Aalto University
School of Science

Master's Programme in Computer, Communication and Information Sciences

# Efficiency in Population Genetics Simulations

A tale of two memory representations in the simulator Nemo

**Lauri Ahlberg**

Master's Thesis
2025

**Aalto University**
**School of Science**

**Author** Lauri Ahlberg

**Title** Efficiency in Population Genetics Simulations — A tale of two memory representations in the simulator Nemo

**Degree programme** Computer, Communication and Information Sciences

**Major** Security and Cloud Computing

**Supervisor** Prof. Jukka Suomela

**Advisor** Prof. Frédéric Guillaume

**Collaborative partner** Helsinki University

**Date** 28 July 2025 **Number of pages** 48 **Language** English

**Abstract**

When population geneticists study how mutations pass through and accumulate in populations, computer simulations are increasingly being used in favour of (or in addition to) the traditional analytic approach. Forward-in-time simulations model genetics rather explicitly and as a result are often resource intensive to run. In this work I study the efficiency, in terms of the memory and runtime requirements, of a new memory representation of the genetics of a population in the simulation software Nemo. To this end I have implemented a new system for keeping track of mutations, based on an implementation found from the literature, which I have modified to integrate it with Nemo. I conduct measurements on the overhead of running Wright–Fisher simulations with varying parameter combinations on a supercomputer and present results. My findings indicate that the new implementation has a wider range of memory and runtime requirements, meaning that for some parameter combinations the simulation performance is improved, while for others it is worse. In the new implementation the memory required for running the simulations is determined by the size of a buffer needed to store references to mutations at segregating sites. This size is determined by the simulation scenario, but in this case the relationship is $O(x^2 \ln(x))$ in the number of individuals, and linear in both the number of simulated genetic loci and the mutation rate. The runtime is heavily dependent on the memory used, as there is extra bookkeeping involved in keeping track of an increased amount of mutations.

**Keywords** Genetics, simulators, efficiency, computational biology, algorithms, data structures

**Aalto-yliopisto**
**Perustieteiden**
**korkeakoulu**

**Tekijä** Lauri Ahlberg

**Työn nimi** Tehokkuus populaatiogenetiikan simulaatioissa — Kertomus kahdesta muistiesityksestä simulaatio-ohjelmassa Nemo

**Koulutusohjelma** Computer, Communication and Information Sciences

**Pääaine** Security and Cloud Computing

**Työn valvoja** Prof. Jukka Suomela

**Työn ohjaaja** Prof. Frédéric Guillaume

**Yhteistyötaho** Helsingin Yliopisto

**Päivämäärä** 28 July 2025 **Sivumäärä** 48 **Kieli** englanti

**Tiivistelmä**

Kun populaatiogeneetikot tutkivat kuinka mutaatioita ilmestyy ja kertyy populaatioihin, sekä poistuu niistä, tietokonesimulaatioita käytetään yhä enenevissä määrin perinteisten analyyttisten menetelmien sijaan ja niiden lisäksi. Ajassa eteenpäin katsovat simulaatiot mallintavat geneettisiä prosesseja varsin eksplisiittisesti, minkä seurauksena niiden ajaminen vaatii paljon laskennallisia resursseja. Tässä työssä tutkitaan uutta tapaa esittää esittää populaation perimää tietokoneen muistissa – erityisesti sen tehokkuutta sekä muistinkäytön, että ajoajan suhteen. Tähän tarkoitukseen olen toteuttanut Nemo simulaattoriin uuden tavan pitää kirjaa mutaatioista, joka perustuu kirjallisuudesta löytyvään aiempaan toteutukseen, ja josta olen muokannut Nemoon yhteensopivan version. Tässä työssä mittaan Wright–Fisher simulaatioiden ajamisen resurssivaatimuksia supertietokoneella vaihtelevilla parametriarvoilla ja esitän saamani tulokset. Löydökseni osoittavat, että uuden implementaation muisti- ja ajoaikavaatimukset kattavat vanhaa laajemman välin, mikä tarkoittaa, että joillakin parametrikombinaatioilla simulaation tehokkuus parantui mutta toisilla huononi. Uudessa toteutuksessa simulaatioiden ajamiseen tarvittavan muistin määrä riippuu siitä, minkä kokoinen puskuri vaaditaan, jotta voidaan säilöä viittaukset sellaisiin mutaatioihin, jotka ovat vain osalla populaatiosta. Vaadittava koko riippuu siitä, minkälainen simulaatioskenaario milloinkin on kyseessä, mutta tämän työn tapauksessa puskurin koko on $O(n^2 \ln(n))$ suhteessa simuloitujen yksilöiden määrään, ja lineaarinen suhteessa niin simuloitujen geneettisten sijaintien määrään kuin mutaatioiden todennäköisyyteen. Ajoaika riippuu vahvasti muistinkäytöstä, mikä johtuu siitä, että kun mutaatioiden määrä kasvaa, myös tarvittavan kirjanpidon laskennalliset kustannukset kasvavat.

**Avainsanat** Genetiikka, simulaatiot, tehokkuus, laskennallinen biologia, algoritmit, tietorakenteet

## Preface

This is a story about time. We humans have a peculiar relationship with time. We are bad at thinking about the future (as indicated by the psychological fallacy known as present bias [20]), we want the thing we want right now and we predict poorly future consequences of present actions. When studying processes that run beyond the limit of a human lifetime, we find them hard to grasp. This is equally true for scientists.

As history has shown, however, the effects of human actions have consequences that reach far and wide, to the point of wiping species off the face of the earth and altering the climate system of our home world. These facts are testament to the power that we as a species have over our environment, and highlight the responsibility that we have as a result. These are some of the reasons why I find biology satisfying – there are questions of remarkable consequence that one can hope to contribute to tackling.

As to how, well, this is also a story about time. Computer scientists often talk about the cost of an operation, referring to the time spent, and then go about trying to be as efficient as possible in scheduling their operations. We have certain tools at our disposal – for this work those tools are mathematical and computational – and we do the best we can to master those tools, hoping and praying that we are well equipped to deal with the challenges that are bound to lie ahead.

I want to thank Professor Jukka Suomela for reminding me of the role of storytelling in science. I feel that through our meetings you were able to clarify for me what aspects are essential for a project such as this and what to focus on.

Thank you to Professor Frédéric Guillaume for the idea for this project and all the support and feedback along the way. Your research group provided me with a sense of belonging and community which was invaluable in a solitary undertaking such as this project mostly was. My thanks for this also goes to Martti, Nick and Petri.

Finally, I would like to thank my friends and family for providing me with reasons and opportunities to occasionally spend time thinking about things other than the thesis – it was not a short project, so pacing is paramount, and optimisation exercises on scheduling in real life include a coefficient for mental well-being.

Helsinki, 26 July 2025

Lauri Ahlberg

# Contents

## Symbols and abbreviations

| | |
|---|---|
| DNA | deoxyribonucleic acid |
| SNP | single nucleotide polymorphism |
| FTPGS | forward time population genetic simulation |
| HPC | high-perfomance computing |
| CSC | (Finnish) centre for scientific computing |
| CPU | central processing unit, processor |
| RAM | rapid access memory |
| GB | gigabyte, $10^9$ bytes |
| GiB | gibibyte, $2^{30} = 1024^3$ bytes |
| AVX | advanced vector extensions |
| GSL | GNU scientific library |
| RSS | resident set size, i.e. RAM used |
| IPC | instructions per clock cycle |

# 1 Introduction

Genetics, on the level of populations, are affected by a myriad of different, interlinked processes. This makes it challenging to study questions of interest, but sounds like a case for mathematical models, which should capture just enough aspects of the real life complexity to yield reliable predictions, but no more – to make them interpretable and approachable. Recently, these models are more and more often implemented in software, as simulators. This brings with it a whole smorgasbord of choices about how to construct simulations that most efficiently use the resources available, both in terms of the utilisation of hardware elements, such as main memory, and time.

Some questions of interest that have been tackled by simulation studies relate to metrics of genetic resilience or the ability of species to adapt to climate change [16, 22]. These results can then be applied to aid in the decision-making in conservation efforts of endangered species [16].

Individual-based simulations attempt to mirror the real life biology fairly closely. Members of the population under study are represented in computer memory and each member is seen to have a number of genetic loci at which mutations may occur at a certain rate (per locus per generation). In addition, portions of these genomes may shuffle via recombination, where two chromosomes swap segments with a certain probability. This occurs between crossover points which appear between subsequent loci at a predefined rate.

Some ballpark estimates for typical values for simulation parameters would be: population size and number of simulated positions both in the order of $10^4$, mutation rate around $10^{-7} \ldots 10^{-6}$, recombination rate around $10^{-8} \ldots 10^{-6}$, and the number of generations that the simulation runs for in the order of $10^4$ [26] [8]. These naturally depend on what is being simulated, but such simulations can take from megabytes to hundreds of gigabytes of RAM (in the case of even larger populations), and may finish in minutes or run for days.

Because of these wide ranges of requirements, it is worthwhile to try to optimise resource use. Computation time is one such resource which has a price tag. There are never limitless pools of resources, monetary or otherwise, so optimal usage is key. Another resource of interest in the context of a computer is memory. Moreover, it is one which can at times be limiting with regards to what our program can do. Modern PCs have RAM in the order of 10 GB, supercomputer cores can have a few orders of magnitude more. For some of the parameter combinations in this work, these memory limits were reached.

There are a few different recipes for constructing a forward simulation. The fundamental difference amongst these is whether to look at it in terms of simulated loci or in terms of mutations accruing in a population. Some simulators, Nemo [6] among them, take the explicit, genetics-based approach and keep track of all the positions on all the individuals where a mutation may occur. The number of these is set when starting a simulation and limited by the amount of memory at our disposal. The benefits of this come in the simplicity of the algorithms for processes such as mutation and recombination. Producing new offspring comes down to copying relevant segments of the parental chromosomes. Another approach considers only the mutations present in the population. The benefit here is that one need not care about unmutated positions and that memory overhead is eliminated. In principle there is also no predefined limit to the number of positions a mutation may hit. The drawback comes in the form of the bookkeeping necessary to map mutations to their genetic positions and the individuals carrying them. This translates to more complex algorithms for the processes of mutation and recombination which incurs a greater computational overhead.

**Goals** In this Master's thesis these are the two levels we are moving among. My goal is to investigate how it might be possible to implement a more efficient simulation architecture, efficient in terms of memory use and runtime, which would enable the running of larger simulations on existing hardware and the running of current simulations in a shorter time. Drilling down to the lower level, my goal is to examine a simulation architecture from the literature, presented by Putnam et al. with the name clotho [21], applying it to the context of an existing simulation software Nemo [6]. To this end, I implement in Nemo the data structures and algorithms required by clotho in order to study the effects of the differing in-silica representations of genetic matter on the memory usage, runtime and scalability of simulations with a variety of different parameter combinations.

**Contributions** This work contributes to the grand old machinery of science results and information about the overhead (in terms of time and memory) of running simulations with the clotho-style memory representations. It also investigates the range of simulations it is most well suited for. Based on measurements conducted on the Puhti supercomputer over a range of different parameter combinations I observe the following trends. Unlike the old implementation, the requirements of the new are sensitive to the mutation rate. Other major determining factors are: the number of simulated loci, which exhibits a linear relationship to the memory and time requirements with both implementations; and number of individuals, whose relationship is linear in the old implementation, but approximately quadratic with the new. These dynamics have to do with the growth of the bitsets used to keep track of mutations when mutations percolate through the population. At high populations and mutation rates a larger buffer is needed to keep track of mutations, contributing to increased memory requirements. The runtime is directly related to the bitset size and the cost of recombination becomes dominating at larger population sizes. In the case of simulations with low mutation rates and fewer individuals, the new implementation is clearly lighter to run.

**Structure** The structure of this written work is as follows. In Section 2, I cover the necessary background from two different directions from which the interested reader may approach this work – that of a biologist and that of a computer scientist. I also discuss the overlap between these domains. In Section 3, I present the algorithms being investigated and the relevant internals of the simulator under study. I further lay the groundwork for the data collected by outlining the measurement process. In Section 4 I analyse the data and present the findings from the perspective of the two resources of interest, memory and time. In Section 5, I address the limitations of this work and suggest some future improvements and potential research directions for further exploration. Finally, Section 6 concludes the work with an overview of the main takeaways that can be gleaned from this project.

# 2 Background

Here I will aim to present a synthesis of background collated from multiple directions. There is very little existing literature out there about population genetics simulation software that would treat them from a technical viewpoint. Most publications are aimed at biologists whose primary interest is in the user-perspective.

This section is initially split into two main veins. Section 2.1 deals with the theoretical aspects from a biologist's perspective. I go into some detail about fundamental genetics, as well as the theory of how to study these processes at the level of populations. Readers with a background in (population) genetics can skim over this section, but I hope readers coming from the engineering side will find it relevant. Section 2.2, on the other hand, approaches the topic with a computer science perspective. I talk about the terms and concepts which I hope will be helpful in understanding how these kinds of topics are dealt with in that domain, and I hope that some of that way of thinking may rub off on the biology-oriented reader. I also discuss hardware-related considerations that affect efficiency. Readers versed in computational sciences may skim over these parts.

Finally, in Section 2.3 I take a look at the literature available about simulations. This combines the previous two aspects and serves as a backdrop for the work presented in this thesis.

## 2.1 Biology

**Genetics** Deoxyribonucleic acid (DNA) is the inherited material within each cell of an organism. Genetics is all about DNA. It is passed down from parents to offspring (in the case of sexual reproduction) and acts as the source defining the physical characteristics (phenotype) of said organism. DNA forms long strands which are made up of bases linked to each other via a deoxyribose sugar and its phosphate group. There are four such bases: adenine (A), thymine (T), cytosine (C) and guanine (G), and they pair up in such a way that A and T bind together and C and G bind together forming two parallel strands [15, Chs 1, 10]. This gives DNA its characteristic double helix structure.

