Introduction to OpenMP

Basics and Simple SIMD

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

2.1 Introduction

KISS stands for *Keep It Simple and Stupid*, which was apparently coined by Kelly Johnson, lead engineer at Lockheed's Skunkworks. It is an ancient and excellent engineering principle; related aphorisms have been coined by C.A.R. Hoare and others. It should be written above every programmer's desk, as I tell myself every time I shoot myself in the foot!

In particular, it is rule number one for OpenMP use, and is one key to all parallel programming. With those, the problems increase exponentially with the complexity of the program. That may sound ridiculous, but unfortunately is true.

2.2 Language Support

2.2.1 Fortran for SIMD

For SIMD, Fortran 90 (i.e. modern Fortran) is by far the best language, and there is no competition in either the ease of coding or the level of optimisation that can be obtained. It is followed by Fortran 77, then C++ and lastly C. The general rule is to start with clean, clear whole array expressions, then call the *BLAS*, when appropriate, or expand calls manually.

Unfortunately, that relies on an excellent compiler and, to a first approximation, none of them are. Compilers only sometimes parallelise array expressions – they are better at parallelising `DO` loops. Also, `MATMUL` is sometimes very slow, which is ridiculous; the compiler could call `D/ZGEMM` itself, and there are sometimes options to do so, such as:

```
gfortran -external-blas
```

But expanding clean and simple array expression code by hand is both easy and reliable, so start by writing them as array expressions – that will make debugging and tuning a lot easier.

2.2.2 Fortran Syntax and Use

This course covers modern free format Fortran only, though OpenMP works perfectly well using with the Fortran 77 subset. As always in Fortran, case is ignored entirely outside strings, so you can use any convention you like. Leading spaces are ignored in free format, and this applies even to OpenMP directives. These take the form of comments –

they are lines starting `!$OMP` and a space – and most of them have start and end forms. For example:

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

Directives may be continued, rather like Fortran statements, with both single `&` and matched `&` forms, but the continuation line also starts with `!$OMP`. For example:

```
!$OMP PARALLEL DO        &
    !$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
```

You can also write `PARALLELDO` and so on, but the use of composite keywords is being discouraged by the Fortran standards committee. You can also can omit `END DO` and `END PARALLEL DO`, but no other `END` directive. I do not recommend doing either, on the grounds of poor style and less checking, but you may see those in other people's code.

A *structured block* in Fortran is a sequence of whole Fortran statements or constructs, which is always entered at the top and left at the bottom. There must be no branching in or out of the block, **however** it is done. Facilities to watch out for include `RETURN`, `GOTO`, `CYCLE`, `EXIT`, `END=`, `EOR=` and `ERR=`. You are allowed to branch around **within** the structured block, including within procedures called from it, and can even execute `STOP` within one.

2.2.3 C++ Coverage

The STL is the Standard Template Library. It is very serial compared with Fortran, though it theoretically allows some scope for parallelisation. However, the relevant aspects are not supported by OpenMP until version 3.0 and, for various reasons, it is unclear how much of it will actually work – some more detail is given later.

This course covers only the C subset of C++, though it mentions C++ where that is relevant. There are two main reasons for this:

- Version 3.0 is needed for serious C++ support.
- There are a **lot** of extra complications and 'gotchas'.

You should use either C or C++ for doing the practicals. 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, you should not use any constructors or destructors, or containers or iterators from the STL.

- You parallelise only C-style arrays and loops.

All of the specimen answers are in C only, for the above reasons, but work when compiled as C++ code.

Why is this? The C++ standard is very ambiguous over how the STL behaves with threading. Constructors and destructors may have side-effects, 'element' and iterator actions may access the container and, in theory, even `const` methods are **allowed** to update it by using `mutable`. None of this is properly specified.

However, there are some empirically safe interpretations, and this course gives a very simplified form of them. All of this applies to other thread interfaces, such as POSIX's or even C++'s, too. C++ code that will be used for parallel working should use `basic_string`, `vector`, `deque` and `array`, and only those. They can be based on built-in integer, floating-point or complex types (**not** including `vector<bool>`). Outside parallel regions, you can use those as normal, but use `front()` to get a C pointer to the first element before entering a parallel region; using the container directly is covered later.

Inside parallel regions, use only that pointer, and access elements using only operators '`*`' and '`[]`'. Do not update the container in any way and, if in **any** doubt, do not use the container at all inside parallel regions.

2.2.4 C and C++ Syntax and Use

C/C++ directives take the form of pragmas, and are all lines starting '`#pragma omp`'. As always in C/C++, case is significant, so you must get it right. Leading spaces are usually ignored, but the C/C++ language has bizarre rules for line handling in the preprocessor, so do not try to be clever.

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

Note that there is no end form, unlike in Fortran, so it is **critical** that the block really is a block and not just a sequence of statements. You should watch out for macro expansions, especially ones from application library headers, because they can occasionally generate a statement sequence.

