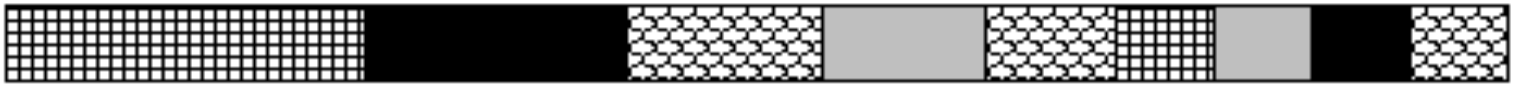
GUIDED schedule

- GUIDED schedule is similar to DYNAMIC, but the chunks start off large and get smaller exponentially.
- The size of the next chunk is proportional to the number of remaining iterations divided by the number of threads.
- The *chunksize* specifies the minimum size of the chunks.
- When no *chunksize* is specified it defaults to 1.

DYNAMIC and GUIDED schedules



1 SCHEDULE(DYNAMIC,3) 46



1 SCHEDULE(GUIDED,3) 46

AUTO schedule

- Lets the runtime have full freedom to choose its own assignment of iterations to threads
- If the parallel loop is executed many times, the runtime can evolve a good schedule which has good load balance and low overheads.

Choosing a schedule

When to use which schedule?

- STATIC best for load balanced loops - least overhead.
- STATIC,*n* good for loops with mild or smooth load imbalance, but can induce overheads.
- DYNAMIC useful if iterations have widely varying loads, but ruins data locality.
- GUIDED often less expensive than DYNAMIC, but beware of loops where the first iterations are the most expensive!
- AUTO may be useful if the loop is executed many times over

SINGLE directive

- Indicates that a block of code is to be executed by a single thread only.
- The first thread to reach the SINGLE directive will execute the block
- There is a synchronisation point at the end of the block: all the other threads wait until block has been executed.

SINGLE directive (cont)

Syntax:

Fortran:

```
!$OMP SINGLE [clauses]
      block
!$OMP END SINGLE
```

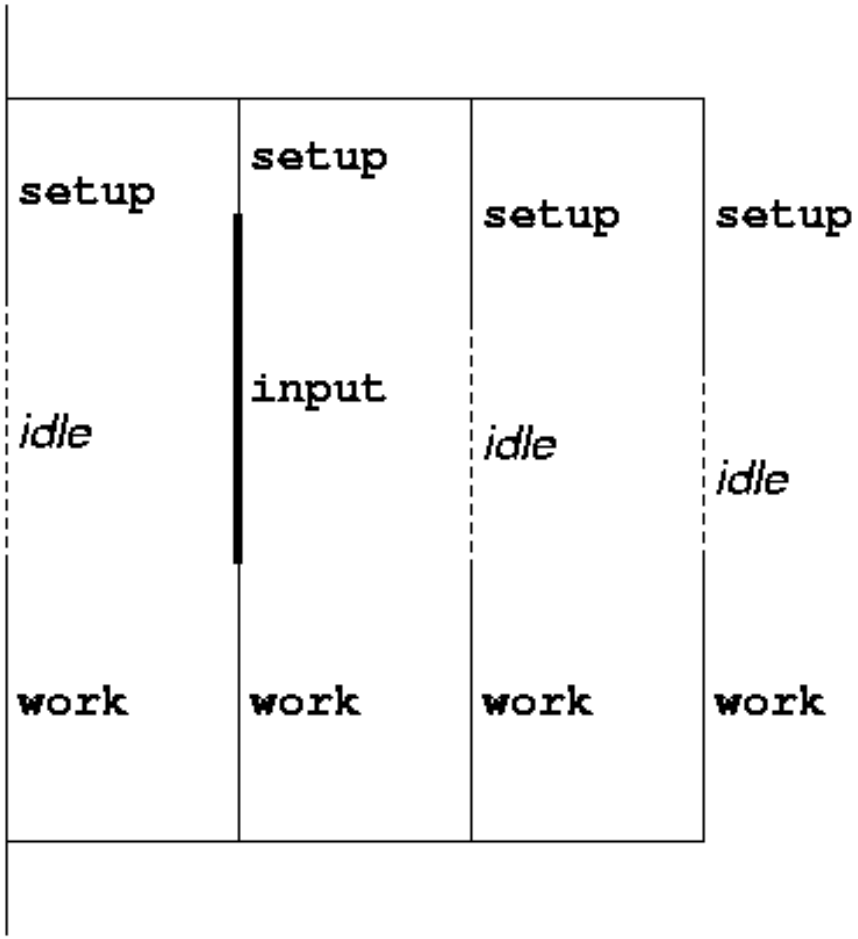
C/C++:

```
#pragma omp single [clauses]
      structured block
```

SINGLE directive (cont)

Example:

```
#pragma omp parallel
{
    setup(x);
#pragma omp single
   {
      input(y);
   }
    work(x,y);
}
```



SINGLE directive (cont)

- SINGLE directive can take PRIVATE and FIRSTPRIVATE clauses.
- Directive must contain a structured block: cannot branch into or out of it.

