Benchmarking Two-Locus Framework and LD Calculator

October 6, 2025

Contents

1 Introduction

1.1 Background

Initially, the purpose of creating the two-locus framework in tskit was to replace and deprecate the LD Calculator. The two-locus framework has more features and flexibility than the LD Calculator and due to some of the design elements, there are some concerns about the performance of the two-locus framework and its suitibility for replacing the LD Calculator. In any case, we might not want to deprecate the LD Calculator depending on the appetite for API changes, but this will serve as a useful comparison between the methods.

The two-locus framework allows users to produce statistics on subsets of the samples in a tree sequence, which means that we must store sample identities and track which mutations contain which sample. The concern is that an increasing number of samples will introduce many more set operations, which in turn may be slower than the existing LD Calculator tool. The LD Calculator tool counts samples under each node as it advances trees along the tree sequence tracking shared haplotypes and the total number of samples. This method is algorithmically efficient, but its code structure (no function pointers) also allows the compiler to perform more aggressive optimization, such as inlining the summary function, which seems to reduce the memory latency and improve cache efficiency.

In this benchmarking study, I will generate some reasonably sized tree sequences, then I will run the LD Calculator and the two-locus r^2 function to produce an LD matrix. When calling the two-locus functions, I use the default behavior of computing r^2 between all sites and all samples within the tree sequence to create a comparable computational problem for each tool. I generate 10 tree sequences with constant demographic parameters and an increasing number of samples to understand how the two-locus framework scales.

After some initial benchmarking, we found the LD Calculator to be much faster than the two-locus framework. During this analysis, we noticed some low-hanging optimizations that could be made to the two-locus framework to improve its performance to be on-par with the LD Calculator. To evaluate the performance increase from each optimization, I present a series of benchmarks that layer each of the three performance improvements on one another with the aim of 1) explaining these changes effectively and 2) allowing for some selectivity in which improvements we end up accepting. Ideally, we would agree that all of my changes are acceptable, but it's important to keep in mind that we should prioritize simplicity. The changes are detailed in the results section below.

1.2 Benchmarking Details

Here, I will generate tree sequences with an increasing number of samples. To roughly mirror human-like parameters, I chose a recombination rate of 10^{-8} and N_e of 10^4. I set the ploidy to 1 just so that we could explicitly set the number of samples. To limit the runtime, I chose a genome length of 2×10^6, which produces a number of sites on the order of 10^3. I also set `discrete_genome=False` to avoid simulating multiallelic sites, which will cause the LD Calculator to fail.

2 Simulations

First, we need a suitable dataset for benchmarking. We will generate our dataset with `msprime`, using the considerations mentioned in the previous section.

```python
from pathlib import Path
import shutil
import msprime
import numpy as np

def gen_data(L, sample_sizes, outpath=Path("trees"), seed=23, n=10):
    rng = np.random.RandomState(seed)
    seeds = rng.randint(1, 2**31, (n, 2))

    if outpath.exists():
        shutil.rmtree(outpath)
    outpath.mkdir()

    out = [["Sample Size", "Num Sites", "Num Trees", "Num Edges"], None]
    for samp_size, (anc_seed, mut_seed) in zip(sample_sizes, seeds):
        ts = run_msprime(L, samp_size, anc_seed, mut_seed)
        fname = f"n={ts.num_samples}.trees"
        outrow = [ts.num_samples, ts.num_sites, ts.num_trees, ts.num_edges]
        out.append([f"\num{{{r}}}" for r in outrow])
        ts.dump(outpath / fname)
    return out

def run_msprime(L, samp_size, anc_seed, mut_seed):
    ts = msprime.sim_ancestry(
        samp_size,
        ploidy=1,
        sequence_length=L,
        discrete_genome=False,
        recombination_rate=1e-8,
        population_size=1e4,
        random_seed=anc_seed,
    )
    return msprime.sim_mutations(
        ts,
        rate=1e-8,
        random_seed=mut_seed,
        discrete_genome=False,
    )
```

```python
gen_data(L=2e6, sample_sizes=np.logspace(2, 5, 10, dtype=int))
```

Sample Size	Num Sites	Num Trees	Num Edges
100	2,109	1,871	6,838
215	2,474	2,151	8,095
464	2,759	2,485	9,941
1,000	2,925	2,673	11,812
2,154	3,307	3,013	15,346
4,641	3,501	3,368	21,841
10,000	3,925	3,694	33,946
21,544	4,379	3,884	57,811
46,415	4,570	4,362	109,366
100,000	4,925	4,652	217,700

Table 1: Properties of the simulated tree sequences

3 Benchmarks

3.1 Implementation Details

3.1.1 Environment

I'm running all of these benchmarks on my laptop, which has an AMD Ryzen 7 7840U (8 cores/16 threads). I'm running benchmarks locally, because running them on our compute cluster will present the problem of heterogeneous hardware and I can't control for all of the details as easily. With smaller, reasonable benchmarks, I find that the runtime variance can increase substantially. To mitigate this, I've taken a few precautions. Hyperthreading is the mechanism by which two threads can share the same core, which usually provides a boost of CPU throughput. In our case, these shared threads will experience cache contention because the two threads will share the same cache, causing variance in runtimes (this is probably the largest contributor to runtime variance). In addition, I reserve the cores that I want to use so that most kernel processes will avoid running on these CPUs. To disable hyperthreading, I use the `nosmt` kernel parameter, and to ensure that most kernel functionality happens on a dedicated cpu (`cpu0`), I use the `isolcpus` parameter, with a few additional flags. More details can be found in the linux kernel documentation. For posterity, I've listed the exact parameters used below.

```
isolcpus=nohz,domain,managed_irq,2,4,6,8,10,12,14
nosmt
```

