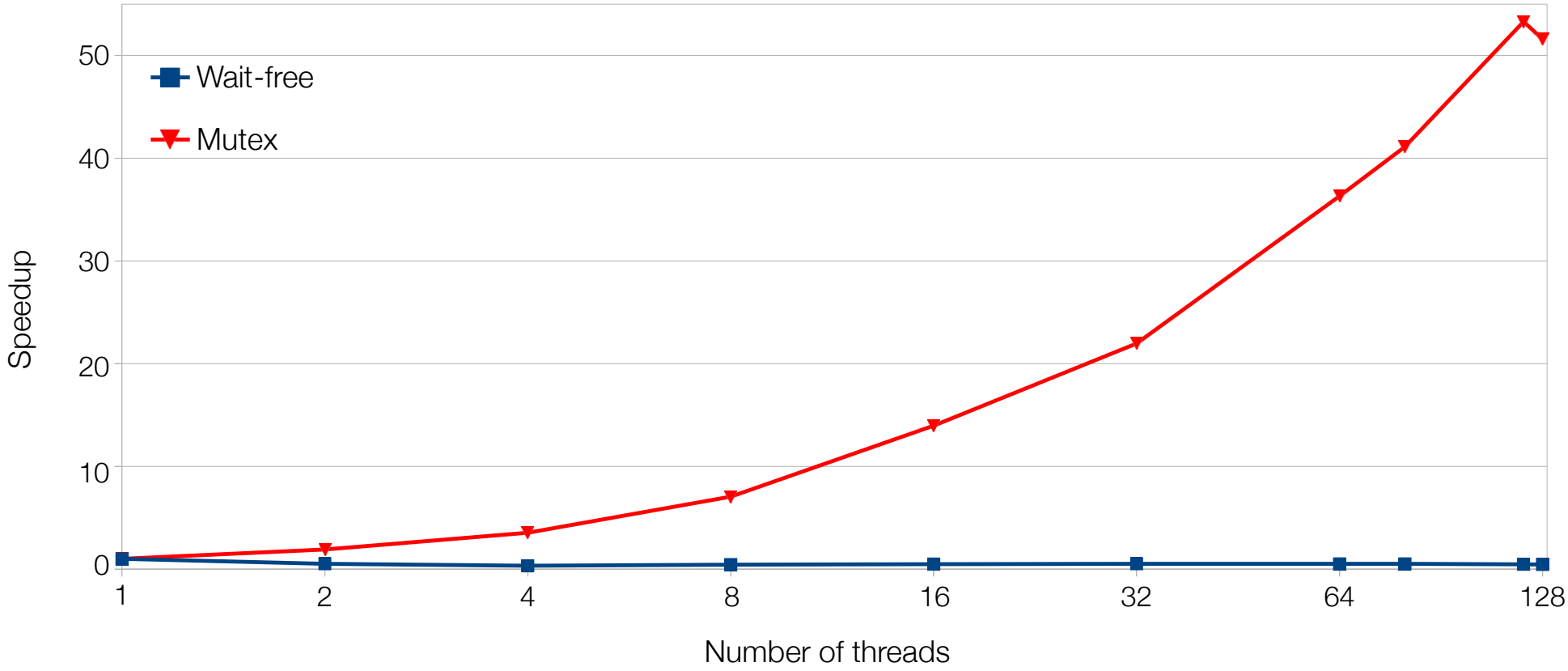
C++ atomics: from basic to advanced. What do they do?

Adapted from
CppCon 2017

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”



Lock-free means “fast”

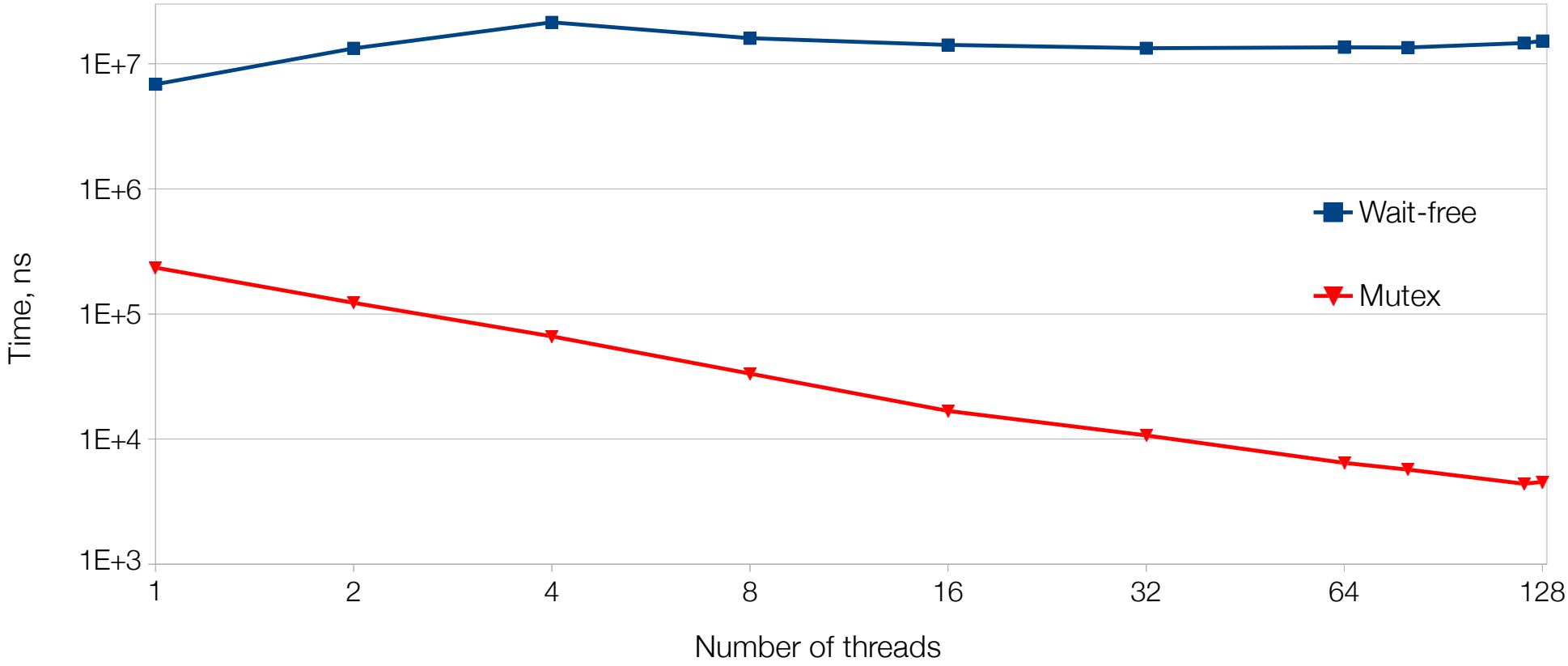
Atomic

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

Mutex

```
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?



