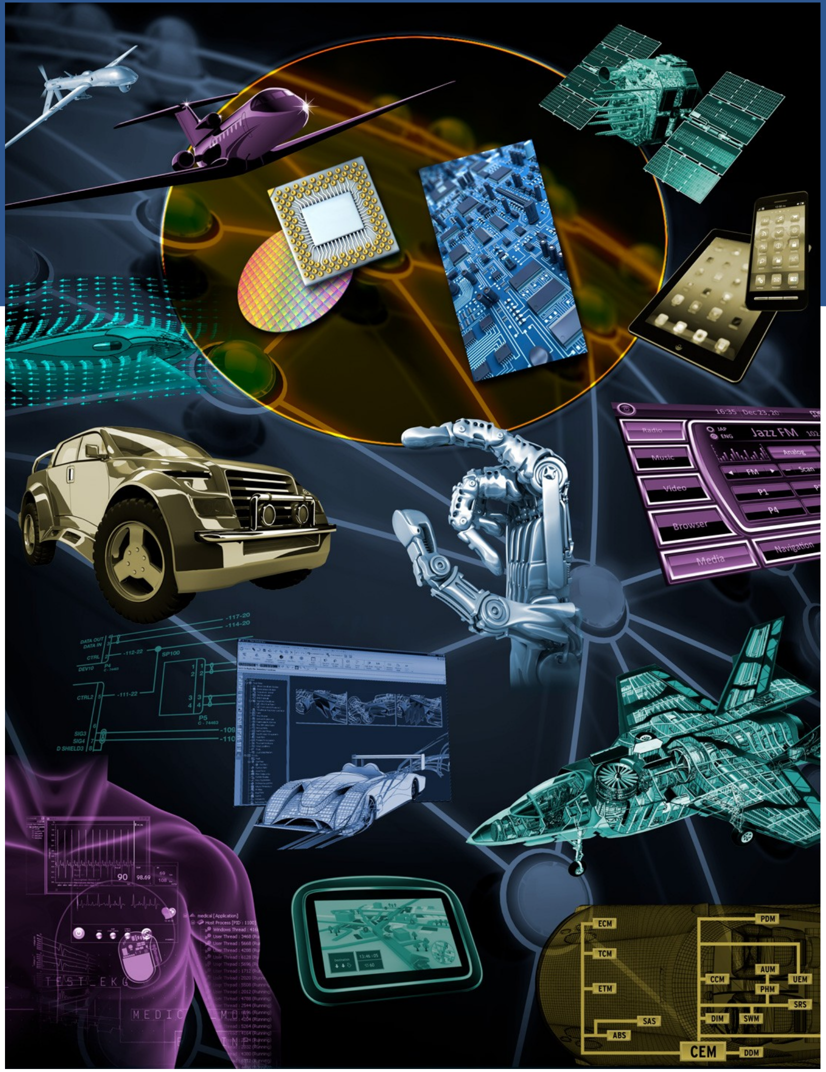


C++ atomics: from basic to advanced. What do they do?

really

Fedor G Pikus
Chief Scientist
Design2Silicon Division

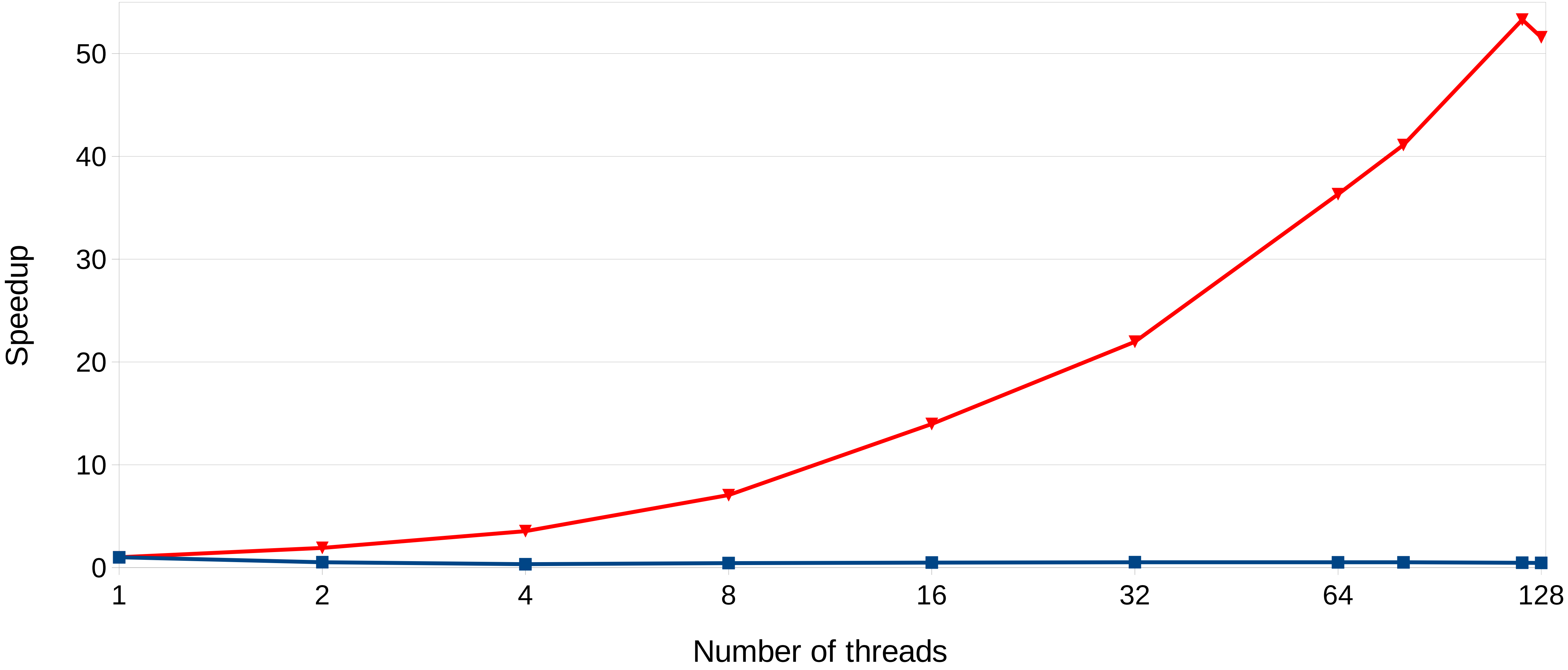
Semptember 25, 2017

Mentor®
A Siemens Business

Atomics: the tool of lock-free programming
Lock-free means “fast”

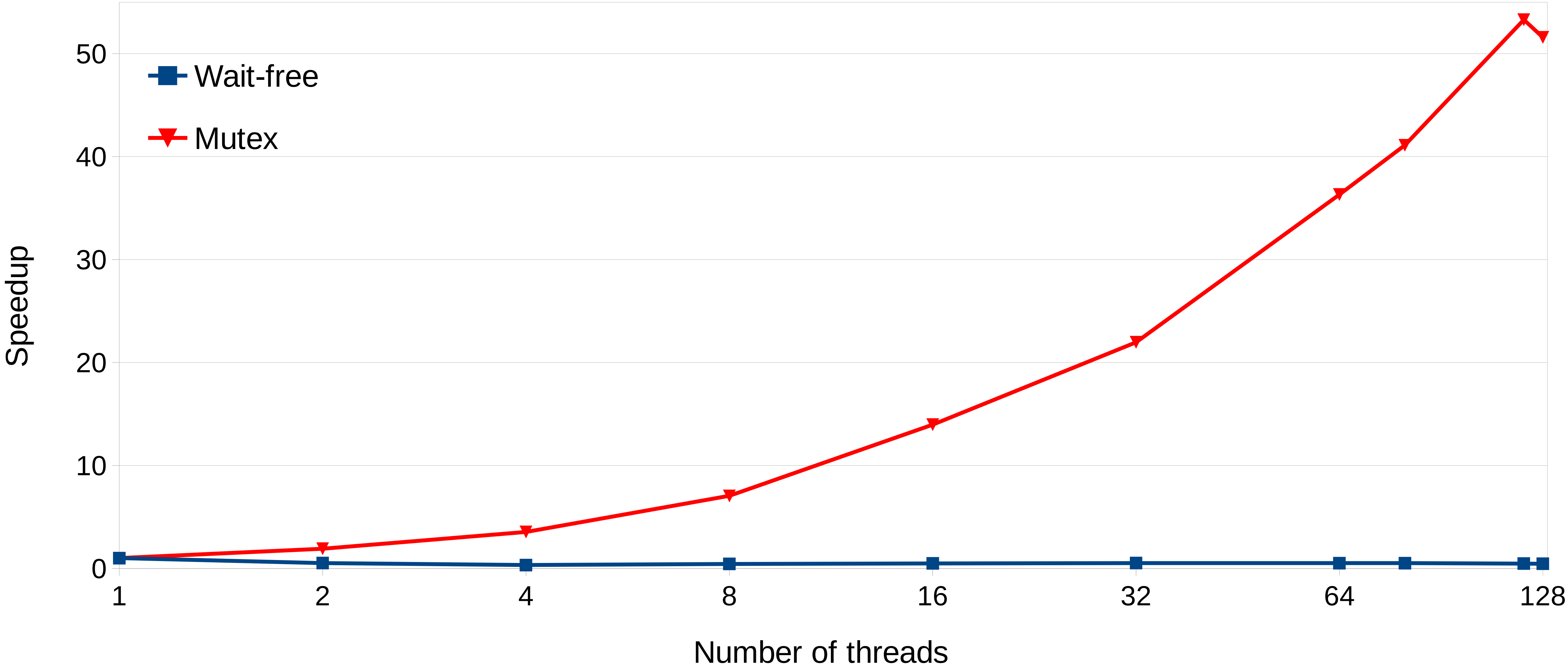
- Compare performance of two programs
- Both programs perform the same computations and get the same results
- Both programs are correct
  - No “wait loops” or other tricks
- One program uses std::mutex, the other is wait-free (even better than lock-free!)

Lock-free means “fast”



Mentor
A Siemens Business

Lock-free means “fast”



Lock-free means "fast"

```
std::atomic<unsigned long> sum;
```

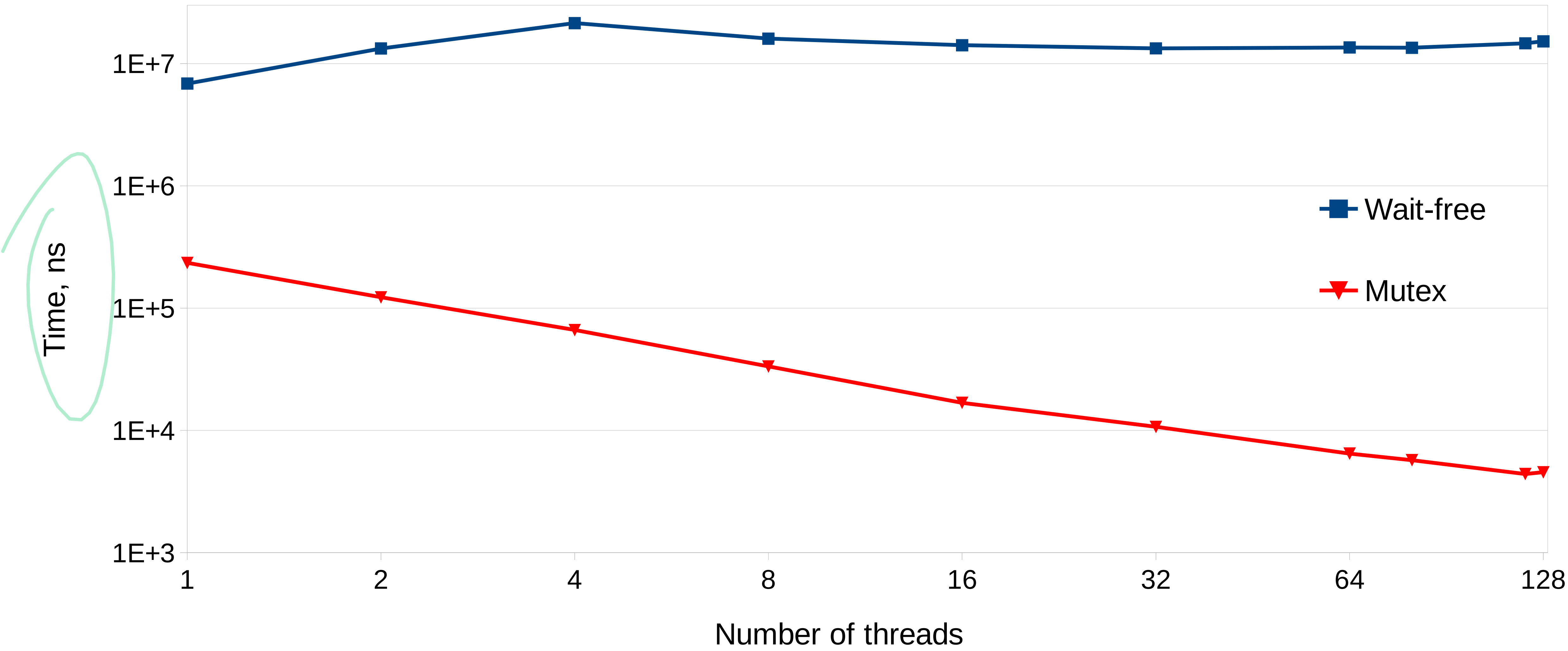
- Program A:

```
void do_work(size_t N, unsigned long* a) {
  for (size_t i = 0; i < N; ++i) sum += a[i];
}
```

- Program B:

```
unsigned long sum(0); std::mutex M;
void do_work(size_t N, unsigned long* a) {
  unsigned long s = 0;
  for (size_t i = 0; i < N; ++i) s += a[i];
  std::lock_guard<std::mutex> L(M); sum += s;
}
```

Is lock-free faster?



