Lab 1: Synchronization Basics post-mortem

CS378 Fall 2023

Chris Rossbach

Outline

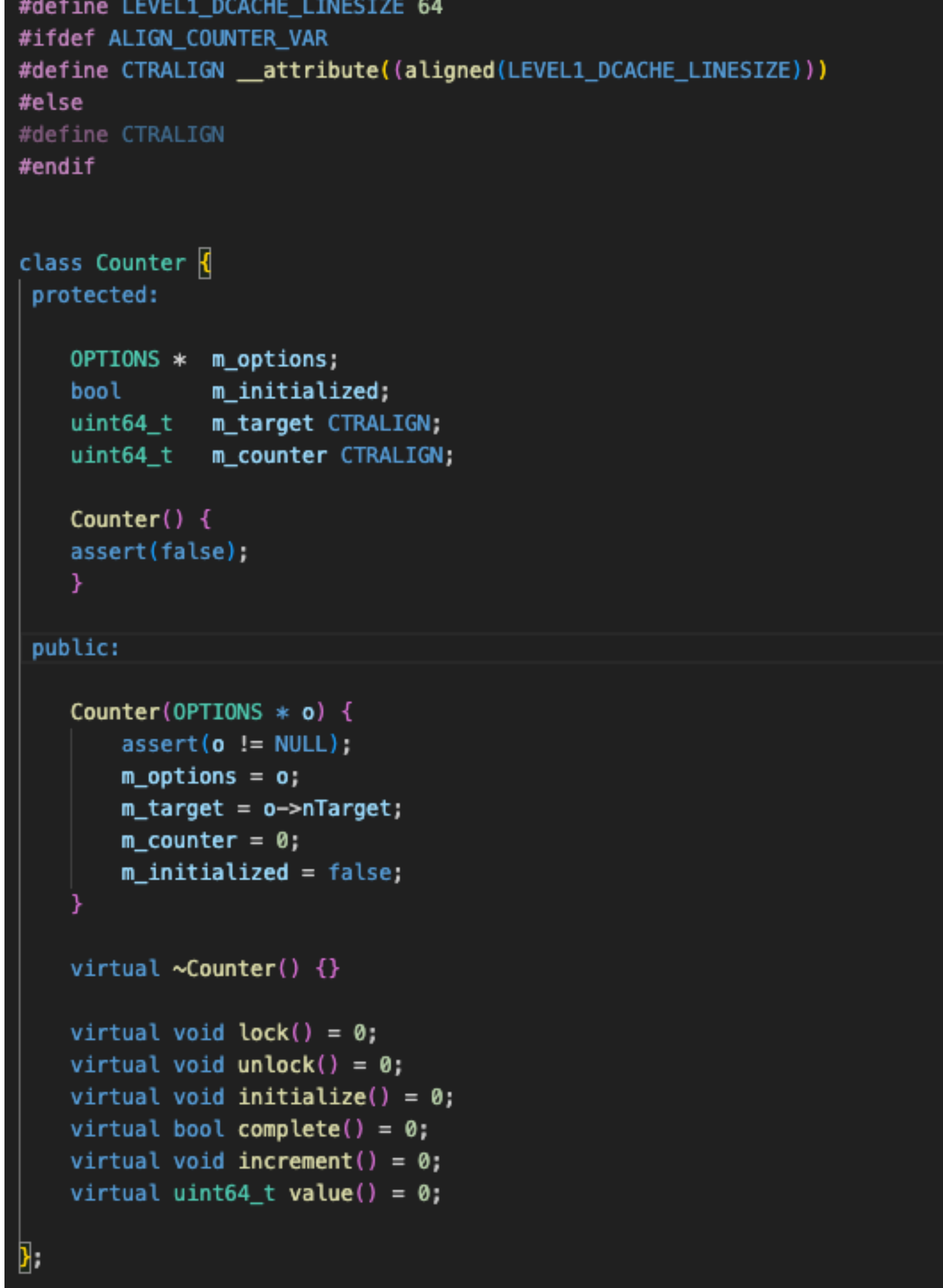
- Some tips and tricks
    - Overview of how I coded it
    - Overview of my approach to scripting
- Analyzing the Data
    - Step 1
    - Step 2
    - Step 3
    - Step 4
- Discussion

Code Structure

- Guiding Principles
  - “DRY” code
  - Maintain readability in the face of many options
  - Avoid performance impact from instrumentation
  - Make it easy to make graphs and write report
    - Produce output that is easy to script and graph
    - This is a critical skill for empirical method

Shared logic/instrumentation from OOP

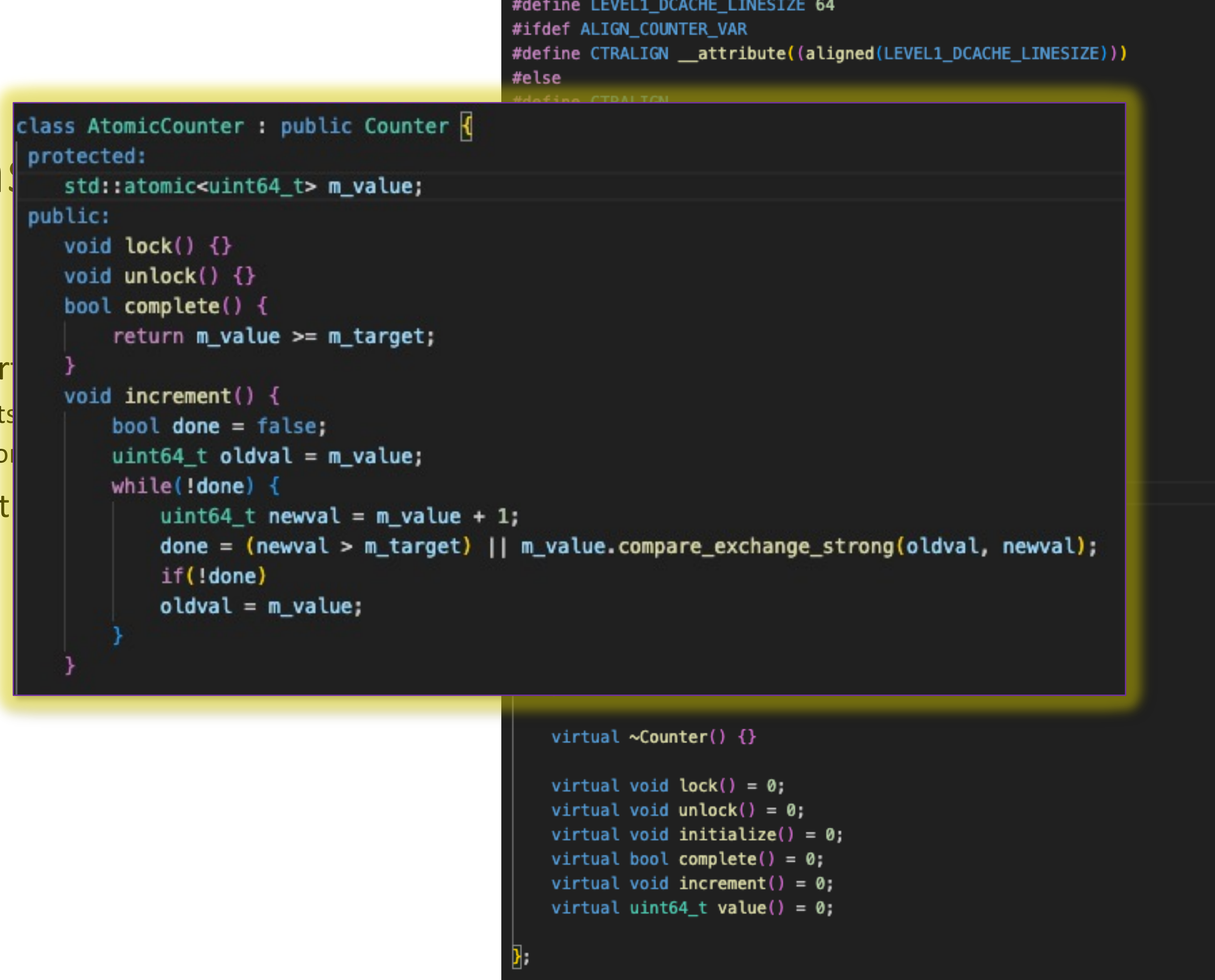
- Use a super-class with virtual methods
  - Each sub-class implements virtual methods differently
  - Maximizes instrumentation reuse
- Note the alignment __attribute
  - What does it do?
  - Why is it important?



```
#define LEVEL1_DCACHE_LINESIZE 64
#ifdef ALIGN_COUNTER_VAR
#define CTRALIGN __attribute((aligned(LEVEL1_DCACHE_LINESIZE)))
#else
#define CTRALIGN
#endif


class Counter {
 protected:

    OPTIONS *  m_options;
    bool       m_initialized;
    uint64_t   m_target CTRALIGN;
    uint64_t   m_counter CTRALIGN;

    Counter() {
    assert(false);
    }

 public:

    Counter(OPTIONS * o) {
        assert(o != NULL);
        m_options = o;
        m_target = o->nTarget;
        m_counter = 0;
        m_initialized = false;
    }

    virtual ~Counter() {}

    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual void initialize() = 0;
    virtual bool complete() = 0;
    virtual void increment() = 0;
    virtual uint64_t value() = 0;

};
```