You can continue lines as normal for C/C++ preprocessor lines, by ending the line with a backslash (\). Note that means really ending the line – trailing spaces will often cause it to fail, which is a real pain with some editors.

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

I recommend using one of three forms for a C/C++ structured block:

- A `for` statement.
- A compound statement: `{...; ...}`.
- A simple expression statement, which includes a void function call and assignments.

Several more are allowed, but those are the sanest, and are enough for all reasonable use; if in doubt, put matching curly braces around the block, because the compiler will simply ignore any surplus matched pairs of them.

A structured block must always be entered at the top and left at the bottom; it must have no branching in or out, **however** it is done, including by `return`, `goto`, `catch/throw`, `setjmp/longjmp`, `raise/abort/signal` or anything else. Note that this applies to all functions called, too, including ones called implicitly by C++ (e.g. constructors and destructors). All functions the structured block calls functions must return to the block.

Your code is allowed to branch around **within** the structured block; e.g. `catch/throw` is fine, if entirely inside the block or a procedure called from it. You can call `exit()` within one, too, but should not use any other form of termination. Clean programs should have little trouble, but chaos awaits if you break these rules.

2.2.5 Course Conventions

The course will often use C/C++ case conventions as well as spaces between all keywords, because that works for Fortran as well as C and C++.

`!$OMP` and `#pragma omp` are called sentinels in OpenMP.

The *clauses* on directives provide extra specification and are separated by commas, in any order. The course will describe their syntax as it uses them, rather than list them. Their syntax is almost entirely language-independent, but the names and expressions in clauses match the language you are using: Fortran, C or C++.

Examples of using OpenMP will usually be 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, and those are all intended to illustrate a single point. In such cases, users of the other language can ignore their detailed syntax, as it is not important.

2.2.6 The Library

There is a runtime library of auxiliary routines , which is included into C/C++ by:

```
#include <omp.h>
```

Fortran is trickier, because there are two possible forms, and the compiler chooses which it supports, **not** the user:

```
USE OMP_LIB
```

or:

```
INCLUDE 'omp_lib.h'
```

The most useful routines are covered as we use them, but it is worth mentioning the two most important here.

`Omp_get_wtime` returns the elapsed (a.k.a. wall-clock, real or physical) time in seconds, as a double precision floating-point result. The value is since some arbitrary time in the past, but:

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

Examples in Fortran, C++ and C are:

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

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

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

`Omp_get_thread_num` returns the thread number being executed, as a default integer from zero up to the number of threads minus one. Examples in Fortran, C++ and C are:

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

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

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

2.3 Using OpenMP

2.3.1 Basics of OpenMP

Warning: this is an oversimplication.

We are now going to skip a lot of essential details. in order to just show the basics of OpenMP usage. For now, do not worry if there are loose ends, because we return and cover these topics in more detail. There are three essentials to OpenMP parallelism:

- Establishing a team of threads.
- Specifying whether data is shared or private.
- Sharing out the work between threads.

The `parallel` directive introduces a parallel region, and has the syntax:

Fortran example:

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

C/C++ example:

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

When thread A encounters a parallel directive, it does the following:

- It logically creates a team of sub-threads, though it may actually just associate itself with existing ones.

- It then becomes thread zero in the new team; thread zero is also called the *master* thread.
- All members of the team then execute the structured block, in parallel.

When the structured block has completed execution, the sub-threads collapse back to thread A, which continues.

A useful trick if you have difficulty working out what OpenMP is doing, is to print out `omp_get_thread_num()` in suitable places, and you can then see which thread is being executed. You should also print any other useful data at the same time, such as a tag indicating which print statement or, in `DO`/`for` loops, the index as well.

I did that when learning OpenMP, to check I had understood. You should not really do I/O in parallel, but it works tolerably well for such diagnostics; later on, we see how to do it properly.

2.3.2 Data Environment

Specifying the data environment is also critical, but the basic principles are very simple. Variables are either *shared* or *private*, and **you** need to keep them separate – OpenMP will assume that you have done it correctly.

Outside all parallel regions, there is very little difference between OpenMP and serial use, and the standard language rules for serial execution apply. Everything is executing in master thread zero, though it is not yet set up as such. Most differences apply only within parallel regions.

Shared means that the object is global across the program, and the same name means the same location in all threads. You can even pass pointers from one thread to another.

- Do not update in one thread and access in another, without using appropriate synchronisation between the actions.

You can use shared data in parallel (i.e. in separate threads, unsynchronised) only if it is read and not updated in **all** of the threads. Failing to follow this rule is called a *race condition*, and is the main cause of obscure bugs, not just in OpenMP, but in any shared memory language.

Those rules apply to base elements, not whole arrays, and you can update different elements of a shared array. Unfortunately, OpenMP does not specify that well, so each language has some issues.