Mentor
A Siemens Business

Is lock-free faster?

- Algorithm rules supreme
- "Wait-free" has nothing to do with time
  - Wait-free refers to the number of compute "steps"
  - Steps do not have to be of the same duration
- Atomic operations do not guarantee good performance
- There is no substitute for understanding what you're doing
  - This class is the next best thing
- Let's now understand C++ atomics

What is an atomic operation?

- Atomic operation is an operation that is guaranteed to execute as a single transaction:
  - Other threads will see the state of the system before the operation started or after it finished, but cannot see any intermediate state
  - At the low level, atomic operations are special hardware instructions (hardware guarantees atomicity)
  - → from single instruction to whole program
  - This is a general concept, not limited to hardware instructions (example: database transactions)

Atomic operation example

int x = 0;

Thread 1	Thread 2
++x;	++x;

x = ?

- Increment is a “read-modify-write” operation:
  - read x from memory
  - add 1 to x
  - write new x to memory

Atomic operation example

```
int x = 0;
```

Thread 1

```
int tmp = x; // 0
++tmp; // 1
x = tmp; // 1
```

Thread 2

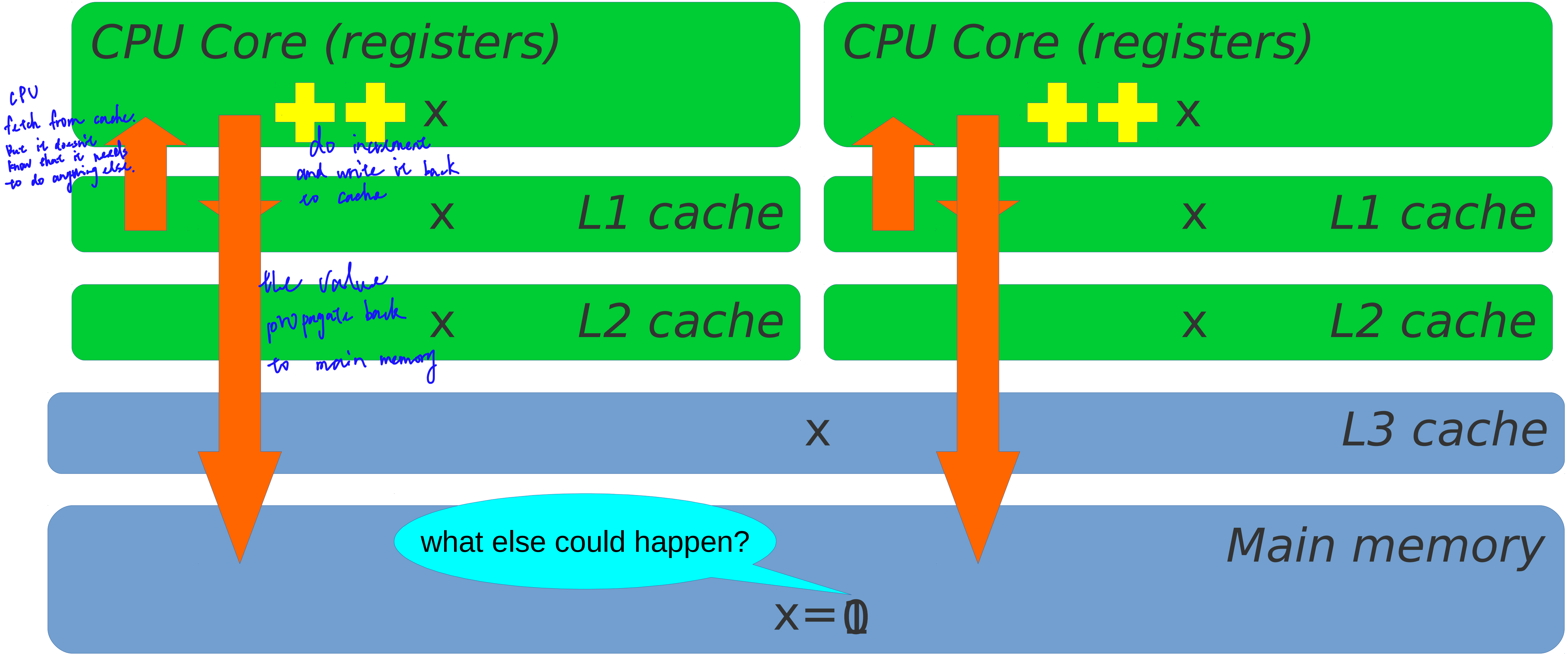
```
int tmp = x; // 0
++tmp; // 1

x = tmp; // 1!
```

```
x = 1
```

- Read-modify-write increment is non-atomic
- This is a data race (i.e. undefined behavior)

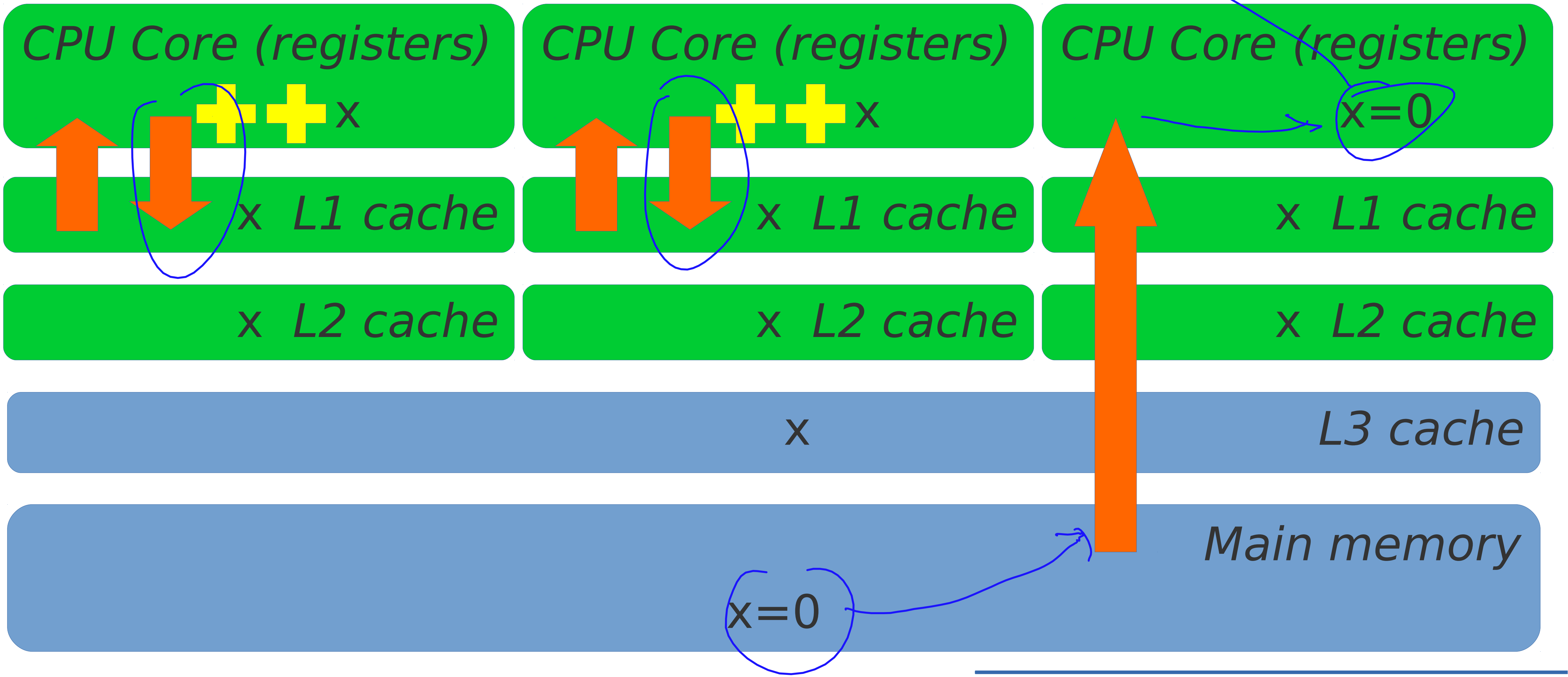
what else could happen?

What’s really going on?



What’s really going on?

If you have another CPU, you could see zero before value propagate to memory.



More insidious atomic operation example

```
int x = 0;
```

Thread 1

```
x = 42424242;
```

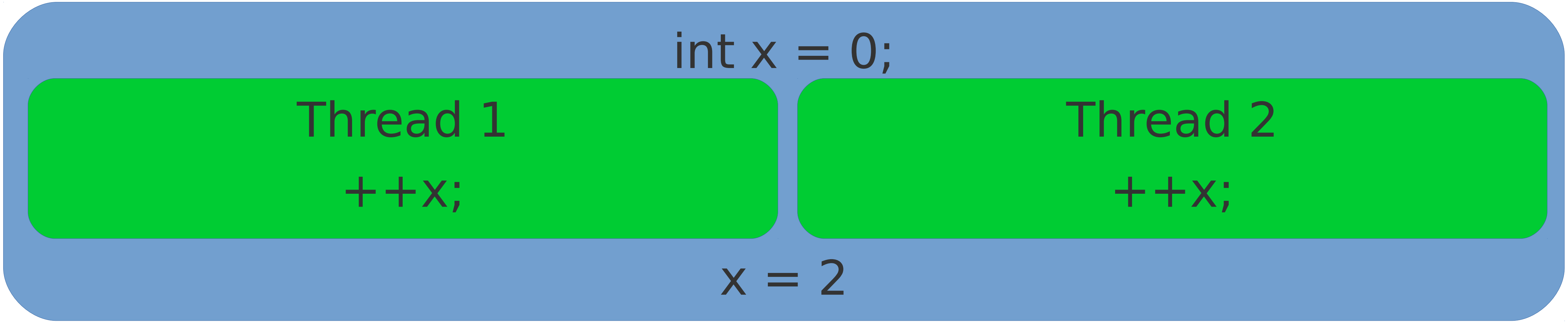
Thread 2

```
tmp = x;
tmp == ?
```

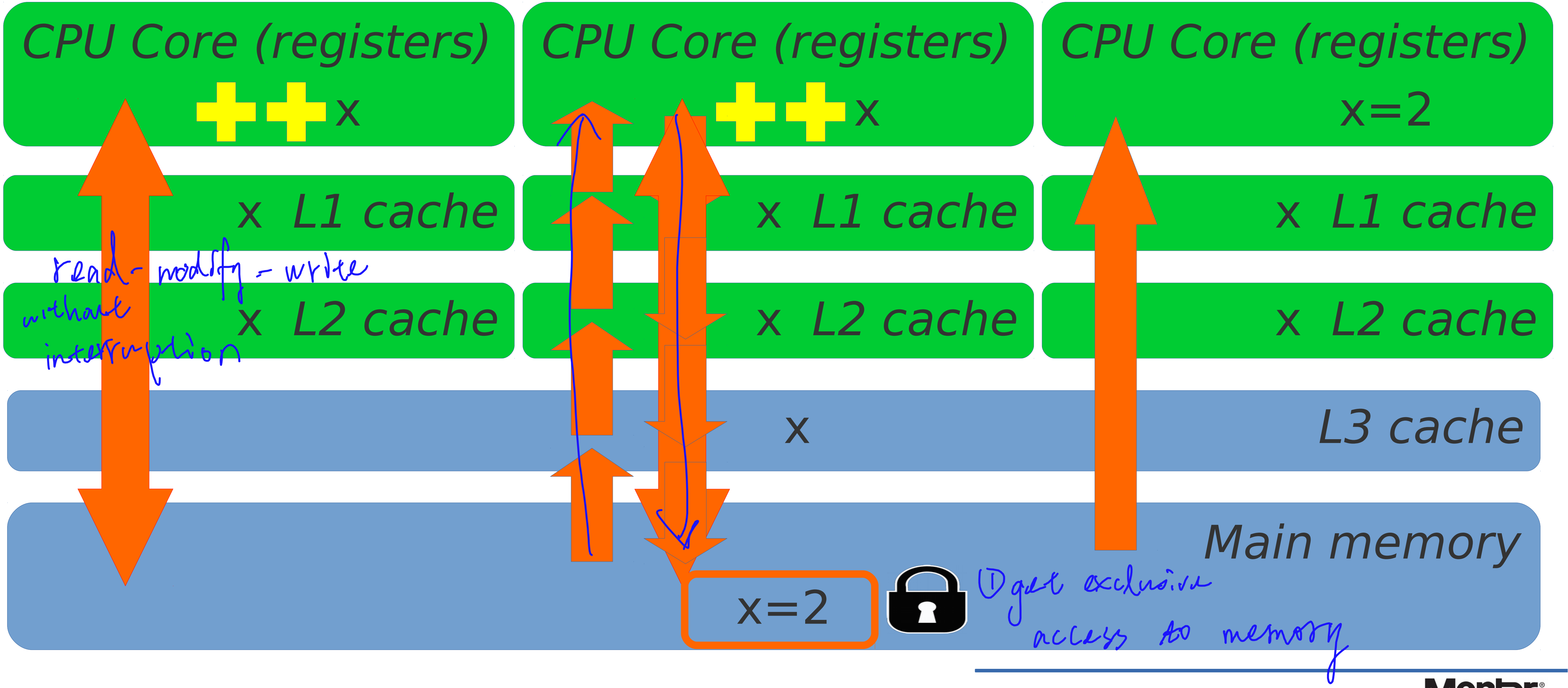
- Reads and writes do not have to be atomic!
  - On x86 they are for built-in types (int, long)
- How to access shared data from multiple threads in C++?