I'll be using 7 cpus for benchmarks, leaving all other processes to run on `cpu0`. In addition to these changes, I've ensured that my CPU governor is set to `performance` to limit the amount of CPU scaling that occurs and I've turned off CPU boosting to obtain a more consistent result.

```
echo 0 | sudo tee /sys/devices/system/cpu/cpufreq/boost > /dev/null
SYS=/sys/devices/system/cpu
for f in $SYS/cpu*/cpufreq/scaling_governor; do
    echo performance | sudo tee $f > /dev/null
done
echo "Frequency scaling: $(cat $SYS/cpu*/cpufreq/scaling_governor | sort -u)"
echo "Frequency boost: $(< $SYS/cpufreq/boost)"
```

Result

```
Frequency scaling: performance
Frequency boost: 0
```

Finally, I use cgroups to limit user processes from running on the benchmark CPUS. For ease of creation and destruction, I use `systemd` slices to manage my cgroups. I first create a cgroup for benchmarks (`benchmark.slice`), specifying the allowed CPUs. Then, I limit the system-level and user processes to running on `cpu0`. This leaves us with a reasonably isolated environment to perform our benchmarks.

```
sudo systemctl set-property \
     --runtime benchmark.slice AllowedCPUs=2,4,6,8,10,12,14
sudo systemctl start benchmark.slice
sudo systemctl set-property --runtime init.slice AllowedCPUs=0
sudo systemctl set-property --runtime system.slice AllowedCPUs=0
sudo systemctl set-property --runtime user.slice AllowedCPUs=0
```

Here, I verify that the above changes have taken effect.

```
systemctl show -p ControlGroup -p EffectiveCPUs \
          system.slice user.slice benchmark.slice
```

Result

```
ControlGroup=/system.slice
EffectiveCPUs=0

ControlGroup=/user.slice
EffectiveCPUs=0

ControlGroup=/benchmark.slice
EffectiveCPUs=2 4 6 8 10 12 14
```

3.1.2 Individual Benchmark Script

The following will serve as our benchmarking script. It will output the wall time for computing an LD matrix, along with relevant information about the tree sequence that was used in the benchmark (useful for plotting). This script also sets `OPENBLAS_NUM_THREADS=1` to prevent the openblas thread server from being activated, which causes an initial lag to the code start. This lag makes flame graphs more difficult to interpret and can add variance to our runtime.

```python
import os

os.environ["OPENBLAS_NUM_THREADS"] = "1"

import argparse
import time

import tskit
parser = argparse.ArgumentParser()
parser.add_argument("tree_sequence_file")
parser.add_argument("method", choices=["two-locus", "ld-calc"])
args = parser.parse_args()
ts = tskit.load(args.tree_sequence_file)

match args.method:
    case "two-locus":
        start = time.time()
        ts.ld_matrix()
        end = time.time()
    case "ld-calc":
        start = time.time()
        tskit.LdCalculator(ts).r2_matrix()
        end = time.time()
```

```
print(args.method, end - start, ts.num_samples,
      ts.num_sites, ts.num_trees, ts.num_edges, sep="\t")
```

3.1.3 Parallel Benchmark Script

Given the changes made above, I'll want a script that orchestrates all of my jobs. These benchmarks will run on 7 cores simultaneously, pinning the processor on which they execute (with taskset), to avoid the linux thread scheduler migrating our processes around.

The following bash script suffices as a parallel benchmark executor, limiting the number of processes and only scheduling another when it's clear that a CPU core has become available. We'll run 15 replicates per simulation. This is done by using the built-in job control mechanisms in bash (note that this lags behind a bit, so we have to sleep to wait for it to recognize that a job is complete before we launch another).

```
#!/usr/bin/env bash
# methods to run (default two-locus and ld-calc)
methods="${1:-two-locus ld-calc}"
n_reps=15
cpu_list="$(seq 2 2 14)"
function avail_cpus {
    if [[ -z "$(jobs -p)" ]]; then
        printf '%s\n' "$cpu_list"
    else
        printf '%s\n' "$cpu_list" \
            | grep -v -f <(for j in $(jobs -p 2>/dev/null); do
                                { taskset -cp "$j" 2>/dev/null || true; } \
                                    | awk '{print "^"$NF"$"}'
                            done)
    fi
}
function first_avail_cpu {
    local cpus
    cpus="$(avail_cpus)"
    [[ -n "$cpus" ]] && head -1 <<< "$cpus"
}
function cleanup {
    for j in $(jobs -p); do echo "cleaning job: $j" >&2; kill "$j"; done
}
trap cleanup exit
for file in trees/*.trees; do
    for method in $methods; do
        for rep in $(seq "$n_reps"); do
            if [[ -z "$(avail_cpus)" ]]; then
                wait -n
            fi
            cpu="$(first_avail_cpu)"
            while [[ -z "$cpu" ]]; do
                cpu="$(first_avail_cpu)"
                # wait for job table to catch up
                sleep 5
            done
            echo "Running $file $method on cpu: $cpu" >&2
            taskset -c "$cpu" \
                python bench.py "$file" "$method" &
        done
    done
done
wait
sleep 10 # ensure job table has emptied before exiting (race cond w/ cleanup)
```

