Lightning-Fast IPC through Lock-Free Shared Memory

Daniel Fox Franke <dfoxfranke@gmail.com>

Purveyors of Fine Gadgets

June 2024

Intro quiz

Global

```
int x = 1;
```

Thread 1

```
x = 2;
```

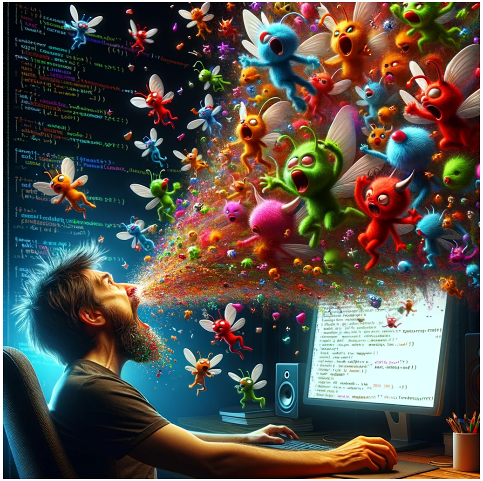
Thread 2

```
int a, b;
a = x;
b = x;
printf("%d %d",a,b);
```

What can this print?

- 1 1?
- 1 2?
- 2 2?
- 2 1?
- Something else?

Answer: all of the above



C17 §5.1.2.4.35:

“The execution of a program contains a *data race* if it contains two **conflicting actions** in different threads, at least one of which is not **atomic**, and neither **happens before** the other. Any such data race results in undefined behavior.”

Avoiding UB with atomics

Global

```
#include <stdatomic.h>
atomic_int x;
atomic_init(&x, 1);
```

Thread 1

```
atomic_store(&x, 2);
```

Thread 2

```
int a, b;
a = atomic_load(&x);
b = atomic_load(&x);
printf("%d %d",a,b);
```

What can this print?

- ✓ 1 1
- ✓ 1 2
- ✓ 2 2
- ✗ 2 1
- ✗ Something else

Same thing in Rust

Global

```
use std::sync::atomic::{AtomicI32,Ordering::SeqCst};
static const x : AtomicI32 = AtomicI32::new(1);
```

Thread 1

```
x.store(2, SeqCst);
```

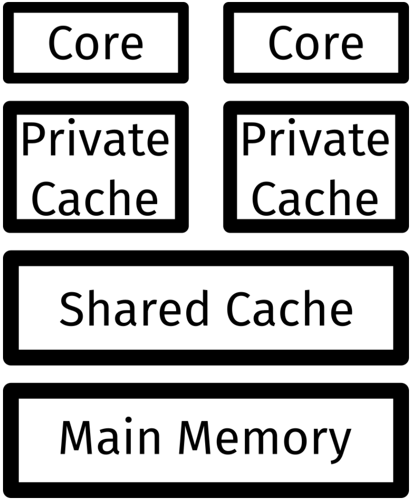
Thread 2

```
let a = x.load(SeqCst);
let b = x.load(SeqCst);
print!("{a} {b}");
```

What can this print?

- ✓ 1 1
- ✓ 1 2
- ✓ 2 2
- ✗ 2 1
- ✗ Something else

Cache hierarchy



The ideal: write-back LRU cache

On read:

1. Is the word already in cache? Return it and done.
2. Otherwise, store least recently used word to parent cache.
3. Load from parent cache into space just freed up, then return it.

On write:

1. Is the word already in cache? Overwrite it and done.
2. Otherwise, store least recently used word to parent cache.
3. Write to space just freed up. Do *not* immediately forward this new value to parent.

The reality: *N*-way associative cache

Big LRUs are expensive to implement in hardware. It's cheaper (lower gate count, more energy-efficient) to have lots of little ones. This leads to some loss of performance, but usually negligible.

Example: 4 MiB 8-way associative cache w/ 64-byte cache lines.

- 4 MiB = 2^{16} 64-byte lines.
- Cache has 2^{13} LRUs that hold 8 lines each.
- Only one LRU can be used for any given address. Bits 18:6 of address determine which one.

Cache coherence: problem statement

Write-back caching + multiple cores w/private cache + shared memory = chaos

To keep our sanity, we need:

1. **Change propagation**: Writes by one core become visible to all other cores in a *reasonably* timely manner.
2. **Sequential consistency**: Writes to any *single* memory location happen in a globally consistent order.

