Introduction to OpenMP

Critical Guidelines

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

7.1 Some Sordid Details

7.1.1 Apologia and Refrain

The previous lectures were an oversimplification, and roughly translate to "*This is a footgun; pull its trigger to see how it works*". Including even critical warnings was just too confusing, so they were separated out and included here, even though a few were still left in. There are also a few forward references to some features that will be described in next lecture.

This lecture is mainly what you need to know, but do not want to, and here is a reminder of the picture we saw at the start:

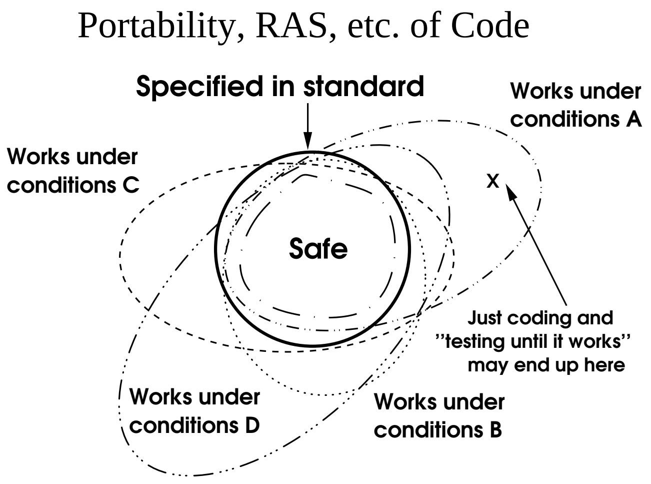


Figure 1.1

This lecture may discourage you from using OpenMP, but that is not its purpose, and the next lecture describes when, why and how to use OpenMP. It is much trickier to use than MPI, but this course has described how to use it safely and simply, and it does have some significant advantages over MPI. The main principle is to keep *gotchas* out of parallel regions because, outside all of them, you are programming serially and the usual language rules apply. Unfortunately, this lecture is about lots and lots of *gotchas*.

Remember that *KISS* stands for *Keep It Simple and Stupid*, problems increase exponentially with complexity, and it is rule number one for using OpenMP.

Shared memory programming is seriously tricky; doing the actual programming is the easy bit, and avoiding the *gotchas* is the hard bit. That includes avoiding deficiencies in the language standards and, even more, deficiencies in the OpenMP specification.

We shall now cover some of the reasons why this is, and some guidelines on how to avoid problems.

7.1.2 Generic Warnings

Most of the warnings apply to both Fortran and C/C++, though C/C++ has many more *gotchas*; Fortran has some specific to it, too, so it is not all one-sided.

Unfortunately, this lecture needs to do a lot of language-flipping. Also, warnings for one language may apply to other languages, so take note if you use the equivalent facility. In particular, some Fortran ones apply to C++ as well.

An example of this is not to touch `volatile` in C/C++ – it is completely broken, from start to finish, though the explanation is far too complicated for this course. But why does that not apply to Fortran?

The actual Fortran standard is much less inconsistent with itself (and implementability) than the C standard, Fortran `volatile` is rarely recommended in books and Web pages, and very few programs use it. It does not actually work any better, of course, but you are much less likely to be tripped up by it.

In Fortran, lines starting `!$` and a space are significant, because they are OpenMP Fortran preprocessing lines (which I do not recommend using). The only safe rule is never to start any comments with `!$` other than OpenMP directives.

In C/C++, remember that C and C++ pragmas get preprocessed, so do not define OpenMP keywords as macros either in your code or in anything that it includes, and watch out for any non-C/C++ headers you include.

I do not recommend using conditional compilation, but if you must, this is how. In C and C++, the preprocessor symbol `_OPENMP` is set to the integer value `yyyymm`, where `yyyymm` is the date of the OpenMP API that the compiler claims to use; there is probably no **canonical** mapping from or to version numbers, so you have to use the date of the specification document! In Fortran, if a line begins with `!$` and a space, then the `!$` is removed (actually replaced by two spaces) in OpenMP mode, and so the line is executed; obviously, it is left unchanged if not in OpenMP mode, and so the line is not executed. There are more variations, but that is the basics.

OpenMP facilities are necessarily impure, except for the simple information functions, of course, so Fortran programmers must not use them in `PURE` or `ELEMENTAL` procedures. It is not advisable to use them even in functions, and that includes in subroutines called from functions. Fortran allows aggressive optimisation of expressions (unlike C and C++), and so function calls are not always executed.