Shared logic/ins
from OOP

- Use a super-class with vir
  - Each sub-class implements
  - Maximizes instrumentatio
- Note the alignment __att
  - What does it do?
  - Why is it important?



```
#define LEVEL1_DCACHE_LINESIZE 64
#ifdef ALIGN_COUNTER_VAR
#define CTRALIGN __attribute((aligned(LEVEL1_DCACHE_LINESIZE)))
#else
#define CTRALIGN
```

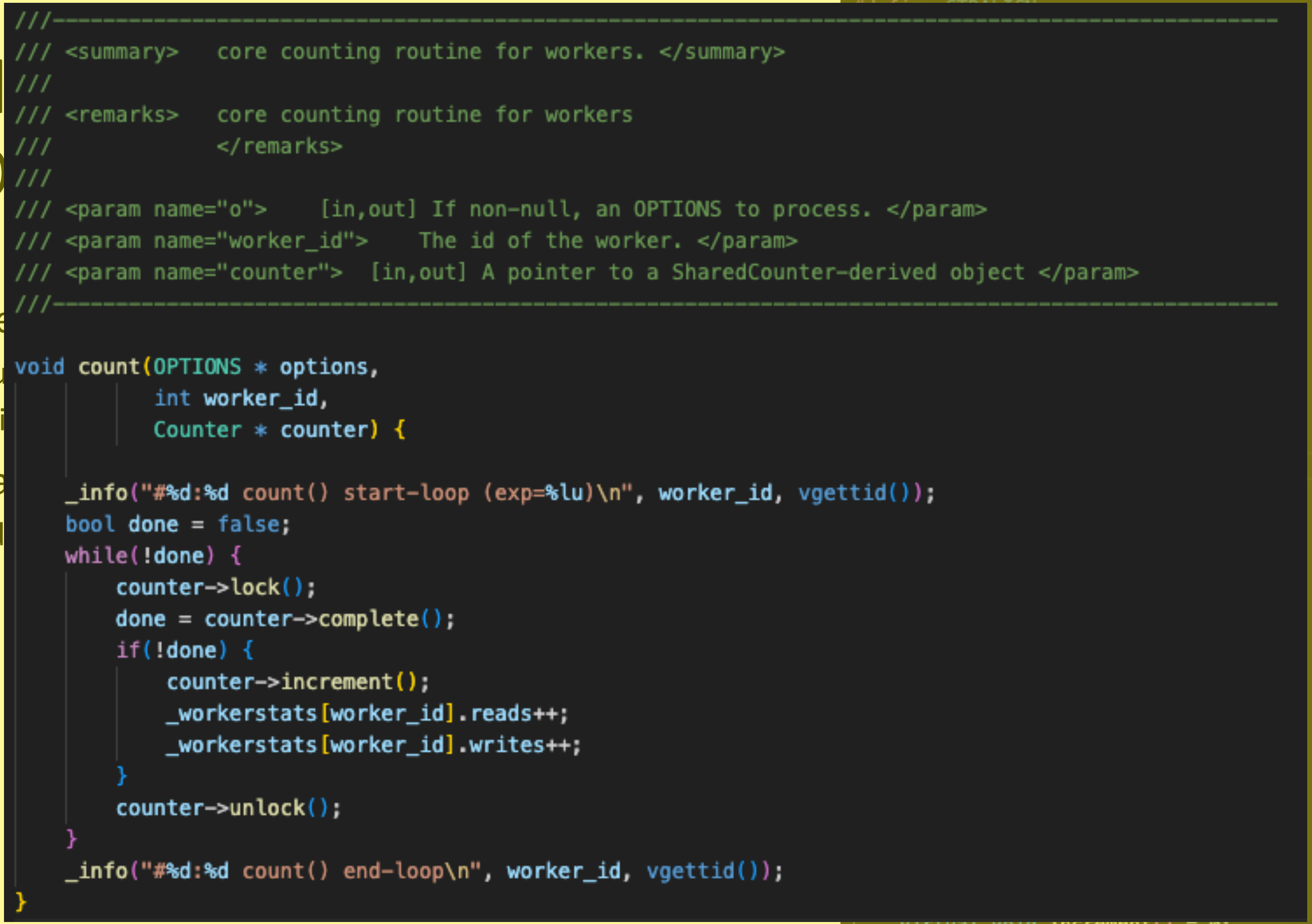
```
class AtomicCounter : public Counter {
 protected:
    std::atomic<uint64_t> m_value;
 public:
    void lock() {}
    void unlock() {}
    bool complete() {
        return m_value >= m_target;
    }
    void increment() {
        bool done = false;
        uint64_t oldval = m_value;
        while(!done) {
            uint64_t newval = m_value + 1;
            done = (newval > m_target) || m_value.compare_exchange_strong(oldval, newval);
            if(!done)
            oldval = m_value;
        }
    }
```

```
    virtual ~Counter() {}

    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual void initialize() = 0;
    virtual bool complete() = 0;
    virtual void increment() = 0;
    virtual uint64_t value() = 0;

};
```

Shared
from O

- Use a supe
  - Each su
  - Maximi
- Note the a
  - What d
  - Why is

```
#define LEVEL1_DCACHE_LINESIZE 64
#ifdef ALIGN_COUNTER_VAR
#define CTRALIGN __attribute((aligned(LEVEL1_DCACHE_LINESIZE)))
#else
```

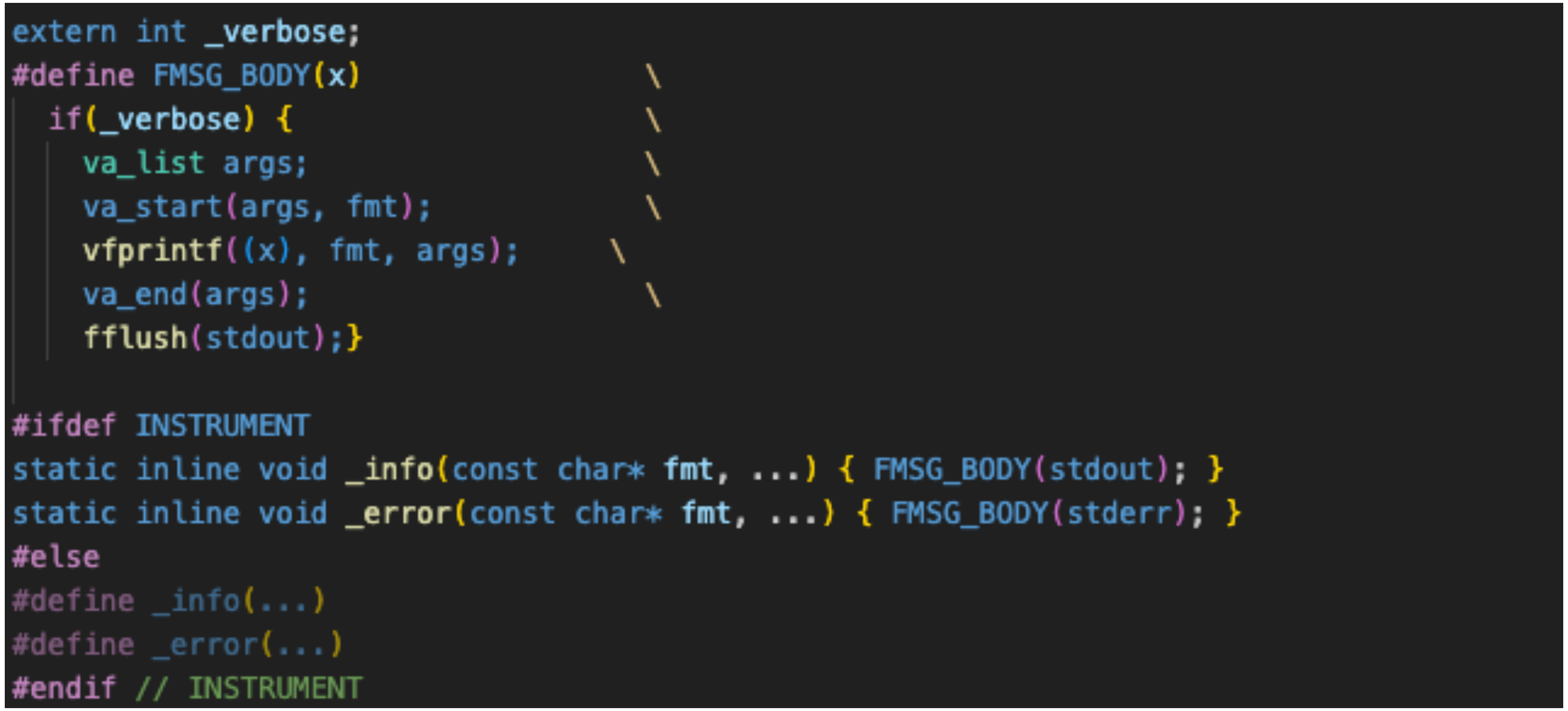


```
///----------------------------------------------------------------------------------------------
/// <summary>   core counting routine for workers. </summary>
///
/// <remarks>   core counting routine for workers
///             </remarks>
///
/// <param name="o">    [in,out] If non-null, an OPTIONS to process. </param>
/// <param name="worker_id">    The id of the worker. </param>
/// <param name="counter">  [in,out] A pointer to a SharedCounter-derived object </param>
///----------------------------------------------------------------------------------------------

void count(OPTIONS * options,
           int worker_id,
           Counter * counter) {

    _info("#%d:%d count() start-loop (exp=%lu)\n", worker_id, vgettid());
    bool done = false;
    while(!done) {
        counter->lock();
        done = counter->complete();
        if(!done) {
            counter->increment();
            _workerstats[worker_id].reads++;
            _workerstats[worker_id].writes++;
        }
        counter->unlock();
    }
    _info("#%d:%d count() end-loop\n", worker_id, vgettid());
}
```

```
    virtual void increment() = 0;
    virtual uint64_t value() = 0;

};
```