The inherited data of the DNA consists of different sections, each having their own role to play in the makeup of the individual. The most familiar type of section being *genes,* those segments of DNA which contain the instructions for how proteins, the functional molecules in cells, are made [15, Ch 13]. The average length of a gene in the human genome is 25 000 basepairs [15, Table 21.1]. Genes can be split up into three-base segments, called *codons,* which each have a specific meaning in the processes of *transcription* and *translation* whose result is the final polypeptide chain of the protein corresponding to the gene in question. Most codons correspond to one of the amino acids which are the building blocks of proteins. Long sequences of these genes spaced by other sections of DNA coil up into units known as *chromosomes* in the cell nucleus.

In sexually reproducing organisms, the genetic material of the parent is not passed down directly to the offspring, but parts of it are shuffled in a process known as *recombination*. Most sexually reproducing organisms are *diploid,* meaning they have two copies of each chromosome (and thus each gene), one of which they have inherited from their mother and the other from the father. These pairs of similar chromosomes are known as homologous chromosomes, or *homologs* for short. These pairs are found in each parent, but the chromosome that they pass along to their child in the sperm or egg cell (known as sex cells or *gametes*) is in fact neither of these, but a hybrid combination of their two homologs. That is because during meiosis (the production of the gametes) there is a phase during which the two homologs line up and form *crossover points*, at which the chromosome which the DNA originates from switches to the other homolog. There can be multiple such crossovers between a single pair of homologs.

Another source of variation in the genetic material in an individual is mutation. This means changes in the DNA sequence, of which there are many different possible types. Sometimes mutation can alter the whole structure of a chromosome, for instance if entire segments of chromosomes are swapped as a result. Often these more drastic mutations can lead to dire consequences for the individual in question up to the point of being lethal. Smaller-scale mutations can include sections of DNA being repeated a number of times (known as *microsatellites*). Additions or deletions of one or more nucleotides can also occur. These result in all the nucleotides being shifted over. This can cause a frameshift, where the triplets forming codons no longer line up nicely. Also possible are cases where the length stays constant, but single bases are exchanged. These are known as a single nucleotide polymorphisms (SNPs). These mutations can sometimes change a codon to another one which corresponds to the same amino acid, resulting in no change, also known as a silent mutation. Or they can cause an amino acid to be swapped for a different one, known as a missense mutation. Or they can insert a codon which signifies the stopping of translation – a nonsense mutation. [15, Ch 15] [2, Ch 17]

Because of recombination, mutations in the parents' chromosomes are passed on to their progeny in a random way. This leads to mutations spreading out through the population in ways that are hard to predict as the processes of recombination and reproduction are repeated over subsequent generations. If a mutation has side-effects which are beneficial to the individual, it may be more likely to be passed on as that individual may succeed in having more children. Similarly mutations with detrimental effects (known as *deleterious* mutations) tend to spread less efficiently. However, the usefulness of a mutation is not always a clear-cut case as they may have multiple side-effects, some of which may be useful only under certain conditions.

**Population genetics** Having examined the chemical and molecular backdrop for the processes under study, let us shift our focus to the effects of these processes as they are studied on the level of populations and the tools used therein. Until recently the approach was quite heavily theory- and mathematics-based, which will be reflected in this presentation. The models aim to shed light on the process of evolution and how it plays out in natural populations. It is becoming common to also run the models on simulators, but having an idea of the theory helps us understand the simulations, their use and the expected results.

Population genetics is the field that concerns itself with studying how the amounts and proportions of certain gene variants, or *alleles*, present in the genetic makeup of a population change through time. With the recent advances in sequencing technology which have made reading of genomes cheaper and more widely accessible, it is possible to examine these things empirically in the cases of some organisms which have suitably short lifespans, and thus generation times, to allow for observation of these processes across the generations. However, the emphasis in this field throughout its history has always been in using the language of mathematics to describe the forces at play in changing the genetics of populations. Defining suitable sets of assumptions and constructing models that encapsulate the dynamics of interest in the situation while also managing to simplify things enough to make calculations tractable.

The classic starting point as presented by Gillespie [5] is the Hardy-Weinberg law which assumes that generations do not overlap, and mating in a population occurs completely at random (with respect to the locus in question) and that mutation, genetic drift and migration are non-existent. Furthermore, it only looks at a single locus in the genome with two possible neutral alleles, meaning neither allele has an effect on the probability of survival or reproduction compared to the other (i.e. they have the same *fitness*). If the alleles $A_1$ and $A_2$ have frequencies of $p$ and $q$, respectively (meaning these are the probabilities of finding those alleles in a randomly chosen gamete), then it is easily derived that at equilibrium, the frequencies of the diploid genotypes are $A_1A_1 : p^2,\ A_1A_2 : pq,\ A_2A_2 : q^2$. This model may be regarded as overly simplistic for most situations.

As we seek to refine our models to capture more of the real-world dynamics, we have to contend with the fact that most natural populations are distinctly finite in size. This makes it so that allele frequencies can fluctuate between generations purely as a result of random events and the small sample size, a phenomenon termed *genetic drift*. An example of such causes could be variation in the numbers of offspring between individuals due to external factors such as environmental conditions. A prevalent model that aims to incorporate random drift is known as the Wright–Fisher model. In this model, the population size is kept constant through the generations and the alleles of a subsequent generation are chosen randomly with replacement from the alleles of the previous one. The long-time effect of genetic drift is to reduce the genetic variation in a population, something which is especially important in the case of very rare alleles – such as new mutations – as these are rather likely to disappear, simply due to happenstance. According to the Wright–Fisher model the probability that two randomly selected alleles in generation $t$ are identical is

$$\mathcal{G}_t = \frac{1}{2N} + \left(1 - \frac{1}{2N}\right)\mathcal{G}_{t-1}, \tag{1}$$

where $N$ is the number of individuals. That is, either the two alleles have a common ancestor in the previous generation (probability $1/2N$) or then they don't but they are identical in generation $t-1$ (probability $\mathcal{G}_{t-1}$). This can be used to obtain a general expression for the probability that two randomly chosen alleles are different,

$$\mathcal{H}_t = \mathcal{H}_0 \left(1 - \frac{1}{2N}\right)^t,$$

where $\mathcal{H}_t = 1 - \mathcal{G}_t$ is that probability in generation $t$ and $N$ is the number of individuals. By this we see that since this probability is decreasing, genetic variation is being lost continually. What is important to note is that the rate of this loss is inversely related to the size of the population and smaller populations are markedly more affected. Of particular interest is the final steady-state case where one allele is established as the only one remaining in the entire population, it has become *fixed*. The fixation probability is given by $i/(2N)$ where $i$ is the number of initial copies of the allele. This is a remarkably simple result and applies to the case where all alleles are considered equally fit, possessing no selective advantage over another, known as the *neutral allele model*. [5]

If genetic drift is a force that removes variation from the gene pool, mutation injects variation back into it. Given $u$, the mutation rate for a single locus, we have $2Nu$ new alleles each generation. One common assumption is that all new mutations are unique, known as the *infinite-site model*, which may be justified if we think of an allele as consisting of a longer stretch of DNA, where a mutation may hit any base, meaning that there are a large number of possible alleles formed by mutation even in the case of SNPs. At equilibrium, when the loss of alleles due to genetic drift is balanced by the creation of new ones by mutation, it can be estimated that the probability that two randomly selected alleles are identical is

$$\hat{\mathcal{G}} = \frac{1}{1 + 4Nu}. \tag{2}$$

This result can be obtained from (1) via a Taylor series approximation and assuming $u/N \approx 0$. This equation tells us that when the value of $4Nu$ is small, most alleles are identical, meaning genetic drift is the dominant force. Whereas when $4Nu$ is large, mutation dominates and most alleles are different.

In the literature we can find an expression for the expected number of segregating sites in a steady-state population, that is, the of positions at which we have something which differs from the reference sequence. In his 1975 paper, Watterson [25] gives an estimator for this in a Wright–Fisher model as

$$E(K_2) = \theta = 4Nu \tag{3}$$

within a diploid individual and for the whole population as

$$E(K_{2N}) = \frac{1}{2}\theta(2\ln(2N) + 1.355076). \tag{4}$$

## 2.2 Engineering

**Computer Science prerequisites** In order to talk about the methods and algorithms in this project in a meaningful way, it is necessary to introduce some concepts which may be familiar to those with more background in computer science.

When assessing algorithms, we are generally interested in the runtime – how long they take to complete their task. When discussing runtimes it is useful to have a way of comparing algorithms to each other that gives a clear idea of how long each of them takes to run as well as relating this to the principal factors affecting runtime, which is often simply the input size. The runtime boils down to how many operations a particular algorithm must complete in order to finish (I/O operations such as reading and writing to/from the disk or just mathematical operations of combining and updating the contents of registers and/or cache memory [23]). A standard way of quantifying the time complexity is by using the big O notation. The point here is to give a reasonably accurate description of how the runtime scales as a function of say, the input size (or a certain subset of the inputs). Only the highest order term in the function is considered as it will dominate the runtime and any constants possibly present in the function are ignored. Often, if the time it takes for the algorithm to terminate varies according to, say, the structure of the input, the worst-case running time is considered as this gives a guarantee of maximum runtime. However, trying to estimate the average runtime complexity can be useful as well. This leads to different classes of complexity, such as linear $O(x)$, quadratic $O(x^2)$, exponential $O(2^x)$, logarithmic $O(\log(x))$ and so forth. [1, 3]

With regards to the performance of data structures, the thinking is very similar. We are interested in the number of times we have to access the elements of a data structure in order to find the one we are looking for or to be able to update a certain element in there. For instance, a linked list is a data structure where each element contains a reference to the next one. This means all accesses to elements further down the list from the first element, the head, have to pass through all the intermediate references. Thus worst-case lookup cost is that of the last, $n$th element, for which we have to go through $n$ references. This gives us a cost of $O(n)$. Contrast this with fixed length arrays where we can access each element directly just by offsetting from where the list begins, effectively an $O(1)$, constant-time operation. Updating or inserting new elements into a data structure may also involve an extra computational cost if some elements have to be moved/shifted/copied in memory. One more data structure to introduce here as it bears some similarity to the scheme in clotho is a hashmap. This is one way of achieving a dynamically sized data structure, via indirection, by having an intermediate function that is computed each time a lookup is initiated and gives the address of the desired datum. Here each element has a corresponding *key* which is linked to the element via a hash function. The cost of lookup is thus constant but you have to account for the extra cost of computing the hash. There are some different ways to define hash functions and handle the memory allocations as well as addressing in this case, but they are not important now.

A central concept in this work is that of efficiency. A general definition to work off of could be: achieving the desired result by using minimal resources. This raises the important question of what is meant by a resource. Memory, which for this work I will take to mean the main memory, RAM, in the CPU (as opposed to the hard disk or solid state drive, which I would call storage), is an obvious candidate for a resource, and indeed one which we are interested in. Another one, perhaps somewhat more subtle, is time. With regards to time, simply comparing the number of operations performed often misses the whole picture. This is because different operations may have different latencies, with some operations requiring something to be done elsewhere, incurring a waiting time. Indeed, something like (machine language) instructions per second is a better (though still limited) measure of efficient processor utilisation. This raises the important issue of scheduling the operations in such a way as to achieve maximum *throughput*, the amount of work that passes through a system in unit time. We will touch on this when discussing caching soon, but suffice to say that scheduling is a fundamental theme in many computer science applications, and it is worthwhile to optimise the order in which operations are performed while making sure the results are consistent. Drawing in both these resources, memory and time, I would like to note that there exists a fundamental trade-off between them, a trade-off between storing and computing results. If we have extra memory which is not needed to do computations, we can often save time by using that memory to store some results or intermediates which can be reused. On the other hand, if we want to save memory, we can compress and pack things down to smaller sizes, but that requires that when we need those objects, we have to spend extra time decoding them.