The same remark applies to C++ constructors and destructors, for different but related reasons.

7.1.3 Call Chain Issues

In all languages, watch out for code like:

```
void fred (...) {
    /* This */  double a = joe(), b = bert();
    /* or */    if (...) {b = joe();}
    /* or */    for (i = 0; i < n; ++i) {c = joe();}
    /* or */    d = bill(bert(),joe());
    /* or */    d = bert() + joe();
}

double joe () {
    #pragma omp ...
}

void bert () {
    #pragma omp ...
}
```

All of the parallel, work-sharing and barrier directives are collective, and must execute on all threads in the **same** order. Three of those are not defined to do so by the C standard, and two will not do so if you make a simple mistake – do **you** know which?

This applies to all loops, conditionals and branching. In addition, it applies to all function calls in Fortran, anywhere, and ones in C/C++ when they occur in argument lists, initializers, constructors and destructors. Only a language lawyer knows what is defined and what is not.

- As usual, the best way to avoid problems is *keep it simple and stupid.*

Nested parallelism and tasking has similar problems, and it is often necessary to share `private` variables. The meaning of *shared* has drifted over time, and is now somewhat inconsistent. It currently means shared over all threads in the current region, not over all threads. For example, you can have `private` in one region and `shared` in a nested region, and that means that each parent thread's variable are shared with its children, but not with its sibling threads or their children.

Alert: treat this as *sweating dynamite*, and use it with extreme care. Both you and compilers may get confused, and the result will be chaos.

For complicated reasons, do **not** make `DO`/`for` loop variables `shared`, especially in Fortran (where they are `private by default`, because many compilers object.

7.1.4 Types in Directives

The OpenMP specification is very sloppy in places. While it defines most of the syntax fairly precisely, it leaves a great many ambiguities in the language bindings, and implementations may vary. You can pass values in variables to some directives, but it does not specify what types they are allowed to be.

- This is **not** about variables in data clauses.

It is about the `N` in `schedule(static,N)`, and other expressions allowed in some directives. Here are some safe rules for portability and reliability:

- Use default integers, when an integer is needed. That is `INTEGER` in Fortran and `int` in C/C++.
- Similarly, when a truth-value is needed, use `LOGICAL` in Fortran and `int` (not `bool`) in C/C++.

You can probably use any size of integer, but there is no need to, so the above rules are not restrictive.

7.1.5 C/C++ Directive Use

C and C++ are very serial languages; consider the expression: `execute(f(),g());`. In Fortran, `f` and `g` may be called in parallel or not called at all, under some circumstances. In C and C++, they are called sequentially in either order: i.e. `f` and then `g`, or `g` and then `f`.

- OpenMP directives take the Fortran approach; any conflicting side-effects are undefined behaviour.

It applies to the values in `schedule` clause, and anywhere you have an expression in a directive. For example:

```
#pragma omp parallel schedule(static,f())
```

Or, using facilities we have not yet covered:

```
#pragma omp parallel num_threads(f()), if(g())
```

7.1.6 The Default Clause

I do **not** recommend this, but OpenMP does; the syntax is `default(which)`, where `which` is `shared`, `private`, `firstprivate` and so on. It is hard to describe exactly what it controls, and I regard that as a recipe for making mistakes.

- It will also introduce other *gotchas*, quietly.
- `default(private)` is particularly dangerous; you will see why this is as we go on.

7.1.7 Parallel Problems

Most bugs do not show up in simple test cases, because failures are almost always probabilistic, and the probability often increases rapidly with the number of threads. I cover this in a little more depth in the course *Parallel Programming: Options and Design.* The solution is to be really cautious when coding, remember that compilers differ considerably, and the more optimisation you have, the more you are at risk.

- Do not just run a test and see if it 'works'; i.e. that your compiler, system or test does not show the problem.

A simple example that we used before shows this:

```
#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)
}
```

If thread 0 finishes last, there is no data race, so checkers will not find one. Otherwise, there may be one, and some threads may use the total where they should use the mean when calculating the variance.

