

kalis: A Modern Implementation of the Li & Stephens Model for Local Ancestry Inference in R

Louis J.M. Aslett^{1*} and Ryan R. Christ²

^{1*}Department of Mathematical Sciences, Durham University, Stockton Road, Durham, DH1 3LE, County Durham, UK.

²Department of Genetics, Yale School of Medicine, 333 Cedar Street, New Haven, 06520, CT, USA.

*Corresponding author(s). E-mail(s): louis.aslett@durham.ac.uk;
Contributing authors: ryan.christ@yale.edu;

Abstract

Background Approximating the recent phylogeny of N phased haplotypes at a set of variants along the genome is a core problem in modern population genomics and central to performing genome-wide screens for association, selection, introgression, and other signals. The Li & Stephens (LS) model provides a simple yet powerful hidden Markov model for inferring the recent ancestry at a given variant, represented as an $N \times N$ distance matrix based on posterior decodings.

Results We provide a high-performance engine to make these posterior decodings readily accessible with minimal pre-processing via an easy to use package **kalis**, in the statistical programming language R. **kalis** enables investigators to rapidly resolve the ancestry at loci of interest and developers to build a range of variant-specific ancestral inference pipelines on top. **kalis** exploits both multi-core parallelism and modern CPU vector instruction sets to enable scaling to hundreds of thousands of genomes.

Conclusions The resulting distance matrices accessible via **kalis** enable local ancestry, selection, and association studies in modern large scale genomic datasets.

Keywords: Li & Stephens Model, R package, probabilistic haplotype model, Hidden Markov Model, genomics, high performance computation

047 Background

048
049 The hidden Markov model (HMM) of haplotype diversity proposed by Li & Stephens
050 [1] (hereinafter, the LS model) has become the basis for several probabilistic phasing,
051 ancestry inference, and demographic inference methods in modern genomics [2, 3].

052
053 Accelerated implementations of the LS model, typically targeting the Viterbi path,
054 are integral to many commonly used genomics software packages, including BEAGLE
055 [4], IMPUTE [5], and tsinfer [6]. A pioneering ancestry inference software package,
056 ChromoPainter, popularized the idea of using the LS model to summarize the ancestry
057 of N haplotypes with an $N \times N$ similarity matrix [7]. This matrix is obtained by
058 running N independent HMMs in which each haplotype is modelled as a mosaic of all
059 of the other haplotypes in the sample. This ‘*all-vs-all*’ copying approach is motivated
060 by the product of approximate conditionals (PAC) likelihood originally proposed by
061 [1] and allows ChromoPainter to render a chromosome-wide estimate of the recent
062 ancestry of the N haplotypes with high resolution.

063
064 The Relate [2] software suite extended this idea to performing local (locus-specific)
065 ancestry inference along the genome. Internally, Relate uses high performance C++
066 implementations of the forward and backward algorithm to perform posterior decoding
067 under a modified version of the LS model that incorporates derived allele informa-
068 tion at many loci spaced along the genome. We will refer to this modified LS model
069 as the derived allele haplotype copying model. These posterior decodings are trans-
070 formed to $N \times N$ local genetic distance matrices and used to initialise variant-specific
071 ancestral trees for downstream population genetic analyses ranging from demography
072 to selection inference.

073
074 The current Relate software suite does not provide an interface for outputting
075 the posterior decodings at a locus of interest and does not support the original LS
076 model, only the derived allele haplotype copying model, which requires derived allele
077 information.

Additionally, a LS-like model is implemented in [8] to run forward and backward recursions to variants of interest. However, the transition kernel used is different to the original LS model: upon a recombination event the transition kernel in [8] does not permit a haplotype to continue copying from the same donor haplotype.

The focus of **kalis** is to provide a high-performance engine to directly obtain the posterior decoding at a set of loci of interest for a dataset with hundreds of thousands of phased haplotypes. **kalis** supports the original LS model and the derived allele haplotype copying model. It provides a simple interface to enable rapid development of a range of future variant-specific ancestral inference pipelines on top, in the easy to use statistical programming language R [9].

At the same time, it has been recognised for over a decade [10] that the serial execution speed of CPUs will increase modestly, with additional performance primarily coming from concurrency via multi-core architectures or the growing width of specialised single instruction, multiple data (SIMD) instruction sets. Whilst multi-core architectures are now somewhat routinely exploited via forked processes or threading, SIMD instructions remain an often overlooked source of performance gains, possibly because they are harder to program. There are a cornucopia of SIMD instruction sets: on the Intel platform the genesis was in the 64-bit wide MMX instruction set [11] which allows simultaneous operation on two 32-bit, four 16-bit or eight 8-bit integers. The most recent incarnation on Intel CPUs is a suite of AVX-512 instruction sets [12], now capable of operating on 512-bits of various data types simultaneously (eg eight 64-bit floating point, or sixteen 32-bit integer values). Other CPU designs have similar SIMD technologies, such as NEON on ARM CPU [13] designs (including the Apple M1 and M2 processors, as well as Amazon Web Services Graviton range). Additionally all modern CPUs are superscalar architectures supporting instruction level parallelism,

an advance that has been in the consumer Intel platform since the Pentium [14]. Judicious programming can make it easier for compilers and the deep reorder buffers of modern pipelined CPUs to exploit this more hidden form of parallelism.

In this work we provide a reformulation of the LS model and an optimised memory representation for haplotypes, which together enable us to leverage *both* multi-core and SIMD vector instruction parallelism to obtain local genetic distance matrices for problem sizes that previously appeared out of reach. This high performance implementation is programmed in C [15], with an easy to use interface provided in R [9]. We provide low-level targets of AVX2, AVX-512 and NEON instruction sets (covering the vast majority of CPUs in use today), and the whole package has an extensive suite of 162,835 unit tests.

In the Implementation section below, we start with a description of the LS model and our reformulation which makes it amenable to these high-performance CPU technologies. We also describe the technical details of the underlying low-level implementation for the interested reader. We then demonstrate the performance that can be achieved with **kalis**, including examples with 100,000 haplotypes capable of running on a single machine. To the best of our knowledge, this is the first example of running the LS model at the scale of hundreds-of-thousands of haplotypes. We also present a real data example using **kalis** to examine the ancestry at the *LCT* gene. In the following Discussion section, we describe the user friendly R interface which enables easy use of the high performance implementation without any knowledge of the underlying CPU technologies.

Implementation

The LS model

To formalize our objective, let h be an $L \times N$ matrix of 0s and 1s encoding N phased haplotypes at L sites. Let $h_i^\ell \in \{0, 1\}$ denote the (ℓ, i) th element of h . For brevity,

let h_i denote the i th haplotype (the i th column of h) and h_{-i} denote all of the haplotypes excluding the i th haplotype. The LS model proposes an HMM for $h_i|h_{-i}$ in which the hidden state at variant ℓ , $X_i^\ell \in \{1, \dots, N\} \setminus i$, is an index indicating the haplotype in h_{-i} that h_i is most closely related to (or “copies from”) at variant ℓ . We present here their proposed emission and transition kernels (see Equation A1 and Equation A2 in [1]) with a simplified parametrisation that is similar, but not identical, to that used by ChromoPainter.

