The Journal of Supercomputing (2022) 78:15338–15357
https://doi.org/10.1007/s11227-022-04477-5

Fiuncho: a program for any-order epistasis detection in CPU clusters

Christian Ponte-Fernández[1] · Jorge González-Domínguez[1] · María J. Martín[1]

Accepted: 9 March 2022 / Published online: 16 April 2022

Abstract
Epistasis can be defined as the statistical interaction of genes during the expression of a phenotype. It is believed that it plays a fundamental role in gene expression, as individual genetic variants have reported a very small increase in disease risk in previous Genome-Wide Association Studies. The most successful approach to epistasis detection is the exhaustive method, although its exponential time complexity requires a highly parallel implementation in order to be used. This work presents Fiuncho, a program that exploits all levels of parallelism present in *x86_64* CPU clusters in order to mitigate the complexity of this approach. It supports epistasis interactions of any order, and when compared with other exhaustive methods, it is on average 358, 7 and 3 times faster than MDR, MPI3SNP and BitEpi, respectively.

Keywords GWAS · Epistasis · Any-order · MPI · SIMD · Multithreading

1 Introduction

With the proliferation of next-generation sequencing technologies, the cost of sequencing genomes has been reduced, and Genome-Wide Association Studies (GWAS) have become more popular. GWAS are observational studies that attempt to decipher the relationship between a particular trait or phenotype and a group of genetic variants from several individuals. Much of the early work in GWAS considered genetic variants in isolation, and the results of those studies were unsatisfactory for the task at hand. The studies commonly reported associations with variants of unknown significance that increased disease risk at very low levels, and thus their usefulness in clinical applications was limited [1]. One hypothesis that explains this outcome is a phenomenon called epistasis: the statistical interaction of genes among themselves, or with the environment, during the expression of a phenotype

✉ Christian Ponte-Fernández
christian.ponte@udc.es

[1] CITIC, Computer Architecture Group, Universidade da Coruña, Facultad de Informática, Campus de Elviña, A Coruña 15071, Spain

so that individual variants by themselves display little to no association with said phenotype. Nevertheless, looking for epistatic interactions instead of individually associated genetic markers is a much more complex task, and it is still an actively researched field.

A multitude of methods for detecting epistasis have been proposed in the literature. In essence, these methods seek to identify the combination(s) of variants that best explain the phenotype outcome observed in the data. This is a computationally intensive problem with a complexity that scales exponentially with the number of variants in combination considered (also known as the epistasis order) and the number of variants included in the input data. As a consequence of that, the methods developed followed two different approaches:

- Exhaustive methods: all genetic variant combinations from the input data (up to a certain size or interaction order) are tested for epistasis.
- Non-exhaustive methods: a fraction of the genetic variant combinations are tested, following a particular heuristic that reduces the search space. Non-exhaustive methods reduce the computational complexity of exhaustive ones. Consequently, they allow for larger GWAS analysis at the cost of the possibility of not finding the target variant combination.

Prior to this work, the performance of exhaustive and non-exhaustive methods has been studied thoroughly in [2]. The paper concluded that exhaustive methods are the only ones capable of identifying epistasis interactions in the absence of marginal effects. Marginal effects refer to the association effect that subgroups of the complete epistasis interaction display with the trait under study. If associated variants do not display marginal effects, non-exhaustive methods are ineffective and the only known alternative is to exhaustively search the combination space. In spite of that, due to implementation constraints, the majority of the proposed exhaustive methods limit the size of the epistasis interactions.

This work presents Fiuncho, an exhaustive epistasis detection tool that supports interactions of any given order, and exploits all levels of parallelism available in a homogeneous CPU cluster to accelerate the computation and make it more scalable with the size of the problem. To the best of our knowledge, the proposed implementation is faster than any other state-of-the-art CPU method.

The text is organized as follows: Sect. 2 covers related works and highlights the different trends in exhaustive epistasis detection. Section 3 describes the association algorithm used, and Sect. 4 details the parallel epistasis search implemented. Section 5 includes the evaluation of Fiuncho. And lastly, Sect. 6 presents the conclusions reached and highlights some future lines of work.

2 Related work

There is abundant literature dedicated to epistasis detection methods. This work focuses specifically on the exhaustive approach to epistasis detection because it is the only one that obtains results in the absence of marginal effects in the data.

