Introduction to OpenMP

Basics and Simple SIMD

Nick Maclaren

nmm1@cam.ac.uk

September 2019

Terminology

I am abusing the term SIMD – tough
Strictly, it refers to a type of parallel hardware

In this course, I mean a type of program design

KISS

That stands for Keep It Simple and Stupid
Kelly Johnson, lead engineer at The Skunkworks

It should be written above every programmer's desk
As I tell myself every time I shoot myself in the foot!

- It's rule number one for OpenMP use

Actually, it's one key to all parallel programming

Problems increase exponentially with complexity
That sounds ridiculous, but it's true

Fortran

Almost all Fortran aspects omitted for MPhil
Included in the notes, for those interested
And feel free to use it for practicals

There are some examples in Fortran
Only ones illustrating a simple point
Please ask if you con't understand the point

Fortran For SIMD (1)

Fortran 90 rules, OK?

No competition in either coding or optimisation
Followed by Fortran 77, then C++ and lastly C

- Start with clean, clear whole array expressions
- Then call BLAS or expand calls manually

Unfortunately, that relies on an excellent compiler
And, to a first approximation, none of them are

Fortran For SIMD (2)

Compilers may not parallelise array expressions
They are better at parallelising DO loops

Also why is MATMUL sometimes very slow?
The compiler could call D/ZGEMM itself

- There are sometimes options to do so

E.g. gfortran –external–blas

But expanding such code is easy and reliable
So using it makes debugging a lot easier

Fortran Syntax (1)

This course covers modern free format only
As always in Fortran, case is ignored entirely
Leading spaces are also ignored in free format

Fortran directives take the form of comments
Lines starting !$OMP and a space
Most of them have start and end forms

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

Fortran Syntax (2)

Directives may be continued, like Fortran statements

```
!$OMP PARALLEL DO          &
     !$OMP [ clauses ]     ! Note the !$OMP
< structured block >
!$OMP END PARALLEL DO

!$OMP PARALLEL DO          &
     !$OMP & [ clauses ]     ! Note the second &
< structured block >
!$OMP END PARALLEL DO
```

Fortran Syntax (3)

You can also write PARALLELDO etc.

You can omit END PARALLEL DO and END DO
But not any other END directive

- I don't recommend doing either – it's poor style

You may see them in other people's code

Fortran Structured Block

Sequence of whole Fortran statements or constructs

Always entered at the top and left at the bottom
No branching in or out, however it is done
Including RETURN, GOTO, CYCLE, EXIT,
    END=, EOR= and ERR=

Allowed to branch around within structured block
You can execute STOP within one, too

C++ vs C

This course covers only the C subset of C++
But it will describe C++ where that is relevant
There are two main reasons for this:

- Version 3.0 needed for serious C++ support

- A lot of extra complications and 'gotchas'
Compilers are likely to be incompatible or buggy

- Use either C or C++ for doing the practicals
More detailed guidelines are given in a moment

C++ STL

That is the Standard Template Library

- Very serial compared with Fortran
But it does allow some scope for parallelisation

- Not supported by OpenMP until version 3.0
Most unclear how much of that will work – see later

C++ Assumptions

This course assumes the following restrictions:

- In the serial code, not affected by OpenMP
  You can use all of C++, including the STL

- In any parallel region or OpenMP construct
  Don't use any constructors or destructors
  Don't update containers or iterators

- Parallelise only C–style arrays and loops

$\Rightarrow$ All specimen answers are in C only

Why?