Data sharing in C++

- C++03: what's a thread?
- C++11: **std::atomic**
  #include <atomic>
  std::atomic<int> x(0); // NOT std::atomic<int> x=0;
- ++x is now atomic!



What's really going on now?



Mentor
A Siemens Business

std::atomic

- What C++ types can be made atomic?
- What operations can be done on these types?
- Are all operations on atomic types atomic?
- How fast are atomic operations?
  - Are atomic operations slower than non-atomic?
  - Are atomic operations faster than locks?
- Is “atomic” same as “lock-free”?
- If atomic operations avoid locks, there is no waiting, right?

What types can be made atomic?

- Any *trivially copyable* type can be made atomic
- What is trivially copyable?
  - Continuous chunk of memory
  - Copying the object means copying all bits (memcpy)
  - No virtual functions, noexcept constructor

```
std::atomic<int> i;            // OK
std::atomic<double> x;         // OK
struct S { long x; long y; };
std::atomic<S> s;              // OK!
```


Mentor
A Siemens Business

What operations can be done on std::atomic<T>?

- Assignment (read and write) – always
- Special atomic operations
- Other operations depend on the type T

OK, what operations can be done on std::atomic<int>?

- One of these is not the same as the others:

```
std::atomic<int> x{0};   // Not x=0!  x(0) is OK
++x;
x++;
x += 1;
x |= 2;
x *= 2;
int y = x * 2;
x = y + 1;
x = x + 1;
x = x * 2;
```

does not compile

OK, what operations can be done on std::atomic<int>?

- Two of these are not the same as the others:

```
std::atomic<int> x{0};
++x;
x++;
x += 1;
x |= 2;
~~x *= 2;~~
int y = x * 2;
x = y + 1;
x = x + 1;
x = x * 2;
```

not atomic

can compile

Are all operations on atomic types atomic?

- All operations ***<u>on the atomic variable</u>*** are atomic:

```
std::atomic<int> x{0};
++x;                // Atomic pre-increment
x++;                // Atomic post-increment
x += 1;             // Atomic increment
x |= 2;             // Atomic bit set
x *= 2;             // No atomic multiplication!
int y = x * 2;      // Atomic read of x
x = y + 1;          // Atomic write of x
x = x + 1;          // Atomic read followed by atomic write!
x = x * 2;          // Atomic read followed by atomic write!
```

std::atomic<T> and overloaded operators

- std::atomic<T> provides operator overloads only for atomic operations (incorrect code does not compile)
- Any expression with atomic variables will not be computed atomically (easy to make mistakes)
- ++x; is the same as x+=1; is the same as x=x+1;
  - Unless x is atomic!

What operations can be done on std::atomic<T> for other types?

- Assignment and copy (read and write) for all types
  - Built-in and user-defined
- Increment and decrement for raw pointers
- Addition, subtraction, and bitwise logic operations for integers (++, +=, –, -=, |=, &=, ^=)
- std::atomic<bool> is valid, no special operations
- std::atomic<double> is valid, no special operations
  - No atomic increment for floating-point numbers!

What “other operations” can be done on std::atomic<T>?

- Explicit reads and writes:

```
std::atomic<T> x;
T y = x.load();          // Same as T y = x;
x.store(y);              // Same as x = y;
```

- Atomic exchange:

```
T z = x.exchange(y);  // Atomically: z = x; x = y;
```

- Compare-and-swap (conditional exchange):

expected → desired

```
bool success = x.compare_exchange_strong(y, z);  T& y
    // If x==y, make x=z and return true
    // Otherwise, set y=x and return false
```

?

- Key to most lock-free algorithms

What is so special about CAS?

- Compare-and-swap (CAS) is used in most lock-free algorithms
- Example: atomic increment with CAS:

```
std::atomic<int> x{0};
int x0 = x;
while ( !x.compare_exchange_strong(x0, x0+1) ) {}
```

I "expect" no body change x before, then I x+1, return true. Otherwise, somebody change x before, return false. I don't need to read it again.

- For int, we have atomic increment, but CAS can be used to increment doubles, multiply integers, and many more

```
while ( !x.compare_exchange_strong(x0, x0*2) ) {}
```


Mentor® A Siemens Business

What “other operations” can be done on std::atomic<T>?

- For integer T:
  std::atomic<int> x;
  x.**fetch_add**(y); // Same as x += y;
  int z = x.**fetch_add**(y); // Same as z = (x += y) - y;
- Also **fetch_sub**(), **fetch_and**(), **fetch_or**(), **fetch_xor**()
  - Same as +=, -= etc operators
- More verbose but less error-prone than operators and expressions
  - Including load() and store() instead of operator=()

Mentor
A Siemens Business

std::atomic<T> and overloaded operators

- std::atomic<T> provides operator overloads only for atomic operations (incorrect code does not compile)
- Any expression with atomic variables will not be computed atomically (easy to make mistakes)
- Member functions make atomic operations explicit
- Compilers understand you either way and do exactly what you asked
  - Not necessarily what you wanted
- Programmers tend to see what they thought you meant not what you really meant (x=x+1)

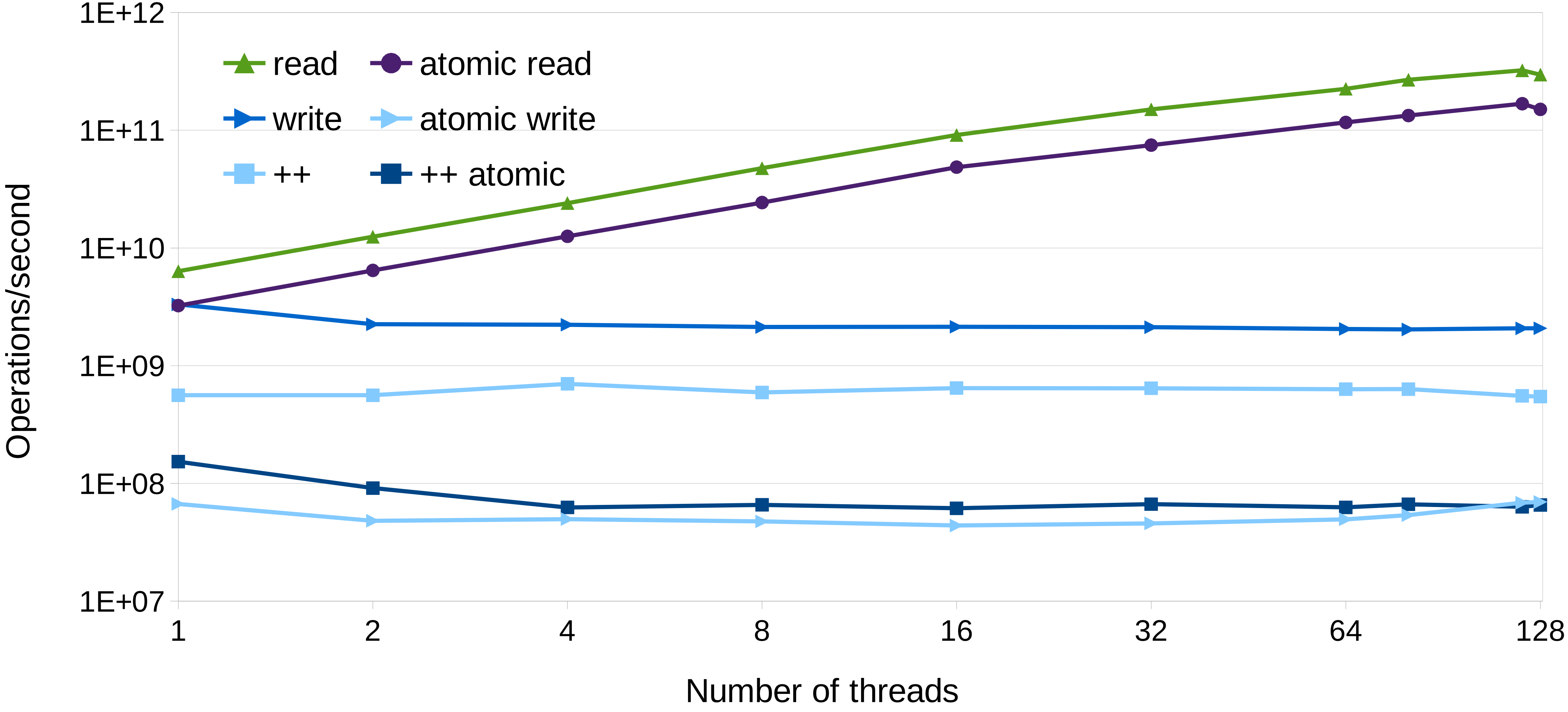
How fast are atomic operations?

How fast are atomic operations?

- Performance should be measured
- Caution: measurement results will be hardware and compiler specific and should not be over-generalized!
- Caution: comparing atomic and non-atomic operations may be instructive for understanding of what the hardware does, but is rarely directly useful
  - Comparing atomic operation with another thread-safe alternative is valid and useful

Are atomic operations slower than non-atomic?

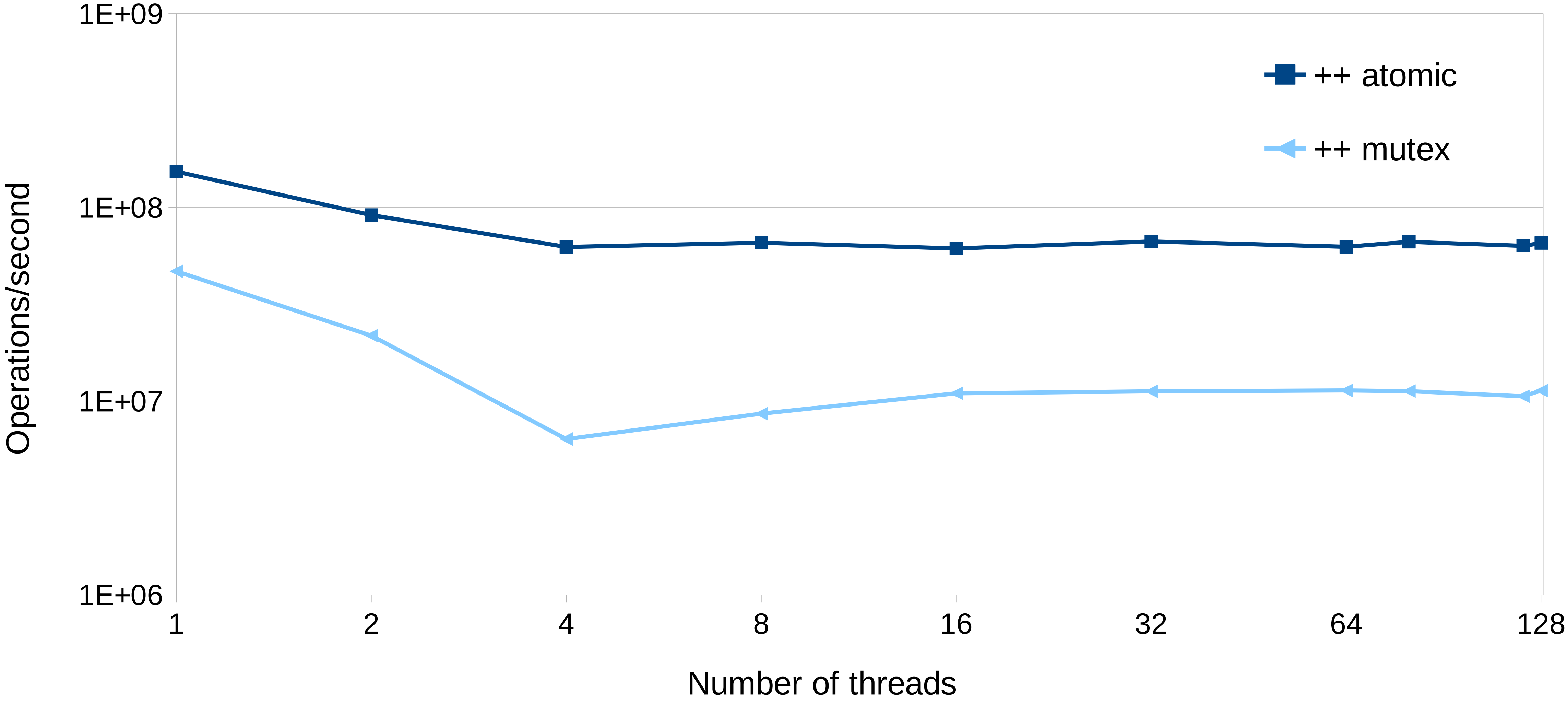
Are atomic operations slower than non-atomic?



