Software Design and Development

Shared-Memory Programming

Nick Maclaren

nmm1@cam.ac.uk

October 2019

Basic Model

Multiple independent threads of execution
(Almost) all memory is accessible to all threads
This is Shared Memory Processing (i.e. SMP)

- But computers don't work that way

Cores are separate CPUs, linked by a network

Programming models are intended to resolve this
They impose enough restrictions to make it usable

Lecture is some commonly–misunderstood issues

General Approach

- Start with a well–structured serial program

Most time spent in small number of components

Time the program on realistic data (need not be huge)
Need component–level timings (e.g. major functions)
Some GUI tools do that, gprof, or roll your own

- Must have clean interfaces and be computational

Critical aspect here is to minimise aliasing
Unfixed aliasing means data races and failure

Libraries

- Look for places with a standard algorithm

There are several good threaded libraries around
Linear algebra is best covered, but there are others
NAG SMP, MKL, ACML and more

Is so, convert to using such a library, and try it out
Now, retime the program and see where time is going

- Don't even attempt to convert whole program

Do it component by component, where possible
Remainder of lecture is how to do that

Key to Success (1)

General threading is too hard for mere mortals
See Hoare's Communicating Sequential Processes!

- First key to success is a good design paradigm
Nowadays, often called a programming pattern

- Second key to success is strict discipline
Important for all programming, critical for this

- Incorrect programs often seem to work
Pass all testing, nothing flagged by debuggers etc.
$\Rightarrow$ But, in real use, often get wrong answers

Key to Success (2)

Will describe why this is so hard, later

- Even top experts don't trust their intuition

For some code, possible to prove 'correctness'
But not feasible for most practical programs

Naturally, it helps to use a proven design
Or one simple enough to be obviously correct
Need only to ensure that the code follows that

Common Designs

Data parallel – think parallel matrix operations
Simple tasking – divide into independent tasks
Data flow – agents connected by a DAG

BUT... – problem is to avoid data races etc.

All accesses need to be independent or synchronised

- Synchronisation must match your chosen design

Including between initialisation and read–only access

SMP Programming

Most thread use is as shared–memory processes
E.g. Web servers/browsers – very little aliasing
Even so, data races sometimes cause problems

For performance, always uses a language extension
Library solutions (e.g. POSIX) don't work reliably
Only the compiler can ensure data are synchronised

- Currently, almost most all such use is OpenMP

Using C++ 11 threading is very much harder
Intel have TBB and Cilk – I am not impressed
Plus a lot of less popular mechanisms

Data Races etc.

⇒ Data races and similar are the main problem

Strictly, data races are two accesses to same location
At least one non–atomic, and not synchronised
But two other aspects cause very similar failures

- Unsynchronised atomic actions in ‘wrong order’
This is the memory model problem – see later

- Data being held in registers or argument copying
Both Fortran and C++ can do this unexpectedly

Pure Functions

Standard meaning (pure(1)) is no side−effects
I.e. no update of anything external to the function
But another aspect is very often needed

- No dependency on anything non−constant
  over a potentially parallel region, that is

This (pure(2)) can be very tricky, and deceptive

Need to consider external read−only data
Because it might be changed by some other thread

A pure(2) function has no data races with anything

Pure(1) But Not Pure(2)

```
int x;
void fred ( int x ) { return x ; }
```

No problem, in itself, but what about?

```
< thread 1 >
x = ... ;
< thread 2 >
y = fred ( ) ;
```

C++ Methods

Many C++ (and Fortran) methods are tricky
Compiler can invent, remove and reorder calls

Copy/move constructors, assignment, destructors
Last is evil if you are using a garbage collector
Add access methods and iterators to that list
Plus any call–backs from the STL

- Make all such methods pure(2) if you can

Not critical, but may help to preserve your sanity

Execution Order

Order of function calls is often unpredictable
Statement sequence is defined and reliable

- Beyond that, leave to language lawyers

And, even then, don't bet on compilers following it!