The C++ standard is very ambiguous here

Constructors and destructors may have side–effects
'Element' and iterator actions may use the container
const methods are allowed to update it (mutable)
None of the above is properly specified

There are some empirically safe interpretations
This course gives a very simplified form of them

All of this applies to other thread interfaces, too

Safe C++ Use

Use basicstring, vector, deque and array (only)
Based on built–in integer, floating–point or complex
Can use them as normal outside parallel regions

Use front( ) etc. to get a C pointer to first element
Using the container directly is covered later

Inside parallel regions, use that pointer only
Access elements using only operators '*' and '[ ]'

- Do not update the container in any way

And, if in any doubt, don't use the container at all

C/C++ Syntax (1)

C/C++ directives take the form of pragmas
They are all lines starting '#pragma omp'

As always in C/C++, case is significant
Leading spaces are also usually ignored

```
#pragma omp parallel for [ clauses ]
< structured block >
```

Note that there is no end form, unlike Fortran
Critical that the block really is a block

C/C++ Syntax (2)

Warning: watch out for macro expansions
Occasionally can generate a statement sequence

Can continue lines as normal in C/C++
End the line with a backslash (\)

```
            #pragma parallel for        \
                    [ clauses ]
```

C/C++ Structured Block (1)

I recommend using one of three forms:

- A for statement
- A compound statement: {...; ...}
- A simple expression statement
  Includes a void function call and assignment

Several more allowed, but those are the sanest

Always entered at the top and left at the bottom
No branching in or out, however it is done
Including return, goto, catch/throw,
setjmp/longjmp, raise/abort/signal etc.

C/C++ Structured Block (2)

Note that this applies to all functions called, too
Including ones called implicitly by C++
Called functions must return to the structured block

Allowed to branch around within structured block
E.g. catch/throw is OK, if entirely inside the block
You can call exit() within one, too

- Clean programs should have little trouble

Chaos awaits if you break these rules

Course Conventions (1)

We will often use C/C++ case conventions
As well as spaces between all keywords

- This works for Fortran as well as C and C++

!$OMP and #pragma omp are called sentinels

Clauses are separated by commas, in any order
Will describe their syntax as we use them
Syntax is almost entirely language–independent

Names and expressions match the language

Course Conventions (2)

Examples of using OpenMP are in both Fortran and C
C and C++ are almost identical in their OpenMP use

- Some are given in only one of Fortran and C

They are all intended to illustrate a single point
Ignore their detailed syntax, as it's not important

- Please interrupt if you can't follow them

Library

There is a runtime library of auxiliary routines

C/C++: #include <omp.h>

Fortran: USE OMP_LIB
or: INCLUDE 'omp_lib.h'
The compiler chooses which, not the user

The most useful routines are covered as we use them
We shall mention the three most important here

Omp_get_wtime (1)

This returns the elapsed (wall–clock) time in seconds
As a double precision floating–point result

The time is since some arbitrary time in the past

- It is guaranteed to be fixed for one execution
- It is guaranteed to be the same for all threads

Omp_get_wtime (2)

Fortran, C++ and C examples:

```
    WRITE ( * , " ( 'Time taken ' , F0.3 , ' seconds' ) " )   &
          omp_get_wtime ( )
```

{\cyan C++} and {\cyan C} examples: \break
\begin{myverb}

```
     std::cout << "Time taken " << omp_get_wtime ( ) <<
          << " seconds" << std::endl ;

     printf ( "Time taken %.3f seconds\n" ,
          omp_get_wtime ( ) ) ;
```

Omp_get_thread_num

This returns the thread number being executed

- It is a default integer from 0 upwards

Fortran, C++ and C examples:

```
    WRITE ( * , " ( 'Current thread ' , I0 ) " )    &
        omp_get_thread_num ( )
```

{\cyan C++} and {\cyan C} examples: \break
\begin{myverb}

```
    std::cout << "Current thread " <<
        omp_get_thread_num ( ) << std::endl ;

   printf ( "Current thread %d\n" , omp_get_thread_num ( ) ) ;
```

Omp_get_num_threads

This returns the number of threads being used

- It is a default integer from 1 upwards

Fortran, C++ and C examples:

```
    WRITE ( * , " ( 'Number of threads ' , I0 ) " )    &
        omp_get_num_threads ( )
```

