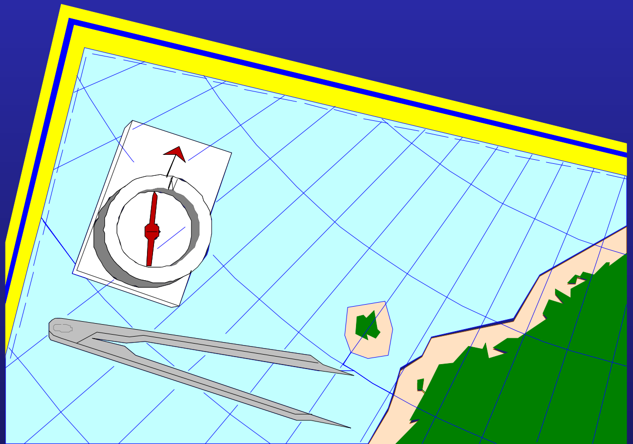
Multithreading dos and don'ts

ACCU 2015, Bristol – June 2015

Hubert Matthews
hubert@oxyware.com



Why this talk?

Don't – why not

- Avoid multithreaded programming if you can
  - It's harder to write, to read, to understand and to test than single-threaded code
  - It may appear to work but may just not have failed sufficiently visibly yet
  - Often distracts from the underlying application problem and focuses developers on technical issues
  - A great consumer of developer time and generator of frustration
  - Often avoidable
  - May not deliver the performance benefits you expect

Do – why

- You need it to access the full power of the machine (measure, don't guess!)
- You need to scale your application and your application is CPU-bound
- You are running in a threaded environment
- There is an obvious parallel decomposition of the problem or algorithm
- Other approaches to covering latency and I/O are worse or not available
- You are brave or a masochist (or have an ego)

Alternatives

Single-threaded approaches

- event-driven code
- asynchronous I/O
  - asio, libaio (linux)
  - overlapped I/O (Windows)
  - non-blocking TCP
  - UDP
- coroutines or fibers
- separate processes

Multi-threaded approaches

- concurrent library (tasks)
  - TBB, PPL
- concurrent library (data)
  - OpenMP
- message passing
  - MPI


start
safe sharing
sharing
decomposition
no sharing
poor perf
(false sharing)
poor perf
(shared writes)
lifetime
TOCTOU
contention
poor perf
(load balancing)
poor perf
(Amdahl's Law)
too many
threads
too few
threads
futures
deletion
appln races
under sync
lock free
over sync
locking
oversubscription
undersubscription
msg passing
errors
poor perf
(serialised)
no perf
(deadlock)
poor perf
(context switch)
poor perf
(coarse grain)
poor perf
(fine grain)

A whole new set of problems

- Getting single-threaded code working well and with good performance can be a challenge
- Multithreading provides a whole new set of ways of getting it wrong or going slower
  - The problems may not show up except under load or at the most inconvenient time
  - They may not be reproducible
  - They will be hard to debug or to measure
  - Knowing which of these problems you have is hard

focus on getting good single-threaded performance
first before going multithreaded

Problem decomposition

- Before considering how to implement a parallel solution you have to split the problem up into pieces and find an algorithm for processing and recombining these pieces
  - This split may be trivial for “embarrassingly parallel problems” or hard (travelling salesman problem)
- Classic approaches
  - Data parallel (sections of an array)
  - Task parallel (web requests)
- Interaction between sub-items is key

Goldilocks

- Parallel approaches have to find the “sweet spot” between two extremes
- Too fine-grained
  - Data – computation dominated by overhead
  - Threads – context switching overhead
- Too coarse-grained
  - Data – load balancing problems
  - Threads – insufficient items to keep threads busy

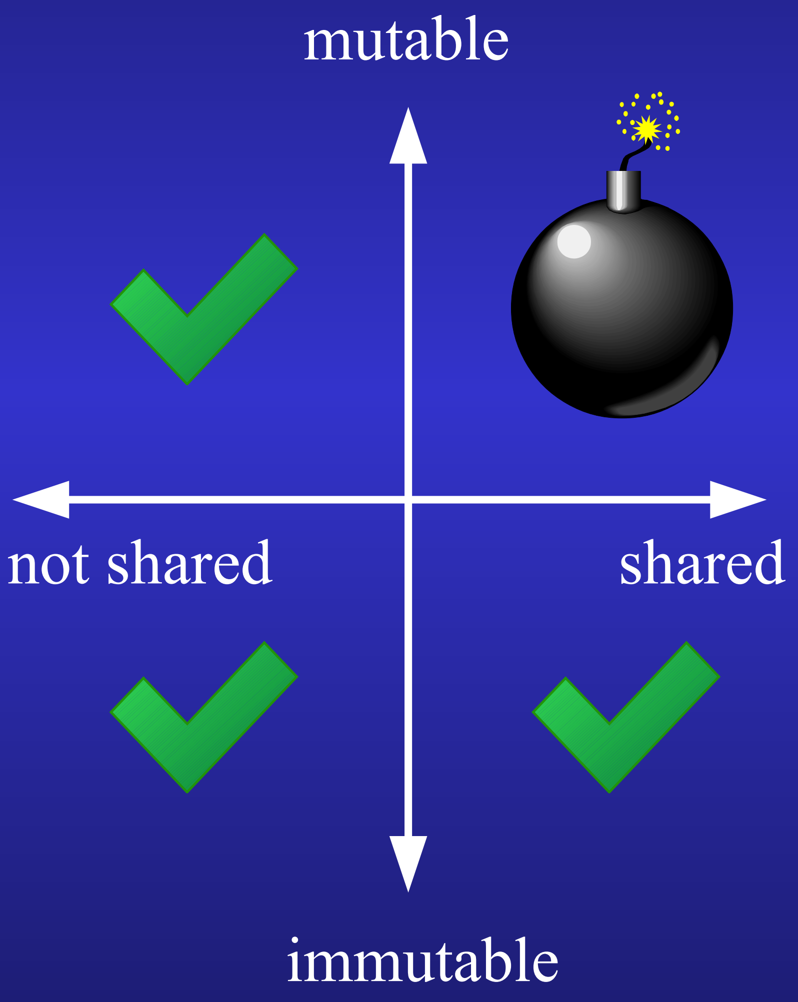
Testing

- Single-threaded code can be unit tested
  - Repeatable results from isolated code
- Multi-threaded code cannot be unit tested easily or reliably
  - Non-deterministic outputs
  - Making them deterministic may be possible
  - Errors are transient (data races)
  - Problems are often performance-related and show up only at scale or under load