3.1.4 Notes about visualizations

From our benchmark data, we're providing a couple of different views. The first is a simple plot of overall runtime for each method. The second view we're providing is a flame graph (more information on flame graphs can be found here and here). The flame graphs I'm producing trim the call stack down to the `TreeSequence_ld_matrix` function so that we're not bogged down by viewing results from the python runtime. Wall time and flame graphs will be produced for each added optimization, but keep in mind that I'm layering things on top of each other as I go. In the Conclusions section, I provide some high level comparisons of all methods to each other.

3.2 Results

Let's review each optimization and its performance improvement. Revisions will be managed with the nix package manager, which gives me a separate isolated environment for each benchmarked commit. The commit used for each test case can be found in the `rev` key in the nix code snippets. In these benchmarks, each optimization will be layered on to the previous, these are not tested in isolation. Here's a brief description of these changes

1. **Base case**: The original code that is currently in tskit (link).
2. **Malloc out of hot loop**: Preallocate a structure of arrays used for temporary calculations instead of allocating and freeing for every pair of sites (link).
3. **Refactor Bit Arrays**: Refactor bit array interface, removing the need for temporary arrays in many cases. All functions now take a row index as a parameter (link).
4. **Precompute Allele Counts and Biallelic Summary Function**: Store precomputed bit arrays for each sample set and allele and the count of samples for each allele. Introduce a biallelic summary function that avoids multiple redundant computations, leaving the original normalized summary function for multiallelic sites (link).

3.2.1 Base Case

We'll start with what will serve as the base case for these benchmarks, which is the most recent main branch at the time of writing. This includes my most recent pull request, which finalizes the interface and behavior of the two-locus framework. Our benchmark of the base case shows that we're anywhere from 5-10x slower than the LD Calculator (figure 1).

```
import "<<analysis-dir()>>/tskit.nix" {
  rev = "9821725e4706f61baa005e3def6fad354083973c";
  sha256 = "sha256-E/jJqQzwCCcgr2HOhyPbnttEi+lu8WKCVT+lfCwIUn4=";
}
```

```
mkdir -p bench
systemd-run --scope -p Slice=benchmark.slice ./run-bench > bench/base.tsv
```

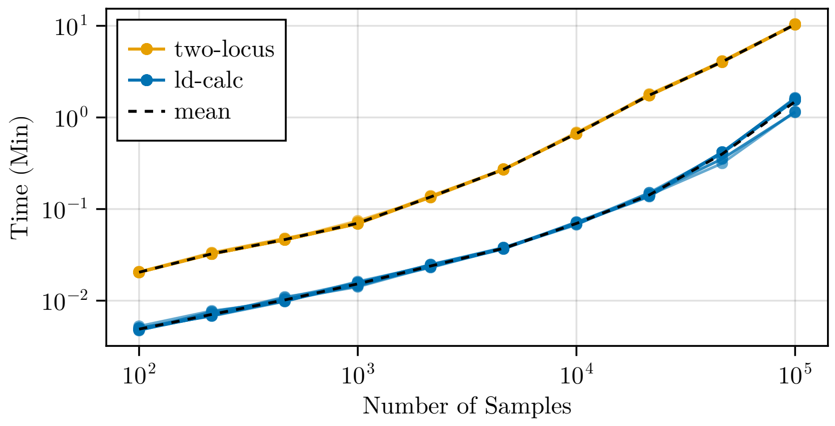

Figure 1: Total (wall) runtime of both methods in the base case.

Looking at figure 2, we can see that when there are a small number of samples in the tree sequence, a large part of the time spent computing the ld matrix is allocating and freeing memory. This is because we're allocating and initializing memory for storing results and intermediate arrays to perform our stats work. If we increase the size of the tree sequence from 100 samples to 10^4 samples, we see that the time spent in the `ld_matrix` function is dominated by bit array counting.

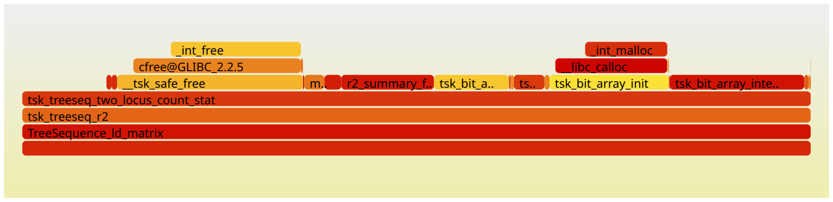

Figure 2: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 100 samples.

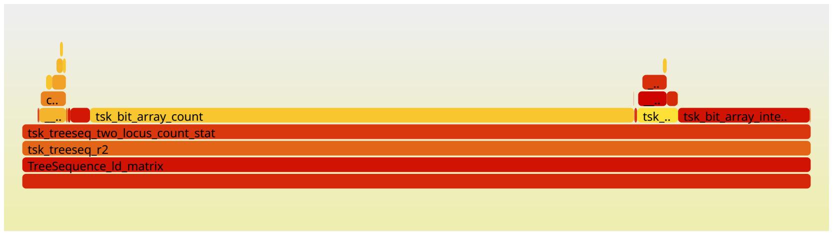

Figure 3: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 10000 samples.