All exhaustive methods follow the same principle: examining every combination of variants available in the data, and locating the most associated ones with the phenotype under study. As a consequence of that, all exhaustive methods present a computational complexity of $O\left(n^k \cdot O_{at}\right)$, with n being the number of variants in the search, k the order of epistasis explored and O_{at} the computational complexity of each individual association test. The expression assumes that the number of combinations without repetition, $\binom{n}{k}$, is equivalent to n^k, since the epistasis order k is smaller than $n-k$. This rigidity in the method itself has led to the development of proposals with more innovation in the different architectures used to tackle the problem than in the algorithmic approach to it.

Initially, exhaustive methods did not target a computer architecture in particular. They were written in languages such as Fortran, Java or C, and could be used in any computer. This is the case of MDR [3], one of the most recognized exhaustive epistasis detection methods in the literature. MDR was written in Java, allows for epistasis interactions of any given order and supports multithreaded execution, although the performance achieved is not ideal in modern computers. Since then, improving performance has become the focal point of the exhaustive methods.

Currently, implementations are more tailored to a particular computer architecture in order to exploit all the resources offered to speed up the search. MPI3SNP [4] and BitEpi [5] are two examples of exhaustive methods that use CPUs, or clusters of CPUs, to perform the search. MPI3SNP implements a 3-locus epistasis search using MPI, in combination with multithreading, to speed up the computation using multiple computing nodes. BitEpi, on the other hand, uses an alternative representation of the genotype information in memory, introducing a tradeoff between the complexity of the association test and the use of a more memory-intensive approach to the computation. BitEpi implements a 2, 3 and 4-locus epistasis search that also uses multithreading to speed up the search. Furthermore, for the *x86_64* CPU architecture, there are some publications that discuss AVX vector implementations of the epistasis search [6, 7].

Aside from CPUs, GPUs and FPGAs are two architectures that have gained some interest from researchers in the field. GPUs are a great fit due to the high degree of parallelism that they offer and the embarrassingly parallel nature of the epistasis search. There are a multitude of methods that fall under this category, with SNPInt-GPU [8] being one of the latest examples. Furthermore, with the introduction of tensor cores in the most recent GPU microarchitectures there has been an effort made to exploit these new instructions in the epistasis detection problem [9]. FPGAs have also been employed, with methods that support exhaustive 2 and 3-locus epistasis detection [10, 11], and more recently, epistasis interactions of any given order [12].

Lastly, some authors have embraced this diversity in architectures with methods that support heterogeneous systems in order to complete the epistasis search. This includes methods written in architecture-agnostic languages so that the same implementation can be compiled for different hardware [13], as well as methods that exploit computing systems with different architectures simultaneously, and thus taking advantage of the benefits of each separate architecture, such as CPUs with iGPUs [6], CPUs with GPUs [14] and GPUs with FPGAs [15].

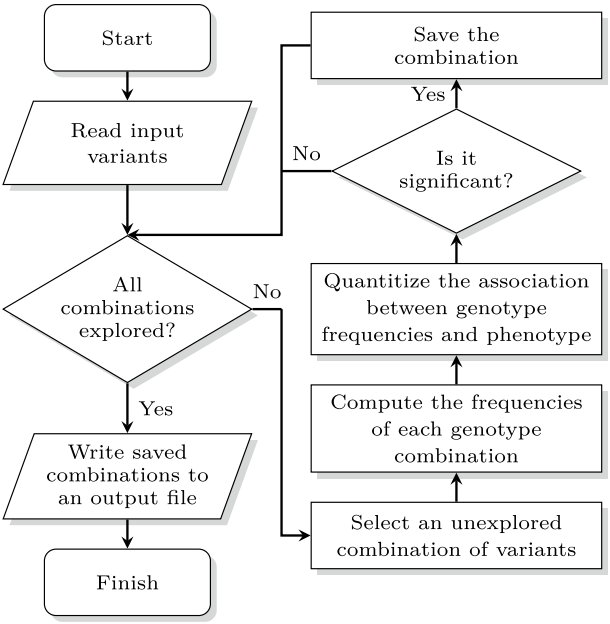


Fig. 1 Flowchart of a typical exhaustive epistasis search

This paper presents Fiuncho, a method targeting CPU architectures that combines explicit vector implementations for *x86_64* CPUs with multithread and MPI multiprocess computing to exploit all resources offered by a *x86_64* CPU cluster. Furthermore, a portable implementation using standard C++ is also included to support other CPU architectures. The exhaustive search implemented contemplates epistasis interactions of any order which, to the best of our knowledge, makes it the only CPU method, besides MDR, that does not limit the size of the interactions, although Fiuncho is significantly faster.

