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

Louis J.M. Aslett[1*] and Ryan R. Christ[2]

[1*]Department of Mathematical Sciences, Durham University, Stockton Road, Durham, DH1 3LE, County Durham, UK.
[2]Department 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 $\boldsymbol{N}$ 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 $\boldsymbol{N} \times \boldsymbol{N}$ distance matrix based on posterior decodings. However, existing posterior decoding implementations for the LS model cannot scale to modern datasets with tens or hundreds of thousands of genomes.

Results We provide a high-performance engine to compute the LS model, enabling users to rapidly develop a range of variant-specific ancestral inference pipelines on top, exposed via an easy to use package, **kalis**, in the statistical programming language R. **kalis** exploits both multi-core parallelism and modern CPU vector instruction sets to enable scaling to problem sizes that would previously have been prohibitively slow to work with.

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

Background

The hidden Markov model (HMM) of haplotype diversity proposed by Li & Stephens [1] (hereinafter, the LS model) has become the basis for several probabilistic phasing, ancestry inference, and demographic inference methods in modern genomics [2, 3]. The LS model provides the basis for the genome-wide ancestry inference software ChromoPainter, which summarizes the ancestry of N haplotypes with an $N \times N$ similarity matrix [4]. This matrix is obtained by running N independent HMMs in which each haplotype is modelled as a mosaic of all of the other haplotypes in the sample. This *'all-vs-all'* copying approach is motivated by the product of approximate conditionals (PAC) likelihood originally proposed by [1] and allows ChromoPainter to render a chromosome-wide estimate of the recent ancestry of the N haplotypes with high resolution.

Beyond chromosome-wide summaries, the LS model can also be used for variant-specific ancestry inference. The contemporary importance of this is highlighted by the recent advances achieved in the RELATE software suite [2], which uses the LS model internally to initialise variant-specific ancestral trees for downstream population genetic analyses ranging from demography to selection inference. **kalis** focuses on providing a high-performance engine to compute exclusively the LS model, enabling users to rapidly develop a range of future variant-specific ancestral inference pipelines on top, in the easy to use statistical programming language R [5].

At the same time, it has been recognised for over a decade [6] 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 [7] 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 [8], 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 [9] 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 [10]. 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 [11], with an easy to use interface provided in R [5]. 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 the (ℓ, i)th element of h. For brevity, let h_i denote the ith haplotype (the ith column of h) and h_{-i} denote all of the haplotypes excluding the ith haplotype. The LS model proposes an HMM for $h_i|h_{-i}$ in which the hidden state at variant ℓ, $X_i^\ell \in \{1, \ldots, N\} \setminus i$, 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 ith 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

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

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

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

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

$$\rho^{\ell} := 1 - \exp\left(-N_e \left(m^{\ell}\right)^{\gamma}\right), \tag{3}$$

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

In keeping with the nomenclature introduced by [4], we refer to h_i as the "recipient haplotype" and the remaining haplotypes, h_{-i}, as the "donor haplotypes", in the context of the HMM where h_i is treated as the emitted observation vector. This reflects the fact that each recipient haplotype h_i is modelled as an imperfectly copied mosaic of the other observed haplotypes under the LS model. Hence, the posterior marginal probability at variant ℓ, $p_{ji}^{\ell} := \mathbb{P}\left(X_i^{\ell} = j \middle| h\right)$, is the probability that donor j is copied

by recipient i at variant ℓ given the haplotypes h. Under the above definitions of the prior copying probabilities Π, the emission kernel (1), and the transition kernel (2), the full $N \times N$ matrix of copying probabilities at ℓ, p^ℓ, can be obtained by running the standard forward and backward recursions [12] for each column (ie for each independent HMM).

