Introduction to OpenMP

Intermediate OpenMP

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

8.1 Features Not Covered

8.1.1 Summary

This is a miscellaneous collection of facilities, most of which are potentially useful, but many of which are more difficult to use correctly. It includes notes on tuning that do not fit elsewhere, but nothing that you critically need to get started. Use these facilities when you actually need them, but not just because they look neat. It does not cover the really hairy facilities, nor does it explain why I regard them as such.

8.1.2 More on Design

Let us start with a reminder of what was said in the first lecture. OpenMP design should start with a well-structured serial program – in this context, that means that most of the time is spent in a small number of components, which have clean interfaces and spend their time in actual computation (rather than, say, I/O). You should not even attempt to convert the whole program at once, but do it component by component, where possible.

Your data may need restructuring for efficiency, and this will affect multiple components, some of which you are not intending to parallelise. Do not restructure your data unless the gains look fairly large, but a new, OpenMP-friendly structure will usually help even the serial performance by being more cache-friendly.

The same program can use both OpenMP and GPUs, but do not use them at the same time; i.e. the OpenMP and GPU components should run serially, though there is no restriction on how often you can run each. You can also use MPI to link multiple systems, but use OpenMP and GPUs within a single system; this is not often done, as using pure MPI is usually easier, but is done for some very large codes.

But what 'most of the time' actually mean? A good rule of thumb is that it means 75% or more of the time, or 85% or more if restructuring needed. Below that, the effort involved in converting to OpenMP is likely to outweigh the gain, and remember that those are practical minima (i.e. for relatively straightforward conversions). The same remarks are true for MPI and GPUs, of course – you can very rarely get any gain for no effort. You should also check that half of the core count is enough speedup factor; if not, you had better think about using MPI.

This approach gives a major advantage over MPI, which is that intermediate results match between the serial and parallel versions of your code, to within its numerical accuracy, of course. You can develop or modify a component using the serial form, and then parallelise it if it becomes a bottleneck. Best of all, you can compare the intermediate results of the serial and parallel forms when debugging. Theoretically you can do this using distributed memory but, in practice, it turns out to be much harder to do and is often infeasible.

As described earlier, the key to simple and safe OpenMP is to keep gotchas out of parallel regions, which is usually fairly straightforward, but not always. Problem areas include Fortran argument copying, fancy C++ class usage (usually with non-trivial constructors), calling external interfaces (such as POSIX), or when component does a lot of critical I/O. If you hit a problem with this, you should always stop and think before coding.

- Is synchronisation likely to solve the problem and be efficient?
- Is restructuring likely to solve the problem and be efficient?
- Or does this component need an actual redesign?

8.1.3 Debugging Tools

Ideally, a tool would check OpenMP's rules, but none currently seem to, possibly because this would need extensive modification to the compiler. Almost all work by trapping data accesses and synchronisation calls, so they run very slowly at best. They also have to analyse the synchronisation calls to work out what ordering they enforce on the data accesses, and most assume POSIX, so fail for C++ and OpenMP. Another consequence is that they will detect only data races that occur in that run, and not errors that will allow data races, so a program can pass testing but fail when actually used, usually giving subtly wrong answers.

`Valgrind 3.11 drd` says it handles `gcc` OpenMP, but running it produced huge numbers of error messages on correct code, and it failed to catch at least one actual data race. `Sun Studio DRT` may work, but has not been investigated. Theere are some under development that look hopeful, such as `Archer`, `Sword` and others.

Currently, the most hopeful is `Intel Inspector`, but unfortunately it needed too much work to recommend for this course. As I am retired, I shall not be doing this. It produced masses of bogus error messages for `libopenblas` if it was searched by the linker, even there were no references to that in the program; it might work with `MKL` or an unoptimised BLAS. There was one bogus message for each `parallel for`, and huge numbers for OpenMP `tasks`, which is concerning. There was no time to investigate how well it catches real data races.

8.1.4 Running Serially

OpenMP directives are ignored when compiling in serial mode with a non-OpenMP compiler or when not using OpenMP option, though you will usually get `pragma ignored` warnings in C/C++.

- Remember to initialise the variable before the reduction; as mentioned, it is best to do it even when running in OpenMP mode.

The main difficulty is using OpenMP library routines, which are obviously not there. The OpenMP specification contains stub routines; e.g. `omp_get_thread_num` always returns zero and `omp_get_num_threads` always returns one.