3 Background

All exhaustive epistasis detection methods follow the same approach: enumerate all combinations of variants for a particular order, test every combination for association with the trait under study and report the relevant combinations. Figure 1 shows a flowchart of the process. Exhaustive methods differ from one another in the association test used. Fiuncho, as MPI3SNP [4], uses a Mutual Information (MI)-based association test. As can be seen in [2], MI obtains a very good detection power.

This section briefly describes how the MI test operates, starting with the construction of genotype tables to represent the genotype information of the variants, followed by the computation of contingency tables to represent the frequency of the genotype combinations corresponding to the selected variant combination, and

concluding with the MI test to assess the association between the genotype frequencies and the phenotype.

3.1 Constructing the genotype tables

Genotype tables represent, in binary format, the genotype information of all individuals under study for a particular variant or combination of variants. They are a generalization of the binary representation introduced in BOOST [16] to simplify the computation of contingency tables for second-order epistasis interactions. The tables contain as many columns as individuals in the data, segregated into cases and controls, and as many rows as genotype values a variant or combination of variants can show. Every individual has a value of `1` in the row corresponding to its genotype and a `0` in every other row. For a human population with biallelic markers, each individual can have three different genotypes, and thus genotypes tables contain 3^k rows with k being the number of variants in combination represented.

Genotype tables are not only used to represent the information of a variant, but also to segment the individuals into different groups by their phenotype and genotype values and to represent the information of multiple variants in combination. This makes them extremely useful later when computing the frequencies of each genotype value. The construction of a genotype table for a combination of multiple variants implies:

(a) the combination of the different rows of the tables corresponding to the individual variants, and
(b) the computation of the intersection of each combination of rows (or genotype groups) via bitwise AND operations.

Figure 2 gives an example of two genotype tables for two variants a and b for 16 individuals (eight cases and controls), and the table resulting from the combination of these two variants.

3.2 Computing the contingency tables

A contingency table is a type of table that holds the frequency distribution of a number of variables, that is, the genotype and phenotype distributions for this domain of application. These frequencies can be directly obtained by counting the number of individuals in each of the phenotype and genotype groups created by the genotype table. This implies counting the number of bits set, an operation commonly known as a population count. Figure 3 shows the contingency tables of the example genotype tables included in Fig. 2.

3.3 Mutual information test

Once the contingency table is calculated, the only step left to assess the association between the genotype distribution and the phenotype affliction is computing the MI

Genotype table of variant a:

	Cases	*Controls*
a_1	00101101	10101001
a_2	10000000	01000010
a_3	01010010	00010100

Genotype table of variant b:

	Cases	*Controls*
b_1	11010000	00010101
b_2	00000110	00100010
b_3	00101001	11001000

Genotype table of variant a × variant b:

	Cases	*Controls*
$a_1 \cap b_1$	00000000	00000001
$a_1 \cap b_2$	00000100	00100000
$a_1 \cap b_3$	00101001	10001000
$a_2 \cap b_1$	10000000	00000000
$a_2 \cap b_2$	00000000	00000010
$a_2 \cap b_3$	00000000	01000000
$a_3 \cap b_1$	01010000	00010100
$a_3 \cap b_2$	00000010	00000000
$a_3 \cap b_3$	00000000	00000000

Fig. 2 Example of two genotype tables of two different variants, a and b, for eight cases and controls, and the combined genotype table of the two variants

Contingency table of variant a:

	Cases	*Controls*
a_1	4	4
a_2	1	2
a_3	3	2

Contingency table of variant b:

	Cases	*Controls*
b_1	3	3
b_2	2	2
b_3	3	3

Contingency table of variant a × variant b:

	Cases	*Controls*
$a_1 \cap b_1$	0	1
$a_1 \cap b_2$	1	1
$a_1 \cap b_3$	3	2
$a_2 \cap b_1$	1	0
$a_2 \cap b_2$	0	1
$a_2 \cap b_3$	0	1
$a_3 \cap b_1$	2	2
$a_3 \cap b_2$	1	0
$a_3 \cap b_3$	0	0

