Introduction to OpenMP

Synchronisation

Nick Maclaren

nmm1@cam.ac.uk

September 2019

Summary

Facilities here are relevant to both SIMD and SPMD
A bit of a hodge–podge, so may be a little confusing

Unfortunately, no ‘right’ order to teach them in
It may be unclear why and when you need them

- Most reasons will be covered later

- May need serial code in a parallel region

E.g. to read some data from stdin

- May have some unavoidable data dependencies

Pass data between threads in a parallel region

Problems With This

Mainly performance and deadlock/livelock

- Performance best done by minimising such uses

And not synchronising all threads unless critical
Some guidelines mentioned as we describe facilities

Deadlock and livelock cause program failure
We cover only techniques that avoid them

- But you must follow the guidelines for safety

First need to explain deadlock and livelock

Deadlock

This is when two or more threads are waiting
and none can make progress until another does

- There are many ways it can occur

But it is easy to give some rules for avoiding it

One of the most common errors when using locks
That is why this course does not recommend them

Another common cause, but that is covered later
Under split parallel and work–sharing constructs

Livelock

Two or more threads are in an indefinite loop
In theory, this will always terminate, eventually

- The logic or scheduling means it doesn't

Or such loops sometimes become ridiculously slow

Problem is that all simple examples are unrealistic
Common with non–trivial inter–thread communication

- So we are going to keep it simple and stupid

Largely by minimising and simplifying communication

Avoiding Livelock

- You need to think in terms of the control flow

Specifically, indefinite looping, however it's done

Simple alternative code (e.g. IF) doesn't matter
Problem is number of times loops are repeated

- Avoid one thread's control depending on another's

That's overkill, but it's the only simple rule

- Don't assume anything about thread scheduling

Looping in one thread may stop another from running

What Is Communication?

Any way of passing information between threads
Locks, files, messages, signals or global data
And that includes any form of program state

- This course will cover mainly updating global data

But the same mechanism can also protect the others

Safe update of and access to shared objects

- Main ones are critical, master and single

atomic is covered later, but rarely useful

Horrible Warning (A)

- Execution order does NOT imply data consistency
Each synch. construct has wildly different rules
And many of them are seriously counter–intuitive

- To synchronise data, it’s best to use a barrier
Using flush constructs is much trickier

- This is a major, but MAJOR, ‘gotcha’
Will often fail as you increase the level of optimisation
Appear to work on some systems and fail on others

The Barrier Construct

All of the threads halt at this construct
When the last one reaches it, they all restart

Use it when you want all threads to be consistent

Fortran specification:

```
!$OMP BARRIER
```

Note that there is no !$OMP END BARRIER

C/C++ specification:

```
#pragma omp barrier
```

Implicit Barriers (1)

An implicit barrier executed by all threads at:

- A barrier construct
- Entry to and exit from a parallel construct
- Exit (ONLY) from a work–sharing construct
  I.e. DO/for, sections, single and workshare
  Except when using nowait (see later)

There are no barriers for any other constructs
master and critical are the main 'gotchas'

The Critical Construct (1)

Executes in each thread, one at a time (serially)
Essential bypass around the aliasing restrictions

- Warning: it is very easy to cause livelock

Nested use of it can also cause deadlock

- Warning: the specification is ambiguous

Complicated data use may not work

You can use it almost anywhere you want to
Probably even in serial code, though that's unclear
The next lecture will describe when you need it

The Critical Construct (2)

Fortran specification:

```
!$OMP CRITICAL ( <name> )
< structured block >
!$OMP END CRITICAL ( <name> )
```

The two <name>s must be the same, of course

C/C++ specification:

```
#pragma omp critical ( <name> )
< structured block >
```

The Critical Construct (3)

The (<name>) can be omitted, but I don't advise it
Unnamed criticals all use the same anonymous name

- The name is an external entity

Make it different from everything else like that:
procedures, modules, extern, COMMON, ...