You may well have a probabilistic race-condition with MTBF (mean time between failures) of many hours. When you run a realistic analysis, it may not work, and tracking down such bugs is a truly evil task. Unfortunately, that is an inherent property of unchecked shared-memory threading, such as OpenMP. For race conditions and similar bugs, very often, erroneous code will work in testing, but:

with a probability of 10^{-12} or less,
or if there is a TLB miss or ECC recovery,
or when moved to a multi-board SMP system,
or if the kernel takes a device interrupt,
or when moving to new, faster CPU models,
or if you are relying on an ambiguous feature,
or . . .

then it will give wrong answers, sometimes.

Consider a race condition involving K entities, where entities can be threads, locations or the combination. The failure rate is $\mathrm{O}(N^K)$ for some $K \geq 2$ (and it is often 3 or 4). This can also occur when assuming more consistency than exists; see later for details of this nightmare area.

As mentioned before, you can sometimes make use of `schedule(static,1)`, which will cause successive iterations to be allocated to threads in a round-robin fashion, and may help to expose conflict between adjacent iterations. Reorganising loops achieves this effect more generally, and a common property is the trick tends to work best when the code is most inefficient!

Updates may not transfer between threads until you synchronise, but they may do so, which is deceptive. In general, shared memory will synchronise itself automatically, but when? The answers is now, later, sometime, mañana, faoi dheireadh; i.e. somewhere between immediately and the end of time, with no guarantee. So incorrect programs often work most of the time, but may fail, occasionally and unpredictably.

- Any diagnostics will often cause them to vanish, by introducing enough of a delay for the memory to synchronise itself.

This also makes it utterly evil investigating data races.

7.1.8 Memory Models

Shared memory seems simple, but is not simple at all, and the obvious orderings of effects often fail to hold. This is too complicated (and evil) to cover properly in this course here, and the following is just an indication of the issues. If you want to learn more, suitable key phrases to look up include: *Data Races* (a.k.a. *Race Conditions*), *Sequential Consistency*, *Strong and Weak Memory Models* and *Dekker's Algorithm.*

Serious masochists should look at the proper references, including academic papers such as those by Peter Sewell and colleagues:

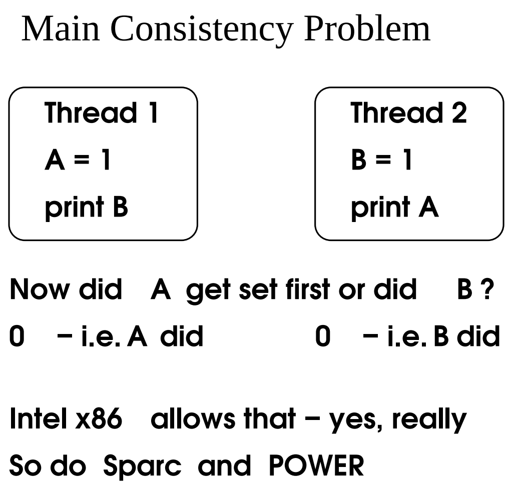
`http://www.cl.cam.ac.uk/~pes20/weakmemory/index.html`

or Intel's actual specification:

Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, 8.2 Memory Ordering

`http://developer.intel.com/products/processor/manuals/index.htm`

If you follow the guidelines here, you can ignore those but, if you start to write 'clever' shared-memory code, you had **better** study them! Here are some graphical examples of what can and does happen, with low probability:



Main Consistency Problem

Thread 1	Thread 2
A = 1	B = 1
print B	print A

Now did A get set first or did B ?