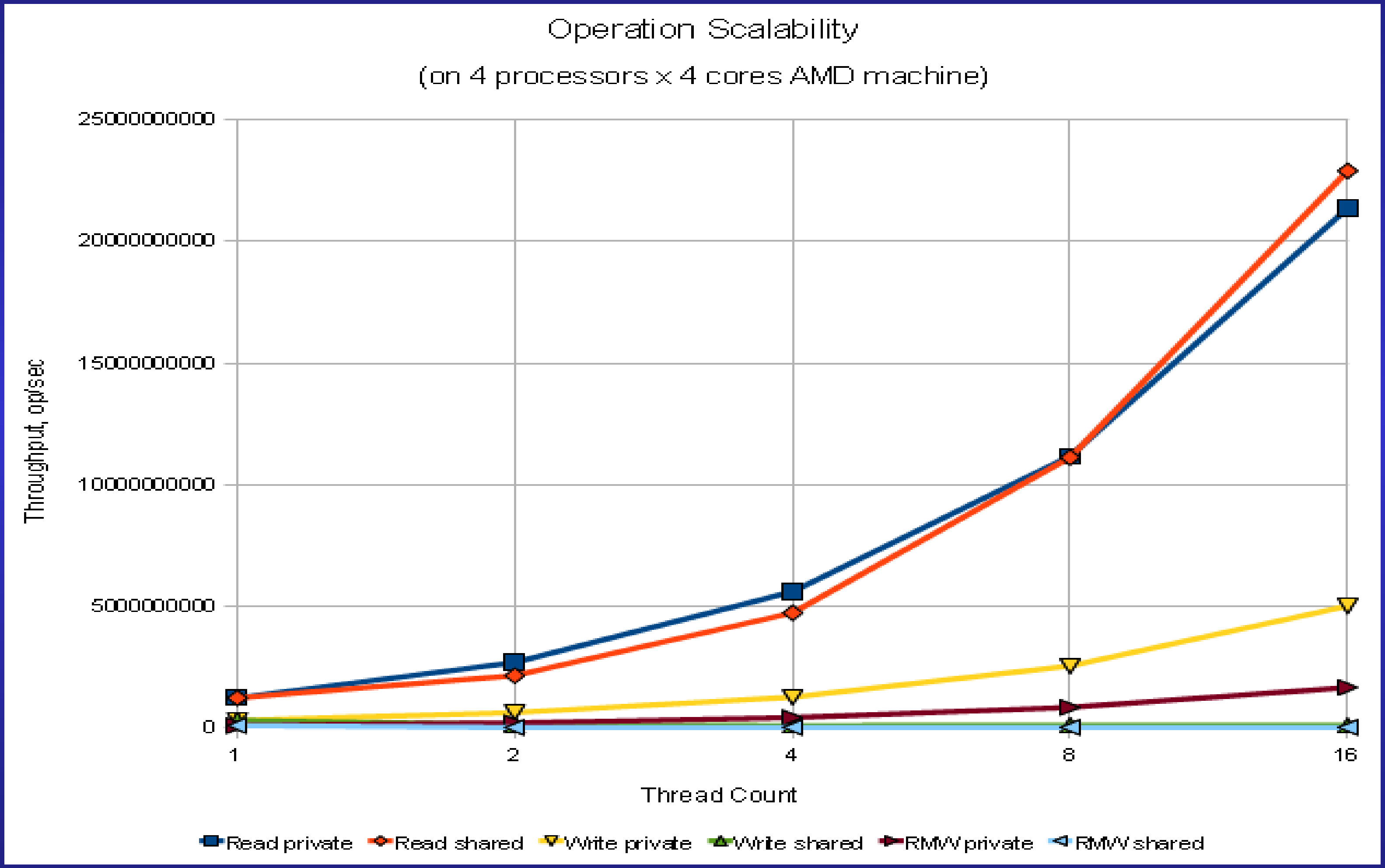
allow for scaling ***down*** to a single thread
for test before scaling ***up*** for production

Avoid sharing mutable data

- Shared mutable data is the evil of all computing!
- Read-only data can be shared safely without locks
- Const is your friend
- Pure message-passing approach avoids this



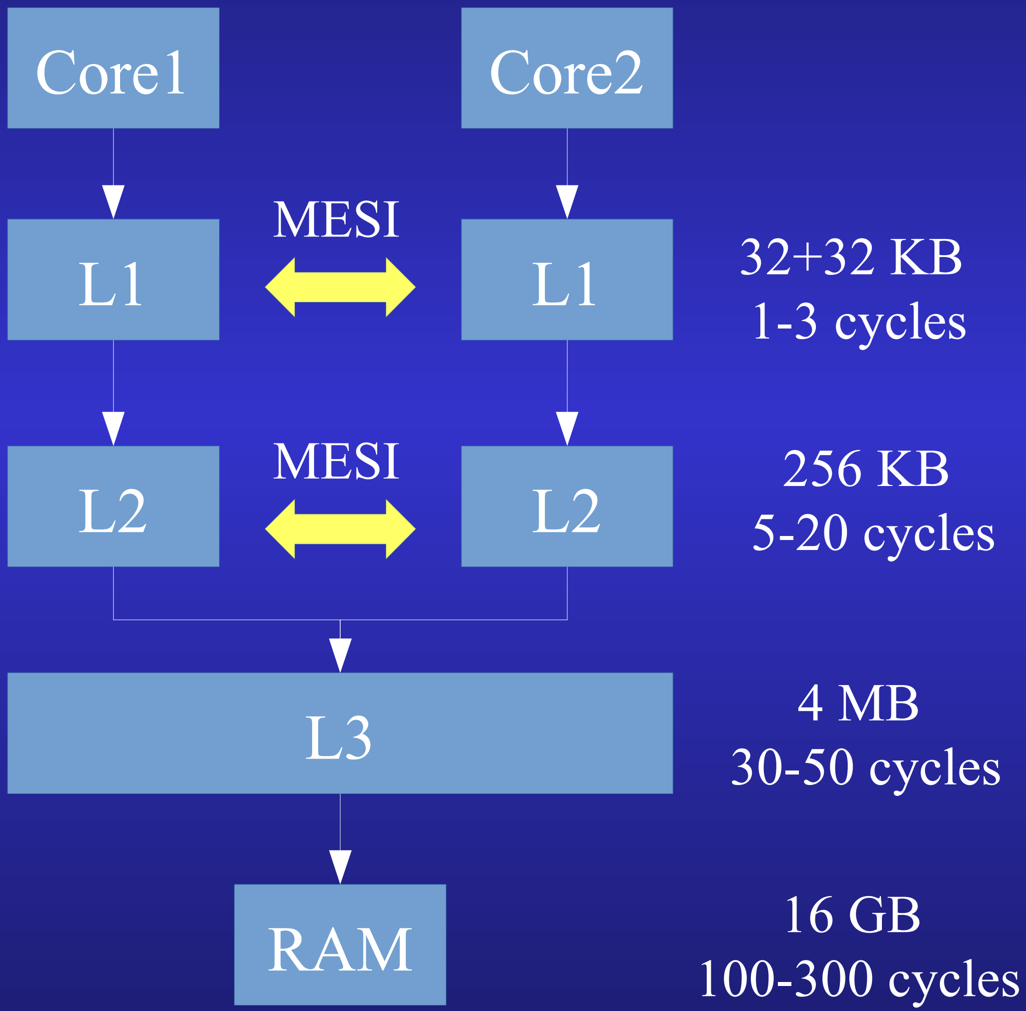
Shared writes don't scale



(graphic by Dmitry Vykov, http://www.1024cores.net, CC BY-NC-SA 3.0)

