A FAST LASSO-BASED METHOD FOR INFERRING PAIRWISE INTERACTIONS

KIERAN ELMES, ASTRA HEYWOOD, ZHIYI HUANG, AND ALEX GAVRYUSHKIN✉

Abstract. Large-scale genotype-phenotype screens provide a wealth of data for identifying molecular alternations associated with a phenotype. Epistatic effects play an important role in such association studies. For example, siRNA perturbation screens can be used to identify pairwise gene-silencing effects. In bacteria, epistasis has practical consequences in determining antimicrobial resistance as the genetic background of a strain plays an important role in determining resistance. Existing computational tools which account for epistasis do not scale to human exome-wide screens and struggle with genetically diverse bacterial species such as *Pseudomonas aeruginosa*. Combining earlier work in interaction detection with recent advances in integer compression, we present a method for epistatic interaction detection on sparse (human) exome-scale data, and an R implementation in the package `Pint`. Our method takes advantage of sparsity in the input data and recent progress in integer compression to perform lasso-penalised linear regression on all pairwise combinations of the input, estimating up to 200 million potential effects, including epistatic interactions. Hence the human exome is within the reach of our method, assuming one parameter per gene and one parameter per epistatic effect for every pair of genes. We demonstrate `Pint` on both simulated and real data sets, including antibiotic resistance testing and siRNA perturbation screens.

1. Introduction

Epistatic gene interactions have practical implications for personalised medicine, and synthetic lethal interactions in particular can be used in cancer treatment [3]. Discovering these interactions is currently challenging [23, 17, 10, 12], however. In particular, there are no methods able to automatically infer interactions from genotype-phenotype data at the human genome scale.

For a given number of genes there are exponentially many potential interactions, complicating computational methods. If we restrict our attention to pairwise effects, it is possible to experimentally knock out particular combinations of genes to determine their combined effect [9]. This approach does not scale to the approximately 200 million pairwise combinations possible among human protein coding genes, however. We instead consider inferring pairwise interactions from large-scale genotype-phenotype data. These include mass knockdown screens, in which we suppress a large number of genes simultaneously, and attempt to measure the resulting phenotypic effect.

We have shown in [12] that a lasso-based approach to inferring interactions from an siRNA perturbation matrix is a feasible method for large-scale interaction detection. In this additive model, we assume fitness is a linear combination of the effects of each gene's effect, and the effect of every combination of these genes. For the sake of scalability, we consider only individual and pairwise effects, and assume gene suppression is strictly binary. The fitness difference f (compared to no knockdowns) in an experiment e is then the sum of individual and pairwise effects $\sum_i^p g_i + \sum_i^p \sum_{j>i}^n g_i \cdot g_j$, where $g_i = 1$ if gene i is knocked down, 0 otherwise. With sufficiently many such mass-knockdowns, we can infer pairwise interactions by finding the pairs of genes whose effect is not the sum of the effects of each gene individually.

Department of Computer Science, University of Otago, New Zealand
E-mail addresses: ✉alex@biods.org.

We acknowledge support from the Royal Society Te Apārangi through a Rutherford Discovery Fellowship (RDF-UOO1702). This work was partially supported by Ministry of Business, Innovation, and Employment of New Zealand through an Endeavour Smart Ideas grant (UOOX1912) and a Data Science Programmes grant (UOAX1932).

Neither of the previously tested inference methods for this model, `glinternet` and `xyz`, are effective at the genome-scale however. `glinternet` suffers from prohibitively long running times,[1] and `xyz` does not accurately predict effects in our larger simulations. Our aim is to fit a model including all $p \approxeq 20,000$ human protein-coding genes, with as many as $n = 200,000$ siRNAs. Doing so requires the development of new methods and software.

We have developed an R-package that is able to perform lasso regression on all pairwise interactions on the same one thousand gene screen in twenty seconds, and is able to fit a genome-scale data set with $19,000$ genes and $67,000$ siRNAs in under two hours using a single eight-core CPU. This is made possible by taking into account that our input matrix X is both sparse and strictly binary. Our package, `Pint`, is available at github.com/biods/pint.

To perform lasso-based regression on this matrix, we begin with an existing fast algorithm, parallelise it, and adapt it for use on our binary perturbation matrices. We provide a detailed explanation of this implementation, followed by the scalability analysis, below. We also perform a simulation study to compare our method's scalability with known methods, and analyse two large-scale experimental data sets.

In the first, an siRNA perturbation screen from [31], we search for pairs of genes that have an epistatic effect when simultaneously silenced. Out of five top interactions identified by our method, two are known protein interactions and three appear to be novel.

The second data set is composed of genetic variants identified in the intrinsically antibiotic resistant bacteria *Pseudomonas aeruginosa*. *P. aeruginosa*is an opportunistic pathogen found in a variety of environments and is a leading cause of morbidity and mortality in immunocompromised individuals or those with cystic fibrosis [16, 24]. *P. aeruginosa*is known to acquire adaptive antibiotic resistance in response to long term usage of antibiotics associated with chronic infections [5, 27, 28]. The genomes included in that data set are from strains that have been isolated from chronic and acute infections as well as environmental samples. The minimum inhibitory concentration for the antibiotic Ciprofloxacin has been used as the phenotypic marker for this dataset. Ciprofloxacin belongs to the fluoroquinolone class of bacteriocidal antibiotics that targets DNA replication and is one of the most widely used antibiotics against *P. aeruginosa*[34]. Our findings identified 16 pairs of interactions, most of which were found in genes that are important in biofilm formation and maintenance, a characteristic of intrinsically antibiotic resistant bacteria.

2. Methods

Our goal is to estimate both the main effects $\beta_1, \ldots, \beta_p$, and the interaction effects $\beta_{1,2}, \ldots, \beta_{p-1,p}$ where pairs of genotypes are simultaneously perturbed. As an example, consider pairwise effects in a siRNA perturbation screen. We can estimate the effect of both silencing individual genes $(\beta_1, \ldots)$ and pairs of genes simultaneously $(\beta_{1,2}, \ldots)$. To do this we add a column for each pair of genes, converting the siRNA matrix $\mathbf{X} \in \{0,1\}^{n \times p}$ into the pairwise matrix $\mathbf{X}_2 \in \{0,1\}^{n \times p'}$, where $p' = \frac{p(p+1)}{2}$. This model includes all pairwise interactions and fitting it is equivalent to finding epistasis as in [12]. The same construction applies to any binary genotype-phenotype data, and the effect $\beta_{a,b}$ will always estimate the simultaneous effect of both genes a and b.

We construct the matrix $\mathbf{X}_2$ as follows. For every column i from 0 to n we take every further column j from $i+1$ to n and form a new column by taking the bit-wise *and* over all elements of the columns i and j (Fig. 1).

[1] Finding interactions in an siRNA screen of $1,000$ genes with ten siRNAs per gene takes several days using ten cores on an Opteron 6276.

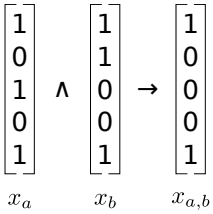


FIGURE 1. Creation of pairwise siRNA effect columns

This gives us the complete pairwise matrix $\mathbf{X}_2$, shown in Fig. 2.

Gene 1 Gene p Genes 1 and 2 Genes 1 and p Genes $p-1$ and p

$$\beta_1 x_{1,1} + \ldots + \beta_p x_{1,p} + \beta_{1,2} x_{1,1\wedge 2} + \ldots + \beta_{1,p} x_{1,1\wedge p} + \ldots + \beta_{p-1,p} x_{1,p-1\wedge p} + \mathscr{E}_1 = y_1$$

$$\beta_1 x_{2,1} + \ldots + \beta_p x_{1,p} + \beta_{1,2} x_{2,1\wedge 2} + \ldots + \beta_{1,p} x_{2,1\wedge p} + \ldots + \beta_{p-1,p} x_{2,p-1\wedge p} + \mathscr{E}_2 = y_2$$

$$\vdots$$

$$\beta_1 x_{n,1} + \ldots + \beta_p x_{n,p} + \beta_{1,2} x_{n,1\wedge 2} + \ldots + \beta_{1,p} x_{n,1\wedge p} + \ldots + \beta_{p-1,p} x_{n,p-1\wedge p} + \mathscr{E}_n = y_n$$

FIGURE 2. Matrix of Pairwise siRNA effects

2.1. **Cyclic Linear Regression.** Our approach to lasso regression is based on a cyclic coordinate descent algorithm from [15], as described in [40]. This method begins with $\beta_j = 0$ for all j and updates the beta values sequentially, with each update attempting to minimise the current total error. Here this total error is the difference between the effects we have estimated and the fitness we observe, given the genes that have been knocked down. Where y_i is the ith element of $\mathbf{Y}$, β_j is the jth element of β, and x_{ij} is the entry in the matrix $\mathbf{X}_2$ at column j of row i, the error is the following.

$$\sum_{i=1}^{n} |y_i - \sum_{j=1}^{p'} x_{ij} \cdot \beta_j| \tag{1}$$

The error affected by a single beta value (Eq. (5)) can then be minimised by updating β_k with the following:

$$\Delta\beta_k = \begin{cases} max(0, \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} - \lambda) & \text{for } \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} > 0 \\ min(0, \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} + \lambda) & \text{for } \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} < 0 \end{cases} \tag{2}$$

We cyclically update each β_k until the solution converges for a particular lambda, reduce the value of lambda, and repeat. See Appendix A for the full derivation and algorithm. Storing the matrix in a sparse column format, this implementation scales up to $p = 1,000$. It would still take several days and use terabytes of memory for $p = 20,000$. To overcome this, we compress the matrix, and parallelise the beta updates (Section 2.3 and Appendix D).

2.2. **Choosing Lambda.** The lasso penalty requires a regularisation parameter lambda. This parameter determines the extent to which we penalise large beta values, and can range from allowing all values ($\lambda = 0$) to allowing only zero ($\lambda \to \infty$). Choosing the correct value of lambda is essential if we want to include only the significant effects. This is typically done by choosing an initial value sufficiently large that all beta values will be zero and gradually reducing lambda, fitting the model for each value until a stopping point chosen with K-fold cross-validation [14]. Cross-validation requires fitting each lambda value K times, however, significantly increasing the runtime. We instead provide two options for choosing lambda in our package. First, we can

choose lambda such that the number of non-zero effects is small enough for OLS regression. In our package, this is called Limited-β and a default limit is $2,000$. Alternatively, we implement a fast method for empirically choosing a reasonable stopping point, the adaptive calibration lambda selection method from [8]. Both of these methods are significantly faster than cross-validation, although using adaptive calibration we tend to predict very few non-zero effects. The best empirical results are generally achieved with the Limited-β approach, and we use this for the remainder of the paper. A detailed explanation of each and their performance impact can be found in Appendix C.

2.3. **Compression.** To reduce memory usage and the time taken to read each column with larger input data, we compress the columns of $\mathbf{X}_2$. Because we read the columns sequentially, we replace each entry with the offset from the previous entry. This reduces the average entry to a relatively small number, rather than the mean of the entire row. These small integers can then be efficiently compressed with any of a range of integer compression techniques (Fig. 3), a subject that has been heavily developed for Information Retrieval. We compare a number of such methods, including the Simple-8b algorithm from [35] (which we implement and use in our package) in Appendix B.