Before we can go into more details about how to go about efficiently using time and memory, we need to have a grasp of the tools we are working with, namely some understanding of the structure of the processor. Modern CPUs (meaning something released in the last 15 years or so) even on consumer laptops generally have multiple cores. The structure of each core is similar, and it consists of a number of different components. Some details can be found in Intel Developer's manuals [12]. For example in the Intel 64 architecture, there are 16 general purpose registers for 64-bit values, and units that operate on those registers. It is also becoming common to have several vector registers which can hold several 64-bit values, 4 in the case of the AVX-instruction set, and operate on all these simultaneously. In this work we are interested in two aspects. Firstly, that the units processing register values are actually pipelines with the possibility of doing multiple operations in sequence. There are several pipelines which may differ slightly in the sequence of operations they support. Secondly, the processor contains integrated cache memory on multiple levels. Taking for example the Intel Sandy Bridge Microarchitecture, each core has its own Level 1 instruction and data caches at 32 KiB each, as well as a 256 KiB Level 2 cache. On top of these, there is a Level 3 cache (sometimes called Last Level Cache LLC) which is shared between cores and could fit 4 MiB. [11]

The CPU pipelines allow for a form of parallel processing, namely instruction level parallelism. By parallel processing I mean executing multiple instructions at the same time, in parallel, resulting in greater throughput. The CPU pipelines consist of sequences of hardware operations such as additions or multiplication of two registers, where each step in the sequence can be executing simultaneously. In effect, we are working on multiple sequences at once, but just in a staggered manner as visualised in Figure 1. As a result, over time such a pipeline achieves a throughput many times greater than doing the same operation for all the data being processed before moving on to the next operation in the sequence, so long as the sequence of operations required for multiple data is more or less the same. This is already true for a single pipeline, and if there are several, the effect is naturally greater. Instruction pipelining is something

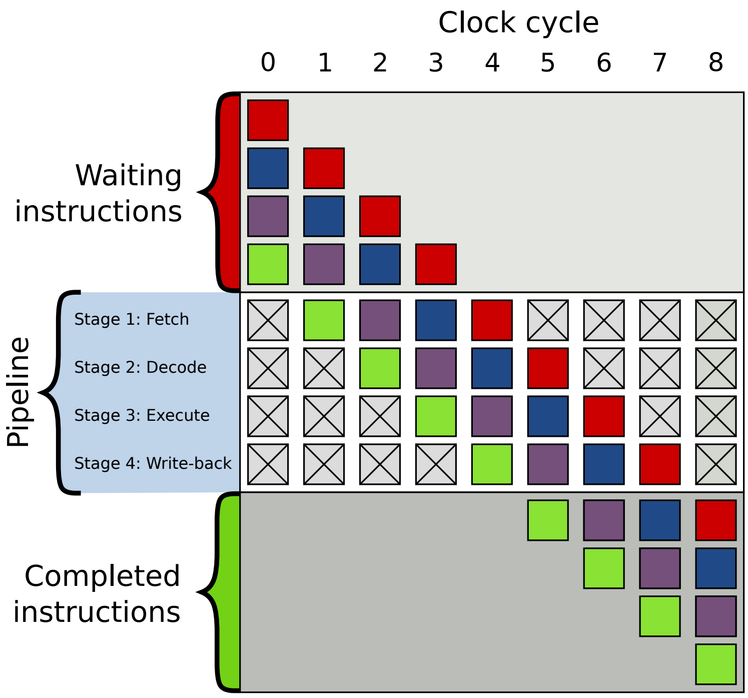



**Figure 1:** Diagram of pipelined processing of four instruction sequences. It takes only seven clock cycles to process all four four-operation sequences.
image by CBurnett / Wikimedia Commons CC BY-SA 3.0 [4]

over which the programmer does not have direct control as this is something that is handled automatically by the compiler. The way to approach this is to write such code that it can be pipelined.

As mentioned, modern processors have a cache hierarchy with multiple levels of memory. This is arranged so that the smaller caches are closer to the processing units and caches get larger as they move farther away. This means that the smallest caches are faster to access while reading from, or writing into, the larger caches is slower. The time for a cache access can grow by an order of magnitude when moving to the next level in the hierarchy [1, Ch 4]. The RAM can also be thought of as the final level of this memory hierarchy. This helps us understand why it is so important to keep in mind the memory access pattern of a program, as this has a huge impact on cache usage. Keeping things in the closest cache possible while we do operations involving that data is much faster. This means we want to group those operations acting on a single piece of data close to each other in the execution order of the code. This applies somewhat also to consecutive pieces of (small) data as what is brought into the cache are so-called cache lines which reflect the structure of the data in the computer's memory, not our way of grouping it [1]. In effect, data which are next to each other in the memory are brought into the cache at once. Thus from a scheduling viewpoint it is also worthwhile to operate on nearby data in rapid succession so they are more likely to be found in the cache, eliminating the extra overhead and waiting-time brought on by having to re-fetch those portions from memory.

## 2.3 Simulations

Having looked at the biological and computational backdrop for this work, let us finally shift our focus to simulations. We have established that there are complex biological processes that are scientifically of interest to study. Historically, this has been done mainly through the use of mathematical, analytical models. This approach, however, requires a mathematical formulation to be constructed, and further, a formulation that we can work with to derive useful properties of the system under study. As the complexity of the desired system under study increases, the models become more difficult to work with, eventually growing to such heights of complexity that there are no known ways to solve for parameters of interest analytically. But we have computers which can be instructed to do large numbers of mathematical operations within a short timespan, and this opportunity has been harnessed in simulations. It is still necessary to describe the system under study mathematically, but the requirement is lifted that the description need be analytically solvable. This is the rationale behind creating simulations [13].

There are a number of situations where the simulation approach is employed. In [10] Hoban et al. list three possible cases. Simulations can be used to predict a future state of affairs by starting with the current-day situation, making some assumptions about the processes at work and the values of the parameters controlling said processes. Then one runs a corresponding simulation to see how things are expected to turn out at a later time-point. Simulations can also be used to attempt to determine something about a currently unknown part of the system under study. For example, if we have a snapshot (or several) of the process state in the form of some real-world data and we have a theory about the processes that generated the data, but we do not know some parameter values of those processes, we can run simulations of our theorised processes with a range of different parameter values. The results of the simulations can then be compared to the real world data to determine the unknown parameter values. Thirdly, simulations are helpful in developing new statistical analysis methods. If there is a method for estimating a parameter value based on some measurement data, a simulation can be run to generate data in the same format, but which has come from a process that the user has complete control over, meaning that they know the ground truth values for the parameters that the statistical method gives estimates for. This can be used to assess the accuracy, precision and robustness of the analysis method for different parameter values.

In the literature, we find that the simulations-scene has been divided into a few different classes. The two main veins are so-called coalescent (or backward) simulation, and forward simulation. On top of these, some authors distinguish sideways simulations [17] and resampling simulations [27], with significant overlap between these last two types. More recently there have also been hybrid approaches employed which combine coalescent and forward simulations [13]. Resampling and sideways simulations differ from the others in that they require the user to have access to an adequately sized set of measurement data points, as those are used to generate new data, as the name suggests. Coalescent simulations are often less computationally intensive, and as a consequence, require less resources (time and memory) to run [27]. They are based on the theory of coalescences [14] which deals with going backwards in time from the current state to find time points at which genealogies of currently living individuals diverged from a common ancestor. Forward-in-time simulations, on the other hand, make very few assumptions and aim for the greatest degree of correspondence to the actual biology. It is easy to understand that this makes the simulations sometimes rather heavy since the natural processes being mimicked on the computer are rather complex and they are often represented in the simulation quite explicitly. However, they have the benefit of being able to deal with, in principle, arbitrarily complex scenarios under investigation, also those which are analytically intractable.

In [9] we find an example of a coalescent simulator being used to assess sampling coverage necessary to preserve genetic diversity in external collections (such as seed banks or gardens). Nemo has been used to study the accuracy of metrics that are used to assess the genetic resilience of small populations of endangered species [16]. Another source [22] presents results from simulating species with differing life-history strategies under different scenarios of environmental change and fluctuation.

# 3 Methods

The aim of this section is to lay out the practical side of the project. I will begin by presenting the algorithms used to simulate the genetic events (mutation and recombination) that change the makeup of the gene pool. Then I will go through an overview of the relevant parts of the simulator implementation and the new parts added as a result of this project. Thirdly, I will outline the measurements that were conducted to quantify the effect of using the new implementation and describe the measurement process.

## 3.1 Algorithms

Let us begin by looking at the algorithms in isolation from the implementation details, which will be dealt with in the next subsection. There are two processes of interest here, namely mutation and recombination. In the original algorithm, all genetic data is explicitly represented and stored for each individual. Thus mutation is simply a case of updating that memory in the correct position. The number of mutations is drawn from a binomial distribution for each individual with parameters $[p = 2\mu, n = l]$ where $\mu$ is the mutation rate and $l$ the number of loci being simulated. Also randomly selected is which simulated locus the mutation hits, and on which homolog, as the simulated organisms are diploid.

Recombination, on the other hand, is a case of copying segments of the genetic data to the new individual being created. Which segment to copy from which homolog is determined by the crossover positions. These are also randomly determined. The number of crossovers per individual per generation is drawn from a Poisson distribution with the rate parameter $\lambda = lr$ where $r$ is the recombination rate.

All this storing and copying has costs, both in terms of memory and time used, as updating large portions of the main memory is relatively slow. The new implementation aims to cut down on these costs. The idea is to achieve this via indirection – lets not store everything explicitly, potentially multiple times, but instead try to only keep one copy in memory and then have (potentially multiple) references to that information. The information in question here are the mutations. And the referencing scheme is via bitsets, strings of ones and zeros where each bit position corresponds to a centrally stored mutation. Each individual then just needs two such bitsets, one for each homolog, and a set bit indicates that the individual carries the corresponding mutation.

The process of mutation in this new way of thinking, is mostly a question of bookkeeping. For each mutation, it involves creating the mutation to be stored centrally, and updating the bitset of the individual to refer to said mutation. An optimisation which we shall see later involves recycling old mutations to avoid the cost of creating new ones each time. The number of mutations and the positions hit are drawn as in the old implementation.

Recombination in this case is somewhat more complex and is outlined in Figure 2. Each individual, being diploid, contains two bitsets, one inherited from each of its parents. I will call these the maternal and paternal bitset. Now recombination involves

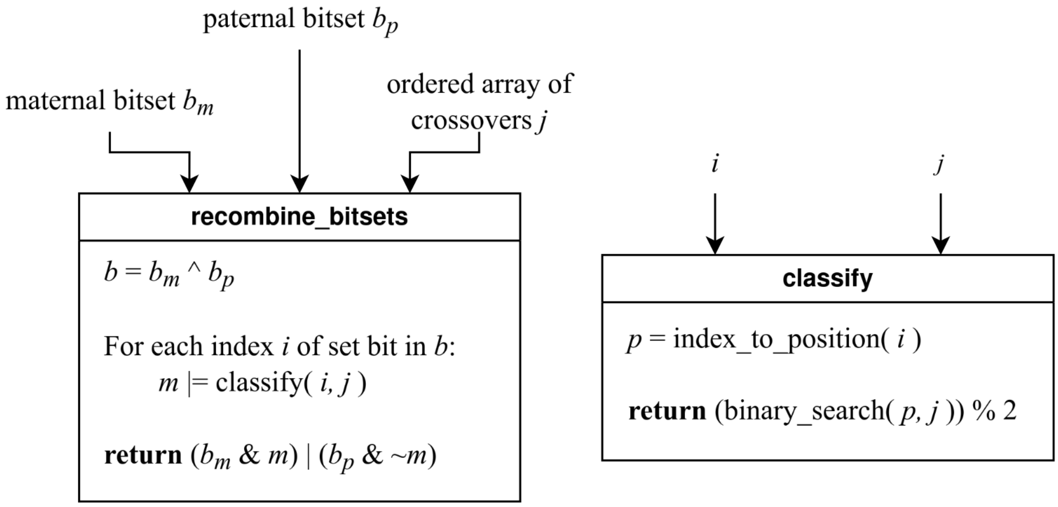



**Figure 2:** Overview of the steps of the new recombination algorithm. Here ~ is the binary negation and operations `xor`, `and`, `or` are denoted by ^, &, |. The modulo operation is %.

transferring the correct mutations to the gamete being created. Those mutations that occupy positions on segments coming from the maternal DNA should be included and mutations in those same positions in the paternal DNA should not, and vice versa for segments coming from paternal DNA. Because the references in the bitsets are not ordered by genetic position, essentially the whole bitsets have to be scanned, and mutation positions looked up, in order to know which mutations should be present in the gamete, i.e. which bits should be set. This is a question of knowing, for each mutation, how many crossover positions there are in the genome before its position. All the mutations which are preceded by an even number of crossovers should come from the same side (maternal or paternal) and the ones after an odd number of crossovers should come from the other side. It is possible to get this information by a binary search over all the (sorted) crossover positions, which gives us the index at which the position of the mutation would fit. The parity of this number then tells us if this mutation is inherited. An optimisation that can be done here, however, is to only check those bits where the two bitsets of the parent differ, as it makes no difference which one to include for those positions where both bits are set. This check can be done by creating a mask indicating those bits where there is a difference (a simple binary `xor` of the two bitsets) and only checking the descent of those bit positions. This checking can be thought of as a classification operation telling us which bitset the differing positions should be inherited from. To create a new individual it takes two gametes, so this procedure needs to be done twice, once in each of its parents.

## 3.2 Software structure

I will start with the user viewpoint, then go briefly into the genetics-perspective before delving into the technical implementation details. In a Nemo simulation,

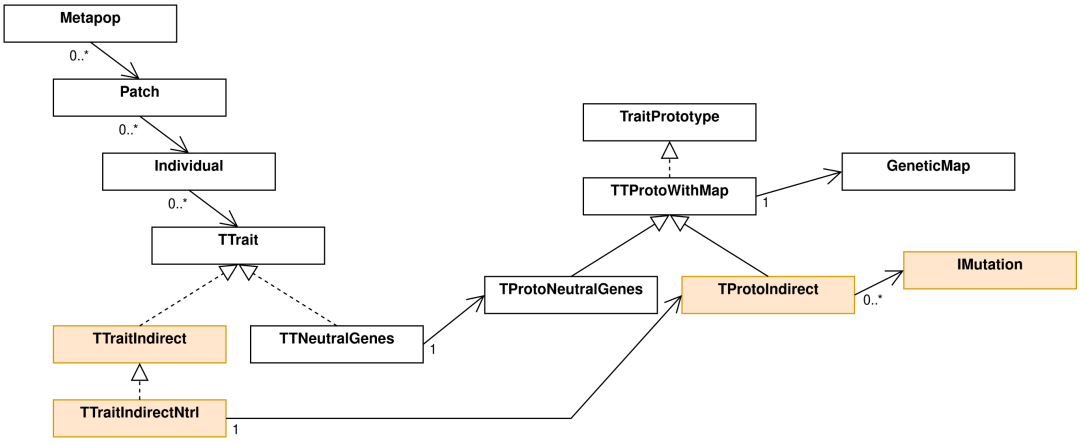



**Figure 3:** Class diagram of the relevant Nemo classes, new classes highlighted

there is always one population that is being changed. The population is spread out over a number of patches, each consisting of a subset of all the individuals in the population. The software allows for migration – moving individuals between patches – but those capabilities are not used in this project. Individuals can reproduce by sexual reproduction, where the genetic data of the produced individual is a mixture of that of its parents, by selfing, where individuals self-fertilise meaning all the genetic data is from the same individual but with recombination, or by cloning which is similar to selfing but without recombination. Individuals can also have a fitness value determined by their genetics which affects the reproduction, but fitness is also not dealt with in this project.