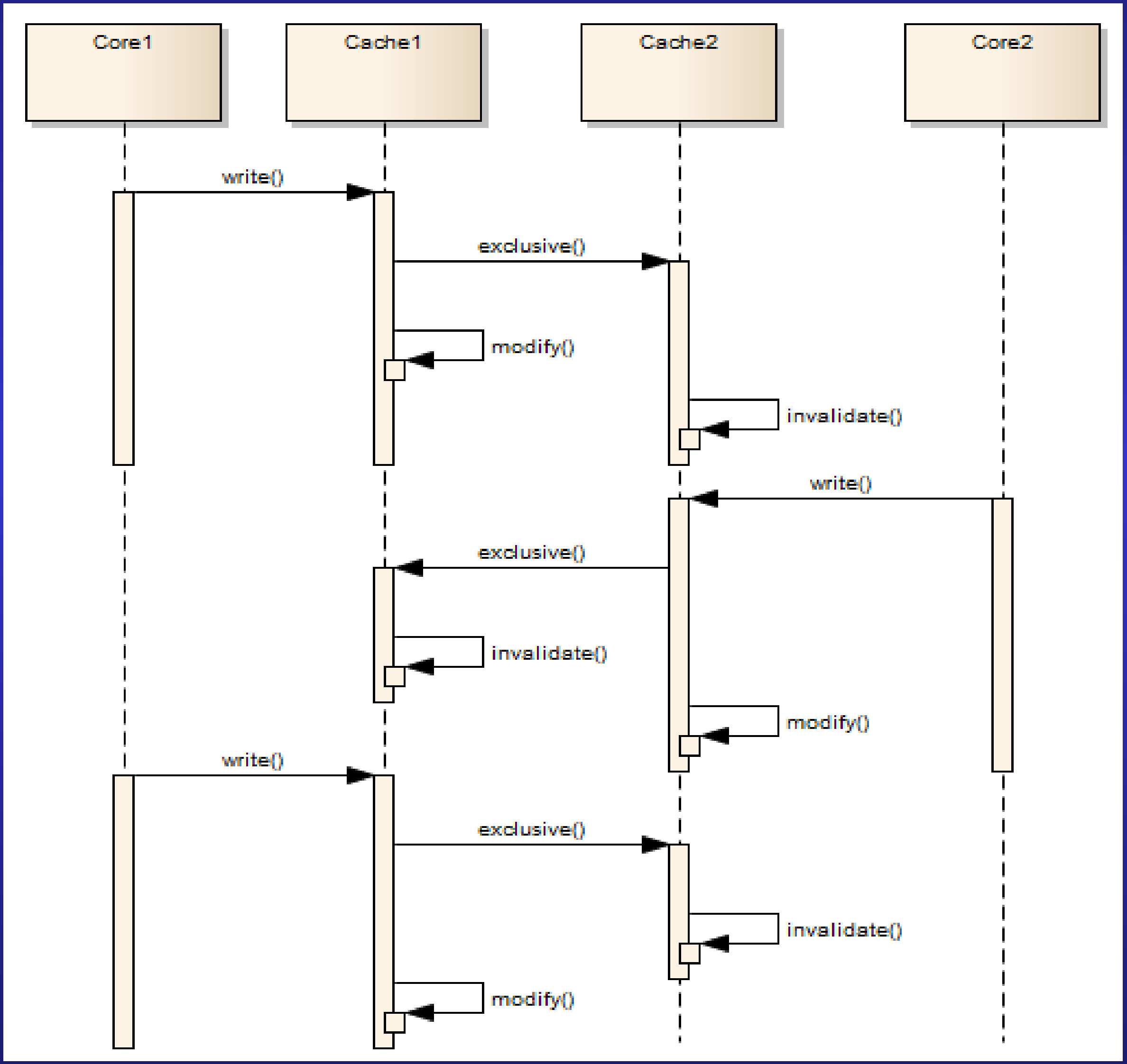
single writer principle
for speed/scale

Why shared writes don't scale



- Caches have to communicate to ensure coherent view
- MESI protocol passes messages between caches
- Shared writes limited by MESI comms

Shared writes – cache ping-pong



- Cache line passed between caches
- Hardware serialises writes to the same line
- Therefore zero scalability!!!
- For speed, don't pass ownership: *Single Writer Principle*

Example – contention costs

```
std::atomic<int> counter(0);

void count()
{
    for (auto i = 0; i != numLoops; ++i)
        counter++;
}

// run with 4 threads on a 4-core machine
// -O3 -march=native
```

3.5 times faster
13.6 times less CPU
when run on one core

```
$ time ./a.out
real    0m1.675s
user    0m6.384s
sys     0m0.007s
```

```
$ time taskset -c 1 ./a.out
real    0m0.476s
user    0m0.470s
sys     0m0.006s
```

Example – contention costs (cont'd)

```
$ perf stat -D 100 -e cycles,instructions,stalled-cycles-backend,cs ./a.out

 Performance counter stats for './a.out':

    16,635,079,957      cycles
       247,668,647      instructions              #    0.01  insns per cycle
                                                  #   65.91  stalled cycles per insn
    16,324,609,637      stalled-cycles-backend    #   98.13% backend  cycles idle
             1,912      cs

       1.569828247 seconds time elapsed
```

```
$ perf stat -D 100 -e cycles,instructions,stalled-cycles-backend,cs taskset -c 1 ./a.out

 Performance counter stats for 'taskset -c 1 ./a.out':

     1,135,801,575      cycles
       197,626,046      instructions              #    0.17  insns per cycle
                                                  #    5.19  stalled cycles per insn
     1,026,469,027      stalled-cycles-backend    #   90.37% backend  cycles idle
               115      cs

       0.480011707 seconds time elapsed
```

Don't undersynchronise

- Shared variables need to be synchronised correctly
  - Do not rely on guesswork
  - Do not try and cheat
  - Do not rely on unspecified ordering or visibility
- Undersychronised variables are subject to data races (at least one reader and one writer)
- Causes transient and unreproducible errors

use locks on shared mutable data structures
or use single atomics as synchronisation points