While the original LS model assumes that each haplotype has an equal *a priori* probability of copying from any other, following ChromoPainter, we define a left stochastic matrix of prior copying probabilities $\Pi \in \mathbb{R}^{N \times N}$ where Π_{ji} is the prior probability that haplotype j is copied by i and, by convention, $\Pi_{ii} = 0$. Here and whenever possible in **kalis**, all matrices are column-oriented such that the i th column pertains to an independent HMM where h_i is treated as the observation. There is some probability of a mis-copy at variant ℓ , μ^ℓ , which under the LS model is set proportional to the mutation rate at ℓ . This leads to an emission kernel of the form

$$\theta_{ji}^\ell := \mathbb{P}(h_i^\ell | X_i^\ell = j) = \begin{cases} 1 - \mu^\ell & \text{if } h_i^\ell = h_j^\ell \\ \mu^\ell & \text{if } h_i^\ell \neq h_j^\ell \end{cases}. \quad (1)$$

The transition kernel between hidden states is based on the recombination rate between sites. Let m^l be the genetic distance between variant l and variant $l + 1$ in Morgans (the expected number of recombination events per meiosis). Define $N_e = 4\tilde{N}_e/N$ where \tilde{N}_e is the effective diploid population size (ie half of the haploid effective population size). Then, under the LS model the transition kernel is

$$P(X_i^\ell = k | X_i^{\ell-1} = j) = \Pi_{ki} \rho^{\ell-1} + \mathbf{1}\{k = j\} (1 - \rho^{\ell-1}), \quad (2)$$

231 where $\rho^\ell = 1 - \exp(-N_e m^\ell)$ and $\mathbf{1}\{\cdot\}$ is the indicator function. [1] observe that in
 232 practice the estimation of recombination rates is improved when the scaled recomb-
 233 nation rate is raised to a power, so we adopt this approach and introduce an exponent
 234 γ . For $\gamma > 1$ the recombination map becomes more heavily peaked, whereas $\gamma < 1$
 235 tempers the recombination map to make it more flat and smooth. Hence, in **kalis**, we
 236 set

$$240 \quad \rho^\ell := 1 - \exp\left(-N_e (m^\ell)^\gamma\right), \quad (3)$$

241 calculated using `expm1()` to help avoid underflow.

242 In keeping with the nomenclature introduced by [7], we refer to h_i as the “recipient
 243 haplotype” and the remaining haplotypes, h_{-i} , as the “donor haplotypes”, in the
 244 context of the HMM where h_i is treated as the emitted observation vector. This reflects
 245 the fact that each recipient haplotype h_i is modelled as an imperfectly copied mosaic
 246 of the other observed haplotypes under the LS model. Hence, the posterior marginal
 247 probability at variant ℓ , $p_{ji}^\ell := \mathbb{P}(X_i^\ell = j | h)$, is the probability that donor j is copied
 248 by recipient i at variant ℓ given the haplotypes h . Under the above definitions of
 249 the prior copying probabilities Π , the emission kernel (1), and the transition kernel
 250 (2), the full $N \times N$ matrix of copying probabilities at ℓ , p^ℓ , can be obtained by
 251 running the standard forward and backward recursions [16] for each column (ie for
 252 each independent HMM).

253 From these posterior probabilities, we calculate a local $N \times N$ distance matrix, d^ℓ .
 254 Firstly, notice that theoretically $p_{ij}^\ell > 0$, but it can be that $p_{ij}^\ell < \varepsilon$, where ε is the
 255 double precision machine epsilon ($\approx 2.22 \times 10^{-16}$, [15], pp.26). Effectively this means
 256 d_{ij}^ℓ is too large to reliably work with precisely, and so for the purposes of distance
 257 calculations we treat ε as the smallest observable posterior probability, yielding

$$271 \quad d_{ji}^\ell = -\frac{\log(p_{ji}^\ell \vee \varepsilon) + \log(p_{ij}^\ell \vee \varepsilon)}{2} \quad \forall j \neq i \quad (4)$$

where \vee is the maximum binary operator. By convention $d_{ii} = 0$ for all i .

We proceed in the next Section to reformulate the forward and backward recursions so that we can more fully exploit modern high-performance CPU instruction sets, while preserving numerical precision.

Modification of the forward-backward algorithm

The N independent HMMs of the LS model have forward and backward probabilities, respectively:

$$\tilde{\alpha}_{ji}^\ell = \mathbb{P}(X_i^\ell = j, h_i^{1:\ell}), \quad \tilde{\beta}_{ji}^\ell = \mathbb{P}(h_i^{\ell+1:L} | X_i^\ell = j), \quad i \in \{1, \dots, N\},$$

where $h_i^{1:\ell}$ denotes haplotype i from variant 1 to ℓ inclusive.

Define,

$$F_i^\ell := \sum_{j=1}^N \tilde{\alpha}_{ji}^\ell \quad F_i^0 := 1 \quad (5)$$

$$G_i^\ell := \sum_{j=1}^N \tilde{\beta}_{ji}^{\ell+1} \theta_{ji}^{\ell+1} \Pi_{ji} \quad G_i^L := 1 \quad (6)$$

Then the forward and backward recursions for the LS model can be written in vector notation (subscript \cdot denoting a vectorised index),

$$\tilde{\alpha}_{\cdot i}^\ell \leftarrow \theta_{\cdot i}^\ell ((1 - \rho^{\ell-1}) \tilde{\alpha}_{\cdot i}^{\ell-1} + \rho^{\ell-1} F_i^{\ell-1} \Pi_{\cdot i}) \quad \text{for } \ell \in \{2, \dots, L\}, \quad (7)$$

$$\tilde{\beta}_{\cdot i}^\ell \leftarrow (1 - \rho^\ell) \tilde{\beta}_{\cdot i}^{\ell+1} \theta_{\cdot i}^{\ell+1} + \rho^\ell G_i^\ell \quad \text{for } \ell \in \{1, \dots, L-1\}. \quad (8)$$

with recursions initialised with $\alpha_{\cdot i}^1 \leftarrow \theta_{\cdot i}^1 \Pi_{\cdot i}$ and $\beta_{\cdot i}^L \leftarrow 1$. Note that Equation (7) corresponds to Equation A5 in [1].