- If synchronising, watch out for conditionals etc.

If a function is not called, chaos awaits!

Best is for functions to be pure(2) – no problem
If not, use separate statements to enforce ordering

```
a = fred ( ) ;        b = a + joe ( ) ;
```

Call Chain Issues

In all languages, watch out for code like:

```
void fred ( . . . ) {
    /* This */ double a = joe ( ) , b = bert ( ) ;
    /* or */ d = bill ( bert ( ) , joe () ) ;
    /* or */ d = bert ( ) + joe ( ) ) ;
    /* or */ x [ fred ( ) ] = joe ( ) ) ;
    /* plus, if synchronising */
    /* or */ if ( . . . ) { b = joe ( ) ;  }
    /* or */ for ( i = 0 ; i < n  ; ++ i ) { c = joe ( ) ;  }
}
void joe ( void ) { < do something parallel > }
void bert ( void ) { < do something parallel > }
```

Copied Data (1)

Active values are often kept in registers

- Including over function calls, in many cases

Some Fortran and C/C++ arguments
Can be implemented as copy–in, copy–out or both
Often described as the array copying problem

- Applies to both scalar and array arguments

Most likely for non–trivial move/copy constructors
which also includes copy assignment
And when using callbacks from the STL or similar
Can occur for reference initialisation

Copied Data (2)

Must avoid this for data shared between threads
In all function calls etc. in the call chain
Methods are very language– and model–dependent

- Don’t pass as C/C++ value arguments

Flag OpenMP arguments as shared at all levels

- Don’t change properties initialising a reference

Remember that arguments use initialisation
However `const <type>& r = v;` is OK

Copied Data (2)

And don’t trust the STL’s data passing an inch
Either in callbacks or returned results
Check the specification if copying is allowed

E.g. values passed to comparisons in sorting
Or const <type> & x = min (y , z) ;

In Fortran use ASYNCHRONOUS if you can
If not, see my other courses for what to do

Cache Synchronisation

Affects data races and ordering of atomic actions

For speed, caches are not synchronised immediately
Memory will synchronise itself automatically

- Now, later, sometime, mañana, faoi dh eireadh

There are special instructions to force synchronisation
Updates may not transfer until you synchronise
But they may, which is deceptive

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

Atomic (1)

- The term atomic is seriously ambiguous

Also means not interruptible or in hardware

Here, means it happens apparently 'instantaneously'

Therefore you never have a data race, as such

Must use atomic operations on variable

- Synchronisation also affects atomic accesses

Can mix atomic and non–atomic only in single thread

The rest of this is about unsynchronised accesses

Atomic (2)

Safe data types are selected integer values

Including types derived from those

Of sizes 1, 2, 4 and usually 8 bytes

Which are aligned on a multiple of their size

Languages also usually allow selected pointers

Anything beyond that is best protected by locking

- And, yes, that means floating–point etc.

- Aside: don't touch volatile – totally broken

The C standard uses it for two incompatible purposes

'Device control' and interrupt handling

It does not specify atomicity for parallelism

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

But, by default, you may not even get that
http://www.cl.cam.ac.uk/~pes20/...
.../weakmemory/index.html