3.2.2 Malloc out of the hot loop

As we can see from the flame graph in figure 2, a large proportion of the time spent in the `ld_matrix` function is in the `tsk_bit_array_init` and `__tsk_safe_free` functions (for small tree sequences). In the innermost loop of our stat computation, we are allocating temporary arrays for work on the heap. This operation is expensive, taking more time than the summary function calls. In order to reduce the performance hit from these repeated allocations, I create a structure that holds temporary arrays and pass it into the stat computation code so that it can be reused, reducing the number of malloc/free cycles that must be done for every pair of sites. For more details, see this diff.

```
import "<<analysis-dir()>>/tskit.nix" {
  rev = "462b6d936e088b5b72e4e2197a2201ee7d134670";
  sha256 = "sha256-pWK2JMIiF3MvtvgvjPzYEdL9DzG8quo5nI/NUOm9Fbg=";
}
```

```
mkdir -p bench
systemd-run --scope -p Slice=benchmark.slice ./run-bench > bench/malloc.tsv
```

As we might expect from inspecting the flame graphs in the previous section, we can see that there is some speed improvement for tree sequences with small numbers of samples ($\leq 10^4$ samples, figure 4). We'll review the relative change at the end in figure 14.

Looking at the flame graphs (figures 5 and 6), we can see that the time spent in the `ld_matrix` function is now dominated by calls to the summary function, bit array counting, and bit array intersection. In larger tree sequences, `ld_matrix` spends most of its time counting bit arrays.

3.2.3 Bit array refactor

As mentioned in this tskit issue, the bit arrays need a bit of refactoring. Since bit arrays are technically a list of N independent bit sets, many operations on bit arrays involve selecting one or two rows, storing these rows in a temporary array, then writing these results into another temporary object. The API would be simplified greatly if we instead passed in the whole array and specified the

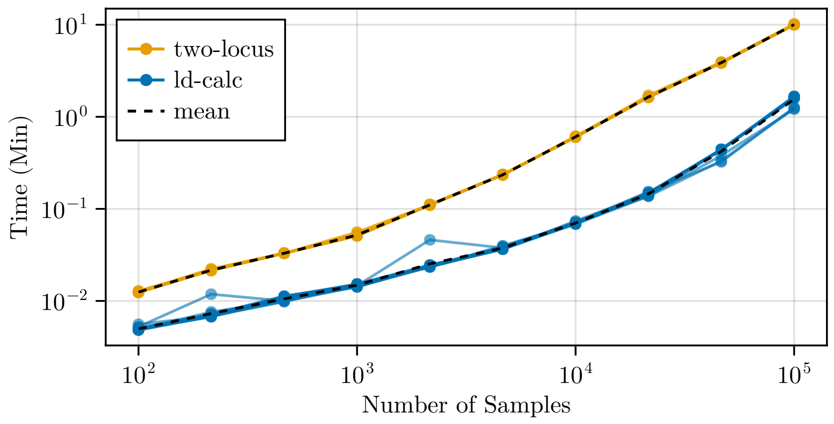

Figure 4: Total (wall) runtime of both methods with malloc operations pulled out of the innermost loop.

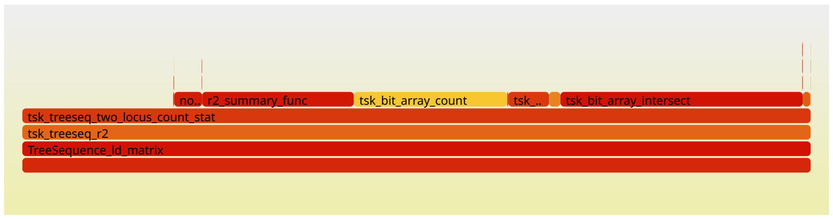

Figure 5: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 100 samples.

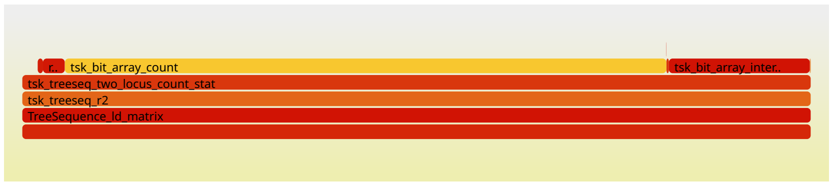

Figure 6: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 10000 samples.

row(s) on which to operate. This removes the reliance on temporary rows, which simplifies the calling code and makes the flow of data much easier to understand. Since we're removing temporary rows and relying on pointer arithmetic to perform operations, we also see a performance improvement from this refactor, which is why it is included in these benchmarks. For more details, see this diff.

```
import "<<analysis-dir()>>/tskit.nix" {
  rev = "8971aa9ef1d0c8bc422034bc1fb152746dfd24f3";
  sha256 = "sha256-rdUeN5OXbkLMFVNSwY4JNHMH4ESGog10mWg07yor70U=";
}
```

```
mkdir -p bench
systemd-run --scope -p Slice=benchmark.slice ./run-bench > bench/bitarray.tsv
```

