Introduction to OpenMP

Intermediate OpenMP

Nick Maclaren

nmm1@cam.ac.uk

September 2019

Summary

This is a miscellaneous collection of facilities
Potentially useful, but more difficult to use correctly

Includes notes on tuning that don't fit elsewhere
Nothing that you critically need to get started

- Use these when you actually need them
Don't use them just because they look neat

- It doesn't cover the really hairy facilities
Nor does it explain why I regard them as such
Please ask if you are interested or need to know

More on Design (1)

This is what was said in the first lecture:

- Start with a well–structured serial program
Most time spent in small number of components
Must have clean interfaces and be computational

- Don't even attempt to convert whole program
Do it component by component, where possible

This is the approach used in the examples

More on Design (2)

Your data may need restructuring for efficiency
Will affect multiple components, some serial
Don’t do this unless the gains look fairly large

- But new structure usually helps serial performance

Same program can use both OpenMP and GPUs

- But don’t use them at the same time

OpenMP and GPU components run serially

Can also use MPI to link multiple systems
But use OpenMP and GPUs within a single system
Not often done, as using pure MPI is easier

More on Design (3)

- Most time usually means 75% or more
Look for 85% or more if restructuring needed

Below that, effort likely to outweigh the gain
- And remember that those are practical minima
Same remarks are true for MPI and GPUs, of course

- Check that half core count is enough speedup
If not, you had better think about using MPI

Advantages

This approach gives a major advantage over MPI

- Intermediate results match, serial versus parallel

Within the numerical accuracy of your code, of course

Can develop components using the serial form
Then parallelise it if it becomes a bottleneck

- Can compare the serial and parallel forms

Theoretically can do this using distributed memory
In practice, it turns out to be much harder to do

Gotchas

- Key is to keep gotchas out of parallel regions

Usually, fairly straightforward, but not always
If you hit a problem with this, stop and think

- Is synchronisation likely to work and be efficient?
- Is restructuring likely to work and be efficient?
- Or does this component need a redesign?

Fortran argument copying, fancy C++ class usage
Or calling external interfaces (e.g. POSIX)
Or when component does a lot of critical I/O

Debugging Tools

Ideally, would check OpenMP's rules – none seem to
Trap data accesses, so very slow or worse
Also need to trap and analyse synchronisation
Mostly assume POSIX, and fail for C++ and openMP

So pick up only actual data races, not potential ones
Data races can appear when program actually used

Valgrind drd says it's OK for gcc – it isn't
    Masses of bogus messages, plus missed errors
Sun Studio DRT may work – not investigated
Some under development – Archer, Sword etc.

Intel Inspector

I investigated this for rewriting this course

Needs too much bug fixing and system work
If I were not retired . . .

Bogus messages for libopenblas even if not used
    Might work with MKL or unoptimised BLAS
One bogus message for each parallel for (bug!)
Huge numbers for OpenMP tasks – not a good sign

No time to investigate if it catches all errors

Running Serially (1)

OpenMP directives are ignored in serial mode
Non–OpenMP compiler or not using OpenMP option
Usually with pragma ignored warnings in C/C++

- Remember to initialise variable before reductions

Best to do it even when running in OpenMP mode

- Main difficulty is using OpenMP library routines

The OpenMP specification contains stub routines
E.g. omp_get_thread_num always returns 0

Running Serially (2)

- Everything we have covered will work serially

Generally, code like that when you can do so

All you need to do is code up stub routines

- Only for library routines you use, of course

Your program should work in serial, just as in parallel

More problems when your algorithm is parallel
But that’s advanced use and not covered here

More on Reductions

There are more allowed accumulation forms
I don't recommend these, as I find them unclear

Fortran:
<var> = <expression> <op> <var>   [Not for –]
<var> = <intrinsic> (<expression> , ... , <var>)

C/C++:
<var> = <expression> <op> <var>   [Not for –]

The Workshare Directive (1)

This is available only for Fortran
It probably has its uses, but I doubt very many

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

The <assignment statements etc.> may contain only:

- Assignments (including WHERE and FORALL)
- OpenMP critical and atomic constructs

The scheduling of the assignments is unspecified

The Workshare Directive (2)

Gotcha!