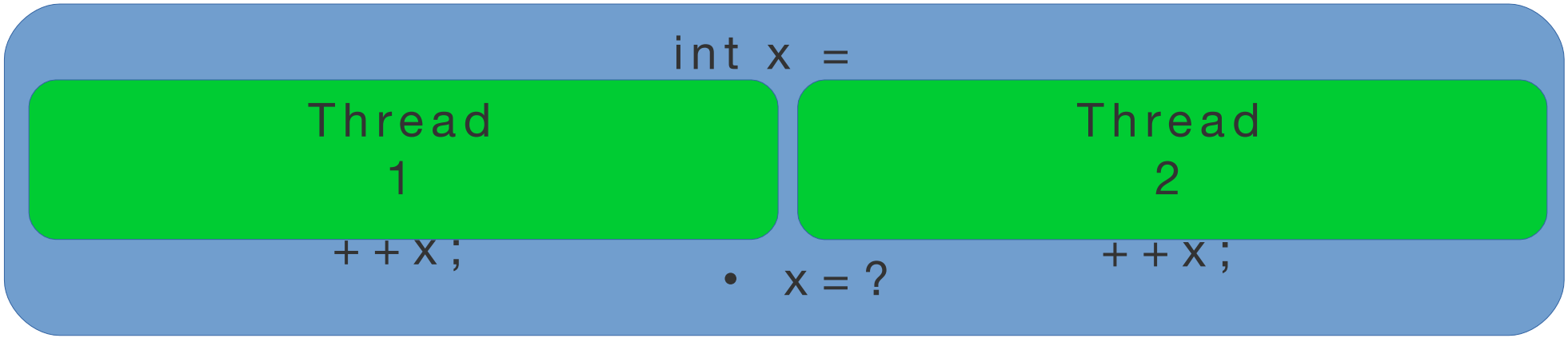
Is lock-free faster?

- 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?

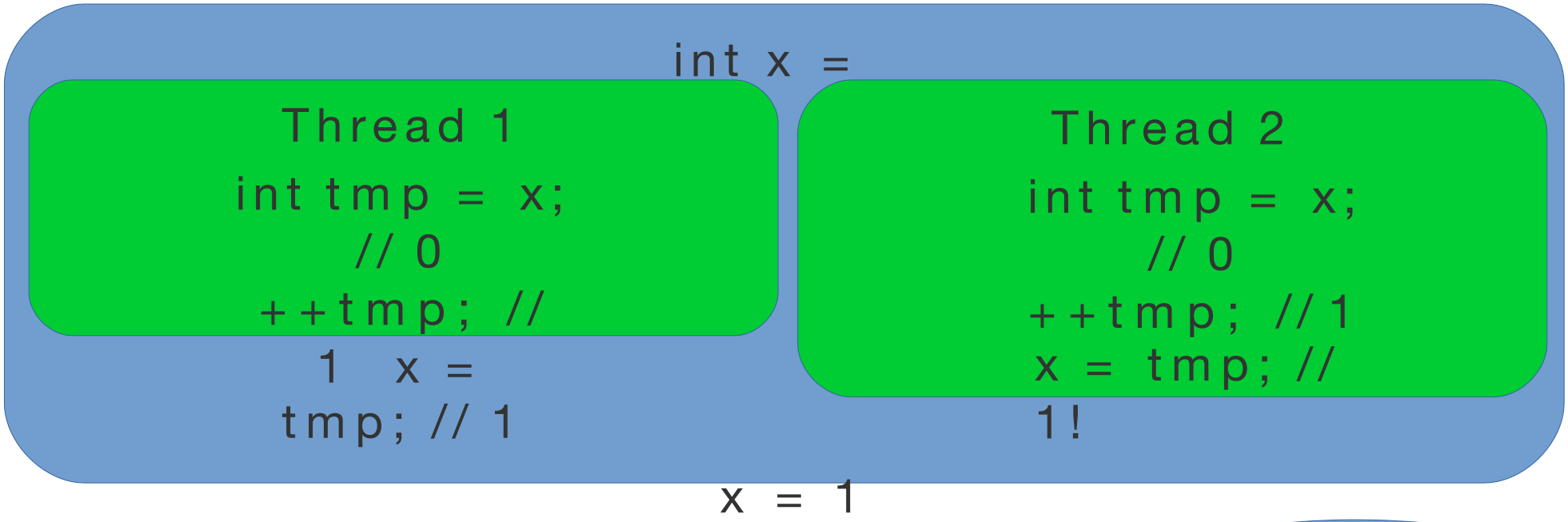
- Atomic operation is an operation that is guaranteed to be execute as a single transaction:
  - Other threads will see the state of the system before the operation started or after it finished, but cannot see any intermediate state