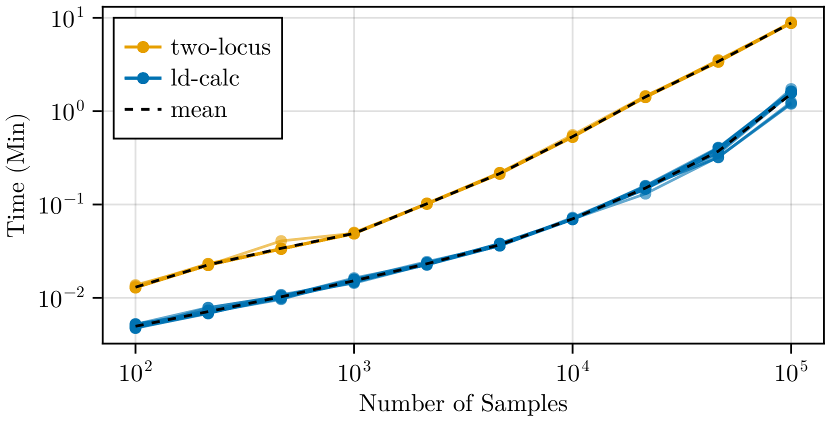

Figure 7: Total (wall) runtime of both methods with refactored bit arrays layered on top of the previous (malloc) changes.

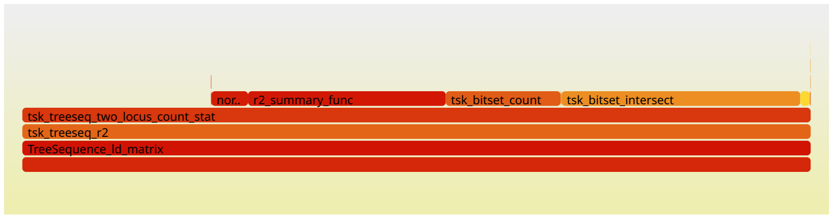

Figure 8: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 100 samples.

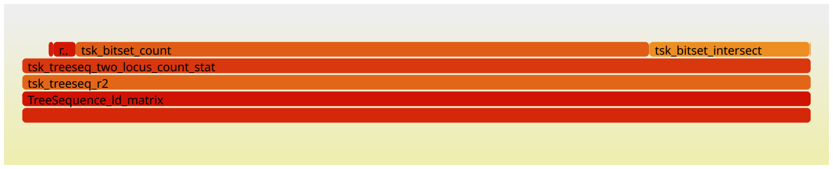

Figure 9: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 10000 samples.

3.2.4 Precompute A/B counts and biallelic summary func

This final optimization includes two main changes, 1) we're precomputing the counts of the A/B alleles and the bitsets we're passing into the summary function 2) providing a more efficient code path for the computation of LD stats on biallelic loci.

The largest and most effective optimization was to move the computation of allele counts from site A and site B out of the quadratic part of the algorithm (which is where we compute the LD matrix from the data). Our base-case algorithm computes counts of the A and B alleles for each pair when iterating over all sites in the LD matrix. This means that we inevitably recompute the counts of samples in the A and B allele unnecessarily. We can see this effect in figure 9, when there are many samples we spend the majority of the time in the `tsk_bitset_count` and `tsk_bitset_intersect` functions. This change will ensure that we only compute the counts of samples in each A and B allele once before we proceed to computing our statistic.

In the original base-case algorithm, we computed a bit array of samples containing each allele in each site. When computing the ld statistic, we had to index into this array and produce haplotype counts A, B, and AB. In the improved version, we still begin with this bit array computed from our tree sequence (table 2).

Site	Allele	Samples
0	0	`11111111 11000000 00000000 00000000`
0	1	`00000000 00111111 11110000 00000000`
0	2	`00000000 00000000 00001111 11111111`
1	0	`11111111 11111111 00000000 00000000`
1	1	`00000000 00000000 11111111 11111111`

Table 2: Hypothetical bit array with samples for each allele in two sites.

In our example, let's say we have the following sample sets. The original base-case algorithm also used this representation of the sample sets when computing ld statistics. Our improved algorithm starts with this example, but precomputes more up-front. Given the bit array of sample sets in table 3, we can precompute a matrix of samples containing each allele for each sample set (table 4).

Sample Set	Bitset
0	01010101 01010101 01010101 01010101
1	10101010 10101010 10101010 10101010

Table 3: Hypothetical bit array containing two sample sets on which to compute statistics.

S	A	SS	Count	Samples
0	0	0	5	01010101 01000000 00000000 00000000
0	0	1	5	10101010 10000000 00000000 00000000
0	1	0	5	00000000 00010101 01010000 00000000
0	1	1	5	00000000 00101010 10100000 00000000
0	2	0	6	00000000 00000000 00000101 01010101
0	2	1	6	00000000 00000000 00001010 10101010
1	0	0	8	01010101 01010101 00000000 00000000
1	0	1	8	10101010 10101010 00000000 00000000
1	1	0	8	00000000 00000000 01010101 01010101
1	1	1	8	00000000 00000000 10101010 10101010

Table 4: Precomputed accounting of alleles contained by each sample set. The column names were shortened: S is the site, A is the site's allele, and SS is the sample set. The precomputed counts of the number of samples in each sample set are stored in another array labeled count.