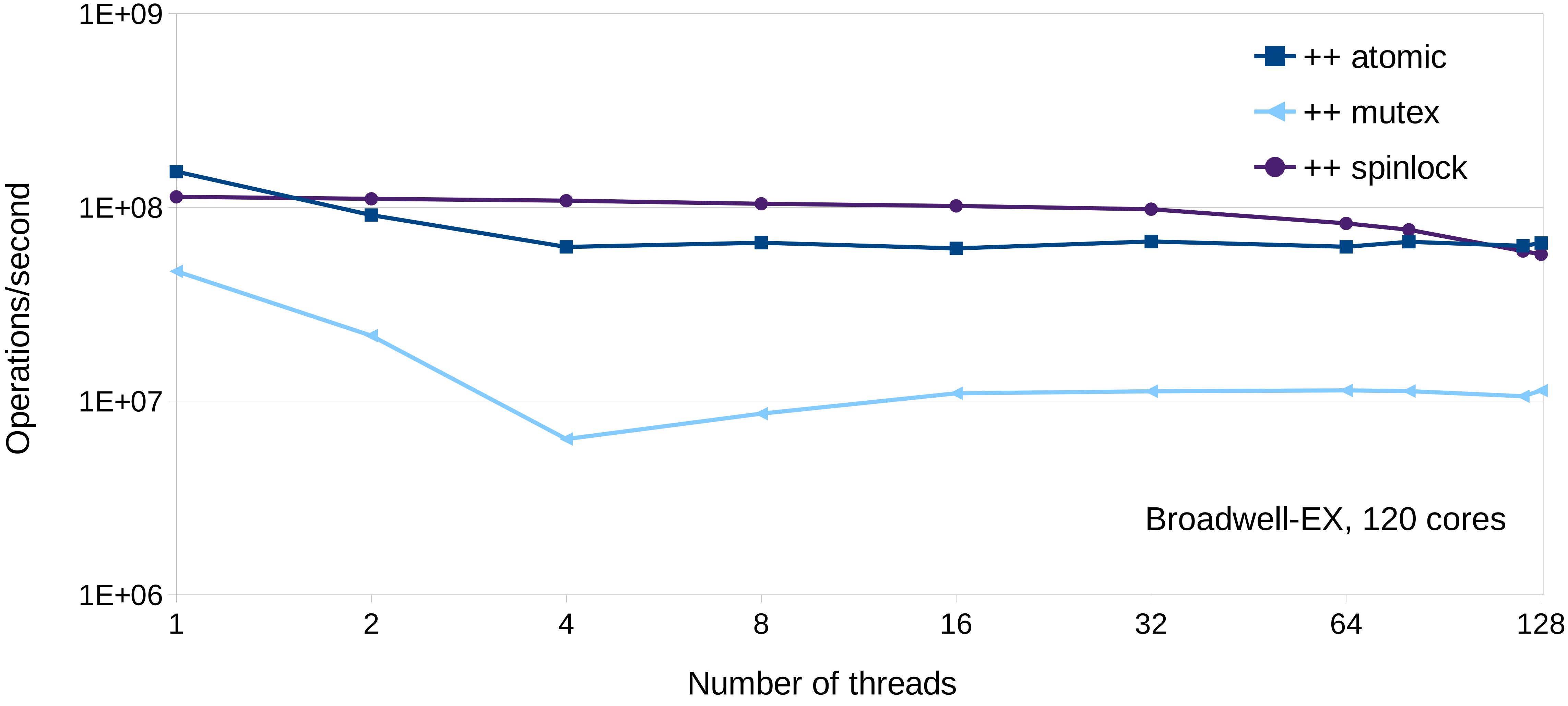
Mentor®
A Siemens Business

Are atomic operations faster than locks?

Are atomic operations faster than locks?

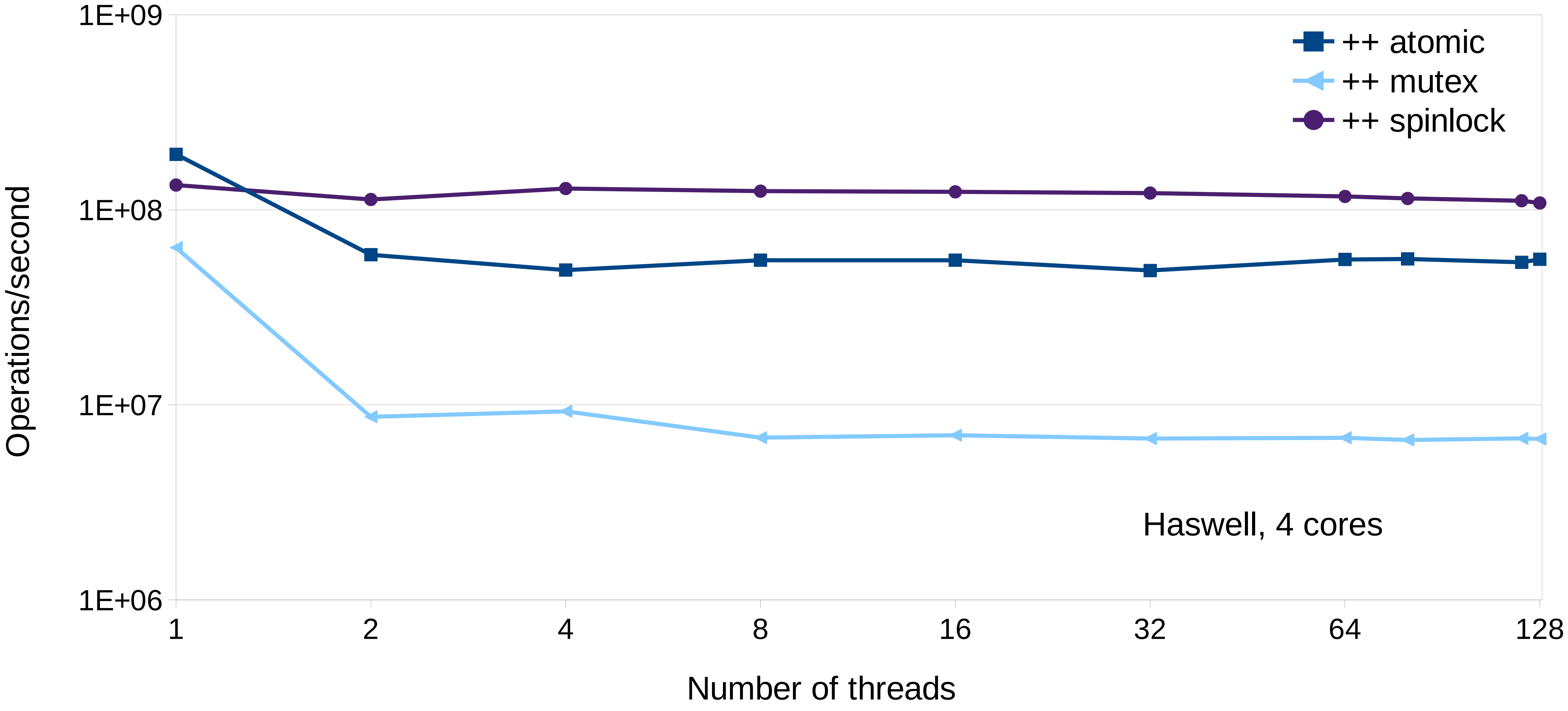


Are atomic operations faster than locks?



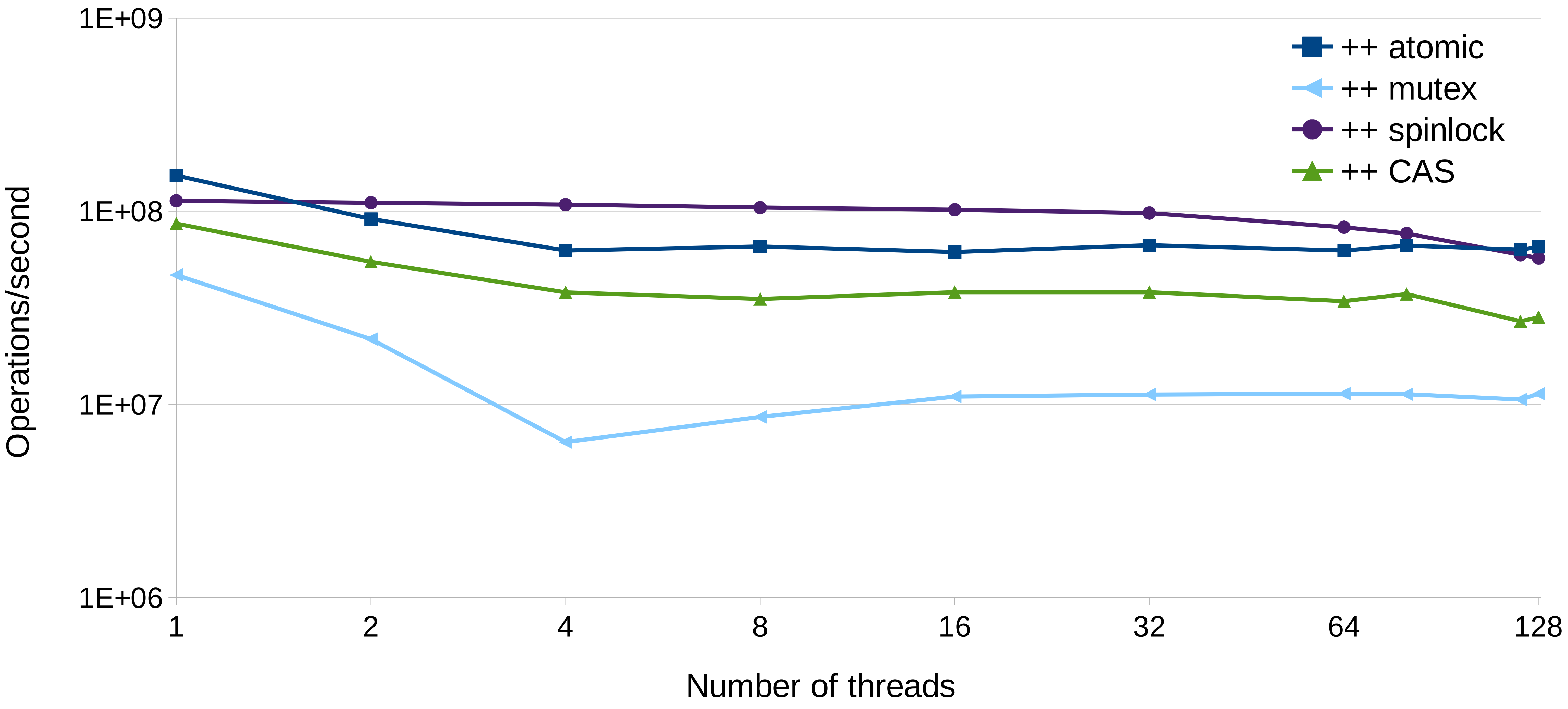
Mentor
A Siemens Business

Are atomic operations faster than locks?



Mentor® A Siemens Business

Remember CAS?



Mentor
A Siemens Business

Is atomic the same as lock-free?

- std::atomic is hiding a huge secret: it’s not always lock-free
  long x;
  struct A { long x; }
  struct B { long x; long y; };
  struct C { long x; long y; long z; };

Is atomic the same as lock-free?