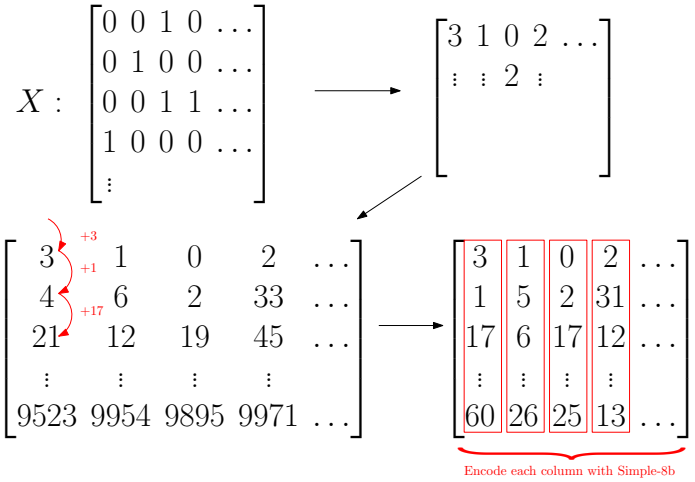
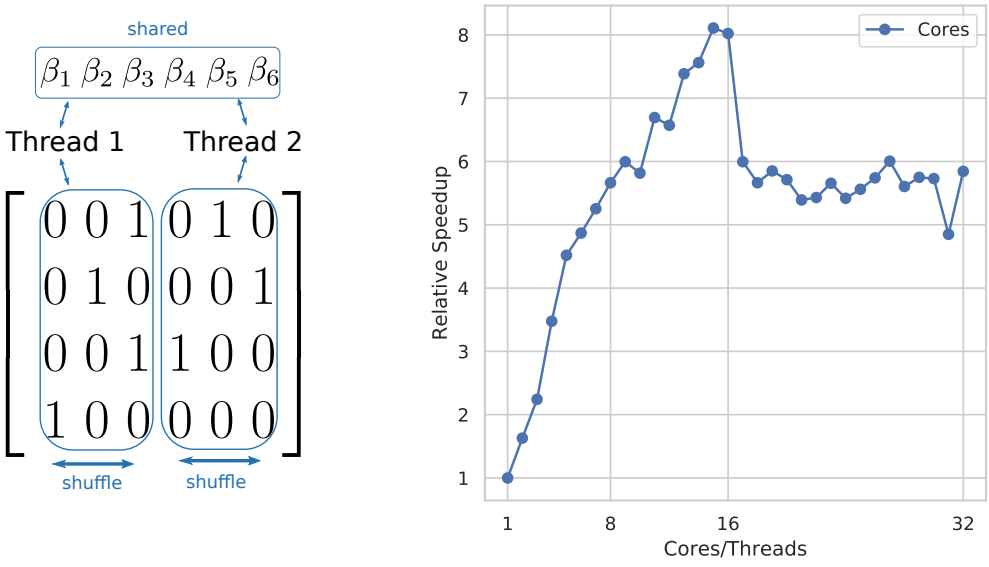


FIGURE 3. Compression of the sparse $\mathbf{X}_2$ matrix.

2.4. **Parallelisation.** While it is trivial to parallelise the update of a single β value, doing so does not improve performance in practice, due to poor cache usage (see Appendix D for details). We instead parallelise our method by assigning whole columns of the compressed $\mathbf{X}_2$ matrix to threads in sections. Each thread is responsible for updating the β values corresponding to its columns, and is therefore the only thread reading it's section of the $\mathbf{X}_2$ matrix.

Simultaneously updating columns with entries in the same row leads to over-compensating for these entries. This can harm performance or in the worst case prevent convergence entirely (Appendices D.2, D.5 and D.5.1). To avoid this, it suffices to ensure that threads do not frequently update the same columns at the same time. We achieve this by shuffling the order each thread updates it's columns every iteration. While it is in principle still possible to update enough overlapping threads in parallel to cause problems, Bradley et al. [6] show that this is rarely a problem in practice.

Updates to the shared β values are atomic, and every thread needs read access to all β values. This limits our method to use on shared memory systems and results in poor performance on NUMA systems. In practice we can only effectively use a single CPU socket, and we have tested this up to eight cores. Fig. 4 summarises the shuffled parallel implementation and it's scalability. For the full details of the parallel implementation see Appendix D.



(a) (b)

FIGURE 4. (a) Each thread is assigned a set of columns, which is then shuffled every iteration.
(b) Relative speedup as the number of cores used increases, running on a dual 8 core/16 thread NUMA system. Cores 1-8 are separate cores on node 1, 8-16 are SMT threads on the same cores. Cores 17-24 are separate cores on the second NUMA node, and 15-32 are SMT threads on those cores.

2.5. **Limited Interaction Neighbourhoods.** When searching for interactions within a large sequence, it may be acceptable to limit the search to pairs that are relatively close on the genome. In a study of epistatic interactions in yeast by Puchta et al. [30] the strength of negative interactions decreases as distance between gene positions on the sequence increases. The median distance between pairs in the hundred strongest interactions was only eighteen nucleotides.

Limiting interactions to those within some distance d drastically reduces both the time and space requirements. Instead of $\Theta(p^2n)$, the size of the interaction matrix becomes $\Theta(pdn)$. Similarly, an iteration of Algorithm 3 would require only $\Theta(pdn)$ operations. For $d << p$ this is a significant reduction. Limiting the interaction search distance to 100 positions, we could process a set of $30,000$ genes and $200,000$ siRNAs using approximately $16GB$ of memory, assuming a comparable density of interactions to our testing data. Such a search could be performed directly on a laptop, without requiring access to a large server. The biological implications of this restriction should be carefully considered before its use, however.

2.6. **Data.** We prepared two experimental data sets to evaluate our method and test the scalability of our implementation. The first is an siRNA perturbation screen in which siRNAs targeting kinases are applied to an infected human cell line. We predict off-target effects across the entire exome, and use this larger set for our analysis.

The second data set contains single nucleotide variants (SNVs) from 259 isolates of *Pseudomonas aeruginosa*, and associated minimum inhibitory concentration (MIC) of Ciprofloxacin.

2.6.1. *InfectX siRNA Data.* To demonstrate our method on real genome-scale data, we use the vaccinia group from InfectX [31]. This set contains $204,288$ siRNA perturbations in the presence of the vaccinia virus. This set is significantly larger than the mock group (siRNA perturbations with no pathogen present). Off-target effects are prediction using Risearch2 [1]. We include a gene as an off-target effect whenever there is a match between the siRNA seed region and some

component of an mRNA for that gene (taken from [18]). We use an energy cutoff of -20 and match the entire siRNA, not only the $3'$ UTR, as suggested in [1].

We then form a matrix of off-target effects with columns for each gene, and rows for each siRNA as in [12]. An entry i, j in this matrix is one if and only the predicted effect of siRNA i on gene j is greater than zero. All other entries are zero. Our fitness vector $\mathbf{Y}$ is the result of B-scoring then Z-scoring the number of cells in the well, to remove systematic within-plate effects and experimentally introduced cross-plate biases. B-scoring corrects for biases across the entire plate, and Z-scoring then normalises each well's score with respect to the rest of it's plate.

2.6.2. *Antibacterial Resistance.* SNVs from 259 isolates of *Pseudomonas aeruginosa*were sequenced using illumina technologies (IPCD isolates on MiSeq and QIMR isolates on HISeq). SNV's from raw reads were mapped to the reference genome PAO1 using Bowtie2 (v. 2.3.4) [21] read aligners. Variant reports were then read into a python script which sorted the reports into a table. The table was set up so that each isolate was represented as a row and the presence / absence of each SNV was along the columns. Only genomes that had associated MIC values were included. The resulting table contains 259 rows and over $700,000$ columns.

Since our method considers p^2 interactions, the scale of this data presents a problem. Including all $> 700,000$ columns, we would need to store over 250 billion interaction columns, each with up to 259 entries. Even if every column fits into a single 64-bit word, simply storing the compressed matrix would require on the order of two terabytes of memory. We instead reduce this to a more manageable scale, by removing all duplicated columns, and then any of the remaining columns that have less than 30 entries. Note that this is likely to remove point mutations occurring from acquired resistance, and effects that are always found in the same isolates cannot be distinguished. While it may be possible to address these limitations we do not attempt to do so here. There are simply too many interactions (over 200 billion) among the full set of variants for our current implementation. After these reductions we have a more tractable $259 \times 75,715$ entry matrix, sufficiently small that all approx. 5.7 billion effects and interactions can be processed using under 250GB of memory.

2.6.3. *Data Sources. P. aeruginosa*genome sequences were selected from strains whose MIC values (Ciprofloxacin) were known. 167 genomes were sourced from the publicly available IPCD International Pseudomonas Consortium Database [19] and 92 genomes were from QIMR Brisbane Australia [20]. The IPCD data consisted of 2 x 300 bp MiSeq reads whilst the QIMR data was 2 x 150 bp reads. The MIC values were obtained as a combination of e-test strips [32] and plate-based assays [33].

3. Results

In this section, we summarise the results of a simulation study we carried out to compare our method against existing approaches. We also demonstrate our method on two large-scale experimental data sets. Note that both sets are too large to attempt using known approaches such as `glinternet` for comparison. In both cases the true interactions are unknown, making the true accuracy of our method in these cases difficult to determine. We nonetheless include these as reasonable examples of cases in which our method is applicable and validate the results by comparing them with known protein interactions [36].

3.1. **Simulation Performance.** Our method aims to have comparable precision and recall to the best performing approach in our previous work [12] while scaling to much larger data sets. To evaluate the accuracy of our method, we compare precision and recall of our method with `glinternet`, the most accurate of the methods tested [12].

Since we achieved the best results only using `glinternet` for variable-selection, then fitting the non-zero beta values with ordinary least squares (OLS) regression, we do the same here. We use `Pint` in the same way and restrict to the first $2,000$ non-zero beta values, rather than using adaptive calibration, which returns too few columns for the OLS regression step.

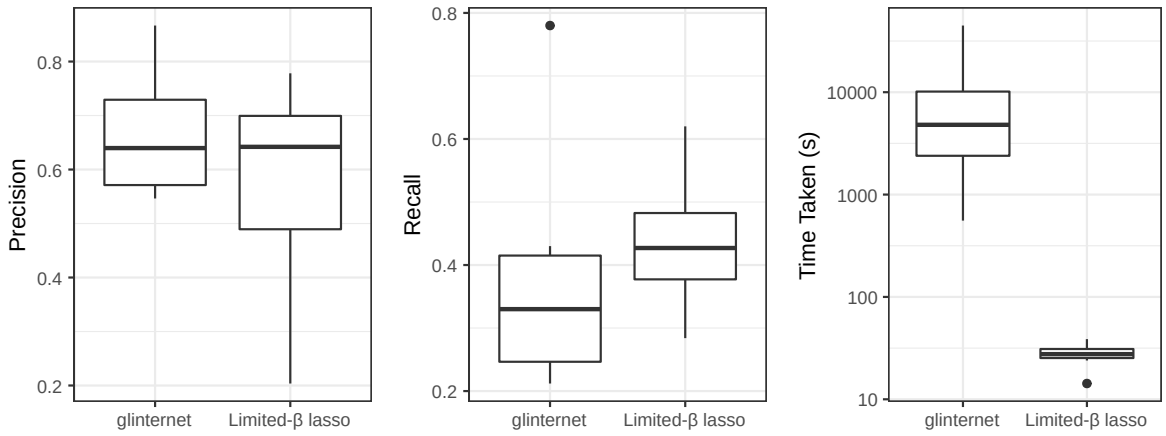
TABLE 1. Runtime comparison between our method and `glinternet`.

X matrix size	`glinternet` time (s)	`Pint` time (s)
$n = 100,\ p = 1,000$	178	2.00
$n = 1,000,\ p = 10,000$	4807	27.6

TABLE 2. Infectx proposed interactions

Gene Names		Estimated Effect	p-value
TTN	KMT2D	0.085	4.35e-05
TTN	PLEC	0.053	1.95e-2
TTN	TTC7B	0.068	2.01e-3
TTN	OBSCN	0.137	8.00e-11
TTN	CDH23	-0.022	1.00e-1

Testing with the same data as in [12], our method is able to identify significantly more correct interactions than `glinternet` (Fig. 5). Precision is largely comparable, with a few outliers in which we see significantly more false positives with our method (Fig. 5a). The run time is orders of magnitude faster than `glinternet`, typically taking 20 to 30 seconds rather than several hours (Table 1 and Fig. 5c). To test the scalability of our implementation, we also run it with the same $2,000$ effect limit on a much larger data set. With $p \approx 27,000, n \approx 30,000$, using 16 SMT threads on a single eight core CPU, we propose 97 main effects and 236 interactions in one and a half hours.



(a) (b) (c)

FIGURE 5. Searching for interactions with `glinternet` vs. our shuffled compressed lasso, using $p = 1,000, n = 10,000$ data from [12]. (a) Precision. (b) Recall. (c) Time taken (log scale).

3.2. **InfectX siRNA Data.** We run our lasso model on the InfectX data (Section 2.6.1) allowing all pairwise interactions, and halting at $\lambda = 0.05$ or the first $2,000$ non-zero effects, whichever comes first. Only the genes and gene-pairs with non-zero predicted effects are then included in the matrix $\mathbf{Z}$. Last, we fit the phenotype $\mathbf{Y}$ to this matrix using least-squares regression $\mathbf{Y} \sim \mathbf{Z}$, using these unbiased estimates and p-values as our final result.