We can also look at all of this from a genetics point-of-view. All the genomes being simulated have a shared genetic map, which determines which chromosome a locus sits on and the distances between them. Genetic elements can be grouped under traits, which consist of a set of loci (positions on the map) and possibly allelic effects. Individuals may contain several different types of traits at once. The genetic map is shared between all the traits and it ensures that during recombination all the traits in the individual have the crossovers at the same positions.

The C++ implementation of all this is object-oriented, meaning the software consists of different classes, which each have their own responsibilities in running the simulation. When the simulation is running, there are (possibly multiple) instances of each class in the CPU memory. The parts of this structure that are relevant for this project are presented in Figure 3. We can see that the genetic map mentioned previously has its own class GeneticMap, and it is coupled to the Trait class via the trait prototype class. There are several different trait types (subclasses). For the sake of this project, the population (class MetaPop) and the individual are just containers for individuals and traits, respectively.

The starting point for this project is a Trait. More specifically, a neutral trait. In the original implementation of a neutral trait the information about the genetics of the trait is stored in two fixed length arrays. The length of these arrays is $l$, the number of loci, and the values are (8-bit) chars. Thus the implementation of mutations is simply an update to the array value at the correct position. Recombination is just a copy of the relevant segments of the arrays into the arrays in the new trait instance (belonging to the new individual being created). The segments are determined by crossover positions (at which the side being copied from switches) which are drawn in the GeneticMap. During a running simulation there are as many instances of each trait as there are individuals. Each trait type also has a corresponding prototype class, of which there is just one instance present and in the original implementation this is only used to create new trait instances.

To be able to run the new algorithms for mutation and recombination, I have implemented a new trait type, TTraitIndirectNtrl in Figure 3. This indirect trait type acts as a container for the bitsets, which hold the information about which mutations an individual has.

As described in the previous subsection, the mutations themselves are stored centrally. Each mutation is an instance of the class IMutation which knows its own genetic position, and the new trait prototype class, TProtoIndirect, is responsible for holding the references to the mutation objects. These are stored in data structures that can be used for for mapping between bitset positions and the corresponding mutations and their genetic positions. The references of bitset position to mutation are stored in an `std::vector`. Because there can be multiple mutations that have hit a single simulated locus (as long as these occur on different individuals), the mapping from locus to mutation is a hash-map of unordered sets, whose keys are the loci positions and values the sets of references to mutations.

During a simulation run, some mutations may become fixed in the population, meaning that all the individuals have them. At this point, storing this information in every individual would be redundant, and the corresponding bitset position can be freed for reuse. The information about the fixed mutations can then be stored centrally, in the prototype. This is done by storing a population level reference sequence, an `std::vector<bool>`, where each boolean indicates whether this position contains a fixed mutation, and a list of references to fixed mutations.

Likewise bitset positions corresponding to lost mutations – mutations that are no longer present in any individual – can be reused. The information about these positions is stored in the prototype, as well as a recycling pool of the mutation objects created previously for fixed or lost mutations.

The procedures for doing mutation and recombination in the new implementation are also handled in the prototype, which then updates the individual bitsets accordingly.

### Implementation details on the algorithms

Both versions use the GNU Scientific Library (GSL) implementations for drawing random variables from the binomial and Poisson distributions.

In the new implementation, mutation is restricted in such a way that if the individual already contains a mutation in the position that is drawn for a new mutation (whether this old mutation is one that has fixed in the population or not), the new mutation is simply discarded. If the position for the new mutation is free, however, first the recycling pool is checked for reusable, previously lost or fixed mutations. If there are reusable bitset positions and mutation objects those are used. If not, a new IMutation instance is created and a new bit is added to the end of the union of all the bitsets in the population. Finally the mappings are updated accordingly.

Recombination is the most complex and computationally costly part of the new implementation. I used the clotho [21] source code as a guide for my implementation where the bitsets are internally represented as vectors of 64-bit blocks. This means that the masking and classifications can be done for each block separately, which results in a better memory access pattern as all the operations where a bit block is needed are close together in the order of operations performed. This means we are likely to benefit from data locality as these values can be kept either in the processor registers or top-level caches, reducing lookup times.

In the classification procedure I use an implementation of binary search available in the C++ standard library (`std::upper_bound` [24]).

The benefits of this new memory representation using bitsets is twofold. Because not every simulated position needs to be held in memory, this will result in a lower memory usage, especially for simulations where the number of loci being simulated is large and the mutation rate is low. This would mean that the genomes will have sizeable swathes devoid of mutations (where they correspond to the population-level reference genome) which will not be represented in memory. Indeed the memory taken by this architecture grows as mutations are accumulated in the population, not in relation to the number of map positions being simulated for the trait. Especially those cases with relatively few mutations, and thus smaller bitset sizes are also going to be fast to simulate because we only have to check the mutated positions at the time of recombination.

The bitset representation can also be beneficial when calculating statistics on the population as for instance the number of mutations in a genome is just the number of set bits in the bitset.

## 3.3 Measurements

To quantify the efficiency of the new implementation, I conducted a few different sets of measurements. The simulation context chosen for all the measurements is a Wright–Fisher simulation. In practice, this means neutral mutations (no fitness-effects), non-overlapping generations, hermaphroditic (self-fertilising) individuals mating at random, and a constant population size. This is a fairly standard model for population genetics studies and it is simple enough that we can see the effects of changing one parameter value more clearly. It is straightforward to implement in the software and there are less random fluctuations than in more complex scenarios since the population size is kept constant. This should enable clearer comparisons between the different architectures, allowing for more reliable observations of the effect of changing the internal architecture on the efficiency of the simulation.

The main set of measurements were conducted on the high-performance computing (HPC) supercomputer Puhti of the Finnish IT Center for Science (CSC). Running simulations and measurements on Puhti allows for testing a larger range of simulation parameters as some of the larger simulations would be infeasible on even a good home computer due to the memory requirements. Puhti consists of 682 nodes, where each node has two Intel Xeon Gold 6230 CPUs with 20 cores. Each core runs at 2.1 GHz frequency, but for the memory there are a few different configurations ranging from 192 GiB to 1.5 TiB. Jobs on Puhti are run using the Slurm scheduling system, where the possible nodes have a maximum of either 373 GiB or 1496 GiB memory available. [19]

Slurm collects accounting data about jobs submitted to the system, so we can use those data to monitor resource usage. I have access to sampling data on the memory usage of the jobs as well as the total times each job took to run. Since these simulations run for thousands of generations and the memory requirements do not change erratically throughout a simulation run, the sampling-based data collection yields reliable results. For the shortest simulations the sampling frequency can be increased not to miss the peak memory usage. The underlying linux utility used to gather information is `procfs`. [18]

The version of Nemo that was used for the measurements was compiled using cmake version 3.26.5 with g++ version 11.3.0, using the most aggressive optimisation level `-O3` and additional options `-funroll-loops -g`.

The simulation space that was explored for this project is four-dimensional. I am testing multiple values for population size (number of individuals $n$) and number of loci $l$ simulated for each individual as well as different values for mutation rate $\mu$ and recombination rate $r$. The parameter combinations come from the Cartesian product of these sets:

$$\begin{aligned}
\text{Let } N &= \text{set of population sizes} && = \{10^3, 10^4, 10^5, 10^6\},\\
L &= \text{set of loci numbers} && = \{10^4, 10^5, 10^6, 10^7\},\\
M &= \text{set of mutation rates} && = \{10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}\},\\
R &= \text{set of recombination rates} && = \{10^{-8}, 10^{-7}, 10^{-6}\}.
\end{aligned}$$

The full space covered by this is $N \times L \times M \times R$, which given the sets above would give us a total of 192 datapoints. However, some of the largest simulations (with the highest numbers of individuals and loci) were not run, as the trends were clear already based on the smaller simulations. Table 1 presents the simulation sizes in terms of total loci in the simulation. The greyed out sizes are simulations for which measurements were not collected.

All in all, simulations were run for 120 different parameter combinations. All the simulations were set to run for 10 000 generations, which is a fairly standard burn-in simulation length for population genetics studies. This means that we expect the population to have reached a steady state, where the rate of incoming mutations is balanced by those being lost randomly (through genetic drift). For the bitset representation we expect the bitset sizes to have more or less stopped growing, meaning the program has reached its peak memory usage.

Table 2 presents estimates for memory usage and runtime of the original implementation extrapolated from some preliminary, shorter simulation runs. In this implementation the rough scale of the memory requirements can be estimated based on

| | | number of loci | | | |
|---|---|---|---|---|---|
| | | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
| number of individuals | $10^3$ | $2 \cdot 10^7$ | $2 \cdot 10^8$ | $2 \cdot 10^9$ | $2 \cdot 10^{10}$ |
| | $10^4$ | $2 \cdot 10^8$ | $2 \cdot 10^9$ | $2 \cdot 10^{10}$ | $2 \cdot 10^{11}$ |
| | $10^5$ | $2 \cdot 10^9$ | $2 \cdot 10^{10}$ | $2 \cdot 10^{11}$ | $2 \cdot 10^{12}$ |
| | $10^6$ | $2 \cdot 10^{10}$ | $2 \cdot 10^{11}$ | $2 \cdot 10^{12}$ | $2 \cdot 10^{13}$ |

**Table 1:** Number of total simulated loci, equivalent values found on the upwards slanted diagonals. Runs corresponding to grayed-out cells were not measured.

| $2nl$ | Memory | Runtime |
|---|---|---|
| $2 \cdot 10^7$ | $\approx$ 10 mb | $\approx$ 1 min |
| $2 \cdot 10^8$ | $\approx$ 100 mb | $\approx$ 10 min |
| $2 \cdot 10^9$ | $\approx$ 1 gb | $\approx$ 1 h |
| $2 \cdot 10^{10}$ | $\approx$ 10 gb | $\approx$ 10 h |
| $2 \cdot 10^{11}$ | $\approx$ 100 gb | $\approx$ 100 h |
| $2 \cdot 10^{12}$ | $\approx$ 1000 gb | ??? |

**Table 2:** Estimates for classes of resource requirements (old implementation)

the total number of loci being simulated ($2nl$). This is because every locus is explicitly stored in memory, meaning also that memory use is rather constant throughout a simulation run.

The situation with the new trait architecture that I have implemented is more complex, as the memory use increases throughout the simulation as more mutations are accumulated in the population (and in the bitsets). This can be seen in Figure 4 which shows the growth of the bitset size, this being the main determining factor for total memory use. However, the user has some level of control over this increase as the bits corresponding to mutations that have become lost from, or fixed in the population can be reused to keep track of new mutations (see Section 3.2 for details). This requires a check for, and freeing of these bit positions, a procedure which I will refer to as *compacting* the bitsets. This is done periodically throughout the simulation and the user has complete control over when this is performed.

Compacting does have its own computational overhead, but it is not very significant. All the measurements were conducted with the same compacting interval, which I selected based on a few simulation runs that I did comparing memory requirements and runtimes for different compacting intervals. These measurements are plotted in Figure 5. The interval chosen for the measurements was 2 as this should result in a low memory usage while being close to optimal in terms of the time-overhead incurred.

From the population genetics literature as presented towards the end of Section 2.1 we can get some estimates for the lengths of bitsets for populations that have reached a mutation-drift balance. If we think of each simulated locus as a longer stretch of DNA which can exhibit many different alleles, the bitsets just store all the segregating sites for each of these stretches. This means that we can expect the bitset sizes to be

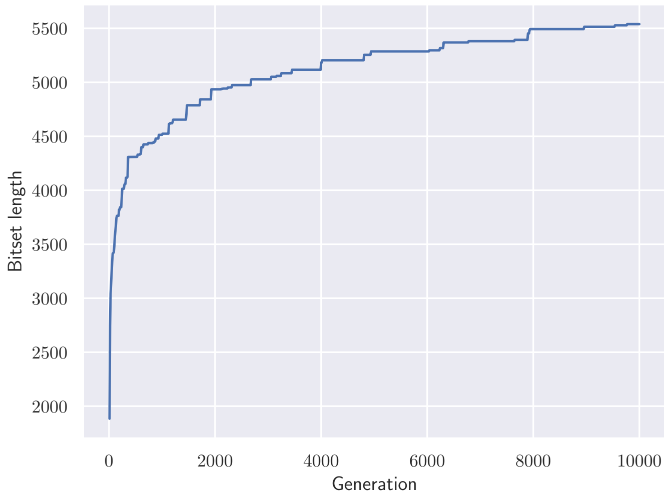



**Figure 4:** The growth of bitset sizes as the simulation progresses.
Parameters: $n = 10^4$, $l = 10^4$, $\mu = 10^{-6}$, $r = 10^{-5}$, compaction at every 10 generations. The main takeaway here is the shape of the curve – that bitset length seems to approach an equilibrium value.

roughly the number of simulated loci times the number of segregating sites in the population per locus from equation (4), so $l \cdot E(K_{2N})$. For this to hold exactly the bitsets would have to be "minimal" and hold no empty positions corresponding to lost or fixed mutations. This is of course not the case (and not even desirable) as we have a buffer of reusable bit positions in the bitsets, but the size of this buffer can be kept small with frequent compacting.

For the parameter space being explored in this project, the estimates range from approximately 3 bit positions to over 6 million positions per bitset. These numbers would have to be multiplied by $2n$ to get the total number of bits in the simulation.