- The only interlocking is between critical sections
  and then only between ones of the same name

- Will synchronise only directly visible, shared data

If doing anything non–trivial, use a barrier

Fortran Example

```
!$OMP PARALLEL DO
      DO index = 1 , limit
            CALL Fred ( index , this , that , the_other )
      END DO
!$OMP END PARALLEL DO


SUBROUTINE Fred ( index , this , that , the_other )
      . . .
      !$OMP CRITICAL ( write_to_stdout )
            WRITE (*,*) . . .
      !$OMP END CRITICAL ( write_to_stdout )
```

C/C++ Example

```
void fred ( int index , double this , double that )
    . . .
    #pragma omp critical ( write_to_stdout )
    {       /* Not essential, but recommended */
            printf ("d %f %f\n" , index , this , that) ;
    }       /* Not essential, but recommended */
}

#pragma omp parallel for private ( index )
for ( index = 0 ; index < limit; ++index )
    fred ( index , this , that  ) ;
```

The Master Construct (1)

The block is executed only in the master thread 0
The other threads effectively just skip over it

The master construct seems exactly equivalent to:

    if (omp_get_thread_num() == 0) ...

If you prefer to use this form, why not?

I have absolutely no idea why OpenMP provides it
Different specifications would be much more useful

But using it makes your intention a little clearer

The Master Construct (2)

Fortran specification:

```
!$OMP MASTER
        < structured block >
!$OMP END MASTER
```

C/C++ specification:

```
#pragma omp master
< structured block >
```

The Master Construct (3)

One very important use is for serialised I/O
Reading from stdin is best done like that
But many programs do I/O only in the master

- You can use master in a parallel region
It will restrict that code to executing on thread 0

- But be warned that it is not synchronised
Other threads will carry on running in parallel
I.e. executing any other code in the parallel region

- And it will NOT synchronise ANY data

Fortran Example

```
!$OMP PARALLEL DO
      DO index = 1 , limit
            CALL Fred ( index , this , that , the_other )
      END DO
!$OMP END PARALLEL DO

SUBROUTINE Fred ( index , this , that , the_other )
      . . .
      < You may want a barrier here >
      !$OMP MASTER
            WRITE ( * , * ) . . .
      !$OMP END MASTER
      < You probably want a barrier here >
```

C/C++ Example

```
void fred ( int index , double this , double that )
      . . .
      < You may want a barrier here >
      #pragma omp master
      {     /* Not essential, but recommended */
            printf ("d %f %f\n", index, this, that) ;
      }     /* Not essential, but recommended */
      < You probably want a barrier here >
}

#pragma omp parallel for private ( index )
for (index = 0 ; index < limit; ++index)
      fred ( index , this , that ) ;
```

The Single Construct (1)

- This IS an ordinary work–sharing construct
Except that one thread does all the work!

- What it does is to execute one thread only
The other threads effectively just skip over it

Which thread? That's unspecified and unpredictable

It doesn't matter for a lot of synchronised code
A very useful facility for I/O and similar uses

The Single Construct (2)

Fortran specification:

```
!$OMP SINGLE [ clauses ]
      < structured block >
!$OMP END SINGLE
```

C / C++ specification:

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

Copyprivate

The copyprivate clause is a little like firstprivate

- It can be used only on single directives

It copies that thread's value to all threads on exit

- Must be threadprivate or on parallel directive

Fortran allocatable variables need 3.0

- In Fortran, it is put on the END SINGLE directive

Copyprivate (Fortran)

```
REAL ( KIND = KIND ( 0.0D0 ) ) :: parameter
!$OMP PARALLEL private ( parameter )
      !$OMP SINGLE
            READ * , parameter
      !$OMP END SINGLE copyprivate ( parameter )
      < can now use parameter in all threads >
!$OMP END PARALLEL
```

Copyprivate (C/C++)

```
static double parameter ;
#pragma omp parallel private ( parameter )
{
        #pragma omp single copyprivate ( parameter )
        {
                scanf ( "%f \n" , & parameter ) ;
        }
        < can now use parameter in all threads >
}
```

Horrible Warning (B)

Remember shared and parallel aliasing problems?
I.e. races between work–sharing and outside