If one statement depends on a previous one
OpenMP is quite seriously inconsistent

- Avoid depending on statement ordering

More Library Functions (1)

Useful mainly with more advanced features
Mentioned here only for completeness

int omp_get_max_threads (void) ;
INTEGER FUNCTION OMP_GET_MAX_THREADS ( )

The maximum number of threads supported

int omp_get_dynamic (void) ;
LOGICAL FUNCTION OMP_GET_DYNAMIC ( )

True if dynamic thread adjustment is enabled

More Library Functions (2)

int omp_get_nested (void) ;
LOGICAL FUNCTION OMP_GET_NESTED ( )

True if nested parallelism is enabled

There are a few others, but I don't cover them
They all set OpenMP's internal state

- And I don't recommend doing that

The Flush Construct (1)

OpenMP regards this as a fundamental primitive

- But it’s deceptive and hard to use correctly

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

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

If a list, synchronises all variables named in it
Except for pointers, where the spec. is inconsistent

- There are specific ‘gotchas’ for arguments

The situation is just too complicated to describe here

The Flush Construct (2)

- If no list, the specification is ambiguous
May apply only to directly visible, shared data
May apply to all shared data, anywhere in code

Latter form is assumed by critical, on entry and exit
%deity help you if the implementation doesn't do it

And remember Fortran association (as with barrier)

- If you use OpenMP flush, be very cautious
I don't recommend using it for arguments at all

The Flush Construct (3)

Despite its name, it is a purely local operation

- And it is also needed for reading

To transfer data between thread A and thread B:

- Update the data in thread A
- Invoke flush in thread A
- Synchronise thread A and thread B, somehow
- Invoke flush in thread B
- Read the data in thread B

There is more information later, under atomic

OpenMP Tuning (1)

- Unbelievably, tuning is worse than debugging

- Most compilers will help with parallel efficiency
I.e. proportion of time in parallel (Amdahl's Law)
But most users know that from their profiling!

- Below that, hardware performance counters
Not easy to use and only recently under Linux
Try Intel's vtune, pfmon, perfex etc.

- Try to avoid having to do detailed tuning

OpenMP Tuning (2)

- Can also lose a factor of 2+ in overheads
Have to analyse the assembler to work out why

- Worst problem is kernel scheduling glitches
Only useful tool is dtrace in Solaris (and Linux)

Most people who try tuning OpenMP retire hurt
[I have succeeded, but not often]

- Same applies to POSIX threads, incidentally
One of the reasons people often back off to MPI

OpenMP Tuning (3)

So these are my recommendations:

- KISS, KISS (again)

- Use the simple tuning techniques in this course
Setting environment variables, schedule options etc.

- Do a rough analysis of data access patterns
See if you can reorganise your data to help

- If that doesn't work, consider redesigning
Yes, it really is likely to be quicker

Tuning Facilities

Important to note some general rules:

- Never, EVER, use them to fix a bug

Hidden bugs almost always resurface later

- Don't use them until you understand the behaviour

Tuning by random hacking very rarely works

- May help on one system and hinder on another

Same remark applies when analysing different data

The Parallel Directive (1)

Most clauses control the data environment
There are only two exceptions, used mainly for tuning

Clause ‘if (<expression>)’
Execute in parallel only if <expression> is true

Clause ‘num_threads (<expression>)’:
<expression> is number of threads used for region

Don’t make num_threads > OMP_NUM_THREADS
OpenMP says that is implementation defined

The Parallel Directive (2)

Fortran:

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

C/C++:

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

Clauses in either order, and both are optional

Number of Subthreads

The general rules are quite complicated
But, in the cases we cover in this course:

- If the if clause is false, then 1 (serial)
- If a num_threads clause, then num_threads
- Otherwise, OMP_NUM_THREADS

Why Are These Useful?

Increasing threads doesn't always reduce time

- Threading often helps only for large problems
Can disable parallelism if it will slow things down

- Often an optimal number of threads
Both less and more run more slowly
Can be different for different places in the code

- But they are a real pain to use effectively
And their best values are very system–specific

More on Threadprivate

How to preserve values between parallel regions

- You must run with OMP_DYNAMIC=false

You also must have set OMP_NUM_THREADS

- Don't use any facilities not taught in this course

- Don't change those and watch out for libraries

Even using if or num_threads clauses is risky

Or read the specification and even then be cautious

The Atomic Construct (1)

There is an atomic construct that looks useful

- However, its appearance is very deceptive

Its actual specification isn’t all that useful
And OpenMP 4.0 makes current uses undefined!