- Everything we have covered will work serially.

Generally, code like that when you can do so, then all you need to do is to code up stub routines, and then only for the library routines you use, of course. Then your program should work in serial mode, just as in parallel, with no change. There are more problems when using an inherently parallel algorithm, but that is advanced use and is not covered, except via simple tasking.

8.1.5 More on Reductions

There are more allowed accumulation forms, but I do not recommend these, as I find them unclear:

Fortran:

var `=` *expression op var*
var `=` *intrinsic* `(`*expression*`, ..., ` *var*`)`

C/C++:

var `=` *expression op var*

In all languages, the above forms are not allowed for the `-` operator, of course.

8.1.6 The Workshare Directive

This is available only for Fortran; while it probably has its uses, I doubt that it has very many.

`!$OMP WORKSHARE`
assignment statements etc.
`!$OMP END WORKSHARE`

The statements may contain only Fortran assignments (including `WHERE` and `FORALL`), and OpenMP `critical` and `atomic` constructs. The scheduling of the assignments is unspecified.

There is one rather nasty *gotcha.* If one statement depends on a previous one in the same WORKSHARE construct, OpenMP is inconsistent with itself, so you should not rely on statement ordering.

8.1.7 More Library Functions

These are useful mainly in conjunction with more advanced features that we have not covered, and are mentioned here only for completeness.

```
int omp_get_max_threads (void);
INTEGER FUNCTION OMP_GET_MAX_THREADS ()
```

This returns the maximum number of threads supported.

```
int omp_get_dynamic (void);
LOGICAL FUNCTION OMP_GET_DYNAMIC ()
```

This returns true if dynamic thread adjustment is enabled.

```
int omp_get_nested (void);
LOGICAL FUNCTION OMP_GET_NESTED ()
```

This returns true if nested parallelism is enabled.

There are a few others, but they all set OpenMP's internal state, and I do **not** recommend doing that.

8.1.8 The Flush Construct

OpenMP regards this as a fundamental primitive, but it is deceptive and hard to use correctly. It is what is normally called a *fence*, and that feature is dangerous in all languages that have it, but not always as ill-defined as in OpenMP.

`#pragma omp flush [ (`*variable list*`) ]`

`!$OMP FLUSH [ (`*variable list*`) ]`

If a *variable list* is supplied, it synchronises all variables named in it , except for pointers, where the specification is inconsistent. There are also specific *gotchas* for arguments, but the situation is just too complicated to describe here.

If there is no *variable list*, the specification is ambiguous; it may apply only to directly visible shared data, or it may apply to all shared data, anywhere in the code.

- The latter interpretation is assumed by `critical`, on entry and exit. Heaven help you if the implementation does not do it in this case.
- And remember Fortran association problems, as with `barrier`.

If you do use OpenMP `flush`, be **very** cautious, and I do not recommend using it for arguments at all. Despite appearances, it is a purely **local** operation, **and** it is also needed for reading. In order to transfer data between `thread A` and thread *B*, you need to do the following:

- · Update the data in thread *A*
- · Invoke `flush` in thread *A*
- · Handshake between thread *A* and thread *B*, somehow
- · Invoke `flush` in thread *B*
- · Read the data in thread *B*

There is more information later, under `atomic`.

8.1.9 OpenMP Tuning

Unbelievably, tuning is a worse problem than debugging! Most compilers will help with parallel efficiency, i.e. the proportion of time spent in parallel mode, as used in *Amdahl's Law*, but most users know that from their profiling, anyway.

At the level below that, all you have is hardware performance counters; they are not easy to use, largely because the measurements are not well-suited for programmers, and are only recently supported under Linux. If you want to use them, try Intel's *vtune*, `pfmon`, `perfex` and so on.

- But try to avoid having to do any detailed tuning.

You can also lose a factor of two or more in overheads, and have to analyse the assembler of both the serial and OpenMP versions to work out why. The very worst problem is kernel scheduling glitches, where the **only** useful tool is `dtrace` in *Solaris*, and recently and partially in Linux.

Most people who try tuning OpenMP retire hurt; I have succeeded, but not often. The same applies to POSIX threads, incidentally, and is one of the reasons people often back off them and OpenMP to MPI. So these are my recommendations:

- KISS, KISS (again).
- Use the simple tuning techniques in this course: setting environment variables, experimenting with `schedule` options and so on.
- Do a rough analysis of the data access patterns in your code, and see if you can reorganise your data to help.
- If that does not work, consider redesigning; yes, it really is likely to be quicker than beating your head against this brick wall.