- They apply to master and critical, too

Just as they do to the single construct

- This is a particular 'gotcha' with master

Executed in thread 0, but not executed serially

You may need to add extra barriers to stop this

Work-Sharing Critical

```
!$OMP CRITICAL ( <name> )
      < structured block >
!$OMP END CRITICAL ( <name> )
!$OMP BARRIER

#pragma omp critical ( <name> )
< structured block >
#pragma omp barrier
```

Work-Sharing Master

```
!$OMP MASTER
        < structured block >
!$OMP END MASTER
!$OMP BARRIER
```

```
#pragma omp master
< structured block >
#pragma omp barrier
```

Performance

- Avoid using critical where performance matters

It necessarily serialises all of the threads

- Also master, single and atomic (see later)

- Don't worry about code that is rarely executed

Initialisation, termination, error handling and so on

If you can't avoid doing that, for any reason

- Don't assume anything about thread scheduling

Split Directives (1)

Compilers may create threads at a parallel directive
And destroy them at the end of the region

- If they do, fewer parallel regions is better

Several work–sharing regions inside each one

- But this is significantly trickier to use

Don't do it unless it is fairly important

- Ask how often are parallel directives executed?

If not very often (relatively), then don't bother

Split Directives (2)

Technique is to use them rather like simple SPMD

> Parallel directive
>> Work−sharing construct
>> Work−sharing construct
>> . . .
>
> End parallel region

Work−sharing constructs can be fairly general
Not just what OpenMP calls work−sharing constructs

Split Directives (3)

Open code is executed in all threads, in parallel

DO/for/sections distribute work across threads

master/single execute in only one thread

critical executes in each thread, serially

barrier synchronises across all threads

- But it's very easy to make a mistake doing this

Avoiding Deadlock (1)

- Top level is a sequence of parallel constructs
Anything not in one is serial code, of course

- Each has a sequence of work–sharing constructs
Anything not in one is executed in all threads
barrier is a work–sharing construct in this sense

- Each has a sequence of critical constructs
Anything not in one is executed in parallel

- Each is sequence of code and atomic constructs

Avoiding Deadlock (2)

Within a single parallel region,
you must match potential barrier constructs
I.e. barrier, DO/for, sections, single and workshare

All threads execute exactly the same sequence
E.g. they all execute DO/for, then barrier, then ...

Ignore master, critical and atomic constructs

What will happen if you get this wrong? It's undefined
Your program may hang or may go weirdly wrong

More on Barriers (1)

This will NOT work reliably – but it may appear to

```
#pragma omp parallel
{
        double av = 0.0 , var = 0.0 ;
#pragma omp for reduction ( + : av )
        for ( i = 0 ; i < size ; ++ i ) av += data [ i ] ;
#pragma omp master
        av /= size ;
#pragma omp for reduction ( + : var )
        for ( i = 0 ; i < size ; ++ i )
                var += ( data [ i ] – av ) * ( data [ i ] – av )
}
```

More on Barriers (2)

There are many, many variations on that ‘gotcha’
None of them are obvious when looking at the code

More in the next lecture – with the rule KISS, KISS

Work–sharing forms of master and critical?
Just follow them by a barrier construct
Examples were given earlier, so look back for them

- Use exactly like another work–sharing construct

More on Barriers (3)

- Consider adding extra barrier constructs
E.g. before all of your work–sharing constructs
They then become fully synchronised forms

This can make both debugging and tuning easier
May slow your program down or may speed it up!

- It's a good idea to use these for SIMD work
Can remove barriers later as part of tuning

Synchronised Constructs (1)

These are fully synchronised on entry and exit
Exactly like the combined forms (parallel for etc.)

```
!$OMP BARRIER
!$OMP DO    ! or SINGLE or SECTIONS
        < structured block >
!$OMP END DO

#pragma omp barrier
#pragma omp for    /* or single or sections */
        < structured block >
```

DO/for/sections share the work across threads
single executes only one of the threads

Synchronised Constructs (2)

```
!$OMP BARRIER
!$OMP MASTER    ! or CRITICAL
    < structured block >
!$OMP END MASTER
!$OMP BARRIER

#pragma omp barrier
#pragma omp master    /* or critical */
    < structured block >
#pragma omp barrier
```

master executes only the master thread (0)
critical executes each in unpredictable order