I have implemented a way of outputting maximum and minimum bitset sizes at compaction-time, but this operation has its own (slight) overhead, so I did not include it in the main measurement set. Thus I do not have that data for most parameter combinations, but for the few data points I do have, the difference between the maximum and minimum is small and the estimate does give us numbers with the same order of magnitude as the observed sizes.

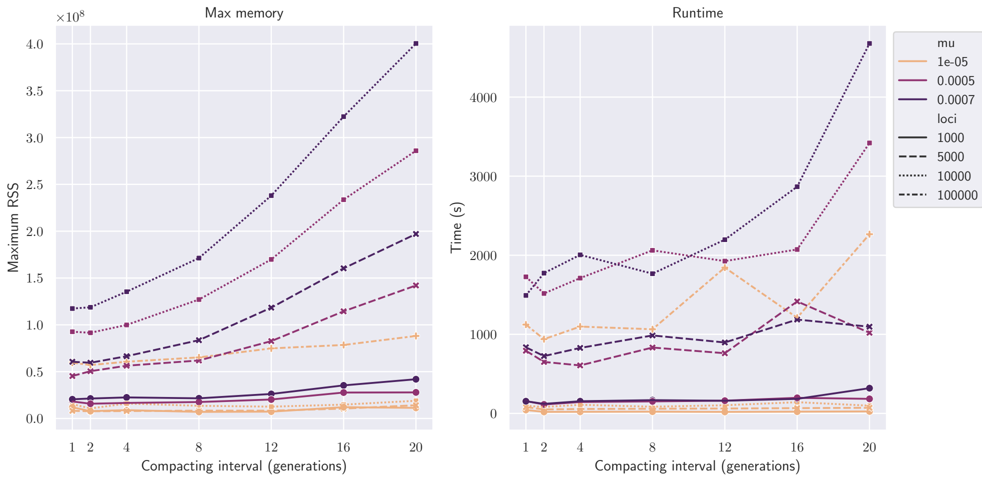



**Figure 5:** Memory and time requirements as a function of compacting interval for $n = 10^4$ and varied values of $\mu$ and $l$.

| set | $\mu$ | $n$ | $l$ | memory |
|---|---|---|---|---|
| $S_1$ | $10^{-5}$ | $10^4$ | $10^6$ | $> 24$ GB |
| $S_2$ | $10^{-6}$ | $10^6$ | $10^4$ | $> 179$ GB |
| $S_3$ | $10^{-5}$ | $10^5$ | $10^5$ | $> 184$ GB |

**Table 3:** The parameter sets and maximum RSS for runs which did not complete within a week – similar requirements for all values of $r$

# 4 Results

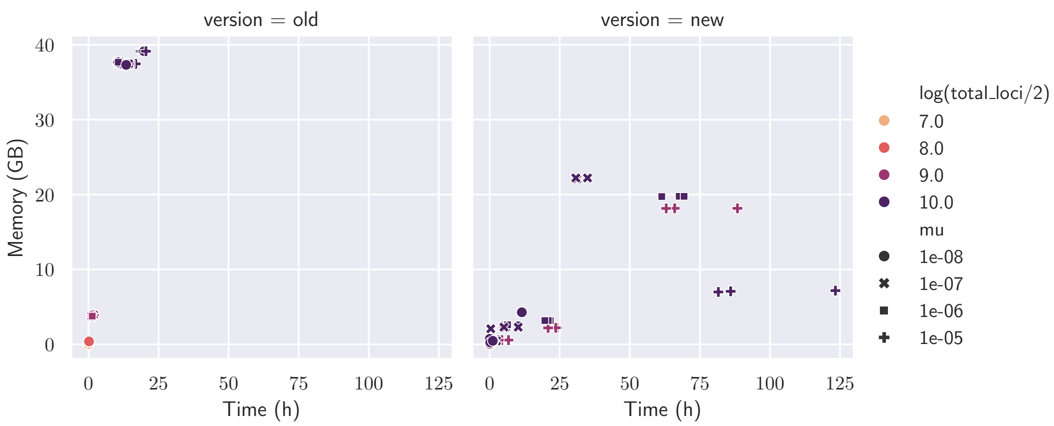
The measurement data collected shows some clear differences in the memory requirements and runtimes of simulations run under the original neutral traits architecture and the new indirect traits version. An overview for all the different parameter combinations for which the data were collected can be seen in Figure 6, which shows the maximum resident set size (RSS – that is, the main memory required to run the simulation) and the runtime of the simulations. In this and all the consequent pairwise plots, the data for the original neutral traits implementation is on the left, and the new indirect traits architecture on the right.

Note that some parameter combinations for the indirect trait architecture are missing in these data, but in Figure 6b, an estimate of the direction of the requirements is given based on incomplete simulation runs. Those simulation runs did not finish due to either taking too much memory or too long to complete. Runs which took over 373 gb memory – more than is available on a single standard CSC computing node, were terminated prematurely, meaning I do not have those final measurement values. Runs that required over seven days to run were similarly discarded. The missing data correspond to the largest simulations with the highest mutation rates. Simulations with $\mu = 10^{-5}$, $n = 10^6$, $l = 10^4$, all recombination rates, would have taken over 373 GB and so are missing. Three sets of simulations hit the one-week time-limit, and those are outlined in Table 3. The memory requirements among different recombination rates within these sets do not vary much.

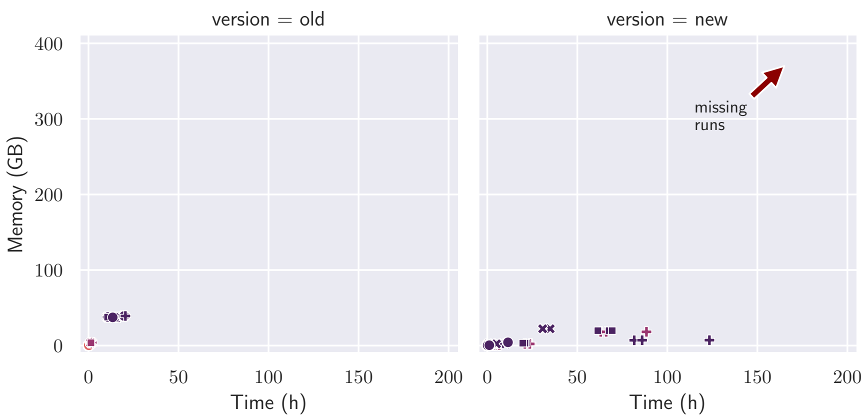
Unless this is accounted for, the overview in Figure 6a would tell a different story. As it is, we can see some trends in the figure of how the growths of memory requirements and runtime relate to each other, but in the full picture, the largest simulations with the indirect architecture are drastically more intensive to run as indicated in Figure 6b. By comparison, these same runs using the original architecture took under 40 GB and finished in less than 24 hours. However, for smaller runs, and for simulations with a low mutation rate, the indirect architecture achieves a smaller resource requirement, which I will now elaborate on. Let us begin by examining memory, and then shift over to runtime.

## 4.1 Memory

For the original implementation, we see in Figure 6a that the data points cluster by total number of loci (colour). This can also be observed in the plots showing memory

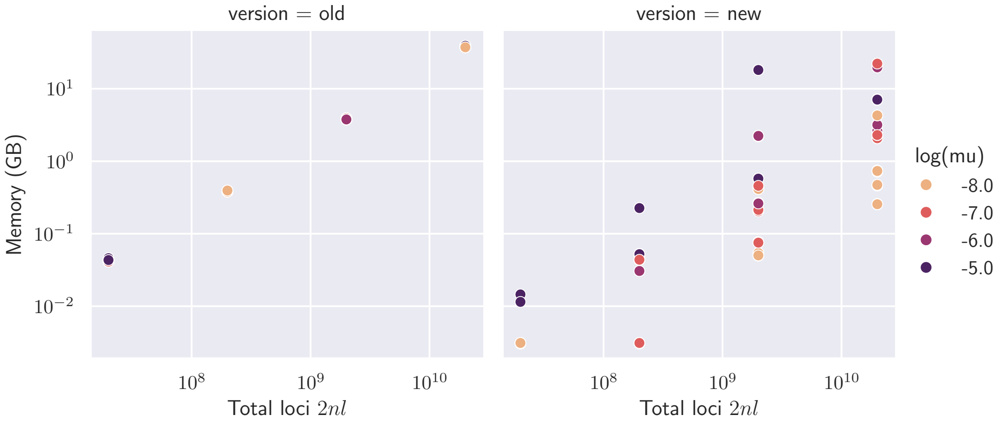



**(a)** Full measurement data, with some values missing for the indirect architecture due to simulation runs which did not complete

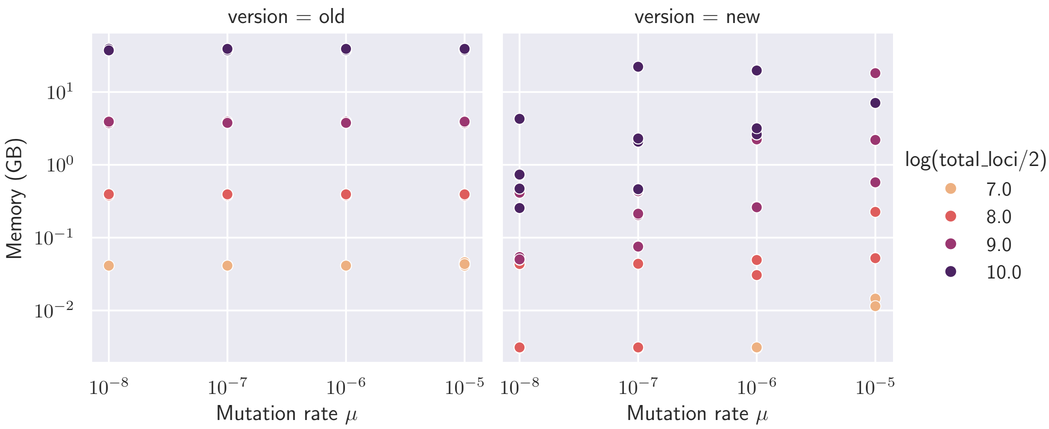



**(b)** With estimate for missing requirements

**Figure 6:** Plot of the relationship between the time- and memory usage under the two architectures. The data covers all the values for $n$ and $r$ used.




**(a)** Memory requirement as a function of total loci simulated




**(b)** Memory requirement as a function of mutation rate

**Figure 7:** Memory usage relationships to simulation parameters

usage in Figure 7, based on which we can conclude that the one factor which explains memory usage is the total number of loci being simulated ($2nl$). This is apparent in Figure 7a. This is exactly as we would expect, since all these positions are represented in memory directly, as explained in Sections 3.1 and 3.2. In the consequent Figure 7b, it can be observed that the mutation rate does not play a major part in the memory requirements of the original implementation, but instead those data points segregate into bands according to the total loci.

For the indirect traits case, however, we can observe in Figure 6a that there seems to be a grouping according to the mutation rate mu (marker type). This can also be seen in Figure 7b, where we see that the mutation rate does seem to somewhat explain the memory use, especially within the groupings by total loci (the data points of the same colour show an upwards trend). In Figure 7a we also notice a clear upwards trend in the indirect traits case meaning that the total number of loci does contribute to explaining memory requirements, but as there is still quite a spread within those vertical bands of points, so there may be something more to the situation.

In Figure 8 we see that for both implementations the memory requirements grow with number of individuals and loci. In the new version we also see how the mutation rate makes a difference. Moving down the panels, as the number of individuals increases by an order of magnitude, the memory requirements take a similar leap (note the logarithmic axes). The lines look for the most part to have a slope of roughly one, which would mean that increasing the number of loci is expected to affect the memory requirements linearly. Furthermore, we see clearly that the recombination rate has no effect on memory requirements as the lines of different styles overlap nearly completely.

Thinking about the internals of the algorithms in this case, certainly for the number of individuals it stands to reason that this should be so. This number is directly the number of bitsets that are stored during the simulation. As for the number of loci – it, together with the number of individuals and mutation rate determines the growth rate of the bitsets as these parameters determine the number of new mutations coming into the population every generation. We can indeed see in Figure 8 that higher mutation rates do correspond to higher memory requirements for the indirect case as shown by the line colours. The number of new mutations seen does not directly translate to growing memory usage, though, as unused bits are reused via the compacting process. The metric visualised here is the maximum memory required, which is mainly affected by how large of a buffer we need to keep in order to be able to store all the mutations coming into and spreading throughout the population each generation. From that point of view, it is logical that the size of the buffer is directly related to how many new mutations we need to have space for each new generation.

Now comparing the two implementations, there is a wider range of memory requirements that the indirect version covers. This is a result of these requirements being determined by three parameters $\mu, n, l$ in contrast to two for the original implementation $(n, l)$. This also means that for certain parameter combinations, the indirect version does provide a noticeably more efficient simulation. This is the case for low mutation rates (lighter coloured lines) and simulations with fewer individuals (upper panels). As a drawback, for other cases, the indirect simulation is much less efficient. This happens when there are a lot of mutations coming into the population in every generation (so when simulating a large population, with a large number of map positions and a high mutation rate). Some of these requirements are known to be outside the area covered by the plots in Figure 8.