We find 26 proposed effects (21 main and 5 interactions) in under two hours. Our method proposes interactions between five genes and TTN, with varying estimated strengths (see Table 2). Two of these interactions, OBSCN and PLEC, are known protein interactions [36].

We find the same set of interactions in repeated runs (bearing in mind that the matrix is shuffled differently each time). This suggests that these are not random choices, but effects strongly supported by the data. The Adjusted R^2 value is only ≈ 0.088, however, indicating

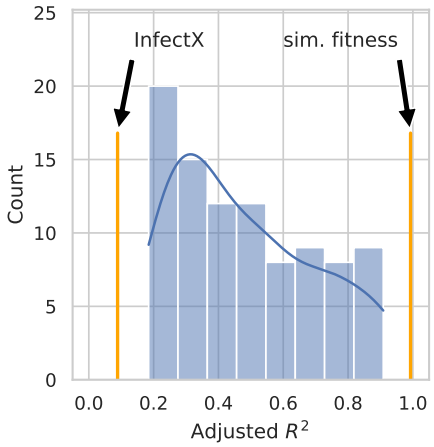


FIGURE 6. Adjusted R^2 density of simulations in Section 3.1, with additional lines indicating the values using InfectX data, or InfectX with simulated fitness, instead.

that while a better than random fit has been found, the chosen effects do not explain the overall observed fitness particularly well. To investigate this, we consider the difference between fitting two different phenotypes. Firstly, the predicted effects with the measured cell counts from InfectX, and secondly a simulated set that reflects our assumptions.

For the simulation we used the same $\mathbf{X}$ matrix, but simulated the fitness effects $\mathbf{Y}$ as a linear combination of randomly chosen gene effects and gene-pair interactions. Every gene had a 10% chance of being assigned an effect, which were sampled from $\mathcal{N}(0, 2)$. We gave every pair of genes a 0.1% chance of an effect, which were also sampled from $\mathcal{N}(0, 2)$. For every row i of $\mathbf{X}$, the fitness value y_i is the sum of both main and interaction effects present, with additional random noise.

$$y_i = \sum_{j=1}^{p} \left(X_{i,j}\ \texttt{effect}(j) + \sum_{k=j+1}^{p} (\texttt{effect}(j,k)) \right) + \mathcal{N}(0, 10)$$

With this simulated phenotype vector, re-running the interaction search with the same parameters, we have an R^2 of ≈ 0.99.

After adjusting for the number of effects proposed, we find that while the fit is better than random using the Z-scored InfectX cell-count as phenotypes, it is not nearly as good as in our simulations. This suggests that at least some of our assumptions are incorrect, namely that our fitness proxy (log cell count) is additive and can be largely explained with individual and pairwise silencing effects, and that the off-target predictions are accurate. While all of these assumptions are somewhat suspect, it should be noted that our siRNA off-target predictions likely miss a significant number of strong effects, and include genes that are not completely silenced [1]. With this in mind, it is plausible that even if the cell count responds to gene silencing according to our assumptions, the predicted effects may not be significantly better than random until accurate siRNA off-target predictions are available.

3.3. **Antibacterial Resistance.** We fit our antibacterial resistance data (see Section 2.6.2) with three different sets of parameters. First, we allow all interactions. Second, we restrict to interactions within 100 columns of each other. Finally, we restrict to interactions within 10 columns of each other. In all cases, we run until the adaptive calibration stopping condition is met. In the first case, allowing all interactions, we find that repeated runs do not suggest any of the same effects. Since the data only contains 259 samples for over 75,000 effects (and over 2.5 billion interactions) it is unsurprising that there are several equally good solutions. We fail to find a reproducible result here because the data simply does not suggest one, and this run is included only to demonstrate that our implementation works at this scale. The second

and third cases produce more consistent results, with some common interactions suggested in both cases. While limiting to interactions within 100 columns rules out the majority of possible interactions, it also limits the number possible solutions enough that we can find one reliably with only 259 samples.

Moreover, when we reduce the interaction matrix to only the non-zero predicted effects, and produce an unbiased fit with least-squares regression, we find that our fit explains the variance in resistance extremely well. Restricting interactions to effects within 100 entries of each other, we have a multiple R^2 of 0.99, and an adjusted R^2 of 0.86. Even limiting to interactions within ten entries, we have a multiple R^2 of 0.78, and an adjusted R^2 of 0.63. These suggest that in this case our model is a particularly good fit.

There were 16 sets of variants found in both limited-distance runs (interactions within 10 or 100 columns only). For each of the 16 SNVs their genes, functions, and interactions were assessed. Genes were identified based on PAO1 reference co-ordinates using Artemis [7]. The STRING[36] database was used to assess the validity of the protein-protein interactions.

Five of the SNV pairs occurred in the same gene. There were four pairs that had high interaction scores > 0.7 and two of pairs were identified twice. Many SNV's were found in genes that encoded for proteins involved in biofilm formation and maintenance indicative of long term chronic infections that are often associated with general antibiotic resistance. Other than pilY1, no other gene was found to be mutated in the lab-based evolution study [33].

There were two pairwise effects that had significant p-values in both runs. The first of these pairs occurred in a gene that encoded a copper resistance protein. The second pair was found in a gene that encodes an RNA binding methyltransferase.

4. Discussion

Genotype-phenotype data sets have recently become available at a never before seen scale. In principle, it is possible to infer not only the effect of individual genomic variants within such data, but of pairwise combinations of their effects. While this has been shown to work in theory, and a number of tools have been developed that work on a smaller scale, there is a shortage of effective methods for human genome-scale data. In this paper we present a regression based method for such large-scale inference of pairwise effects.

Our method performs coordinate descent lasso-regression on a matrix containing all pairwise interactions present in the data. For such an approach to work at scale, we had to make a number of improvements. First we parallelised the algorithm by dividing the matrix into shuffled sets for each thread. We then drastically increased the scale of tractable data sets by compressing columns of the matrix using Simple-8b. Combined with the typically sparse binary nature of genotype-phenotype screens, our method is able to effectively consider hundreds of millions of possible interactions.

We compared the accuracy and running time of our work to `glinternet`, the best of the methods we used previously [12], and found that our method provides comparable accuracy and precision while running hundreds of times faster. We also tested our method using two genome-scale real data sets. One is an exome-wide siRNA perturbation screen ($n \approxeq 67,000$ siRNAs and $p \approxeq 19,000$ genes). The other measures antibacterial resistance with respect to genetic variations in *Pseudomonas aeruginosa*, and includes over two billion possible pairwise interactions. In both cases our method finds a number of effects that are either plausible or previously known.

In some cases we can significantly improve the running time and memory use by only considering local interactions. If interactions are restricted to those within $1,000$ positions of each other, we can search our siRNA screen using $\approx$ 40GB of memory in $\approx$ 20 minutes.

While our method is effective on this scale, there are some limitations that would make it difficult to use on significantly larger data sets. Both the time and space requirements are quadratic in the input sequence, and performance does not scale well with non-uniform memory access. This essentially limits our approach to data that fits in memory on a single machine. The pairwise additive model is also something of an oversimplification. It remains unclear to

what extent genetic effects be treated as additive, and ignoring interactions among of more than two items could well be leaving out the most important effects. In this case we may end up spuriously associating phenotype changes with individual and pairwise effects that just happen to be present, rather than the true, more complicated, interaction.

There are nonetheless a number of opportunities to expand upon this work. If the original $\mathbf{X}$ matrix is sparse, and the pairwise interaction matrix $\mathbf{X}_2$ is very sparse, we would expect three-way interaction columns of an $\mathbf{X}_3$ matrix to be even more so. If there are few enough non-zeros in such a matrix, it may be possible to extend our method beyond pairwise interactions without any fundamental changes. While there would be p^3 columns in a three-way interaction matrix, if the vast majority contain only zeros we may still be able to store it. The indices of non-zero three-way interaction columns could themselves be stored in a compressed list of offsets. Any column whose index is not in this list could then be presumed to be zero and left out of beta updates. Since the memory and time requirements only grow with the number of non-zero entries, this could provide a well be enough for sufficiently sparse data.

Alternatively, as we showed in Section 2.5, we can significantly increase the scale of interaction inference methods by reducing the search space. A more targeted approach than restricting the genome distance, estimating distance in 3D space using Hi-C [4] for example, would drastically reduce the time and space requirements, allowing higher order interactions to be considered.

Finally, the interactions proposed in Section 3.2 that have not already been confirmed may well be real, and are worth further investigation.

Our method is implemented in C, and an R package is provided at github.com/bioDS/pint.

References

[1] Ferhat Alkan et al. “RIsearch2: Suffix Array-Based Large-Scale Prediction of RNA–RNA Interactions and siRNA off-Targets”. In: *Nucleic Acids Research* 45.8 (May 5, 2017), e60–e60. ISSN: 0305-1048. DOI: 10.1093/nar/gkw1325. URL: https://academic.oup.com/nar/article/45/8/e60/2929519 (visited on 11/17/2020).

[2] Vo Ngoc Anh and Alistair Moffat. “Inverted Index Compression Using Word-Aligned Binary Codes”. In: *Information Retrieval* 8.1 (Jan. 1, 2005), pp. 151–166. ISSN: 1573-7659. DOI: 10.1023/B:INRT.0000048490.99518.5c. URL: https://doi.org/10.1023/B:INRT.0000048490.99518.5c (visited on 08/30/2020).

[3] Alan Ashworth, Christopher J. Lord, and Jorge S. Reis-Filho. “Genetic Interactions in Cancer Progression and Treatment”. In: *Cell* 145.1 (Apr. 1, 2011), pp. 30–38. ISSN: 0092-8674. DOI: 10.1016/j.cell.2011.03.020. URL: http://www.sciencedirect.com/science/article/pii/S0092867411002972 (visited on 06/02/2020).

[4] Jon-Matthew Belton et al. “Hi-C: A Comprehensive Technique to Capture the Conformation of Genomes”. In: *Methods (San Diego, Calif.)* 58.3 (Nov. 2012), pp. 268–276. ISSN: 1095-9130. DOI: 10.1016/j.ymeth.2012.05.001. pmid: 22652625.

[5] João Botelho, Filipa Grosso, and Luísa Peixe. “Antibiotic Resistance in Pseudomonas Aeruginosa – Mechanisms, Epidemiology and Evolution”. In: *Drug Resistance Updates* 44 (May 1, 2019), p. 100640. ISSN: 1368-7646. DOI: 10.1016/j.drup.2019.07.002. URL: http://www.sciencedirect.com/science/article/pii/S1368764619300238 (visited on 01/25/2021).

[6] Joseph K. Bradley et al. *Parallel Coordinate Descent for L1-Regularized Loss Minimization*. May 26, 2011. arXiv: 1105.5379 [cs, math]. URL: http://arxiv.org/abs/1105.5379 (visited on 07/14/2019).

[7] T. Carver et al. “Artemis: An Integrated Platform for Visualization and Analysis of High-Throughput Sequence-Based Experimental Data”. In: *Bioinformatics* 28.4 (Feb. 15, 2012), pp. 464–469. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/btr703. URL: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btr703 (visited on 01/07/2021).

[8] Michael Chichignoud, Johannes Lederer, and Martin J Wainwright. “A Practical Scheme and Fast Algorithm to Tune the Lasso With Optimality Guarantees”. In: (), p. 20.

[9] Michael Costanzo et al. "The Genetic Landscape of a Cell." In: *Science* (2010).

[10] Kristina Crona et al. "Inferring genetic interactions from comparative fitness data". en. In: *Elife* 6 (Dec. 2017).

[11] Richard Durstenfeld. "Algorithm 235: Random Permutation". In: *Communications of the ACM* 7.7 (July 1964), p. 420. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/364520.364540. URL: https://dl.acm.org/doi/10.1145/364520.364540 (visited on 10/28/2020).

[12] Kieran Elmes et al. "Learning Epistatic Gene Interactions from Perturbation Screens". In: *bioRxiv* (Aug. 25, 2020), p. 2020.08.24.264713. DOI: 10.1101/2020.08.24.264713. URL: https://www.biorxiv.org/content/10.1101/2020.08.24.264713v1 (visited on 08/31/2020).