- std::atomic is hiding a huge secret: it's not always lock-free
- std::atomic<T>::is_lock_free()

```
long x;
struct A { long x; }
```
lock-free

```
struct B { long x; long y; };
```
C maybe

```
struct C { long x; long y; long z; };
```
not lock-free

- Results are run-time and platform dependent
  – Why not compile-time? - alignment
  – C++17 adds constexpr is_always_lock_free()

Is atomic the same as lock-free?

16 byte alignment

- std::atomic<T>::is_lock_free() - x86 example

```
long x;
struct A { long x; }
struct B { long x; long y; };  // atomic move to %mmx
```

lock-free

```
struct C { long x; int y; };
struct D { int x; int y; int z; };
```

How about C and D?

```
struct E { long x; long y; long z; }; // >16 bytes
```

not lock-free

Mentor
A Siemens Business

Is atomic the same as lock-free?

- std::atomic<T>::is_lock_free() - x86 example

```
long x;
struct A { long x; }
struct B { long x; long y; };  // 16-byte atomic move -
struct C { long x; int y; };    // atomic move to %mmx
struct D { int x; int y; int z; };         // 12 bytes!
struct E { long x; long y; long z; };   // >16 bytes
```

lock-free

not lock-free

- alignment and padding matter!

Do atomic operations wait on each other?

- Compare accessing shared vavariable

```
std::atomic<int> x;
```

Thread 1	Thread 2
`++x;`	`++x;`

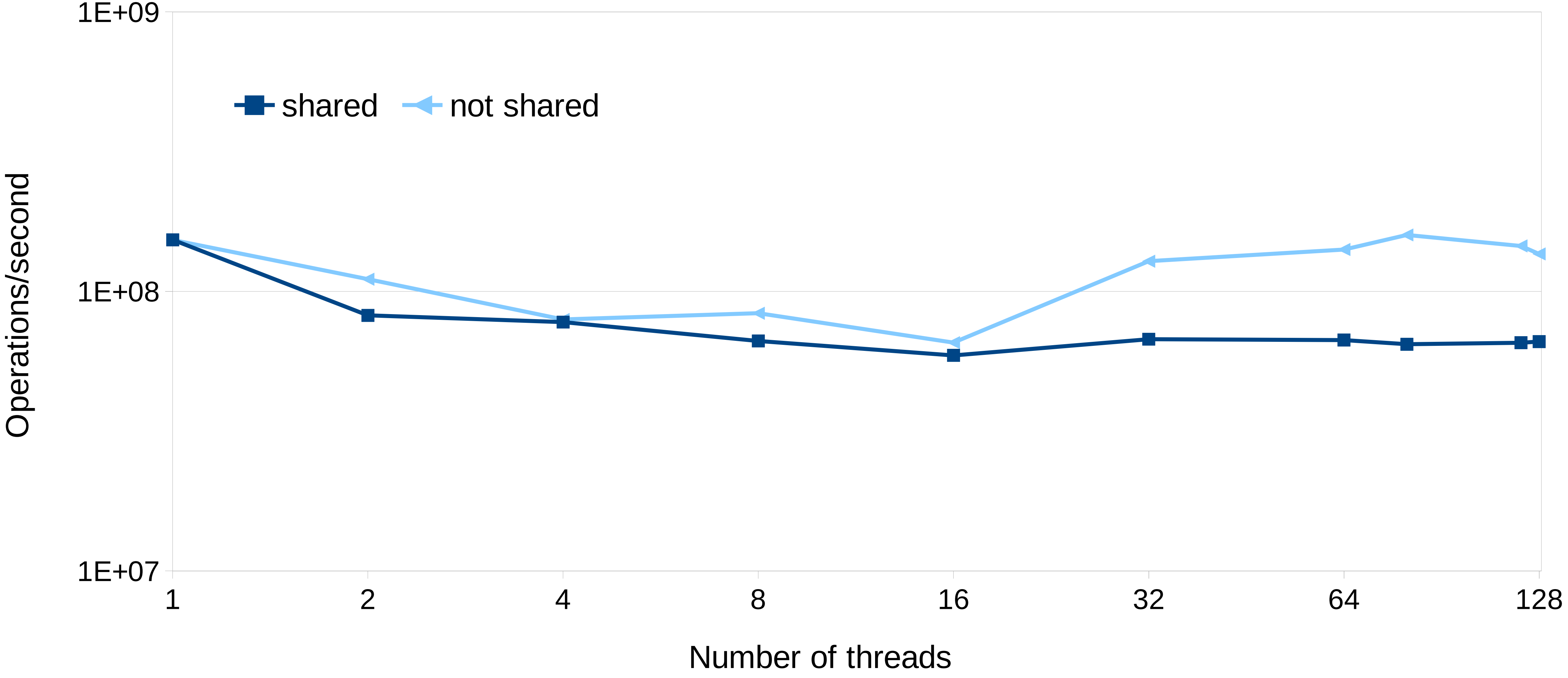
- vs non-shared variable

```
std::atomic<int> x[N];
```

Thread 1	Thread 2
`++x[0];`	`++x[1];`

Mentor A Siemens Business

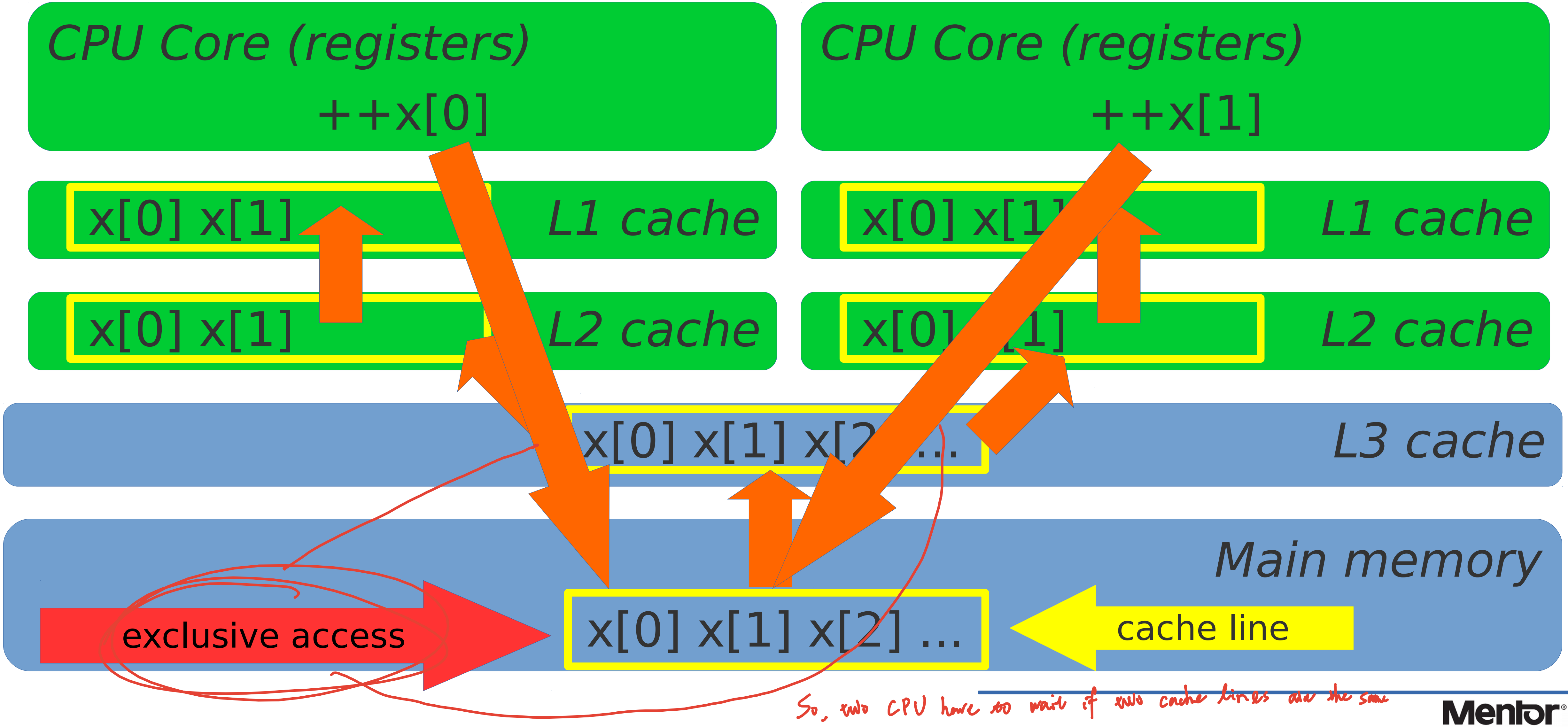
Do atomic operations wait on each other?



Mentor
A Siemens Business

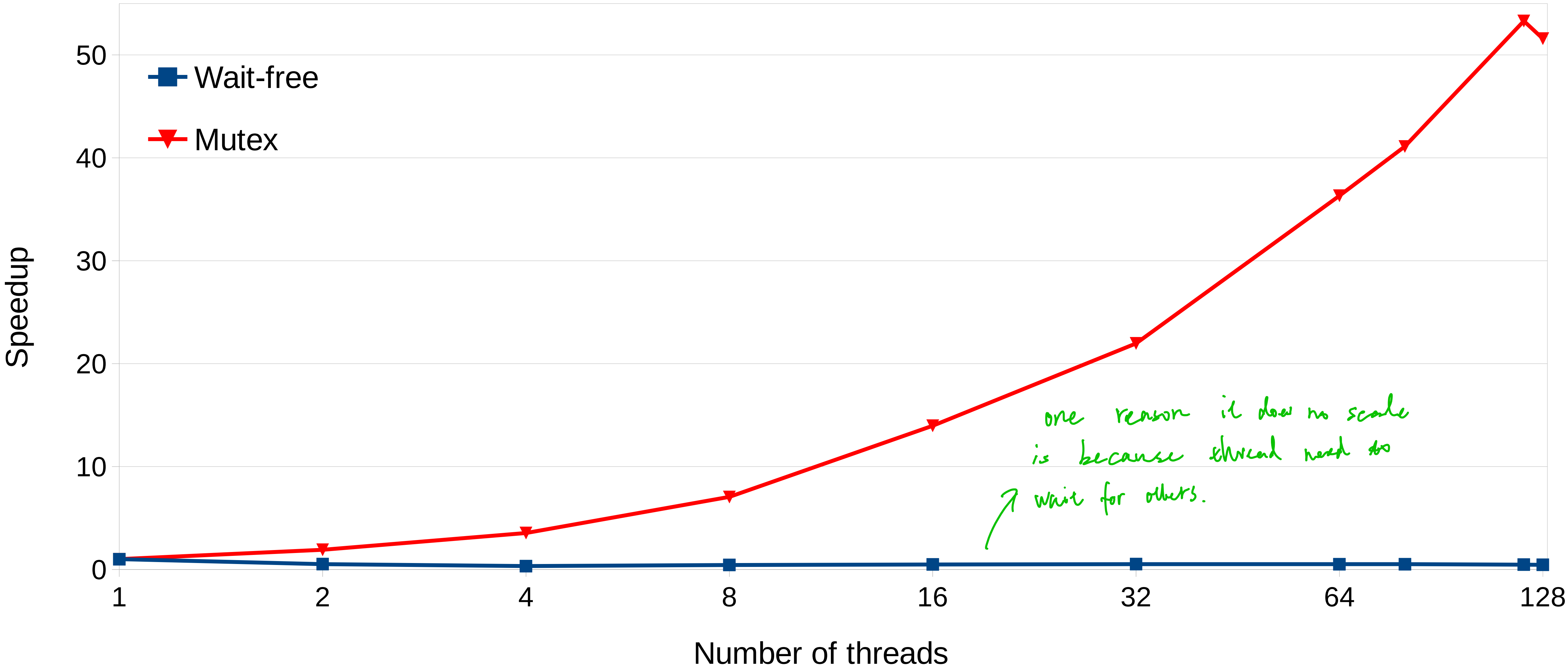
We previously said variables trickle up and down is "like". Actually, it is the whole cache line.

What's really going on?



So, two CPU have to wait if two cache lines are the same

Do atomic operations wait on each other?