Single-location sequential consistency

Global

```
#define store_relaxed(p, v) atomic_store_explicit( \
    p, v, memory_order_relaxed);
#define load_relaxed(p) atomic_load_explicit( \
    p, memory_order_relaxed);
atomic_int x;
atomic_init(&x, 0);
```

Thread 1

```
store_relaxed(&x, 1);
```

Thread 2

```
store_relaxed(&x, 2);
```

Threads 3 & 4

```
int a, b;
a = load_relaxed(&x);
b = load_relaxed(&x);
printf("%d %d",a,b);
```

Could print:

- ✓ any of seven possibilities, including...
- ✓ `1 2`
- ✓ `2 1`

But **never these two at the same time**!

Sequential *in*consistency with multiple addresses

Global

```
#define store_relaxed(p, v) atomic_store_explicit( \
    p, v, memory_order_relaxed);
#define load_relaxed(p) atomic_load_explicit( \
    p, memory_order_relaxed);
atomic_int x, y;
atomic_init(&x, 1);
atomic_init(&y, 1);
```

Thread 1

```
store_relaxed(&x, 2);
store_relaxed(&y, 2);
```

Threads 2 & 3

```
int b = load_relaxed(&y);
int a = load_relaxed(&x);
printf("%d %d",a,b);
```

Could print:

- ✓ 1 1
- ✓ 2 2
- ✓ **1 2**
- ✓ **2 1**

Simultaneously!

Cache coherence protocols

- Simplest possible solution: write-through instead of write-back
- More bandwidth-efficient: MESI protocol. Snoop on other cores' memory accesses and maintain state per cache line:
  - **Modified** - Cache line dirty (a write is pending). All other cores in invalid state for this address.
  - **Exclusive** - Cache line clean. All other cores in invalid state for this address.
  - **Shared** - Cache line clean. Other cores may also be in shared state for this address.
  - **Invalid** - Cache line treated as empty; must load from parent before use.

  State machine transitions happen on issuance of load/store instructions, or on observation of bus activity.

  Exercise: draw this state machine on your own.

CPU support for concurrency: fences

Most architectures have some equivalent to these instructions:

- **Load fence**: all **loads** before the fence instruction execute before any **loads/stores** after it.
- **Store fence**: all **loads/stores** before the fence instruction execute before any **stores** after it.
- **Full fence**: all **loads/stores** before the fence instruction execute before any **loads/stores** after it.

A full fence is not just two half fences! Only a full fence prevents earlier stores from being reordered with later loads.

CPU support for concurrency: load-acquire/store-release

- **load-acquire**: A load that won't be reordered past later instructions.
- **store-release**: A store that won't be reordered past earlier instructions.

These are weaker (and therefore more efficient) than fences: they only affect their own ordering and not that of earlier loads or later stores.

CPU support for concurrency: atomic compare-and-set

Most architectures have an instruction that does this, atomically:

```
bool atomic_cas(intptr_t* addr, intptr_t old, intptr_t new) {
    if (*addr == old) {
        *addr = new;
        return true;
    } else {
        return false;
    }
}
```

From atomic CAS to atomic anything

```
intptr_t atomically(intptr_t *addr, intptr_t (*f)(intptr_t)) {
    do {
        intptr_t old = atomic_load(addr);
        intptr_t new = f(old);
    } while (!atomic_cas(addr, old, new));
    return new;
}
```

Very rough latency intuition

- Register $\leftrightarrow$ register: 0.25ns
- Register $\leftrightarrow$ private cache: 1–4ns
- Private cache $\rightarrow$ private cache: 6ns
- Register $\leftrightarrow$ shared cache: 25ns
- Register $\leftrightarrow$ main memory: 100ns

Acquire/release and fences don’t add communication, but they add interdependency between communications. Later instructions can’t start until their dependencies have finished.

The C++ Memory Model

- Abstracts away architecture-specific details about atomicity and ordering.
- Originally for defining `std::atomic` in C++11.
- Backported to C (since C11) and D.
- Adopted by Rust, Zig, probably others.