[13] Ronald A Fisher and Frank Yates. *Statistical Tables: For Biological, Agricultural and Medical Research*. Oliver and Boyd, 1938.

[14] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "Regularization Paths for Generalized Linear Models via Coordinate Descent". In: *Journal of Statistical Software* 33.1 (2010). ISSN: 1548-7660. DOI: 10.18637/jss.v033.i01. pmid: 20808728. URL: http://www.jstatsoft.org/v33/i01/ (visited on 07/14/2019).

[15] Wenjiang J. Fu. "Penalized Regressions: The Bridge versus the Lasso". In: *Journal of Computational and Graphical Statistics* 7.3 (Sept. 1998), pp. 397–416. ISSN: 1061-8600, 1537-2715. DOI: 10.1080/10618600.1998.10474784. URL: http://www.tandfonline.com/doi/abs/10.1080/10618600.1998.10474784 (visited on 06/19/2020).

[16] Robert Gaynes, Jonathan R. Edwards, and National Nosocomial Infections Surveillance System. "Overview of Nosocomial Infections Caused by Gram-Negative Bacilli". In: *Clinical Infectious Diseases* 41.6 (Sept. 15, 2005), pp. 848–854. ISSN: 1058-4838. DOI: 10.1086/432803. URL: https://doi.org/10.1086/432803 (visited on 01/25/2021).

[17] Alison L Gould et al. "Microbiome interactions shape host fitness". en. In: *Proc. Natl. Acad. Sci. U. S. A.* 115.51 (Dec. 2018), E11951–E11960.

[18] *GRCh38.P13 - Genome - Assembly - NCBI*. URL: https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39 (visited on 12/01/2020).

[19] *IPCD International Pseudomonas Consortium Database*. URL: https://ipcd.ibis.ulaval.ca/ (visited on 01/07/2021).

[20] Timothy J. Kidd et al. "Pseudomonas Aeruginosa Exhibits Frequent Recombination, but Only a Limited Association between Genotype and Ecological Setting". In: *PLoS ONE* 7.9 (Sept. 6, 2012). Ed. by Sam Paul Brown, e44199. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0044199. URL: https://dx.plos.org/10.1371/journal.pone.0044199 (visited on 01/07/2021).

[21] Ben Langmead and Steven L. Salzberg. "Fast Gapped-Read Alignment with Bowtie 2". In: *Nature Methods* 9.4 (4 Apr. 2012), pp. 357–359. ISSN: 1548-7105. DOI: 10.1038/nmeth.1923. URL: https://www.nature.com/articles/nmeth.1923 (visited on 01/10/2021).

[22] D. Lemire and L. Boytsov. "Decoding Billions of Integers per Second through Vectorization". In: *Software: Practice and Experience* 45.1 (2015), pp. 1–29. ISSN: 1097-024X. DOI: 10.1002/spe.2203. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2203 (visited on 07/13/2020).

[23] Caitlin Lienkaemper et al. "The geometry of partial fitness orders and an efficient method for detecting genetic interactions". en. In: *J. Math. Biol.* 77.4 (May 2018), pp. 951–970.

[24] Jeffrey B Lyczak, Carolyn L Cannon, and Gerald B Pier. "Establishment of Pseudomonas Aeruginosa Infection: Lessons from a Versatile Opportunist1*Address for Correspondence: Channing Laboratory, 181 Longwood Avenue, Boston, MA 02115, USA". In: *Microbes and Infection* 2.9 (July 1, 2000), pp. 1051–1060. ISSN: 1286-4579. DOI: 10.1016/S1286-4579(00)01259-4. URL: http://www.sciencedirect.com/science/article/pii/S1286457900012594 (visited on 01/25/2021).

[25] Antonio Mallia, Michał Siedlaczek, and Torsten Suel. "An Experimental Study of Index Compression and DAAT Query Processing Methods". In: *Advances in Information Retrieval*. Ed. by Leif Azzopardi et al. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 353–368. ISBN: 978-3-030-15712-8.

[26] Ramachandra Nanjegowda et al. "Scalability Evaluation of Barrier Algorithms for OpenMP". In: *Evolving OpenMP in an Age of Extreme Parallelism*. Ed. by Matthias S. Müller, Bronis R. de Supinski, and Barbara M. Chapman. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 42–52. ISBN: 978-3-642-02303-3. DOI: 10.1007/978-3-642-02303-3_4.

[27] Preeti Pachori, Ragini Gothalwal, and Puneet Gandhi. "Emergence of Antibiotic Resistance Pseudomonas Aeruginosa in Intensive Care Unit; a Critical Review". In: *Genes & Diseases* 6.2 (June 1, 2019), pp. 109–119. ISSN: 2352-3042. DOI: 10.1016/j.gendis.2019.04.001. URL: http://www.sciencedirect.com/science/article/pii/S2352304219300170 (visited on 01/25/2021).

[28] Zheng Pang et al. "Antibiotic Resistance in Pseudomonas Aeruginosa: Mechanisms and Alternative Therapeutic Strategies". In: *Biotechnology Advances* 37.1 (Jan. 1, 2019), pp. 177–192. ISSN: 0734-9750. DOI: 10.1016/j.biotechadv.2018.11.013. URL: http://www.sciencedirect.com/science/article/pii/S0734975018301976 (visited on 01/25/2021).

[29] powturbo. *Powturbo/TurboPFor-Integer-Compression*. July 9, 2020. URL: https://github.com/powturbo/TurboPFor-Integer-Compression (visited on 07/09/2020).

[30] Olga Puchta et al. "Network of Epistatic Interactions within a Yeast snoRNA". In: *Science* 352.6287 (May 13, 2016), pp. 840–844. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aaf0965. pmid: 27080103. URL: https://science.sciencemag.org/content/352/6287/840 (visited on 08/13/2020).

[31] Pauli Rämö et al. "Simultaneous Analysis of Large-Scale RNAi Screens for Pathogen Entry". In: *BMC Genomics* 15.1 (Dec. 22, 2014), p. 1162. ISSN: 1471-2164. DOI: 10.1186/1471-2164-15-1162. URL: https://doi.org/10.1186/1471-2164-15-1162 (visited on 11/14/2019).

[32] Kay A. Ramsay et al. "Genomic and Phenotypic Comparison of Environmental and Patient-Derived Isolates of Pseudomonas Aeruginosa Suggest That Antimicrobial Resistance Is Rare within the Environment". In: *Journal of Medical Microbiology* 68.11 (Nov. 1, 2019), pp. 1591–1595. ISSN: 0022-2615, 1473-5644. DOI: 10.1099/jmm.0.001085. URL: https://www.microbiologyresearch.org/content/journal/jmm/10.1099/jmm.0.001085 (visited on 01/07/2021).

[33] Attika Rehman, Wayne M. Patrick, and Iain L. Lamont. "Mechanisms of Ciprofloxacin Resistance in Pseudomonas Aeruginosa: New Approaches to an Old Problem". In: *Journal of Medical Microbiology*, 68.1 (2019), pp. 1–10. ISSN: 0022-2615, DOI: 10.1099/jmm.0.000873. URL: https://www.microbiologyresearch.org/content/journal/jmm/10.1099/jmm.0.000873 (visited on 01/07/2021).

[34] Tracey Remmington, Nikki Jahnke, and Christian Harkensee. "Oral Anti-Pseudomonal Antibiotics for Cystic Fibrosis". In: *Cochrane Database of Systematic Reviews* (July 14, 2016). Ed. by Cochrane Cystic Fibrosis and Genetic Disorders Group. ISSN: 14651858. DOI: 10.1002/14651858.CD005405.pub4. URL: http://doi.wiley.com/10.1002/14651858.CD005405.pub4 (visited on 01/25/2021).

[35] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. "Fast Integer Compression Using SIMD Instructions". In: *Proceedings of the Sixth International Workshop on Data Management on New Hardware - DaMoN '10*. The Sixth International Workshop. Indianapolis, Indiana: ACM Press, 2010, pp. 34–40. ISBN: 978-1-4503-0189-3. DOI: 10.1145/1869389.1869394. URL: http://portal.acm.org/citation.cfm?doid=1869389.1869394 (visited on 07/13/2020).

[36] *STRING: Functional Protein Association Networks*. URL: https://string-db.org/cgi/about.pl (visited on 07/22/2020).

[37] Robert Tibshirani. "Regression Shrinkage and Selection Via the Lasso". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (1996), pp. 267–288. ISSN: 2517-6161. DOI: 10.1111/j.2517-6161.1996.tb02080.x. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1996.tb02080.x (visited on 08/29/2020).

[38] Andrew Trotman and Jimmy Lin. "In Vacuo and In Situ Evaluation of SIMD Codecs". In: *Proceedings of the 21st Australasian Document Computing Symposium*. ADCS '16. Caulfield, VIC, Australia: Association for Computing Machinery, Dec. 5, 2016, pp. 1–8. ISBN: 978-1-4503-4865-2. DOI: 10.1145/3015022.3015023. URL: https://doi.org/10.1145/3015022.3015023 (visited on 07/08/2020).

[39] Sara Van de Geer. "The Deterministic Lasso". In: 2007.

[40] Tong Tong Wu and Kenneth Lange. "Coordinate Descent Algorithms for Lasso Penalized Regression". In: *The Annals of Applied Statistics* 2.1 (Mar. 2008), pp. 224–244. ISSN: 1932-6157. DOI: 10.1214/07-AOAS147. arXiv: 0803.3876. URL: http://arxiv.org/abs/0803.3876 (visited on 07/14/2019).

Appendix A. Cyclic Linear Regression

As noted in Section 2.1, where y_i is the ith element of $\mathbf{Y}$, β_j is the jth element of β, and x_{ij} is the entry in the matrix $\mathbf{X}_2$ at column j of row i, the error is the following.

$$\sum_{i=1}^{n} |y_i - \sum_{j=1}^{p'} x_{ij} \cdot \beta_j| \tag{3}$$

Note that we assume that the fitness vector $\mathbf{Y}$ is already centred around 0, and omit the offset u present in [40]. In the context of pairwise genetic interactions we would rather have a smaller number of definitely relevant effects, than a large number of marginally relevant ones. To this end, we add the lasso penalty to the error in Eq. (1). This penalises large beta values according to a parameter λ, and results in a smaller set of, typically larger, non-zero beta values [37]. With this added penalty we minimise the value:

$$\sum_{i=1}^{n} |y_i - \sum_{j=1}^{p'} x_{ij} \cdot \beta_j| + \lambda \sum_{j=1}^{p'} |\beta_j| \tag{4}$$

We do this by minimising the component of this error that each β_j is able to account for. For a particular β_k, the component of this error that is affected by changing β_k is:

$$f(\beta_k) = \sum_{i=1}^{n} |x_{ik} \cdot (y_i - \sum_{j=1}^{p'} x_{ij} \cdot \beta_j)| + \lambda |\beta_k| \tag{5}$$

This error comes from the non-zero entries in column k of $\mathbf{X}$. Since in our case all entries are either 1 or 0, this is simply the sum of errors of rows where column k has an entry, with a penalty imposed for large beta values.