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 , POWER and ARM

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

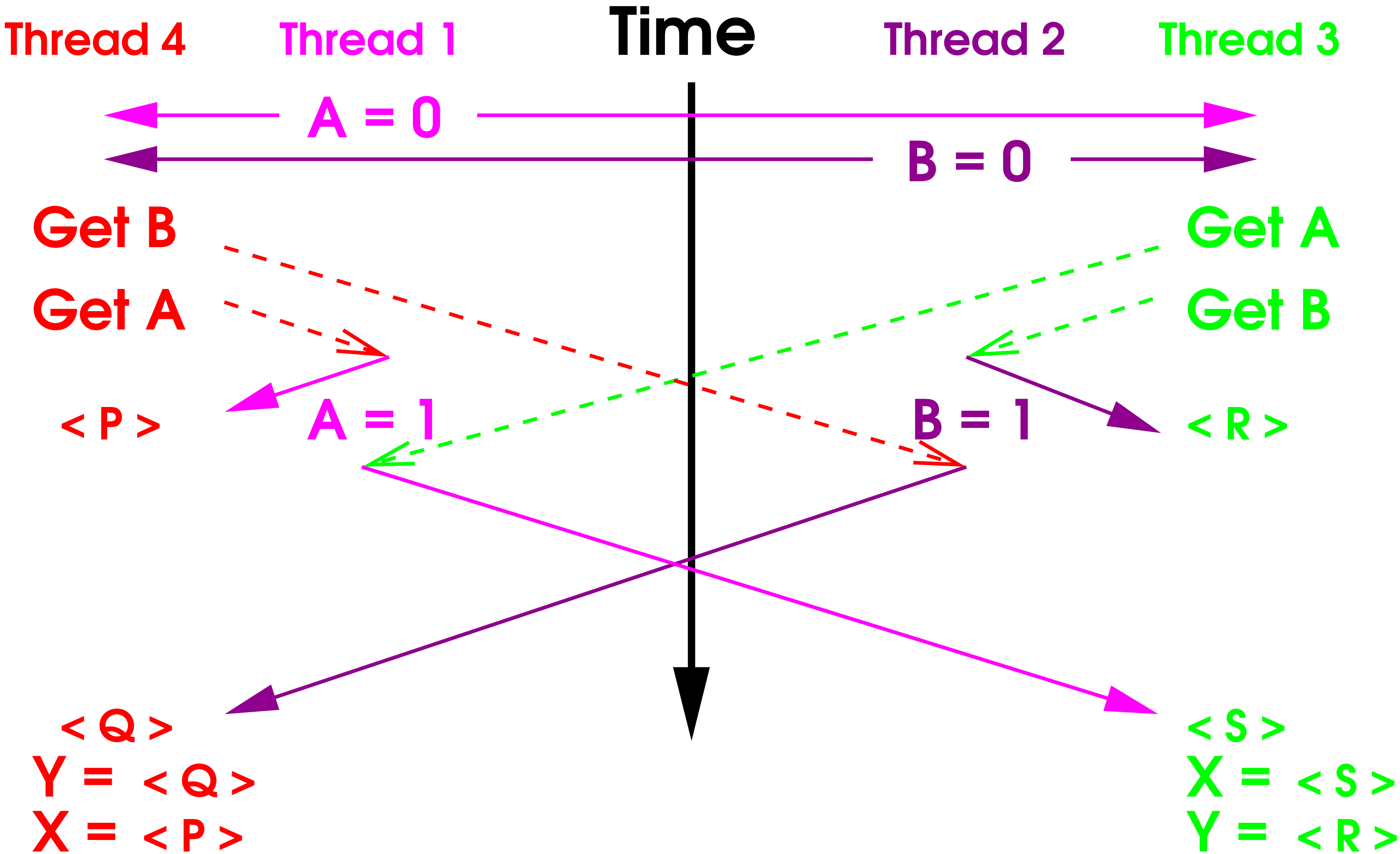
X = A

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
⇒ It can happen even in unoptimised code

- Regard parallel time as being like special relativity

Different observers may see different global orderings

In extreme cases (e.g. IBM Power), it's even worse

Maintaining Your Sanity

In OpenMP, specify seq_cst atomics
In C++, sequential consistency is default
In Fortran, consistency is always unspecified
POSIX does not have atomics, as such

Even simple update (e.g. ++ x ;) may be inefficient
Capture (e.g. y = x ++ ;) almost certainly will

As will using any fancy data types, even if allowed
Reminder: floating–point is fancy in this respect

Synchronisation (1)

Barriers are simple – synchronise all threads
Using them on a subset of threads is not simple

Common mechanisms include locks, critical sections,
mutexes, condition variables
All roughly equivalent – first two are simplest

Fortran uses very different primitives

- But may not be sequentially consistent

In C++, they probably are, but it's murky
In OpenMP and Fortran, it's not specified

Synchronisation (2)

How can you force consistency between A and B?