0 – i.e. A did   0 – i.e. B did

Intel x86 allows that – yes, really

So do Sparc and POWER

Figure 1.2

Another Consistency Problem

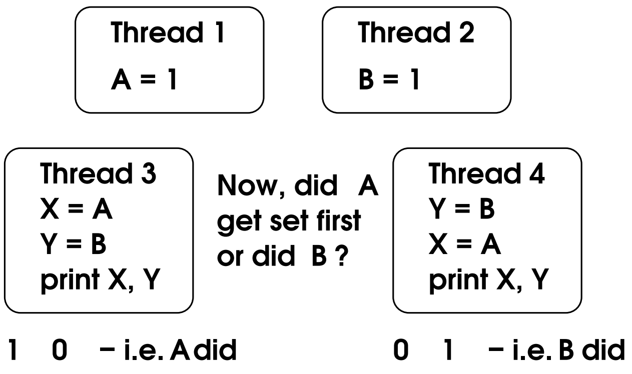


Figure 1.3

How That Happens

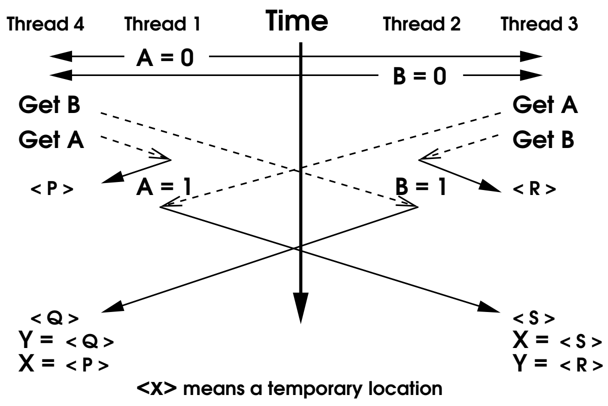


Figure 1.4

It is often said on the Web and in books that it is just due to too much optimisation, but that is **not**, repeat **not**, true! It is allowed by all of the C99, C++03 and Fortran standards, and it is one of the common hardware optimisations.

- It can happen even in unoptimised code.

Regard parallel time as being like special relativity, where different observers may see different global orderings.

7.1.9 OpenMP Debugging

As a summary of the above:

- Failure is often unpredictably incorrect behaviour.
- Variables can change value 'for no reason', and failures are critically time-dependent.
- Serial debuggers will usually get confused, and even many parallel debuggers often get confused, especially if you have an aliasing bug.

- A debugger changes a program's behaviour, and the same applies to diagnostic code or output; problems can change, disappear and appear.

That sounds like a counsel of despair, but there are things you can do, which is why this course has so many "dos" and "don'ts". The object is to not make errors in the first place, and especially avoid ones that are hard to debug. You should try to avoid ever needing a debugger; follow the guidelines here and you rarely will. Some tools that may help are described in the next lecture.

7.1.10 Data Environment

The OpenMP specification is a bit sloppy in this area, too, compilers vary and simple tests can be misleading, so write very conservative code and do not try to be clever. It is also a very hard issue to get your head around, and it dominates bugs, debugging and tuning.

- Rule number two is *KISS, KISS*.

The second *KISS* is *Keep It Separate, Stupid*; i.e. keep private and shared variables very distinct. OpenMP is such that the same name can mean both, depending on the context, and the same rule also applies to the use of pointers. For example, this is not good practice:

```
static int fred ;
void fred ( void ) {
    fred = 123 ;    /* shared */
#pragma omp parallel private ( fred )
    {
        fred = omp_get_thread_num ( ) ;    /* private */
    }
    fred = 456 ;    /* shared */
}
```

The precise rules are very complicated, and I recommend you to ignore them. It is best to think in terms of the following model, except where behaviour is explicitly specified, such as for `firstprivate`:

- Private versions exist only in parallel regions; their values are undefined on entry, and lost on exit.
- Do not access the shared version in a parallel region. Also, a shared variable becomes undefined on exit if it also has a private version.

The second problem is that other procedures (subroutines and functions) can access global data directly, if they are in Fortran modules or common, C external or static variables, or C++ class members (i.e. ones marked `static`); and, of course, the same can be done using pointers.

- If you access a private version in a parallel region, then **never** access it directly during that.

Doing either is fine, doing both leads to chaos. It is easier to obey this rule than describe it. Here are some examples of what OpenMP specifies to show why I recommend avoiding this mess:

Fortran Example:

```
module pete
    integer :: joe = 123
end module pete

integer function fred
    use pete
    fred = joe
end function fred

     use pete
     print *, joe       ! 123
!$omp parallel private(joe)
     print *, joe       ! Undefined value
     print *, fred()    ! Undefined behaviour
     joe = omp_get_thread_num()
     print *, joe       ! Thread number
!$omp end parallel
    print *, joe        ! Undefined value
```

C/C++ Example:

```
int joe = 123;    /* joe is an external static variable */

int fred (void) {
    return joe;
}

    printf("%d\n",joe);       /* 123 */
#pragma omp parallel private(joe)
{
    printf("%d\n",joe);       /* Undefined value */
    printf("%d\n",fred());    /* Undefined behaviour */
    joe = omp_get_thread_num();
    printf("%d\n",joe);       /* Thread number */
}
    printf("%d\n",joe);       /* Undefined value */
```

7.1.11 Classes of Code

There are three important classes of code:

- Serial code, outside all parallel regions.

- Synchronised code, protected by `critical`, `single` or, with reservations, `master`.
- All other code, which may run in parallel.