To minimise this component $f(\beta_k)$ alone, we define r_i and S_k:

$$r_i = \sum_{j=1}^{p'} x_{ij} \cdot \beta_j$$

$$S_k = \sum_{i=1}^{n} x_{ik}$$

The error affected by a single beta value (Eq. (5)) can then be minimised by updating β_k with Eq. (2), repeated below:

$$\Delta\beta_k = \begin{cases} max(0, \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} - \lambda) & \text{for } \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} > 0 \\ min(0, \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} + \lambda) & \text{for } \beta_k + \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k} < 0 \end{cases} \tag{6}$$

This is equivalent to the solution from [40], as we will now show. Their solution is defined separately for positive and negative β_k:

$$\beta_{k-} = min\{0, \beta_k - \frac{\frac{\delta}{\delta\beta_k} g(\theta) - \lambda}{\sum_{i=1}^{n} x_{ik}^2}\}$$

$$\beta_{k+} = max\{0, \beta_k - \frac{\frac{\delta}{\delta\beta_k} g(\theta) + \lambda}{\sum_{i=1}^{n} x_{ik}^2}\}$$

$$\text{Where } \frac{\delta}{\delta\beta_k}g(\theta) = -\sum_{i=1}^{n} q_i x_{ik},$$

$$\text{and } q_i = y_i - u - \sum_{j=1}^{p'} x_{ij}\beta_j$$

Note that we assume the intercept term $u = 0$, because Y is centred around 0, and u can therefore be omitted. We shall first focus on proving the equivalence of our construction for β_{k-}. Since $x_{ik} \in \{0, 1\}$, it follows $x_{ik} = x_{ik}^2$, and therefore $\sum_{i=1}^{n} x_{ik}^2 = S_k$. This gives us

$$\beta_{k-} = min\{0, \beta_k - \frac{\frac{\delta}{\delta\beta_k}g(\theta) - \lambda}{S_k}\}$$

Also substituting $\frac{\delta}{\delta\beta_k}g(\theta)$, we have:

$$\begin{aligned}\beta_{k-} &= min\{0, \beta_k - \frac{-\sum_{i=1}^{n} x_{ik}(y_i - r_i) - \lambda}{S_k}\} \\ &= min\{0, \beta_k + \frac{\sum_{i=1}^{n} x_{ik}(y_i - r_i) + \lambda}{S_k}\} \\ &= min\{0, \frac{S_k\beta_k + \sum_{i=1}^{n}(y_i - r_i) + \lambda}{S_k}\}\end{aligned}$$

This is equivalent to Eq. (2) for $\beta_k < 0$. The positive solution is equivalent, substituting min for max and subtracting λ. Iteratively minimising beta values until the solution converges, we have Algorithm 1. We consider the algorithm to have converged when $\frac{e_{prev}}{e_{after}} < t$ for some threshold t, where e_{prev} is the error before the iteration and e_{after} the error after the iteration. We arbitrarily chose $t = 1.0001$ as the default in our implementation.

```
while not converged do
    foreach β_k do
        Δβ_k ← (Σ_{i=1}^{n}(x_ik(y_i − r_i))) / S_k;
        if |β_k + Δβ_k| > λ then
            β_k ← β_k + Δβ_k;
            if β_k > 0 then
                β_k ← β_k − λ;
            end
            else
                if β_k < 0 then
                    β_k ← β_k + λ;
                end
            end
        end
    end
end
```

Algorithm 1: Sequential Cyclic Algorithm

A naive implementation would read every entry of the $\mathbf{X}_2$ matrix, and every value in the vector $\mathbf{Y}$, every iteration, for every beta update. With $\mathbf{X}_2 \in \{0, 1\}^{n \times p'}$, $\beta \in \mathbb{R}^{p'}$ and $\mathbf{Y} \in \mathbb{R}^n$, this is $\Theta(np^4)$ operations per iteration. Since x_{ij} and y_i are constant, $r_i = \sum_{j=1}^{p} x_{ij}\beta_j$ only changes when β_j changes. Updating this value every iteration, rather than re-calculating it, we only need to perform n operations per β update. This brings the number of operations for an iteration down to np^2. To update β_j, we now read a single column, X_j, and the values of r_i and Y_i for each non-zero entry x_{ij}. By including $\mathbf{Y}$ in $\mathbf{R}$ such that $r_i = y_i - \sum_{j=1}^{p} x_{ij}\beta_j$ for all i, we no longer need to read $\mathbf{Y}$.

We can further reduce the work that needs to be done by storing a sparse representation of $\mathbf{X}_2$. While $\mathbf{X}$ is a sparse matrix, $\mathbf{X}_2$ is an extremely sparse matrix. In a typical simulated data set from [12] we go from, on average, 112 out of $1,000$ entries per column in the $\mathbf{X}$ matrix to 16 out of $1,000$ in $\mathbf{X}_2$. We therefore store $\mathbf{X}_2$ in the following format. Each column is a list of the positions of its non-zero entries (Fig. 7). Since these are one by definition, we don't store their value. We store the matrix column-wise to ensure the column X_j can be read quickly when updating β_j. Each column is therefore stored as a separate array of integers.

$$X: \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 3 & 1 & 0 & 2 \\ & & 2 & \\ & & & \\ & & & \end{bmatrix}$$

FIGURE 7. Simple sparse matrix representation

APPENDIX B. COMPRESSION

We can considerably reduce the size of this matrix in Fig. 7 by compression the columns. Since we have a sequence of increasing integers we can store only the offset from the previous entry, keeping the entries small. The resulting sequence of (mostly) small numbers can then be efficiently stored using integer compression methods. We describe the compression method we use in Appendix B.1 and compare it to other methods in Appendix B.2.

B.1. **Simple-8b.** Simple-8b is a non-SIMD compression scheme, with performance comparable to other state of the art methods [35, 25, 38]. While SIMD-based compression schemes can often offer significantly improved compression and decompression speed [22] [35], their implementation is architecture dependant. Simple-8b only requires a CPU be able to efficiently handle 64-bit arithmetic, and does not significantly underperform compared to state-of-the-art SIMD techniques in our testing (Appendix B.2).

Simple-8b is a 64-bit variation of the Simple-9 encoding scheme [2], and stores a sequence of integers in a single 64 bit word. The number of integers stored depends on the size of the largest one, and is indicated by a four bit 'selector'. The remaining 60 bits are divided into integers of size $1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 15, 20, 30$ or 60, with between 240 (only possible if all values are zero) and one integer stored. As seen in Fig. 8, this considerably reduces the size of $\mathbf{X}_2$ in our test data (two sets from [12], one with $p = 100$, $n = 1,000$, another with $p = 1,000$, $n = 10,000$). In the larger $p = 1,000$ set, total memory use is reduced by over 85% compared to storing integers directly. It is worth noting that this compression works well even for non-sparse sections of the matrix, since the offsets are extremely small. In an extreme case, we can store up to 240 sequential 1's in a single 64-bit word. While the earlier offset-based format (Fig. 7) relies on the sparsity of the matrix for its efficiency, compression works well regardless.

B.2. **Comparing Methods.** While Simple-8b allows our implementation to be used on any 64-bit CPU, we could also take advantage of SIMD-based methods where the such CPU instructions are available. To determine whether this is a worthwhile improvement, we compare our Simple-8b implementation to a number of state of the art alternatives.

Recent work suggests TurboPFor [29] has a particularly high compression ratio [38]. We therefore compare the best performing methods from TurboPFor against our implementation of Simple-8b (Fig. 9). The tests are performed using 32 threads across two eight-core (16 SMT threads) Intel(R) Xeon(R) Gold 6244 CPUs in a NUMA system. To compare these methods, we perform 50 regression iterations on a test data set of $p = 1,000$ genes and $n = 10,000$ siRNAs. We examine the total time taken for the process, as well as the total memory used and time for the regression function alone (excluding calculating and compressing the interaction matrix).

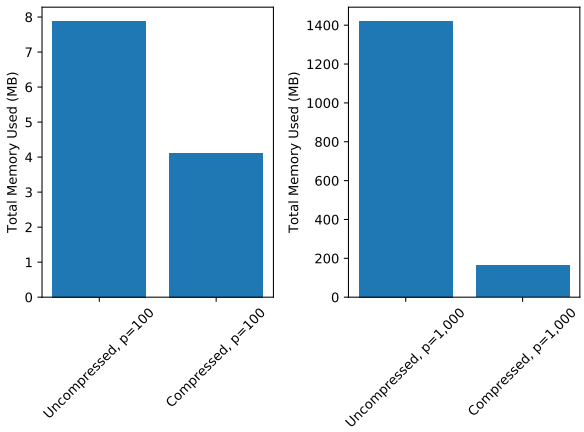
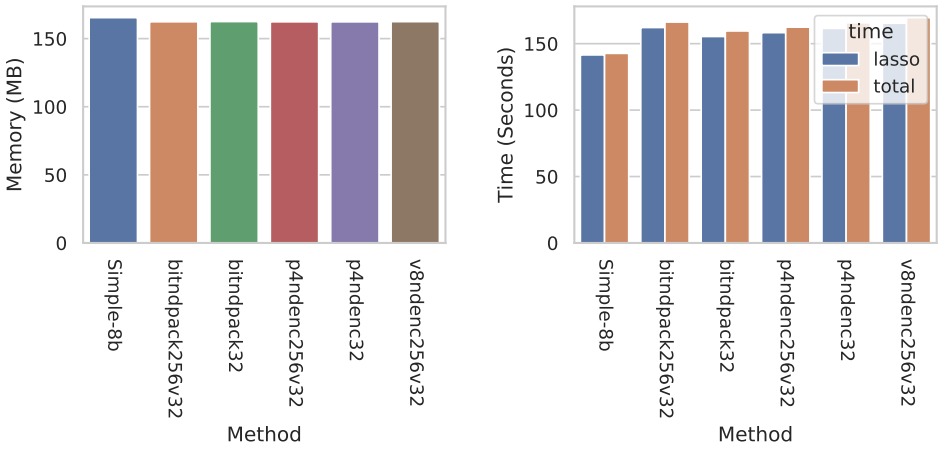


FIGURE 8. Compression effect on memory use. Note that this is the total peak memory use of the program, not solely the memory used by the matrix $\mathbf{X}_2$. In both cases $n = 10 \cdot p$.



(a) (b)

FIGURE 9. Comparison of Compression Methods. (a) Total memory used, compressing the sparse $\mathbf{X}_2$ matrix with each method. (b) Total time taken and time taken (including compressing $\mathbf{X}_2$) and time taken for lasso regression alone, using each method.

We see that both the time to produce the compressed matrix (seen in Fig. 9 as the difference between total time and lasso-only time), and the running time are comparable for all TurboPFor methods.[2] While every TurboPFor method we tested improved the compression ratio compared to Simple-8b (Fig. 9a), we consistently found that the running time was longer (Fig. 9b). It is possible that this is a result of the way the columns are being read in each method. Using TurboPFor, we compress and decompress entire columns at a time. With our Simple-8b implementation, we process each 64-bit word separately. This allows us to use the column as it is being decompressed. Avoiding re-reading the column after decompression also allows the entries to be evicted from the cache earlier.

While it is also possible to process compressed words as they are read using the tested TurboPFor methods, there does not appear to be a significant difference in compression.

[2]The compression time is not comparable for all methods. Our Simple-8b implementation compresses columns in parallel, whereas TurboPFor does not. Regression is done in parallel using all cases, using the method described in Appendix D.5.

Appendix C. Choosing Lambda

The regression parameter lambda determines how large a change to β needs to be before it will actually be made. During Lasso-regression we begin with an extremely large value of lambda, and gradually decrease this until we think smaller effects are only going to over-fit the data. For a better estimate of effect strengths, we only use lasso regression to choose the non-zero beta values, and then perform Ordinary Least Squares (OLS) regression on the non-zero entries. The β_i estimates from OLS regression are the effect strength estimates and the p-values are used to determine whether an effect is significant. It suffices then to continue decreasing lambda until an arbitrary small lambda value, relying on OLS regression to filter for significant results. As lambda decreases the number of non-zero effects increases, however, eventually becoming too many for OLS regression. Since small effects are less likely to be correct [12], we prefer to stop at a larger lambda.

We provide two options for choosing the final lambda in our package. First, we choose lambda such that the number of non-zero effects is small enough for OLS regression. Second, we implement a fast method for empirically choosing a reasonable stopping point.

In either case we begin with lambda sufficiently large that all beta values will be zero. Lambda is then gradually decreased, setting the new value at each step to $\lambda_{new} = 0.9 \cdot \lambda_{prev}$. We decrease lambda until we reach or pass the minimum value (0.05 by default). After fitting with each lambda, we optionally check one (or both) of the two stopping conditions. First, we can check whether we have reached the maximum number of non-zero beta values. Alternatively, we perform the adaptive calibration test [8], stopping if the conditions are met.

We use the adaptive calibration lambda selection method from [8] instead of the standard K-fold cross-validation because cross-validation requires fitting each lambda value K times, and this increase to the run time is unacceptable for large data. Adaptive calibration only requires a single relatively small calculation for each lambda. It aims to choose the minimum value of lambda that is sufficient to control fluctuations. Assuming $\mathbf{X}_2$ satisfies the design condition from [39], the value chosen is within a constant factor of this ideal value, and precision and recall are comparable to cross-validation [8].

