Introduction to OpenMP

Critical Guidelines

Nick Maclaren

nmm1@cam.ac.uk

September 2019

Apologia (1)

The previous lectures were an oversimplification
“This is a footgun; pull trigger to see how it works”

- Including even critical warnings was confusing

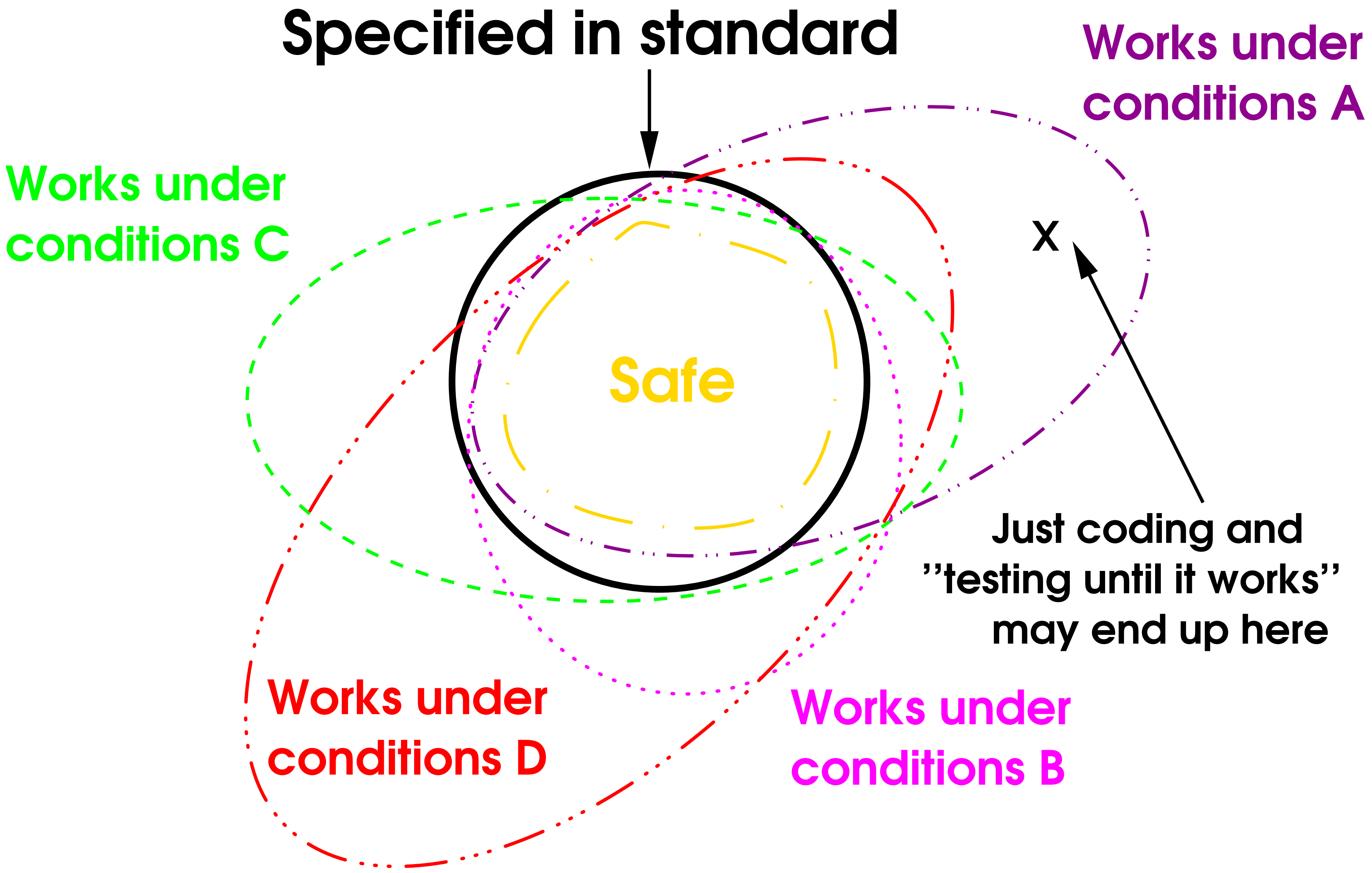
So they were separated out and included here