Fig. 3 Contingency table examples using the same variants as in Fig. 2

of the table. Considering two random variables X and Y representing the genotype and phenotype variability, respectively, the MI can be obtained as:

$$MI(X;Y) = H(X) + H(Y) - H(X, Y) \tag{1}$$

where $H(X)$ and $H(Y)$ are the marginal entropies of the two variables, and $H(X, Y)$ is the joint entropy. Marginal entropies of one and two variables are obtained as:

$$H(X) = -\sum_{x \in X} p(x) \log p(x) \tag{2}$$

$$H(X,Y) = -\sum_{x,y} p(x,y)\log p(x,y) \tag{3}$$

The computational complexity of constructing the genotype and contingency tables, and applying the MI test is $O(3^k \cdot m)$, with k being the number of variants in combination tested, and m the number of individuals represented in the tables.

4 Parallel method

Fiuncho implements a parallel exhaustive detection method using a static distribution strategy. Given a collection of genotype variants from two groups of samples (cases and controls), Fiuncho tests for association every combination of variants for a particular interaction order using the association test presented in Sect. 3, and reports the most associated combinations. To do this, Fiuncho combines three different levels of parallelism:

- Task parallelism: the search method is divided into independent tasks that are distributed among the processing resources available in a cluster of CPUs. MPI multiprocessing and multithreading are used for the implementation.
- Data and bit-level parallelism: each task exploits the Vector Processing Units (VPUs) by using the Single Instruction Multiple Data (SIMD) algorithm proposed in [7], including the explicit vector implementations for the *x86_64* CPU architecture of the three stages of the association test presented in Sect. 3. Furthermore, this algorithm uses 64-bit word arrays to represent each of the rows of the genotype tables, and as a consequence of that, each intersection operation (bitwise AND) works with 64 samples at once.

This section discusses the method used to exploit the task parallelism. It starts by describing the distribution strategy followed in order to divide and distribute the workload among the computational resources available, and concludes with an algorithm that implements the epistasis search using the presented strategy.

4.1 Distribution strategy

In the epistasis search, the workload is implicitly divided by the combinations themselves, and the association tests can be carried out in parallel using a pool of processing units. Each association test involves the same computations. However, many of the combinations share sub-combinations with one another, and as such, many repeating computations concerning the construction of the genotype tables can be avoided attending to how the combinations are scheduled on the different units. For instance, when searching for fourth-order epistasis, the analysis of the combinations with variants (1,2,3,4), (1,2,3,5), (1,2,3,6), etc. requires the construction of the same genotype table corresponding to the pair (1,2) and the triplet (1,2,3). Therefore,

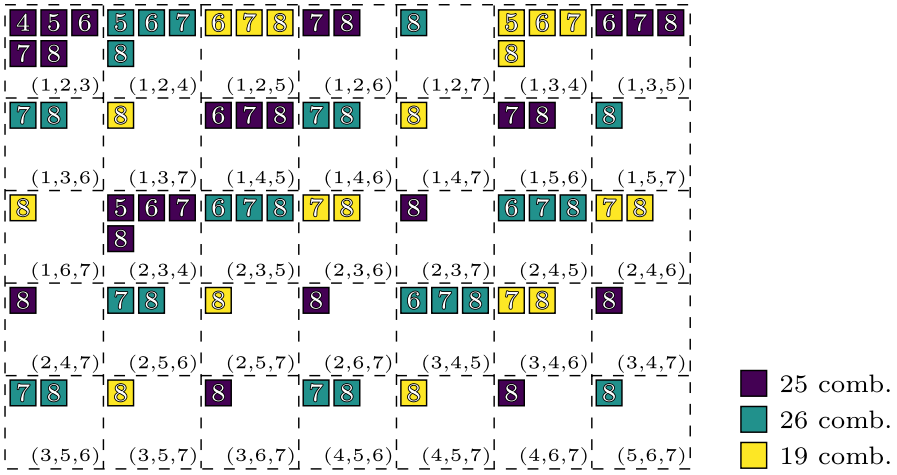


Fig. 4 Example of the distribution strategy, arranging combinations of four variants among three processing units. Each prefix of three variants (represented as large squares with dotted lines) is assigned to a unit (shown as different colors) following a round-robin distribution, and that unit tests for association every combination of four variants starting with the prefix (represented as small colored squares)

assigning all these combinations to the same unit will allow reusing the genotype tables of (1,2) and (1,2,3) for all fourth-order combinations that contain them.