The one drawback is that all three of the data bit arrays defined above will need to be held in memory simultaneously, increasing the maximum memory usage. However, the increase in memory usage is still heavily outweighed by the size of the output matrix, which we store greedily in memory. Let's consider a scenario where we have a tree sequence with 10^5 sites and 10^5 samples. The output matrix would be $(10^5 \times 10^5 \times 8)/2^{30} = 74.5$ GiB, where 8 is the size of a double. In comparison, the base-case code would use $(10^5 \times 2 \times 4)/2^{10} = 195$ KiB (assuming all sites are biallelic and 4 is the size of a `uint32`). Our new implementation would use $(10^5 \times 2 \times 2 \times 4)/2^{20} = 1.5$ MiB memory for its intermediate data structures (assuming that there are 2 sample sets). In any case, for tree sequences with large numbers of sites, users will either need to subset the sites or process the tree sequence in chunks and store results on disk.

It's also worth mentioning that our new code packs the input data to the size of the largest sample set. In other words, if we have two sample sets of size 10 and 16, but our tree sequence has 10^4 samples, our intermediate data structures will be sized to store up to 16 values instead of 10^4 values. Table 5 shows what our example from table 4 would look like after we pack it into our intermediate data structures. This packing is done by enumerating the samples in a sample set, then filling out the bit array row with the enumerated value instead of the sample value. This works because each sample set is independent in our computations. For instance, if we had a sample set with $\{0, 2, 4, 6\}$ and allele with samples $\{2, 3, 4, 5\}$, our packed bit array row would have bits $\{1, 2\}$ set (implying that bits $\{0, 3\}$ are unset).

S	A	SS	Count	Samples
0	0	0	5	`11111000 00000000`
0	0	1	5	`11111000 00000000`
0	1	0	5	`00000111 11000000`
0	1	1	5	`00000111 11000000`
0	2	0	6	`00000000 00111111`
0	2	1	6	`00000000 00111111`
1	0	0	8	`11111111 00000000`
1	0	1	8	`11111111 00000000`
1	1	0	8	`00000000 11111111`
1	1	1	8	`00000000 11111111`

Table 5: Precomputed accounting of alleles contained by each sample set packed into the size of the smallest sample set. The column names were shortened: S is the site, A is the site's allele, and SS is the sample set. The precomputed counts of the number of samples in each sample set are stored in another array labeled count.

To illustrate the improvement in computational complexity when we precompute these data, let's consider the example from above. Site 1 has 3 alleles and site 2 has 2 alleles. If we let $\boldsymbol{S} = \begin{pmatrix} 3 & 2 \end{pmatrix}$, then the base-case implementation would have $2 \sum_{i,j} (\boldsymbol{S}^T \boldsymbol{S})_{i,j} = 50$ operations (the factor of 2 comes from having 2 sample sets). When we precompute our allele counts and bit arrays, we see $2 \sum_i \boldsymbol{S}_i = 10$ operations. This disparity grows quickly because computational scaling to compute these data are $\mathcal{O}(n^2)$ in the base-case and $\mathcal{O}(n)$ with our improvements.

In addition to precomputing the allele counts and bit arrays, we also introduce a biallelic results function. In the original base-case, we use our general functionality for computing statistics in all cases, which means that we must compute the desired statistic for each pair of alleles in a pair of sites and combine them. For biallelic sites, this is not necessary and we end up performing redundant computations. For example, if we have a biallelic site, we would compute a 2×2 matrix of statistics (for the ancestral and derived allele), then we would combine and normalize these four statistics into one. With our biallelic results function, we simply compute our statistic for the derived alleles and return this result for our given site. This means we're doing 1/4 of the summary function calls and we forgo the need to perform any normalization. The full diff for these changes can be found here.

```
import "<<analysis-dir()>>/tskit.nix" {
  rev = "eb568405b371b8314e9ad6d4a1f928169b38b9eb";
  sha256 = "sha256-5nKsnoPg0MLzaiHPNEd6GMCesuWdwZmgawoxUN3EF54=";
}
```

```
mkdir -p bench
systemd-run --scope -p Slice=benchmark.slice ./run-bench > bench/final.tsv
```

From figure 10, we can see that the two-locus method is actually faster than the LD Calculator up until we hit an inflection point at around 4.5×10^3 samples. My interpretation of these results, after playing with some further optimizations is that the inflection point in the two-locus method comes from exhausting one of the larger (L1/L2/L3) CPU caches. We can see this in the flame graphs as well (figure 11 and figure 12). It seems that with few samples, the `ld_matrix` run time is dominated equally by the summary function and bit array counting. With sufficient data in the pipeline, we see that bit array counting becomes the bottleneck. My CPU has 16MiB unified L3 cache (shared between cores) and 1MiB L2 per core. My estimate of the size of the data we process in the tree with 4.5×10^3 samples is about 31 MiB $(4641 \times 3501 \times 2)/2^{20}$. Overall, it seems that the framework is not as cache friendly as it could be and the CPU cache cannot keep up. My understanding is that this stems from having a general summary function (which is a pointer to a function and therefore cannot be inlined).