- There are a few forward references

Some features will be described in next lecture

Mainly what you need to know, but don’t want to
Here is a reminder of the picture we saw at the start

Portability, RAS, etc. of Code



Apologia (2)

This lecture may discourage you from using OpenMP

- Next lecture describes when, why and how to

OpenMP is much trickier to use than MPI
But this course describes how to use it safely

- And it does have some advantages over MPI

- Mainly, keep gotchas out of parallel regions

Outside all of them, you are programming serially

This lecture is about lots and lots of gotchas – sorry

Reminder: KISS

That stands for Keep It Simple and Stupid

- Remember it's rule number one for OpenMP use

Why Is That Critical?

Shared memory programming is seriously tricky

- Doing the actual programming is the easy bit

- Avoiding the ‘gotchas’ is the hard bit

Including deficiencies in the language standards

Worse, deficiencies in the OpenMP specification

Will now cover some of the reasons why this is

- And some guidelines on how to avoid problems

Lecture Structure (1)

Most warnings apply to both Fortran and C/C++

- But C/C++ has many more “gotchas”

Fortran has some specific to it, too

Unfortunately, need to do a lot of language–flipping

One–language warnings may apply to all languages
Take note if you use the equivalent facility
E.g. some Fortran ones apply to C++ as well

Lecture Structure (2)

An example of this is:

- Don't touch volatile in C/C++ – totally broken

Explanation is too complicated for this course

But why not warn about Fortran?

The Fortran standard is much less inconsistent
Rarely recommended in books and Web pages
Very few Fortran programs use volatile

It doesn't work any better, of course

Syntax Warnings

Fortran:

Lines starting !$ and a space are significant
That is an OpenMP Fortran preprocessing line

- Don't start any other comments with !$

C/C++:

Remember C and C++ pragmas get preprocessed

- Don't define OpenMP keywords as macros
- Watch out if using any non–C/C++ headers

Conditional Compilation

I don't recommend this, but if you must

C/C++: the preprocessor symbol _OPENMP is set
To the integer yyyymm, where yyyymm is API date

- Probably no canonical mapping from/to versions!

Fortran: if a line begins with !$ and a space
then the !$ is removed in OpenMP mode

There are more variations, but that is the basics

Fortran PURE and Functions

OpenMP facilities are necessarily impure
Except for the information functions, of course

- Don’t use them in PURE or ELEMENTAL

I don’t advise using them even in functions
This includes in subroutines called from functions
Fortran allows aggressive optimisation of expressions

- Function calls are not always executed

- Same applies to C++ with non–trivial classes

See the notes for the most likely gotchas

Call Chain Issues (1)

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

Call Chain Issues (2)

parallel, worksharing and barrier are collective

- Must execute on all threads in same order

Applies to all loops, conditionals and branching
Plus all function calls in Fortran, anywhere
And in C/C++ when they occur in argument lists,
initializers, constructors and destructors

Only a language lawyer knows what is defined

- As usual, best way to avoid problems is KISS

Sharing PRIVATE Variables

shared has changed meaning, and is inconsistent
Now, it means shared over the current region

E.g. private in a region and shared in nested region
Each parent thread's variable shared with its children

Alert: this is sweating dynamite

Both you and compilers may get confused

- And DON'T make DO/for loop variables shared

Especially in Fortran, because many compilers object

Types in Directives (1)

The OpenMP specification is very sloppy in places
It defines most of the syntax fairly precisely
Leaves many ambiguities in the language bindings

Can pass values in variables to some directives
But it doesn’t specify what types they are allowed

- This is not about variables in data clauses

It’s about the N in schedule (static , N)
And other expressions allowed in some directives

Types in Directives (2)

Here are safe rules for portability and reliability:

- Use default integers, when 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 in C/C++
Note that bool is not allowed in C++

This may well be overkill, but it's not restrictive

C/C++ Directive Use (1)

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: f and then g   or   g and then f

C/C++ Directive Use (2)

- OpenMP directives take the Fortran approach
Any conflicting side−effects are undefined behaviour

- Applies to values in schedule clause
Anywhere you have an expression in a directive

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

Or, using facilities we haven't covered yet:

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

Default Clause

I don't recommend this, but OpenMP does
default(<which>), where which is shared, private etc.