{\cyan C++} and {\cyan C} examples: \break
\begin{myverb}

```
    std::cout << "Number of threads " <<
        omp_get_num_threads ( ) << std::endl ;

   printf ( "Number of threads %d\n" , omp_get_num_threads ( )
```

Warning: Oversimplication

We are going to skip a lot of essential details
Just going to show the basics of OpenMP usage

For now, don’t worry if there are loose ends
We return and cover these topics in more detail

Three essentials to OpenMP parallelism:

- Establishing a team of threads
- Saying if data is shared or private
- Sharing out the work between threads

The Parallel Directive

This introduces a parallel region
Syntax: sentinel parallel [clauses]

Fortran:

```
!$OMP PARALLEL
        < code of structured block >
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel
{
        < code of structured block >
}
```

How It Works

When thread A encounters a parallel directive

- It logically creates a team of sub–threads

- It then becomes thread 0 in the new team

That thread is also called the master thread

- The team then executes the structured block

When the structured block has completed execution

- The sub–threads collapse back to thread A

A Useful Trick

Difficulty working out what OpenMP is doing?

- Try printing out omp_get_thread_num()
You can then see which thread is being executed

- Plus any other useful data at the same time
E.g. a tag indicating which print statement
In DO/for loops, the index as well

Yes, I did that, to check I had understood
- But you shouldn't really do I/O in parallel

Data Environment

- The data environment is also critical

But the basic principles are very simple

- Variables are either shared or private

⇒ Warning: you need to keep them separate

Outside all parallel regions, very little difference
Standard language rules for serial execution apply
Everything is executing in master thread 0

Most differences apply only within parallel regions

Shared Data

- Shared means global across the program
Same name means same location in all threads
Can pass pointers from one thread to another

- Don't update in one and access in another
Without appropriate synchronisation between actions
I.e. can use in parallel if read–only in all threads

- Such race conditions main cause of obscure bugs
Not just OpenMP, but any shared memory language

Arrays and Structures

Those rules apply to base elements, not whole arrays
Can update different elements of a shared array

- But watch out for Fortran's aliasing rules

- Unclear if applies to members of structures

Very complicated and language dependent
And C standard is hopelessly inconsistent here

So KISS – Keep It Simple and Stupid
Clean, simple code will have no problems

Private Data

Private means each thread has a separate copy
Same name means different location in each thread

- Don't pass pointers to data to other threads
- Don't set shared pointers to private data

⇒ This applies even to global master thread 0

- The above is not like POSIX's rules

OpenMP is more restrictive for better optimisation

Default Rules

Start by assuming that everything is shared

The following variables are private:

- Indices in OpenMP–parallelised DO/for–loops
- C automatic vars declared inside parallel regions
- Fortran DO–loop, implied–DO and a bit more

That's often enough on its own for the simplest cases
You can override the defaults, when you need to
By adding clauses to the parallel directive

Specifying Usage

Specify shared by shared(<names>)
But private(<names>) is far more often needed

```
!$OMP PARALLEL shared ( array ) , private ( x , y , z , i , j , k )
        < code of structured block >
!$OMP END PARALLEL
```

Clauses are identical for Fortran and C/C++

- It's good practice to specify everything explicitly

But it makes no difference to the code generated

Default(none)

You can set the default to effectively unset

- Should give a diagnostic if you forget to declare

Very like the Fortran statement IMPLICIT NONE

```
!$OMP PARALLEL default ( none ) , shared ( array ) ,    &
              !$OMP private ( i , j , k )
       < code of structured block >
!$OMP END PARALLEL
```

Only allowed in combination with parallel directives
Some variables have defaults even if it is used

- Usually omitted, because of space on slides

Parallelising Loops (1)

This is the main SIMD work–sharing directive

- Obviously, each iteration must be independent
I.e. the order must not matter in any way

Similar rules to Fortran DO CONCURRENT
Next slide describes what that means

⇒ It's also relevant to C/C++ people
Same rules apply, unlike for ordinary for loops

Parallelising Loops (2)

What does Fortran DO CONCURRENT require?