You must be careful to obey Fortran's aliasing rules, as they apply to whole arrays in Fortran. If you break them, an optimising compiler may generate broken code – which will still be **your** bug!

In all the languages, it is unclear if this applies to members of structures, and this is a **very** complicated and language dependent area; worse, the C standard is hopelessly inconsistent about this. So KISS – *Keep It Simple and Stupid* – clean, simple code will have no problems, but 'clever' code may fail horribly.

Private means each thread has a separate copy, and the same name means a different location in each thread. There are some strict rules to make this work:

- Do not pass pointers to private data to other threads.
- Do not set shared pointers to private data.

And, be warned, this applies even to the global master thread zero. These are not like POSIX's rules; OpenMP is more restrictive to allow compilers a better chance at optimisation.

2.3.3 Default Rules

The default rules are fairly simple. Start by assuming that everything is shared, but the following variables are private:

- Indices in OpenMP-parallelised `DO`/`for`-loops.
- C automatic variables declared inside parallel regions, but **not** Fortran local variables without `SAVE` or initialisation; those use the normal default rules.
- Fortran `DO`-loop, implied-DO and `FORALL` indices, but this does **not** apply to C/C++..

That is often enough on its own for the simplest cases, though you can override the defaults, when you need to, by adding clauses to the parallel directive. You can specify shared by the clause `shared(<names>)`, but the clause `private(<names>)` is far more often needed. For example:

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

The clauses are identical for Fortran and C/C++, except that the variable names follow the language rules. It is good practice to specify everything explicitly, but it makes no difference to the code generated.

You can set the default to being effectively unset, which should give a diagnostic if you forget to declare a variable, though some cases will escape. This is very like the Fortran statement `IMPLICIT NONE` and, like that, its use is strongly recommended. It is only allowed in parallel directives (i.e. not in worksharing directives that are not combined with `parallel`), and some variables will be given default settings even if it is used. For example:

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

2.3.4 Parallelising Loops

The `DO`/`for` directive is the main SIMD work-sharing directive. Obviously, each iteration must be independent; i.e. the order must not matter in any way. This implies similar rules to Fortran `DO CONCURRENT`, but it is also relevant to C/C++ programmers – the same rules as Fortran apply, unlike for ordinary `for` loops in C/C++.

- But 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 is not not just setting a variable in one iteration and using in a later one. It also includes calling any procedures with state, including all I/O, most random numbers, and so on. It is not hard, but be very cautious as you start coding, especially if you are a C/C++ programmer.

Fortran and C/C++ use very similar syntax, with a different keyword. In both, it is best to declare the loop variable as private, for clarity, though it is not needed for OpenMP.

Fortran form: *sentinel* `DO` [*clauses*]

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

C/C++ form: *sentinel* `for` [*clauses*]

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

In C/C++, the `<loop control>` must be very Fortran-like; most people do that anyway, so there is little problem in practice. We will describe the exact rules later, for C/C++ programmers who know no Fortran.

2.3.5 Combined Directives

There also combined parallel and work-sharing directives, and you can use any clauses that are valid on either. For example, in Fortran:

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

or in C/C++:

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

For now, we shall use the combined forms, and we shall come back to the split forms later. That is mainly for convenience in the slides; having two directives instead of one is messy. Also, the split forms are very deceptive, and there are a lot of subtle problems to avoid, but there is more you can do with them. In the simple cases, both are equivalent.

Be Warned: all of the threads execute all of a parallel block, unless controlled by other directives (covered later); in particular, apparently serial code is.

You must not update any shared variables in such code, though reading them and calling procedures are fine, but not passing them as arguments to be updated. Because of this,

it is easier to use the combined forms safely, so always start by using them, where that is feasible. For example, the following code is badly broken:

```
!$OMP PARALLEL
! REDUCTION is covered in the next lecture.
!$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
```

2.3.6 A Semi-Realistic Example

Let us use matrix addition as an example. This is just to show how OpenMP is used, though you can do it in one line in Fortran 90.

We need to compile using commands like:

`ifort -O3 -ip -openmp ...`	Intel Fortran
`icc -O3 -ip -openmp ...`	Intel C
`gfortran -O3 -fopenmp ...`	GNU Fortran
`gcc -O3 -fopenmp ...`	GNU C

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];
        }
}
```

2.3.7 Practicals

Unfortunately, adding the directives is the easy bit, though you now have enough information to start coding. There are a couple of very simple practicals, using the above, then the next lecture will give a few more details of using OpenMP, followed by some more SIMD practicals.

All details of what to do are on the handouts provided, but please note:

- Check the results and look at the times, because different results and poor times are the most likely result of errors.
- Do exactly what the instructions tell you to, because they are intended to show specific problems and solutions.
- Do not worry that you cannot match the tuned libraries, because they use blocked algorithms, which are better for caching.