To partially mitigate the risk of underflow, the forward recursion can be rearranged in terms of $\alpha_{\cdot i}^\ell := \frac{\tilde{\alpha}_{\cdot i}^\ell}{F_i^{\ell-1}}$, and the backward recursion in terms of $\beta_{\cdot i}^\ell := \frac{\tilde{\beta}_{\cdot i}^\ell}{G_i^\ell}$ (see Additional file 1 for details). Thus, in full for $\ell \in \{1, \dots, L\}$ we compute,

$$\alpha_{\cdot i}^1 \leftarrow \theta_{\cdot i}^1 \Pi_{\cdot i} \quad \text{for } \ell = 1 \quad (9)$$

$$\alpha_{\cdot i}^\ell \leftarrow \theta_{\cdot i}^\ell \left((1 - \rho^{\ell-1}) \frac{\alpha_{\cdot i}^{\ell-1}}{\sum_j \alpha_{ji}^{\ell-1}} + \rho^{\ell-1} \Pi_{\cdot i} \right) \quad \text{for } \ell > 1 \quad (10)$$

and

$$\beta_{\cdot i}^L \leftarrow 1 \quad \text{for } \ell = L \quad (11)$$

$$\beta_{\cdot i}^\ell \leftarrow (1 - \rho^\ell) \frac{\beta_{\cdot i}^{\ell+1} \theta_{\cdot i}^{\ell+1}}{\sum_j \beta_{ji}^{\ell+1} \theta_{ji}^{\ell+1} \Pi_{ji}} + \rho^\ell \quad \text{for } \ell < L \quad (12)$$

Given $\alpha_{\cdot i}^\ell$ and $\beta_{\cdot i}^\ell$, the vector of posterior probabilities for recipient i , $p_{\cdot i}^\ell$, can be calculated directly by normalising,

$$p_{\cdot i}^\ell = \frac{\alpha_{\cdot i}^\ell \odot \beta_{\cdot i}^\ell}{\sum_j \alpha_{ji}^\ell \odot \beta_{ji}^\ell} \quad (13)$$

where \odot denotes the Hadamard product. In the event that $\sum_j \alpha_{ji}^\ell \odot \beta_{ji}^\ell = 0$, the distance between the recipient haplotype i and all of the donor haplotypes is beyond numerical precision, so as per the earlier discussion we define $p_{ji}^\ell = \varepsilon \forall j \neq i$.

Finally, the local distances follow by taking the negative log and symmetrising. Note that if the distances are standardised for one of these columns, to account for the fact that the standard deviation will be 0, we set all of the standardised distances to 0. Please see Additional file 1 for a discussion on parameter values and exactly how **kalis** performs certain computations to maintain the numerical stability of the algorithm.

Core Implementation Details	369
	370
The R interface described hereinbefore is a thin wrapper layer around a high-	371
performance implementation of the core algorithm which is written in standards	372
compliant C18 [15]. Most data structures are represented with native R types enabling	373
user inspection and manipulation, except for the haplotype sequences themselves.	374
	375
Computationally, the innermost forward and backward recursions are implemented	376
using compiler intrinsics to exploit a variety of modern CPU instruction sets, including	377
Streaming SIMD Extensions (SSE2 and SSE4.1), Advanced Vector Extensions (AVX,	378
AVX2, AVX-512 and FMA) and Bit Manipulation Instructions (BMI2) on Intel plat-	379
forms; as well as NEON on ARM platforms. AVX2 is supported in Intel CPUs of the	380
Haswell generation (released Q2 of 2013) or later, AVX-512 tends to be available only	381
in recent Intel server and workstation grade CPUs, and NEON is available for ARM	382
Cortex-A and Cortex-R series CPUs, as well as Apple M1/M2 and Amazon Web Ser-	383
vices Graviton processors. Although this covers most CPUs likely to be in use today,	384
we none-the-less provide reference implementations in pure standards compliant C	385
which will operate on any CPU architecture with a C18 compliant compiler. During	386
package compilation, the correct code-paths are compiled based on detection of the	387
presence or absence of the required instruction sets, or at the direction of the user	388
via compiler flags. See Additional file 1 for more details, and for guidance on how to	389
directly check your CPU for SIMD support.	390
	391
It may be worth noting at this juncture that it was an explicit design choice to tar-	392
get CPUs and not GPU or tensor cards initially. This is because most University high	393
performance computing clusters have plentiful CPU resources, often with untapped	394
power in advanced SIMD instructions sets. We believe that the problem size that can	395
be realistically tackled in many genetics studies can be massively increased <i>without</i>	396
needing to resort to add-on cards, though to scale beyond even this we may explore	397
heterogeneous computing architectures in future kalis research.	398
	399
	400
	401
	402
	403
	404
	405
	406
	407
	408
	409
	410
	411
	412
	413
	414

415 In this section, we now describe the inner workings and design principles of the
416 package, first covering in detail the data structures (both user facing and internal),
417 followed by the computational implementation.
418
419

421 Data structures

422
423 There are three user accessible data structures utilised in the package and a low
424 level binary haplotype representation which is not directly user accessible. The two
425 principle data structures of interest to users are forward and backward table objects,
426 represented as native R lists with respective S3 class names `kalisForwardTable` and
427 `kalisBackwardTable` (detailed in Table 2 and discussed later), which are created with
428 package functions `MakeForwardTable()` and `MakeBackwardTable()` respectively. The
429 third user accessible data structure holds the LS model parameters, represented as
430 a native R environment with S3 class name `kalisParameters`, which can be created
431 with the package function `Parameters()`.
432
433
434

439 Haplotype data

440
441 The haplotypes are stored in an optimised binary representation which is only natively
442 accessible from within C. Note that here “optimised” is not a reference to space-
443 optimisation: it would be possible to represent the haplotypes in an even more
444 compressed manner, but we aim for streaming compute speed optimisation instead.
445
446
447

448 The haplotypes are loaded from disk and transformed to an in memory cache in this
449 representation via `CacheHaplotypes()`, but this function does not return any handle
450 to the loaded data. Thus the package provides the accessor function `QueryCache()`,
451 which copies genome segments from the binary representation into native R integer
452 vectors for user inspection.
453
454

455 When `CacheHaplotypes()` loads haplotypes into the cache, they are interleaved
456 into a flat memory space which is organised as variant-major. That is, variant 1 of each
457 haplotype is loaded, converted to a binary 0/1 and then 32 consecutive haplotypes are
458
459
460

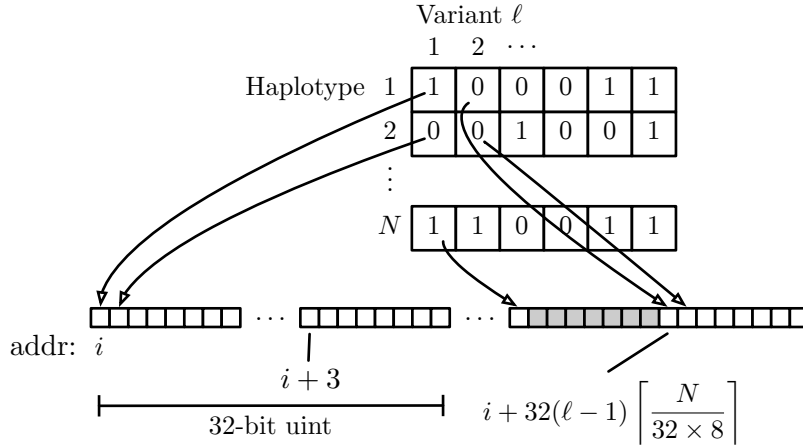


Fig. 1 Efficient binary representation of interleaved haplotypes in memory, with 32-byte boundary alignment for each variant start for SSE/AVX instructions (here $i \bmod 32 = 0$). The grey boxes indicate essentially ‘wasted’ bits which are ignored to ensure alignment for the start of the next variant.