Clean instrumentation

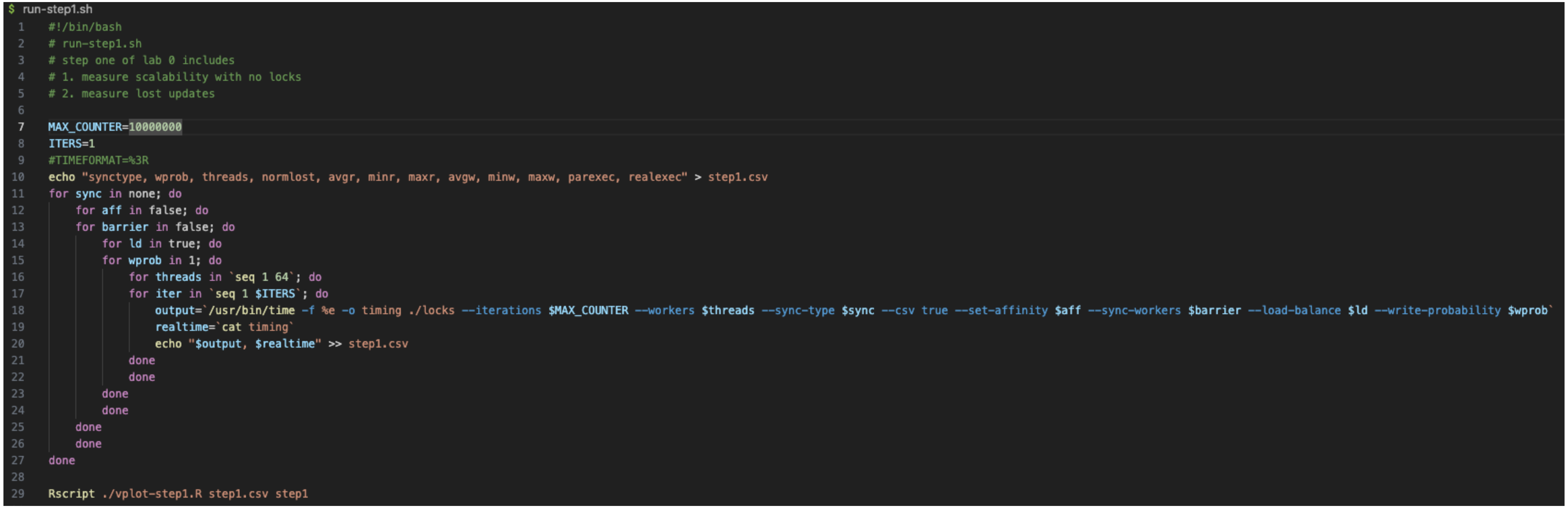
- [Performance] debug output is useful
  - It also impacts performance
  - And gets in the way of scriptable output
- I almost always use a verbose flag, and variadic _info()



```
extern int _verbose;
#define FMSG_BODY(x)                 \
  if(_verbose) {                     \
    va_list args;                    \
    va_start(args, fmt);             \
    vfprintf((x), fmt, args);      \
    va_end(args);                    \
    fflush(stdout);}

#ifdef INSTRUMENT
static inline void _info(const char* fmt, ...) { FMSG_BODY(stdout); }
static inline void _error(const char* fmt, ...) { FMSG_BODY(stderr); }
#else
#define _info(...)
#define _error(...)
#endif // INSTRUMENT
```

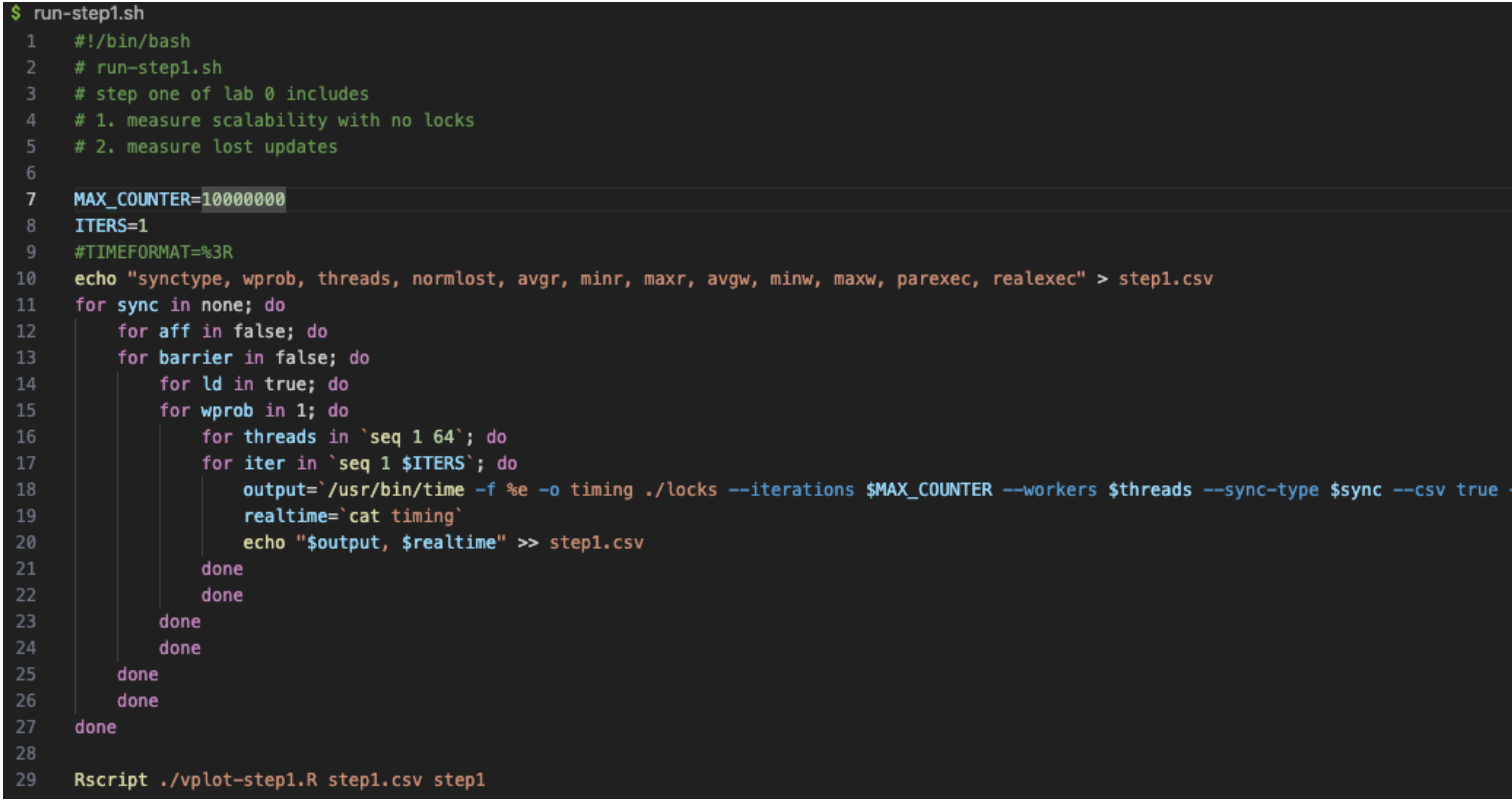
Producing Scriptable Output

- This lab requires you to run the same program 100s of times
- Collecting data to graph and keeping track of it manually is tedious
- Good empirical method: ***program*** your measurements/experiments
    - Produce machine-readable output
    - Write scripts manage runs and collect data
    - Write scripts to run and graph steps