Remember the following are **not** synchronised:

`critical name1` and `critical name2`

Anything else versus `critical` constructs

Anything else versus unbarriered `master`

A variable accessed in both synchronised and other code must be protected against race conditions between the two; that is not needed for read-only variables, of course. One of the best approaches is to:

- Divide sensitive actions up into separate groups.
- Ensure no overlap of actions between groups.
- Protect every use of each group by one of a single `critical` name, `single` or a barriered `master`.

Using a fully-synchronised form is safest, as was described in the previous lecture, because there is no automatic synchronisation on entry to a work-sharing construct.

7.1.12 Calling Procedures

A construct has an associated lexical scope, which is the actual text of the program to which it applies, such as:

```
!$OMP PARALLEL
        < lexical scope >
!$OMP END PARALLEL

#pragma omp parallel
{
        < lexical scope >
}
```

We described the shared/private defaults earlier, but what rules apply to a procedure called in that? Such procedures are called several times in parallel; for example:

```
!$OMP PARALLEL
        CALL Fred    ! What are the rules inside Fred?
!$OMP END PARALLEL

#pragma omp parallel
{
        fred ( ) ;    /* What are the rules inside Fred? */
}
```

We shall start with code compiled with an OpenMP option; the rules are almost identical to the lexical scope ones; we shall repeat them, but more precisely than before.

All these inherit from what they refer to:

- All Fortran arguments except VALUE
- C++ reference arguments
- Pointers, which are described later, and are not easy

The following variables are shared:

- Any form of global or static data

This includes Fortran module variables, `COMMON` and anything in it, any initialised variable or one with the `SAVE` attribute, C/C++ `static` and `extern` variables, C++ class members (i.e. ones marked `static`) and, somewhat bizarrely, C++ const variables with no mutable members.

The following variables are private:

- Anything explicitly declared as `threadprivate`

You can use this to override the defaults, but be careful.

- Ordinary C/C++ automatic variables and Fortran VALUE arguments

This includes C/C++ non-reference arguments. Remember that Fortran initialisation sets `SAVE`, which is easy to miss, and that Fortran local variables use the default rules.

- Fortran `DO`-loop, implied-do and `FORALL` indices

C/C++ programmers need to watch out for nested loops; only the indices of the actual loops parallelised by an OpenMP `for` directives are automatically private.

7.1.13 Fortran Association

Fortran passes arguments by *association*; this is usually implemented as either reference or copy-in/copy-out, though other techniques have been used in the past and will be in the future. Copy-in/copy-out is the one that causes the problems. This is often described as the array copying problem, though it applies to both scalar and array arguments and, generally, it is dependent on both the compiler and the optimisation used. It will necessarily happen in some circumstances, most commonly:

- Passing assumed-shape or non-contiguous array sections (e.g. ones with vector indices) to explicit-shape or assumed-size dummy arguments.

This is often viewed as passing Fortran 90 data to Fortran 77 interfaces, which is not a bad way to think of it. The problem is that OpenMP `barrier` operates only on the current data and, upon return, the copy-out does not synchronise.

- Do not rely on barriers to synchronise arguments, unless you are sure they have not been copied.

Note that this applies to all levels of call, and not just to calling the procedure that actually calls OpenMP `barrier` or another directive that implies it. It is not a catastrophic problem in practice, if you watch out for it.

C++ classes potentially have similar issues to Fortran, and this applies to non-trivial classes whether you define yourself or use them from the C++ STL; you are relatively unlikely to hit them, but it is fairly likely for move or copy constructors, copy assignment, and when using callbacks from the STL or similar.

- Always assume such things may be called in parallel.

You should avoid using any directive that implies a barrier or other synchronisation in such places, unless you are sure that the specification makes it safe.

7.1.14 'Dynamic Storage Duration'

OpenMP includes the following in two critical places: *Objects with dynamic storage duration are shared.* OpenMP 2.5 used '*heap-allocated storage*', which is even more confusing. Heaven alone alone knows what that means – **Watch Out!** There used to be at least three different opinions among experts, and may well still not be.

Only C++ has the concept of '*dynamic storage duration*', which means objects allocated by operator `new`. We can extend that to C objects allocated by `malloc` and Fortran `POINTER` variables allocated by `ALLOCATE`, though there are also other possibilities. However, all of those are executed and not declared, so each thread will necessarily do them separately, so they are **necessarily** private to each thread! That makes no sense.