packed into an unsigned integer. Moreover, the initial flat memory allocation is aligned on a 32-byte boundary to satisfy memory alignment requirements for some CPU vector instructions¹, and after all haplotypes at a given variant are packed into consecutive unsigned integers the pointer is wound forward to the next 32-byte boundary to ensure the next variant starts on an SSE/AVX vector compatible memory boundary. This is depicted in Figure 1.

Firstly, note that this orientation is natural, since the forward and backward recursions operate variant by variant, meaning variant-major storage ensures sequential memory locations are fetched during a recursion. Indeed, with the cache line size of 64-bytes (starting Intel Pentium IV), we essentially trigger the loading of $64 \times 8 = 512$ neighbouring variants upon accessing the first variant in a recursion. This effect is even more pronounced on Apple M1/M2 whose cache line size is 128-bytes, resulting in 1024 variants being pre-fetched upon access to the first variant in a recursion.

Secondly, a possible drawback is that we must extract the individual bit into a double floating point representation in order to compute with it in the recursion. However, efficient CPU instructions can help here too: take for example the following

¹Certain modern CPUs do not require specific alignment to be able to load memory to SSE/AVX registers, but for maximum compatibility we honor the alignment anyway.

strategy **kalis** uses on an AVX2 capable CPU. Using the PDEP instruction in BMI2, we can efficiently deposit a bit into every ninth bit of an `int` (so there are now 4 8-bit integers taking on the value of the haplotype at this variant packed in an `int`). Then, using SSE2, SSE4.1 and AVX instructions one can inflate through representations from 4 8-bit integers packed in an `int` up to 4 64-bit doubles packed in a 256-bit AVX register. As such, we are then ready to operate with this variant in parallel using AVX instructions.

During development, testing indicated the memory bandwidth and cache efficiency savings of the packed binary representation provided speed-ups thanks to these instructions efficiently enabling unpacking and spreading a haplotype variant bit for parallel use. Furthermore, such a compact representation means that more of L1/L2 cache and memory bus bandwidth is left available for forward and backward tables, which are the largest objects we work with in this problem.

Parameters

The `kalisParameters` object uses an environment rather than list for parameters for two reasons: (i) the parameter environment and its bindings are locked which prevents changes in parameter values between forward or backward table propagation steps, since parameters must be fixed for all steps of a given forward or backward computation; and (ii) an environment explicitly ensures the (often large) parameter vectors are not copied when associated with potentially many different tables, but will always be purely referenced.

The environment contains only two members: another environment with the actual parameter values (which is locked with `lockEnvironment()`); and a SHA-256 hash of those parameter values (details in Table 1). The purpose of the hash is to be able to efficiently determine whether the correct parameter set for a given forward or backward table has been passed when computing forward or backward recursions

kalisParameters object	Data type
pars	Locked R environment, containing:
	rho vector length L
	mu vector length L , or scalar
	Pi $N \times N$ matrix, or scalar
sha256	character

Table 1 The content of the data structure representing parameter objects.

kalisForwardTable object	kalisBackwardTable object	Data type
alpha = α^ℓ	beta = β^ℓ	$N \times N$ matrix
alpha.f = F^ℓ	beta.g = G^ℓ	vector length N
l = ℓ	l = ℓ	integer scalar
from_recipient	from_recipient	integer scalar
to_recipient	to_recipient	integer scalar
pars.sha256	pars.sha256	character
	beta.theta	logical scalar

Table 2 The content of the core data structures representing forward and backward table objects, together with their correspondence to mathematical quantities.

from an already initialised table (since it would be incorrect to propagate forward or backward using different parameter sets in different parts of the genome).

Forward/backward tables

Recall that the recipients (columns) in the forward/backward tables correspond to independent HMMs. Therefore, **kalis** enables storing only a ‘slice’ of recipients in a forward/backward table, making parallelisation across non-shared memory clusters much simpler: given all haplotype data, these recipient slices can be independently propagated in a communication free manner.

The forward and backward table objects contain not only the (upto) N independent forward/backward vectors at variant ℓ , but also supporting meta-data. This includes the variant the table is currently at, the scaling constants F^ℓ (forward, Equation (5)) or G^ℓ (backward, Equation (6)), the range of recipient haplotypes represented (that is, the recipient HMMs to which the column corresponds), and a hash of the parameter values used in propagating this table.

599 In total, a full-size forward table for example requires $8N^2 + 8N + 1576$ bytes of
600 memory² for storage and the small overhead of R object management. Since this grows
601 quadratically in the number of haplotypes, most functions in the package operate on
602 forward and backward table objects in-place, rather than via the idiomatic copy-on-
603 write mechanism of standard R. The most important consequence of this for users
604 is that standard assignment of a table object to another variable name only creates
605 a reference and so an explicit copy must be made by using the `CopyTable()` utility
606 function provided in the package.

613 Core SIMD code

614 The two most important core algorithms which are accelerated with SIMD vector
615 instructions are the forward and backward recursions. This code is fully implemented
616 in C, with tailored modifications accounting for all combinations of: scalar/vector μ ,
617 scalar/matrix Π , and use of the asymmetric mutation model of RELATE [2] or not
618 (ie 8 combinations); to ensure that minimal memory accesses are performed where
619 possible. So, for example, scalar μ and scalar Π parameters will be faster than any
620 other combination since these values are likely to be held in registers (or at least L1
621 cache) for the duration of the recursion.

622 Additionally, in all places where we identify SIMD instructions may be used, a
623 macro is deployed, with a header file providing all mappings from these macros to
624 a specific SIMD instruction for all supported instruction sets. Taking arguably the
625 simplest non-trivial example, all `src/ExactForward*.c` and `src/ExactBackward*.c`
626 files make use of the custom macro `KALIS_MUL_DOUBLE(X, Y)` when they need to mul-
627 tiply `KALIS_DOUBLEVEC_SIZE` double precision floating point values together. The file,
628 `src/StencilVec.h` then provides definitions for these macros under each instruction
629 set `kalis` supports (via assembly intrinsics), together with a pure C alternative. For
630 this example, we have (with ... indicating other macro definitions):

644 ²Measured under R 4.2.2

```

// Extract from src/StencilVec.h
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690

```

The inner-most loop in these core files then includes a programmatically generated unroll to the depth specified during compilation. All this is wrapped in code which dispatches using `pthread`s to multiple threads, with automatic detection of the ability to pin to specific cores if that option is passed (important in some settings to ensure a hot L1/L2 core cache). In particular, each thread operates on a subset of columns of the forward and backward tables, ensuring spatial locality for memory accesses. Furthermore, when propagating by more than a single variant position, each column (ie each independent HMM) is propagated all the way to the target variant before proceeding to the next column, ensuing temporal locality of memory accesses.