MASTER directive

- Indicates that a block of code should be executed by the master thread (thread 0) only.
- There is no synchronisation at the end of the block: other threads skip the block and continue executing: N.B. different from SINGLE in this respect.

MASTER directive (cont)

Syntax:

Fortran:

```
!$OMP MASTER
      block
!$OMP END MASTER
```

C/C++:

```
#pragma omp master
      structured block
```

BARRIER directive

- No thread can proceed past a barrier until all the other threads have arrived.
- Note that there is an implicit barrier at the end of DO/FOR, SECTIONS and SINGLE directives.

- Syntax:

Fortran: `!$OMP BARRIER`

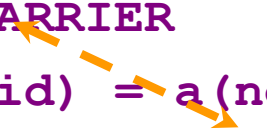
C/C++: `#pragma omp barrier`

- Either all threads or none must encounter the barrier: otherwise DEADLOCK!!

BARRIER directive (cont)

Example:

```
!$OMP PARALLEL PRIVATE(I,MYID,NEIGHB)
   myid = omp_get_thread_num()
   neighb = myid - 1
   if (myid.eq.0) neighb = omp_get_num_threads()-1
   ...
   a(myid) = a(myid)*3.5
!$OMP BARRIER
   b(myid) = a(neighb) + c
   ...
!$OMP END PARALLEL
```



- Barrier required to force synchronisation on **a**

Critical sections

- A critical section is a block of code which can be executed by only one thread at a time.
- Can be used to protect updates to shared variables.

CRITICAL directive

- Syntax:

Fortran: `!$OMP CRITICAL`

block

`!$OMP END CRITICAL`

C/C++: `#pragma omp critical`

structured block

CRITICAL directive (cont)

Example: pushing and popping a task stack

```
!$OMP PARALLEL SHARED(STACK),PRIVATE(INEXT,INEW)
      ...
!$OMP CRITICAL
      inext = getnext(stack)
!$OMP END CRITICAL
      call work(inext,inew)
!$OMP CRITICAL
      if (inew .gt. 0) call putnew(inew,stack)
!$OMP END CRITICAL
      ...
!$OMP END PARALLEL
```

ATOMIC directive

- Used to protect a single update to a shared variable.
- Applies only to a single statement.
- Syntax:

Fortran: **!$OMP ATOMIC**

statement

where *statement* must have one of these forms:

x = *x op expr*, *x* = *expr op x*, *x* = *intr* (*x, expr*) or

x = *intr* (*expr, x*)

op is one of **+**, *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**

intr is one of **MAX**, **MIN**, **IAND**, **IOR** or **IEOR**

ATOMIC directive (cont)

C/C++: **`#pragma omp atomic`**
statement

where *statement* must have one of the forms:

x binop = *expr*, *x*`++`, `++`*x*, *x*`--`, or `--`*x*

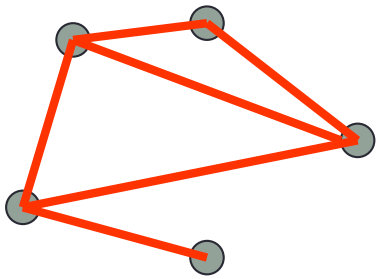
and *binop* is one of `+`, `*`, `-`, `/`, `&`, `^`, `<<`, or `>>`

- Note that the evaluation of *expr* is not atomic.
- May be more efficient than using CRITICAL directives, e.g. if different array elements can be protected separately.
- No interaction with CRITICAL directives

ATOMIC directive (cont)

Example (compute degree of each vertex in a graph):

```
#pragma omp parallel for
      for (j=0; j<nedges; j++){
#pragma omp atomic
         degree[edge[j].vertex1]++;
#pragma omp atomic
         degree[edge[j].vertex2]++;
      }
```



Lock routines

- Occasionally we may require more flexibility than is provided by CRITICAL directive.
- A lock is a special variable that may be *set* by a thread. No other thread may *set* the lock until the thread which set the lock has *unset* it.
- Setting a lock can either be blocking or non-blocking.
- A lock must be initialised before it is used, and may be destroyed when it is not longer required.
- Lock variables should not be used for any other purpose.