Fiuncho implements a static distribution strategy in which the combinations of any given order k are distributed among homogeneous processing units using the combinations of size $k-1$, following a round-robin distribution of the combinations sorted by ascending numerical order. In other words, every combination of size $k-1$ is scheduled among units, and every unit computes all combinations of k variants starting with the given $k-1$ prefix. This strategy finds a middle ground between a good workload balance among processing units and avoiding overlaps in computations between them. By distributing the workload using the $k-1$ combination prefixes we guarantee that every combination of size k reuses the genotype tables of its prefix of size $k-1$, but it introduces an overlap between units during the calculation of the tables of the $k-1$ prefix. Nonetheless, repeating these calculations results in a negligible overhead due to the exponential growth of the combinatorial procedure, as the experimental evaluation included in Sect. 5 proves.

Figure 4 exemplifies this strategy, showing the distribution of the computations resulting from a fourth-order search ($k=4$) of eight variants using three processing units. The figure uses squares with dotted lines to represent all prefixes of $k-1=3$ variants derived from combining the eight inputs, displayed in sorted order from left

to right and top to bottom. Each prefix square includes one or more colored squared in its interior, representing a combination of four variants to be tested for association, and the colors indicate the unit which will carry out its test. Every combination under the same prefix is assigned to the same unit, guaranteeing that the genotype table of the prefix is computed only once, and every prefix is assigned to one of the three units following a round-robin distribution. At the same time, there are small overlaps between the computations corresponding to the different prefixes. For example, the prefix (1,2,3) and (1,2,4) require constructing the same genotype table for the combination (1,2), and since they were assigned to different units, the table will be constructed more than once. This strategy assigns twenty-five, twenty-six and nineteen combinations to the three processing units, respectively. Although it does not create the most balanced distribution possible, the strategy does not require synchronization or communication between units, takes the reuse of genotype tables into account and achieves very good results for a more realistic input size.

4.2 Algorithmic implementation

With the previous distribution strategy in mind, Algorithm 1 presents the pseudocode for the parallel epistasis detection method. It follows the Single Program Multiple Data (SPMD) paradigm in which all computing units execute the same function, while each unit analyzes a different set of variant combinations. The implementation combines MPI multiprocessing with multithreading to efficiently exploit the computational capabilities of CPU clusters. Every MPI process reads the input variants and stores each one in a genotype table, maintaining the individual variant information replicated in each process. After that, each MPI process spawns a number of threads that execute the function presented over a different set of variant combinations. The input data is provided to the different threads through shared memory, making an efficient use of the memory inside each node. This procedure allows the parallel strategy to be abstracted from the topology of the cluster, so that the workload is assigned to each core partaking in the computation regardless of its location.

Algorithm 1 PARALLEL_SEARCH: Tests a subset of variant combinations for association with the trait under study.

Input: A – Array of genotype tables representing n input variants
Input: L – List of variant combinations to analyse
Input: B – Size of the block of operations
Output: List of the S highest ranking k-combinations

```
function SORTED_INSERT(list, ct, mi)
    if size of list < S or mi > list[0] then
        Find first i so that list[i] > mi
        Insert {ct, mi} before i
        if size of list > S then
            Remove list[0]
        end if
    end if
end function

sorted_list ← Empty list
ct ← Array of B contingency tables
b ← 0
for {i_1, ..., i_{k-1}} in L do
    gt ← A[i_1]
    for j ← 2 to k − 1 do
        gt ← COMBINE(gt, A[i_j])
    end for
    for j ← i_{k-1} + 1 to n do
        if b = B then
            for l ← 0 to B do
                mi ← MUTUAL_INFORMATION(ct[l])
                SORTED_INSERT(sorted_list, ct[l], mi)
            end for
            b ← 0
        end if
        ct[b] ← COMBINE_AND_POPCOUNT(gt, A[j])
        b ← b + 1
    end for
end for
for l ← 0 to b do
    mi ← MUTUAL_INFORMATION(ct[l])
    SORTED_INSERT(sorted_list, ct, mi)
end for
return sorted_list
```

The input arguments to the function are the array A of n genotype tables representing the individual variants, the list of variant combinations L to analyze and the size B of the blocks in which the integer and floating-point vector operations will be segmented. The list of combinations L is provided as an iterator that traverses through the combinations assigned to each core without the need of storing the list in memory. In turn, it returns the list of combinations of k variants with the highest MI values.