In Fig. 10 we compare precision, recall, and running time when using the adaptive calibration stopping condition to running as many iterations as we can, on a simulated data set from [12]. In one case we decrease lambda until the adaptive calibration condition is met. In the other we limit the number of non-zero effects to $2,000$. In both cases, we then perform the OLS regression step and filter out results with a p-value ≥ 0.05.

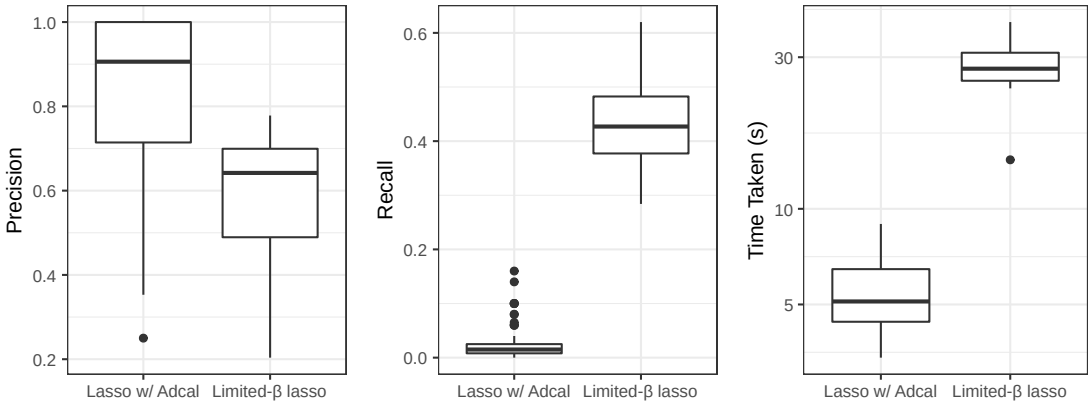


Figure 10. Adaptive calibration effect on large sets. 'Limited-β Lasso' runs until $2,000$ beta values are non-zero, continuing until the cutoff otherwise. 'Lasso w/ Adcal' halts once adaptive calibration conditions are met, continuing until the cutoff otherwise.

We see significantly higher recall when running until a very small cutoff, allowing only $2,000$ effects in total to allow OLS regression. Adaptive calibration on the other hand stops very early.

While the predictions at this point are almost entirely correct, the majority of the effects that would be found with lower lambda values are missed. Since the loss of precision at lower lambda values is mitigated effectively by OLS p-value filtering, we suggest this approach in general.

Both the adaptive calibration option and limiting non-zero betas can prevent the algorithm from finding small effects. This is in many cases beneficial, as it is the small values of lambda, and therefore the small effects, that are computationally expensive. Moreover, our previous benchmarks [12] show that small effects are also the least likely to be correctly identified. With that in mind, we do not consider ignoring these effects to be a problem.

Appendix D. Parallelisation Details

In this section we provide an overview of the challenges in parallelising Lasso-regression, and our attempts to overcome them. Appendices D.1 to D.3 describe in detail the way effect strengths (beta values) are updated, and the barriers to doing so in parallel. Appendix D.4 covers the methods we investigated, but do not use in our final implementation. The shuffled method we finally used is described in detail in Appendix D.5.

D.1. **Beta updates.** We refer to an update of a β_j corresponding to column X_j as a column update. Within a column update, there are no barriers to running in parallel (i.e. parallelising over rows). We can iterate through the elements of the column in parallel using openMP, and calculate the sums with a reduction. The contents of a single column are stored sequentially in memory, which limits the effectiveness of such an approach. The contents of the columns are only read, and not written, in this process, so there is no overhead in maintaining cache coherency. Once a single value has been read on one core, an entire cache-line will be available from its local cache, however. Since these have been read from memory already, there is no advantage to reading them into another core's cache for parallel processing. We could attempt to offset the work of each core, so that each will be working on a separate cache line within the same column of the matrix. Such an approach, however, assumes that the column contains at least $k = \frac{\text{cache line size}}{\text{entry size}} \cdot (\text{num cores})$ entries, which is unlikely. There is also considerable overhead in thread barriers [26], and the work done must be enough to justify this. To solve these problems, and avoid having threads idle when their component of the work is finished, we would need to have several times k entries. In our test set of $n = 10,000$ siRNAs, the mean number of non-zero entries in a column is only 150, or seven compressed 64-bit words. The L1 data cache of our test CPU (A Xeon Gold 6244) has a 64 byte cache line size, enough for eight entries. Even with hundreds of thousands of siRNAs, each column could only be expected to be a few cache lines. Running the iterations over columns in parallel, rather than rows, is therefore the focus of our parallelisation attempts.

D.2. **Overlap Error.** We cannot simply perform several column updates in parallel. Each column update both reads and writes r_i values for every non-zero entry in the column. If two columns are updated in parallel, and they both have non-zero entries in a common row, there is a time-of-check to time-of-use problem. An update can occur in an r_i after that value has been read by another thread. While the cached r_i themselves will never be incorrect, as the updates are atomic and always the result of real changes to a beta value, both columns will be updated based on the old value. Both columns are partially responsible for the difference between the current fit, $\sum_{i=1}^{p'} x_{ij}\beta_j$, and observed fitness, y_i, of this common row. Each of these updates will attempt to minimise this as much as possible, without taking the other update into account. This can result in overcorrecting for the error in the common row, potentially increasing the overall error. To compensate for the increased error, the next update may make an even larger change to β (Fig. 11a). In the worst case, if two or more updates repeatedly overcorrect for each other, this can prevent convergence entirely (Fig. 11b).

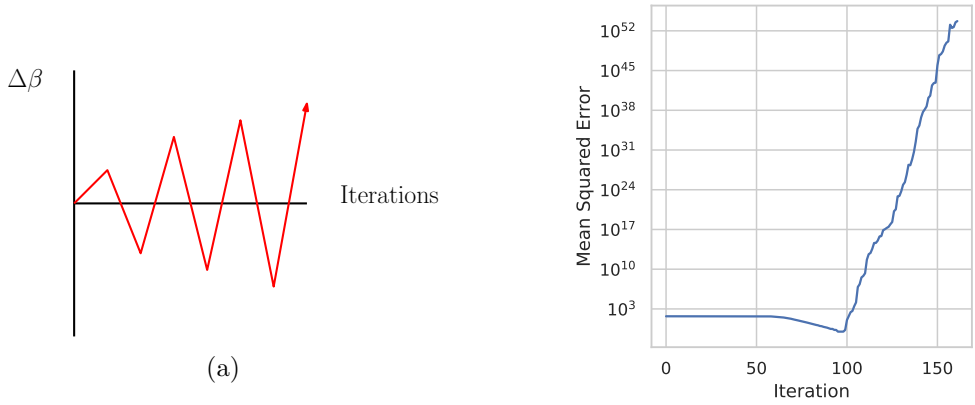


FIGURE 11. Effect of repeated overcorrection. (a) Repeated overcorrections lead to increasingly large changes to error. (b) Effect of repeated overcorrection on fit. This example is the result of fitting a 50×20 random binary $\mathbf{X}_2$ matrix, with random $\mathbf{Y}$ values between -1 and 1.

D.3. **Deriving Overcorrection Error.** Simultaneous updates may result in overcorrection, but we can analytically determine exactly how much. Using the definitions from Appendix A, ignoring the lasso penalty for the moment, we update a single β_k by $\Delta\beta_k$ as follows.

$$\Delta\beta_k = \frac{\sum_{i=1}^{n}(x_{ik}(y_i - r_i))}{S_k}$$

Let us define $\hat{\Delta}\beta_k$ to be the value that $\Delta\beta_k$ would take if, for every $j < k$, the update to β_j, had already been performed. If we perform all updates strictly sequentially, then $\Delta\beta_k = \hat{\Delta}\beta_k$. Similarly, we define $\overset{\wedge_k}{r_i}$ to be the value of r_i after all updates prior to k have been performed.

$$\overset{\wedge_k}{r_i} = \sum_{j=1}^{p'}(x_{ij}\beta_j) + \sum_{j=1}^{k-1}(x_{ij}\hat{\Delta}\beta_j)$$

Since the only difference between $\hat{\Delta}\beta_k$ and $\Delta\beta_k$ is the change in r_i caused by previous beta updates, it follows:

$$\begin{aligned}
\hat{\Delta}\beta_k &= \frac{\sum_{i=1}^{n}(x_{ik}(y_i - \overset{\wedge_k}{r_i})}{S_k} \\
&= \frac{\sum_{i=1}^{n}(x_{ik}(y_i - \sum_{j=1}^{p'}(x_{ij}\beta_j) - \sum_{j=1}^{k-1}(x_{ij}\hat{\Delta}\beta_j)))}{S_k} \\
&= \frac{\sum_{i=1}^{n} x_{ik}(y_i - r_i)}{S_k} - \frac{\sum_{i=1}^{n} x_{ik}\sum_{j=1}^{k-1} x_{ij}\hat{\Delta}\beta_j}{S_k} \\
&= \Delta\beta_k - \frac{\sum_{i=1}^{n} x_{ik}\sum_{j=1}^{k-1} x_{ij}\hat{\Delta}\beta_j}{S_k} \\
&= \Delta\beta_k - \frac{\sum_{j=1}^{k-1}\hat{\Delta}\beta_j\sum_{i=1}^{n} x_{ij}x_{ik}}{S_k}
\end{aligned}$$

Note that $\sum_{i=1}^{n} x_{ij}x_{ik}$ is constant with respect to changes in β and $\mathbf{R}$. We can compute these values once after the input has been read, and re-use them in every iteration. If we define the overlap between columns j and k of $\mathbf{X}_2$ to be $\gamma_{jk} = \sum_{i=1}^{n} x_{ij}x_{ik}$, we have the following.

$$\hat{\Delta}\beta_k = \Delta\beta_k - \frac{\sum_{j=1}^{k-1} \gamma_{jk}\hat{\Delta}\beta_j}{S_k}$$

Remark 1. $\hat{\Delta}\beta_k \neq \Delta\beta_k$ if and only if $\gamma_{jk} \neq 0$ for some $j < k$.

For $\lambda \neq 0$ we define the soft threshold function $f_\lambda(x)$.

$$f_\lambda(x) = \begin{cases} \min(0, x + \lambda) & \text{for } x < 0 \\ \max(0, x - \lambda) & \text{for } x > 0 \end{cases}$$

We find that the value of $\hat{\Delta}\beta_k$ for $\lambda \neq 0$ is the following, and use this definition in our package.

$$\begin{aligned} \hat{\Delta}\beta_{k\lambda} &= f_\lambda(\beta_k + \hat{\Delta}\beta_k) - \beta_k \\ &= f_\lambda(\beta_k + \Delta\beta_k - \frac{\sum_{j=1}^{k-1} \gamma_{jk}\hat{\Delta}\beta_j}{S_k}) - \beta_k \end{aligned}$$

We now consider a concrete example of Remark 1. Let each residual r_i be fixed (for the moment), and update first β_1, then β_2, the intended sequential update effect is then:

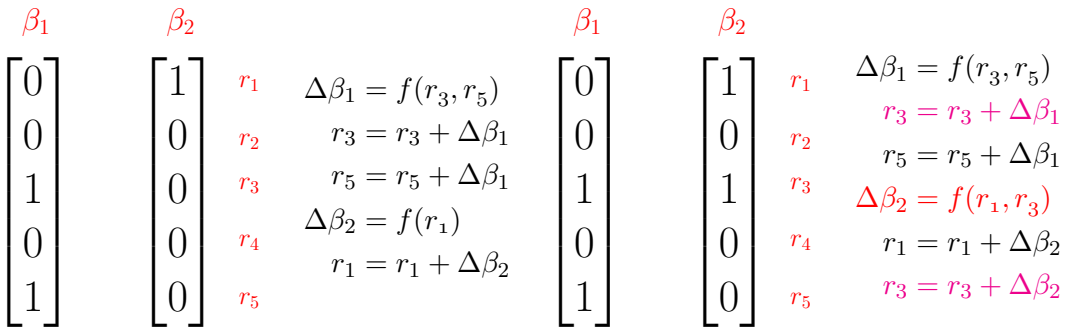
$$\begin{aligned} \Delta\beta_1 &= \frac{\sum_{i|x_{i,1}=1} y_i - r_i}{\sum_{i=1}^{n} x_{i,1}} \\ \Delta\beta_2 &= \frac{\sum_{i|x_{i,2}=1} y_i - r_i + \gamma_{12}\Delta\beta_1}{\sum_{i=1}^{n} x_{i,2}} \end{aligned}$$

When both updates are instead performed at the same time we get:

$$\begin{aligned} \Delta\beta_1 &= \frac{\sum_{i|x_{i,1}=1} y_i - r_i}{\sum_{i=1}^{n} x_{i,1}} \\ \Delta\beta_2 &= \frac{\sum_{i|x_{i,2}=1} y_i - r_i}{\sum_{i=1}^{n} x_{i,2}} \end{aligned}$$

Updating in parallel, the effect of β_2 is estimated based on the original $\mathbf{R}$, rather than those that account for the changes made to β_1. We can easily calculate the expected overcorrection in this case, $\gamma_{1,2}\Delta\beta_1$. Note that we are atomically updating the residuals r_i that are affected by both updates. If we fail to do this, overcorrection becomes difficult to predict.