Don't oversynchronise

- Shared variables need to be synchronised correctly
  - Too much locking will make the code serialised
  - Locks are there to slow your program down until it is (hopefully) correct
  - Watch out for deadlock and livelock
  - Performance reduces back to slower than a single thread in the worst case because of locking overhead (locks are shared writes)
  - Amdahl's Law kicks in

don't keep adding locks – have a clear plan

Amdahl's Law

Serial portion of code	Maximum speedup
1%	100x
5%	20x
10%	10x
20%	5x
25%	4x

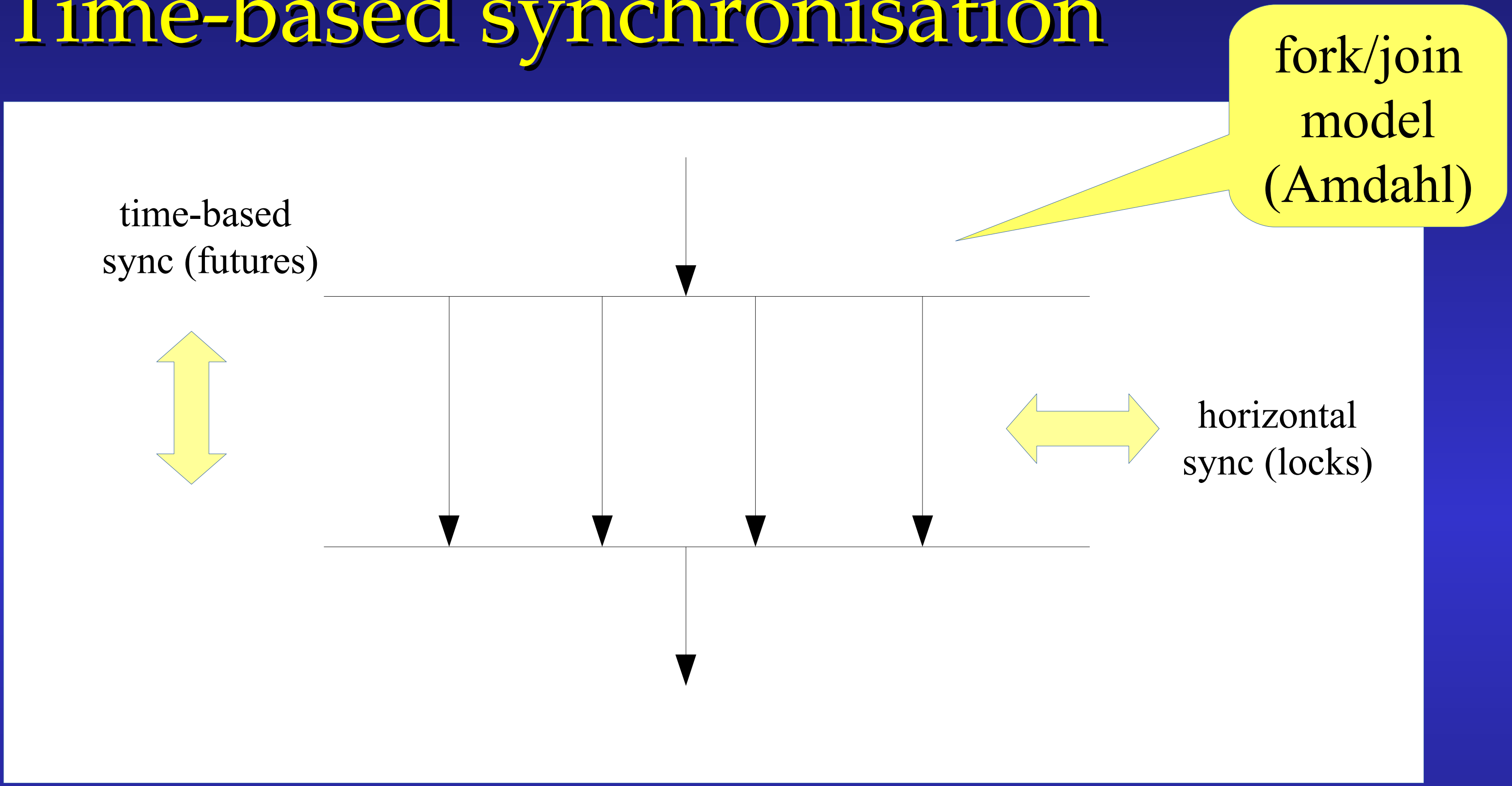
- Serial code limits scale, regardless of the number of threads or cores available

avoid non-read-only data sharing to allow for maximum parallelism

Deadlock and livelock

- If you have more than one lock in your program you may end up in a deadlock (deadly embrace)
  - Have only one lock (may limit performance)
  - Increases lock hold time
- Locking order is important
  - C++11's std::lock
  - Use addresses of locks to guarantee ordering
  - Release order is not important
  - May require exposing internal locks to callers

Time-based synchronisation



locks don't sequence start and finish
use futures to synchronise in time (A comes after B)

Hardware v. software threads

- There are a limited number of hardware threads available
  - 1 per core
  - 2 per core for Intel hyperthreading
- If there are more s/w than h/w threads then they will have to take turns (oversubscription)
  - Leads to context switching
  - Slow; 1000s of cycles to switch
    - Call to operating system
    - Scheduling
    - Cold cache and TLB

one software thread per hardware thread

Queue-based systems

- Systems that are based on queues can have performance issues caused by:
  - Context switching when queues are empty or full
  - Voluntary context switching
  - Shared writes to queue (insert and remove)
  - Processing per item is too small
  - Can be difficult to run in single-threaded mode
  - c.f. Disruptor pattern

be careful with queues if performance is important

Lock hold time and scope

- The time that a lock is held for determines the amount of parallelism
  - Shorter hold times are better
  - Shorter times may also indicate less shared state
- However, small lock scopes may not protect the data across lock scopes adequately
  - Need to consider business-level transactions and logical unit of works
  - Can lead to application-level errors because of concurrently changing data