Combining and reordering non-atomic operations

Source

```c
void foo(int *restrict a, int *restrict b) {
  *a += 1;    *b += 1;    *a += 1; //Tight slide, excuse the formatting
}
```

x86_64

```
inc  dword ptr [rsi]
mov  eax, dword ptr [rdi]
add  eax, 2
mov  dword ptr [rdi], eax
```

aarch64

```
ldr  w8, [x1]
ldr  w9, [x0]
add  w8, w8, #1
add  w9, w9, #2
str  w8, [x1]
str  w9, [x0]
```

riscv64

```
lw      a2, 0(a1)
lw      a3, 0(a0)
addiw   a2, a2, 1
sw      a2, 0(a1)
addiw   a3, a3, 2
sw      a3, 0(a0)
```

Atomic operations and memory orders

Types of atomic operation:

- Load (L)
- Store (S)
- Atomic Read-Modify-Write (RMW)

RMW = atomically mutate a memory location, *e.g.*, atomic increment. Can be built on CAS.

Memory orders and (applicability):

- Relaxed (L, S, RMW)
- Acquire (L, RMW)
- Release (S, RMW)
- Acq+Rel (RMW)
- SeqCst (L, S, RMW)

Memory orderings: basic overview

- **Relaxed**: Still atomic, but no memory ordering guarantees beyond cache coherence. Generates ordinary load/store instructions.
- **Acquire**: Generates load-acquire instructions instead of regular loads, therefore won't be reordered with later instructions.
- **Release**: Generates store-release instructions instead of regular stores, therefore won't be reordered with earlier instructions.
- **AcqRel**: combines acquire semantics with release semantics when implementing RMWs.
- **SeqCst**: Makes the operation sequentially consistent with all other SeqCst operations (only), even when multiple memory locations are involved. Usually implemented using full fences.

To keep these slides readable…

```
<operation>_<order>(...)
```

is a macro for

```
atomic_<operation>_explicit(
    ...,
    memory_order_<order>
);
```

For everything defined in `<stdatomic.h>`.

Sequential *in*consistency with multiple addresses

Global

```
atomic_int x, y;
atomic_init(&x, 1);
atomic_init(&y, 1);
```

Thread 1

```
store_relaxed(&x, 2);
store_relaxed(&y, 2);
```

Threads 2 & 3

```
int b = load_relaxed(&y);
int a = load_relaxed(&x);
printf("%d %d",a,b);
```

Could print:

- ✓ 1 1
- ✓ 2 2
- ✓ **1 2**
- ✓ **2 1**

Simultaneously!

Sequential consistency with multiple addresses

Global

```
atomic_int x, y;
atomic_init(&x, 1);
atomic_init(&y, 1);
```

Thread 1

```
store_seq_cst(&x, 2);
```

Thread 2

```
store_seq_cst(&y, 2);
```

Threads 3 & 4

```
int a = load_seq_cst(&x);
int b = load_seq_cst(&y);
printf("%d %d",a,b);
```

Could print:

- ✓ 1 1
- ✓ 2 2
- ✓ **1 2**
- ✓ **2 1**

But **impossible to get** ***both*** 1 2 and 2 1.

The “happens before” relation

A **strict partial ordering** of events:

- Irreflexive: `!(A < A)`
- Asymmetric: `(A < B)` implies `!(B < A)`
- Transitive: `(A < B && B < C)` implies `(A < C)`

In general, **no total ordering exists**, not even an unknown one. Some events are incomparable.

Axioms of happens-before

A happens before B when:

- A is **sequenced before** B in the same thread, according to ordinary language semantics, even if they're non-atomic.
- A has release ordering, B has acquire ordering, and the operations **synchronize with** each other. A synchronizes with B when the value read by B belongs to a **release sequence** headed by A.

Even if they’re non-atomic? But wait, what about this?

Source

```c
void foo(int *restrict a, int *restrict b) {
  *a += 1;    *b += 1;    *a += 1;
}
```

x86_64

```
inc  dword ptr [rsi]
mov  eax, dword ptr [rdi]
add  eax, 2
mov  dword ptr [rdi], eax
```

aarch64

```
ldr  w8, [x1]
ldr  w9, [x0]
add  w8, w8, #1
add  w9, w9, #2
str  w8, [x1]
str  w9, [x0]
```

riscv64

```
lw      a2, 0(a1)
lw      a3, 0(a0)
addiw   a2, a2, 1
sw      a2, 0(a1)
addiw   a3, a3, 2
sw      a3, 0(a0)
```

Observability

- These reordering optimizations are legal only if they're *inobservable*.
- If no other thread is concurrently accessing `*a` or `*b`, there's no observable change in program behavior.
- If some other thread *is* concurrently accessing them, that's a data race. Data races are UB, so the optimizer can ignore the possibility.