$ run-step1.sh

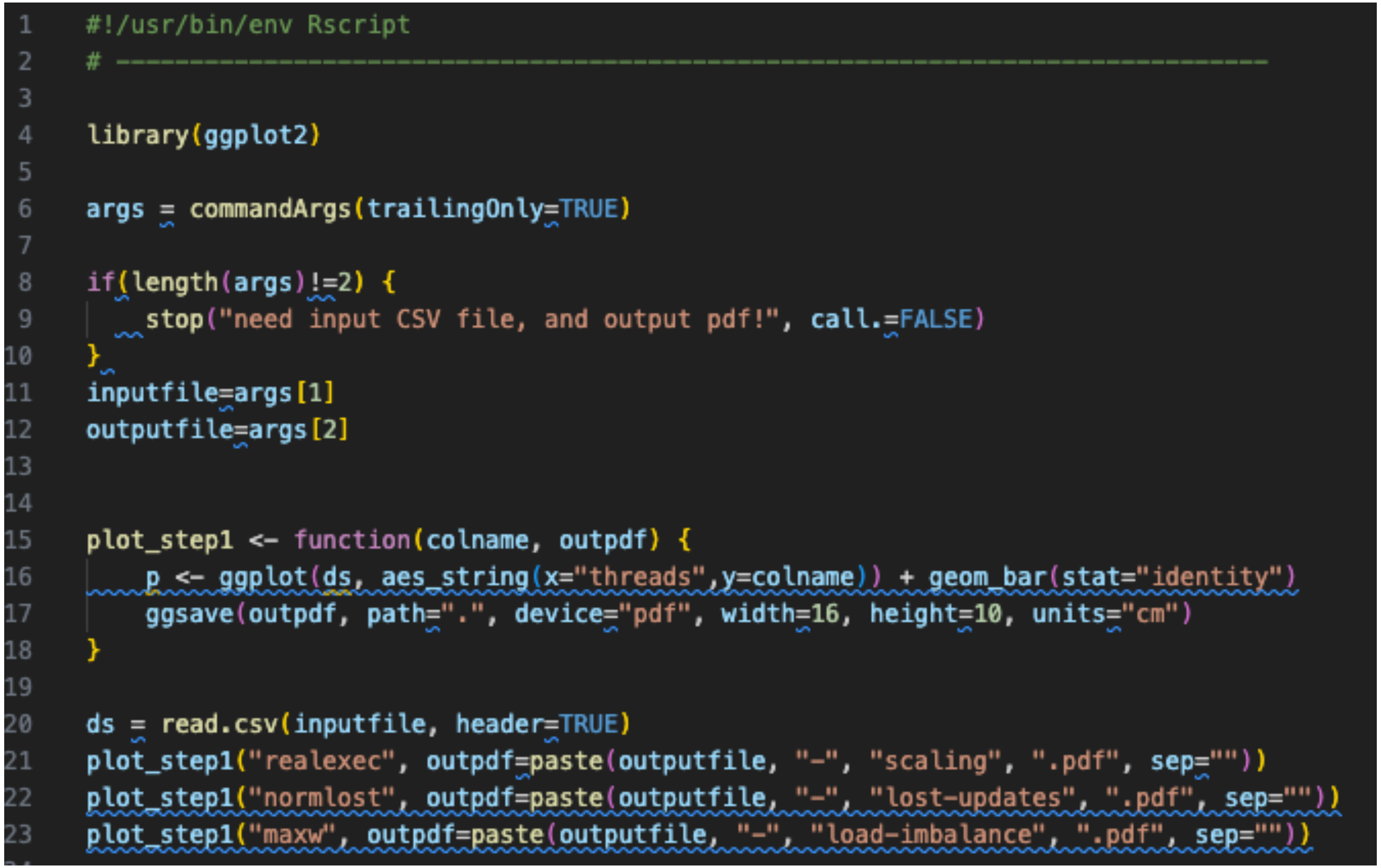
```
#!/bin/bash
# run-step1.sh
# step one of lab 0 includes
# 1. measure scalability with no locks
# 2. measure lost updates

MAX_COUNTER=10000000
ITERS=1
#TIMEFORMAT=%3R
echo "synctype, wprob, threads, normlost, avgr, minr, maxr, avgw, minw, maxw, parexec, realexec" > step1.csv
for sync in none; do
    for aff in false; do
    for barrier in false; do
        for ld in true; do
        for wprob in 1; do
            for threads in `seq 1 64`; do
            for iter in `seq 1 $ITERS`; do
                output=`/usr/bin/time -f %e -o timing ./locks --iterations $MAX_COUNTER --workers $threads --sync-type $sync --csv true --set-affinity $aff --sync-workers $barrier --load-balance $ld --write-probability $wprob`
                realtime=`cat timing`
                echo "$output, $realtime" >> step1.csv
            done
            done
        done
        done
    done
    done
done

Rscript ./vplot-step1.R step1.csv step1
```



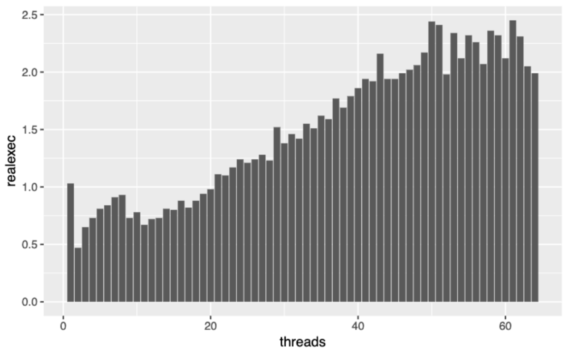
$ run-step1.sh

```bash
#!/bin/bash
# run-step1.sh
# step one of lab 0 includes
# 1. measure scalability with no locks
# 2. measure lost updates

MAX_COUNTER=10000000
ITERS=1
#TIMEFORMAT=%3R
echo "synctype, wprob, threads, normlost, avgr, minr, maxr, avgw, minw, maxw, parexec, realexec" > step1.csv
for sync in none; do
    for aff in false; do
    for barrier in false; do
        for ld in true; do
        for wprob in 1; do
            for threads in `seq 1 64`; do
            for iter in `seq 1 $ITERS`; do
                output=`/usr/bin/time -f %e -o timing ./locks --iterations $MAX_COUNTER --workers $threads --sync-type $sync --csv true -
                realtime=`cat timing`
                echo "$output, $realtime" >> step1.csv
            done
            done
        done
        done
    done
    done
done

Rscript ./vplot-step1.R step1.csv step1
```

Scripting Graph Production

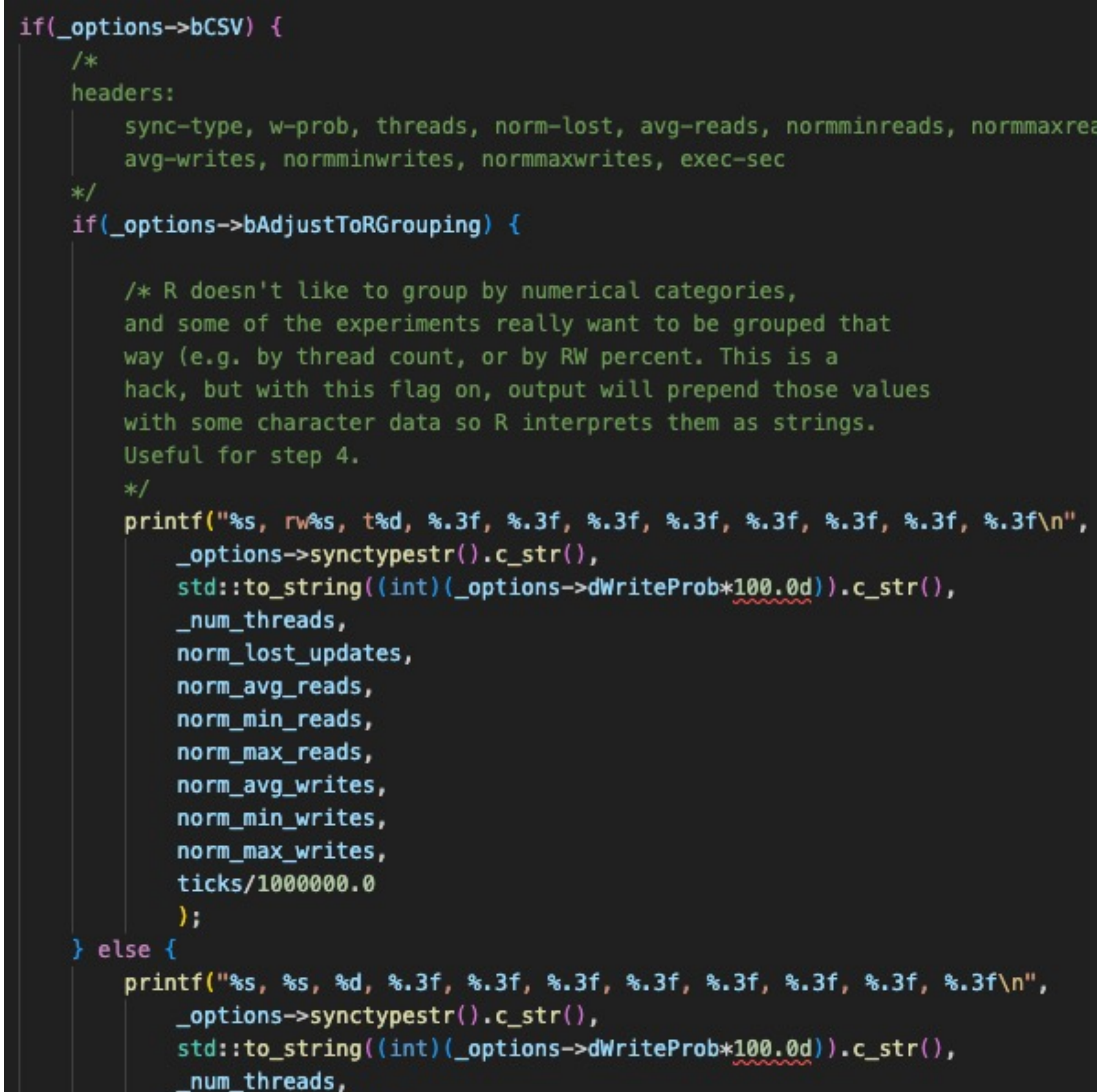


```r
#!/usr/bin/env Rscript
# ------------------------------------------------------------------------------

library(ggplot2)

args = commandArgs(trailingOnly=TRUE)

if(length(args)!=2) {
    stop("need input CSV file, and output pdf!", call.=FALSE)
}
inputfile=args[1]
outputfile=args[2]


plot_step1 <- function(colname, outpdf) {
    p <- ggplot(ds, aes_string(x="threads",y=colname)) + geom_bar(stat="identity")
    ggsave(outpdf, path=".", device="pdf", width=16, height=10, units="cm")
}

ds = read.csv(inputfile, header=TRUE)
plot_step1("realexec", outpdf=paste(outputfile, "-", "scaling", ".pdf", sep=""))
plot_step1("normlost", outpdf=paste(outputfile, "-", "lost-updates", ".pdf", sep=""))
plot_step1("maxw", outpdf=paste(outputfile, "-", "load-imbalance", ".pdf", sep=""))
```