TOCTOU and application errors

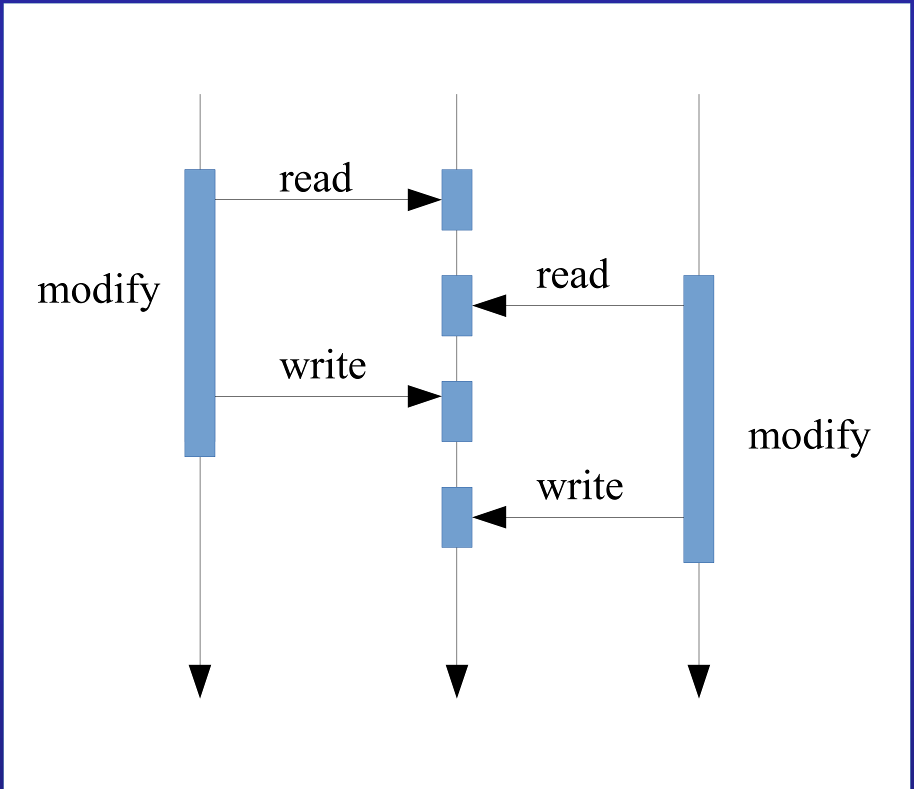
- Time-of-check to time-of-use errors (TOCTOU) can lead to application errors
  - **<u>Note:</u>** these are not data races caused by synchronisation errors (i.e. locking errors)
  - These are caused by concurrent modifications at the application level
  - Usually caused by inappropriate APIs

<u>Sample conversation</u>

Me: Is there any ice cream left, please?
Waiter: I'll check... yes there is
Me: I'll have some please
Waiter: Oops, we've just run out

TOCTOU and application errors (2)

- Locking doesn't help
- Need a different API
  - e.g. putIfAbsent()
  - popIfNotEmpty()
- Single batched operation with expection and failure notification
- Compare-and-set (CAS) is a classic approach (retries)



Volatile

- Volatile in C and C++ is of no use for multithreading
  - In Java and C# it means "atomic"
- It disables caching in registers
- It forces memory accesses
- It doesn't ensure cross-thread visibilty
- It doesn't affect compiler or hardware reordering of operations

> do not use volatile variables except for memory-mapped device I/O

Spin loops and polling

- Spin loops are disastrous on single-threaded machines
  - They just burn CPU cycles until the OS reschedules the thread
- On a multi-thread machine they are of use when they use less CPU than the overhead of a context switch (1000s of cycles)
- Polling with a sleep() uses little CPU but has longer wakeup latency (avg 1/2 polling time)

avoid spin loops unless you have measured the latency-CPU tradeoff

Blocking I/O

- Programs can be CPU, memory, network or I/O limited
- I/O limited programs that use blocking I/O will often use too many threads to handle blocking calls for I/O
- This causes lots of context switching
- Investigate asynchronous approaches
  - libaio, non-blocking sockets, overlapped I/O
- Damage-limitation approaches such as I/O thread pools
- Watch out for copying of data (zero copy)

Interrupting threads and shutdown

- Don't even think about trying to interrupt another thread
- Plan a clean shutdown mechanism for your program
- Often will involve a cooperative approach
  - Shared stop/start/state flag
  - Shutdown message in message-passing applications

Thread priorities and scheduling

- If your application requires the use of thread priorities to operate correctly then it's almost certainly broken
- Beware of priority inversion and locking issues
- Thread scheduling is rarely the correct solution
  - Probably implies locking issues and too much contention; fix that first
  - Can be useful in limited circumstances to provide run-to-completion semantics

Deletion

- Be careful about deleting data in concurrent systems; another thread may still have a reference
- Reference counting can help but counters must be thread-safe (std::shared_ptr is, mostly...)
- Avoid concurrent deletions: tbb::concurrent_vector is append-only
- Separate the program into phases so that deletion is in a safe serial part
- Garbage collection is a big win here

Multiple atomics

- Can be used successfully individually
- The problem becomes more about transactional correctness
  - Do the atomics make sense together?
  - Race window between modifying both
  - Initialisation order
  - Visibility of updates

1) use atomic<Data *> instead of multiple atomics
(even better, use atomic<const Data *>)

2) use std::call_once for initialisation

Immutable data and safe publishing

- Immutable data can be shared without locking
- Fewer errors and easy to understand
- Be careful about deletion; are there still references to the object?!
- std::shared_ptr<> has atomic counters but not its body so it must be locked
- Helps with exception safety, transactions and copy-on-write optimisation

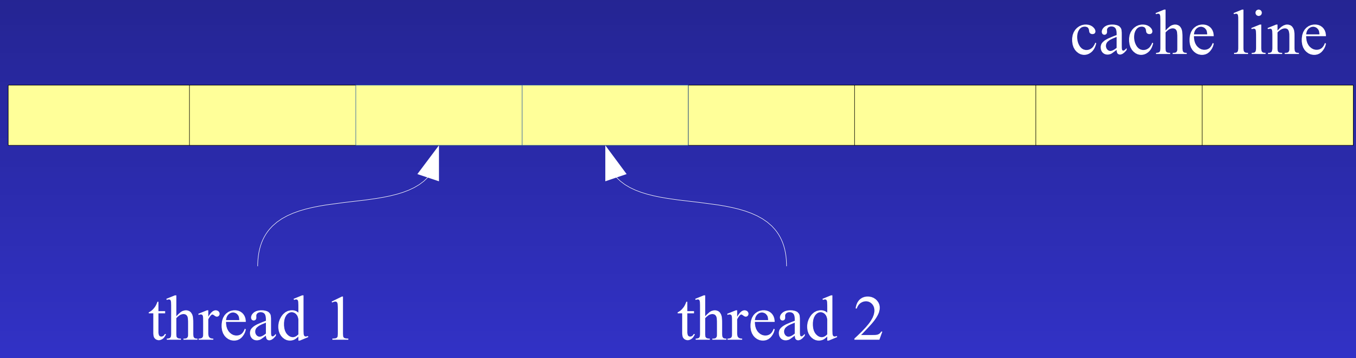
publish safely using std::atomic<const Data *>