Specifically, its memory consistency is the issue
That concept is explained a bit later

- Don’t start off by using it

The Atomic Construct (2)

Performs an assignment statement ‘atomically’
It may be more efficient than using critical

- Most of the rules of reductions apply to it

I.e. those that apply to the accumulation statements

In C/C++, ‘<var> = <var> <op> <expr>’ is not allowed
I can think of no good reason for that

- Note the RHS expression is not atomic

That is really quite a nasty “gotcha”

Atomic Examples

Fortran example:

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

Note that there is no !$OMP END ATOMIC

C/C++ example:

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

What Not To Do

These examples are wrong in all of the languages

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

This is a bit more subtle – easy to do by accident

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

The Atomic Construct (3)

OpenMP 3.1 allows a clause changing the use:

        update, read, write, capture

update is the default and is the form described above

read and write are simple:

        <non−atomic var> = <atomic var>

        <atomic var> = <expr>

- But that doesn't necessarily provide consistency

Only <atomic var> is accessed atomically

- You should convert atomic assignments to these

The Atomic Construct (4)

capture is like update, but gets the old value
Useful, but too complicated to describe here
See the specification if you need it

- Watch out for compiler bugs!

Testing just C++ and two compilers, I found one

May be fairly slow, as it will often need a lock
This is because the hardware rarely supports it

Simple Atomic Read/Write (1)

It is possible to read and write fairly safely

- It's not guaranteed, but is pretty reliable

Do either of the following but not both:

- Set a variable in a single thread

Read its value in any of the threads

- Set a variable in any of the threads

Read its value in a single thread

And there are more restrictions ....

Simple Atomic Read/Write (2)

- Don't rely on any other ordering
Not between two atomic objects, nor in other threads

- Use the value only within the receiving thread
That has some non–obvious consequences
You mustn't pass derived information on, either

- Don't communicate without synchronising first
Including using or setting any shared objects
Whether atomic, reductions or anything else

Simple Atomic Read/Write (3)

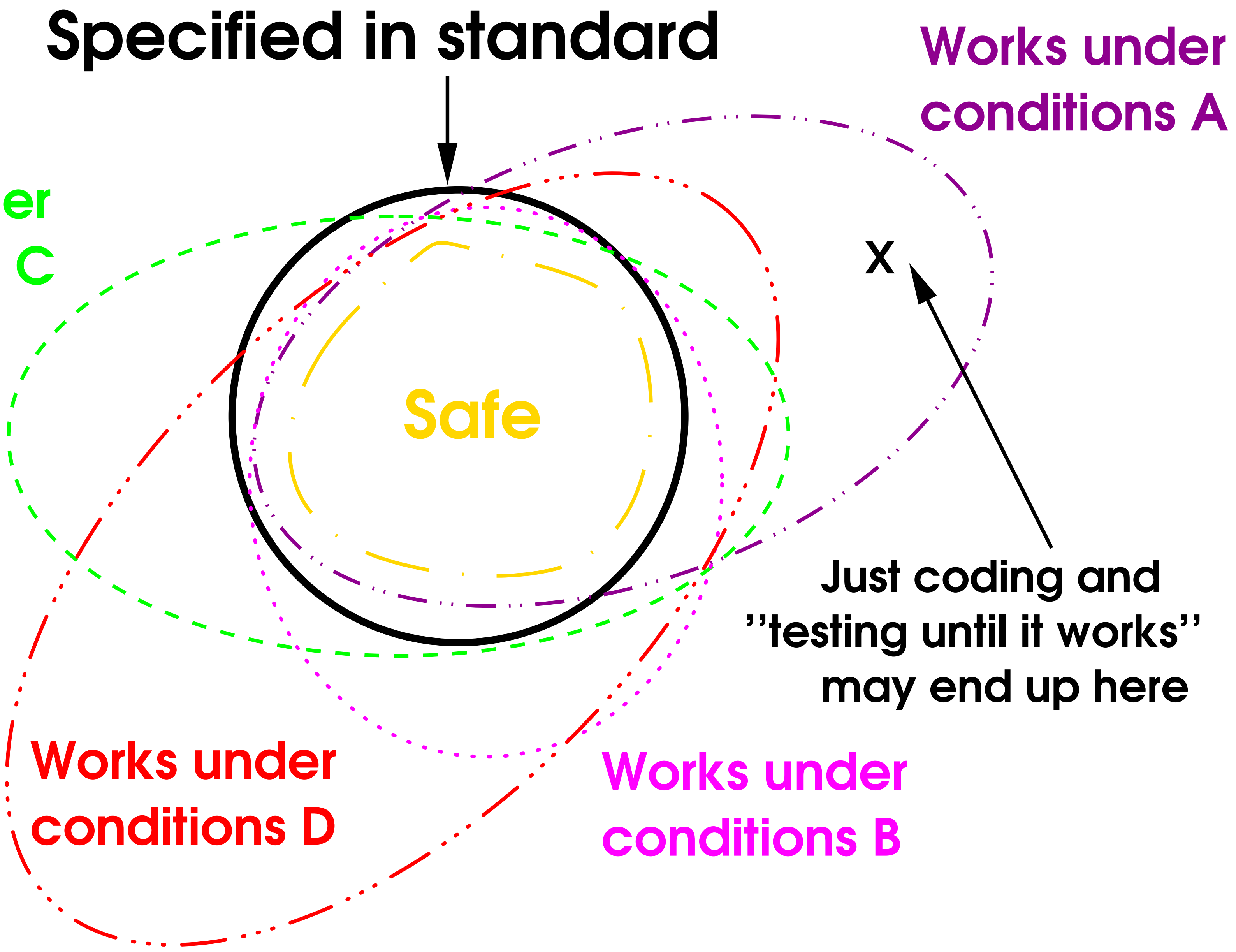
⇒ And, if in doubt, use critical
That should provide consistency, but watch out