Mentor
A Siemens Business

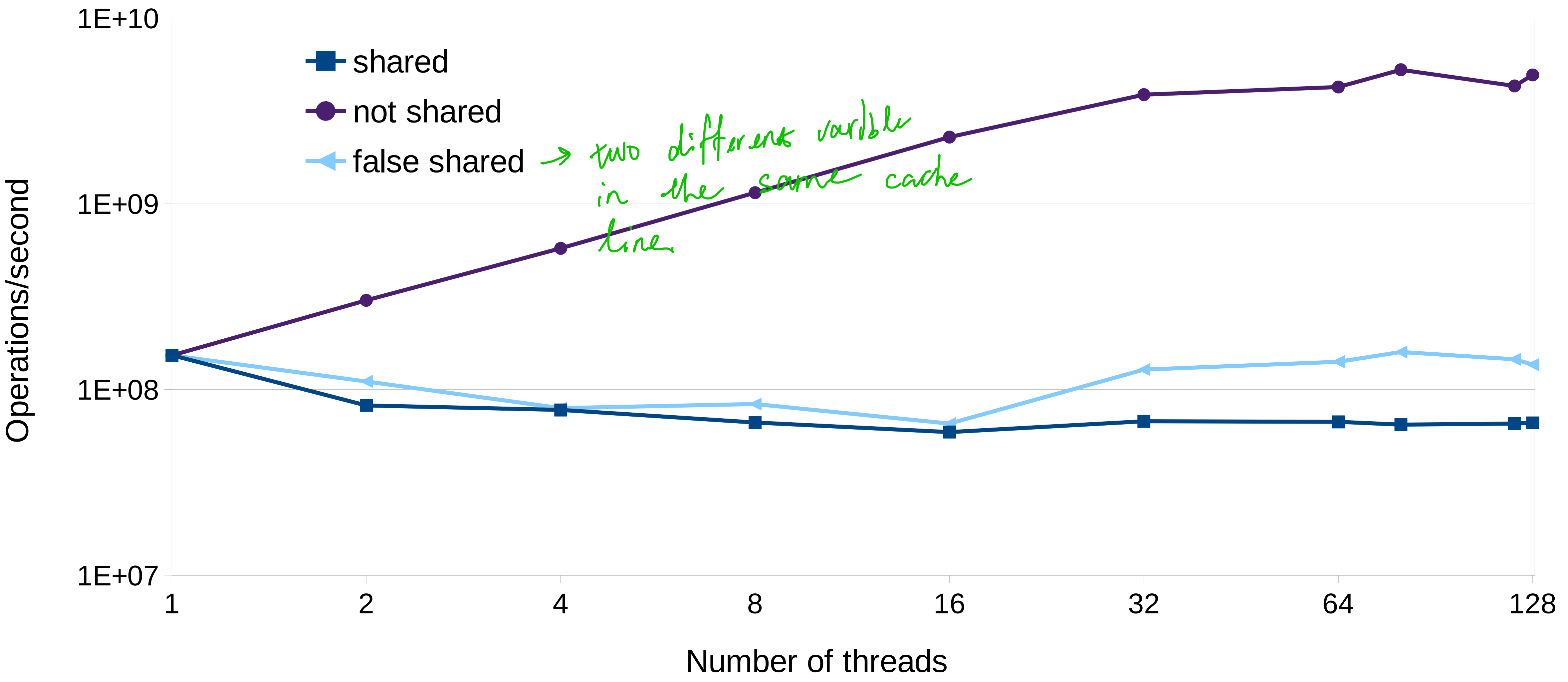
Do atomic operations wait on each other?

- Algorithm rules supreme
- “Wait-free” has nothing to do with time
  - Wait-free refers to the number of compute “steps”
  - Steps do not have to be of the same duration
- Atomic operations do wait on each other
  - In particular, write operations do
  - Read-only operations can scale near-perfectly

if datas are in seperate cache line → no problem.

Mentor A Siemens Business

Do atomic operations wait on each other?



Mentor
A Siemens Business

Do atomic operations wait on each other?

- Atomic operations have to wait for cache line access
  - Price of data sharing without races
  - Accessing different locations in the same cache line still incurs run-time penalty (false sharing)
  - Avoid false sharing by aligning per-thread data to separate cache lines
    - On NUMA machines, may be even separate pages

Strong and weak compare-and-swap

- C++ provides two versions of CAS – weak and strong
- x.**compare_exchange_strong**(old_x, new_x):
  if (x == old_x) { x = new_x; return true; }
  else { old_x = x; return false; }
- x.**compare_exchange_weak**(old_x, new_x): same thing but can "spuriously fail" and return false even if x==old_x
- What is the value of old_x if this happens?

Strong and weak compare-and-swap

- C++ provides two versions of CAS – weak and strong
- x.**compare_exchange_strong**(old_x, new_x):
  if (x == old_x) { x = new_x; return true; }
  else { old_x = x; return false; }
- x.**compare_exchange_weak**(old_x, new_x): same thing but can "spuriously fail" and return false even if x==old_x
- What is the value of old_x if this happens? Must be old_x!
- If weak CAS correctly returns x == old_x, why would it fail?

Strong and weak compare-and-swap

- CAS, conceptually (pseudo-code):

```
bool compare_exchange_strong(T& old_v, T new_v) {
  Lock L;              // Get exclusive access
  T tmp = value;       // Current value of the atomic
  if (tmp != old_v) { old_v = tmp; return false; }
  value = new_v;
  return true;
}
```

- Lock is not a real mutex but some form of exclusive access implemented in hardware

Strong and weak compare-and-swap

- Read is faster than write:

```
bool compare_exchange_strong(T& old_v, T new_v) {
  T tmp = value;   // Current value of the atomic
  if (tmp != old_v) { old_v = tmp; return false; }
  Lock L;          // Get exclusive access
  tmp = value;     // value could have changed!
  if (tmp != olv_v) { old_v = tmp; return false; }
  value = new_v;
  return true;
}
```

- Double-checked locking pattern is back!

Strong and weak compare-and-swap

- If exclusive access is hard to get, let someone else try:

```
bool compare_exchange_weak(T& old_v, T new_v) {
  T tmp = value;   // Current value of the atomic
  if (tmp != old_v) { old_v = tmp; return false; }
  TimedLock L;       // Get exclusive access or fail
  if (!L.locked()) return false;   // old_v is correct
  tmp = value;      // value could have changed!
  if (tmp != olv_v) { old_v = tmp; return false; }
  value = new_v;
  return true;
}
```

MUCH

But wait, there is more…

- Atomic variables are rarely used by themselves
- Atomic queue:

```
int q[N];
std::atomic<size_t> front;
void push(int x) {
  size_t my_slot = front.fetch_add(1);
  q[my_slot] = x;
}
```

atomic

exclusive slot

- Atomic variable is an index to (non-atomic) memory

But wait, there is ~~more~~ MUCH more…

- Atomic list:

```
struct node { int value; node* next; };
std::atomic<node*> head;
void push_front(int x) {
  node* new_n = new node;
  new_n→value = x;
  node* old_h = head;
  do { new_n→next = old_h; }
  while (!head.compare_exchange_strong(old_h,new_n);
}
```

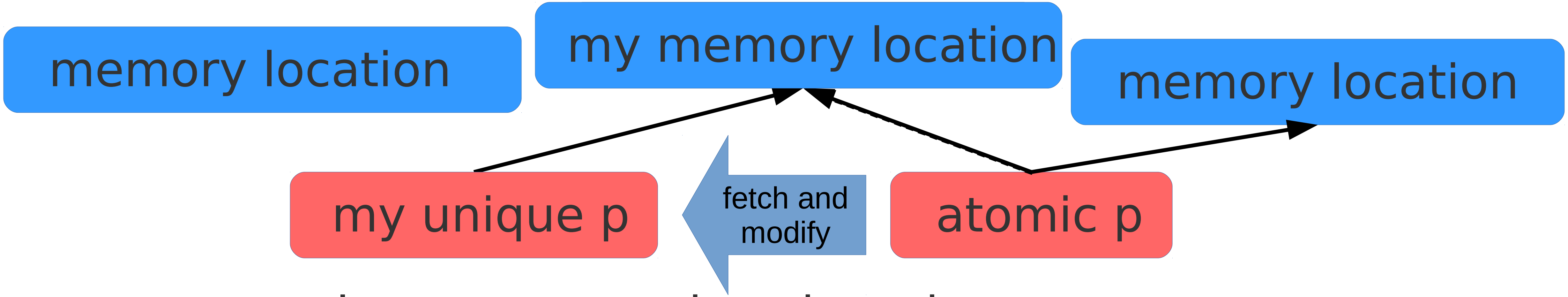
new node is new head

head has not changed

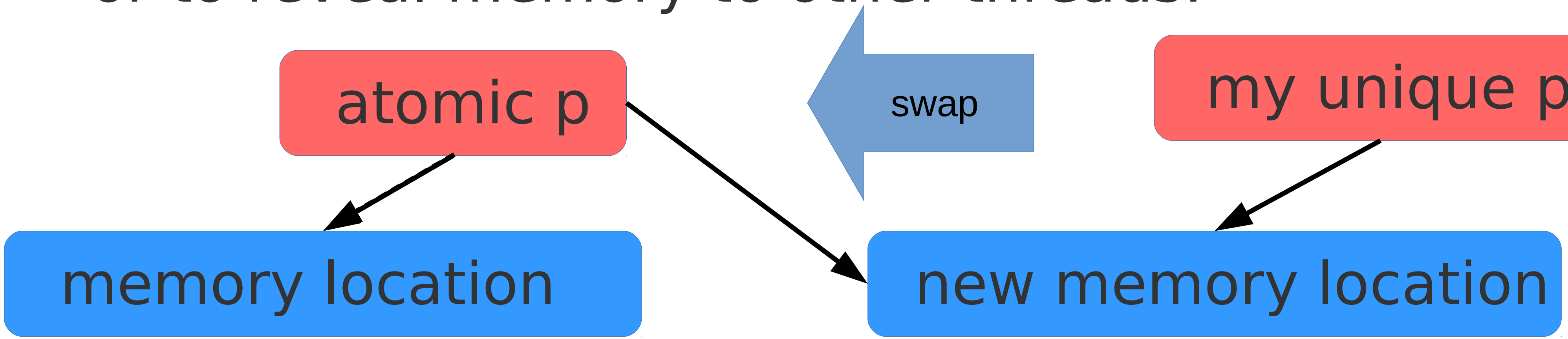
- Atomic variable is a pointer to (non-atomic) memory

Atomic variables as gateways to memory access (generalized pointers)

- Atomics are used to get exclusive access to memory:



- or to reveal memory to other threads:

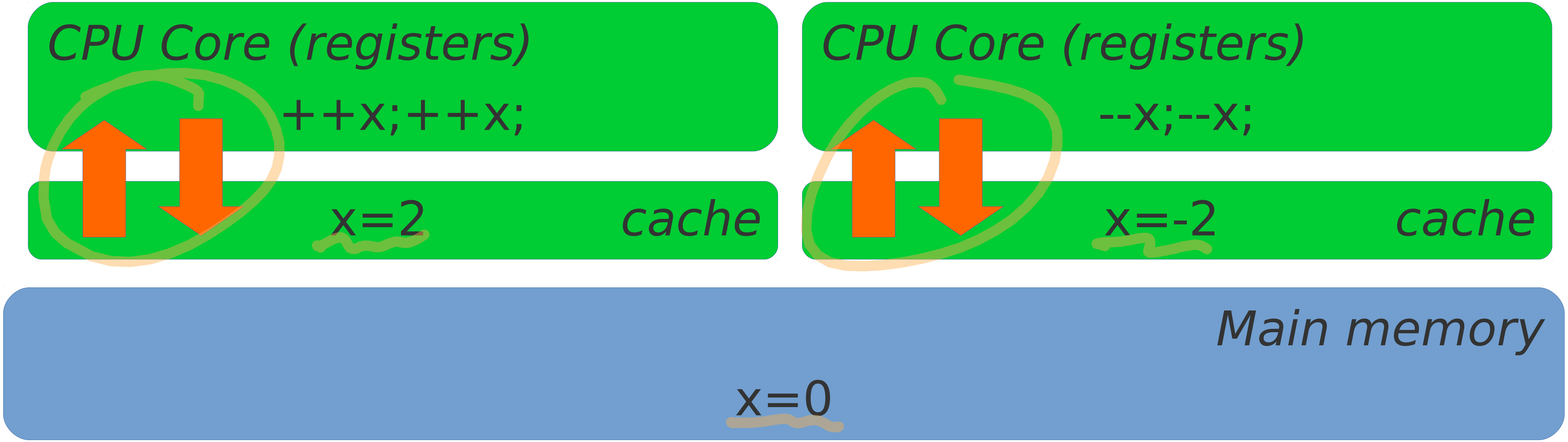


Atomic variables as gateways to memory access (generalized pointers)

- Atomics are used to get exclusive access to memory or to reveal memory to other threads
- But most memory is not atomic!
- What guarantees that other threads see this memory in the desired state
  - For acquiring exclusive access: data may be prepared by other threads, must be completed
  - For releasing into shared access: data is prepared by the owner thread, must become visible to everyone

Memory barriers – the other side of atomics

- Memory barriers control how changes to memory made by one CPU become visible to other CPUs