  - At the low level, atomic operations are special hardware instructions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

Atomic operation example



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

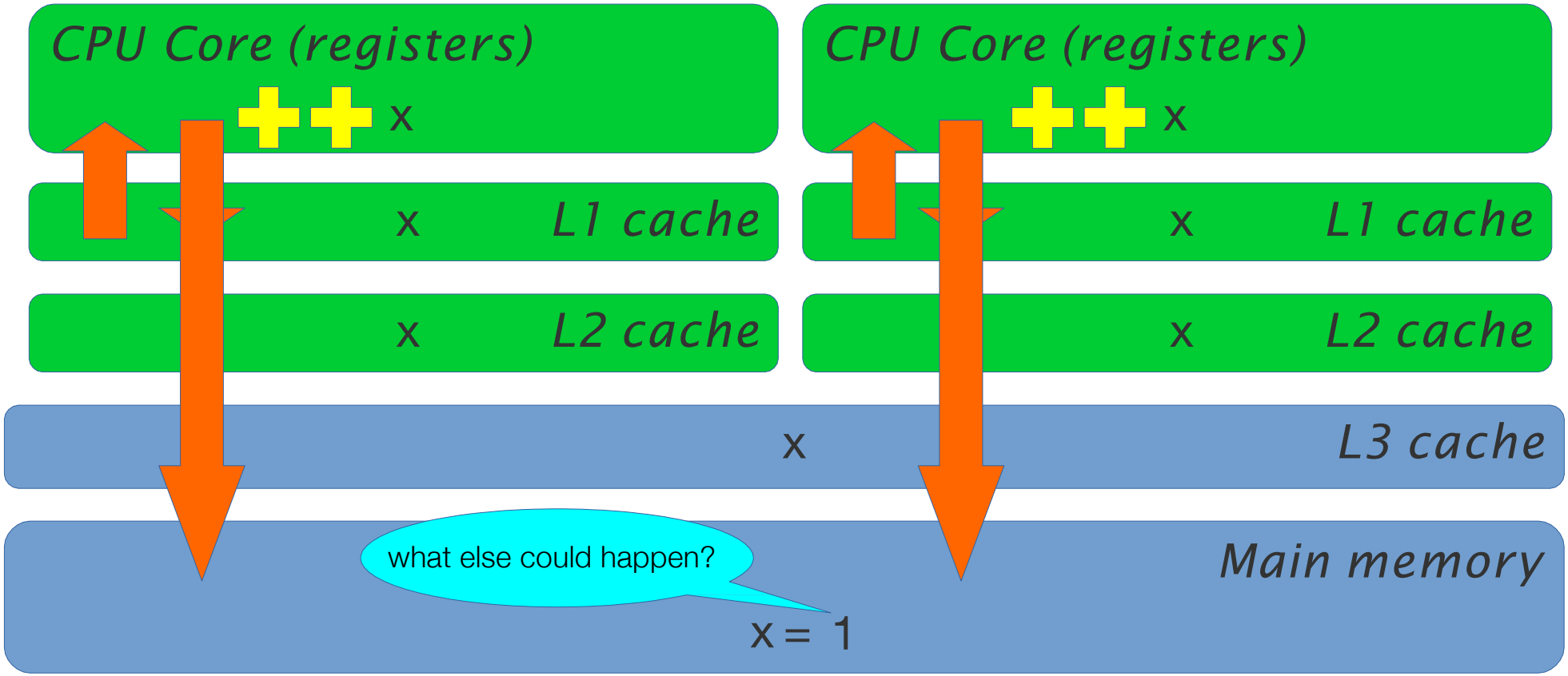
Atomic operation example



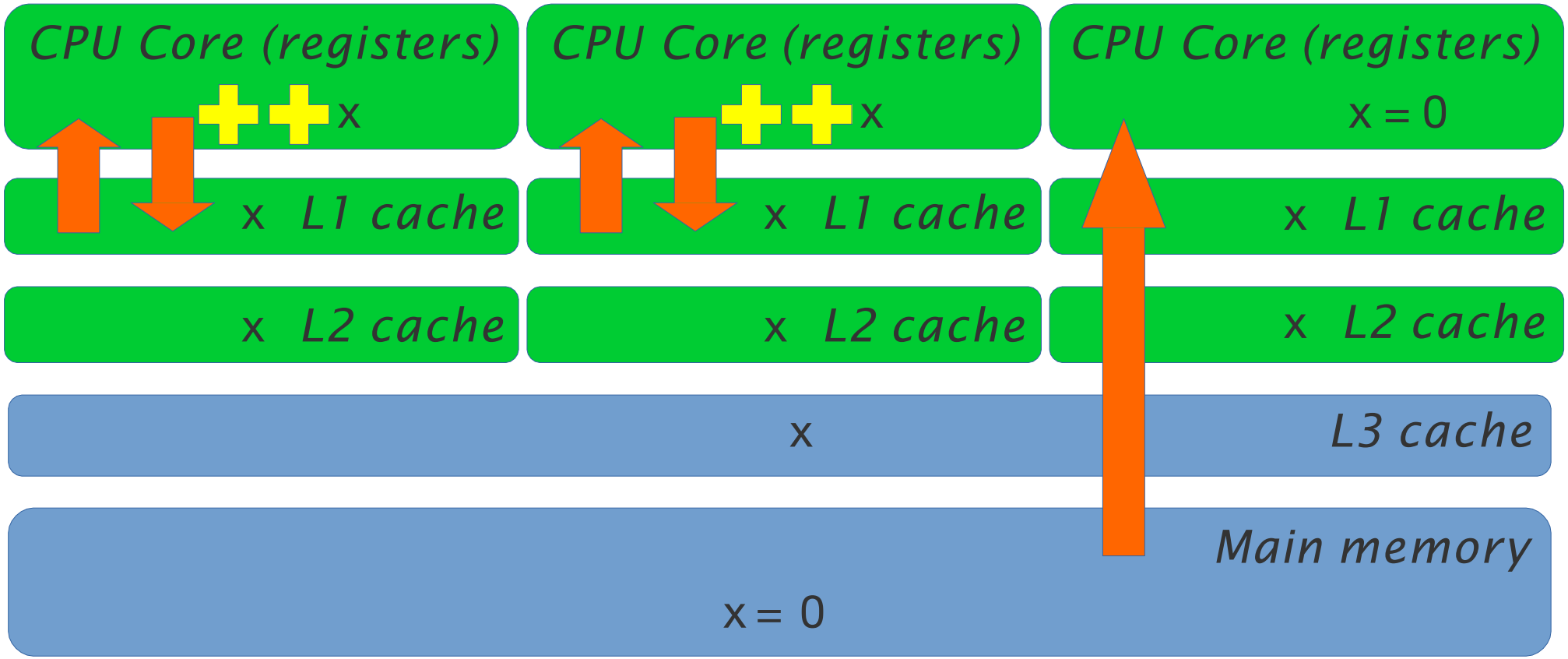
- Read-modify-write increment is non-atomic
- This is a data race (i.e. undefined behavior)