Yes, I know that this sounds paranoid, but it isn't

The new C++ standard does define this
And OpenMP 4.0 intends to follow it (see later)

The picture we saw at the start is very relevant

Portability, RAS, etc. of Code



Memory Consistency

Sequential consistency is what most people expect
Accesses are interleaved in some sequential order
Constrained only by explicit synchronisation

Causal consistency is like special relativity
Ordering of events depends on the observer
But with no ‘time warps’ – i.e. impossibilities

OpenMP has never specified the former
OpenMP 4.0 says you don’t even get the latter
%deity alone knows what you do get

Consistent Atomics

OpenMP 4.0 has a clause seq_cst to request this
The intent (in a footnote!) is to follow C++11

There are a lot of subtle aspects that it leaves unclear
OpenMP's model and C++'s are not fully compatible

- This makes no sense at all for Fortran

And, for various complicated reasons, not much for C

Unsynchronised Atomic Access (1)

Will usually get atomicity if all of these hold:

- Reading or writing single integer values
  Including boolean, enums etc.
- of sizes 1, 2, 4 and usually 8 bytes
- which are aligned on a multiple of their size

That's all you need, isn't it? Unfortunately, NO!

- It doesn't guarantee the consistency you expect

That applies even on single socket, multi–core CPUs

It gets rapidly worse on distributed memory systems

Unsynchronised Atomic Access (2)

Pointer algorithms that assume atomicity are common
It is usually possible to code them, fairly safely
A decade ago, it wasn't – and may not be in a decade
Also very language– and compiler–dependent

- You must know your hardware and compiler details

Issues are far too complicated for this course

Same applies to loading and storing floating–point

- Actual operations on it are very rarely atomic

Beyond that (e.g. struct or complex), forget it

Nowait (1)

A work–sharing construct has barrier at its end
Consider a parallel region with several of them
Would it run faster if the barrier were removed?

- MPI experience is generally “no”

It might help with some code, especially SPMD

Fortran: NOWAIT after the !$OMP END ...
C/C++: nowait after the #pragma omp ...

Warning: get it wrong, and you are in real trouble
Need to be very, very careful about aliasing issues

Nowait (2)