For example, β_1 and β_2 correspond to the columns in Fig. 12a, and suppose these columns of $\mathbf{X}_2$ are chosen for simultaneous updates. We can calculate the changes $\Delta\beta_1$ and $\Delta\beta_2$ in parallel and update the residuals safely, because there are no shared values being updated by both threads. In Fig. 12b, we find that both the update to β_1 and the update to β_2 affect residual r_3. While atomic updates to the actual value of r_3 will guarantee that we finish with the value $r_3 + \Delta\beta_1 + \Delta\beta_2$, we have not taken the changed value of r_3 into account when calculating $\Delta\beta_2$. The correct update would have been $r_3 + \Delta\beta_1 + \Delta\beta_2 - \frac{\Delta\beta_1}{2}$



(a) Non-overlapping updates (b) Overlapping updates

FIGURE 12

Given that we can safely perform updates for columns that have no overlap, and we can explicitly compensate for the error of sets of columns, we investigate three approaches for parallelisation: compensating for the error of pre-determined sets (Appendix D.4.1), simultaneously updating non-overlapping sets (Appendix D.4.5), and randomly updating shuffled columns (Appendix D.5).

D.4. **Alternative Parallelisation Methods.** Two alternative approaches to parallelisation were considered, but not used in the final implementation. In this section we provide a detailed explanation of the barriers to parallelised lasso regression and the methods we attempted, but did not use.

D.4.1. *Explicit error compensation.* To update β_a and β_b at the same time, we need to subtract the overcorrection from one of them (arbitrarily chosen) afterwards. The final value is then the same as if we had updated the values sequentially. Subtracting the overcorrection from $\Delta\beta_a$ we update β_a as follows:

$$\beta_a = \beta_a + f_\lambda(\beta_b + \Delta\beta_k - \frac{\gamma_{ab}\hat{\Delta}\beta_b}{S_k}) - \beta_a$$

Similarly, we can update any subset of the beta values $\{\beta_1, \ldots, \beta_{p'}\}$ simultaneously, as long as we account for overcorrection in each update. For every β_k in the subset $\{\beta_1, \ldots, \beta_l\}$, we make the following correction:

$$\beta_k = \beta_k + f_\lambda(\beta_k + \Delta\beta_k - \frac{\sum_{j=1}^{k-1}\gamma_{jk}\hat{\Delta}\beta_j}{S_k}) - \beta_k$$

This method has been implemented in the `error_comp` branch of our repository[3], and is marginally faster than sequential updating with the right parameters. It does not scale well enough that we can recommend its use, however.

To understand the scalability of this approach, we begin by noting that the time taken to correct C simultaneous beta updates is on the order of C^2. This is because each update $0 \leq i \leq C$ requires reading the $i-1$ previous corrected values, resulting in $\frac{(i-1)(i-2)}{2}$ reads. If we were to attempt to update the entire interaction matrix in parallel, followed by correcting errors, there would be on the order of p'^2 corrections. Since a sequential iteration requires only $p'\mu$ steps, where μ is the mean number of non-zero entries per column, we would spend more time on corrections than updates. Even if corrections were run in parallel, this would be slower than sequential updates.

We do not, however, have to update the entire matrix at once. If we restrict ourselves to updating small sets, where 'small' is some function of the number of threads we are able to effectively use, the problem becomes tractable. Performing C parallel updates, where C is some constant multiple of the number of available threads, we have in total $\frac{p'}{C}$ sets to update, resulting

[3] https://github.com/bioDS/Pint/tree/error_comp

in $\frac{p'\mu}{C} + p'C$ update operations on the main thread.[4] For $\frac{C^2}{C-1} < \mu$ this is an improvement, even with single-threaded correction.[5]

Remark 2. Ahmdahl's Law implies a best-case improvement of:

$$\frac{p'\mu}{p'(\frac{\mu}{C} + C)}$$

D.4.2. *Parallel correction does not scale.* To improve upon Remark 2, we have to correct beta updates in parallel. This requires calculating $\hat{\Delta}\beta_k$ without using any previously corrected values, $\hat{\Delta}\beta_j$ for $j < k$. Substituting $\hat{\Delta}\beta_j$, a corrected update $\hat{\Delta}\beta_k$ then becomes the following:

$$\hat{\Delta}\beta_k = \Delta\beta_k - \frac{\sum_{j=1}^{k-1} \gamma_{jk}\Delta\beta_j - \frac{\sum_{j'=1}^{j-1} \gamma_{j'j}\Delta\beta_{j'} - \Gamma}{S_j}}{S_k}$$

Where Γ is a nested sum containing a further $(k-4)!$ additions. Even for very small sets, computing this is not feasible. We must therefore fix overcorrection sequentially, and accept the limit in Remark 2.

D.4.3. *OpenMP barrier overhead requires large C.* To achieve the improvement in Remark 2, we would need a set of eight columns, updated on eight CPUs, to finish eight times faster. This is in practice not the case. We demonstrate this by running a single iteration on the same test set with varying parallel update set sizes C. Using sets of size m times the number of available cores we gradually increase m and measure the time taken to perform all p' column updates, without compensating for overcorrection. As we can see in Fig. 13, we require blocks of 64 to 128 times as many columns as there are cores to achieve speed-ups close to the theoretical limit, and at least eight times as many for any significant improvement.

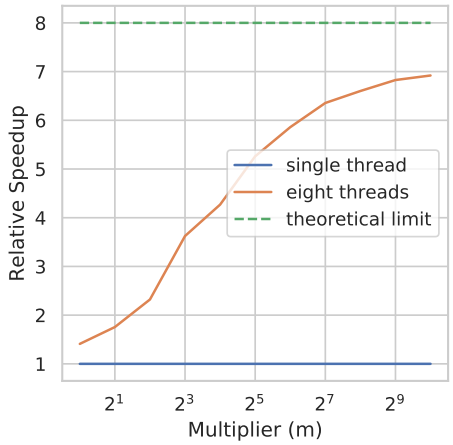


FIGURE 13. Set size multiplier effect on parallelisation speed-up. Sets contain (number of cores) $\times$ C columns.

We find that for small block sizes, the majority of the time all threads are handling an openMP barrier at the end of a parallelised loop.[6] There are two likely causes. First, when running a small number of column updates in parallel, some of these will often finish before others. If there are no further columns in the set, the thread will then have to wait for the set to finish updating. Secondly, updating columns with entries in nearby rows will mean accessing nearby memory locations to update the residuals. Since these are both being read and written

[4]Assuming that the column updates are done in parallel, and the overcorrection adjustments are done on the main thread.

[5]Note that $\frac{C^2}{C-1} \approx C$ for sufficiently large C

[6]Note that it is not the case that one thread is still updating while the others are waiting.

to, maintaining cache coherency across CPU cores is likely to affect performance. The barrier at the end of each parallel section may also take a significant amount of CPU time, as it requires some communication between threads.

Memory access is also less efficient with multiple threads. When a single thread performs a sequence of column updates, these columns are stored in memory sequentially, and each column may be as little as a single word. The core will read in an entire cache-line at once, and this may contain part of the following column, if not the following several. In this case, additional updates for these columns may be performed without any extra memory reads. If, on the other hand, we have eight columns shared between eight threads, they will each still need to read in these values. In our case the eight cores have a shared L3 cache, individual L1 and L2 caches, and 64 byte cache lines. Since one Simple-8b word is 64 bits, we have eight such words in a single cache line. In the worst case, this is eight separate columns. Supposing this is the case, a single read from memory will bring all eight columns into the shared L3 cache. It will then take eight separate reads into various L1 caches, for eight separate cores to perform updates for these columns, whereas a single core would require only one read from L3. With larger data sets, and hence larger columns, this becomes less of a problem. The fact remains though, that cache use is significantly better when each core updates several sequential columns.

All of these issues are mitigated by increasing the size of sets, and updating several sequential columns on each thread. Fig. 13 suggests sets should be at least sixteen times the number of available threads in size. This increases the time required to compensate for overcorrection.

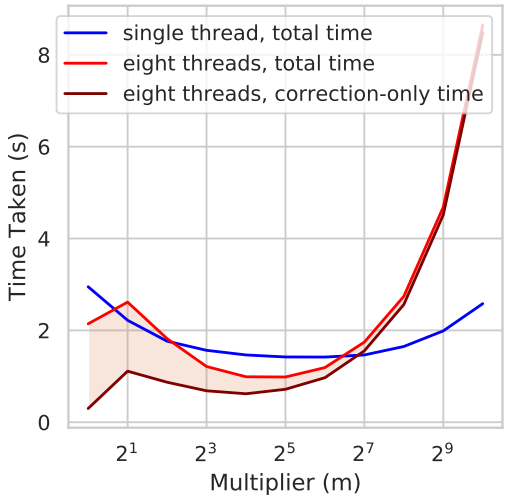


FIGURE 14. Total time taken (in seconds) for various block size multipliers. Time spent in parallel section is highlighted. Single-threaded time is included for comparison.

D.4.4. *There is no effective value of C.* The results in Appendix D.4.2 require us to perform error correction sequentially, i.e. on a single core. As we saw in Remark 2, for sufficiently small sets this is not a serious limitation. However Appendix D.4.3 also suggests that we may need large sets.

We first note that we can't significantly reduce the time taken in the error correction step. Updating $\mathbf{R}$ and calculating corrections take approximately half the time each. While updating $\mathbf{R}$ can be parallelised, there is so little work here that the thread barriers again result in worse performance. In the overlap matrix, containing the overlap between columns in the set (γ_{ij} for columns i, j), around 50% of the entries are non-zero. Removing these or using a sparse compressed matrix to store overlap is therefore unlikely to be a significant improvement.

Fig. 14 shows the amount of time spent in error correction increases quadratically with the block size. At a multiplier of 256 this overtakes the entire update time on a single core, and we

see that the multi-threaded implementation becomes slower than the single-threaded. The best multi-threaded result we see is at a multiplier of thirty-two, where the parallel version achieves a 44% improvement in iterations per second. Even with improvements to the error correction routine, it is unlikely that this approach to parallelisation can achieve more than double the performance of the sequential version. We therefore arrive at the conclusion:

Remark 3. Parallel updates of sets, followed by error correction on those sets, is not a feasible approach for parallelisation.

D.4.5. *Limiting overlap.* While we cannot compensate for overcorrection after the fact, Fig. 13 nonetheless suggests that there is some hope for performing a block of updates in parallel. Rather than allowing these blocks to be arbitrary, we now consider restricting updates to blocks of non-overlapping columns of $\mathbf{X}_2$. Again, we first divide the matrix into sets of columns for which we can perform updates simultaneously, then update these sets one at a time. Here, these sets will be collections of columns that either do not overlap, or overlap very little.

D.4.6. *Sets of no overlap.* Since the columns of $\mathbf{X}$ are relatively sparse, and the columns of $\mathbf{X}_2$ particularly so, it is plausible that we could find sets of a few non-overlapping columns purely by chance. In our test set of $1,000$ columns and $10,000$ rows the mean number of non-zero entries in a column is 0.58%. In this case, we would expect the fraction of entries in common between two randomly chosen columns to be $(0.58\%)^2$, or 0.34 entries per column. If we choose two random columns we can reasonably expect them not to overlap. Extending this we find sets of non-overlapping columns using the following method.

We start by randomly shuffling the columns. We then compare every second column with its neighbour, recording a set of two non-overlapping columns where possible (Fig. 15b). Smaller sets are then repeatedly merged to form larger sets, where no overlap exists between columns (Algorithm 2). The algorithm described in Algorithm 1 is then run, updating beta in parallel for each set of non overlapping columns.