691 Unit tests

692
693 Given the complexity of the development described above, we have implemented a
694 comprehensive suite of unit tests to ensure correctness. Internal to the package is a
695 “gold master” implementation of the LS model, which is a pure R implementation
696 that has been written for correctness and is not optimised for speed. These pure R
697 implementations are callable with an undocumented argument option to the standard
698 `Forward()` and `Backward()` functions: if the argument `nthreads = "R"` rather than
699 a numeric value, then the gold master implementation is used (at the cost of running
700 significantly slower).

701
702 Unit tests fall broadly into two categories, one verifying the correctness of loading
703 from the different supported input formats (via R matrix, `.hap.gz` and `h5`) into the
704 optimised binary representation of Figure 1, the other checking forward and backward
705 computations against a ground truth computed by the gold standard R implemen-
706 tation. The latter category of tests are the most extensive, since they cover tests of
707 all combinations of: single threaded and multi-threaded computation; moving differ-
708 ent numbers of variants in a single call; different problem sizes where the numbers of
709 haplotypes is either exactly divisible by the CPU vector unit length (i.e. 256-bits for
710 AVX2 etc), or has different remainders; original LS and derived allele haplotype copy-
711 ing model; scalar and vector mutation probabilities (μ); uniform and matrix copying
712 probabilities (Π); and in the case of backward recursions, all combinations of start-
713 ing and ending a recursion in standard or rescaled probability space (`beta.theta`
714 argument to `Backward()`).

715
716 All these combinations give rise to over 162,000 tests (note also that the exact
717 number of tests varies by architecture due to the differing vector unit lengths). This
718 large number of tests ensures all the separately optimised code paths for the various
719 combinations of run-time options are covered. We note that the tests take quite some

720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736

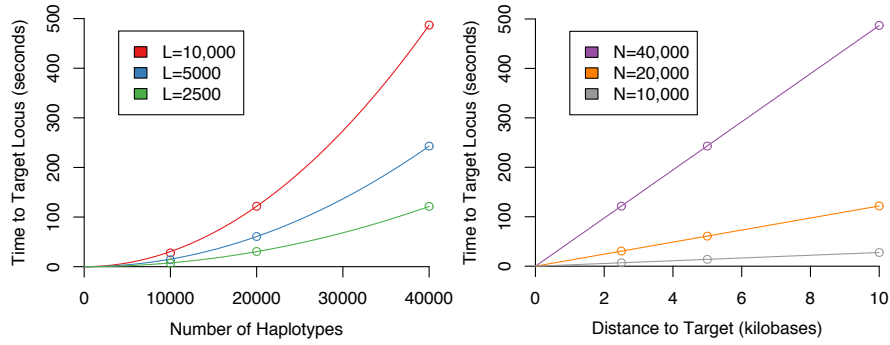


Fig. 2 **kalis** shows the expected order N^2 and order L scaling of the LS model. Computed on an Amazon Web Services **c4.8xlarge** instance (36 vCPUs, 60 GB of RAM).

time to run (e.g. potentially 30-60 minutes on a laptop), precisely because the gold master R code is run to provide the ground truth for these tests.

If a user wishes to confirm correctness on their particular platform, they can be run with the following commands:

```
remotes::install_github("louisaslett/kalis",
                        INSTALL_opts = "--install-tests")
testthat::test_package("kalis")
```

Results

We provide a brief overview of some example performance figures, though due to the highly tuned nature of **kalis**, the exact performance you can expect will be heavily dependent on your exact computer architecture and resources.

First, it is important to note we do *not* claim to have altered the scaling properties of the LS model, only that we provide an implementation which is highly optimised within the scaling constraints inherent to the model. As such, Figure 2 demonstrates that **kalis** indeed inherits the $\mathcal{O}(N^2)$ and $\mathcal{O}(L)$ properties of the original LS model.

We turn now to the benefits **kalis** does provide.

Firstly, for some of the reasons highlighted in the previous Section, **kalis** exhibits accelerated performance when propagating the forward/backward recursions over

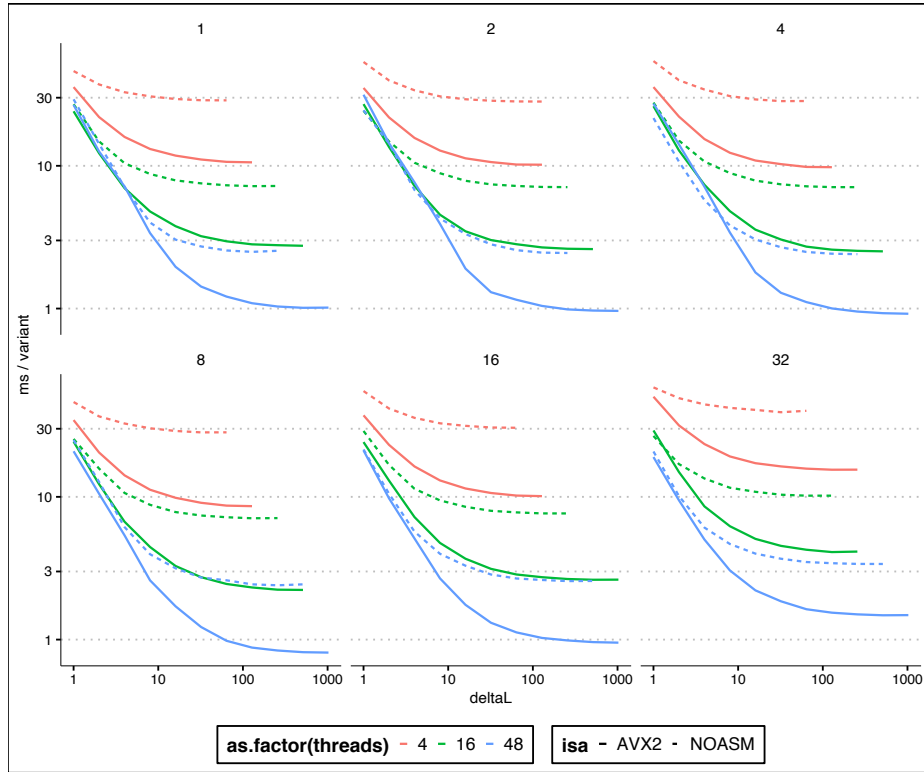


Fig. 3 Log-log plot of milliseconds per variant performance (y -axis) of the forward algorithm on 10,000 haplotypes, against the number of variants propagated (x -axis). Each panel is a different loop unrolling depth (panel title gives loop unrolling level). Line colour denotes number of CPU threads, whilst a dashed line indicates vanilla C and a solid line indicates hand-coded AVX2 instructions. In total, using AVX2, 48 threads, and loop unrolls to depth 8, it takes less than 10 seconds to propagate a 10000×10000 forward table over 10,000 variants.

more extended stretches of the genome. This is because every effort has been made to be cache efficient, so that when more than a single variant step is taken, the strong cache locality design ensures that we are not memory bandwidth limited. This effect can be seen quite dramatically in Figure 3 by the rapid decrease in compute time per variant as longer stretches are propagated.