It is important to note some general rules:

- Never, **ever**, use tuning facilities to fix a bug, because hidden bugs almost always resurface later.
- Do not use them until you understand the behaviour, because tuning by random hacking very rarely works.
- What helps on one system may well hinder on another, and the same remark applies when analysing different data with different properties.

8.1.10 The Parallel Directive

Most clauses control the data environment, and there are only two exceptions, used mainly for tuning.

`if (`*expression*`)`

This causes the region to be executed in parallel only if the expression is true.

`num_threads(`*expression*`)`

The expression is the number of threads to use for running the parallel region. Do not make the `num_threads` expression greater than `OMP_NUM_THREADS`; OpenMP says that is implementation defined.

Fortran example:

```
!$OMP PARALLEL IF (size > 1000), NUM_THREADS(4)
< code of structured block >
!$OMP END PARALLEL
```

C/C++ example:

```
#pragma omp parallel if (size > 1000), num_threads(4)
{
    < code of structured block >
}
```

The clauses may be in either order, and both are optional.

The general rules for the number of subthreads used to run a parallel region are quite complicated but, in the cases we cover in this course:

If an `if` clause is present and its expression is false, then use one (i.e. run in serial).

If a `num_threads` clause is present, then the value of the expression is used.

Otherwise, use `OMP_NUM_THREADS`.

These are useful because increasing the number of threads does not always reduce the time, due to overheads, caching issues and increased communication. Because of this threading often helps only for large problems, and you can disable parallelism if it will slow things down. There is often an optimal number of threads for a particular problem or size of problem, and both fewer and more run more slowly; this can be different for different places in the code.

- But these clauses are a real pain to use effectively, and their best values are very system- and even data-specific.

If you want to preserve `threadprivate` values between parallel regions for all threads, you should also run with `OMP_DYNAMIC=false` and set `OMP_NUM_THREADS` explicitly. Also, should not change those, watch out for libraries doing so behind your back, and use only facilities taught in this course. Even using `if` or `num_threads` clauses is a little risky. Or you can read the OpenMP specification (and even then be cautious).

8.1.11 The Atomic Construct

There is an `atomic` construct that looks useful; however, its appearance is very deceptive, because its actual specification is not all that useful and it is unclear exactly what it means. Specifically, its memory consistency is the issue; that concept is explained a bit later.

- Do not start off by using it.

It performs an assignment statement 'atomically', and may be more efficient than using `critical`. Syntactically, it is very like a reduction, and most of the rules of reductions apply to it (i.e. those that apply to the accumulation statements). In C/C++, the form *var* `=` *var op expr* is not allowed, but I can think of no good reason for that.

- Note the evaluation of the expression on the right hand side is **not** atomic; that is really quite a nasty *gotcha*.

Fortran example (note that there is no `!$OMP END ATOMIC`):

```
!$OMP ATOMIC
min_so_far = min_so_far - delta
```

C/C++ example:

```
#pragma omp atomic
min_so_far -= delta ;
```

The following examples are wrong in all of the languages:

```
!$OMP ATOMIC
min_so_far = min_so_far - search(start,min_so_far)
```

This one is a bit more subtle, and is easy to do by accident:

```
#pragma omp atomic
lower_bound += upper_bound-y
#pragma omp atomic
upper_bound -= x-lower_bound
```

OpenMP 3.1 allows a clause to change the use, which can be any of `update`, `read`, `write` or `capture`. `update` is the default and is the OpenMP 2.5 form described above. `read` and `write` are fairly simple and have the forms:

```
<non-atomic var> = <atomic var>

<atomic var> = <expr>
```

But even that **doesn't** necessarily provide consistency. Also, only `<atomic var>` is accessed atomically. Even so, you should convert atomic assignments to these forms rather than just writing them and hoping.

`capture` is like `update`, but allows the program to obtain the value that was updated. It is useful, but is too complicated to describe here, so see the specification if you need it. Watch out for compiler bugs – testing just C++ and two compilers, I found one consistency bug. It also may be fairly slow, as it will often need a lock. All of this is because the hardware rarely supports it directly, and it has to be emulated.