what else could happen?

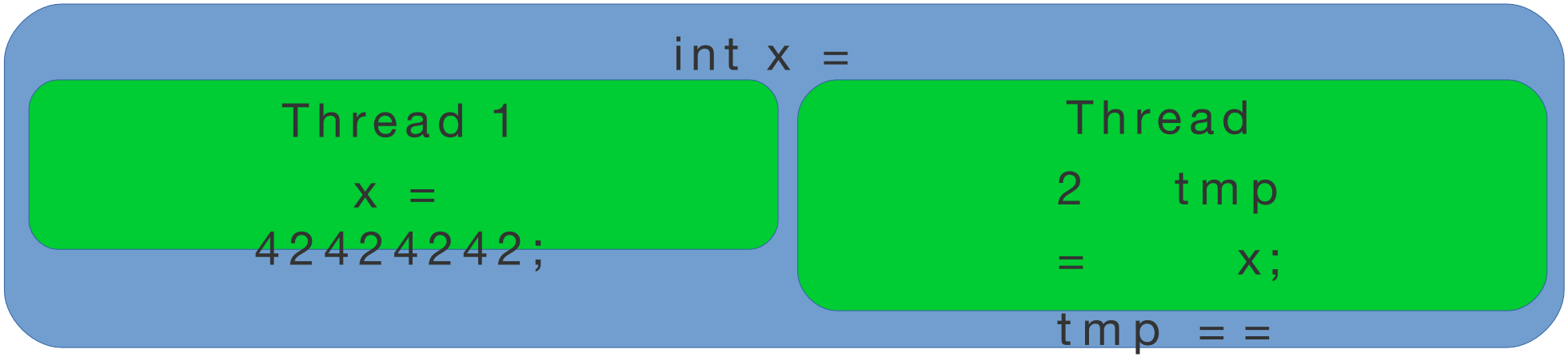
What's really going on?



What's really going on?



More insidious atomic operation example



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++?

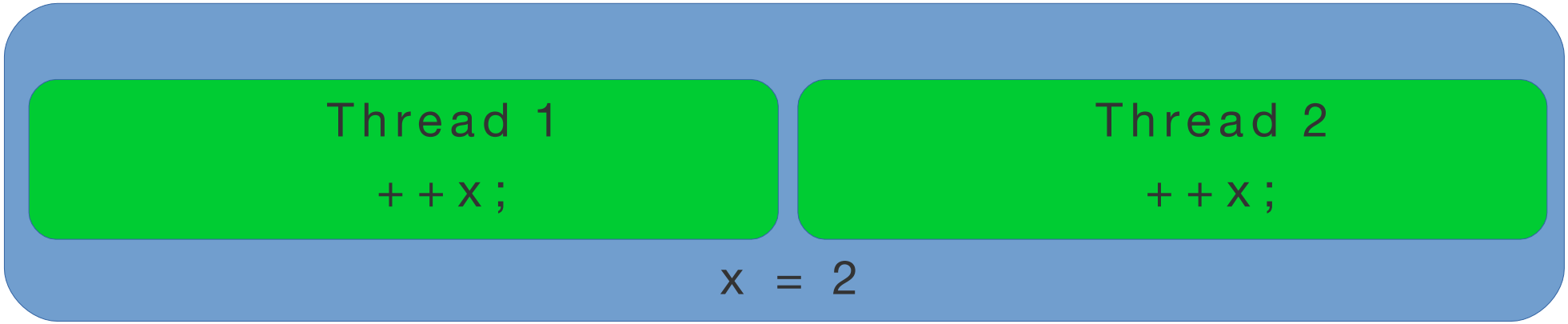
Data sharing in C++

C++11: std::atomic

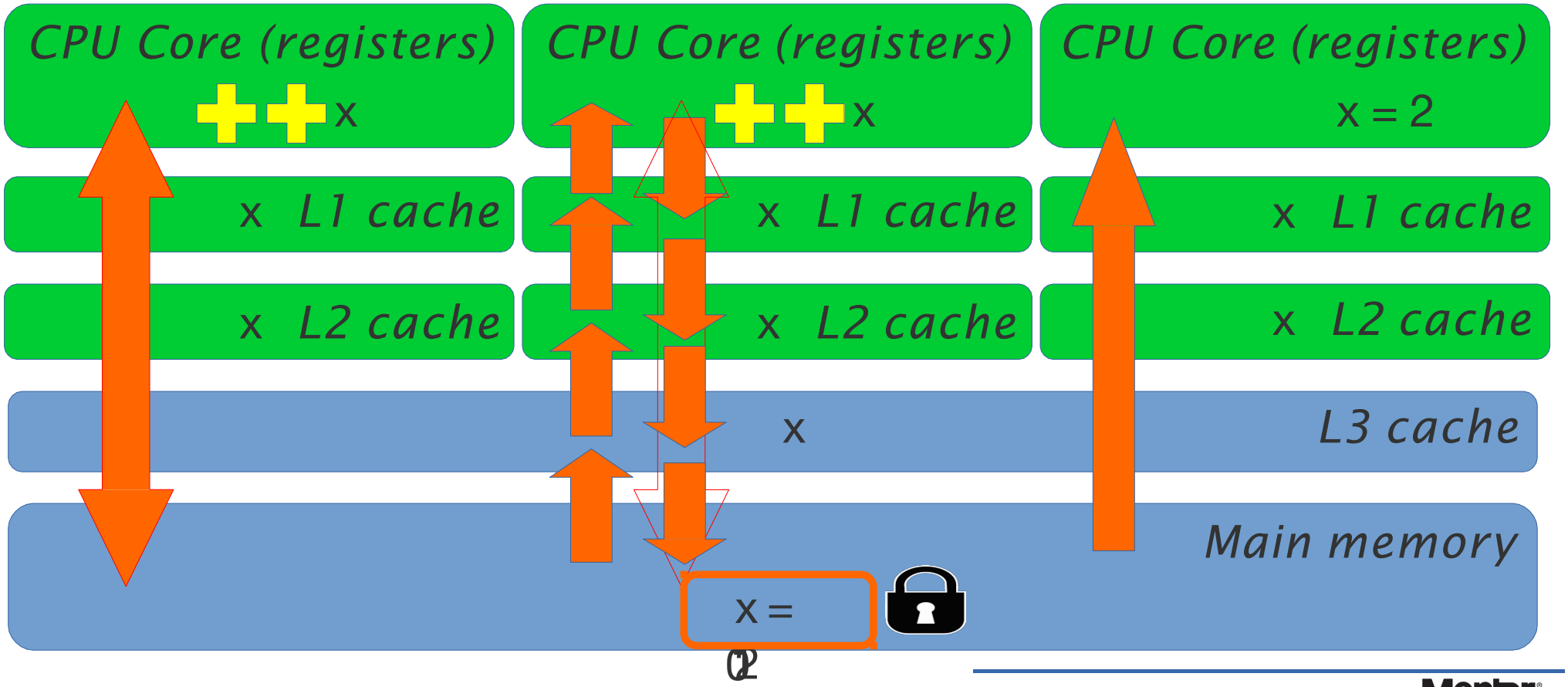
```
#include <atomic>
std::atomic<int> x(0);          // NOT std::atomic<int> x=0;