Lock routines - syntax

Fortran:

```
USE OMP_LIB
SUBROUTINE OMP_INIT_LOCK(OMP_LOCK_KIND var)
SUBROUTINE OMP_SET_LOCK(OMP_LOCK_KIND var)
LOGICAL FUNCTION OMP_TEST_LOCK(OMP_LOCK_KIND var)
SUBROUTINE OMP_UNSET_LOCK(OMP_LOCK_KIND var)
SUBROUTINE OMP_DESTROY_LOCK(OMP_LOCK_KIND var)
```

var should be an INTEGER of the same size as addresses (e.g. INTEGER*8 on a 64-bit machine)

OMP_LIB defines OMP_LOCK_KIND

Lock routines - syntax

C/C++:

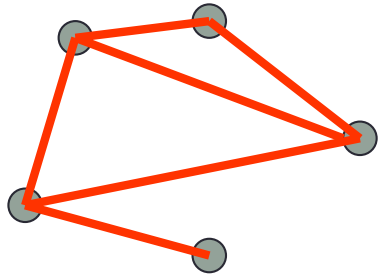
```
#include <omp.h>
  void omp_init_lock(omp_lock_t *lock);
  void omp_set_lock(omp_lock_t *lock);
  int omp_test_lock(omp_lock_t *lock);
  void omp_unset_lock(omp_lock_t *lock);
  void omp_destroy_lock(omp_lock_t *lock);
```

Lock example

Example (compute degree of each vertex in a graph):

```
for (i=0; i<nvertexes; i++){
  omp_init_lock(lockvar[i]);
}

#pragma omp parallel for
      for (j=0; j<nedges; j++){
         omp_set_lock(lockvar[edge[j].vertex1]);
         degree[edge[j].vertex1]++;
         omp_unset_lock(lockvar[edge[j].vertex1]);
         omp_set_lock(lockvar[edge[j].vertex2]);
         degree[edge[j].vertex2]++;
         omp_unset_lock(lockvar[edge[j].vertex2]);
       }
```



Brief history of OpenMP

- Historical lack of standardisation in shared memory directives.
  - each hardware vendor provided a different API
  - mainly directive based
  - almost all for Fortran
  - hard to write portable code

- OpenMP forum set up by Digital, IBM, Intel, KAI and SGI. Now includes most major vendors (and some academic organisations, including EPCC).

- OpenMP Fortran standard released October 1997, minor revision (1.1) in November 1999. Major revision (2.0) in November 2000.
- OpenMP C/C++ standard released October 1998. Major revision (2.0) in March 2002.

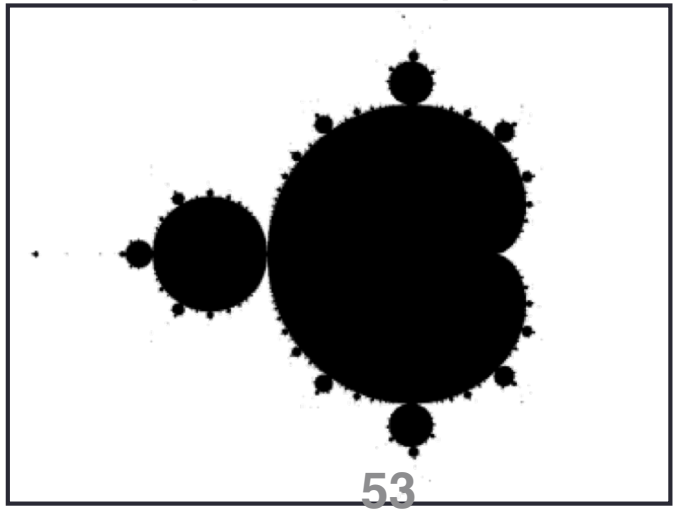
History (cont.)

- Combined OpenMP Fortran/C/C++ standard (2.5) released in May 2005.
  - no new features, but extensive rewriting and clarification
- Version 3.0 released in May 2008
  - new features, including tasks, better support for loop parallelism and nested parallelism
- Version 3.1 released in June 2011
  - corrections and some minor new features
  - most current compilers support at least this
- Version 4.0 released in July 2013
  - accelerator offloading, thread affinity, more task support,...
  - now in most implementations
- Version 4.5 released November 2015
  - corrections and a few new features
  - some full implementations ?

Exercise

Area of the Mandelbrot set

- Aim: introduction to using parallel regions.
- Estimate the area of the Mandelbrot set.
  - Generate a grid of 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!



Reusing this material