Introduction to OpenMP

Synchronisation

N.M. Maclaren

Computing Service

nmm1@cam.ac.uk
ext. 34761

August 2011

1.1 Summary

The facilities here are relevant to both SIMD and SPMD, and are a bit of a hodge-podge, so may be a little confusing. Unfortunately, there is no 'right' order to teach this sort of facility and when to use them, so it may be unclear why and when you need them. Most of the reasons will be covered later. But, for example, you may need serial code in a parallel region (e.g. to read some data from `stdin`); or you may have some unavoidable data dependencies, and need to pass data between threads in a parallel region.

The problems with this are mainly performance and deadlock or livelock. The performance ones are best minimised by minimising the use of synchronisation, and not synchronising all threads unless it is critical for correctness. Some guidelines are mentioned as we describe the facilities.

Deadlock and livelock cause program failure, and we cover only techniques that avoid them, but you must follow the guidelines for safety. We first need to explain deadlock and livelock.

1.2 Deadlock and Livelock

Deadlock is when two or more threads are waiting, and none can make progress until another does. There are many ways it can occur, but it is easy to give some rules for avoiding it. It is one of the most common errors when using locks, so this course does not recommend them. There is another common cause with OpenMP, but that is covered later, under split parallel and work-sharing constructs.

Livelock is when two or more threads are in an indefinite loop which, in theory, will always terminate, eventually, but the actual logic or scheduling means that it never does. Sometimes, it occurs in a probabilistic form, and such loops sometimes become ridiculously slow. The problem with teaching this is that all simple examples are unrealistic, even though the issue is fairly common with non-trivial inter-thread communication.

So we are going to *keep it simple and stupid*, largely by minimising and simplifying communication.

- You need to think in terms of the control flow: specifically, indefinite looping, however it is done. Simple alternative code (e.g. `IF`) does not matter, in itself, because the problem is primarily the number of times that loops are repeated.

- You should avoid one thread's control depending on another's; that is overkill, but it is the only simple rule. Naturally, even simple alternative code can cause problems here.
- Do not assume **anything** about the system thread scheduling; an indefinite loop in one thread may stop another from running at all.

1.3 What Is Communication?

It include any way of passing information between threads, including by using locks, files, messages, signals or global data, and that includes any form of program state. This course will cover mainly updating global data, but the same mechanism can also protect the other. It covers the safe update of and access to shared objects. The main facilities are the `critical`, `master` and `single` directives; `atomic` is covered later, but is rarely useful.

Horrible Warning: execution order does **not** imply data consistency.

Each synchronisation construct has wildly different rules, and many of them are seriously counter-intuitive. To synchronise data, it is best to use a `barrier`, despite what most books and Web pages say; using `flush` constructs is **much** trickier.

- This is a major, but **major**, *gotcha*.

Erroneous code will often fail as you increase the level of optimisation, or appear to work on some systems and fail, occasionally, on others.

1.4 The Barrier Construct

All of the threads halt at this construct and, when the last one reaches it, they all restart. You should use it when you want all threads to be consistent.

Fortran specification:

```
!$OMP BARRIER
```

Note that there is no `!$OMP END BARRIER`.

C/C++ specification:

```
#pragma omp barrier
```

There is an implicit barrier executed by all threads at:

- A `barrier` construct.
- Entry to and exit from a `parallel` construct.
- Exit (**only**) from a work-sharing construct; i.e. `DO/for`, `sections`, `single` and `workshare`, and then not when you use `nowait` (covered later).

There are no barriers for any other constructs; `master` and `critical` are the main *gotchas*.

1.5 The Critical Construct

The block executes in each thread, one at a time (i.e. serially, but not in serial mode), and it is an essential bypass around the aliasing restrictions.

- It is very easy to cause livelock; nested use of it can also cause deadlock.
- The specification is seriously ambiguous, and it may not work in a useful fashion in some (arguably poor) implementations.

You can use it almost anywhere you want to, probably even in serial code, though the specification is most unclear and I do not recommend doing that. The next lecture will describe when you need it.

Fortran specification:

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

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

C/C++ specification:

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

In all languages, the `(<name>)` can be omitted, but I do not advise it, and unnamed `critical`s all use the same anonymous name.

- The `name` is an external entity in your language.

You must make it different from everything else like that: external procedures, modules, almost everything declared `extern`, `COMMON`, and so on.

- The only interlocking is between `critical` sections, and then only between ones of the same name.
- It will synchronise only directly visible, shared data.

Some implementations will synchronise all data being accessed by the thread, whether or not it is visible in the scope where the `critical` occurs, but others may not. If you are anything non-trivial, use a `barrier` and be safe, or at least be safer.

Example:

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

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

1.6 The Master Construct

The block is executed only in the master thread zero, and the other threads effectively just skip over it. The `master` construct seems exactly equivalent to:

```
if (omp_get_thread_num() == 0) <structured block>
```

I have absolutely no idea why OpenMP provides it; there are several different specifications that would be much more useful. But using it makes your intention a little clearer.

Fortran specification:

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

C/C++ specification:

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

One very important use is for serialised I/O; reading from `stdin` must be done like that, and many programs do I/O only in the master (for good reasons). You can use `master` in a parallel region, and it will restrict that code to executing on thread zero.

- But be warned that it is not synchronised.