Release sequences

Global

```
atomic_int x;
atomic_init(&x, 0);
```

Thread A

```
store_release(&x, 1);
store_relaxed(&x, 2);
fetch_add_relaxed(&x, 1);
store_relaxed(&x, 6);
```

Thread B

```
fetch_add_release(&x, 1);
store_relaxed(&x, 4);
```

Release sequences begin with a releasing store or RMW, continue with:

- A store by the releasing thread
- An RMW by any thread

Suppose our modification order is 1 2 3 4 5 6. What are the release sequences?

Release sequences

Thread A

```
store_release(&x, 1);
store_relaxed(&x, 2);
fetch_add_relaxed(&x, 1);
store_relaxed(&x, 6);
```

Thread B

```
fetch_add_release(&x, 1);
store_relaxed(&x, 4);
```

Release sequences begin with a releasing store or RMW, continue with:

- A store by the releasing thread
- An RMW by any thread

Suppose our modification order is 1 2 3 4 5 6. What are the release sequences?

Answer: 1 2 3 and 3 4 5.

- 2: same thread as 1, continues 1's sequence.
- 3: RMW continues 1's sequence, release starts a new one.
- 4: different thread from 1, breaks that sequence but continues 3.
- 5: RMW continues 3's sequence.
- 6: different thread from 3, breaks that sequence.

A simple spinlock

```
struct some_big_global_state foo;
atomic_flag foo_lock;

void lock(atomic_flag *lck) {
    while(!flag_test_and_set_acquire(lck));
}

void unlock(atomic_flag *lck) {
    flag_clear_release(lck);
}

void safely_frobnicate_foo() {
    lock(&foo_lock);
    frobnicate_nonatomically(&foo);
    unlock(&foo_lock);
}
```

Spinlock correctness

Thread 1..N

```
while(!flag_test_and_set_acquire(&foo_lock)); // 1
frobnicate_nonatomically(&foo);              // 2
flag_clear_release(&foo_lock);               // 3
```

1. Locking must happen before frobnicating.
2. Frobnicating must happen before unlocking.
3. Unlocking must happen before other threads lock.

- Reqs 1&2 satisfied by sequencing.
- Req 3: release on line 3 heads release sequence on `&foo_lock`; acquiring RMW on line 1 synchronizes w/ it, continuing only if it reads the 0 that a release wrote; extend by induction to any thread pair.

Fences: synchronizing without an address

Global

```
atomic_int x, y;
atomic_init(&x, 0);
atomic_init(&y, 0);
```

Thread 1

```
store_relaxed(&x, 1);    //1
thread_fence_release(); //2
store_relaxed(&y, 1);    //3
```

Thread 2

```
int a, b;
b = load_relaxed(&y);    //4
thread_fence_acquire(); //5
a = load_relaxed(&x);    //6
printf("%d %d", a, b);
```

Fences synchronize on *any* address written after a release fence and read before an acquire fence. These fences synchronize on *y*.

If $b = 1$ then $2 < 5$.

Fences: synchronizing without an address

Global

```
atomic_int x, y;
atomic_init(&x, 0);
atomic_init(&y, 0);
```

Thread 1

```
store_relaxed(&x, 1);    //1
thread_fence_release(); //2
store_relaxed(&y, 1);    //3
```

Thread 2

```
int a, b;
b = load_relaxed(&y);    //4
thread_fence_acquire(); //5
a = load_relaxed(&x);    //6
printf("%d %d", a, b);
```

Assuming $b = 1$:

- $2 < 5$
- By sequencing:
  - $1 < 2 < 3$
  - $4 < 5 < 6$
- By transitivity: $1 < 2 < 5 < 6$.
- Therefore, since $1 < 6$, $a = 1$.

We conclude that printing `0 1` is impossible.

Happens-before coheres with modification order

Four coherence properties:

- read/read
- read/write
- write/read
- write/write

If *A* happens before *B*, both act on a memory location *M*, *A* is a read (resp. write), and *B* is a read (resp. write), then the value read (resp. written) by *A* comes not later in *M*'s modification order than the value read (resp. written) by *B*.