```

++x is now atomic!

- another thread cannot access during increment



What's really going on now?



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!
```

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;
```

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

not atomic

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):

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

T& y

?

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)) {}
- 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)) {}

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

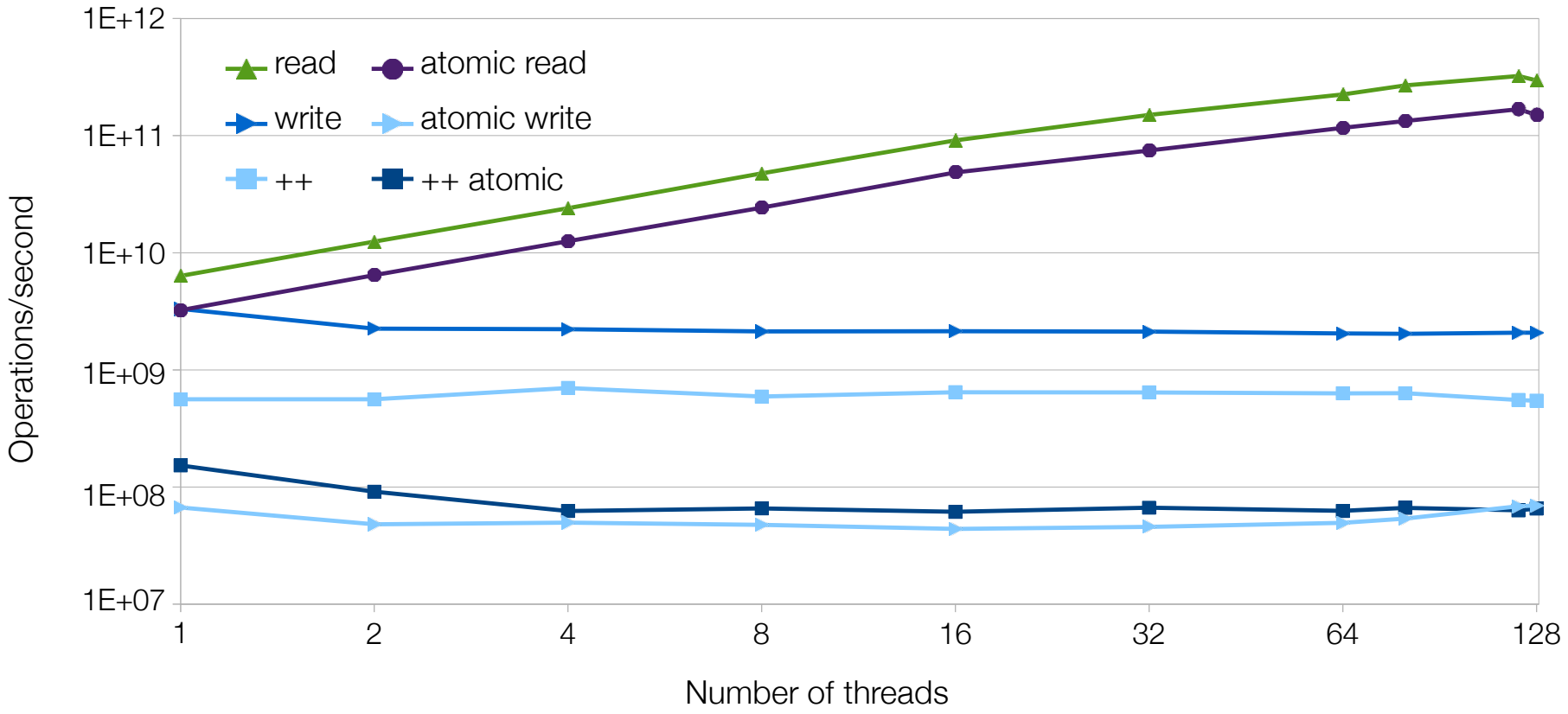
- Including load() and store() instead of operator=()

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)

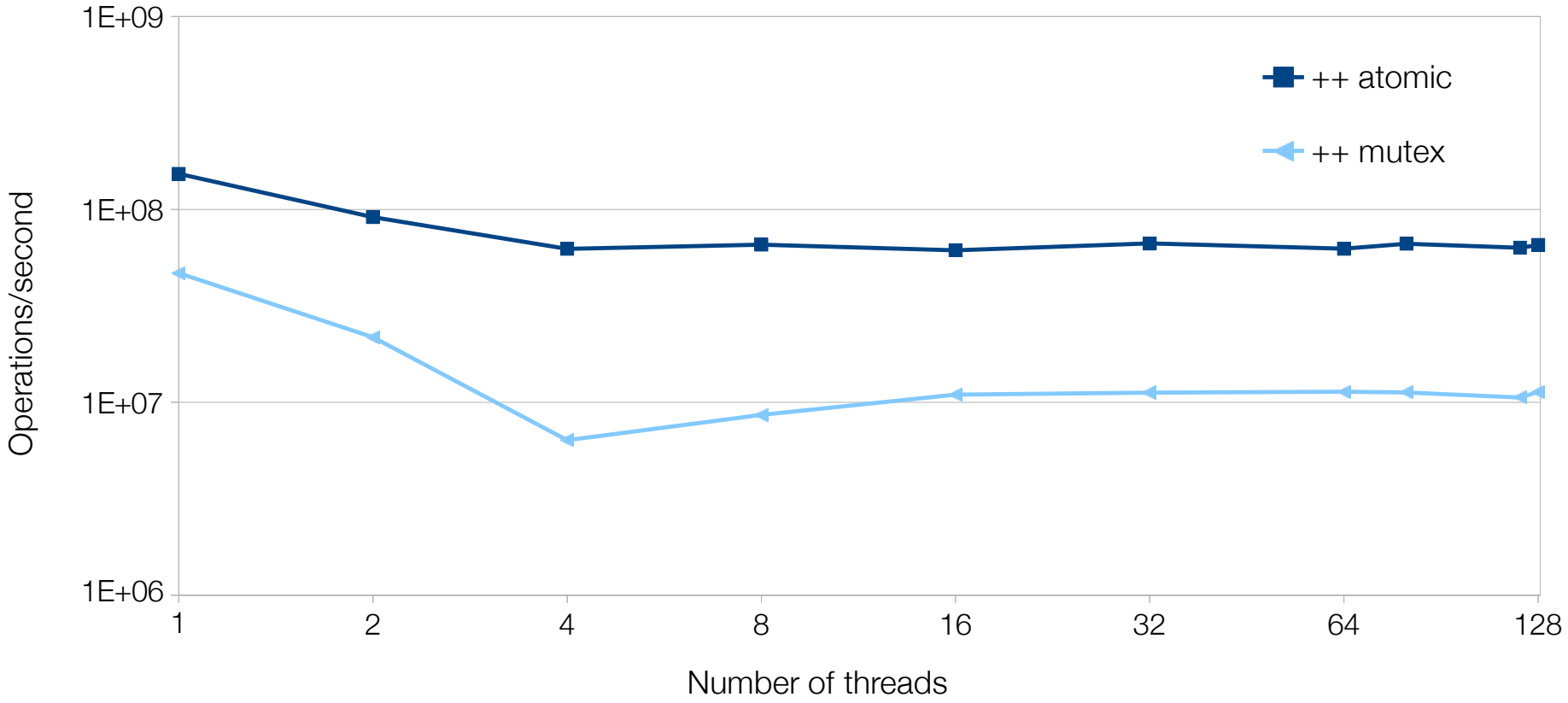
How fast are atomic operations?