Producing Scriptable Output

- Added command line options to control:
    - Human vs machine-readable
    - Manage quirks in Rscript
    - etc

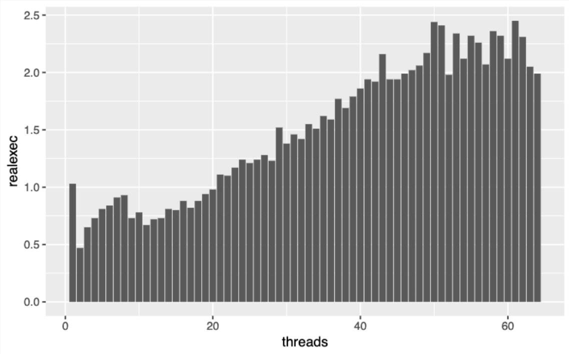


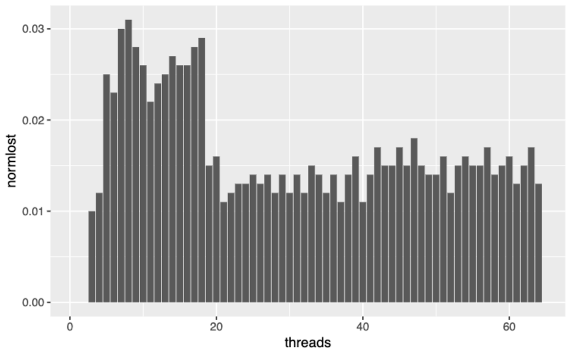
```
if(_options->bCSV) {
    /*
    headers:
        sync-type, w-prob, threads, norm-lost, avg-reads, normminreads, normmaxrea
        avg-writes, normminwrites, normmaxwrites, exec-sec
    */
    if(_options->bAdjustToRGrouping) {

        /* R doesn't like to group by numerical categories,
        and some of the experiments really want to be grouped that
        way (e.g. by thread count, or by RW percent. This is a
        hack, but with this flag on, output will prepend those values
        with some character data so R interprets them as strings.
        Useful for step 4.
        */
        printf("%s, rw%s, t%d, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f\n",
            _options->synctypestr().c_str(),
            std::to_string((int)(_options->dWriteProb*100.0d)).c_str(),
            _num_threads,
            norm_lost_updates,
            norm_avg_reads,
            norm_min_reads,
            norm_max_reads,
            norm_avg_writes,
            norm_min_writes,
            norm_max_writes,
            ticks/1000000.0
            );
    } else {
        printf("%s, %s, %d, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f\n",
            _options->synctypestr().c_str(),
            std::to_string((int)(_options->dWriteProb*100.0d)).c_str(),
            _num_threads,
```

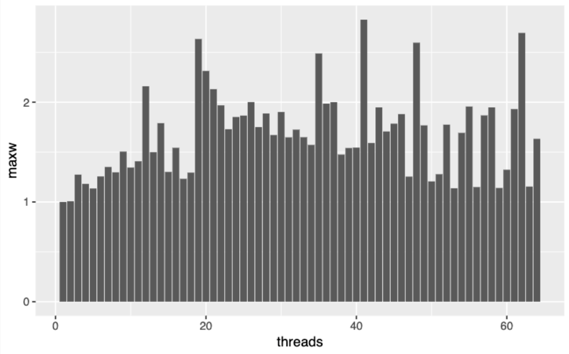
Methodology

Languedoc.csres.utexas.edu

- 64 cores 2-way hyperthreaded
- Ubuntu 20.04
- 192GB RAM
- Lightly shared







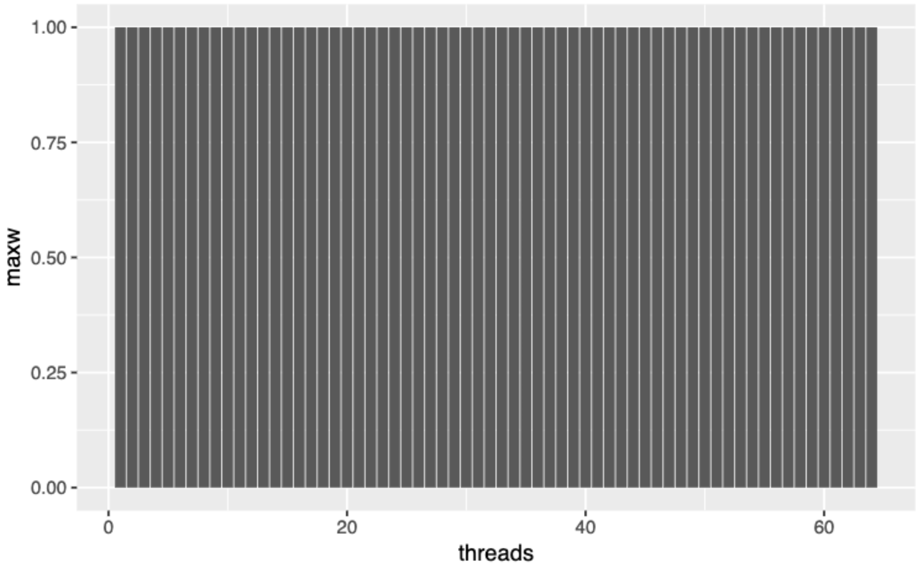
Step 1: Unsynchronized Counting

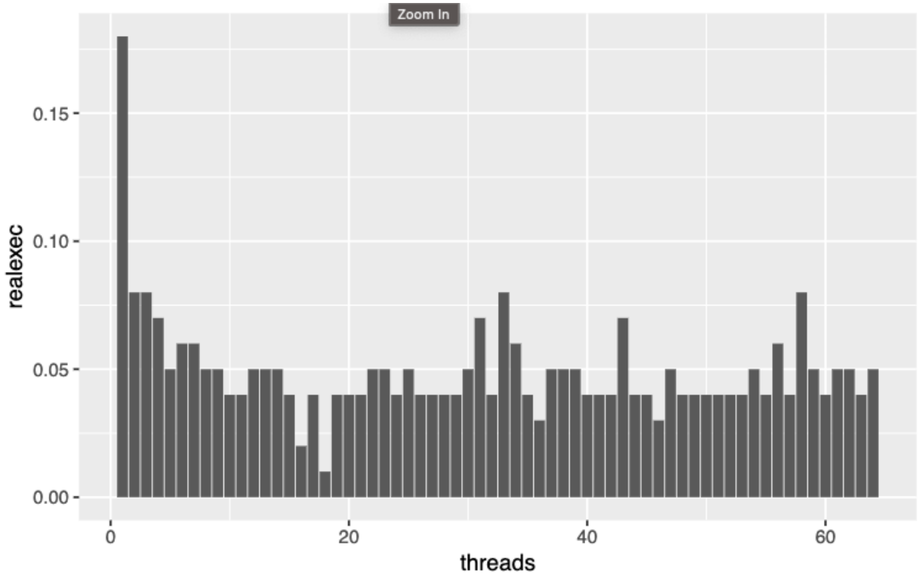
- The program gets slower with more threads. Why?
- Many lost updates. Obvious why…
- Load imbalance can be greater than 2x