Integer and floating-point vector operations are part of the vector functions implementing the association test as presented in [7]. The function COMBINE implements the construction of a genotype table from two previous input genotype tables, and the function COMBINE_AND_POPCOUNT combines in one function the construction of a genotype table with the computation of the contingency table from the previous genotype table using a population count function as explained in Sect. 3.2. These two functions are implemented using boolean and integer vector arithmetic. The function MUTUAL_INFORMATION implements the MI test, and uses floating point vector arithmetic.

In addition, *x86_64* processors are known to reduce the clock frequency attending to three factors: the number of active cores, the width of the VPU used and the type of operations used. For instance, the specification document [17] of the Intel Xeon processor (the processor used during the evaluation) defines different base frequencies attending to the number of cores and width of the AVX operations used. Furthermore, this processor reduces its turbo frequencies if vector floating-point arithmetic is used. To mitigate the impact of this frequency reduction, the SIMD algorithm, previously referenced, segments the operations into blocks so that each block can operate at a different frequency [7], and the same technique is applied to Algorithm 1.

The algorithm primarily consists of a *for* loop that traverses the list of variant combinations provided to the function (Line 13). The loop begins by computing the genotype table for each combination prefix $\{i_1, \ldots, i_{k-1}\}$. This is done in a progressive manner, starting with the table of the first variant of the prefix i_1, and adding one extra variant to the genotype table at a time using the function COMBINE, until the whole prefix is included in the table (Lines 14–Lines 17). Once this table is computed, every combination of k variants starting with the given prefix, $\{i_1, \ldots, i_{k-1}, j\}$ with $j \in [i_{k-1}, n-1]$, is examined using a *for* loop (Line 18). On each iteration, the genotype frequencies of the combination are obtained through the function COMBINE_AND_POPCOUNT, using the genotype table of the prefix and the table of the variant j (Line 26). The frequencies are stored in an array ct of contingency tables. Only when B contingency tables are available, the loop enters an *if* branch where the table array ct is processed altogether using a *for* loop (Lines 19–25), effectively25 separating the floating-point vector computations of the MUTUAL_INFORMATION function from the genotype table construction operations. On each iteration, a contingency table is processed by computing the MI of the table, and its result is stored in a list of S elements, sorted by its MI value using the auxiliary function defined in Lines 1–9.

Table 1 Hardware and software description of the SCAYLE cluster nodes from the `cascadelake` partition

SCAYLE node (cascadelake partition)	
CPUs	2x Intel 6240 (36 cores) @ 2.6GHz
Memory	192GB
Network	Infiniband HDR @ 100Gbps
GPUs	NVIDIA V100
OS	CentOS 7.7
Kernel	3.10
Compiler	GCC 11.2
Libraries	glibc 2.34
	OpenMPI 4.1.1

When the outermost *for* loop ends, the remaining contingency tables stored in the array *ct* are processed (Lines 30–33) and the algorithm returns the sorted list of the top-ranking *S* combinations (Line 34).

The beginning and the end are the only two points in the program requiring synchronization among threads and MPI processes. Once all threads of a process terminate, the different lists of top-ranking combinations kept in the shared memory of the process are joined into one, then sorted by their MI value and truncated to *S* combinations. Analogously, once all MPI processes have assembled their joint lists, the results are gathered into a single joint list through the MPI collective `MPI_Gatherv`. This list is then sorted by MI and truncated to *S* combinations again. To conclude, the program writes the final list to a file and exits.

5 Evaluation

This evaluation examines the proposed parallel method in terms of the balance achieved by the parallel distribution, the overhead introduced by the overlap in computations among the different processing units, the parallel efficiency achieved for an increasing number of processing units and a comparison with state of the art exhaustive epistasis detection software. Table 1 describes the characteristics of each node from the SCAYLE cluster used throughout the evaluation.

5.1 Parallel distribution balance

The distribution strategy presented in Sect. 4.1 does not assign the same exact number of combinations of k variants to test for association to every computing unit. Instead, the strategy makes a compromise between the balance in combinations assigned and the reuse of intermediate results.