`mmap(2)`

Map some virtual memory to

- Physical memory
- A file (on disk or on a tmpfs)
- Even device registers! (Linux `uio` facility)

Underlies

- `malloc(3)`
- `shm_open(3)` (semi-obsolete thanks to mmap)
- `brk(2)/sbrk(2)` (very obsolete)

Address translation example: RISC-V

- Page size: 2^{12} bytes = 4 KiB.
- Physical address: 56 bits = 64 PiB addressable = 2^{44} pages.
- Virtual address: 9 + 9 + 9 + 9 + 9 + 12 = 57 bits, sign-extended to 64.
- Page table entry: 64 bits — 44-bit page number + 20 bits for flags or reserved. 2^9 entries fill one page.
- Top 9 bits of virtual address index into top-level page table, gives page number of next-level table.
- Repeat for each of 5 levels.
- 12-bit page offset untranslated (same in physical address as virtual).

Coping with SIGBUS

`SIGBUS`:

- Virtual address points to non-existent physical address, or past the end of a file
- Invalid address alignment
- Hardware error (bad sector, machine check exception)

(*c.f.* `SIGSEGV`: address not mapped at all, or without permissions for the operation)

In C:

- Call `setjmp()` before any access which might raise `SIGBUS`.
- Stash a pointer to the `jmp_buf` in thread-local storage.
- Recover by `longjmp()`ing out of the signal handler.
- Use `atomic_signal_fence()` as needed.

In Rust:

- Just use my hw`-exception` crate.

`memfd_create(2)`

- Returns an fd for an “anonymous file”.
- Pass the fd over a UNIX domain socket using `SCM_RIGHTS` (see `unix(7)`) to share it.
- Memfds can be “sealed” to prevent shrinking — no more worries about SIGBUS.

`futex(2)`

Spinlocks in userspace are rude! Use futex to schedule threads appropriately.

`futex(FUTEX_WAIT, addr, val, timeout):`

- Assert that `*addr` has value `val`.
- If not, return immediately.
- Otherwise, go to sleep for `timeout` or until another thread wakes us.

`futex(FUTEX_WAKE, addr, val):`

- Wake other threads, up to `val` of them, that are waiting on `addr`.

Lock-free algorithms

Standard terms:

- **Obstruction-free**: For all threads *t*, if every thread except *t* is suspended, *t* completes its operation in bounded time. (Livelock and starvation possible)
- **Lock-free**: There exists a thread which will complete its operation in bounded time, regardless of other suspensions. (Livelock impossible, starvation possible)
- **Wait-free**: All non-suspended threads will complete their operation in bounded time, regardless of other suspensions. (Livelock and starvation impossible)

My term: quasi-wait-free

- Assume distinct reader/writer roles.
- Suspended writers can still obstruct other threads…
- …but no thread ever waits on a reader.

Sequence locks: a quasi-wait-free “blackboard”

```
struct seqlocked_state {
    atomic_uint version;
    struct state a_big_state_structure;
}
```

One Writer

1. Increment version field even → odd.
2. Update state.
3. Increment version field odd → even.
4. `FUTEX_WAKE` on version field.

***N* Readers**

1. Read version field.
2. Read state.
3. Read version field again.
4. If versions match and are odd, `FUTEX_WAIT`.
5. Loop until versions match and are even.

Sequence locks: a buggy attempt

Writer

1. Store-release odd version
2. Store-relaxed each word of state
3. Store-release even version

Reader

4. Load-acquire version
5. Load-relaxed each word of state
6. Load-acquire version

Problem: CPU can reorder steps 1/2, and 5/6.

- Suppose writer runs just once; version goes 0 $\rightarrow$ 1 $\rightarrow$ 2.
- If step 4 reads `version=2`, then 3 happens before 4; we're good.
- But if steps 4&6 read `version=0`, no interthread happens-before relation is proven. Step 2 could be intermixed with 5.

Sequence locks: corrected

Writer

1. Store-relaxed odd version
2. Store-fence
3. Store-relaxed each word of state
4. Store-release even version

Reader

5. Load-acquire version
6. Load-relaxed each word of state
7. Load-fence
8. Load-relaxed version

- Store-fence prevents CPU reordering of 1/3; load-fence prevents reordering of 6/8.
- If any result of 3/4 is seen by 5/6, then 2 happens before 7, so 1 happens before 8.
- If 8 reads `version=0`, reason by modus tollens: 1 did not happen before 8, so 6 does not read anything written by 3.