It's hard to describe exactly what it controls
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

Parallel Problems (1)

Most bugs don't show up in simple test cases

Failures are almost always probabilistic
Probability often increases rapidly with threads
See Parallel Programming: Options and Design

- Solution is to be really cautious when coding

- Remember that compilers differ considerably

The more optimisation, the more you are at risk

Parallel Problems (2)

- Don’t just run a test and see if it ‘works’

I.e. that your compiler doesn’t show the problem

- You may well have a probabilistic race−condition

MTBF (mean time between failures) of many hours

When you run a realistic analysis, it may not work
And tracking down such bugs is an EVIL task

- Sorry, but that’s shared−memory threading for you

Example

```
#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
Otherwise, there may be one – and chaos awaits

Debugging Hell

- 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

Failure Rate

Consider a race condition involving K entities
Entities can be threads, locations or both
R is the rate at which interactions occur

- Failure rate is O(R^K) for K $\geq$ 2 (often 3 or 4)

Also when assuming more consistency than exists
See later for details of this nightmare area

A Useful Trick

- You can sometimes use schedule(static,1)

Successive iterations round–robin between threads
Helps to expose conflict between adjacent iterations

Reorganising loops achieves this more generally

- Tends to work best when is most inefficient!

Sharing Memory

Updates may not transfer until you synchronise
But they may, which is deceptive

Memory will synchronise itself automatically

- Now, later, sometime, mañana, faoi dheireadh

So incorrect programs often work – usually
But may fail, occasionally and unpredictably

- Any diagnostics will often cause them to vanish

Makes it utterly evil investigating data races

Memory Models

Shared memory seems simple, but isn’t
‘Obvious orderings’ often fail to hold

Too complicated (and evil) to cover in this course
The following is just an indication of the issues

Suitable key phrases to look up include:

> Data Races / Race Conditions
> Sequential Consistency
> Strong and Weak Memory Models
> Dekker’s Algorithm

For Masochists Only

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

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

Follow the guidelines here, and can ignore them

- Start to be clever and you had better study them

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

Another Consistency Problem

```
Thread 1
A = 1
```

```
Thread 2
B = 1
```

```
Thread 3
X = A
Y = B
print X, Y
```

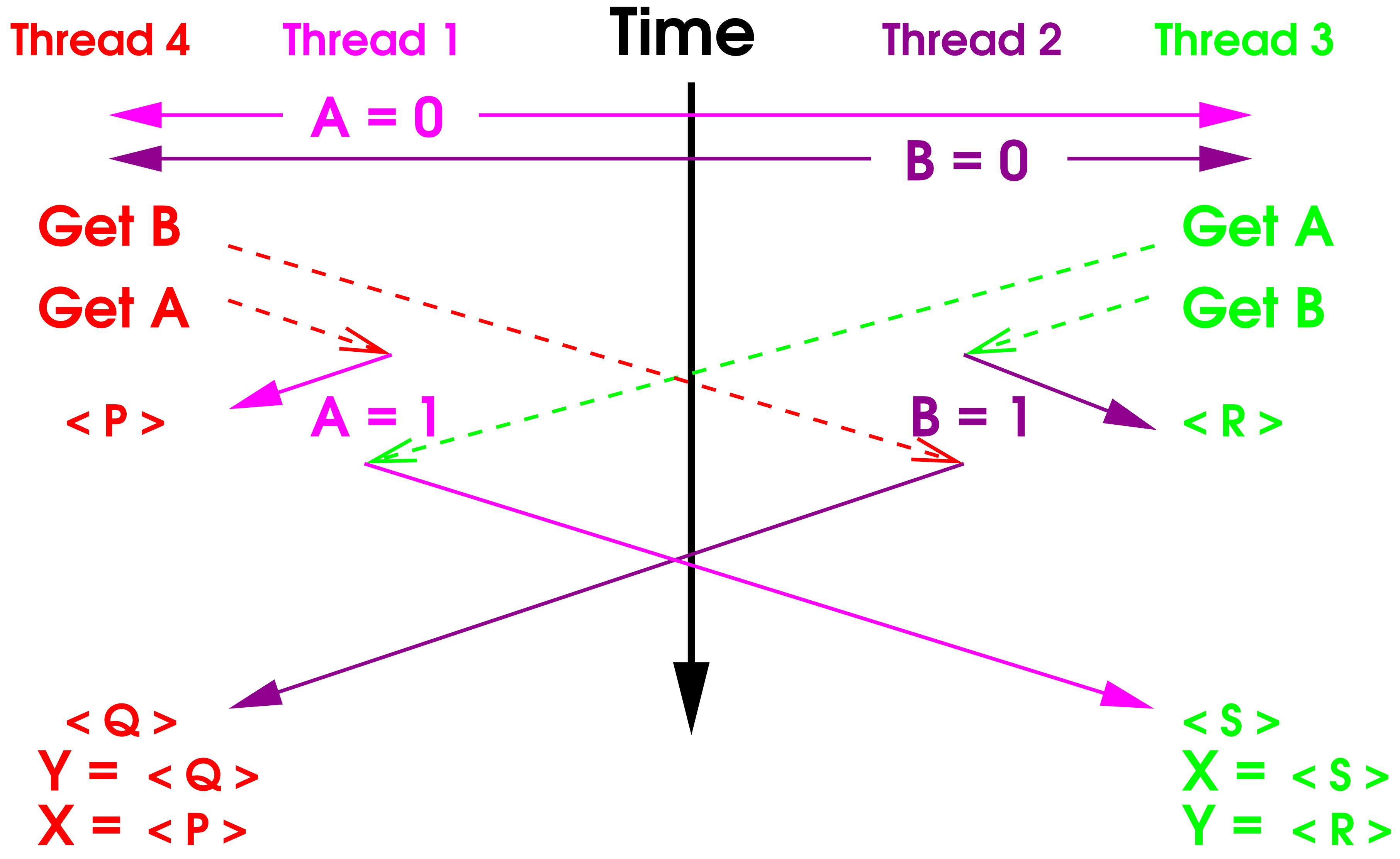
Now, did A get set first or did B ?