- What it **probably** means is that the pointer may be stored in a shared variable used **only** by that thread and, after synchronisation, may be used as shared data.

7.1.15 Language Built-ins and Global State

This refers to the Fortran intrinsic procedures and the functions in the C and C++ libraries, except for I/O and exception handling, which are described later. OpenMP specifies that they are all thread-safe, but there are some cases when that is obviously impossible. There is no problem for Fortran, except for two procedures, and C++ is OK, too, but its inheritance from C is not. Watch out! That is quite a lot of the C++ library.

As an aside, POSIX now includes the whole of C99, and specifies parallel (threading) semantics. Well, it does in **theory**. However, this area of POSIX makes very little sense (it is inconsistent with both C99 and itself and unimplementable in places), and it is unlikely it will match reality on most systems.

- It is best to ignore POSIX in this regard.

Here are some rules that are generally reliable.

- Never change program state in parallel code.

Do it in the main, serial code and propagate it, and it is best to do it before first parallel region. Fortran has very little, but C (and hence C++) has more. You should call all of the following from serial code only:

`EXECUTE_COMMAND_LINE`, `RANDOM_SEED`

`system`, `srand`, `atexit` (and `exit` if there are any `atexit` functions), `setlocale`

As far as random numbers are concerned, OpenMP conflicts with C and POSIX; using `rand` unsynchronised may fail horribly. Calling `RANDOM_NUMBER` in Fortran might fail as well, but that is less clear. The simplest and safest solution is to synchronise the calls to `RANDOM_NUMBER` and `rand`.

The C++ random numbers should also work if each thread uses a separate engine instance, but the statistical properties may be poor. Parallel random generation is a specialist area, and most books and Web pages are misleading or just plain wrong. A reliable authority is *Pierre L'Ecuyer*.

Some C functions return pointers to internal strings, and will often use a single internal string for all threads. You should use all of them within synchronised code only, and copy the data to somewhere safe as soon as possible, and definitely before leaving the synchronised region. These are mainly `tmpnam`, `getenv`, `strerror` and most of the C functions that return date strings.

There are some extra *gotchas* for the multibyte functions, but those monstrosities are best avoided, for reasons outside this course. As mentioned, I/O and exceptions are described later. Most of the rest of the C library should work, though some of it may be very slow, because of interlocking.

Remember that C++ inherits a lot from C.

7.1.16 The C++ Standard Library

OpenMP 3 has added support for this, so here are some warnings; even following these, some implementations may misbehave. The C++ standard is specifies this area very poorly, and can be read in many different and incompatible ways. The following guidelines follow the intent of WG21 (the C++ standards committee) and are generally safe:

- `const` functions and methods read the container and object but do not update it.
- Using an iterator `it` may read the container it corresponds to – indexing (`it[n]`) will do so with some libraries. **Only** dereferencing (`*it` and `it->mem`) definitely do not, though equality comparison (`it1 == it2` and `it != it2`) probably will not.
- Just using a container's elements does not use the container, though assigning to elements and using `swap` on iterators may do under some circumstances in some implementations.
- Updating separate containers does not conflict.
- Updating separate elements of a single container does not conflict, except for `vector<bool>`, where it does.
- Replacing elements by assigning to them and using `swap` iterators works for `basicstring`, `array`, `deque` and `vector`, but doing that on elements of other containers may update the container, under some circumstances in some implementations.
- Updating a container in any way whatsoever may conflict with all iterators, even if the C++ standard says that they are not affected by the update. The reason for this is the way that OpenMP needs to parallelise code.

In summary, the following is safe in parallel regions:

- Any operation that is entirely read-only.

- Updating separate containers.
- Updating separate iterators to the same container.
- Updating, but not replacing or exchanging, separate elements of the same container.
- Replacing or exchanging separate elements of a `basic_string`, `array`, `deque` and `vector` (though not `vector<bool>`) container.

For these reasons you should use only iterators to those containers in `omp for`.

When using other parts of the C++ standard library, though should follow guidelines given elsewhere in this lecture, and the following extra ones for code in parallel regions:

- Update only separate objects in parallel regions, and watch out for indirect objects like locales; traits and similar read-only classes are not a problem.
- Do not use allocators, because doing so safely is extremely tricky.
- Do not use C++ threading, atomics or futures, because they may conflict with the OpenMP implementation.
- Avoid `valarray` and smart pointers, though it is probably safe, because I found that trying to match up the C++ and OpenMP wording, together with a knowledge of how the latter is implemented, was just too confusing.
- Use separate streams or see earlier comments on parallel I/O; even opening and closing separate files is risky.
- Remember that the C library has worse problems, which are described elsewhere in this lecture.