From these posterior probabilities, we calculate a local $N \times N$ distance matrix, d^ℓ. Firstly, notice that theoretically $p^\ell_{ij} > 0$, but it can be that $p^\ell_{ij} < \varepsilon$, where ε is the double precision machine epsilon ($\approx 2.22 \times 10^{-16}$, [11], pp.26). Effectively this means d^ℓ_{ij} is too large to reliably work with precisely, and so for the purposes of distance calculations we treat ε as the smallest observable posterior probability, yielding

$$d^\ell_{ji} = -\frac{\log\left(p^\ell_{ji} \vee \varepsilon\right) + \log\left(p^\ell_{ij} \vee \varepsilon\right)}{2} \quad \forall\, j \neq i \tag{4}$$

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:

$$\tilde{\alpha}^\ell_{ji} = \mathbb{P}\left(X^\ell_i = j, h^{1:\ell}_i\right), \qquad \tilde{\beta}^\ell_{ji} = \mathbb{P}\left(h^{\ell+1:L}_i \middle|\, X^\ell_i = j\right), \qquad i \in \{1, \ldots, N\},$$

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

Define,

$$F_i^\ell := \sum_{j=1}^{N} \tilde{\alpha}_{ji}^\ell \qquad F_i^0 := 1 \tag{5}$$

$$G_i^\ell := \sum_{j=1}^{N} \tilde{\beta}_{ji}^{\ell+1} \theta_{ji}^{\ell+1} \pi_{ji} \qquad G_i^L := 1 \tag{6}$$

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

$$\tilde{\alpha}_{\cdot i}^\ell \leftarrow \theta_{\cdot i}^\ell \left(\left(1 - \rho^{\ell-1}\right) \tilde{\alpha}_{\cdot i}^{\ell-1} + \rho^{\ell-1} F_i^{\ell-1} \pi_{\cdot i} \right) \qquad \text{for } \ell \in \{2, \ldots, L\}, \tag{7}$$

$$\tilde{\beta}_{\cdot i}^\ell \leftarrow \left(1 - \rho^\ell\right) \tilde{\beta}_{\cdot i}^{\ell+1} \theta_{\cdot i}^{\ell+1} + \rho^\ell G_i^\ell \qquad \text{for } \ell \in \{1, \ldots, L-1\}. \tag{8}$$

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 the Appendix (Additional file 1) for details). Thus, in full for $\ell \in \{1, \ldots, L\}$ we compute,

$$\alpha_{\cdot i}^1 \leftarrow \theta_{\cdot i}^1 \pi_{\cdot i} \qquad \text{for} \quad \ell = 1 \tag{9}$$

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

and

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

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

Given $\alpha^{\ell}_{\cdot i}$ and $\beta^{\ell}_{\cdot i}$, the vector of posterior probabilities for recipient i, $p^{\ell}_{\cdot i}$, can be calculated directly by normalising,

$$p^{\ell}_{\cdot i} = \frac{\alpha^{\ell}_{\cdot i} \odot \beta^{\ell}_{\cdot i}}{\sum_j \alpha^{\ell}_{ji} \odot \beta^{\ell}_{ji}} \tag{13}$$

where $\odot$ denotes the Hadamard product. In the event that $\sum_j \alpha^{\ell}_{ji} \odot \beta^{\ell}_{ji} = 0$, 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 $p^{\ell}_{ji} = \varepsilon \; \forall \; j \neq i$.

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 the Appendix (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

The R interface described hereinbefore is a thin wrapper layer around a high-performance implementation of the core algorithm which is written in standards compliant C18 [11]. Most data structures are represented with native R types enabling user inspection and manipulation, except for the haplotype sequences themselves.

Computationally, the innermost forward and backward recursions are implemented using compiler intrinsics to exploit a variety of modern CPU instruction sets, including Streaming SIMD Extensions (SSE2 and SSE4.1), Advanced Vector Extensions (AVX, AVX2, AVX-512 and FMA) and Bit Manipulation Instructions (BMI2) on Intel platforms; as well as NEON on ARM platforms. AVX2 is supported in Intel CPUs of the Haswell generation (released Q2 of 2013) or later, AVX-512 tends to be available only in recent Intel server and workstation grade CPUs, and NEON is available for ARM

Cortex-A and Cortex-R series CPUs, as well as Apple M1/M2 and Amazon Web Services Graviton processors. Although this covers most CPUs likely to be in use today, we none-the-less provide reference implementations in pure standards compliant C which will operate on any CPU architecture with a C18 compliant compiler. During package compilation, the correct code-paths are compiled based on detection of the presence or absence of the required instruction sets, or at the direction of the user via compiler flags. See the Appendix (Additional file 1) for more details, and for guidance on how to directly check your CPU for SIMD support.