```
Thread 4
Y = B
X = A
print X, Y
```

1 0 – i.e. A did

0 1 – i.e. B did

How That Happens



<x> means a temporary location

Consistency Issues

But that’s just due to too much optimisation, isn’t it?

NO!!!

It is allowed by all of C99, C++03 and Fortran
AND it is one of the common hardware optimisations
$\Rightarrow$ It can happen even in unoptimised code

- Regard parallel time as being like special relativity

Different observers may see different global orderings

OpenMP Debugging

- Failure is often unpredictably incorrect behaviour

- Variables can change value ‘for no reason’
Failures are critically time–dependent

- Serial debuggers will usually get confused
Even many parallel debuggers often get confused
Especially if you have an aliasing bug

- A debugger changes a program’s behaviour
Same applies to diagnostic code or output
Problems can change, disappear and appear

We’re All Doomed!

That sounds like a counsel of despair

- But there are things you can do
That is why I have so many ‘dos’ and ‘don’ts’

- Object is to not make errors in the first place
Especially ones that are hard to debug

- Try to avoid ever needing a debugger
Follow the guidelines here and you rarely will

Next lecture will describe some tools that may help

Data Environment

The OpenMP specification is a bit sloppy here, too
Compilers vary and simple tests can be misleading

- Write very conservative code and don't be 'clever'

It is also a very hard issue to get your head around
It dominates bugs, debugging and tuning

- Rule number two is KISS, KISS

Second KISS is Keep It Separate, Stupid
I.e. keep private and shared very distinct

Keep It Separate

This relates to private versus shared variables
OpenMP is such that the same name can mean both
Also applies to the use of pointers

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

If You Must Do It

Precise rules are very complicated – ignore them
Best to think in terms of following model:

Private versions exist only in parallel regions

- Values are undefined on entry, and lost on exit

- Don’t access shared version in parallel region

Shared variable becomes undefined in 2.5
Preserved if not accessed from 3.0 on

All except where behaviour is explicitly specified

Global Data

Other procedures can access global data directly
That term means subroutines and functions

Fortran module data and common
C external and static and C++ class members
And, of course, using pointers

If you access that as private in a parallel region
then never access it directly during that
⇒ Either is fine, both leads to chaos

- It's easier to obey this rule than describe it

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    ! 123 or undefined
```