7.1.17 Non-OpenMP Procedures

This refers to anything **not** compiled with an OpenMP option, including libraries that do not explicitly support OpenMP.

- You can always call such procedures from serial code, and almost always from synchronised code.
- Calling them in parallel is undefined behaviour, and anything may happen; you should check if you need to select a special library for OpenMP.

There are a few other things that are fairly safe, but you need to know quite a lot about the code, the language and compiler technology to call such things reliably.

7.1.18 Fortran and Private

OpenMP may need to allocate shadow versions; the following will use 256 MB per thread:

```
COMPLEX(KIND=dp) :: array(256,256,256)
!$OMP PARALLEL PRIVATE(array)
```

You are allowed private `COMMON` blocks, but do not use the facility, because needing them is a sure sign of being out of control; you should be using modules instead, anyway.

Never make anything `EQUIVALENCE`d private, not even if all `EQUIVALENCE`d names are private, because OpenMP's rules are truly mind-boggling. You should not be using `EQUIVALENCE`, anyway, though some older codes do.

Remember that all variables become undefined on entry and exit to parallel regions; that is not good news for `ALLOCATABLE` variables, and OpenMP requires them deallocated in both places, so you must:

- Deallocate the shared version before entering.
- Deallocate the private versions before leaving.

7.1.19 C/C++ Private Arrays

Do not even think of using them

C/C++ arrays are often not really arrays; except when actually allocating the space to store them, they are usually pointers. Regrettably, the C and C++ standards are badly ambiguous here, and the OpenMP specification is inconsistent with them, anyway.

- C/C++ private arrays do not work reliably.

7.1.20 Pointers

Pointers in parallel code are a snare and a delusion, and many experts think languages should not have them at all; let us not be dogmatic, but stick to the following for safety:

- Use shared pointers to point to shared data. You should set or change them **only** in serial code, and can then read their values anywhere in parallel code.
- Use private pointers to point to private data. You must use them only within the same thread, and they become undefined on leaving the parallel region.

It might appear reasonable to include changing shared pointers in synchronised code, and using private pointers to read-only shared data – both of which, theoretically, should work reliably. But there are some truly evil language standard issues so, in practice, doing that is living dangerously. You may need to do so for some codes, but watch out.

Treat private pointers (even in C/C++) like Fortran allocatable:

- Set them to null pointers before entering a parallel region.
- And again before leaving the parallel region.

Remember to free `malloc`ed memory first, if needed, to avoid leaks; Fortran will release the memory automatically. There is no problem if the private pointers are declared inside the parallel region.

- Cray pointers are a common Fortran extension, and some programs use them; **never** make them private.

Using them is not advisable even in serial code but, if you really have to use them, treat them as always shared, and never let OpenMP default them to private. But they are a minefield together with OpenMP.

7.1.21 Reduction Constraints

I advise being cautious, whatever OpenMP implies. OpenMP says the variable must be shared; that is only so that the compiler can treat it specially, and is not a major problem.

There are also some truly evil problems with argument passing that you need to be an implementation technique expert to recognise. The following are some safe rules:

- Never pass the reduction variable as an argument to a called procedure.
- Never set a pointer to the variable, or even take its address.
- Stick to scalars of built-in arithmetic types. This is any of the integer, real or logical ones, plus complex in Fortran only. You can almost certainly use any Fortran `KIND` or standard C/C++ size, but avoid the optional C/C++ sizes.

Most compilers will get those right, or complain. Beyond that, you are asking for trouble.

If you need reductions on arrays, C++ classes or Fortran derived types, you probably need OpenMP 3.1 support or even 4.0, certainly for the last two. You need to check that they work in your compiler.

- Do not rely on them working in any other compiler.

7.1.22 I/O

I/O is a problem in all parallel languages; OpenMP says almost nothing, leaving it ambiguous. The following is what is almost certainly safe, and it will work even if you use OpenMP on a cluster:

- Open and close files in the serial code.
- Ideally, do all I/O in the global master thread, and definitely do all I/O on `stdin`, `stdout` and `stderr` there.