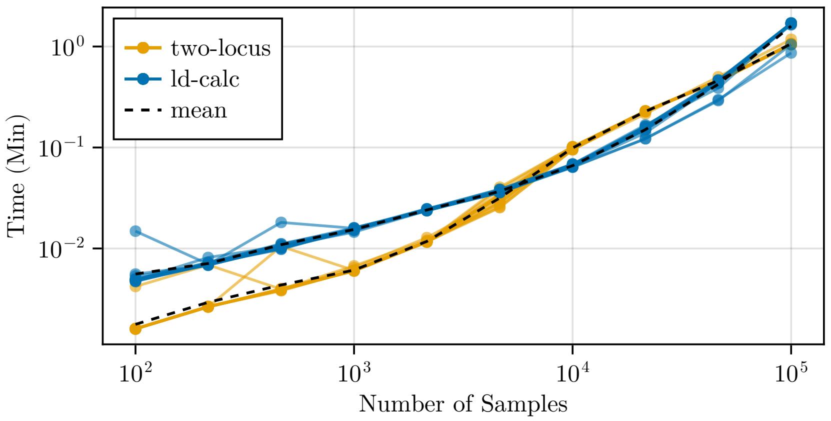

Figure 10: Total (wall) runtime of both methods with the precomputation of counts and a method for biallelic loci, layered on top of the previous (malloc and bit array) changes.

Because figure 10 shows an inflection point that suggests the two methods might be diverging after 10^5 samples, I created a series of larger tree sequences with a sequence length of 6×10^6 and sample sizes ranging from 10^3 to 10^6. The goal of this last benchmark is to observe the behavior of the two methods on a longer time scale.

```
gen_data(L=6e6, sample_sizes=np.logspace(3, 6, 10, dtype=int))
```

```
mkdir -p bench
systemd-run --scope -p Slice=benchmark.slice ./run-bench > bench/final-big.tsv
```

We can see that the runtime of the two-locus method on the larger tree

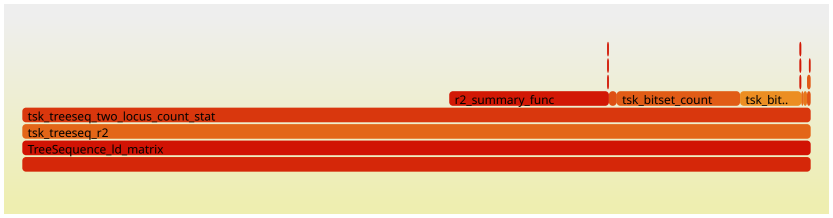

Figure 11: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 100 samples.

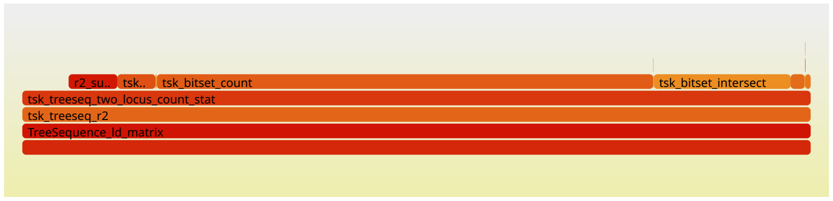

Figure 12: Flame graph of the `two-locus` framework with trimmed call stack, emphasizing all calls under `ld_matrix`. Run on a simulated tree with 10000 samples.

sequences scales linearly with the number of samples (figure 13). This roughly tracks with the performance of the LD Calculator, which has a dip in run time from $10^{3.5}$ samples to $10^{5.5}$ samples. It's difficult to explain why the LD Calculator has this sub-linear performance, but the two-locus framework does end up matching or exceeding the performance of the LD Calculator outside of this range (see figure 15 for the relative differences in detail).

4 Conclusions

It turns out that memory latency is an important factor in our optimization story. Given the way we've structured the code, we eventually become limited by the speed of loading data into registers. Despite these limitations, the relative performance of the two-locus framework ranges from about half the speed to twice the speed of the LD Calculator. The performance behavior depends on the tree topology, longer tree sequences appear to increase the relative difference in performance in both directions, though more testing would need to be performed with trees of varying sequence lengths to understand this behavior.

In figure 14, I show a comparison of the relative runtime difference between the two-locus framework and the LD Calculator, demonstrating the relative speedup that we obtain from each optimization. To recapitulate the results presented in the previous section, we see that the memory allocation optimizations mostly give us a speedup in tree sequences with fewer samples ($10^2 - 10^4$). The bit

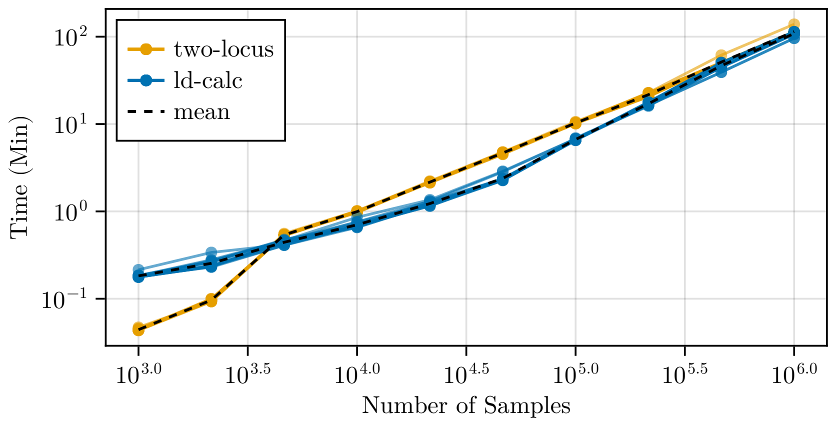

Figure 13: Total (wall) runtime of both methods with the precomputation of counts and a method for biallelic loci, layered on top of the previous (malloc and bit array) changes. Benchmark was performed on larger tree sequences with $L = 6 \times 10^6$ and sample sizes from 10^3 to 10^6.

array refactor improves runtime with larger numbers of samples ($\geq 10^3$). Finally, the precomputation of sample sets and counts in each allele set gives us the largest improvement, exceeding the performance of the LD Calculator in some cases, though the LD Calculator remains faster in tree sequences with $10^4 - 10^{4.7}$ samples.