- A and B must be protected by the same ‘lock’
Using a separate ‘lock’ for each won’t work

- Protect everything to be made consistent
Either by the ‘lock’ or by serialising it from it

- Separately locked data should be independent
Not just different data, but no ordering assumed

Deadlock

This is when two or more threads are waiting
and none can make progress until another does

One of the most common errors when using locks

- Risk when holding a lock and setting another

Simplest solution is not to do that

If can't, design your locking logic – and KISS

Livelock

Two or more threads are in an indefinite loop
In theory, this will always terminate, eventually

- The logic or scheduling means it doesn't
Or such loops sometimes become ridiculously slow

- You need to think in terms of the control flow
Specifically, indefinite looping, however it's done

- Avoid one thread's control depending on another's
That's overkill, but it's the only simple rule

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

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

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
That was covered above

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

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 thread specifications

OpenMP is ghastly, POSIX is worse
C++ language is OK, but library is hopeless

Debugging Tools (1)

At least Intel Inspector, valgrind and more
I tested those for OpenMP and rejected them
Both clang and gcc/g++ have thread sanitizers
I haven't tried them, but they may work better

OpenMP, C++ threads and POSIX are different
May work for only some uses in some of those
Symptoms are false error reports and missed errors

Debugging Tools (2)

- They all work by instrumenting the code
Must include all accesses and all synchronisation
Will rarely work for object code (e.g. libraries)

- Costs 5–15× in time and 5–10× in memory

- Will pick up only data races that actually occur
Data– or timing–dependent ones may escape
As may the non data race ones described above

- And optimisation may add or remove data races

Efficiency (NUMA)

Stands for Non Uniform Memory Architecture
In this, ‘closer’ means that access is faster

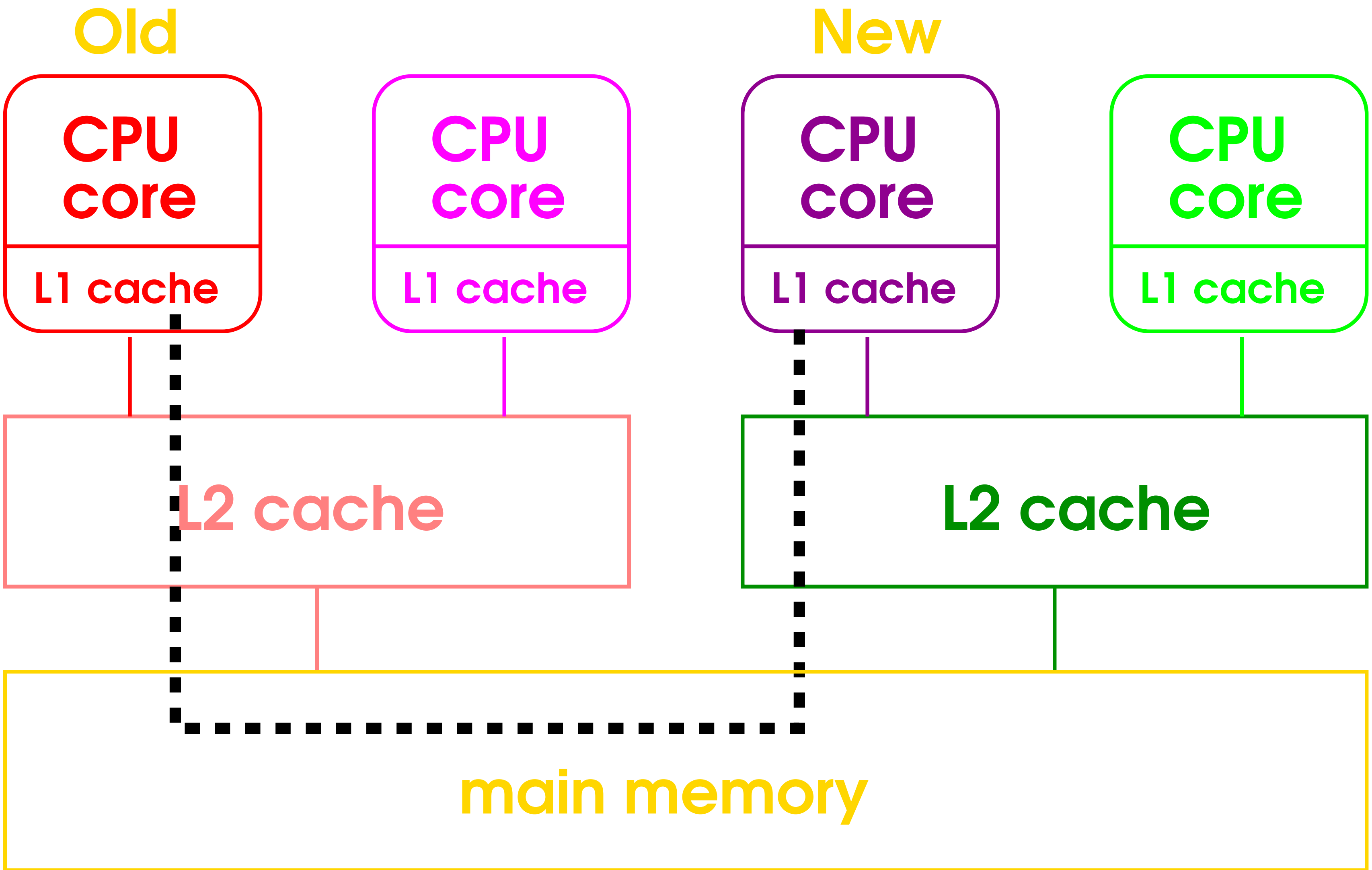
- Some memory is closer to some cores than others