Secondly, the hard-coded loop unrolling functionality which can be controlled at compile time by the user can be seen to be beneficial in Figure 3. Clearly excessive loop unrolling is harmful, with depth 32 unrolls actually being substantially slower than no unrolling. However, unrolling to depth 8 does give a clear improvement. The best

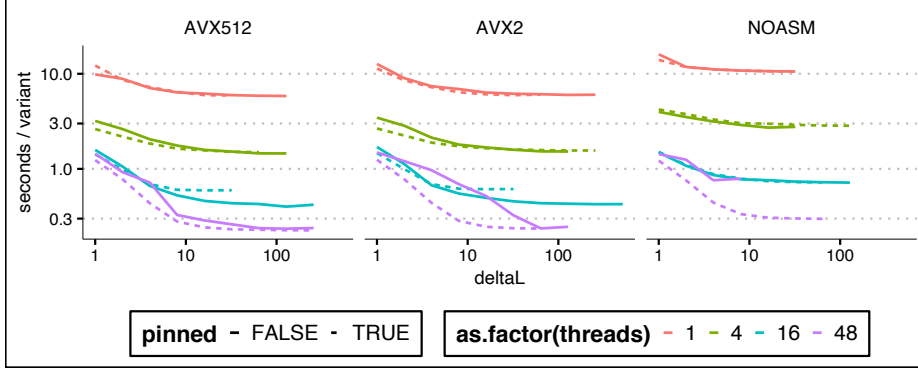


Fig. 4 Log-log plot of seconds per variant performance (y -axis) of the forward algorithm on 100,000 haplotypes, against the number of variants propagated (x -axis). Each panel is a different instruction set (AVX-512/AVX2/none). Line colour denotes number of CPU threads, whilst a dashed line indicates pinned threads and a solid line indicates no thread pinning. In total, using AVX-512, 48 threads, and pinned threads, it takes less approximately 38 minutes to propagate a 100000×100000 forward table over 10,000 variants.

choice of unrolls will be both problem and architecture dependent, so we recommend testing different unroll levels on the target problem before performing long compute runs.

Figure 3 also illustrates that the hand-designed use of low-level vector SIMD instructions is not superfluous, with substantial speed-up afforded by their use (the difference between dashed and solid lines of the same colour).

Finally, Figure 4 shows that in certain very large problem settings **kalis'** ability to pin threads can make a substantial difference. In this setting, AVX2 showed the greatest benefit from eliminating context switching, ensuring that the cache is not invalidated by threads migrating between cores. The lack of substantial difference between AVX2 and AVX-512 here once thread pinning is employed calls for some investigation, though this may be a result of thermal/power throttling which is known to occur especially for AVX-512 heavy code [17].

These performance examples again highlight the importance of pilot benchmark runs with different configurations of instruction set and unroll settings before embarking on long compute runs to ensure the greatest compute efficiency is achieved for a given problem and compute architecture.

875 Benchmarking comparison

876
877 We performed two benchmarking experiments to compare the implementations of the
878 forward and backward algorithms in **kalis** to those in Relate [2]. We based all of our
879 benchmarks off on the same set of haplotypes, taken from the 1000 Genomes Project
880 [18], as used in our real-data example which follows below. The data include 5008
881 haplotypes observed at 29193 variants.

882 **kalis** can perform the forward and backward recursions under either the orig-
883 inal LS model (the default) or the derived allele haplotype copying model if
884 `use.speidel=TRUE` is passed to the `Parameters()` function. Since Relate only com-
885 puts these recursions for the derived allele copying model, it can exploit the
886 asymmetry in the emission kernel based on the derived allele orientation of each vari-
887 ant. When painting a given recipient haplotype as a mosaic of donor haplotypes, this
888 allows Relate to effectively skip all variants where a recipient haplotype does not carry
889 the derived allele. This acceleration cannot be applied to the original LS model, which
890 **kalis** was primarily designed for, because the symmetric emission kernel requires both
891 the forward and backward algorithms to iterate over every variant for every recipient
892 haplotype. Even with the derived allele copying model activated, **kalis** will still visit
893 every variant for every recipient haplotype.

894 Accordingly, we found that the forward and backward recursions were approxi-
895 mately $4\times$ faster using Relate rather than using **kalis**. However, if Relate visits every
896 locus, as would be necessary to compute the original LS model, we found that the for-
897 ward and backward recursions were approximately $6\times$ faster using **kalis** rather than
898 using Relate. This demonstrates the benefit of the low-level optimisations made in
899 **kalis**. In principle **kalis** could also employ the same optimisation as Relate and visit
900 only derived sites for every recipient haplotype. We consider this an exciting avenue
901 of future research.

902
903
904

Full details of how this benchmarking was performed are provided in Additional
file 1, Section D.

Real-data example: recent selection for lactase persistence

LCT is a gene on chromosome 2 that encodes lactase, the enzyme responsible for the breakdown and digestion of lactose, the sugar commonly found in milk. Ancestral humans had a regulatory ‘switch’ on chromosome 2 that stops lactase production after infancy when children would be weaned off breast milk. Mutations that disrupt this switch allow lactase production to persist into adulthood, conferring a lifelong ability to extract energy from milk [19]. Such mutations have arisen independently at least twice in human history, in Europe and in East Africa, and are among the strongest examples of recent positive natural selection in humans [20, 21]. These mutations have been shown to spread across standard human population boundaries. For example, [22] used another implementation of the LS model to compare haplotypes at the *LCT* locus sampled from the West African Fula population to haplotypes collected from across Europe and Asia as part of the 1000 Genomes project [18]. They found that the genetic distance between Fulani haplotypes and Eurasian haplotypes was unusually small at the *LCT* locus. With some further analysis, they interpreted this as evidence that a European haplotype conferring lactase persistence became prevalent within the West African Fula population due to recent natural selection sometime over the past two thousand years.