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 ) ;    // 123 or undefined
```

Classes of Code

There are three important classes of code:

- Serial code, outside all parallel regions
- Synchronised code, protected by critical, single (perhaps 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

Synchronisation

A var. accessed in both synchronised and other code
must be protected against race conditions
Not needed for read−only variables, of course

- 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 barriered master

Using a fully−synchronised form is safest

Calling Procedures (1)

A construct has an associated lexical scope
The actual text 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

Calling Procedures (2)

But what rules apply to a procedure called in that?
Such procedures are called several times in parallel

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

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

Calling Procedures (3)

- Start with code compiled with an OpenMP option

Those are almost identical to the lexical scope ones

Will repeat them, but more precisely than before

All these inherit from what they refer to:

- All Fortran arguments except VALUE
- C++ reference arguments
- See later for pointers, which are not easy

Calling Procedures (4)

The following variables are shared:

- Any form of global or static data
  - Fortran module variables, COMMON, SAVE
    - Including all initialised variables
  - C/C++ static and extern, C++ class members
- C++ const variables with no mutable members

Calling Procedures (5)

The following variables are private:

- Anything explicitly declared as threadprivate

You can use this to override defaults, but be careful

- C/C++ automatic variables and Fortran VALUE
  inc. C/C++ non–reference arguments

Remember Fortran initialisation sets SAVE
And Fortran local variables use the default rules

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

C/C++ programmers – watch out for nested loops

Fortran Association (1)

- Fortran passes arguments by association
Implemented as either reference or copy–in/copy–out
The latter is the one that causes the problems

- Often described as the array copying problem
Applies to both scalar and array arguments
Generally, dependent on compiler and optimisation

- Will necessarily happen in some circumstances:
Passing assumed–shape or non–contiguous section
to explicit–shape or assumed–size arguments
Often viewed as passing Fortran 90 data to Fortran 77

Fortran Association (2)

OpenMP barrier operates only on current data
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
Not just to calling the procedure that calls barrier

It's not a catastrophic problem, if you watch out for it

C++ Classes

- C++ potentially has similar issues to Fortran

This applies to both user classes and the C++ STL
It is relatively unlikely to hit them, but it is possible

Most likely for non–trivial move/copy constructors
    Which also includes copy assignment
And when using callbacks from the STL or similar

- Assume such things may be called in parallel

You should avoid using barriers in such places
Unless you are sure the specification makes it safe

Dynamic Storage Duration (1)

OpenMP includes the following in two critical places:
Objects with dynamic storage duration are shared
OpenMP 2.5 used “heap−allocated storage”!

%deity alone knows what that means – Watch Out!

Only C++ has “dynamic storage duration”

C++ int * a = new int [10] ;

We can extend that to:

C int * a = malloc (10*sizeof(int)) ;

Fortran ALLOCATE on POINTER variables

Dynamic Storage Duration (1)

But sharing is a property of variables not objects
All of those are executed and not declared!
Each thread will necessarily do them separately

⇒ So they are necessarily private to the thread

- It probably means the pointer may be stored
  In a shared variable used only by that thread

After synchronisation may be used as shared data

Language Built-ins

The Fortran intrinsics and the C/C++ library
I/O and exceptions are described later

OpenMP specifies that they are all thread–safe

- Some cases when that is obviously impossible

Fortran is fine, except for two procedures
C++ is OK, too, but its inheritance from C is not
⇒ Watch out! That's quite a lot of the C++ library

- Here are some rules that are generally reliable

Aside: POSIX

POSIX now includes the whole of C99
And specifies parallel (threading) semantics

Well, in theory

However, this area of POSIX makes very little sense
Some obvious impossibilities, just like OpenMP
Unlikely it will match reality on most systems

- So it's best to ignore POSIX in this regard

Program Global State

Never change program state in parallel code

- Do it in the main, serial code and propagate it

- Best to do it before first parallel region

Fortran has very little (e.g. RANDOM_SEED)
C (and so C++) has more (locales, srand etc.)

- Call all of the following from serial code only:
  EXECUTE_COMMAND_LINE,
  RANDOM_SEED,