Finally, figure 15 shows the relative improvement for all combined optimizations on the smaller ($L = 2 \times 10^6$) and larger tree sequences ($L = 6 \times 10^6$). From this plot, we can see that the LD Calculator remains more performant over a larger range of sample sizes and that the performance advantage of the LD Calculator is higher at the maximum relative difference in run time.

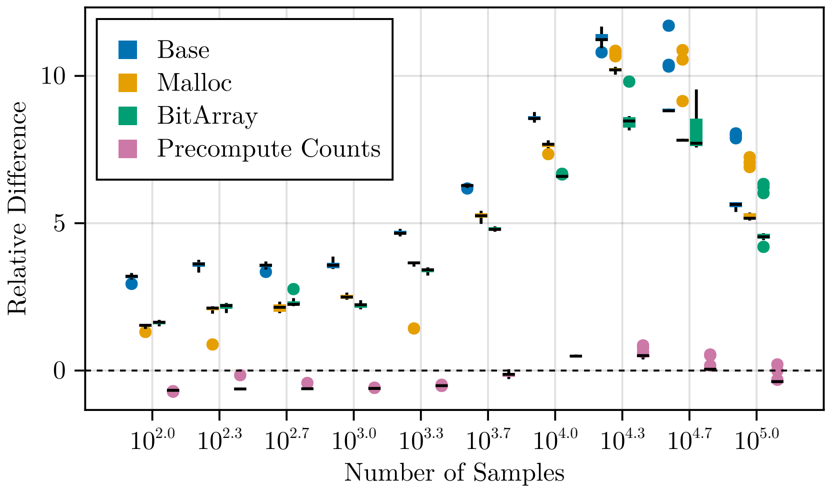

Figure 14: The relative difference in runtime for each method, compared to the LD Calculator. Relative difference was computed as (tl − ldc)/ldc where ldc is the LD Calculator and tl is the two-locus framework. As with previous benchmarks, each change is layered on the next.

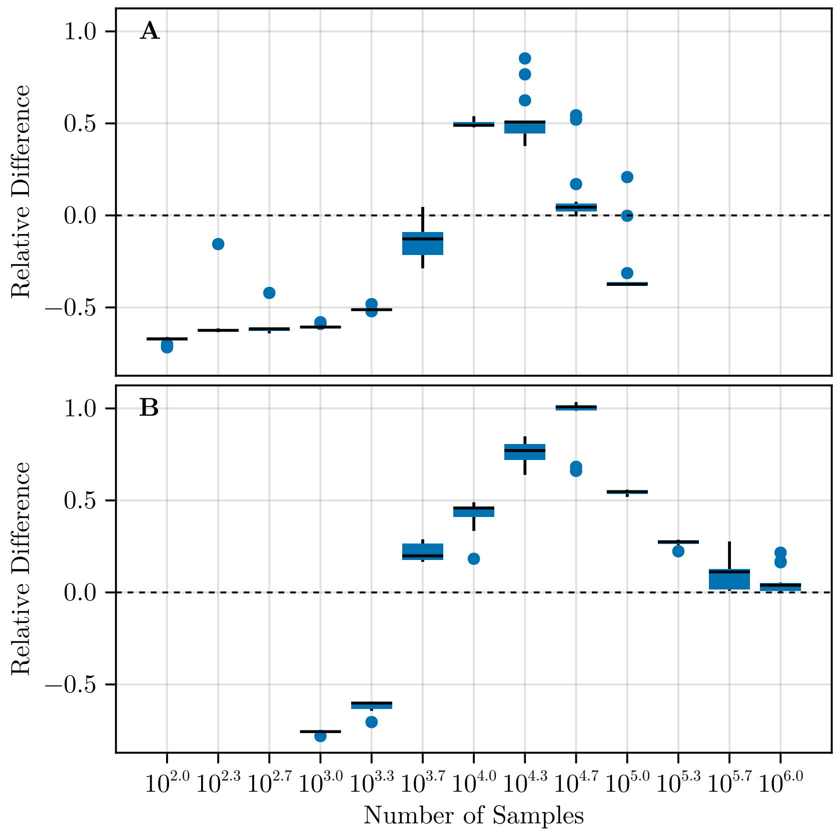

Figure 15: The relative difference in runtime for all combined optimizations, compared to the LD Calculator. Relative difference was computed as (tl−ldc)/ldc where ldc is the LD Calculator and tl is the two-locus framework. As with previous benchmarks, each change is layered on the next. Panel **A** shows the relative difference for the smaller tree sequences ($L = 2 \times 10^6$) and panel **B** shows the relative difference for the larger tree sequences ($L = 6 \times 10^6$).