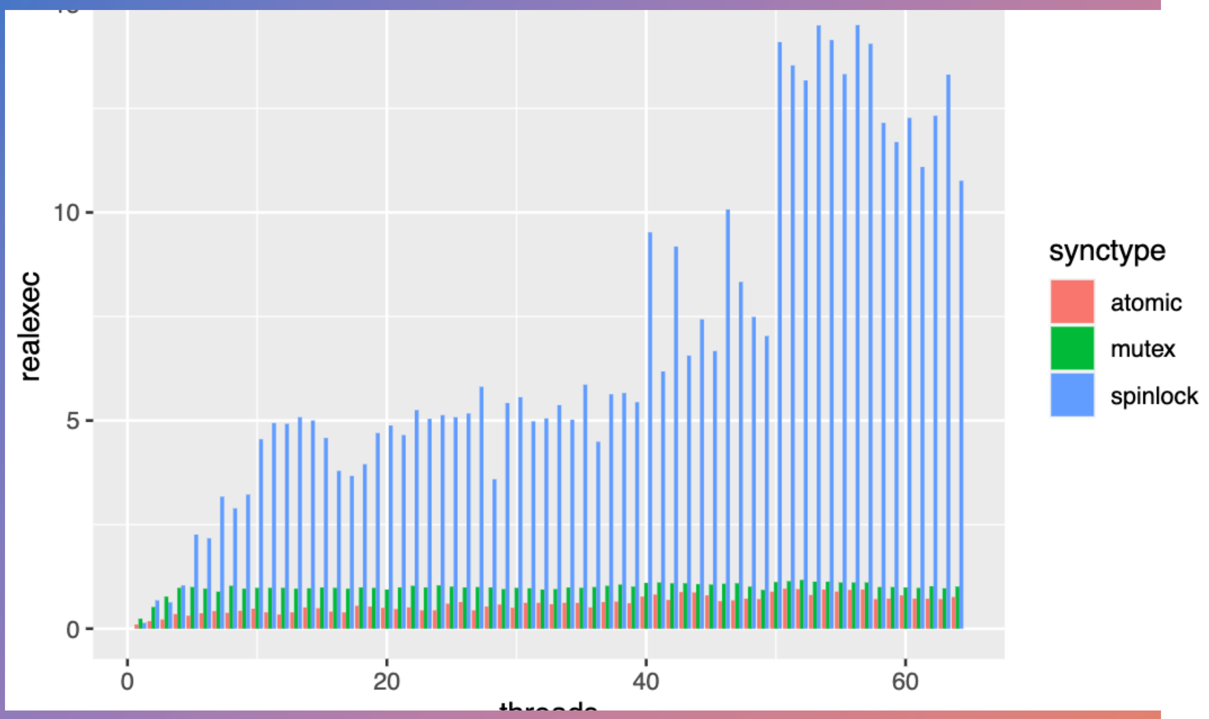
- Ideas:
  - Balance load
  - Eliminate contention
  - Is it possible to scale *at all* if you remove all contention?

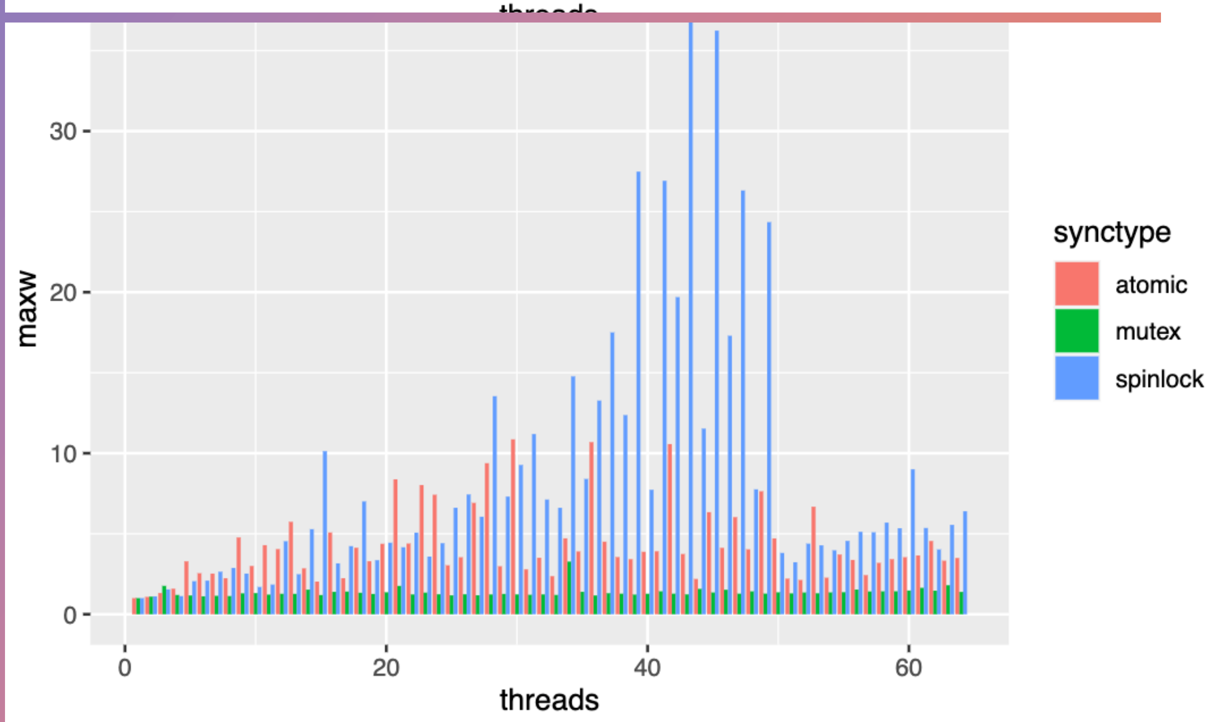
Step 1a: Privatize the Counter

- It can indeed be made to scale!
- Load can be balanced
- Diagnosis: lost updates and coherence are perf-killers; in my case lost updates were worse
- Problem: there is no actual shared state
- Solution: synchronize