Error handling

- Propagating errors from one thread to another is tricky
- std::future<> catches exceptions thrown in the called thread and rethrows them in the calling thread when f.get() is called
- Make sure you have a try/catch at the top-level of every thread you start

use std::future<> for time sequencing and easier error handling

False sharing



- Separate threads can access separate variables on the same cache line (often 64 bytes long)
- Writes by one thread invalidate the cache line for the other thread, leading to "cache ping-pong"
- Major performance killer – effectively shared writes

> watch out for false sharing
> use padding to length of cache line

Parallel algorithms

- Some libraries can run in parallel mode without you having to start any threads or do any synchronisation
- Gcc does this for STL algorithms if compiled with -D_GLIBCXX_PARALLEL and -fopenmp

use “free” parallelism if available

Read/write ratio

- Different approaches are appropriate for read-mostly or write-mostly access patterns
- Also depends on lock hold time
- Short hold, lots of writes => CAS, spin lock et al
- Short hold, mixed R/W => distributed mutex
- Short (zero) hold => RCU-style lock free
- Beware of reader/writer lock scaling

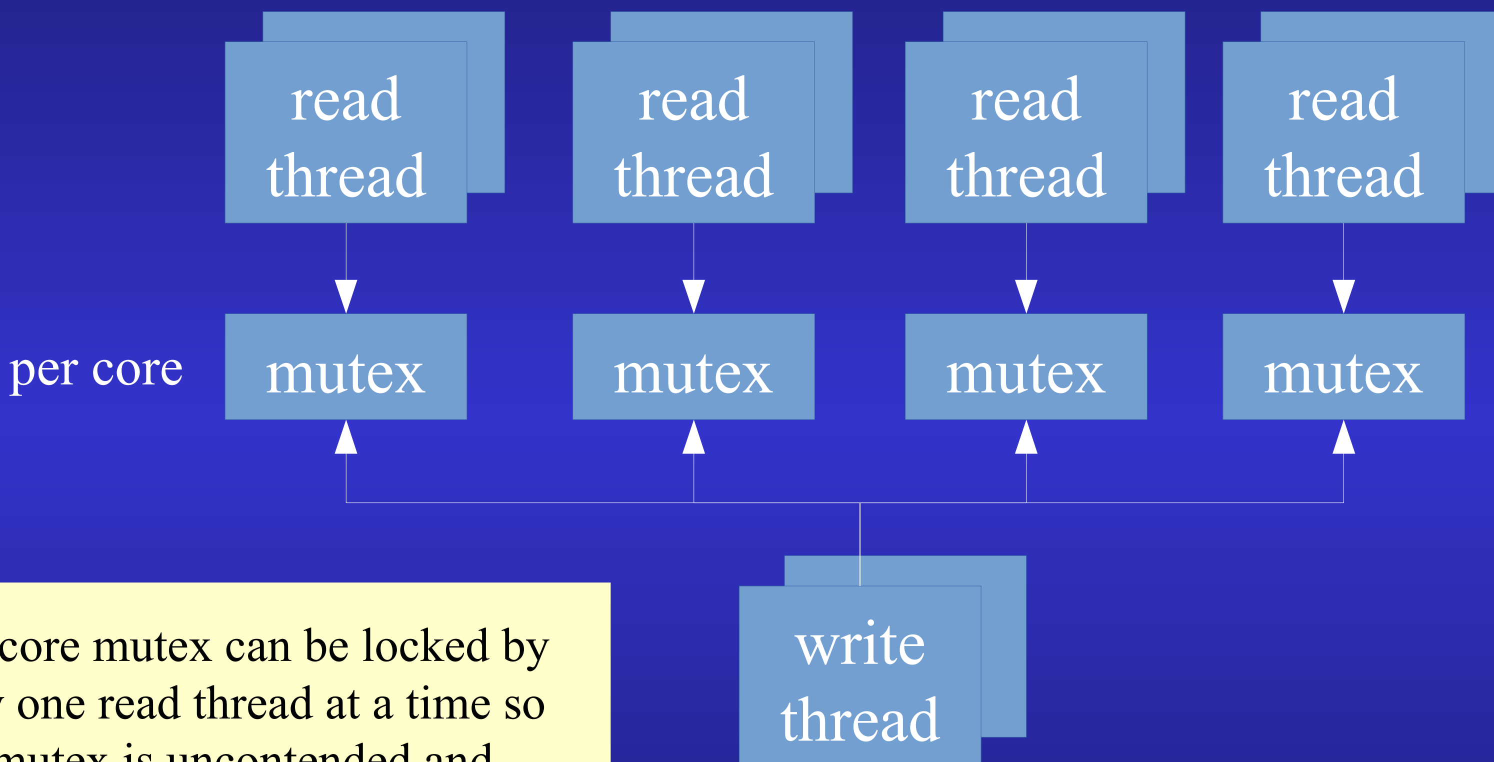
select an approach based on
data-access patterns

Fast/slow paths

- Know what operations need to be fast
  - Frequent operations
- Avoid locks on the fast path
  - Mutexes, I/O, memory allocation, etc
- Push work to the slow path
  - Maybe use a queue or a background thread
  - Block slow path until there are no fast path users
  - RCU, garbage collection, distributed read/write mutex

know what needs to be fast

Example – distributed R/W mutex



- Per-core mutex can be locked by only one read thread at a time so the mutex is uncontended and therefore fast; read mutex cache line is not shared across cores
- Write thread locks all mutexes to block readers; slow operation

- Windows: GetCurrentProcessorNumber()
- Linux: sched_getcpu()
- See http://1024cores.net for more details

Distributed R/W mutex read performance

```
struct alignas(64) PerCoreLock {
    std::mutex lock;
};

PerCoreLock locks[numThreads];

void lockLoop()
{
    for (auto i = 0; i != numLoops; ++i) {
        auto core = sched_getcpu();
        std::lock_guard<std::mutex> guard(locks[core].lock);
    }
}
```

code as shown

```
$ time ./a.out
real    0m0.537s
user    0m2.019s
sys     0m0.003s

$ time taskset -c 1 ./a.out
real    0m1.992s
user    0m1.985s
sys     0m0.005s
```

no alignas(64) – false sharing

```
$ time ./a.out
real    0m4.204s
user    0m10.514s
sys     0m0.005s

$ time taskset -c 1 ./a.out
real    0m2.091s
user    0m2.077s
sys     0m0.004s
```