  system, srand, atexit (and then exit), setlocale

Random Numbers

- OpenMP conflicts with C and POSIX
Using rand unsynchronised may fail horribly
Might fail in Fortran, as well, but less clear

- Simplest solution is to synchronise the calls
That is RANDOM_NUMBER and rand

- The C++ random numbers should also work
If each thread uses a separate engine instance
$\Rightarrow$ But the statistical properties may be poor
Ask me offline about parallel random numbers

Internal String Results

- Some C functions return pointers to internal strings

Often use a single internal string for all threads

- Use all of them within synchronised code only

Copy the data to somewhere safe ASAP
Do that before leaving the synchronised region

Mainly:
  tmpnam, getenv, strerror
  Most of the C functions that return date strings

Other C Library Functions

Some extra ‘gotchas’ for the multibyte functions
Please ask for help if you use those monstrosities

- I/O and exceptions are described later

- Most of the rest of the C library should work
Some of it may be very slow, because of interlocking

And remember:

- C++ inherits a lot from C

C++ Containers (1)

C++ is very poorly specified in this area
The following rules are generally safe

- const functions and methods are read–only

- Using an iterator it may read its container
  Indexing (it[n]) will do so with some libraries
  Only dereferencing (*it and it–>mem) don't
  And (it1 == it2 and it != it2) probably don't

- Just using elements does not use the container
  But assigning to elements and swap may do

C++ Containers (2)

- Updating separate containers does not conflict

- Updating separate elements does not conflict
Except for vector<bool>, where it does

- Replacing elements and swap are OK for
 basic_string, array, deque and vector
All others may update the container

- Any container update conflicts with all iterators
C++ does not say this, but it is needed for OpenMP

C++ Containers (3)

In summary, the following is safe in parallel regions

- Anything that is entirely read–only
- Updating separate containers or iterators
- Updating but not exchanging separate elements
- Updating separate elements using any operation

basic_string, array, deque and vector (not <bool>)

⇒ Use iterators in omp for only with these

Other C++ Libraries (1)

- Update only separate objects in parallel regions
Watch out for indirect objects like locales
Traits and similar read–only classes are no problem

- Don’t use allocators – they update global state
Precise rules are too complicated to describe

- Avoid updating valarray and smart pointers
The C++ and OpenMP wording is just too confusing
Doing that to completely separate ones is safe

Other C++ Libraries (2)

- Don’t use C++ threads, atomics or futures
May conflict with the OpenMP implementation

- Use separate streams or see earlier comments
Even opening and closing separate files is risky

- Remember that the C library has worse problems
See elsewhere in this lecture for a description of those

Non-OpenMP Procedures

- Anything NOT compiled with an OpenMP option
Inc. libraries that don’t explicitly support OpenMP

- Can always call such procedures from serial code
  And almost always from synchronised code

- Calling in parallel is undefined behaviour
Check if you need to set a special library for OpenMP

There are a few other things that are fairly safe
Won’t cover here, but please ask if you need to

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 – don't
Needing them is a sure sign of being out of control

- NEVER make anything EQUIVALENCEd private
Not even if all EQUIVALENCEd names are private
EQUIVALENCE shouldn't be used, anyway

PRIVATE ALLOCATABLE

These are the 2.5 rules, but are good practice:

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

The same will apply to many C++ classes

⇒ And don't access the shared version!
That applies whether or not you deallocate it

C/C++ Private Arrays

DON'T

C/C++ arrays are often not really arrays
Except when defining space, they are usually pointers

- The C/C++ standard is badly ambiguous here

The OpenMP specification is inconsistent with them

⇒ C/C++ private arrays do not work reliably

Pointers (1)

Pointers in parallel code are a snare and a delusion
Many experts think languages shouldn't have them
Let's not be dogmatic, but stick to the following:

- Use shared pointers to point to shared data
Set or change them only in serial code
Can then read their values anywhere in parallel code

- Use private pointers to point to private data
And use them only within the same thread
Become undefined on leaving the parallel region

Pointers (2)

But what about the following?

- Changing shared pointers in synchronised code
- Using private pointers to read–only shared data

Theoretically, those should work, reliably
But there are some evil language standard issues
In practice, doing that is living dangerously
If you need to do this, watch out

Private Pointers

Treat pointers (even C/C++) like Fortran allocatable