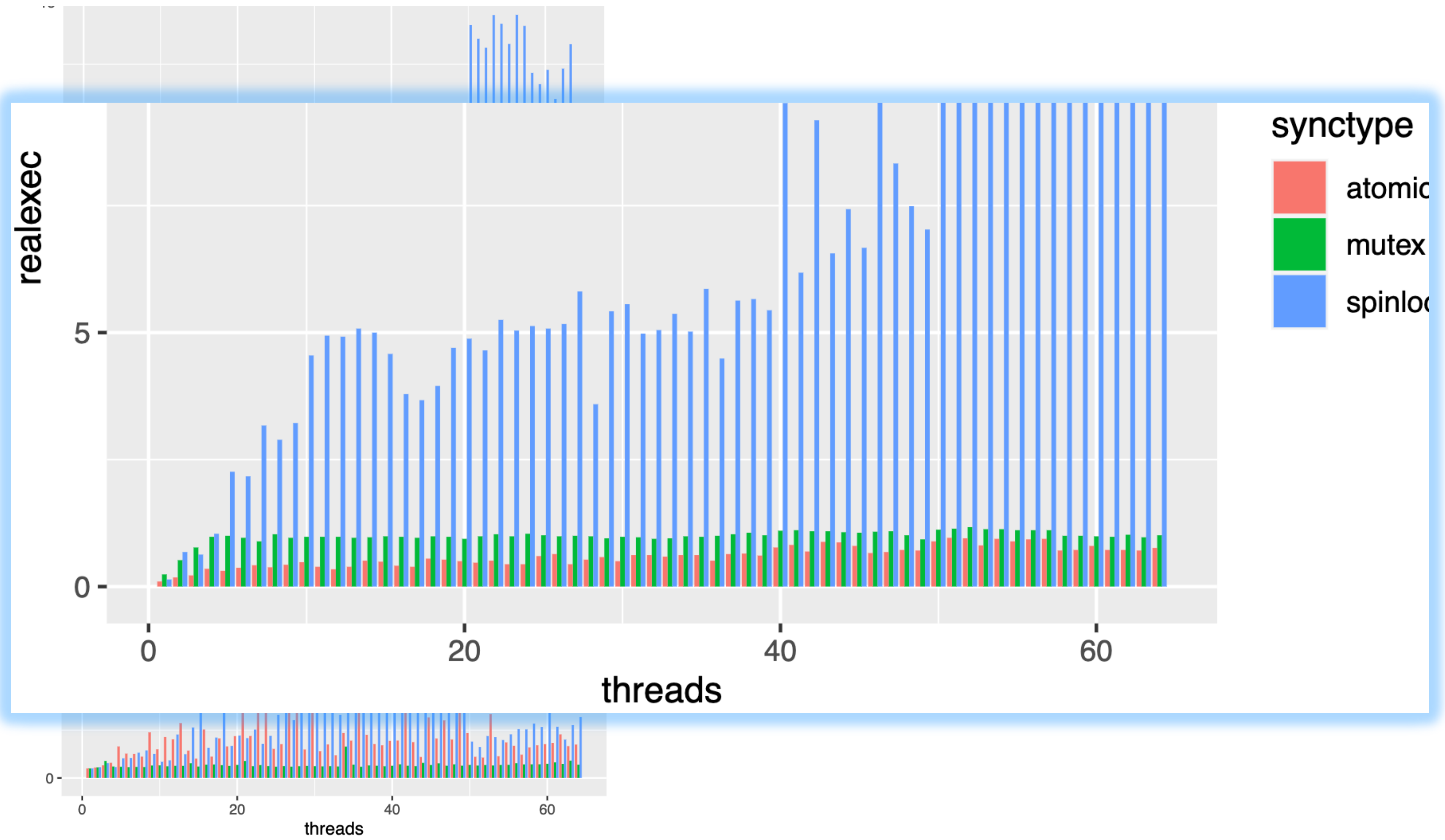


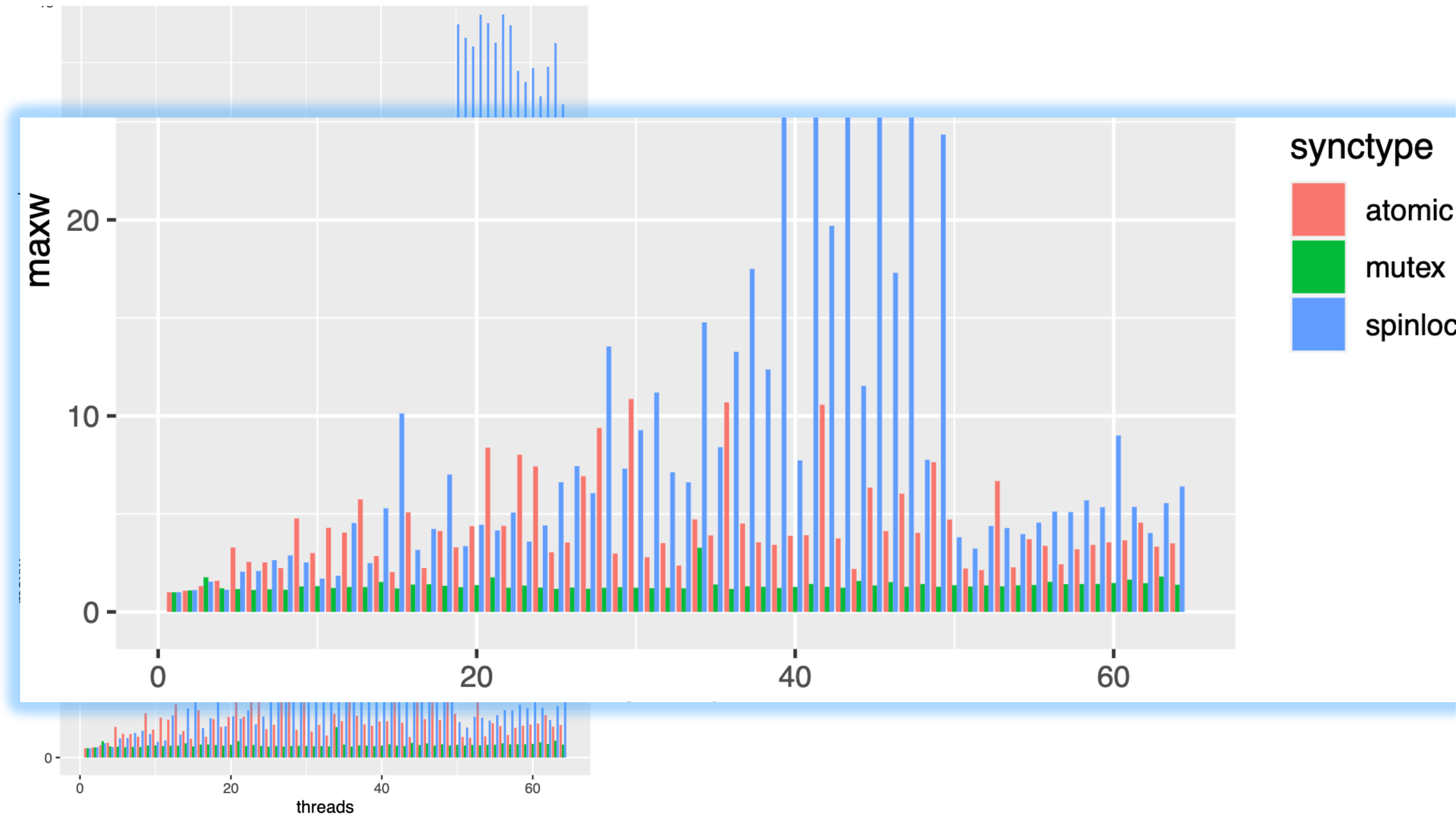




Step 2: Synchronize the Counter

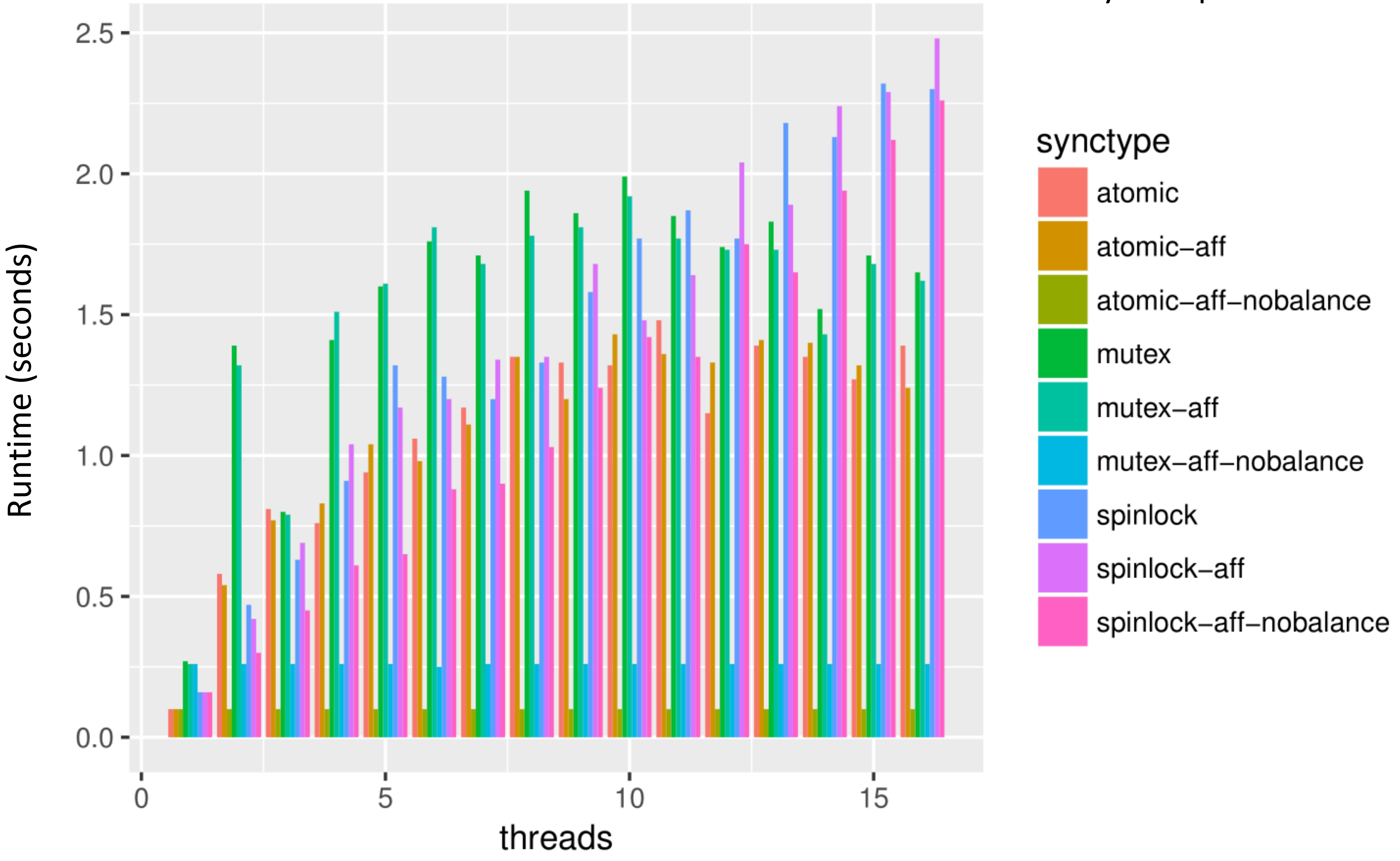
- Scalability is outrageous
- Load imbalance differs by synctype
- Does it get any better if we balance the load?