It may be worth noting at this juncture that it was an explicit design choice to target CPUs and not GPU or tensor cards initially. This is because most University high performance computing clusters have plentiful CPU resources, often with untapped power in advanced SIMD instructions sets. We believe that the problem size that can be realistically tackled in many genetics studies can be massively increased *without* needing to resort to add-on cards, though to scale beyond even this we may explore heterogeneous computing architectures in future **kalis** research.

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

Data structures

There are three user accessible data structures utilised in the package and a low level binary haplotype representation which is not directly user accessible. The two principle data structures of interest to users are forward and backward table objects, represented as native R lists with respective S3 class names `kalisForwardTable` and `kalisBackwardTable` (detailed in Table 2 and discussed later), which are created with package functions `MakeForwardTable()` and `MakeBackwardTable()` respectively. The third user accessible data structure holds the LS model parameters, represented as

a native R environment with S3 class name `kalisParameters`, which can be created with the package function `Parameters()`.

Haplotype data

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

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

When `CacheHaplotypes()` loads haplotypes into the cache, they are interleaved into a flat memory space which is organised as variant-major. That is, variant 1 of each haplotype is loaded, converted to a binary 0/1 and then 32 consecutive haplotypes are 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 instructions[1], 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 $64 \times 8 = 512$

[1] Certain 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.

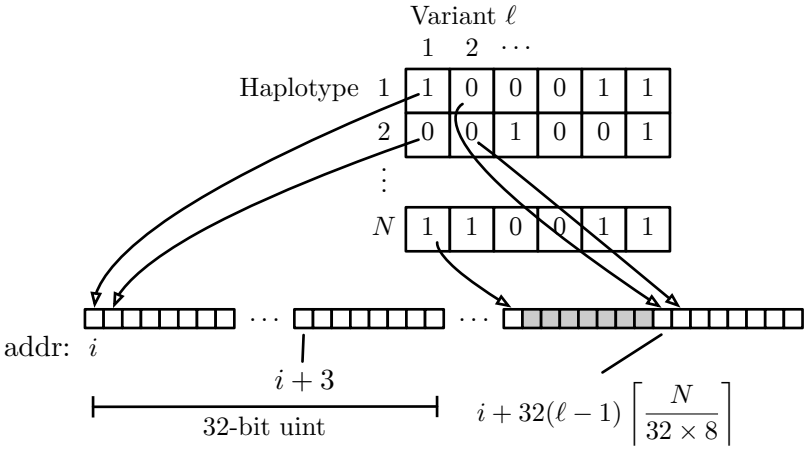

Fig. 1 Efficient binary representation of interleaved haplotypes in memory, with 32-byte boundary alignment for each variant start for SSE/AVX instructions (here $i \mod 32 = 0$). The grey boxes indicate essentially 'wasted' bits which are ignored to ensure alignment for the start of the next variant.

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 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

`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.

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 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

`kalisForwardTable` object		`kalisBackwardTable` object		Data type
`alpha`	$= \alpha_{\cdot\cdot}^{\ell}$	`beta`	$= \beta_{\cdot\cdot}^{\ell}$	$N \times N$ matrix
`alpha.f`	$= F^{\ell}$	`beta.g`	$= G^{\ell}$	vector length N
`l`	$= \ell$	`l`	$= \ell$	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.

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.

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

Core SIMD code

The two most important core algorithms which are accelerated with SIMD vector instructions are the forward and backward recursions. This code is fully implemented

[2] Measured under R 4.2.2

in C, with tailored modifications accounting for all combinations of: scalar/vector μ, scalar/matrix Π, and use of the asymmetric mutation model of RELATE [2] or not (ie 8 combinations); to ensure that minimal memory accesses are performed where possible. So, for example, scalar μ and scalar Π parameters will be faster than any other combination since these values are likely to be held in registers (or at least L1 cache) for the duration of the recursion.