8.1.12 Consistency

It is possible to read and write fairly safely; it is not guaranteed, but is pretty reliable. In addition to the above rules, you should do either of the following (but not both):

A : Set a variable in a single thread, and read its value in any of the threads.

B : Set a variable in any of the threads, and read its value in a single thread.

- Do not rely on **any** other ordering, not between two atomic objects, nor the order as seen in other threads.
- Use the value **only** within the receiving thread.

The last constraint has some non-obvious consequences, because you must not pass derived information on, either. If you have used the atomic value to control the logic of or change any values in your code, do not communicate with any other thread without synchronising first. That includes using or setting any shared objects, whether atomic, reductions or anything else.

- And, if in any doubt, use `critical`.

That should provide consistency, but waitch out for compiler bugs; you can probably do nothing if it does not. I know that this sounds paranoid, but it is not. The new C++ standard does define memory consistency, and OpenMP 4.0 intends to follow it. The picture we saw at the start is very relevant:

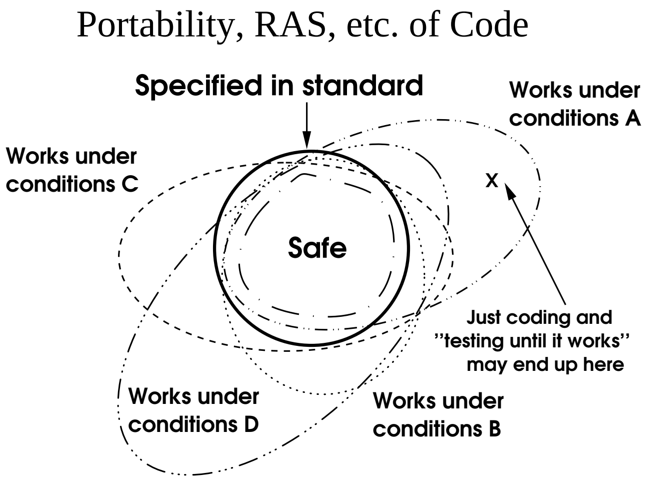

Figure 1.1

Sequential consistency is what most people expect, and is that accesses are interleaved in some unpredictable sequential order, constrained only by the explicit synchronisation constructs. Causal consistency is like special relativity, in that the ordering of remote events depends on the observer, but with no time warps – i.e. impossible orderings. OpenMP has never specified the former, but OpenMP 4.0 says that you do not even get the latter, and it is completely unclear what you do get.

OpenMP 4.0 has a clause `seq_cst` to request sequential consistency, and the stated intent (in a footnote) is to follow C++11. However, OpenMP's model and C++'s model are not fully compatible, and there are a lot of subtle aspects that it leaves unclear. Obviously, this makes no sense at all for Fortran, but it also, for various complicated reasons, makes little sense for C.

8.1.13 Unsynchronised Accesses

Even if you do not synchronise, you will usually get atomicity (but not consistency) if **all** of these conditions hold:

- You are reading or writing single **integer** values; this includes boolean values, `enums` etc.
- They are of sizes 1, 2, 4 and usually 8 bytes.
- They are aligned on a multiple of their size.

Pointer algorithms that assume atomicity are common, and it is usually possible to code them, fairly safely, even if not portably. A decade ago, that was not possible – and it may not be a decade from now. It is also very language- and compiler-dependent and, even today, there are systems where even the cleanest C code will not generate atomic pointer loads and stores.

- You must know the details of your hardware and how the compiler generates pointer accesses; the issues are far too complicated for this course.

Similar (though even stronger) remarks apply to loading and storing floating-point, and the actual operations on it are **very rarely** atomic; I strongly advise never assuming atomicity for those types. Beyond that (e.g. for structures or complex numbers), do not even think about it – always use `critical`.

People who know a little about hardware architecture think that is all that you need, but unfortunately it is not.

- It does not guarantee the consistency you expect, and that applies even on single socket, multi-core CPUs.

It gets rapidly worse on distributed memory systems, of course.

8.1.14 Nowait

A work-sharing construct has an implicit barrier at its end; now consider a parallel region with several of them, and ask whether it would run faster if the barrier were removed.

- MPI experience is generally that it does not.

However, that is for MPI, and it might help with some OpenMP code, especially for SPMD. In Fortran, you just put a `NOWAIT` clause after the `!$OMP END ...` directive. In C/C++, with no end directive, you add a `nowait` clause after the `#pragma omp ...`.