Although it is difficult to directly replicate [22] since the Fulani samples they studied are not a part of the 1000 Genomes project, we take inspiration from their analysis. Here we present a small example using *kalis* to informally investigate whether there is evidence of recent gene-flow from Eurasia into any of the African populations in the 1000 Genomes dataset at the lactase locus. We run *kalis* on 5008 haplotypes from the 1000 Genomes Phase 3 release to revisit the haplotype structure around

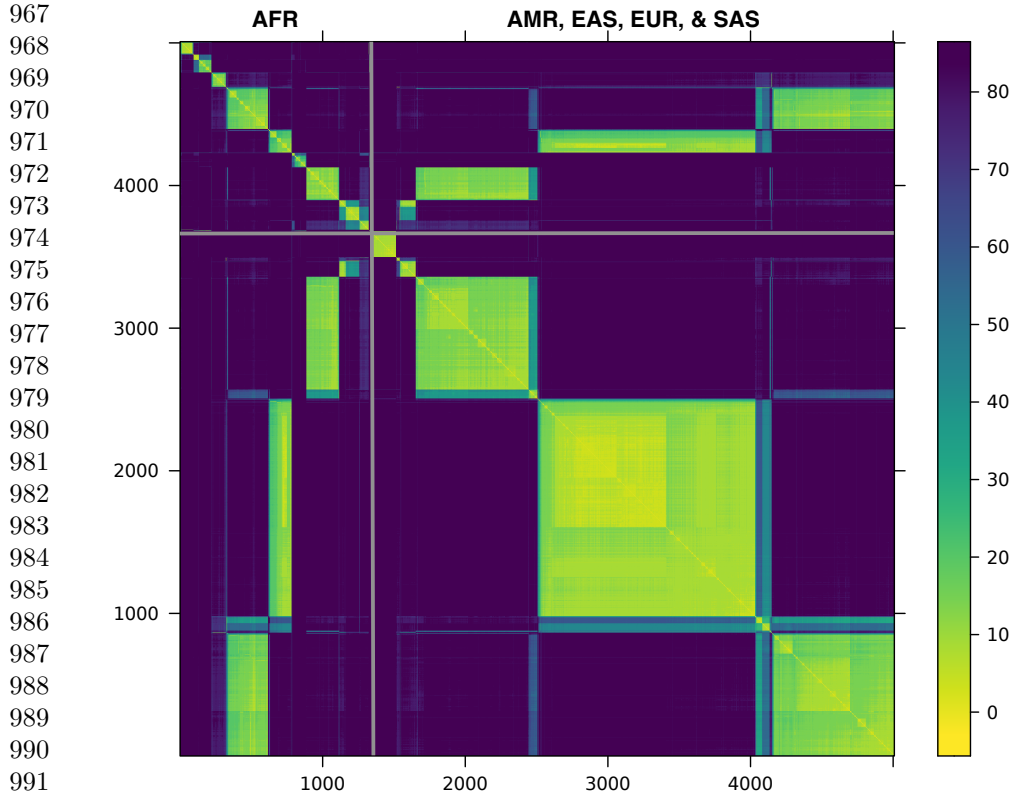


Fig. 5 Distance matrix among 5008 haplotypes calculated at **rs4988235**, upstream of *LCT*. African haplotypes are clustered in the upper left corner and separated by grey lines from non-African haplotypes from the Americas (AMR), East Asia (EAS), Europe (EUR), and SAS (South Asia). The scale on the right maps the colours to distances.

LCT; the haplotypes are sampled from 26 sub-populations all over the world [18]. Figure 5 shows a clustered version of a distance matrix, calculated as in Equation (4), at a variant in the regulatory region of *LCT* (**rs4988235**). To see if we could observe a pattern of gene-flow into or out of Africa similar to what was observed by [22], we use average pairwise linkage [23] to cluster the African haplotypes separately from the non-African haplotypes. In Figure 5, distances between African haplotypes are shown in the upper left corner; non-African haplotypes, in the lower right corner.

Rather than 26 clusters reflecting the 26 sampled human populations, we see that there are three very distinct lactase haplotypes that are common both within and

outside Africa. This suggests that these three haplotypes, under strong positive selection pressure, recently spread across population boundaries and presumably confer lactase persistence. We cannot confirm whether any of these three haplotypes correspond to the one identified in the Fulani by [22]. These three haplotypes are not the only structure we see: in the upper left corner of the African (AFR) block we see some haplotypes that are only found inside Africa; and in the non-African block, a haplotype that is only found outside Africa. We can also see some sub-structure within the clear haplotype blocks.

The code to reproduce this example is available in the **examples** directory of repository associated with this paper (<https://github.com/louisaslett/kalis-bmc>), as a vignette in the package (if vignettes built at install time), or directly at the **kalis** package website https://kalis.louisaslett.com/articles/lct_example.html

Discussion

In Additional file 2, we introduce the package from a user perspective, from package installation right through to decoding a single variant position in R using **kalis**.

There are many avenues for future research in developing **kalis**. On the model side, for example, allowing for different recombination rates between sub-populations as done in fastPHASE [24] would be a natural extension.

On the computational side, ARM scalable vector extensions [25] represent an interesting new approach to SIMD instruction sets, where the width of instructions need not be hard coded prior to compilation. At present it is not widely available, but as this rolls out, it would be natural to extend **kalis** to enable targeting this new instruction set.

An important utility extension is expanding the file formats that **kalis** can natively read via `CacheHaplotypes()`, to enable simpler and more streamlined software pipelines when bioinformaticians incorporate **kalis** into their workflows.

1059 Finally, a future avenue of potential development is extension of **kalis** to support
1060 GPU or tensor cards. Note that it was an explicit design choice to initially target CPU
1061 SIMD extensions, since the vast majority of University high performance computing
1062 clusters have a huge amount of untapped compute power in this form, but often
1063 much more limited availability of specialist extension cards. Therefore, by pushing
1064 performance as extensively as possible via CPU only means, we provide the greatest
1065 potential impact for end users. This does not preclude future versions adding support
1066 for add-on compute cards.

1072 1073 **Conclusion**

1074 **kalis** provides a R interface to a highly optimized C implementation of the LS model
1075 that enables local ancestry, selection, and associations studies in modern large genomic
1076 datasets.

1081 1082 **Availability and requirements**

1083 *Project name:* **kalis**
1084 *Project home page:* <https://kalis.louisaslett.com/>
1085 *Operating system(s):* Linux, MacOS, Windows
1086 *Programming language:* R, C
1087 *Other requirements:* R ($\geq 3.5.0$)
1088 *License:* GPL (≥ 3)
1089 *Any restrictions to use by non-academics:* None beyond GPL (≥ 3).