Often that is not feasible, or at least very inconvenient, and the following should be reliable on multi-core CPUs:

- Synchronise open and close against all other I/O.
- Use any one file or unit in a single thread only; that will also work on clusters, usually not in the way that you expect.
- Read from `stdin` in the global master only; synchronising its use may work, but will not always do so.

In addition, you must do all of the following:

- Set line buffering on `stdout` and `stderr` in C/C++, such as by using `setvbuf(stdout,NULL,_IOLBF,BUFSIZ)`. You **must** do that in serial code, and do it early.
- Synchronise all output to `stdout` and `stderr`.
- Write whole lines in a single synchronised section, and do not assume that `stdout` either is or is not the same as `stderr`.

If you cannot set line-buffering (as in Fortran), before leaving every synchronised section with an I/O transfer in it:

- Use the `FLUSH(`*unit*`)` statement in Fortran; if your compiler does not have it, try using `CALL FLUSH(`*unit*`)`.
- Call `fflush(`*stream*`)` in C/C++.

Regrettably, this applies even for diagnostics, and you should use one or the other technique even for `stderr`, or your diagnostics may be almost incomprehensible.

7.1.23 Exception Handling

Cross-context exception handling is pure poison, and you should handle exceptions only in the raising context. This includes `errno`, C++ exceptions and more – anything that indicates an exceptional condition. But what is a context in this sense?

> A parallel or work-sharing or similar construct (i.e. anywhere that OpenMP may switch system thread), and remember the context changes on both entry and exit.

This also includes pretty well anywhere in a task with `untied`. The reason is that exceptions are bound to a system thread, which may well not be the same as an OpenMP thread, and the consequence is:

- Exceptions become undefined at every boundary; i.e. both the entry to and exit from the closest enclosing construct.
- Never include a construct in a `try` block, or do the equivalent actions using `setjmp/longjmp`.
- Do not trust the value of `errno` across a boundary.

Signal handling is simple: **do not even think of doing it.**

It is almost impossible to describe how broken this area is, at all levels from what the hardware provides, through POSIX, up to the language standards that 'specify' it. As usual with such areas, almost everything said about it on the Web or in books is just plain wrong, because of the pressure to say something 'positive'.

7.1.24 IEEE 754 Facilities

Do not use the fancy IEEE 754 facilities (i.e. setting or testing the modes and flags). These are sometimes available in Fortran 2003 and C99, and just occasionally may even work, but are associated with system threads, not OpenMP ones. I do not recommend using them in C99, even in serial code, because C99 got them catastrophically wrong.

There are some things that can be done reliably, but they are too complicated to be worth describing, and few people want to.

7.1.25 Native System Threads

OpenMP implementations almost always use POSIX or Microsoft threads behind the scenes, but OpenMP threads may not the same as those; a single system thread may be used for several OpenMP threads, or conversely, and the binding may vary with time.

- Do not use system threads directly, or a library that does.

The combination may work – or may fail horribly. OpenMP may assume that it is the only thread user. The detailed reasons are too complicated to describe here, but include signal handling and scheduling, as well as the thread state mentioned above.

7.1.26 Compiler Bugs?

95% of 'compiler bugs' reported by even experienced programmers are not that at all, and are typically user errors; subtle breaches of the language standard are perhaps the most common. Unfortunately, that is only 90% for OpenMP, even when the OpenMP specification is unambiguous and implementable.

In 1999, only a few Fortran compilers worked at all; none of the C compilers did, as far as I know.

By 2006, almost all Fortran and many C/C++ ones did.

Today, even C/C++ ones work for simple use.

Note that 'not working' was and is quite often crashing in the compiler itself or when executing the most trivial OpenMP code; even today, there are probably a good many that cannot handle anything complicated. Performance is another matter entirely, and I have not investigated it.

Unfortunately, you must locate the cause before knowing whose bug it is, even in simple examples like the ones in the practicals for this course; I spent a day tracking one trivial one down. Also, for the reasons given above, most bugs are not reproducible; major factors in exposing them include more independent cores (even hyperthreading in this context), the complexity of the code and its synchronisation, and a higher interaction rate between threads.

- Solution: KISS!

That may not eliminate bugs, but does help to identify them, and is an essential first step to fixing your bug or bypassing a compiler bug.

- Triple check your code against the specification.

Trivial breaches often cause extreme effects in parallel code. And, of course, follow the guidelines given in this course.