- Mainly that no iteration may affect any other
And it is important to stress in in any way
It's not just setting a variable used in a later one

- Includes calling any procedures with state
Including all I/O, random numbers, and so on

Not hard, but be very cautious as you start coding

Fortran Form

Syntax: sentinel DO [clauses]
Loop variable best declared as private

```
!$OMP DO PRIVATE(<var>)
        DO <var> = <loop control>
                < structured block >
        END DO
!$OMP END DO
```

C/C++ Form

Syntax: sentinel for [clauses]
Loop variable best declared as private

```
#pragma omp for private(<var>)
        for ( <loop control> )
                < structured block >
```

- <loop control> must be very Fortran–like

Most people do that anyway, so little problem
Will describe rules later, for C/C++–only people

Combining Them (1)

Also combined parallel and work−sharing
You can use any clauses that are valid on either

Fortran:

```
!$OMP PARALLEL DO [clauses]
        < code of structured block >
!$OMP END PARALLEL DO
```

C/C++:

```
#pragma omp parallel for [clauses]
    < code of structured block >
```

Combining Them (2)

For now, we shall use the combined forms
We shall come back to the split forms later

That's mainly for convenience in the slides
Having two directives instead of one is messy

- Also, the split forms are very deceptive
There are a lot of subtle gotchas to avoid

There is more you can do with the split forms
But, in the simple cases, both are equivalent

Be Warned!

All threads execute all of a parallel block
Unless controlled by other directives – see later

- In particular, apparently serial code is

Don’t update any shared variables in such code
Reading them and calling procedures are fine

- It’s easier to use the combined forms safely

Always start by using them, where that is feasible

Split Directives

The following code is badly broken:

```
!$OMP PARALLEL
  !$OMP DO PRIVATE(i), REDUCTION(+:av)
    DO i = 1,size
      av = av+values(i)
    END DO
  !$OMP END DO
  av = av/size     ! Executed once on each thread
  !$OMP DO PRIVATE(i)
    DO i = 1,size
      values(i) = values(i)/av
    END DO
  !$OMP END DO
!$OMP END PARALLEL
```

Semi-Realistic Example

Let’s use matrix addition as an example

- This is just showing how OpenMP is used

You can do it in one line in Fortran 90

We need to compile using commands like:

```
ifort/icc –O3 –ip –openmp ...
gfortran/gcc –O3 –fopenmp ...
```

Fortran Example

```
SUBROUTINE add (left, right, out)
   REAL ( KIND=dp ) , INTENT ( IN ) :: left ( : , : ) , right ( : , : )
    REAL ( KIND=dp ) , INTENT ( OUT ) :: out ( : , : )
    INTEGER :: i , j

!$OMP PARALLEL DO
    DO j = 1 , UBOUND ( out , 2 )
        DO i = 1 , UBOUND ( out , 1 )
            out ( i , j ) = left ( i , j ) + right ( i , j )
        END DO
    END DO
!$OMP END PARALLEL DO

END SUBROUTINE add
```

C/C++ Example

```
void add ( const double * left , const double * right ,
            double * out , int size ) {
      int i , j ;

#pragma omp parallel for private ( j )
      for ( i = 0 ; i < size ; ++ i )
            for ( j = 0 ; j < size ; ++ j ) {
                  out [ j + i * size ] =
                        left [ j + i * size ] + right [ j + i * size ] ;
            }
}
```

So Far, So Good

Unfortunately, adding the directives is the easy bit
You now have enough information to start coding

- There are a couple of very simple practicals

- Then a few more details of using OpenMP

- Then some more SIMD practicals

Actual Examples

All details of what to do are on the handouts provided
Do check the results and look at the times
Errors often show as wrong results or poor times

- Do exactly what the instructions tell you to

Intended to show specific problems and solutions

- You can’t match the tuned libraries

They use blocked algorithms – better for caching