one shared lock – context switching

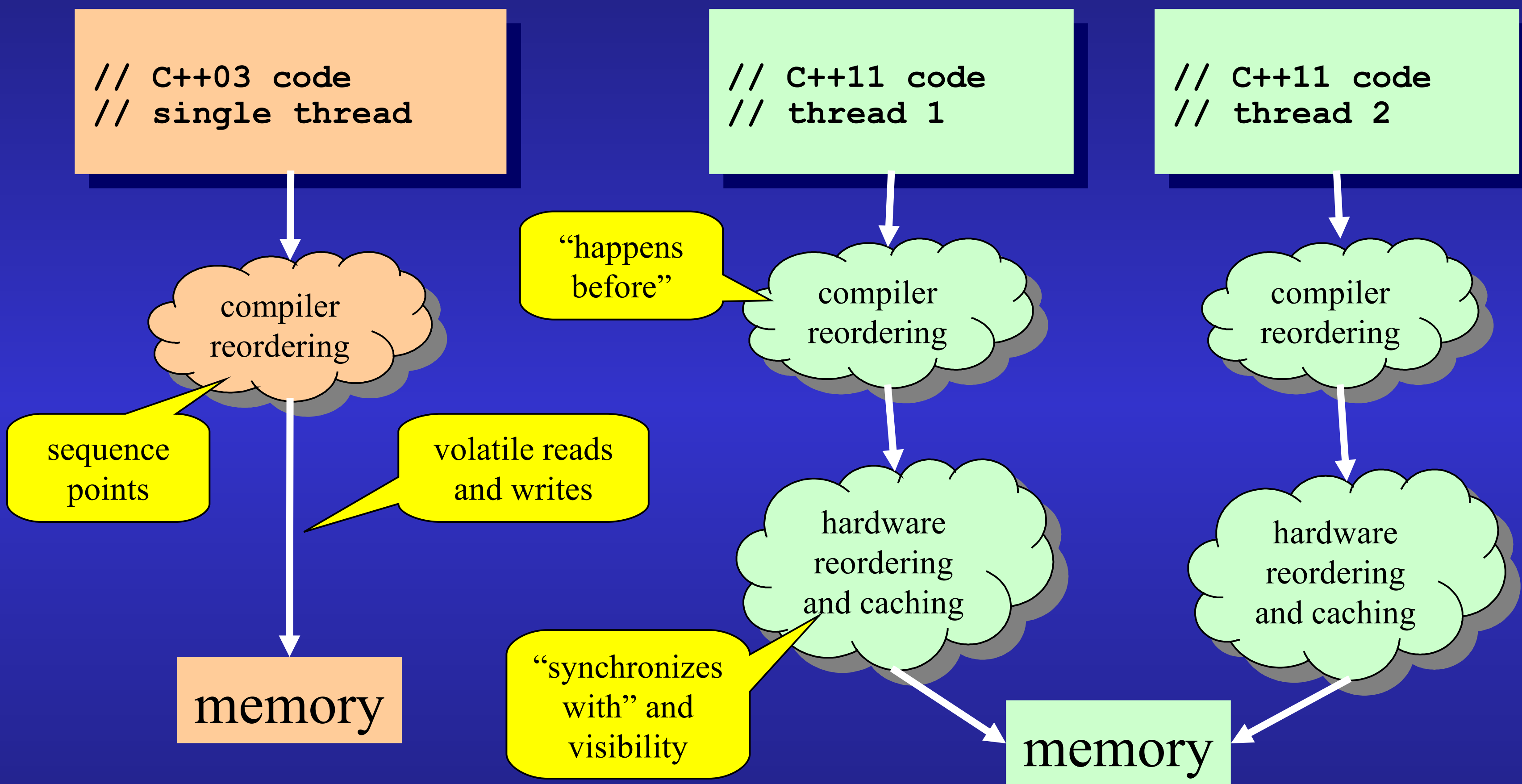
```
$ time ./a.out
real    0m10.097s
user    0m11.862s
sys     0m24.078s

$ time taskset -c 1 ./a.out
real    0m2.057s
user    0m2.045s
sys     0m0.003s
```

Memory model and ordering

- We want our programs to run quickly
- Modern hardware reorders instructions and can run multiple instruction at once
- Compilers can reorder instructions too (e.g. to cover possible cache misses, delay slots, etc)
- Some languages (Java, C++11) have defined a memory model to say what reordering means at the language level
- Don't go there unless you can prove through measurement it's necessary

Need for a memory model



- Correctness now based on memory, not just code
- Need to control caching (register, L1, L2, etc)

Instruction interleaving

```
// thread 1
x = 1;   // 1
r1 = y;  // 2
```

```
// thread 2
y = 1;   // 3
r2 = x;  // 4
```

interleave

```
// 6 possible sequentially consistent
// execution orders

1234// x=1; r1=y; y=1; r2=x;
1324// x=1; y=1; r1=y; r2=x;
1342// x=1; y=1; r2=x; r1=y;
3412// y=1; r2=x; x=1; r1=y;
3142// y=1; x=1; r2=x; r1=y;
3124// y=1; x=1; r1=y; r2=x;
```

- "Sequential consistency" means operations in separate threads are interleaved and that all threads see the same interleaving
  - Sequence is preserved within and across threads
- This is the "natural" mental model for programmers to think of thread execution order and memory
  - It is also the C++11 default memory ordering

Instruction reordering

```
// thread 1
x = 1;   // 1
r1 = y;  // 2
```

```
// thread 2
y = 1;   // 3
r2 = x;  // 4
```

reorder

```
// 4 factorial (== 24)
// possible execution orders

1234 // x=1; r1=y; y=1; r2=x;
4321 // r2=x; y=1; r1=y; x=1;
3124 // y=1; x=1; r1=y; r2=x;
// etc...
```

could be executed in reverse order

- In order to gain performance both the compiler and the hardware may reorder instructions
  - compiler may move loads earlier (to allow for cache misses)
  - hardware may not write back to memory immediately (store buffers)
- x, y, r1 and r2 are all independent so code can be reordered
- Even worse, changes in one thread may not be visible in another thread so results are not defined – ***data race***

Hardware memory reordering

	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

http://en.wikipedia.org/wiki/Memory_ordering

- Hardware can reorder memory operations in different ways
  - Can also depend on operating system
    - Solaris on SPARC uses Total Store Order (TSO)
    - Linux on SPARC uses Relaxed Memory Order (RMO)

Synchronisation with seq. consistency

```
// thread 1
x = 42;
x_init = true;
```

```
// thread 2
while (! x_init) {}
y = x;
```

- This code is correct when sequentially consistent
  - thread 2 doesn't access x until it has been set by thread 1
- But in the presence of reordering it can fail
- The problem is that we haven't specified that cross-thread order or visibility is important
- We need to use synchronisation variables – atomics
- Making everything atomic is slow – 30-60 cycles
  - cache synchronisation is slow and has limited bandwidth

Synchronisation with atomics

```
// thread 1
std::atomic<bool> x_init;
int x;

x = 42;
x_init.store(true);

// or x_init = true;
```

```
// thread 2
extern std::atomic<bool> x_init;
extern int x;

while (! x_init.load());
y = x;

// or while (! x_init);
```