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

```
// Extract from src/StencilVec.h
#if defined(KALIS_ISA_AVX512)
#define KALIS_DOUBLEVEC_SIZE 8
#define KALIS_MUL_DOUBLE(X, Y) _mm512_mul_pd(X, Y)
...
#elif defined(KALIS_ISA_AVX2)
#define KALIS_DOUBLEVEC_SIZE 4
#define KALIS_MUL_DOUBLE(X, Y) _mm256_mul_pd(X, Y)
...
#elif defined(KALIS_ISA_NEON)
#define KALIS_DOUBLEVEC_SIZE 2
#define KALIS_MUL_DOUBLE(X, Y) vmulq_f64(X, Y)
...
```

```
#elif defined(KALIS_ISA_NOASM)
#define KALIS_DOUBLEVEC_SIZE 1
#define KALIS_MUL_DOUBLE(X, Y) (X) * (Y)
...
#endif
```

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 `pthreads` 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.

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 more extended stretches of the genome. This is because every effort has been made to

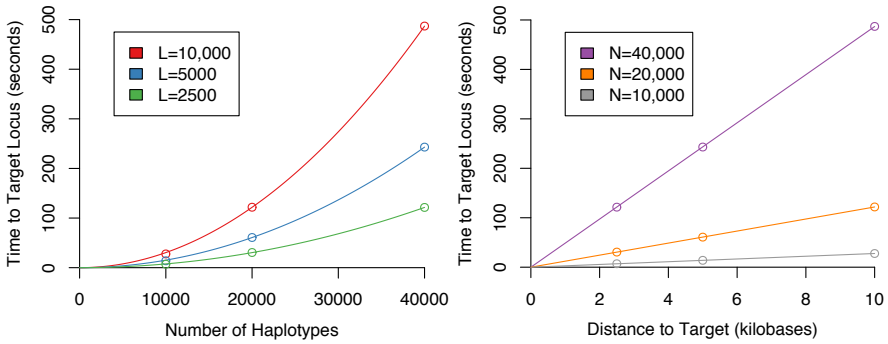

Fig. 2 **kalis** shows the expected order N^2 and order L scaling of the LS model. Computed on an Amazon Web Services `c4.8xlarge` instance (36 vCPUs, 60 GB of RAM).

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 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