In order to evaluate how good the designed strategy is, Fig. 5 plots the maximum percentual difference between the number of combinations assigned to a computing unit and the mean number of combinations assigned to any unit, relative to the latter. It can be defined as:

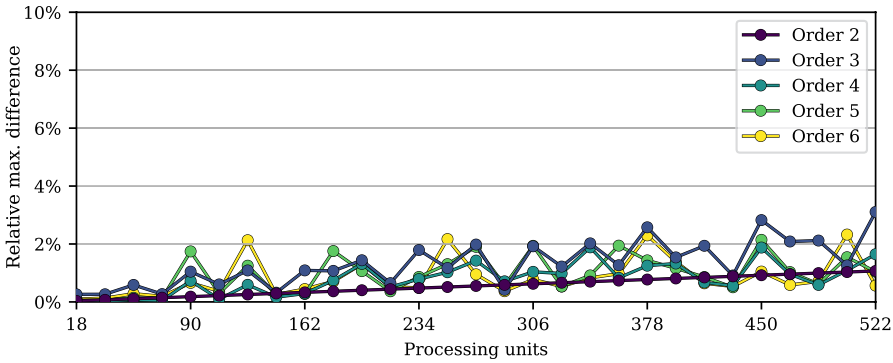


Fig. 5 Maximum difference of assigned combinations to any processing unit from the average number of combinations assigned per unit, relative to the latter, for orders ranging from 2 to 6

$$100\,\frac{\max d_i - \binom{n}{k}/p}{\binom{n}{k}/p} \tag{4}$$

with d_i being the number of combinations assigned to the unit i, n the number of variants, k the size of the combinations and p the number of processing units used. The figure represents the differences in workload distribution using combination sizes from 2 to 6 and a number of units from 18 to 522. In order to keep a similar number of k-combinations, and thus a similar distribution difficulty across combination orders, a number of variants of 48 828, 1928, 413, 172 and 100 were used for orders 2–6, respectively.

The results show that the proposed distribution keeps the differences under 3% for every scenario tested. For scenarios with a larger variant count, as is the case during the experimental evaluations of Sects. 5.3 and 5.4, the differences in assigned workload are even smaller.

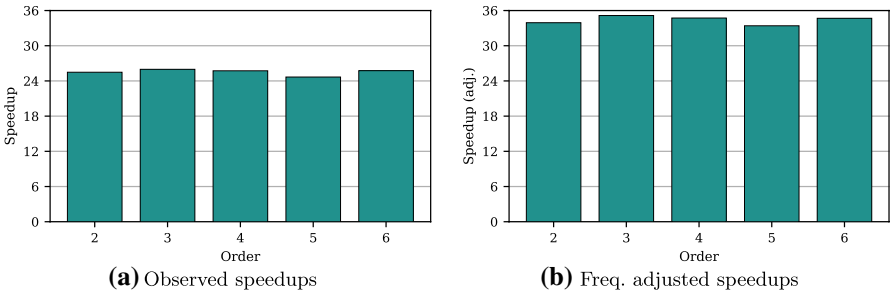
5.2 Parallel overhead

Although the distribution strategy takes into consideration the reuse of genotype tables to avoid repeating the same computations in different processing units, it certainly does repeat some operations during the construction of the genotype table corresponding to the combination prefix assigned by the distribution. In order to measure the overhead introduced, we compared the elapsed time of a single-thread execution of the proposed implementation with an alternative implementation of the same epistasis detection method that examines every combination of variants sequentially and avoids repeating any calculation pertaining to any genotype table.

Table 2 shows the overhead, measured as a percentage and calculated as $100 \cdot \left(T - T_{alt}\right)/T_{alt}$, with T being the elapsed time of the proposed implementation

Table 2 Overhead of the parallel algorithm (run using a single CPU core) compared to a sequential implementation of the same operation, for interaction orders between four and six

Order	Variants	Combinations	T (s)	T_{alt}(s)	Overhead (%)
4	464	1906472876	1514.61	1526.48	− 0.78
5	152	632671880	1477.57	1453.21	1.68
6	76	218618940	1518.63	1506.17	0.83



(a) Observed speedups **(b)** Freq. adjusted speedups

Fig. 6 Speedups of Fiuncho for multithread executions using 36 threads, compared to a single-thread run, representing both the observed and frequency-adjusted speedups

and T_{alt} being the elapsed time of the alternative sequential implementation. The number of input variants selected is inversely proportional to the order of the interaction in order to maintain the runtime manageable, while the number of samples per variant was kept constant (2048). The table omits the second and third-order overheads because, for those combination sizes, the distribution strategy does not produce any overlap in the computations associated with the calculation of genotype tables. The results indicate that there is no significant difference between the two elapsed times.