1095 1096 **List of abbreviations**

1097 LS model = Li & Stephens model
1098 HMM = hidden Markov model
1099 SIMD = single instruction, multiple data

Supplementary information.	1105
	1106
	1107
Declarations	1108
	1109
	1110
Ethics approval and consent to participate. Not applicable.	1111
	1112
Consent for publication. Not applicable.	1113
	1114
Availability of data and materials. The package source code repository is at	1115
https://github.com/louisaslett/kalis . All scripts for reproducing the results of this	1116
paper are available in this repository https://github.com/louisaslett/kalis-bmc . The	1117
two external dependencies are: 1000 Genomes data which are available for download	1118
from https://www.internationalgenome.org/ ; and the msprime simulator, which may	1119
be downloaded from https://tskit.dev/software/msprime.html .	1120
	1121
	1122
	1123
	1124
Competing interests. The authors declare that they have no competing interests.	1125
	1126
	1127
Funding. This project was supported by the NHGRI Centers for Common Disease	1128
Genomics grant (UM1-HG008853), active from 2015-2022.	1129
	1130
	1131
Authors' contributions. LA architected and wrote the C-core. LA and RC collab-	1132
orated on the R interface. RC conducted the real-world lactase persistence example.	1133
LA and RC wrote and approved the final manuscript.	1134
	1135
	1136
Acknowledgements. Both authors would like to acknowledge Professor Ira Hall,	1137
Professor Chris Holmes, and Dr Chris Spencer for their discussions and advice on this	1138
project.	1139
	1140
	1141
	1142
References	1143
	1144
	1145
[1] Li, N. & Stephens, M. Modeling linkage disequilibrium and identifying recom-	1146
bination hotspots using single-nucleotide polymorphism data. <i>Genetics</i> 165 ,	1147
2213–2233 (2003). URL http://www.genetics.org/content/165/4/2213 .	1148
	1149
	1150

1151 [2] Speidel, L., Forest, M., Shi, S. & Myers, S. R. A method for genome-wide
1152 genealogy estimation for thousands of samples. *Nature Genetics* **51**, 1321–1329
1153 (2019).
1154
1155
1156 [3] Song, Y. S. Na Li and Matthew Stephens on Modeling Linkage Disequilibrium.
1157 *Genetics* **203**, 1005–1006 (2016). URL [http://www.genetics.org/content/203/3/](http://www.genetics.org/content/203/3/1005)
1158 [1005](http://www.genetics.org/content/203/3/1005).
1159
1160
1161 [4] Browning, B. L., Tian, X., Zhou, Y. & Browning, S. R. Fast two-stage phasing
1162 of large-scale sequence data. *The American Journal of Human Genetics* **108**,
1163 1880–1890 (2021).
1164
1165
1166 [5] Rubinacci, S., Delaneau, O. & Marchini, J. Genotype imputation using the
1167 positional burrows wheeler transform. *PLoS genetics* **16**, e1009049 (2020).
1168
1169
1170 [6] Kelleher, J. *et al.* Inferring whole-genome histories in large population datasets.
1171 *Nature genetics* **51**, 1330–1338 (2019).
1172
1173
1174 [7] Lawson, D. J., Hellenthal, G., Myers, S. & Falush, D. Inference of population
1175 structure using dense haplotype data. *PLoS genetics* **8**, e1002453 (2012).
1176
1177
1178 [8] Rosen, Y. M. & Paten, B. J. An average-case sublinear forward algorithm for
1179 the haploid Li and Stephens model. *Algorithms for Molecular Biology* **14**, 1–12
1180 (2019).
1181
1182
1183 [9] R Core Team. *R: A Language and Environment for Statistical Computing*. R
1184 Foundation for Statistical Computing, Vienna, Austria (2023). URL [https://](https://www.R-project.org/)
1185 www.R-project.org/.
1186
1187
1188 [10] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in
1189 software. *Dr. Dobbs's Journal* **30**, 202–210 (2005).
1190
1191
1192
1193
1194
1195
1196

[11] Peleg, A. & Weiser, U. MMX technology extension to the Intel architecture.	1197
<i>IEEE Micro</i> 16 , 42–50 (1996).	1198
	1199
	1200
[12] Intel Corporation. Intel Architecture Instruction Set Extensions and Future	1201
Features. Tech. Rep. 319433-046 (2022).	1202
	1203
	1204
[13] ARM. NEON Programmer’s Guide. Tech. Rep. DEN0018A ID071613 (2013).	1205
	1206
	1207
[14] Alpert, D. & Avnon, D. Architecture of the Pentium microprocessor. <i>IEEE Micro</i>	1208
13 , 11–21 (1993).	1209
	1210
	1211
[15] ISO. <i>ISO/IEC 9899:2018 Information technology — Programming languages —</i>	1212
<i>C</i> Fourth edn (BSI, 2018). URL https://www.iso.org/standard/74528.html .	1213
	1214
	1215
[16] Rabiner, L. R. A tutorial on hidden Markov models and selected applications in	1216
speech recognition. <i>Proceedings of the IEEE</i> 77 , 257–286 (1989).	1217
	1218
	1219
[17] Schöne, R., Ilsche, T., Bielert, M., Gocht, A. & Hackenberg, D. IEEE (ed.)	1220
<i>Energy efficiency features of the Intel Skylake-SP processor and their impact on</i>	1221
<i>performance.</i> (ed.IEEE) <i>2019 International Conference on High Performance</i>	1222
<i>Computing & Simulation (HPCS)</i> , 399–406 (2019).	1223
	1224
	1225
	1226
	1227
[18] Consortium, . G. P. <i>et al.</i> A global reference for human genetic variation. <i>Nature</i>	1228
526 , 68 (2015).	1229
	1230
	1231
[19] Ingram, C. J., Mulcare, C. A., Itan, Y., Thomas, M. G. & Swallow, D. M. Lactose	1232
digestion and the evolutionary genetics of lactase persistence. <i>Human genetics</i>	1233
124 , 579–591 (2009).	1234
	1235
	1236
	1237
[20] Ranciaro, A. <i>et al.</i> Genetic origins of lactase persistence and the spread of	1238
pastoralism in Africa. <i>The American Journal of Human Genetics</i> 94 , 496–510	1239
(2014).	1240
	1241
	1242

1243 [21] Bersaglieri, T. *et al.* Genetic signatures of strong recent positive selection at the
1244 lactase gene. *The American Journal of Human Genetics* **74**, 1111–1120 (2004).
1245
1246
1247 [22] Busby, G. *et al.* Inferring adaptive gene-flow in recent African history. *BioRxiv*
1248 205252 (2017).
1249
1250
1251 [23] Sokal, R. R. A statistical method for evaluating systematic relationships. *Univ.*
1252 *Kansas, Sci. Bull.* **38**, 1409–1438 (1958).
1253
1254
1255 [24] Scheet, P. & Stephens, M. A fast and flexible statistical model for large-scale pop-
1256 ulation genotype data: applications to inferring missing genotypes and haplotypic
1257 phase. *The American Journal of Human Genetics* **78**, 629–644 (2006).
1258
1259
1260
1261 [25] Stephens, N. *et al.* The ARM Scalable Vector Extension. *IEEE Micro* **37**, 26–39
1262 (2017).
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288