- If you get it wrong, you are in real trouble, because you need to be very, very careful about aliasing issues.

This will not work in any language – but it may appear to:

```
!$OMP PARALLEL
        !$OMP DO REDUCTION (+:total)
            < some DO-loop that calculates total >
        !$OMP END DO NOWAIT
        . . .
        !$OMP DO
            DO n = 1, ...
                array(n) = array(n)/total
            END DO
        !$OMP END DO
!$OMP END PARALLEL
```

8.1.15 Tasking

The `task` construct has clauses `if` and `final`, that can be used to cause the current thread to be suspended in favour of the subthread. The specification is confusing, so read it carefully if you need these. There is also an even trickier `mergeable` clause, which could have performance advantages.

There is also a threadyield construct, allowing temporary suspension, which may be critical if you use both tasks and locks. It may not be needed if the tasks have the untied clause, but that is merely a guess, and the specification is unclear. But, generally, the combination of tasks and locks is **Bad News**, because the OpenMP features do not work well together.

If you use tasks and either locks or thread-specific state, you need to learn about task scheduling and synchronisation. This course avoids that complicated area by simply saying do not combine those features.

Data are tied to threads, not tasks, and tasks are tied to arbitrary threads, but at least they do not change thread dynamically. This can cause performance problems and even deadlock. You can specify the untied clause to allow the implementation to migrate them between threads (and add a few extra features).

But be warned that this will break **all** thread-specific state, including threadprivate, OpenMP thread identifiers (i.e. the result from omp_get_thread_num), errno, IEEE 754 flags and modes, C++ exceptions, and it may even break constructs like critical. You are **strongly** advised to avoid untied.

8.1.16 Environment Variables

We have already covered OMP_NUM_THREADS, and the settings of OMP_SCHEDULE. OMP_DYNAMIC=true has been mentioned and is mainly for SPMD; it allows the number of threads to vary dynamically. OMP_NESTED=true enables nested parallelism; the details are too complicated to cover in this course, and I shall give just a summary of the intent.

Ideally, we want as many threads as possible, and the compiler and system choose which ones to run; that is what I call the sea of threads model, but OpenMP does not handle it very well. It does not even nested parallelism very well, where subthread can spawn a parallel region, but that can be done, and can be useful. Doing that is is advanced OpenMP and is not covered here.

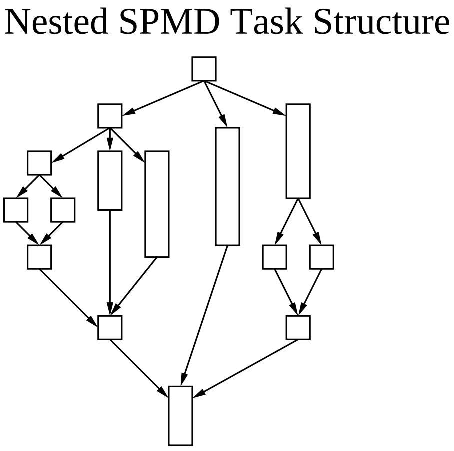

Figure 1.2

8.1.17 C++ Iterators

OpenMP 3.1 claims to support C++ iterators, and the only constraint is that they must be random access.

Don't you believe it!

Class and iterator methods must be *pure* – this is not a standard C++ concept, and is rather like `const`, but applies to updates, too. It is also considerably stronger than than the rules C++ uses for `const`. The main rule is that there should be no side-effects in the methods, and there must be no reference to anything that might change; for example, container elements must not move or be added.

As far as the library goes, classes `vector`, `deque`, `array` and probably `basic_string` and `string` should be safe, but do not add or remove elements. You should use Fortran rules for iterators in OpenMP `for`, as taught earlier, and access elements using **only** operators '`*`' and '`[ ]`'. If you use your own classes, you should model them on these.

8.1.18 Locks

OpenMP has facilities for thread locking, which are essentially a dynamic form of `critical`, but I do not recommend using locking.

- It is very easy to cause deadlock or hard-to-debug livelock.
- It can often cause very poor performance or worse.
- It generally indicates that the program design is wrong.

But, if you really must use them, there are two kinds, which OpenMP calls simple locks and nested locks, though the usual terminology for them is simple and recursive mutexes. OpenMP also uses the term setting rather than locking, for some reason.

- Do not mix these types in any way; that is an almost sure sign of a completely broken design.