- Visibility of non-atomic changes is not guaranteed

Memory barriers

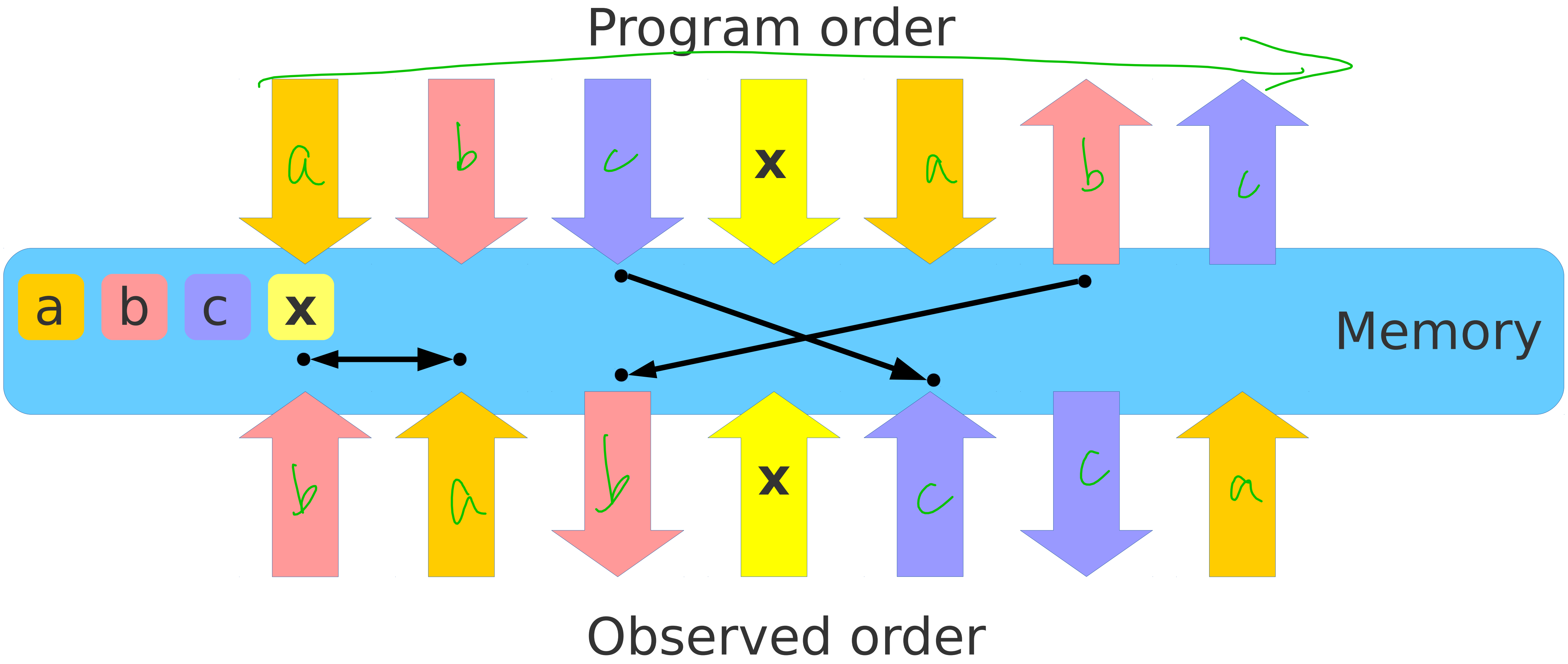
- Synchronization of data access is not possible if we cannot control the order of memory accesses
- This is global control, across all CPUs
- Such control is provided by memory barriers
- Memory barriers are implemented by the hardware
- Memory barriers are invoked through processor-specific instructions (or modifiers on other instructions)
  - Barriers are often “attributes” on read/write operations, ensuring the specified order of reads and writes

Memory barriers in C++

- C++03 as no portable memory barriers
- C++11 provides standard memory barriers
- Memory barriers are closely related to “memory order” – they are what ensures the memory order
- C++ memory barriers are modifiers on atomic operations
  - Actual implementation may vary
- Example:
  std::atomic<int> x;
  x.store(1, **std::memory_order_release**);

No barriers – std::memory_order_relaxed

- x.fetch_add(1, **std::memory_order_relaxed**);

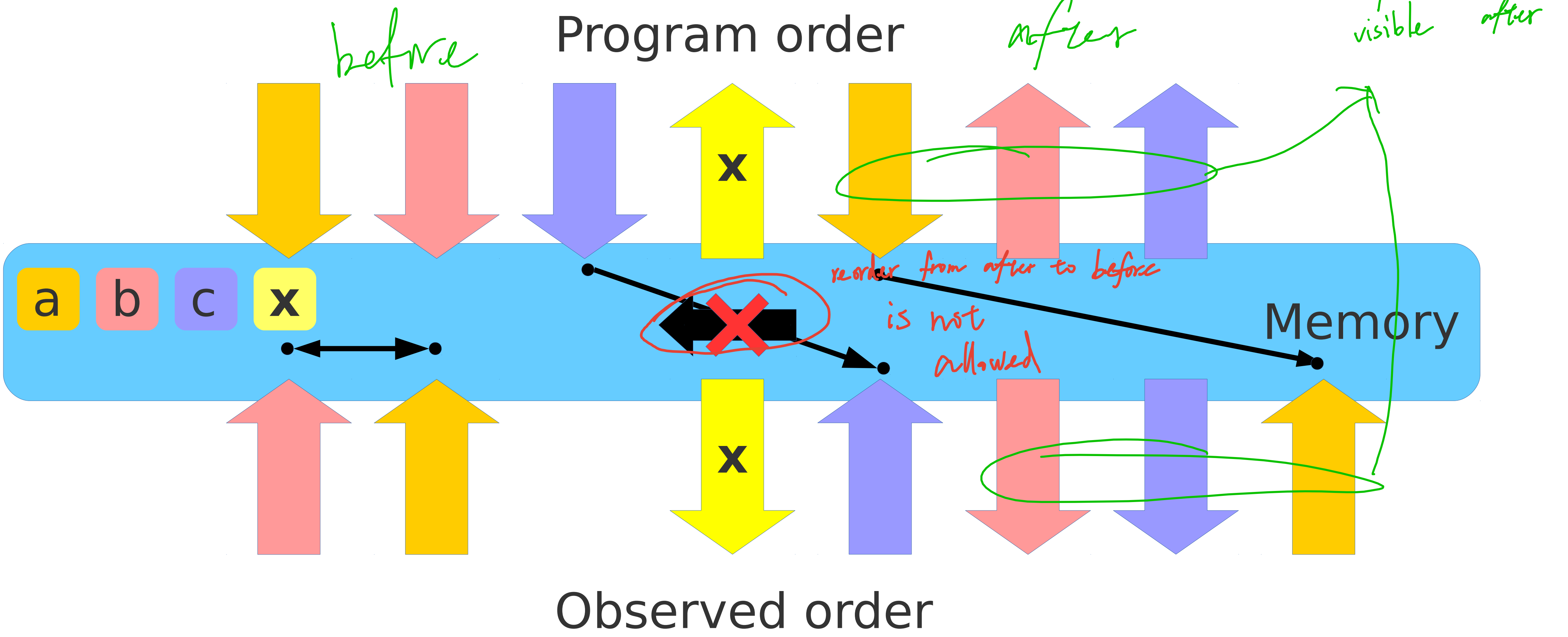


Acquire barrier

- Acquire barrier guarantees that all memory operations scheduled after the barrier in the program order become visible after the barrier
  - “All operations” not “all reads” or “all writes”, i.e. both reads and writes
  - “All operations” not just operations on the same variable that the barrier was on
- Reads and writes cannot be reordered from after to before the barrier
  - Only for the thread that issued the barrier!

Acquire barrier – std::memory_order_acquire

- x.load(std::**memory_order_acquire**);

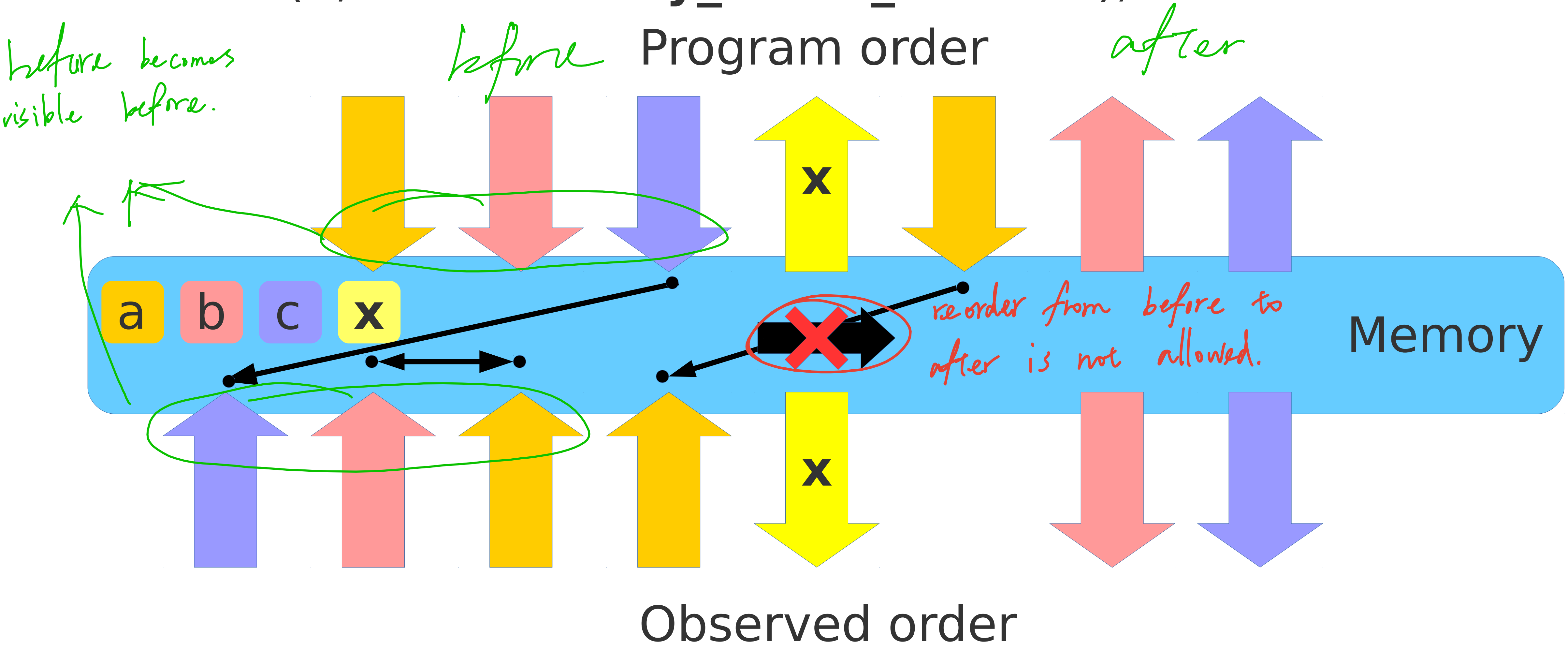


Release barrier

- Release barrier guarantees that all memory operations scheduled before the barrier in the program order become visible before the barrier
- Reads and writes cannot be reordered from before to after the barrier
  - Only for the thread that issued the barrier!

Release barrier – std::memory_order_release

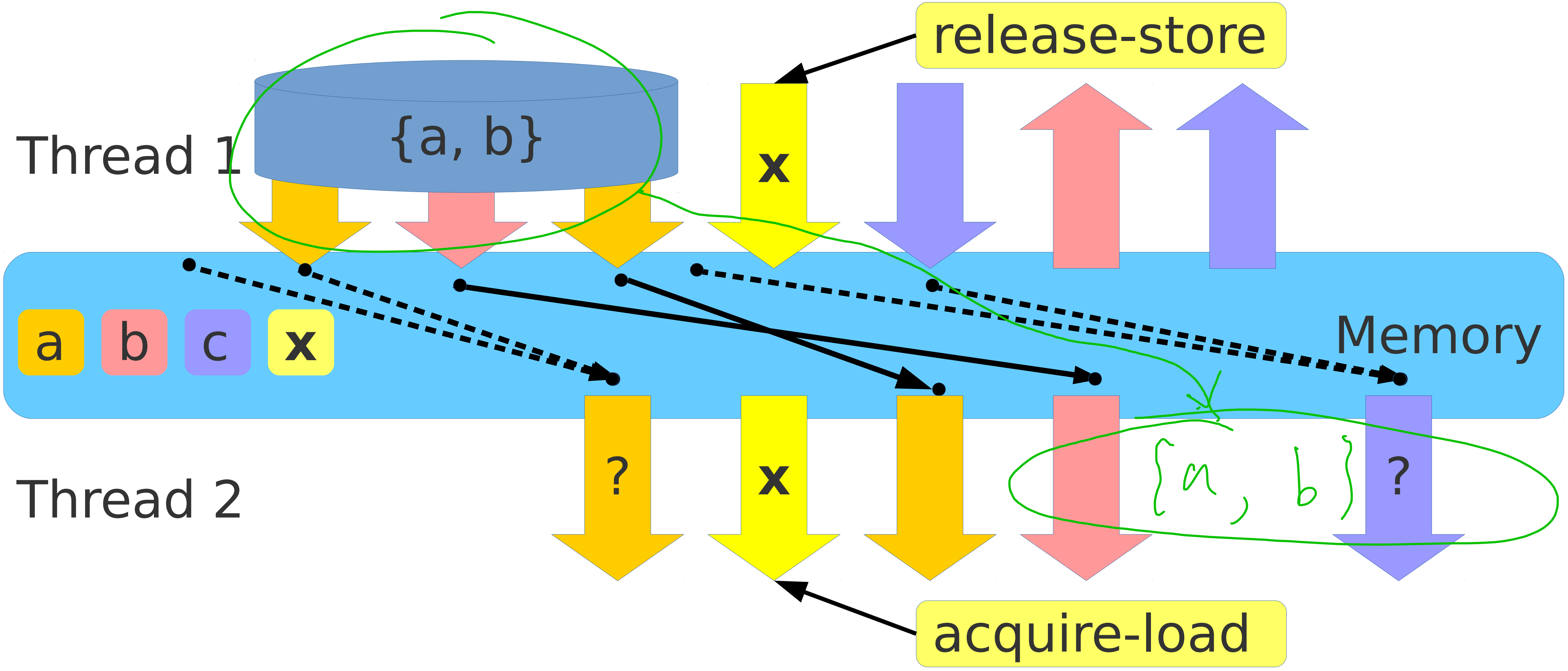
- x.store(1, std::**memory_order_release**);