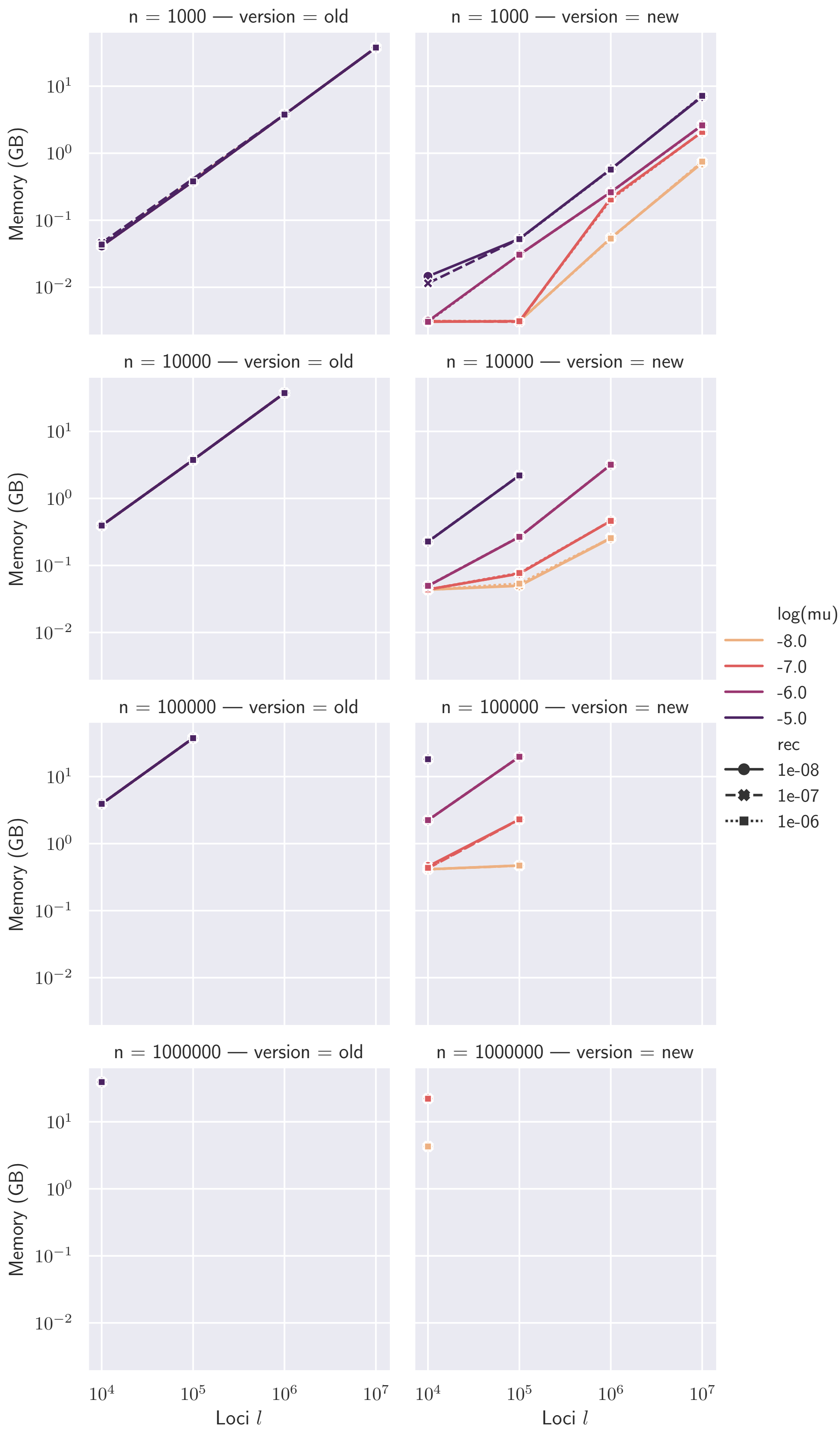


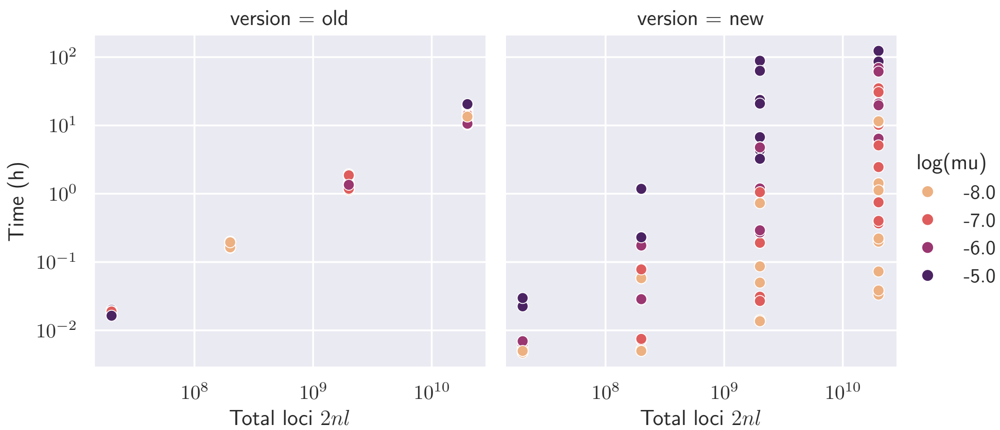

**Figure 8:** Memory requirements plotted as a function of simulated loci. Data are grouped according to mutation rate (line colour) and number of individuals (panels). Note that a few measurement points are missing for the indirect version.

## 4.2 Time

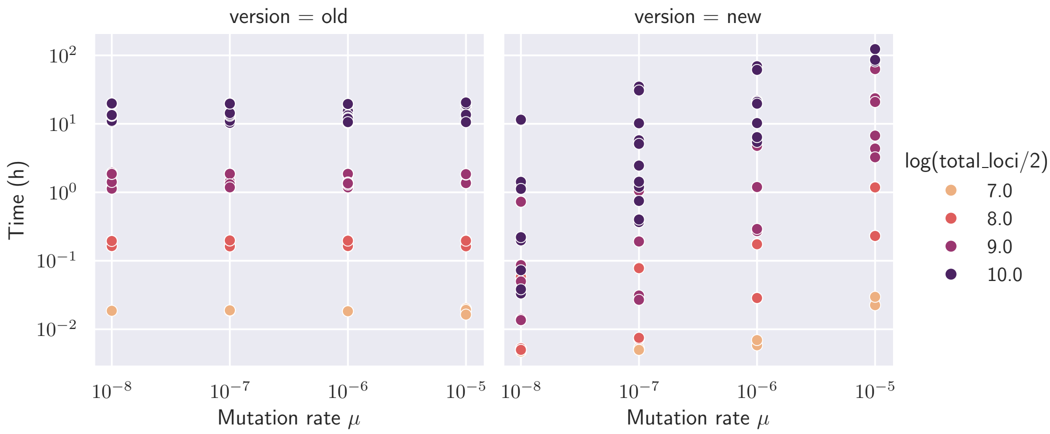
Data collected for the runtimes of these simulations are presented in Figure 9. We see that the trends visible in these plots are essentially all the same as for the memory requirements plots. Total number of loci determines runtime for the original neutral traits, and is a major factor for the new indirect architecture also (Figure 9a). The mutation rate does not seem to affect runtime for the original implementation, but it does correlate with runtime for the indirect architecture (Figure 9b), especially within the groupings by total number of loci (points of the same colour).

In Figure 10 I present an examination of the effects of different mutation rates, numbers of individuals and numbers of loci on the runtime, similar to that on memory in Figure 8. In comparison to the memory requirements, here the differences exhibited by the simulations with different mutation rates are even clearer. Once again, we observe that the recombination rate has no significant effect on the results.

Comparing Figures 8 and 10 we notice that the memory lines exhibit very similar trends to the runtime as seen, for example, in the middle two panels, with the slower




**(a)** Runtime as a function of total loci simulated




**(b)** Runtime as a function of mutation rate

**Figure 9:** Runtime relationships to simulation parameters

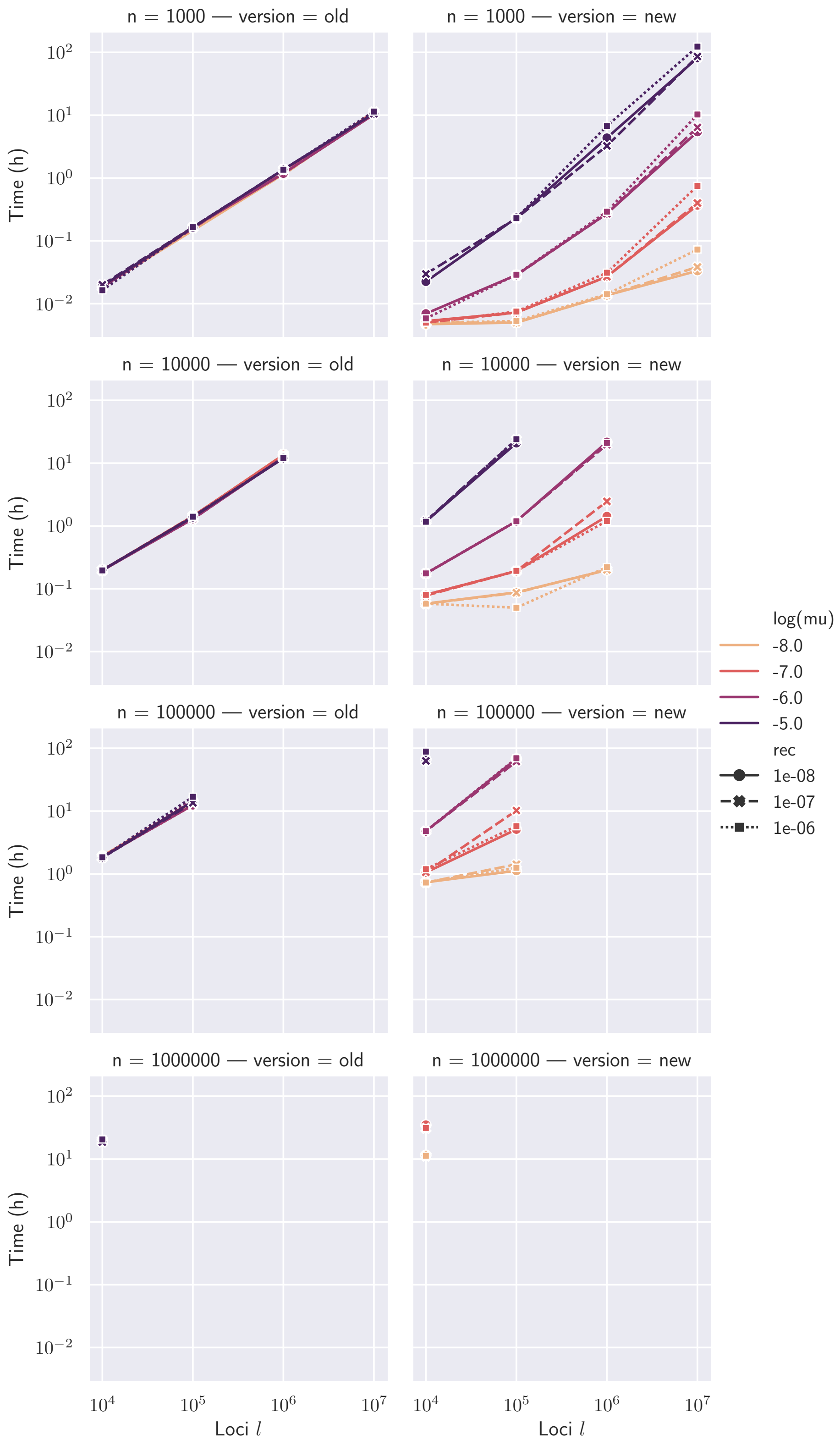



**Figure 10:** Runtime as a function of the number of simulated loci, grouped by mutation rate (line colour), recombination rate (line style) and number of individuals (panels).

increase for $\mu = 10^{-8}$ and for segments with lower numbers of loci (between $10^4$ and $10^5$). This is expected, because memory use in the algorithms is mainly determined by the lengths of bitsets and longer bitsets mean more operations to perform for each recombination (Section 3.1), thus providing a basis for the direct link between memory use and runtime observed here.

This link allows us to relate the time complexity of the simulations to their parameters. In the following, assume the simulation is in a steady state, meaning the bitsets have grown to the size they need to be to handle the influx of mutations. Each mutation takes a constant number of operations, and it is performed on average $E(\mathrm{Bin}(\mu, 2l)) = 2l\mu$ times for each of the $n$ individuals, so this is an $O(nl\mu)$ operation per generation.

As we have discussed, recombination is a more complex operation. For each individual, for each block in its longer bitset this incurs a constant number of operations. On top of that there is a constant number of operations plus a binary search on the set of crossovers, both performed as many times as there are differing positions on the two bitsets. We can use here the estimate for the length of the bitsets seen at the end of Section 3.3. Similarly we have an estimate for the number of differing positions as $l \cdot E(K_2)$ using equation (3). The expected number of crossovers is $lr$ and binary search is an $O(\log_2(x))$ operation. Combining these, the whole runtime estimate for recombination is

$$\begin{aligned} & n \cdot \frac{l \cdot E(K_{2n})}{64} + nl \cdot E(K_2) \cdot (1 + \log_2(lr)) \\ = {} & n \cdot \frac{l \cdot 2n\mu(2\ln(2n) + 1.355076)}{64} + nl \cdot 4n\mu(1 + \log_2(lr)). \end{aligned}$$

Discarding constants and lower-order terms gives us the O-complexity

$$O(n^2 l\mu(\ln(n) + \log_2(lr))). \tag{5}$$

Note that for some parameter combinations $lr < 1$ in which case the last term would become negative but in those simulations the binary search does not contribute in any major way to the runtime. Consider also that though there may be a constant number of operations associated with the terms in this expression, some of these are binary operations on 64 bit values (those corresponding to the first term) where others may involve memory lookups, so this estimate does have its limitations.

In Figure 11 we can observe that the lines showing runtime as a function of number of individuals have a slope greater than one and closer to two, which given the logarithmic axes would indicate a quadratic relationship. This is very much in line with the theoretical expectation of $O(n^2 \ln(n))$ complexity.