And most levels of cache are not fully shared

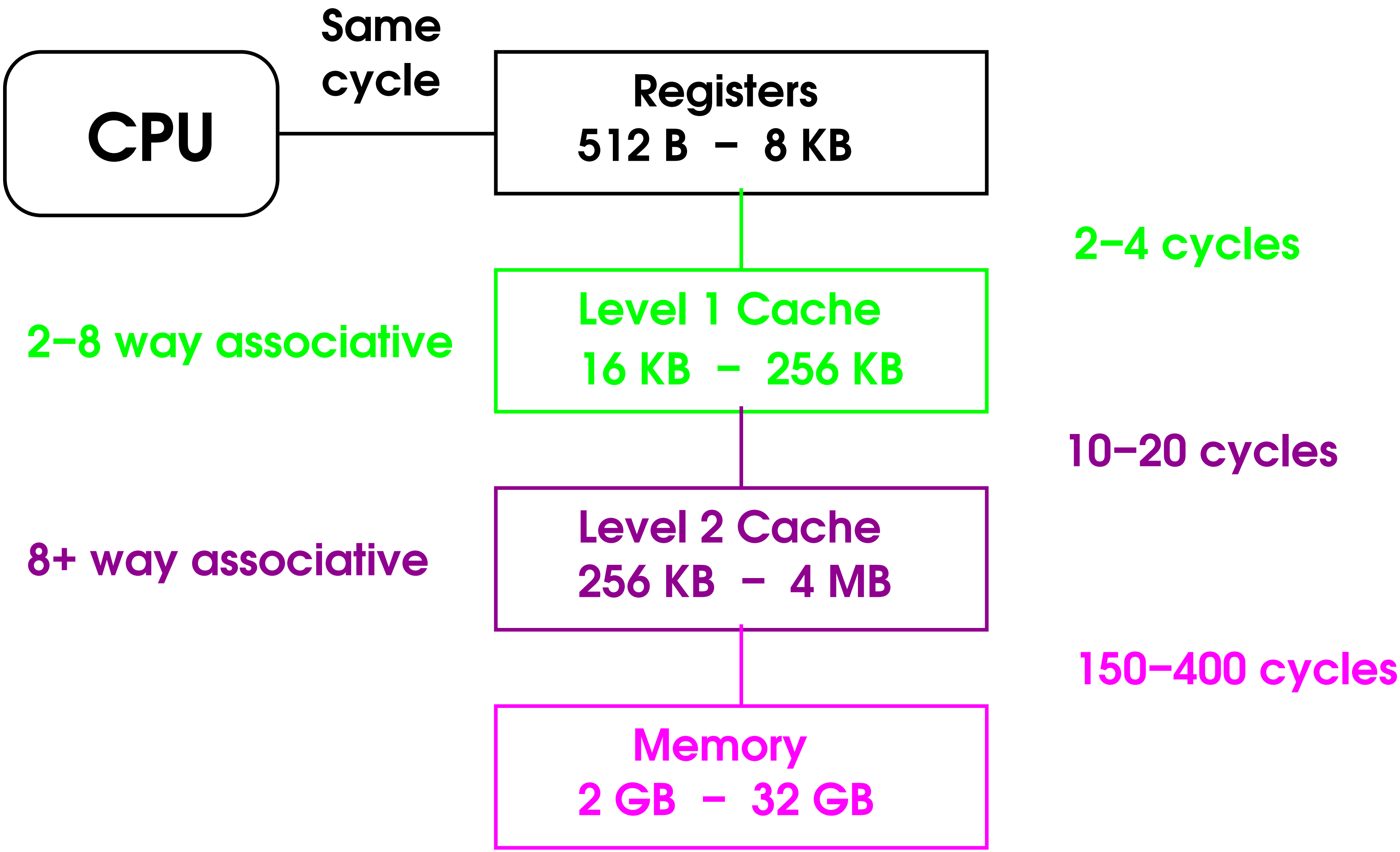
Groups of cores usually share a common access path
The grouping can be different for each level of caching