Are atomic operations slower than non-atomic?



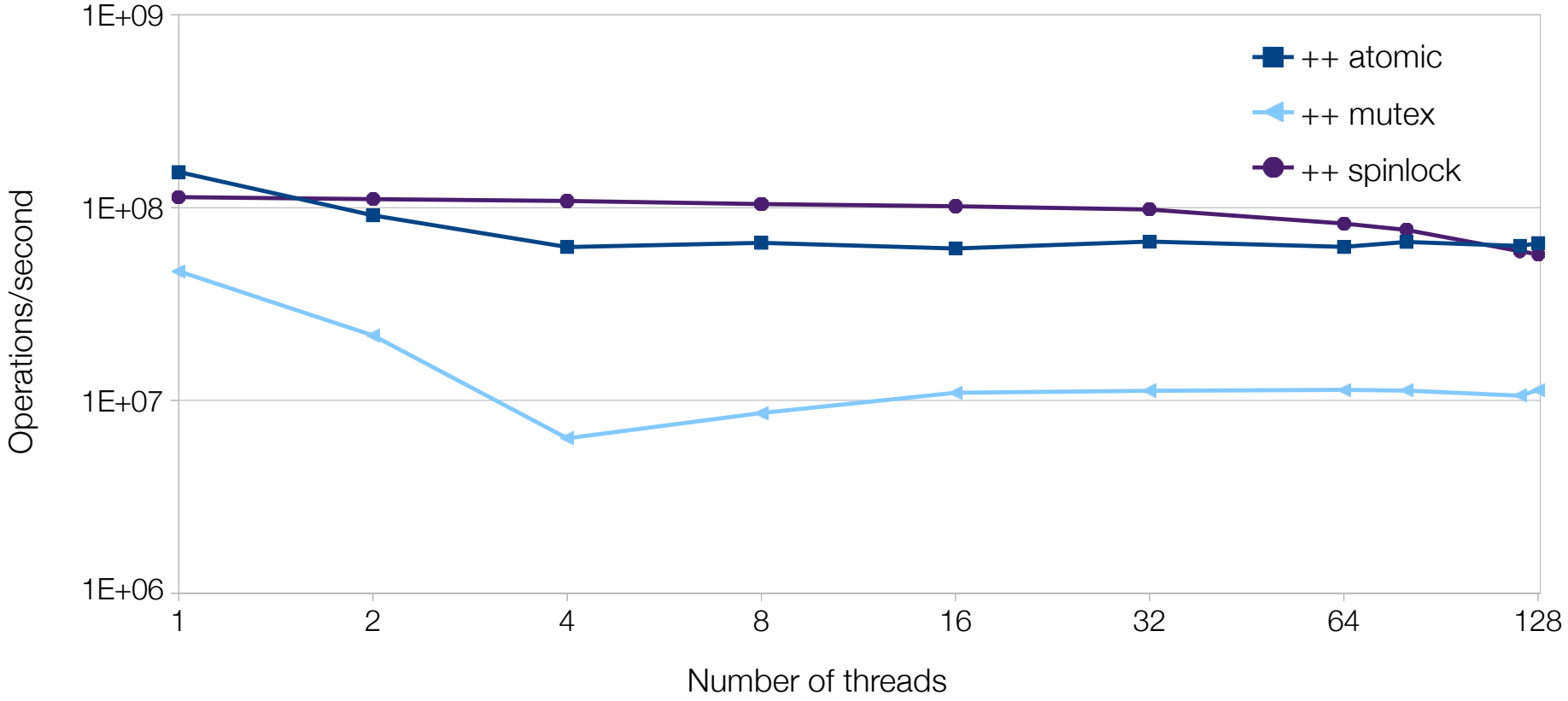
Are atomic operations faster than locks?

Are atomic operations faster than locks?

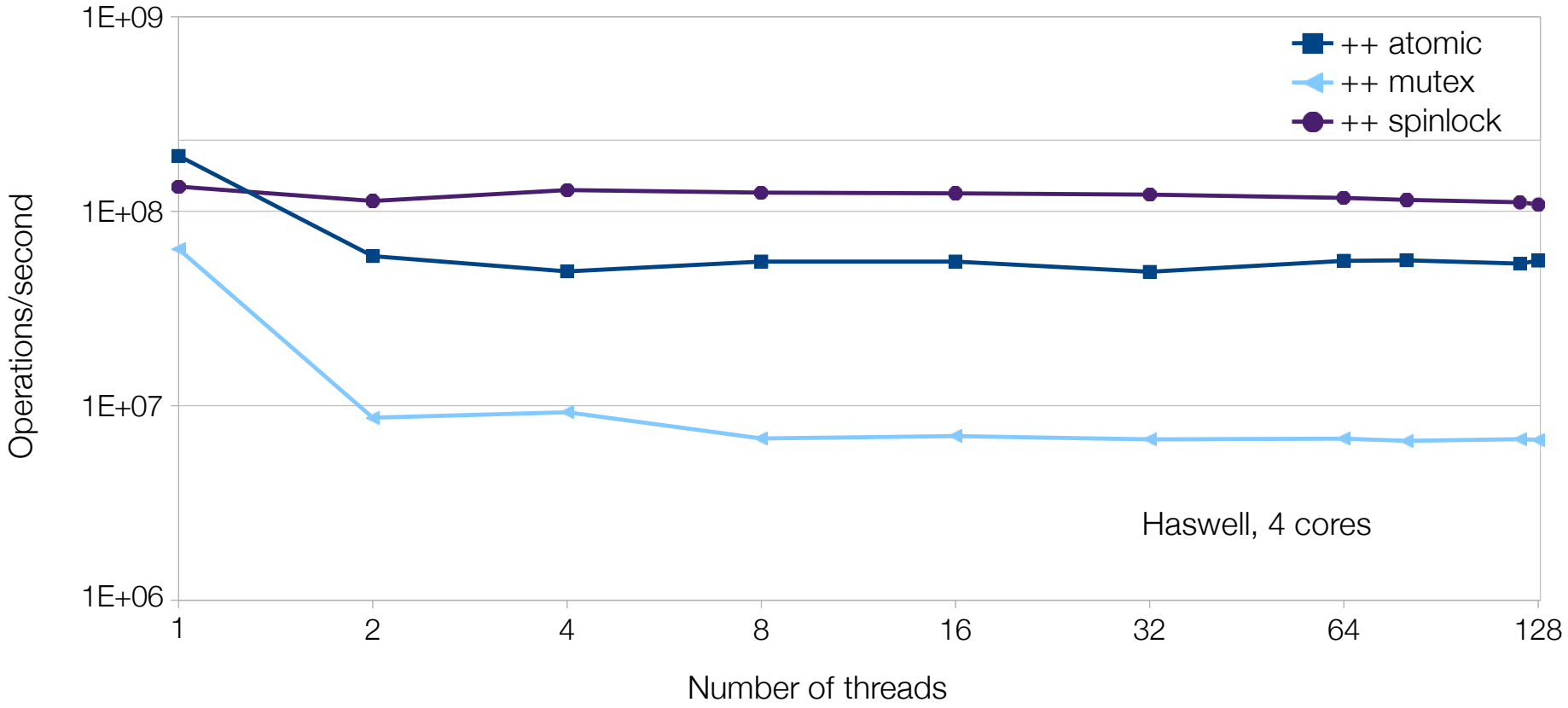