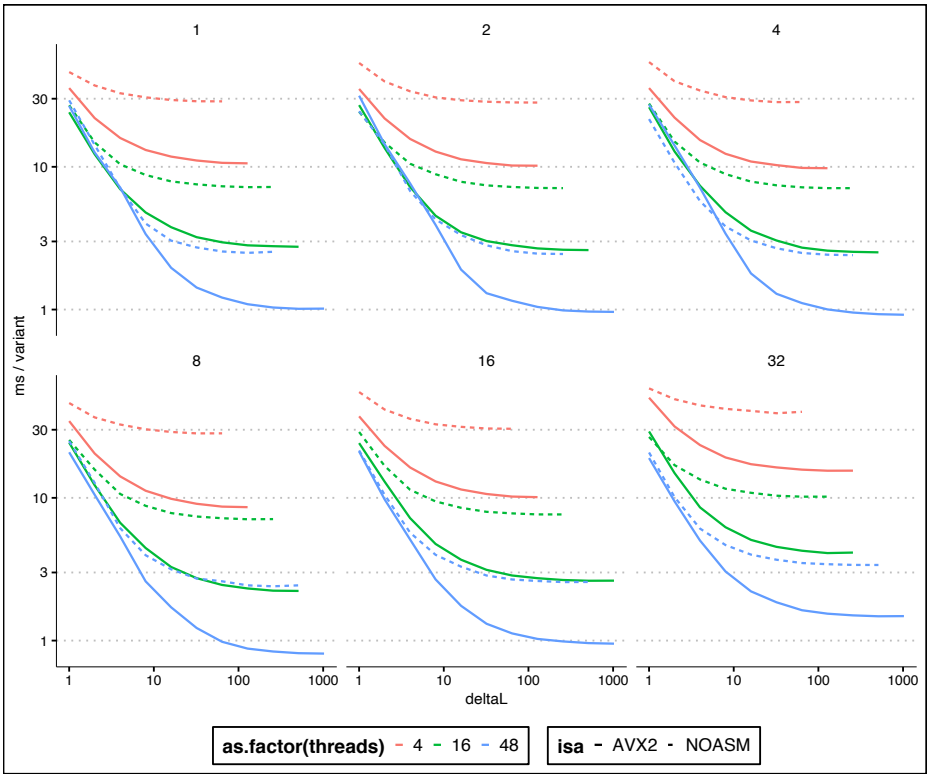

Fig. 3 Log-log plot of milliseconds per variant performance (y-axis) of the forward algorithm on 10,000 haplotypes, against the number of variants propagated (x-axis). Each panel is a different loop unrolling depth (panel title gives loop unrolling level). Line colour denotes number of CPU threads, whilst 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.

investigation, though this may be a result of thermal/power throttling which is known to occur especially for AVX-512 heavy code [13].

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.

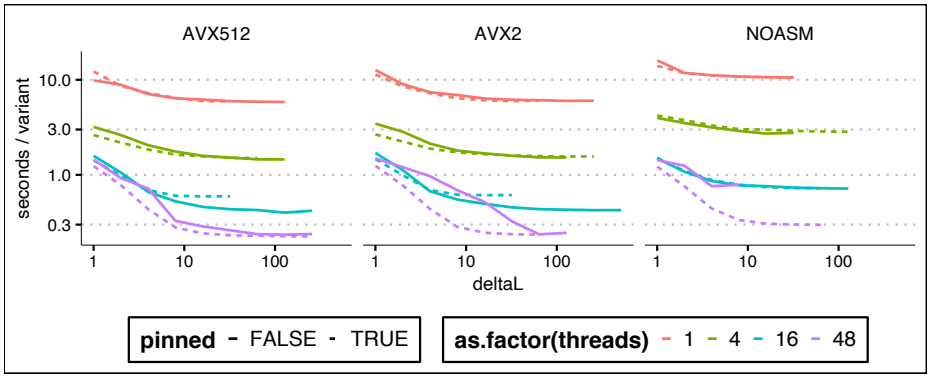

Fig. 4 Log-log plot of seconds per variant performance (y-axis) of the forward algorithm on 100,000 haplotypes, against the number of variants propagated (x-axis). Each panel is a different instruction set (AVX-512/AVX2/none). Line colour denotes number of CPU threads, whilst a dashed line indicates 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.

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 [14]. 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 [15, 16]. These mutations have been shown to spread across standard human population boundaries. Using another implementation of the LS model, [17] estimated that a European haplotype conferring lactase persistence became prevalent within the West African Fula population due to natural selection sometime over the past two thousand years.

Here we run **kalis** on 5008 haplotypes from the 1000 Genomes Phase 3 release to revisit the haplotype structure around *LCT*; the haplotypes are sampled from 26 sub-populations all over the world [18]. Figure 5 shows a clustered version of a