Thread A writes data and thread C reads it
Its ownership has to be moved from A to C

Moving Ownership



A Typical Cache Hierarchy



Cache Line Sharing

```
            int A[N];
Thread i:   A[i] = <value_i>;
```

- That can be Bad News in critical code

Leads to cache thrashing and dire performance

⇒ Highly active data should be well separated

Cache lines are 32–256 bytes long

- Don’t bother for occasional accesses

The code works – it just runs very slowly

Thread Affinity

Causes major trouble if threads change cores
Everything in the unshared caches must be copied

- Always happens if more active threads than cores

Remember to allow for system threads
⇒ Reducing parallelism is often faster

- Hope that the system's heuristics guess right

Even administrators have very little control over this

Can link threads or processes to CPU cores
Usually needs system privilege and is unreliable

More on Design

Your data may need restructuring for efficiency
Will affect multiple components, some serial

Don’t do this unless the gains look fairly large
Generally means a potential gain of $4\times$ or more

- But new structure usually helps serial performance

- Check that half core count is enough speedup

If not, you had better think about using MPI

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)

The following is safe without synchronisation

- 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>)

Other C++ Facilities

- Update only separate objects in parallel
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

‘Thread-safe’ Library Functions

POSIX and C specify thread–safe functions
And C++ includes almost all the C library
Plus what C++ says about I/O

Well, in theory

They specify some obvious impossibilities
And miss out some ones that are almost certainly safe
Very unlikely they will match reality

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 starting first thread

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

- Using rand unsynchronised may fail horribly
  But it's a ghastly generator, anyway
  Strongly recommended to use a better one

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

- 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

I/O (1)

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

Probably safe to synchronise and change thread, too

- Read from stdin in the initial thread only

Synchronising its use may work, but won't always

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 $\neq$ stderr

Exceptions

- Cross–thread exception handling is pure poison
C++ allows it – but some aspects can't work
Handle them only in the raising thread

- This includes errno, C++ exceptions etc.
Each thread will have its own, independent copy

Signal Handling

DON'T

Please contact me if you really need to

- Words fail me about how broken this area is