Other threads will carry on running in parallel while that code is executing – i.e. they may be executing any other code in the parallel region.

- And it will not synchronise any data.

Generally, it has to be used in combination with a `barrier`.

Example:

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

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

1.7 The Single Construct

This **is** an ordinary work-sharing construct, except that one thread does all of the work! What it does is to execute one thread only, and the other threads effectively just skip over

it. Which thread? That is unspecified and unpredictable. That does not matter for a lot of synchronised code, and it is a very useful facility for I/O and similar uses.

Fortran specification:

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

C/C++ specification:

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

The `copyprivate` clause is a little like `lastprivate`, but only a little, and it is really a hack to make the `single directive` more useful. Indeed, it can be used **only** on `single` directives, and copies the single executing thread's value to all threads. The variable must be declared `threadprivate` or declared as `private` on the on `parallel` directive itself. And, again, it cannot be used for Fortran allocatable variables.

Warning: in Fortran, it is put on the `END SINGLE` directive, for some bizarre reason.

Fortran example:

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

Fortran example:

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

1.8 Horrible Warning

Remember the shared and parallel aliasing problems (i.e. races between work-sharing regions and the code outside them)?

- * They apply to `master` and `critical`, too, just as they do to the `single` construct.

This is a particular *gotcha* with `master`, because it is executed in thread zero, but not executed serially. You may need to add extra barriers to stop this, and we shall return

to the `barrier` construct later, but here is how to create forms that behave like ordinary work-sharing contructs.

Fortran `critical`:

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

C/C++ `critical`:

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

Fortran `master`:

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

C/C++ `master`:

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

1.9 Performance

You should avoid using `critical` where performance matters, because it necessarily serialises all of the threads, and this warning also applies to `master`, `single` and `atomic` (see later). But do not worry about code that is rarely executed, which includes most initialisation, termination, error handling and so on.

If you cannot avoid doing that, for any reason, do not assume **anything** about thread scheduling, or you will have to learn about advanced tuning.

1.10 Split Parallel and Work-Sharing Directives

Compilers may create threads at a parallel directive, and destroy them at the end of the region. If they do, fewer parallel regions is better, and you can use several work-sharing regions inside each one. This is significantly trickier to use, so do not do it unless it is fairly important. Start by asking how often are parallel directives executed; if not very often, relative to the running time of your program, then it is not worth bothering unless you need the functionality. The technique is to use them rather like simple SPMD, even if they are SIMD:

Parallel directive
    Work-sharing construct
    Work-sharing construct
    . . .
End parallel region

Work-sharing constructs can be fairly general, not just what OpenMP calls work-sharing constructs, and here is a list:

- Open code (i.e. code not in an explicit construct) is executed in all threads, in parallel.
- `DO`, `for` and `sections` distribute the work across threads.
- `single` and (barriered) `master` execute in only one thread.
- (Barriered) `critical` executes in each thread, serially.
- `barrier` synchronises across all threads.

But it is very easy to make a mistake doing this, and remember that both `master` and `critical` do not, of themselves, synchronise anything.

1.11 Avoiding Deadlock

The following design is guaranteed to avoid deadlock:

- The top level is a sequence of `parallel` constructs; anything not in one is serial code, of course.
- Each has a sequence of work-sharing constructs; anything not in one is executed in all threads. In this sense, `barrier` is a work-sharing construct, but plain `master` and `critical` are not, though the work-sharing uses are.
- Each has a sequence of `critical` constructs; anything not in one is executed in parallel (in some sense).
- Each is a sequence of code and `atomic` constructs (covered later).
- Within a single parallel region, you must match all of the potential barrier constructs: `barrier`, `DO`/`for`, `sections`, `single` and `workshare` (not yet mentioned).
- All threads execute exactly the same sequence; e.g. they **all** execute `DO`/`for`, then `barrier`, and so on.

When considering this matching, you should ignore `master`, `critical` and `atomic` constructs. What will happen if you get this wrong? It is undefined: your program may hang or may go weirdly wrong.

In case it is not clear, the above design avoids deadlock by restricting the directives that may be used while executing the structured block of another directive.

1.12 Synchronised Constructs

Above, we mentioned where constructions had implicit barriers. The following will not work reliably – but it may appear to:

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

There are many, many variations on that *gotcha*, and none of them are obvious when looking at the code. There is more in the next lecture under the description of the rule *KISS, KISS*.

As a reminder, to get work-sharing forms of `master` and `critical`, just follow them by a `barrier` construct, and you can use them **exactly** like another work-sharing construct. Examples were given earlier, so look back for them.

- Consider adding extra `barrier` constructs.

For example, you can put them **before** all of your work-sharing constructs; they then become fully synchronised forms. This can make both debugging and tuning easier, and may slow your program down or may speed it up!

- It is a good idea to use these for SIMD work, and you can later remove barriers as part of tuning, if it helps.

The following are fully synchronised on entry and exit, exactly like the combined forms (`PARALLEL DO` etc.):

Fortran forms:

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

C/C++ forms:

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

`DO`, `for` and `SECTIONS` share the work across threads, and `SINGLE` executes only one of the threads.

Fortran forms:

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

C/C++ forms:

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

`MASTER` executes only the master thread (thread zero), and `CRITICAL` executes each in unpredictable order.