Table 4 shows a breakdown of the runtime of the indirect representation into some of the times (and portions) taken by the algorithms in the simulation. Note that classification is a step within recombination, so those times are portions of the time spent doing recombination. They were collected in two ways. For $\mu = 10^{-6}$, I ran (locally, on a laptop) different versions of the simulation code, each of which had some section of the code replaced with a dummy implementation, and compared

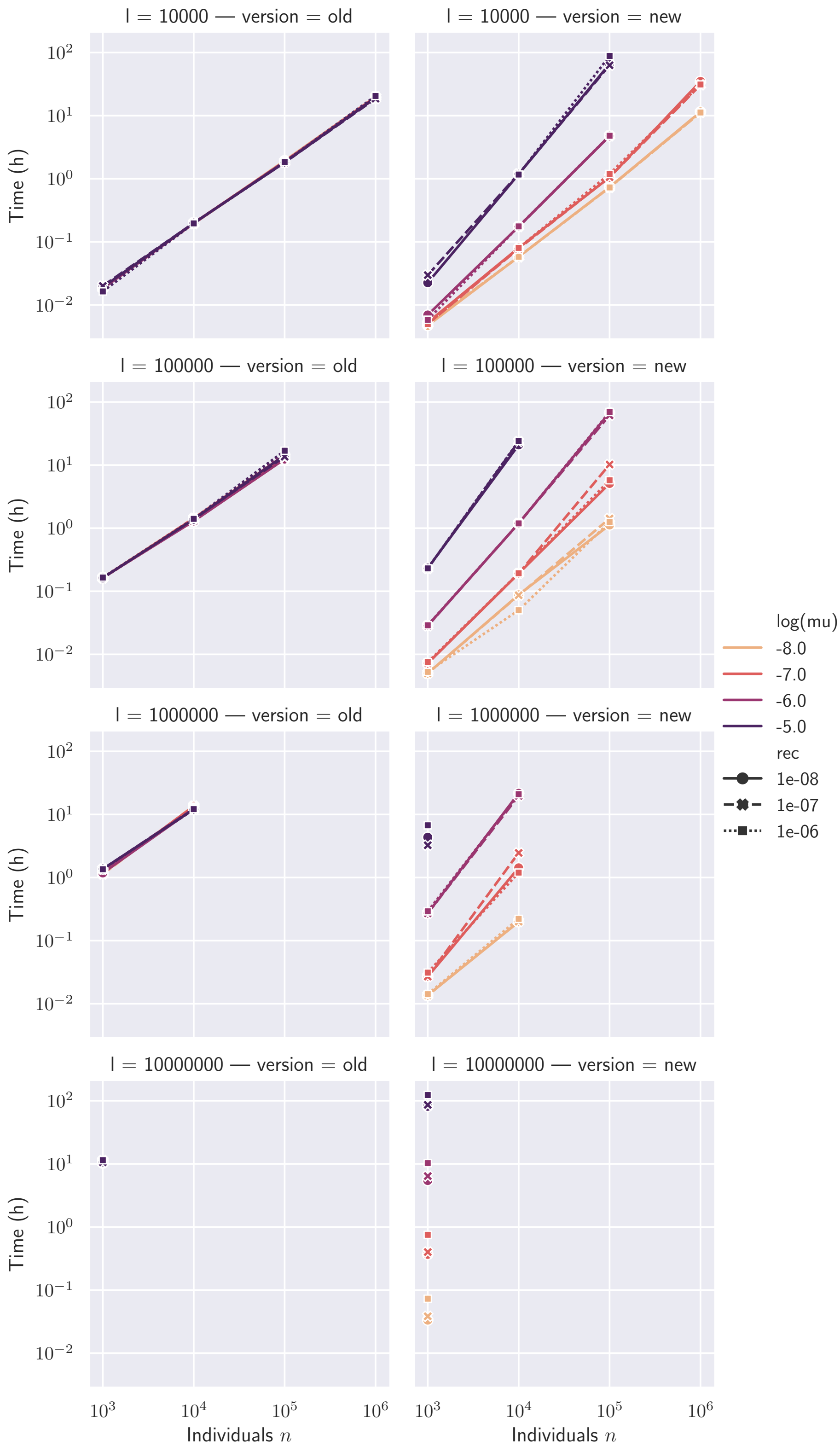



**Figure 11:** Runtime as a function of number of individuals. Note the slopes of roughly 2 which would indicate a quadratic relationship between $n$ and runtime.

| $\mu$ | r | n | l | full | recombination | classification |
|---|---|---|---|---|---|---|
| $10^{-8}$ | $10^{-8}$ | $10^4$ | $10^4$ | 240 s (100 %) | 42 s (18 %) | 2 s (1 %) |
| $10^{-6}$ | $10^{-5}$ | $10^4$ | $10^4$ | 566 s (100 %) | 351 s (62 %) | 221 s (39 %) |
| $10^{-5}$ | $10^{-8}$ | $10^4$ | $10^4$ | 4200 s (100 %) | 3840 s (91 %) | 2316 s (55 %) |

**Table 4:** Some runtime-breakdown values. Here recombination is a part of the full runtime, and classification is a part of recombination.

those to the full, unaltered version. For the rows with $\mu = 10^{-8}$ and $\mu = 10^{-5}$, I ran the simulations on a node on the CSC cluster, using Intel Vtune Profiler to collect sampling data about where the time was spent. We can see that irrespective of the collection method, when looking at the percentages, the results tell a similar story, yielding confidence to the results.

The largest chunk of time is spent in performing recombination even at low recombination rates. This was further examined by running Intel Vtune Profiler to get sampling info on a more fine-grained level. We see that the breakdown of time usage changes completely with a higher mutation rate. This goes to explain previous results (Figures 10 and 11) which showed that increasing the mutation rate increases runtime drastically. These sampling results indicate that the cost of recombination starts dominating the runtime. In the sampling data I noticed that up to about 48 % of the runtime is being spent in the inner loop of the inner function doing binary searches to determine which homologous chromosome the mutations reside on. This is a sub-step of the 55 % spent in classification. In the $\mu = 10^{-5}$ case, everything else except for recombination takes about 9 % of the total runtime. This includes drawing the mutations with the related bookkeeping, compacting bitsets and the set-up code for the simulation. Indeed the time spent doing the mutation is roughly the same whether the mutation rate is $10^{-8}$ (103 s) or $10^{-5}$ (132 s) and most of this time is spent in random number generation (implemented using GSL).

# 5 Discussion

Based on our measurement results presented in the last section, we can start to build a more fleshed-out understanding of the new architecture. The older, somewhat more straightforward way of keeping track of the genetics is very consistent in its resource requirements, whereas the new implementation has more variability, as it, contrary to the classic implementation, is affected by the mutation rate in the simulation.

On the level of the algorithms, this seems to be mainly a question about bitset lengths. Of the algorithms, recombination is the one which takes the most time and increasingly so for longer bitsets. The longer the bitsets, the more positions the algorithm needs to classify and the longer it takes. However, the bitset lengths are dynamic, and they grow as the population accumulates more mutations that need to be kept track of. In the simulation scenario used for this project, the length grows predictably as a function of the number of simulated positions (across the whole population) and the mutation rate.

That said, one question of possible future interest would be the behaviour of the bitsets, both in terms of length as well as density, and this across the entire bitset. This would give an idea of how the bitsets behave as the simulation progresses. It should be a fairly straightforward task to measure those statistics for the bitsets, something which I did, in fact, partially implement, but only at a point where most of the measurements of memory and runtime where already in, and thus those statistics were largely not dealt with in this project. There is also a slight overhead which would have affected the other measurements. The relationship has been partly explored from a theoretical perspective in this work (equation 5) for the simple simulation scenario employed. But it may be more interesting when running some other types of simulations (with more variable behaviour of individuals, for instance) and could be examined in more detail. Bitsets where the differing positions are clustered close to each other would be expected to be more efficient as we do not have to bit-walk blocks containing no differences.

It may be that some optimisations could be possible for the Wright–Fisher model used in this work. For example, since every individual goes through recombination and reproduces every generation, the compaction step could be incorporated into this instead of doing it separately. However, to facilitate future usage, I have attempted to keep the implementation such that it does not force a certain model, but could be used for other scenarios as well.

The model used here is a rather simple one, but as a result it is one that is suitable for large simulations. Since this simulation scenario only incorporates mutation and recombination, the costs of performing those are inevitably those which the measurements reflect. Thus we cannot know for sure how this implementation would fare if, for instance, there were fitness-effects of mutations to be tracked and computed. Even so, recombination is expected to be the most costly part of these simulations, no matter whether on the old or new implementation, so taking that into account, it is a sensible choice to take the measurements in a context where that is pivotal.

The key idea in the new implementation is to use indirection and store mutations in a shared data structure in order to avoid storing duplicate mutations and thus reduce memory requirements. Unlike time, memory is a resource for which there are hard limits, so any improvements are directly reflected in the simulations which it is possible to run. In principle a good idea, but the reality is a bit more complex. Even though using a single bit per mutation may seem like the minimal overhead, on the level of individuals this is not realised. In reality there will be portions of each individual's bitsets that are filled with zeroes. This is because mutations are stored centrally, the bitset addressing scheme is universal for the whole population, and individuals only have a subset of all the mutations present. Thus for simulation scenarios where there are a large number of mutations in total, but only a few per individual, the bitsets will be very sparse, resulting in suboptimal memory use. In this way, the memory needed per individual is determined by how many mutations are present globally. However, if the bitsets do stay dense the trade-off may be worth it, though the cost of this approach comes in the higher overhead of translating between mutation references and their genetic loci.

In the original clotho implementation [21], they use references to bitsets within each individual, but even so whenever there is a mutation or a recombination round which creates a unique bitset, a new bitset is required. This behaviour should be feasible to implement and could be worth testing in the future. Even though it would likely result in more dynamic memory handling due to the creation of new objects (as opposed to currently resizing), with certain parameter combinations the memory savings may be worth it.

Another way to approach the problem using indirection might be to allocate each new mutation a new unique id, or perhaps just to use memory addresses. This would save space by requiring memory only for those mutations an individual has, but doing recombination would become costly. And since recombination is the most costly operation here in general, this may not be a worthwhile solution. Also, especially when thinking of diallelic mutations, the space taken by a memory address is rather a lot, since if we are simulating a fixed number of positions, each position should only need one bit in memory. But if simulating many positions, resulting in very sparsely found mutations, say, much less than one mutation per 64 positions, the memory address-approach may be worthwhile in terms of memory required. Of course this approach would be endlessly customisable and the overhead may become acceptable in the case of some other, more complex types of mutations such as microsatellites or polymorphic sites with more than two variants.

In summary, this is a problem for which there is no known solution that would be optimal in every way. It is a question of balancing between minimal memory use and on the other hand keeping things quick and easy for the simulation steps, especially recombination.

One new feature in the new implementation is the ability to distinguish between two mutations in the population that have hit the same locus. This was impossible previously as all mutations that hit the same locus were equivalent across all individuals from the data structures point of view. In the new implementation each new mutation is unique, so it is possible to tell apart two mutations that have hit the same locus on different individuals. A caveat that should be mentioned is that if an incoming mutation attempts to hit either a locus at which the population contains a fixed mutation, or a locus at which the currently mutating individual has already mutated, the mutation will be rejected.

### Some Limitations

The first, obvious limiting factor was the fact that the implementation has to work within the existing software. This means no complete overhaul of the way mutations are mapped onto their positions on the chromosomes, for instance, as this should be handled via the shared genetic map. However, for this project, as it is keeping closely to the clotho-approach, this posed no major challenges as it is just a way of implementing the internal bookkeeping within a trait and does not affect other traits. That said, it may have limited the avenues that could be explored with respect to optimising the procedures, for example related to how crossovers are handled. As the crossover positions are drawn at each generation and have to be shared between (potentially) multiple traits, this precludes the possibility of pre-computing these positions at the beginning of the simulation and optimising the bitset addressing based on these, for instance.

The choice to do most measurements remotely on a supercomputer has many upsides. It allows measurements over a wider range of values because we are able to customise the main memory allocations to a high degree. It is also convenient, because measurements can be run as batch jobs and results fetched once the runs are finished. It is realistic, because this is in many universities the way these studies are done, and anyway since the research community is adopting the computing power as a service –mindset, you can often buy cloud computing time for such simulations. Still, running simulations in the cloud has some drawbacks as well. It is not quite as close to home for someone looking to run the software on their own laptop, even if the scenarios tested do cover those cases quite well. The supercomputer is also somewhat of an opaque platform, and access to some hardware-related tools or statistics is limited by what the service provider has decided to allow. This meant for instance that some of the more fine-grained profiling routines that would make use of the hardware counters were blocked. All things considered, it was nonetheless a useful experience of cloud computing in practice. I would also add that through it I gained an appreciation for the complexity of measuring software performance exactly.

The degree to which parallelism was explored on the level of constructing the algorithms was quite elementary. There are some reasons for this, though. With these kinds of simulations and the studies they are used for it is very common to run replicates of the same simulation. A trivial parallelisation is of course to run these replicates on different CPU-cores. On a cluster this can be achieved just by starting the replicates as separate batch jobs, as was done for the different measurement simulations in this work. Thus actually making a single simulation run effectively on multiple cores is not going to optimise the whole process which requires multiple simulation runs. Parallelising the algorithms in this project is also difficult, because parts of the implementation rely on one shared source of information as a way to save on memory requirements. That source needs to be queried/updated when things change and trying to do this in parallel could result in inconsistencies, such as multiple mutations being mapped to the same positions in the bitset.

### Future directions

I mentioned previously potentially gathering data on the lengths and densities of bitsets. Based on this, the algorithm for allocating new positions could perhaps be improved.

In the clotho paper [21], they propose that there could be an ordering within bit blocks and this could be beneficial when classifying mutations. This could be something worth thinking about, but keeping in mind that this approach would present an additional overhead when allocating bit positions.