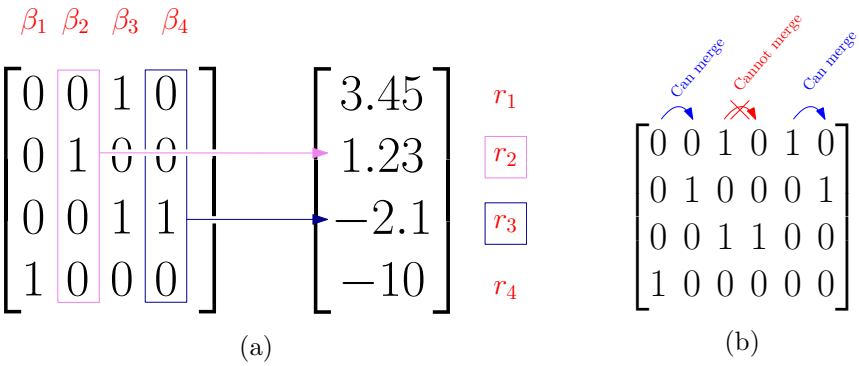
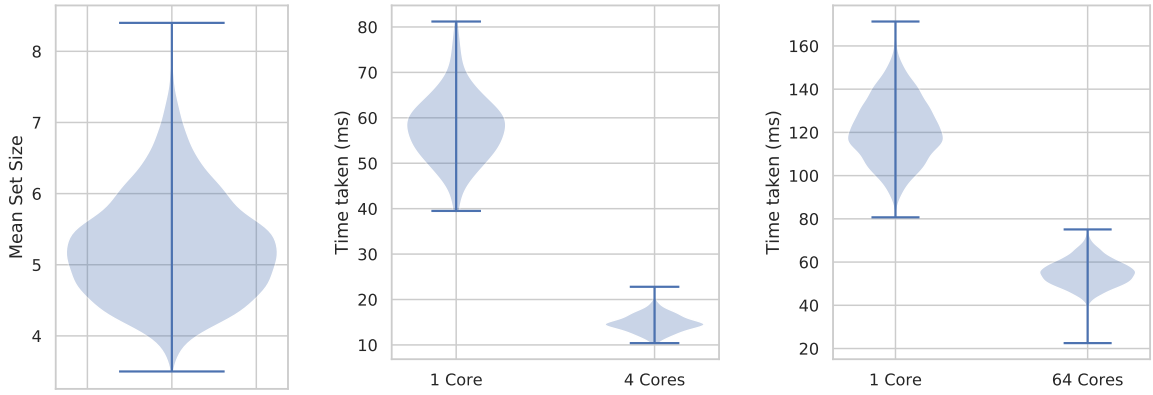


FIGURE 15. (a) Completely non-overlapping updates affect different residual values and can be done safely in parallel. (b) Attempting to merge neighbouring columns.

Input: $X \in \{0,1\}^{n \times p}$
Result: Sets of non-overlapping columns
Initialise all columns as one element sets
Merge neighbouring sets where no overlap exists
Place sets in appropriate bin (bin 1: sets of size 1, bin 2: sets of size 2, ...)
for small_bin *in* bins **do**
  for large_bin *in* bins *s.t.* large_bin > small_bin **do**
    n ← `min(sizeof(small_bin), sizeof(large_bin))`;
    for offset *in* $0, \ldots, 50$ **do**
      Attempt to merge the first n elements of small_bin with the first n elements of large_bin, starting at offset.
    end
  end
  Attempt to merge the first half of small_bin with the second half.
end

Algorithm 2: Mergesets Algorithm

D.4.7. *Performance of Overlap Method.* We compare run time on one and four cores, using simulated sets of 1000 siRNAs and 100 genes (i.e. pairwise interactions on a 1000×100 matrix) in Fig. 16. Once sets of non-overlapping columns have been found, updating non-overlapping sets in parallel improves run time compared to the single-threaded version (Fig. 16b).

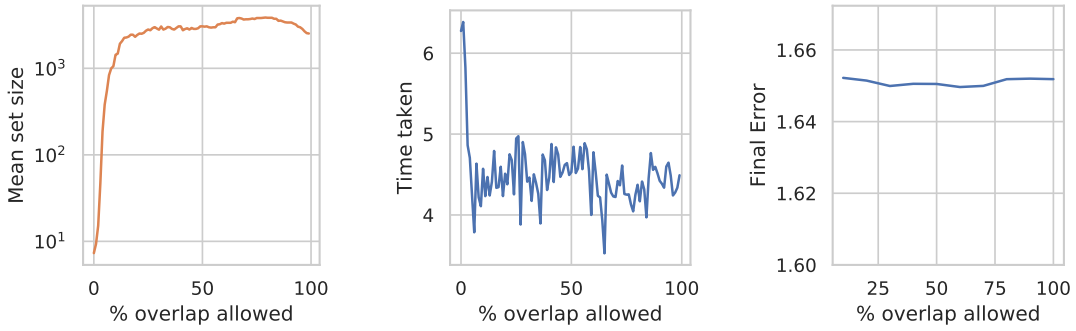


(a) (b) (c)

FIGURE 16. (a) Mean non-overlapping set sizes. (b) Run time after finding mergesets on four vs one core. (c) Run time after finding mergesets on 64 vs one core. Note that the 64 core system has considerably slower memory than the four core system.

As the size of input data increases, however, finding sets with absolutely no overlap becomes more difficult. Since, as explained in Appendix D.4.1, efficiently using more threads requires larger set sizes than those in Fig. 16a, we aim to improve on these. Running more comparisons with our current implementation is not feasible, finding sets already takes as long as the regression step. Instead, we relax the criteria from no overlap to very little overlap.

D.4.8. *Partial Overlap.* We can find much larger sets of simultaneously updateable columns if we allow a small amount of overlap between these columns. While this does allow some error to be introduced in the calculation of beta updates, we expect that by limiting the overlap this error will remain small, and not result in the drastic overcorrection seen in Fig. 11.



(a) (b) (c)

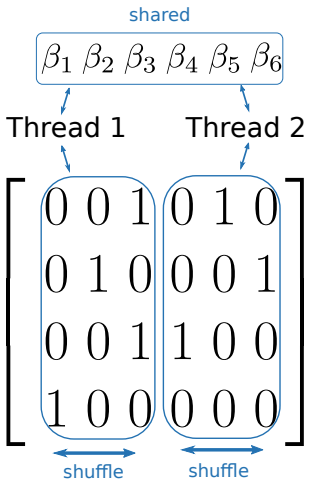
FIGURE 17. (a) Mean set size, (b) run time, and (c) final mean squared error on 64 cores as allowed overlap increases from 1% to 100%.

In small test sets ($n = 1000,\ p = 100$), increasing the available overlap significantly increases the found set size (Fig. 17a), also improving the run time. Interestingly, increasing the allowed overlap to 100% does not harm the run time (Fig. 17b) or the mean squared error of the final fit (Fig. 17c). We would expect both of these to suffer when the columns significantly overlap, as either further iterations are required to correct for the introduced error, or overcorrection error is allowed to remain. It appears that even allowing 100% overlap, there is a negligible amount of overcorrection occurring.

D.5. **Random Overlap.** Despite the possibility of over-correction, simply updating columns in parallel often works in practice. In fact, as long as we do not update the same columns at the same time every iteration, over-correction does not occur. To avoid this we shuffle the columns every iteration (see Appendix D.4 for a discussion of alternative methods). This was shown to work by Bradley et al. [6] for their lasso implementation, with up to $\frac{p}{\rho} + 1$ parallel updates, where ρ is the spectral radius of $\mathbf{X}^T\mathbf{X}$, so long as the columns being updated simultaneously were chosen at random. In our case, using the matrix of pairwise interactions, this allows $\frac{p(p+1)}{\rho} + 1$ parallel updates, where ρ is the magnitude of the largest eigenvalue of $\mathbf{X}_2^T\mathbf{X}_2$. In our smallest test case ($n = 1,000,\ p = 100$), this would allow 222 simultaneous updates. This number increases with larger input data. Combining this with the lambda sequence from Section 2.2, we have our final lasso algorithm, Algorithm 3 (see Appendix A for the full details of the non-parallel algorithm). In this example we have two threads, each working on three columns. Both threads shuffle their columns, then iteratively calculate changes and update each columns β value. Note that this calculation requires reading all β values, including those being updated by the other thread.

Input: $X \in \{0,1\}^{n \times p}$, $Y \in \mathbb{R}^n$, error_cutoff
Result: beta values
for lambda *in lambda sequence* **do**
  while $\frac{old\ error}{new\ error} >$ error_cutoff **do**
    shuffle columns
    for column *in* columns **do**
      $\beta_{\text{column}} \leftarrow \beta_{\text{column}} + \Delta\beta_k$
    end
    error $\leftarrow \sum_{i=1}^{n} y_i - \sum_{j=1}^{p'} x_{ij}\beta_j$
  end
  if *adaptive calibration conditions met OR maximum non-zero betas found* **then**
    Stop after current lambda
  end
  else
    $\lambda \leftarrow \lambda \times 0.9$
  end
end

(a)



(b)

Algorithm 3. Shuffled Lasso Algorithm (a). Parallel implementation (b).

D.5.1. *Shuffling Method.* For lasso regression on k cores, our aim in shuffling the columns is to ensure that columns are not updated at the same time in many iterations. To improve performance when p is extremely large, we shuffle the columns in several simultaneous batches, one for each thread.

We therefore implemented a parallel variation of the Fisher and Yates [13] algorithm, as described in [11]. We begin by dividing the n columns into $\frac{n}{k}$ chunks, where k is the number of available threads. When iterating through columns in Algorithm 3, these are exactly the chunks that each thread will process. Moving a column from one chunk into another effectively moves it from one thread's workload into another's. This is unnecessary and potentially harms cache performance, so we only shuffle within a chunk. As long as the total number of columns is significantly larger than the number of available threads,[7] which column in a chunk a thread is working on is a random sample from a large list, and the combinations are unlikely to often repeat themselves. This is sufficiently random to avoid the overcorrection effect observed in Fig. 11.

D.5.2. *Final parallel vs sequential performance.* Running this shuffled version (Algorithm 3) on a 16-core NUMA system (two 8-core CPUs with 2-way SMT), we see a reasonable speedup using up to eight cores, with continuing improvements up to 15 threads on the same cores (Fig. 19). Surprisingly, we see a slight drop in performance adding the 16th thread, which is the final available thread on the first NUMA node. This could be because the columns don't divide as evenly among 16 threads as they do among 15, or simply that there is some noise in the time taken on a busy system. Above 16 threads we don't see any further improvement, the performance actually worsens. Given that this is the first core of the second CPU, this appears to be the result of slower memory access on the second node, and the overhead of synchronising beta updates across nodes.[8]

[7] And we can assume this true in genome-scale data, with anywhere from thousands to billions of columns

[8] Note that steps have been taken to avoid sharing cache lines between CPUs, and each component of the compressed matrix is only accessed by the thread that allocated it.

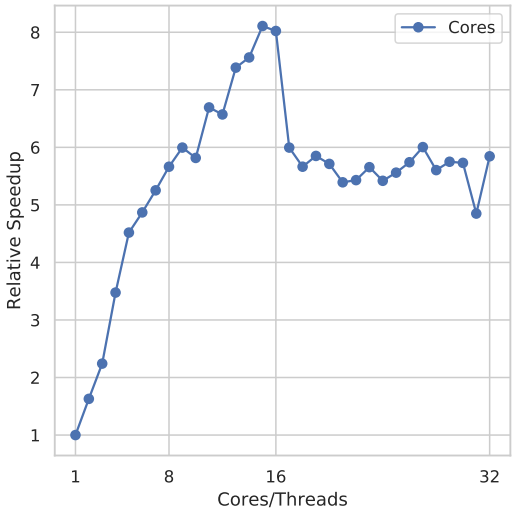


FIGURE 19. Relative speedup as the number of cores used increases, running on a dual 8 core/16 thread NUMA system. Cores 1-8 are separate cores on node 1, 8-16 are SMT threads on the same cores. Cores 17-24 are separate cores on the second NUMA node, and 15-32 are SMT threads on those cores.