Simple locks are set or unset. Once a thread has set a lock, it owns that lock. If it already owns it, then attempting to set it is undefined behaviour. Another thread attempting to set it when it is set waits until it is unset. Only the owning thread can unset a lock, and attempting to unset a lock that it does not own or that is not set is also undefined behaviour. Examples are given only for simple locks.

Nested locks are very similar in most respects; the only difference from simple locks is that an owning thread can set a lock, and what that does is to increment a lock count. Similarly, unsetting just decrements the lock count, and only when that is zero does the lock become unset. Again, it is undefined behaviour to attempt to unset a lock that is not owned or where the count is zero.

Generally, you should avoid nested locks, but they have some uses, though nothing that you cannot program in other ways. If you want to, see the specification for details of their use.

Lock variables should be `static` or `SAVE`; OpenMP does not say that, but not doing so may fail, because it is essentially unimplementable if they are not. It is generally best for them to have file scope or be in a module. You should initialise and destroy them in serial code, though you could do that in a single, synchronised thread, with care.

- You must initialise them before using them in any other way.

Preferably, you should destroy them after their last use as locks. You could then reinitialise them, but doing that is not recommended.

C/C++ initialisation example:

```
static omp_lock_t lock;

omp_init_lock (& lock);
    . . . use the lock . . .
omp_destroy_lock (& lock);
```

Fortran initialisation example:

```
INTEGER(KIND=omp_lock_kind), SAVE :: lock

CALL omp_init_lock (lock)
    . . . use the lock . . .
CALL omp_destroy_lock (lock)
```

C/C++ usage example:

```
omp_set_lock (& lock);
    . . . we now own the lock . . .
omp_unset_lock (& lock);
```

Fortran usage example:

```
CALL omp_set_lock (lock)
    . . . we now own the lock . . .
CALL omp_unset_lock (lock)
```

You can also test whether a lock is set; if it is not set, the action also sets the lock; you must not test in owning thread for simple locks. I do **not** recommend using this feature, because it is trivial to cause livelock or dire performance. It can also cause some **extremely** subtle consistency problems, so subtle that I do not fully understand them myself! Using this facility to improve performance is very hard indeed.

- Using lock testing (or nested locks) to ensure correctness is a mistake, and almost always indicates a broken design.

8.1.19 Locks and Synchronisation

Remember `flush`? Locks have the same issues and, as usual, OpenMP is seriously ambiguous about this.

- A lock is global, but only the lock itself.

It only does local synchronisation on the the memory, and the following is all that is guaranteed:

> If some data are accessed **only** under a particular lock, then all such accesses will be consistent.

That can be extended to serial code as well, but it cannot be extended to other synchronisation. That is less useful than it appears, as the problem then becomes how to get information into or out of the locked region, and you need another synchronisation mechanism to do that. So, how can you use locks to force consistency between two variables *A* and *B*? *A* and *B* must be protected by the same lock; using a separate lock for each will not work. The basic rules for using locks correctly are:

- Protect everything to be made consistent, either by a lock or putting it in serial code.
- Separately locked data should be independent; that means that they should not just be different data, but no ordering between them should be assumed.

This is how you set up the lock:

```
static omp_lock_t lock;
int A = 0, B = 0, X, Y;
omp_init_lock (& lock);
#pragma omp parallel shared(A,B), private(X,Y)
{
    . . .
}
omp_destroy_lock( & lock);
```

and this is how you use the lock:

```
omp_set_lock (& lock);
switch (omp_thread_num()) {
case 1 :        A = 1;        break;
case 2 :        B = 1;        break;
case 3 :        X = A;        Y = B;        break;
case 4 :        Y = B;        X = A;        break;
}
omp_unset_lock (& lock);
```

8.1.20 Not Covered At All

Many other things in OpenMP 2.5 have been deliberately not covered, mostly because they are too difficult to teach. This usually means that they are very hard to use correctly, but some are hard to implement, and may not be reliable. They include:

- Library functions to set OpenMP's state.
- The `ordered` clause, because it is probably not useful.

And quite a few minor features and details, plus the areas mentioned earlier and not recommended.

OpenMP 4.0 added GPU features, which makes me gibber, given the mess the current specification makes of even quite simple activities.

There has also been no discussion about how to configure your system in order to use OpenMP; there is a little in *Parallel Programming: Options and Design*.