Another possible improvement would be to contrast the different implementations at different points of a typical simulation workflow. In this project only the so-called burn-in phase of a simulation was examined, and after that is when the actual simulations would start, the ones whose results are of interest. The resource requirements for simulations are expected to be more constant after the genetics have had a chance to stabilise during the burn-in, but this should be verified. Some kind of a sensitivity analysis could also be of interest, where the genetics being simulated is perturbed somehow either by changing the population or the parameters and the effects on the resource use are recorded.

Another aspect that started to become clearer only towards the end of the measurements is the breakdown of runtime. A crucial observation about this once the data did come in was how drastically the breakdown changes for different simulation parameter combinations. It may be fruitful to investigate this in more detail as the data exist only for a few points even though the tentative relationship seems fairly clear-cut. It could be verified that cost of recombination does indeed become dominating in all cases when bitset length increases, as this is also affected by the distribution of the set bits within the bitsets. The entire situation may change if running completely different simulation models, however, and that would be useful information. For instance, the model used here assumed a panmictic population (all individuals breeding randomly) but if there were subpopulations that were more isolated from each other the structure of the bitsets may change and this could affect the cost of recombination.

One major reason for these observations taking so long is the fact that it is not a completely straightforward task to get accurate time breakdowns. This is for a large part a reflection of the way C++ works. As lines of code pass through the compiler, it does its own optimisations before outputting the final assembly language version, i.e. the program that is actually run by the processor. Thus the relationship between lines of code written by the programmer and the segments executed by the CPU can be somewhat convoluted, making it difficult to move from one to the other. This presents an obstacle to obtaining accurate profiling information and runtime breakdowns. There are ways around this, some of which were indeed used in this project to get an idea of where time was spent. But due to the process being slower and more involved, not many measurements were collected and these measurements had to be conducted separately. It would be useful to put some work into integrating these kinds of measurements more tightly into the benchmarking. This would likely involve writing some routines for collecting information about hardware events initiated by certain code segments. These could be used to get more fine-grained runtimes and memory usages as well as efficiency metrics such as instructions per clock cycle (IPC) and to tell us about potentially costly behaviour such as cache misses and main memory lookups. This information could be used to optimise the algorithms and data structures. However, using hardware counters would likely present its own extra overhead.

In general, development of more methodical and possibly automatable procedures for testing and benchmarking Nemo would be an important step in refining the software further. This should start at the level of planning.

For the benchmarks, with establishing some metrics that allow for comparing different implementations under varied simulation scenarios and that tell us something useful about the efficiency of the implementation. The two used in this work, total running time and maximum memory required, are a useful starting point but eventually, rather superficial information which does not give us details about the resource utilisation of the implementation or where the improvements may be found. Something like IPC values could be used to examine if the program is stuck waiting for high latency operations to complete. I mentioned some metrics above with relation to time breakdowns, but indeed these are things that should be made easily measurable for all current and future implementations in order to mark out targets for optimisation.

For testing, the starting point would be to define the expected behaviour of the program under some controlled scenarios, so that automated test cases for these could be implemented. Currently, the way to test the program is to run simulations for which the literature can be used to derive some expected statistics and to verify the "empirical" values given by the simulation. Even this approach could be automated.

Down a different vein, a follow-up on this project could take a look at how this bitset-based recording of mutations may be useful when calculating statistics of interest to population geneticists. Especially with cases where we want to know something like the number of mutations an individual is carrying, this info is easily readable from the bitset.

Another prevalent simulator software for FTPGS is SLiM [8]. That implementation, contrary to Nemo, is based much more on keeping track of mutations that percolate through the population. In the manual [7] we find an explanation about a concept of mutation runs which bears some resemblance to the ideas of the clotho approach in that it also groups mutations in blocks. However, it differs in that the grouping is done by the genomic location of the mutations. This allows their recombination to be optimised so that only mutations within a block that is hit by a crossover need to be examined, other blocks can be copied over directly. It may be worthwhile to dig into the details of their implementation if there are some ideas that would be useful also in the context of Nemo.

# 6 Conclusions

Now we can attempt to sketch out some outlines of the bigger picture underlying all these algorithms, measurements and results. We can assess the practical ramifications of the findings presented in this report and provide some guidelines as to how they can inform future projects as well as different use-cases for the software in question.

Despite being a technically more advanced way of handling mutations, the new implementation is not always better than the old straightforward approach. The spread of the runtimes and memory usage of the new implementation covers a larger swath, meaning that the higher requirements of some parameter combinations exceed those of the old implementation, but it also means that for parameter combinations with lower requirements it is possible to run simulations with considerably less time and memory. It can even be extrapolated that for specific corners of the parameter space (lower values of $n$ and relatively low $\mu$), the implementation explored here does enable the running of simulations that are infeasible with the old version due to the memory required, though these simulations are likely to take a considerably longer time. The old version is somewhat more consistent and its requirements easier to predict, but the cost of the consistency comes as higher overhead in some situations. In this work we have seen how it is possible to also predict requirements for the new implementation, but these are derived specifically for the simulation scenario examined here. As the resources needed are dependent on the number of mutations that need to be accounted for, this is expected to vary between scenarios. The major factors underlying these dynamics are the greater computational complexity of the new implementation and the fact that there is essentially one more simulation parameter which controls the resources needed.

At this point it may be fruitful to take a look back at the research questions outlined in Section 1 and consider to what degree the project has been a success. With regards to a more efficient way of running simulations, the results are a mixed bag, as discussed in the previous paragraph, but in some scenarios the measurements presented here indicate a clear improvement and those areas of the parameter space are relevant for applications. However, we ought to be aware that the testing procedure here covered only a very narrow scope of different kinds of simulation scenarios, and thus the assessments that can be made are likewise narrowly applicable. In that sense, we may regard the original goal as somewhat too general. But this realisation is in itself an indication of an increased awareness of the intricacies of the problems explored in this work, something which I would certainly consider a success. In terms of quantifying the requirements of the memory representation being examined, I would say this project has indeed explored those to a sufficient degree – for the simulation scenario chosen – and the results paint an accurate picture of the performance of this approach.

The question of scalability emerges as one of great interest. Indeed it is a perspective underlined in the original clotho paper. However, the results from testing presented therein are, in some respects, rather more limited than those obtained in this project. This may well be due to limitations in the computational capacity (hardware) at their disposal which also arise simply due to developments in this arena since the publication of that paper over a decade ago. Nevertheless, a general point can be made here about the importance of conducting testing at multiple scales, encompassing enough of the space occupied by potential use cases. If I had tested the new implementation only on small-scale simulations, the results and conclusions may have looked very different indeed. Now, since I had the opportunity to test also large-scale simulations a much more comprehensive picture emerges and the reliability of the results is on a different level.

Let us finish off with a few words about practical considerations and recommendations. Throughout this project we have seen that in the new implementation the factors determining memory use and runtime are the number of individuals, simulated loci and mutation rate. Of these, the number of individuals has the greatest effect. Thus this new architecture performs best, and can indeed do better than the fixed-length representation, in simulation scenarios where there are fairly few individuals and a low mutation rate. The ranges we are talking about are around 10 000 – 100 000 individuals and mutation rates of $10^{-8}$ – $10^{-7}$. Naturally, it depends on the organism and population system under study, but for many natural populations, these are the relevant sizes.

# Sources


[1] Björn Andrist, Ben Garney, and Viktor Sehr. *C++ High Performance.: Master the art of optimizing the functioning of your C++ code*. eng. Second edition. Expert Insight. Packt Publishing Limited, 2020. ISBN: 978-1-83921-258-1.

[2] Neil A. Campbell et al. *Biology: a global approach*. eng. Twelfth edition, Global edition. Harlow, United Kingdom: Pearson, 2021. ISBN: 978-1-292-34170-5.

[3] Thomas H. Cormen et al. *Introduction to algorithms*. eng. 3rd ed. Cambridge, MA: MIT Press, 2009. ISBN: 978-0-262-03384-8.

[4] en:User:Cburnett. *A generic 4-stage pipeline*. Dec. 2006. URL: https://commons.wikimedia.org/wiki/File:Pipeline,_4_stage.svg (visited on 07/25/2025).

[5] John H. Gillespie. *Population genetics : a concise guide*. eng. Baltimore, Md : The Johns Hopkins University Press, 1998. ISBN: 978-0-8018-5755-3. URL: http://archive.org/details/populationgeneti00john (visited on 12/01/2024).

[6] Frédéric Guillaume and Jacques Rougemont. "Nemo: an evolutionary and population genetics programming framework". eng. In: *Bioinformatics* 22.20 (2006). Place: Oxford Publisher: University Press, pp. 2556–2557. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btl415.

[7] Benjamin C Haller and Philipp W Messer. *SLiM: An Evolutionary Simulation Framework*. en. 2016. URL: http://benhaller.com/slim/SLiM_Manual.pdf.

[8] Benjamin C. Haller and Philipp W. Messer. "SLiM 4: Multispecies Eco-Evolutionary Modeling". In: *The American Naturalist* 201.5 (May 2023). Publisher: The University of Chicago Press, E127–E139. ISSN: 0003-0147. DOI: 10.1086/723601.

[9] Sean Hoban. "New guidance for ex situ gene conservation: Sampling realistic population systems and accounting for collection attrition". In: *Biological Conservation* 235 (July 2019), pp. 199–208. ISSN: 0006-3207. DOI: 10.1016/j.biocon.2019.04.013.

[10] Sean Hoban, Giorgio Bertorelle, and Oscar E. Gaggiotti. "Computer simulations: tools for population and evolutionary genetics". eng. In: *Nature Reviews. Genetics* 13.2 (Jan. 2012), pp. 110–122. ISSN: 1471-0064. DOI: 10.1038/nrg3130.

[11] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. en-US. 050US. Vol. Volume 2: Optimizing Earlier Generations of Intel® 64 and IA-32 Processor Architectures, Throughput, and Latency. online: Intel Corporation, Apr. 2024.

[12] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. en-US. 085US. Vol. Volume 1: Basic Architecture. online: Intel Corporation, Oct. 2024.

[13] Olivia L. Johnson et al. "Population genetic simulation: Benchmarking frameworks for non-standard models of natural selection". en. In: *Molecular Ecology Resources* 24.3 (2024), e13930. ISSN: 1755-0998. DOI: 10.1111/1755-0998.13930.

[14] J. F. C. Kingman. "The coalescent". In: *Stochastic Processes and their Applications* 13.3 (Sept. 1982), pp. 235–248. ISSN: 0304-4149. DOI: 10.1016/0304-4149(82)90011-4.

[15] William S. Klug. *Concepts of genetics*. eng. Twelfth Edition, Global Edition. Boston: Pearson, 2020. ISBN: 978-1-292-26551-3.

[16] Deborah M. Leigh et al. "Disentangling adaptation from drift in bottlenecked and reintroduced populations of Alpine ibex". en. In: *Molecular Ecology Resources* 21.7 (2021), pp. 2350–2363. ISSN: 1755-0998. DOI: 10.1111/1755-0998.13442.

[17] Youfang Liu, Georgios Athanasiadis, and Michael E Weale. "A survey of genetic simulation software for population and epidemiological studies". In: *Human Genomics* 3.1 (Sept. 2008), pp. 79–86. ISSN: 1473-9542. DOI: 10.1186/1479-7364-3-1-79.

[18] SchedMD LLC. *Slurm Workload Manager - slurm.conf*. URL: https://slurm.schedmd.com/slurm.conf.html (visited on 07/21/2025).

[19] CSC – It Center For Science Ltd. *Puhti - Docs CSC*. URL: https://docs.csc.fi/computing/systems-puhti/ (visited on 07/21/2025).

[20] Ted O'Donoghue and Matthew Rabin. "Doing It Now or Later". en. In: *American Economic Review* 89.1 (Mar. 1999), pp. 103–124. ISSN: 0002-8282. DOI: 10.1257/aer.89.1.103.

[21] Patrick P. Putnam, Philip A. Wilsey, and Ge Zhang. "Clotho: addressing the scalability of forward time population genetic simulation". In: *BMC Bioinformatics* 16 (June 2015), p. 191. ISSN: 1471-2105. DOI: 10.1186/s12859-015-0631-z.

[22] Max Schmid et al. "A Trade-Off between Robustness to Environmental Fluctuations and Speed of Evolution". In: *The American Naturalist* 200.1 (July 2022). Publisher: The University of Chicago Press, E16–E35. ISSN: 0003-0147. DOI: 10.1086/719654.

[23] William Stallings. *Operating Systems: Internals and Design Principles, Global Edition*. NOIDA, INDIA: Pearson Education Limited, 2014. ISBN: 978-1-292-06194-8. URL: http://ebookcentral.proquest.com/lib/aalto-ebooks/detail.action?docID=5483494 (visited on 10/21/2024).

[24] *std::upper_bound - cppreference.com*. URL: https://en.cppreference.com/w/cpp/algorithm/upper_bound (visited on 12/01/2024).

[25] G. A. Watterson. "On the number of segregating sites in genetical models without recombination". In: *Theoretical Population Biology* 7.2 (Apr. 1975), pp. 256–276. ISSN: 0040-5809. DOI: 10.1016/0040-5809(75)90020-9.

[26] Meike J Wittmann, Sylvain Mousset, and Joachim Hermisson. "Modeling the genetic footprint of fluctuating balancing selection: from the local to the genomic scale". In: *Genetics* 223.4 (Apr. 2023), iyad022. ISSN: 1943-2631. DOI: 10.1093/genetics/iyad022.

[27] Xiguo Yuan et al. "An Overview of Population Genetic Data Simulation". In: *Journal of Computational Biology* 19.1 (Jan. 2012). Publisher: Mary Ann Liebert, Inc., publishers, pp. 42–54. DOI: 10.1089/cmb.2010.0188.