- Set them to NULL before entering parallel region
  And again before leaving the parallel region

⇒ And never access the shared version inside!

Remember to free malloced memory first, if needed
And use DEALLOCATE if necessary in Fortran

No problem if declared inside the parallel region

Cray Pointers (Fortran)

DON'T

I don't advise their use even in serial code

If you really have to, treat them as shared
And never let OpenMP default them to private

- But they are a minefield together with OpenMP

Reduction Constraints (1)

I advise being cautious, whatever OpenMP implies

- OpenMP says the variable must be shared

That is so that the compiler can treat it specially

Some evil problems with argument passing

- Don’t pass the variable as an argument

- Don’t set a pointer to the variable

Reduction Constraints (2)

- Stick to scalars of built–in arithmetic types

Conformable Fortran arrays and sections are OK, too

Any of the integer, real or logical ones

Plus complex, but watch out for C

Use any Fortran KIND or C/C++ size

Most compilers will get those right, or complain

Reduction Constraints (3)

If you need derived types or classes

- Need OpenMP 3.1 or even 4.0 support
- Check that they work in your compiler
- Don’t rely on them in another compiler
- Write them strictly functionally

Same applies to using them in firstprivate etc.

Safest I/O Usage

This is a problem in all parallel languages
OpenMP says almost nothing, leaving it ambiguous

- The following is what is almost certainly safe
This 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
Definitely do all I/O on stdin, stdout and stderr there

Fancier I/O (1)

Often that’s not feasible, or at least very inconvenient
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

Will also work on clusters, usually not as you expect

- Read from stdin in the global master only

Synchronising its use may work, but won’t always

Fancier I/O (2)

And you must do all of the following:

- Set line buffering on stdout and stderr in C/C++
E.g. 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
Don't assume that stdout $\not\equiv$ stderr

Fancier I/O (3)

If you can't set line–buffering (as in Fortran)

Before leaving every synchronised section with I/O:

- Use the FLUSH(<unit>) statement in Fortran
If you don't have it, try using CALL FLUSH(<unit>)

- Call fflush(<stream>) in C/C++

Regrettably, this applies even for diagnostics
Use one or the other technique even for stderr

Exceptions (1)

- Cross–context exception handling is pure poison

Handle them only in the raising context

- This includes errno, C++ exceptions etc.

But what is a context in this sense?

- A parallel or work–sharing or similar construct

Anywhere OpenMP may switch system thread

- And remember this means both entry and exit

And pretty well anywhere in an untied task

Exceptions (2)

- Exceptions are bound to a system thread
May well not be the same as an OpenMP thread

- Exceptions become undefined at every boundary
I.e. entry/exit of the closest enclosing construct

- Never include a construct in a try block
Or do the equivalent actions using setjmp/longjmp

- Don't trust the value of errno across a boundary

Signal Handling

DON'T

Please contact me if you really need to

- Words fail me about how broken this area is

IEEE 754 Facilities

- Don't use the fancy IEEE 754 facilities
Available (sometimes) in Fortran 2003 and C99
Associate with system threads, not OpenMP ones

- I don't recommend using them in C99, anyway
C99 got them catastrophically wrong

There are some things that can be done reliably
But they are too complicated to be worth describing

- Please ask for help if you want to do that

Native System Threads

OpenMP uses POSIX or Microsoft threads
But OpenMP threads may not the same as those

- Don’t use them 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 reasons are too complicated to describe here
But include signal handling and scheduling
As well as the thread state mentioned above

Compiler Bugs? (1)

- 95% of ‘compiler bugs’ aren’t that at all
Typically user errors (e.g. standards breaches)

- Unfortunately, that is only 90% for OpenMP
Even when the OpenMP specification is OK

In 1999, only a few Fortran compilers worked at all
By 2006, almost all Fortran and many C/C++ did
Today, even C/C++ ones work for simple use

Performance is another matter entirely

Compiler Bugs? (2)

- Must locate cause before knowing whose bug

Even in simple examples like ones in course

I spent a day tracking one trivial one down

- Also most bugs are not reproducible

Major factors in exposing them include:

- More independent cores (even hyperthreading)
- Complexity of code and synchronisation
- Higher interaction rate between threads

Compiler Bugs? (3)

Solution: KISS !

May not eliminate bugs, but helps to identify them
First step to fixing yours or bypassing theirs

- Triple check your code against the specification
Trivial breaches often cause extreme effects

And follow the guidelines in this course