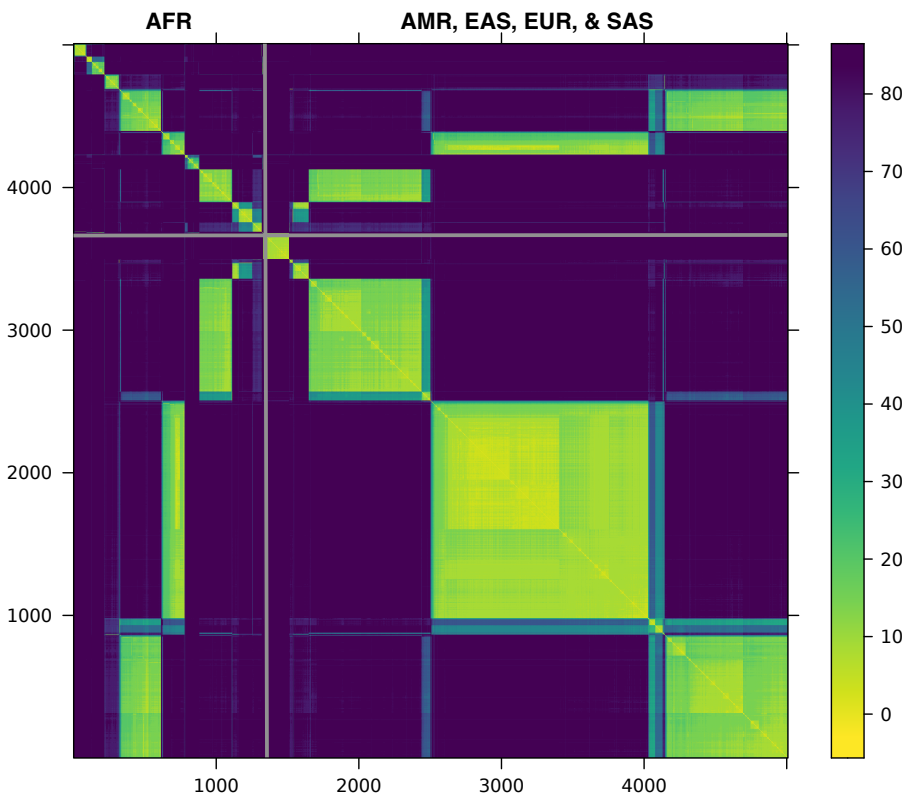

Fig. 5 Distance matrix among 5008 haplotypes calculated at `rs4988235`, upstream of *LCT*. African haplotypes are clustered in the upper left corner and separated by grey lines from non-African haplotypes from the Americas (AMR), East Asia (EAS), Europe (EUR), and SAS (South Asia). The scale on the right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

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 [17], we use average pairwise linkage [19] 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. However, 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.

Discussion

In the User Guide (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 [20] would be a natural extension.

On the computational side, ARM scalable vector extensions [21] 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.

Finally, a future avenue of potential development is extension of **kalis** to support GPU or tensor cards. Note that it was an explicit design choice to initially target CPU SIMD extensions, since the vast majority of University high performance computing clusters have a huge amount of untapped compute power in this form, but often much more limited availability of specialist extension cards. Therefore, by pushing

performance as extensively as possible via CPU only means, we provide the greatest potential impact for end users. This does not preclude future versions adding support for add-on compute cards.

Conclusion

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

Availability and requirements

Project name: **kalis**

Project home page: https://kalis.louisaslett.com/

Operating system(s): Linux, MacOS, Windows

Programming language: R, C

Other requirements: R ($\geq$ 3.5.0)

License: GPL ($\geq$ 3)

Any restrictions to use by non-academics: None beyond GPL ($\geq$ 3).

List of abbreviations

LS model = Li & Stephens model

HMM = hidden Markov model

SIMD = single instruction, multiple data

Supplementary information.

Additional file 1. `Additional_file_1.pdf`

File format: pdf

Title of data: Appendix

An appendix to the main paper containing: full mathematical derivation of the reformulation for the hidden Markov model; detailed guidance on package compilation options and installation; explanation of the HDF5 file format supported by the package.

Additional file 2. `Additional_file_2.pdf`

File format: pdf

Title of data: User Guide

A tutorial-style introduction to using **kalis**, including: installation and package loading; overview of the package API; loading haplotype data; specifying Li & Stephens model parameters; setting up hidden Markov model storage; running the Li & Stephens model; and an example of decoding a single variant.

Declarations

Ethics approval and consent to participate. Not applicable.

Consent for publication. Not applicable.