Are atomic operations faster than locks?

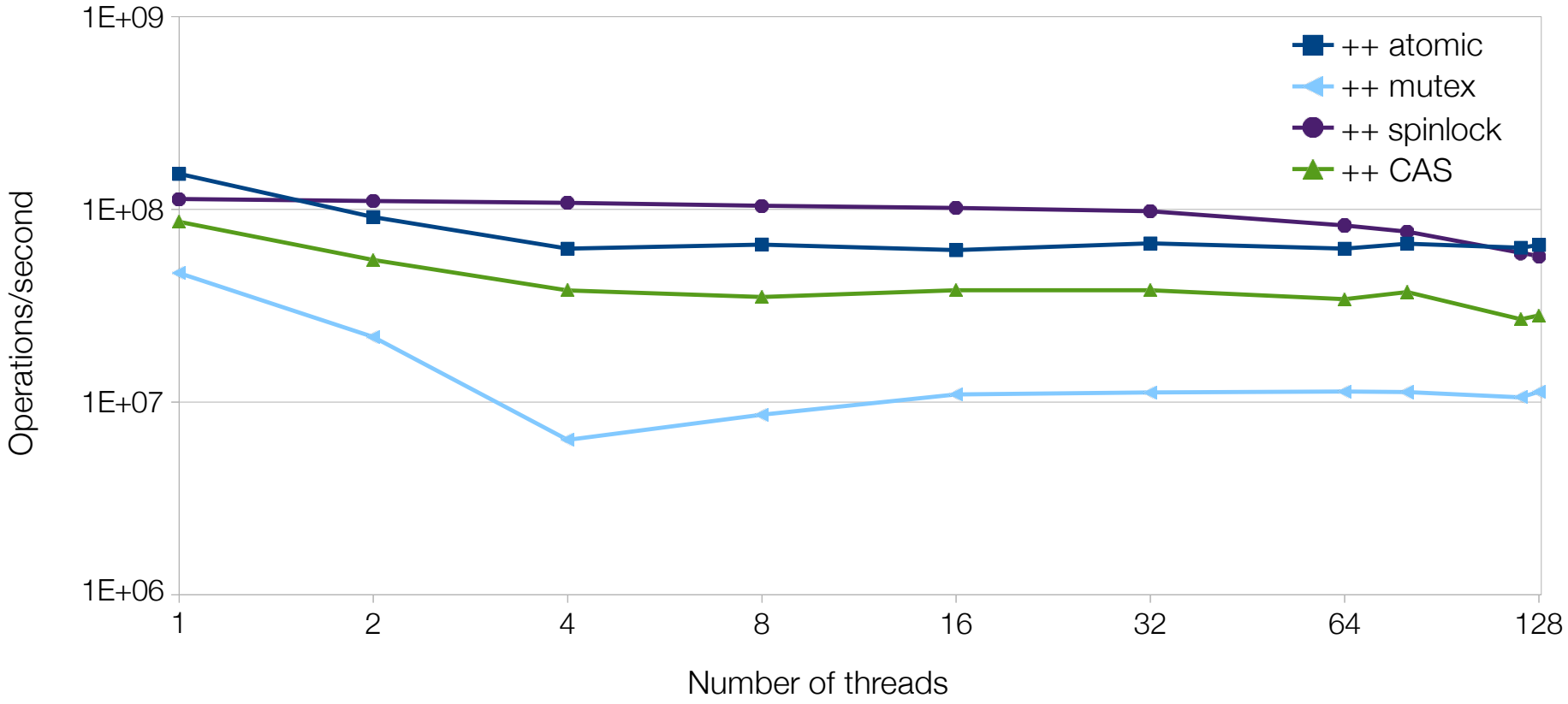
which



Are atomic operations faster than locks?



Remember CAS?



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; };