realexec
5
0
0
20
40
60
threads
synctype
atomic
mutex
spinloc
0
0
20
40
60
threads


maxw
20
10
0
0
20
40
60
synctype
atomic
mutex
spinloc
0
0
20
40
60
threads

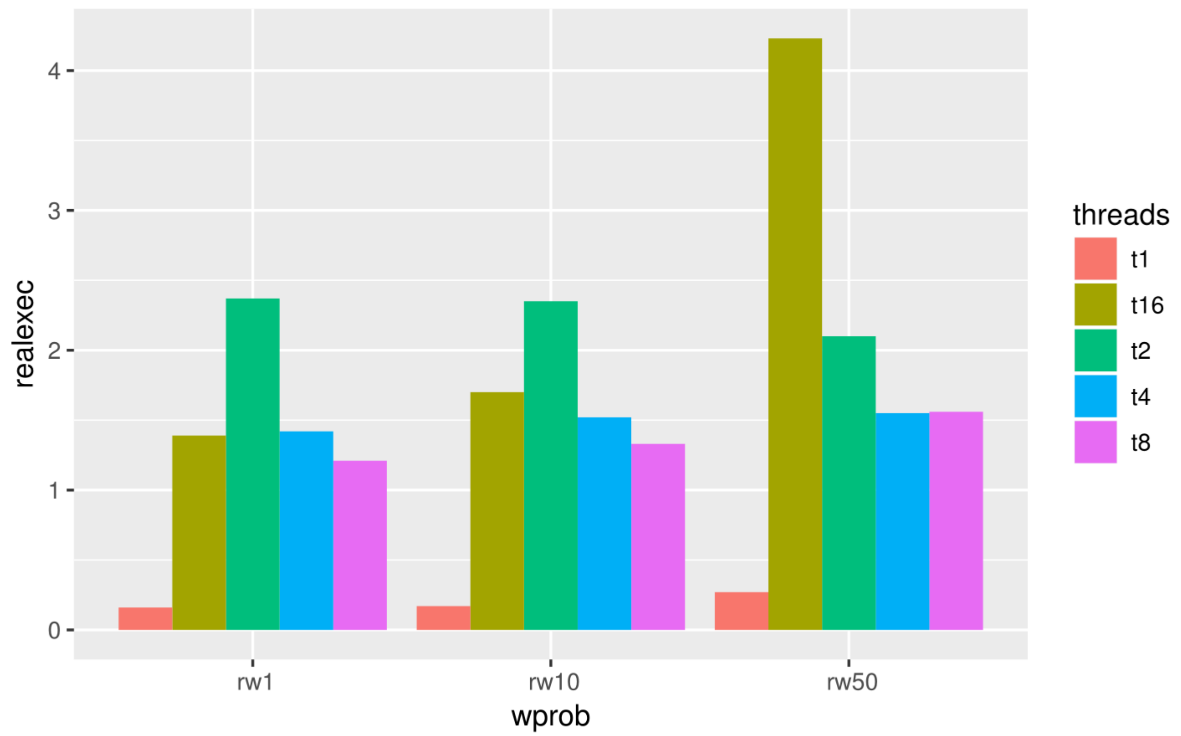
Step 3

- Use/abuse affinity
    - 1 core vs all cores?
- What did you expect for load imbalance
- What did you expect for scaling?

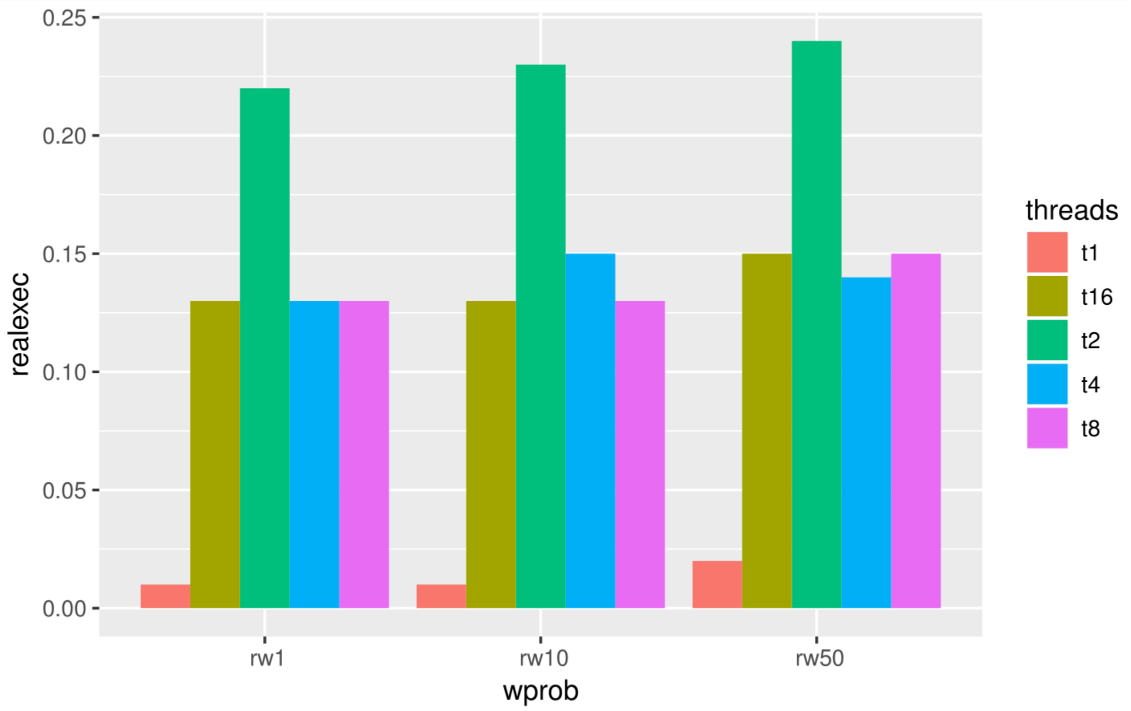


Step 4

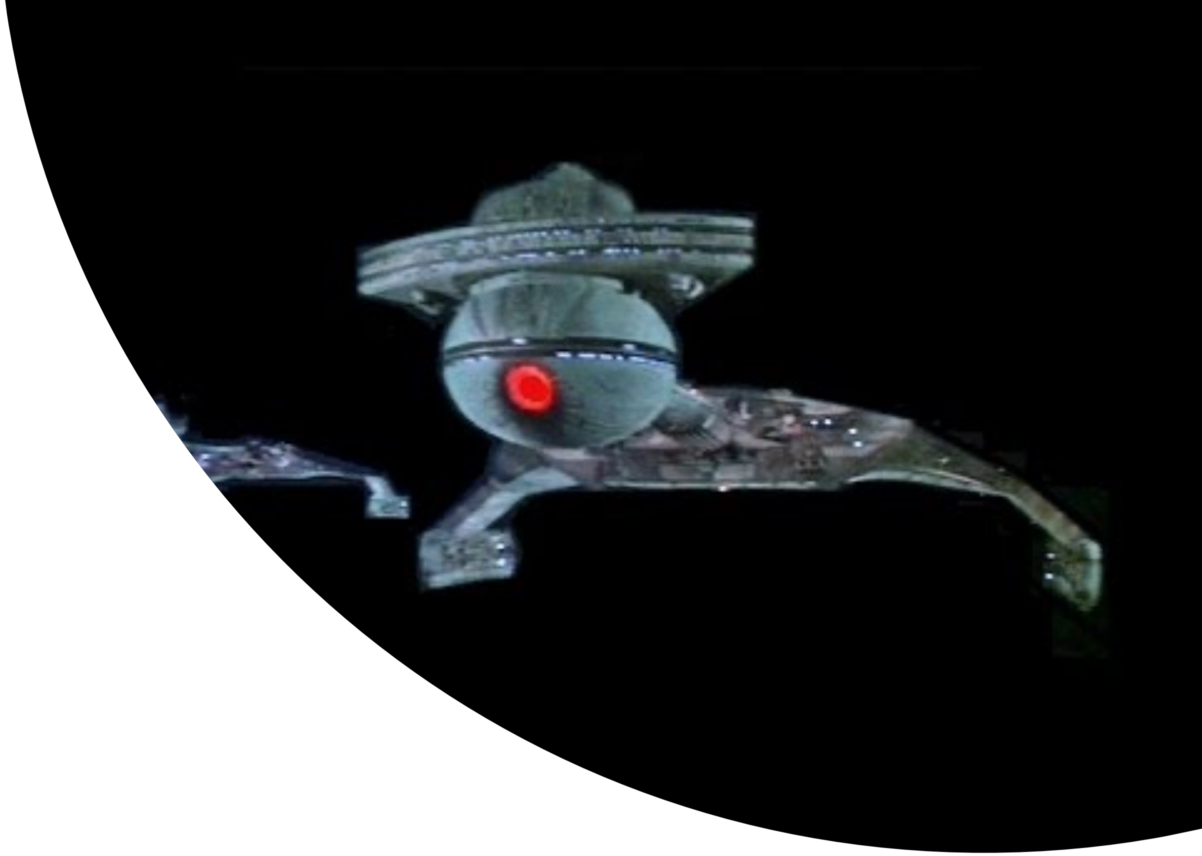
- Read-write ratios
- What did you expect for scaling?



Spinlocks



Atomics



Discussion