Availability of data and materials. The package source code repository is at https://github.com/louisaslett/kalis. All scripts for reproducing the results of this paper are available in this repository https://github.com/louisaslett/kalis-bmc. The two external dependencies are: 1000 Genomes data which are available for download from https://www.internationalgenome.org/; and the msprime simulator, which may be downloaded from https://tskit.dev/software/msprime.html.

Competing interests. The authors declare that they have no competing interests.

Funding. This project was supported by the NHGRI Centers for Common Disease Genomics grant (UM1-HG008853), active from 2015-2022.

Authors' contributions. LA architected and wrote the C-core. LA and RC collaborated on the R interface. RC conducted the real-world lactase persistence example. LA and RC wrote and approved the final manuscript.

Acknowledgements. Both authors would like to acknowledge Professor Ira Hall, Professor Chris Holmes, and Dr Chris Spencer for their discussions and advice on this project.

References

[1] Li, N. & Stephens, M. Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. *Genetics* **165**, 2213–2233 (2003). URL http://www.genetics.org/content/165/4/2213.

[2] Speidel, L., Forest, M., Shi, S. & Myers, S. R. A method for genome-wide genealogy estimation for thousands of samples. *Nature Genetics* **51**, 1321–1329 (2019).

[3] Song, Y. S. Na Li and Matthew Stephens on Modeling Linkage Disequilibrium. *Genetics* **203**, 1005–1006 (2016). URL http://www.genetics.org/content/203/3/1005.

[4] Lawson, D. J., Hellenthal, G., Myers, S. & Falush, D. Inference of population structure using dense haplotype data. *PLoS genetics* **8**, e1002453 (2012).

[5] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria (2023). URL https://www.R-project.org/.

[6] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal* **30**, 202–210 (2005).

[7] Peleg, A. & Weiser, U. MMX technology extension to the Intel architecture. *IEEE Micro* **16**, 42–50 (1996).

[8] Intel Corporation. Intel Architecture Instruction Set Extensions and Future Features. Tech. Rep. 319433-046 (2022).

[9] ARM. NEON Programmer's Guide. Tech. Rep. DEN0018A ID071613 (2013).

[10] Alpert, D. & Avnon, D. Architecture of the Pentium microprocessor. *IEEE Micro* **13**, 11–21 (1993).

[11] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C* Fourth edn (BSI, 2018). URL https://www.iso.org/standard/74528.html.

[12] Rabiner, L. R. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* **77**, 257–286 (1989).

[13] Schöne, R., Ilsche, T., Bielert, M., Gocht, A. & Hackenberg, D. IEEE (ed.) *Energy efficiency features of the Intel Skylake-SP processor and their impact on performance.* (ed.IEEE) *2019 International Conference on High Performance Computing & Simulation (HPCS)*, 399–406 (2019).

[14] Ingram, C. J., Mulcare, C. A., Itan, Y., Thomas, M. G. & Swallow, D. M. Lactose digestion and the evolutionary genetics of lactase persistence. *Human genetics* **124**, 579–591 (2009).

[15] Ranciaro, A. *et al.* Genetic origins of lactase persistence and the spread of pastoralism in Africa. *The American Journal of Human Genetics* **94**, 496–510 (2014).

[16] Bersaglieri, T. *et al.* Genetic signatures of strong recent positive selection at the lactase gene. *The American Journal of Human Genetics* **74**, 1111–1120 (2004).

[17] Busby, G. *et al.* Inferring adaptive gene-flow in recent African history. *BioRxiv* 205252 (2017).

[18] Consortium, . G. P. *et al.* A global reference for human genetic variation. *Nature* **526**, 68 (2015).

[19] Sokal, R. R. A statistical method for evaluating systematic relationships. *Univ. Kansas, Sci. Bull.* **38**, 1409–1438 (1958).

[20] Scheet, P. & Stephens, M. A fast and flexible statistical model for large-scale population genotype data: applications to inferring missing genotypes and haplotypic phase. *The American Journal of Human Genetics* **78**, 629–644 (2006).

[21] Stephens, N. *et al.* The ARM Scalable Vector Extension. *IEEE Micro* **37**, 26–39 (2017).