A lock-free blackboard

```
struct blackboard {
    atomic_uint version;
    struct state a_big_state_structure[N]; // N>=2
}
```

One Writer

1. Store-fence
2. Let `i = (version + 1) % N`.
3. Store-relaxed *i*'th copy of state.
4. Store-release increment to version field.

N Readers

1. Load-acquire version field.
2. Load-relaxed `(version % N)`'th copy of state.
3. Load-fence
4. Load-relaxed version field again.
5. Loop until version increased by less than N.

Dealing with counter overflow

For k-bit version counter, we must assume fewer than 2^k writes occur while any single read is in progress. But in long-running systems, we must be robust against eventual overflow.

```
bool version_is_good(unsigned v1, unsigned v2) {
    unsigned vmin = v1;
    unsigned vmax = v1 + N; // OK to wrap

    if (LIKELY(vmax > vmin)) {
        //Normal case
        return v2 >= vmin && v2 < vmax;
    } else {
        //Overflow case
        return v2 >= vmin || v2 < vmax;
    }
}
```

Copying state: the safe, portable, slow way

In shared memory

```
struct shared_state {
    atomic_uint version;
    atomic_int field1;
    atomic_char field2;
    atomic_char field3;
    //...
}
```

In private memory

```
struct state {
    unsigned int version;
    int field1;
    char field2;
    char field3;
    //...
}
```

- Copy each field*N* one-by-one using `load_relaxed/store_relaxed`.
- Maximally portable, but will generate inefficient code.
- Annoying for library APIs: can't be generic over type of state.

Copying state: the fast, slightly dodgy way

```
void *racy_memcpy(void *restrict dest, const void *restrict src, size_t n) {
    asm volatile(""); // Prevent optimizer from proving data race possibility
    void *ret = memcpy(dest, src, n); // Full speed ahead, damn the torpedoes!
    asm volatile("");
    return ret;
}
```

- `asm volatile` “black box” could stand in for lock acquisition/release, so no data race ever provable at compile time.
- Still UB in a real sense: architecture-dependent assumption that instructions used by memcpy don’t side-effect if a data race happens.

Q&A + further reading + contact

Primary sources:

- C17 standard: §5.1.2.4 (memory model) and §7.17 (`<stdatomic.h>`)
- Rust library reference: `https://doc.rust-lang.org/stable/std/sync/atomic`
- Your CPU's architecture manual
- Linux manpages: `mmap(2)`, `memfd_create(2)`, `futex(2)`, `unix(7)`

Secondary sources:

- C++ memory model: `https://en.cppreference.com/w/cpp/atomic/memory_order`
- *Rust Atomics and Locks* by Mara Bos: `https://marabos.nl/atomics/`

These slides: `https://github.com/dfoxfranke/self2024`
Email: `<dfoxfranke@gmail.com>`

Appendix: What about `volatile`?

In theory:

- `volatile` has nothing to do with concurrency!
- Tells the compiler to treat load and stores as having side effects, so it won't reorder or combine the instructions.
- But the processor still can!
- Not atomic; does not prevent data races or UB.

In practice:

- Probably won't literally cause nasal demons.
- Adequately-aligned volatile pointers usually behave like relaxed atomics.
- On MSVC (only), volatile loads have acquire ordering and volatile stores have release ordering.
- But seriously, **just use real atomics**.

Appendix: Go sucks at concurrency!

- Channels: Go’s favorite and only hammer
  - Real “share nothing” $\rightarrow$ never send a pointer over a channel $\rightarrow$ tons of unnecessary copying.
  - Sharing pointers $\rightarrow$ data races possible $\rightarrow$ not memory-safe any more. *E.g.* conflicting access to maps = heap corruption.
  - “CSP-inspired!” — Someone who’s never read the CSP papers.
  - Looks more like π-calculus, but buffering and channel-closure semantics are kludgy.
- Go atomics: `SeqCst` is the only supported memory ordering. Order-of-magnitude performance penalty.
- Can escape into C and do low-level concurrency there, but Go FFI has high overhead: hundreds of nanoseconds per call into C. Have to do enough per call to make it worth it. Even Python doesn’t have this problem!