Acquire-release order

- Acquire and release barriers are often used together:
- Thread 1 writes atomic variable **x** with release barrier
- Thread 2 reads atomic variable **x** with acquire barrier
- All memory writes that happen in thread 1 before the barrier (in program order) become visible in thread 2 after the barrier
- Thread 1 prepares data (does some writes) then **releases** (publishes) it by updating atomic variable x
- Thread 2 **acquires** atomic variable x and the data is guaranteed to be visible

Acquire-release protocol

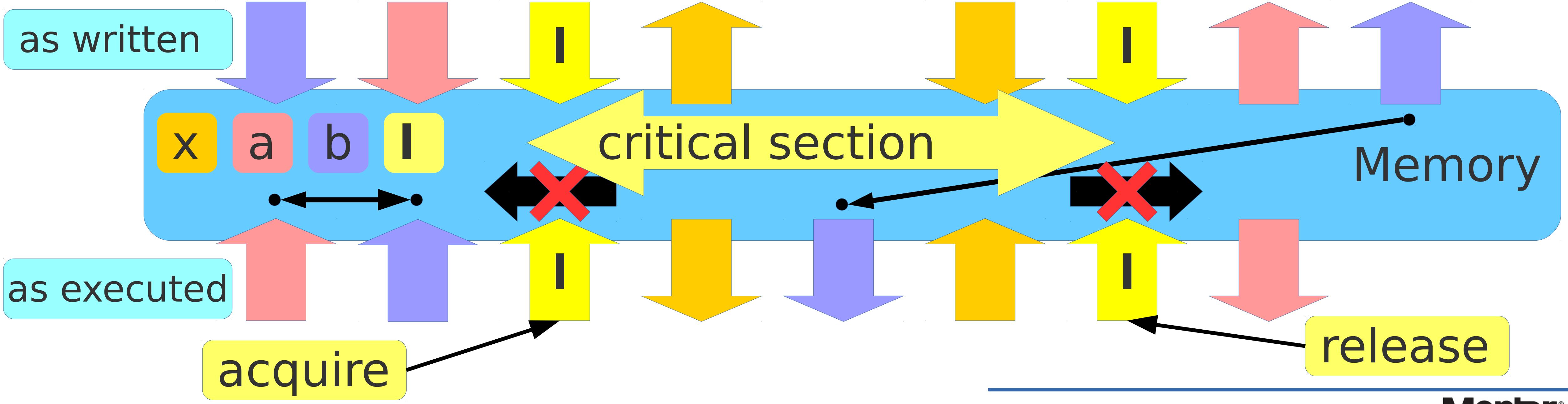


- Both threads must use matching barriers and the same **x**!

Barriers and locks

- Acquire and release barriers are used in locks:

```
Lock L;       std::atomic<int> l(0);
L.lock();     l.store(1, std::memory_order_acquire);
              while (l.exchange(1, std::memory_order_acquire));
++x;          ++x;
L.unlock();   l.store(0, std::memory_order_release);
```



Mentor
A Siemens Business

Bidirectional barriers

- Acquire-Release (std::**memory_order_acq_rel**) combines acquire and release barriers – no operation can move across the barrier
  - But only if both threads use the same atomic variable!
- Sequential consistency (std::**memory_order_seq_cst**) removes that requirement and establishes single total modification order of atomic variables

Why does CAS have two memory orders?

- Read is faster than write:

```
bool compare_exchange_strong(T& old_v, T new_v,
memory_order on_success, memory_order on_failure) {
  T tmp = value.load(on_failure);
  if (tmp != old_v) { old_v = tmp; return false; }
  Lock L;              // Get exclusive access
  tmp = value;         // value could have changed!
  if (tmp != olv_v) { old_v = tmp; return false; }
  value.store(new_v, on_success);
  return true;
}
```

Default memory order

- What is the **default** memory order if none is specified?
  y=x.load(); x.fetch_add(42);
- std::**memory_order_seq_cst** – the strongest order
- Same for the overloaded operators:
  y=x; x += 42;
- Can't change the memory order for the operators
- Can specify memory order for functions to be weaker than the default:
  y=x.load(std::memory_order_acquire);
  x.fetch_add(42, std::memory_order_relaxed);

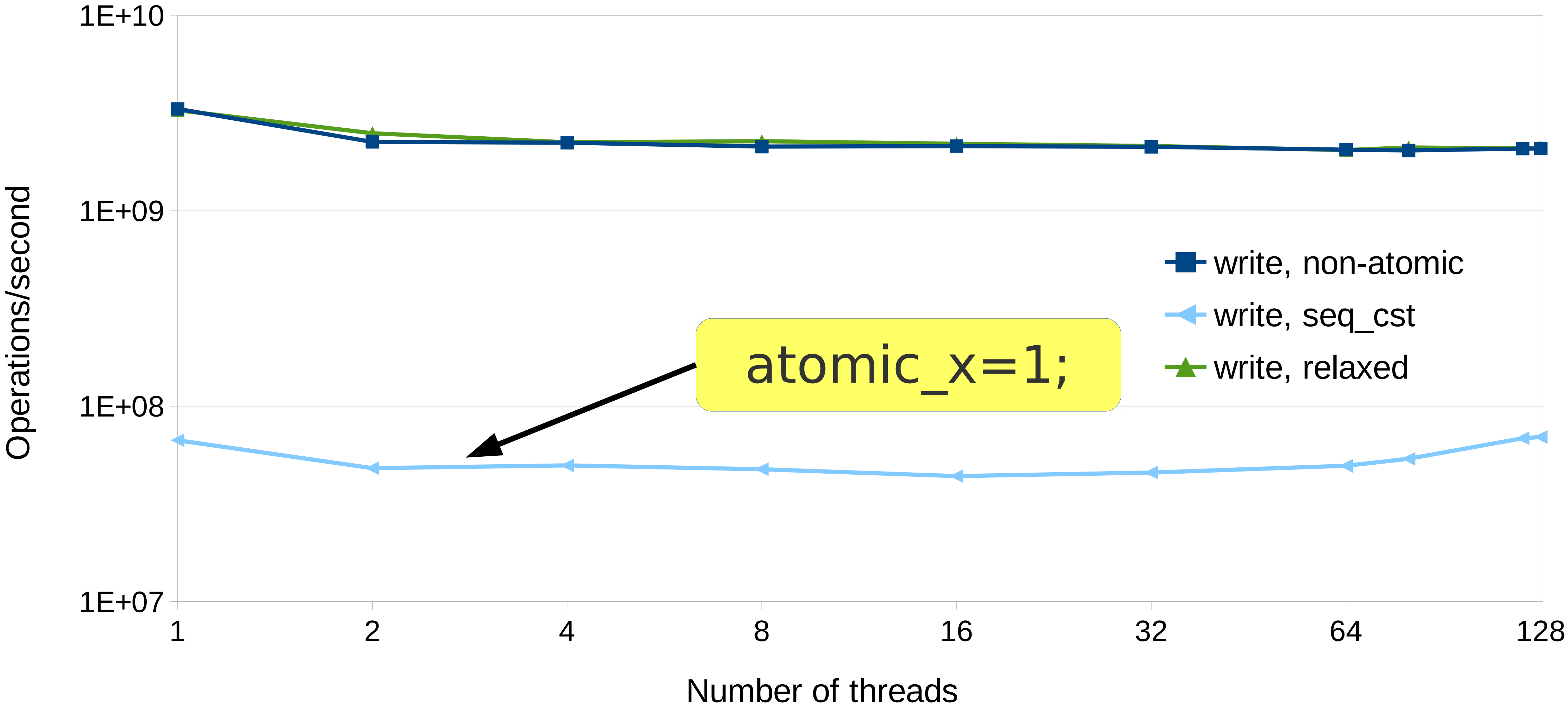
Why change memory order?

- Performance
- Expressing intent
- As programmers we address two audiences...

Why change memory order?

- Performance
  - Audience #1 – computers
- Expressing intent
  - Audience #2 – other programmers

Memory barriers and performance



Memory barriers are expensive

- Memory barriers may be more expensive than atomic operations themselves
- Caution: not all platforms provide all barriers, so performance measurements may be misleading
- On x86:
  - all loads are acquire-loads, all stores are release-stores
    - but adding the other barrier is expensive
  - all read-modify-write operations are acquire-release
  - acq_rel and seq_cst are the same thing

Memory order expresses programmer’s intent

- Lock-free code is hard to write
  – It’s harder to write if you want it it work correctly
- It’s also hard to read, so clarity matters
  – Also to the writer, to reason that it is correct
- Memory order specification is important to express why the atomic operations are used and what the programmer wanted to happen

Mentor® A Siemens Business

Memory order expresses programmer's intent

- What you wrote:
  std::atomic<size_t> count;
  count.**fetch_add**(1, **std::memory_order_relaxed**);
- What you meant:
  count is incremented concurrently, not used to index any memory or as a reference count (no other memory access depends on it) – this is some sort of counter
- Note: on x86, fetch_add() is actually memory_order_acc_rel
- But note: the compiler could know the difference and reorder some operations across fetch_add()

Memory order expresses programmer's intent

- What you wrote:
  std::atomic<size_t> count;
  count.**fetch_add**(1, **std::memory_order_release**);
- What you meant:
  count indexes some memory that was prepared by this thread and is now released to other threads, like this:
  T data[max_count];
  initialize(data[count.load(std::memory_order_relaxed)]);
  count.**fetch_add**(1, **std::memory_order_release**);

now they can see it

nobody can see new data yet

Memory order expresses programmer’s intent

- What you wrote:
  std::atomic<size_t> count;
  ++count;
- What you meant:
  count one of several atomic variables used to access the same memory and kept in sync by some very tricky code
- or:
  I have no idea what I am doing but it seems to work; using a lock would probably work just as well but this is way cooler!

Note on sequential consistency

- Sequential consistency makes your programs slow

Note on [illegible]ential consiste[illegible]

- Sequentia[illegible]ncy makes y[illegible]ams slow

Note on sequential consistency

- Sequential consistency makes your program easier to understand and often has no performance penalty
- But making every atomic operation memory_order_seq_cst is not necessary for sequential consistency and usually obscures the programmer’s intent
- Consider:
  - Lock-based program can be sequentially consistent, but
  - Lock implementation does not need memory_order_seq_cst, only memory_order_acquire and memory_order_release

Mandatory gripe about the C++ standard

- What you wrote:
  class C { std::atomic<size_t> N; T* p; … };
  C::~C() { cleanup(p, N.**load**(std::memory_order_relaxed));
- What you said:
  C::*N may be accessed by another thread while the object is being destructed – be very afraid!*
- What you probably meant:
  I wish the standard let me say N.**load_nonatomic**() *so I don't have to terrify people unless I really want to*

C++ and std::atomic

- Atomic variables and operations on them
  - Member function operations (use them) and operators
- Performance of atomic operations (not always fastest)
- Memory barriers
  - Essential for interaction of threads through memory
  - Significantly affect performance

you will get better performance on lock-free programming often.
But you need to "work on it", use right memory barrier...

Mentor
A Siemens Business

When to use std::atomic in your C++ code

- High-performance concurrent lock-free data structures
  - Prove it by measuring performance
- Data structures that are difficult or expensive to implement with locks (lists, trees)
- When drawbacks of locks are important (deadlocks, priority conflicts, latency problems)
- When concurrent synchronization can be achieved by the cheapest atomic operations (load and store) – see my talk on RCU

Mentor
A Siemens Business

```
std::atomic<questions> any_questions;
any_questions.load();
```

Mentor®

A Siemens Business

www.mentor.com