This will NOT work – 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
```

Tasking (1)

There are clauses if and final
May suspend current thread to run subthread
- Specification is confusing, so read carefully

Plus an even trickier mergeable clause

Also threadyield, allowing temporary suspension
May be critical if use both tasks and locks
May not be needed with untied, but that’s a guess

Tasking (2)

- But, generally, tasks+locks == Bad News

The OpenMP features do not work well together
If you use tasks+locks or thread–specific state

- Learn about task scheduling and synchronisation

This course avoids that area by simply saying don't

Untied Tasks

- Data are tied to threads, not tasks
  Tasks are tied to arbitrary threads
  But at least they don’t change thread dynamically
  The clause untied can allow them to do so (and more)

  But this will break all thread–specific state
  Including threadprivate, OpenMP thread ids, errno,
  IEEE 754 flags/modes, even C++ exceptions
- And it may even break constructs like critical

⇒ You are strongly advised to avoid untied

Environment Variables

We have already covered OMP_NUM_THREADS

And the settings of OMP_SCHEDULE

OMP_DYNAMIC=true is mainly for SPMD
Allows the number of threads to vary dynamically

OMP_NESTED=true enables nested parallelism
Details are too complicated to cover in this course
Will give just a summary of the intent

SPMD Variants

Ideally, we want as many threads as possible
The compiler and system choose which ones to run
That’s what I call the sea of threads model

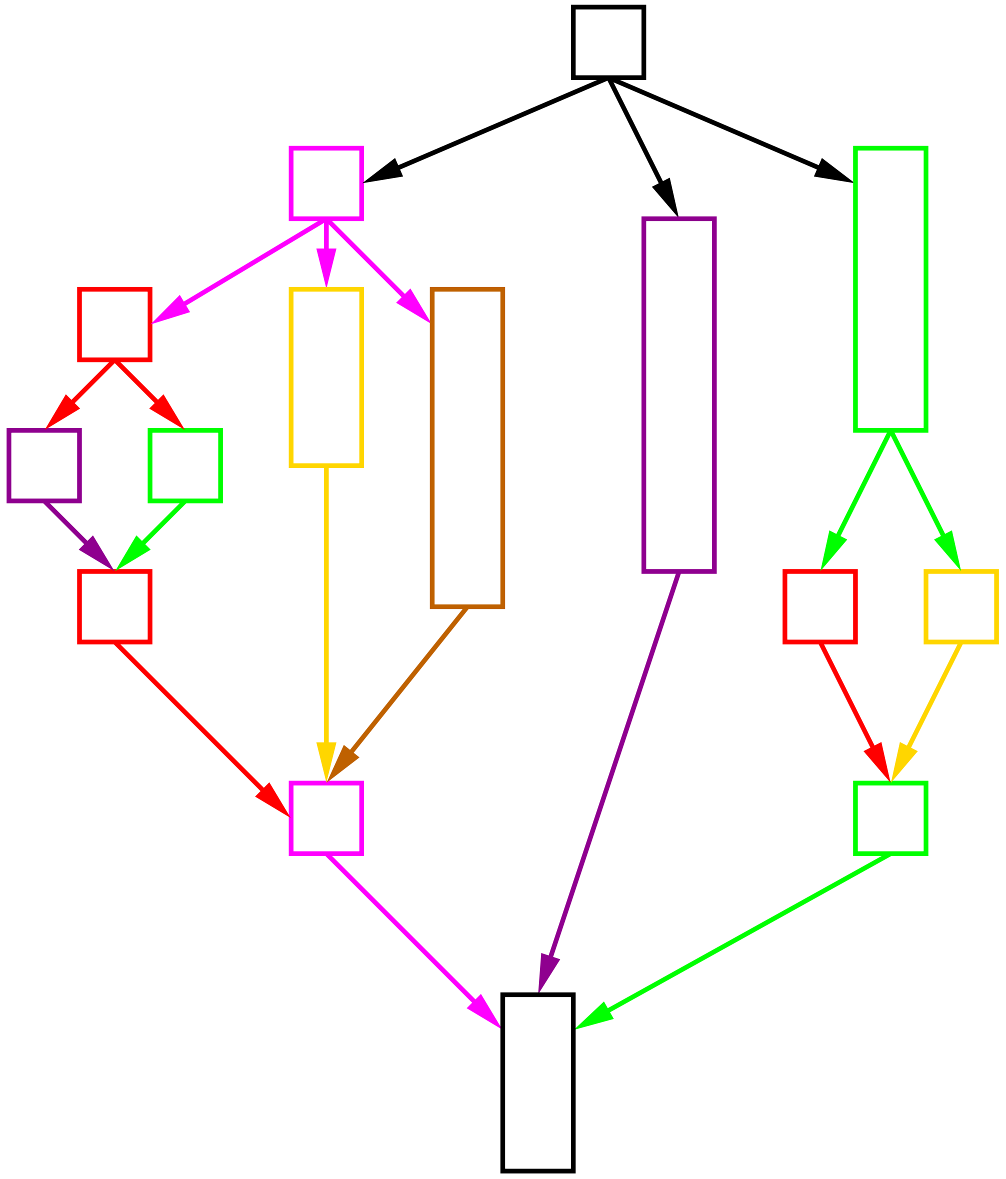
- But OpenMP doesn’t handle that very well

- It doesn’t handle even nested parallelism very well

Where a subthread can spawn a parallel region

But that can be done, and can be useful
Doing it is advanced OpenMP and isn’t covered

Nested SPMD Task Structure



C++ Iterators (1)

OpenMP 3.1 claims to support C++ iterators
Only constraint is must be random access

Don't you believe it!

- Class and iterator methods must be pure

Rather like const, but applies to updates, too
Rules are stronger than C++ uses for const

The main rule is no side–effects in the methods
And no reference to anything that might change
E.g. container elements must not move or be added

C++ Iterators (2)

As far as the library goes, these should be safe:

- Classes vector, deque and array
  and probably basic_string and string

- Use Fortran rules for iterators in OpenMP for

- Access elements using only operators '*' and '[ ]'

- And model your own classes on the above

Locks (1)

OpenMP has facilities for thread locking
Essentially a dynamic form of critical
But I do not recommend using locking

- Easy to cause deadlock or dire livelock
- Often cause very poor performance or worse
- Generally indicate the program design is wrong

Locks (2)

But, if you really must use them:

Two kinds: simple locks and nested locks
Usually called simple and recursive mutexes
OpenMP also uses setting rather than locking

- Do NOT mix them in any way
  OR with critical or master

Almost sure sign of a completely broken design

Simple Locks

- Simple locks are set or unset
Once a thread has set a lock, it owns that lock
If it already owns it, that is undefined behaviour

- Another thread setting it waits until it is unset

- Only the owning thread can unset a lock
If not, that is undefined behaviour

Examples are given only for simple locks

Nested Locks

- Nested locks are very similar in most respects
Only difference is that an owning thread can set a lock
What that does is to increment a lock count

- Similarly, unsetting just decrements the lock count
Only when that is zero does the lock become unset
Undefined behaviour if not owned or count is zero

Generally, avoid these, but they have some uses
Nothing that you can't program in other ways
See the specification for details on their use

Initialization etc.

Lock variables should be static or SAVE
OpenMP doesn't say this, but not doing so may fail
Best to have file scope or be in a module

- Initialise and destroy in serial code
Could do in a single, synchronised thread – with care

- Must initialise before any other use
Preferably destroy after last use as lock
Could then reinitialise, but not recommended

Examples

C/C++:

```
static omp_lock_t lock ;

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

Fortran:

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

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

Locking and Unlocking

C/C++:

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

Fortran:

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

Testing Locks

You can also test whether a lock is set

- If the answer is "no", it also sets the lock

Mustn't test in owning thread for simple locks

- I do NOT recommend using this feature

Trivial to cause livelock or dire performance
Also some extremely subtle consistency problems

Using this to improve performance is very hard

- Using to ensure correctness is a mistake

It almost always indicates a broken design

Synchronisation (1)

Remember flush? Locks have the same issues
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
The following is all that is guaranteed:

> If some data are used only under a lock P,
> Then all such uses will be consistent

That can be extended to serial code as well

- It cannot be extended to other synchronisation

Synchronisation (2)

How can you use locks to force consistency?

A and B must be protected by the same lock

- Using a separate lock for each won't 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

Not just different data, but no ordering assumed

Synchronisation (3)

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 ) ;
```

Synchronisation (4)

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 ) ;
```

Not Covered (1)

Many other things deliberately not covered
Mostly because they are too difficult to teach

- Usually, means very hard to use correctly

Some are hard to implement, and may not be reliable

- Library functions to set OpenMP's state
- The ordered clause (probably not useful)
- And quite a few minor features and details

Plus areas mentioned earlier and not recommended

Not Covered (2)

- OpenMP 3.1 adds a certain amount more

The more useful features have been mentioned

- OpenMP 4.5 adds GPU features – gibber!

You are far better off programming in CUDA

It also adds array sections for C and C++

And a huge amount I wouldn't touch with a bargepole

Not Covered (3)

- Discussion about how to configure your system

This is obviously very system−specific but see:

Parallel Programming: Options and Design

https://www−internal.lsc.phy.cam.ac.uk/nmm1/Parallel/