- This now works without relying on having sequential consistency everywhere (just atomics)
  - atomics prevent the compiler moving code across accesses
  - atomics also cause memory updates to be visible
- Load and store uses sequential consistency
  - uses default parameter of std::memory_order_seq_cst

Low-level synchronisation detail

```
// thread 1
std::atomic<bool> x_init;

x = 42;
// blue fence
x_init.store(true);
// red fence
```

prevents x and x_init being reordered

makes store to x_init visible

```
// thread 2
extern std::atomic<bool> x_init;

while (/*rfence*/! x_init.load());
// blue fence
y = x;
```

prevents reordering

rfence here in loop forces load of latest value of x_init

- Blue fences prevent the compiler reordering code
  - they don't generate any run-time code
- Red fences force memory to make changes visible
  - they do generate code: fence, lock prefix, CAS opcodes
  - depends heavily on underlying hardware (c.f. reordering)
  - only need one of the two red fences, usually on store
- One reason that threads can't just be a library

Generated assembler code

sequential consistent by default

```
// thread 1
x = 42;
// blue fence
x_init.store(true);
// red fence
```

```
// thread 2

while (/*rfence*/ ! x_init.load());
// blue fence
y = x;
```

```
// with atomic bool x_init
 mov      DWORD PTR x, 42
 mov      BYTE PTR x_init, 1
 mfence

// with bool x_init
 mov      DWORD PTR x,
 mov      BYTE PTR x_init, 1
```

fence needed on store for X86

```
// with atomic bool x_init
L25:
 movzx    eax, BYTE PTR x_init
 test     al, al
 je       .L25
 mov      eax, DWORD PTR x
 mov      DWORD PTR y, eax

// with bool x_init
 cmp      BYTE PTR x_init, 0
 jne      .L3
.L5:
 jmp      .L5
.L3:
 mov      eax, DWORD PTR x
 mov      DWORD PTR y, eax
```

no fence needed on load for X86

infinite loop because visibility not specified

g++ 4.7 output, x86

Using memory order flags

acquire means no reads in this thread reordered before here

```
// thread 1
x = 42;
// blue fence
x_init.store(true,
std::memory_order_release);
```

```
// thread 2
while (!
x_init.load(std::memory_order_acquire));
// blue fence
y = x;
```

```
// with bool x_init seq_cst
 mov      DWORD PTR x, 42
 mov      BYTE PTR x_init, 1
 mfence

// with bool x_init release
 mov      DWORD PTR x, 42
 mov      BYTE PTR x_init, 1
```

needed on seq_cst store

no fence for release store

```
// with bool atomic x_init
L25:
 movzx    eax, BYTE PTR x_init
 test     al, al
 je       .L25
 mov      eax, DWORD PTR x
 mov      DWORD PTR y, eax

// same code for x_init acquire
```

no fence needed on load for X86

- Release provides only the blue fence (no writes in this thread reordered after the store)
- Controls compiler reordering but not hardware

Memory model advice

- This is a complex and subtle area and you should avoid using it unless you can prove that you can’t get adequate performance without it
  - yes, really, I mean it....
- Even experts get confused by this stuff!
  - did I mention you should avoid it....
- If you do use it, use acquire on load and release on store
  - Anything else will be a source of subtle bugs

Memory model - example

```
std::atomic<int> counter(0);

void count()
{
    for (auto i = 0; i != numLoops; ++i)
      counter.store(5);
      //counter.store(5, std::memory_order_seq_cst);
      //counter.store(5, std::memory_order_release);
}
```

code as shown

```
$ time ./a.out
real    0m2.039s
user    0m5.932s
sys     0m0.002s

$ time taskset -c 1 ./a.out
real    0m1.004s
user    0m1.002s
sys     0m0.001s
```

m_o_seq_cst (default)

```
$ time ./a.out
real    0m2.118s
user    0m6.496s
sys     0m0.009s

$ time taskset -c 1 ./a.out
real    0m0.999s
user    0m0.994s
sys     0m0.004s
```

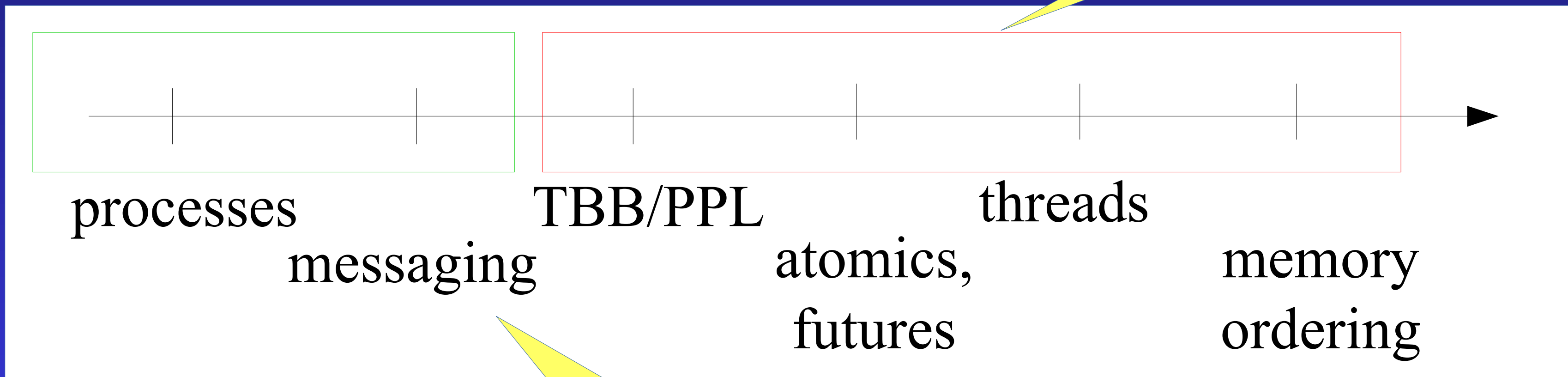
m_o_release (no mfence)

```
$ time ./a.out
real    0m0.097s
user    0m0.225s
sys     0m0.002s

$ time taskset -c 1 ./a.out
real    0m0.057s
user    0m0.055s
sys     0m0.002s
```

Concurrency spectrum

shared memory



keep as far to the left as possible

- Invest your time in splitting up the problem
- You know your domain
- Leave concurrency parts to others

Summary

- Multithreaded programming is tricky
  - New skills and ideas and ways to get it wrong
- Focus on partitioning the problem
  - Determines data sharing, locking, work breakdown and scheduling
- Avoid shared mutable data where possible
- Know your access patterns
- Scale down as well as up
- Balance extremes of grain size, lock extent, etc
- Don't try and be clever