5.3 Speedup and efficiency

This subsection evaluates the speedup and efficiency of Fiuncho using one and multiple nodes. For both scenarios we selected a number of input variants inversely proportional to the order of the interactions so that the elapsed times of the analysis are similar, while the number of samples per variant was kept constant at 2048.

Figure 6 represents the speedups obtained by Fiuncho using a whole node (36 cores) when compared to single thread execution as seen in Table 3, for epistasis orders ranging between two and six. The figure shows two different metrics for the speedup: the observed and the frequency-adjusted speedup. The observed speedup is calculated as T_1/T_N, with T_1 being the elapsed time using a single CPU core and T_N the elapsed time using N CPU cores. This metric is far from the ideal efficiency, and this is due to the frequency scaling present in modern processors. Intel CPUs, in particular, adjust their maximum core frequencies attending to the number of active cores, with a larger frequency disparity if AVX instructions are used [17], as is the

Table 3 Elapsed times of single-thread executions of Fiuncho for interaction orders between two and six

Order	Variants	Combinations	Elapsed time (s)
2	184865	17087441680	2491.07
3	3246	5694987980	1612.47
4	464	1906472876	1514.61
5	152	632671880	1477.57
6	76	218618940	1518.63

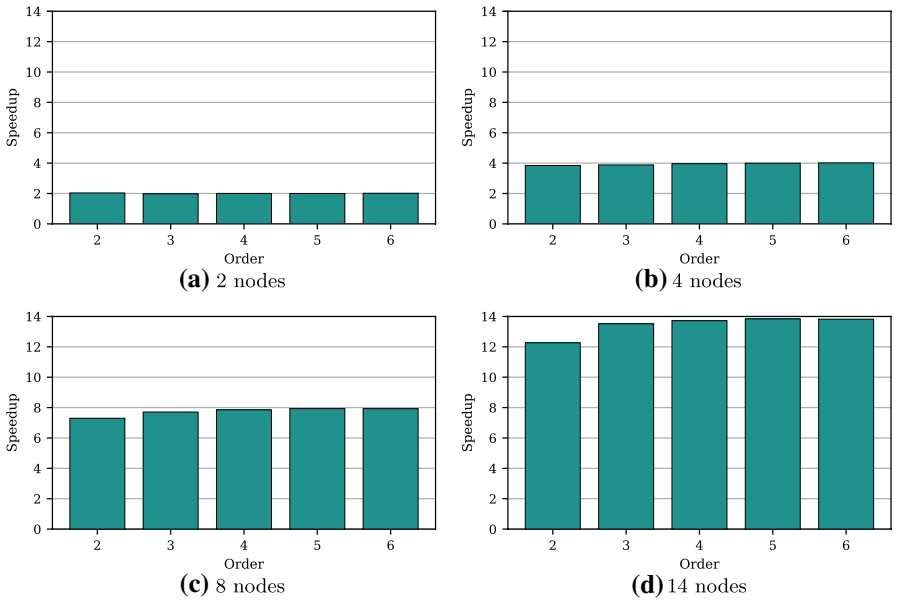


Fig. 7 Speedups of Fiuncho using 2, 4, 8 and 14 nodes with 36 threads per node, compared to a single-node execution

case with Fiuncho. Therefore, to get a better grasp of the efficiency of the parallel implementation, an adjusted speedup compensating for the discrepancy in average CPU frequency is included in the figure, calculated as $T_1/T_N \cdot F_1/F_N$, where F_1 is the average single-core frequency when Fiuncho uses a single core and F_N is the average multicore frequency when N cores are used. The results for a single-node (36 CPU cores) execution show very good efficiencies when the speedup is adjusted for the frequency differences between single-core and multicore executions.

Figure 7 shows the speedups obtained for multinode executions using one MPI process per node with 36 threads each, comparing the elapsed times obtained with a single-node run (36 cores) presented in Table 4. The datasets used in this second scenario are substantially larger than those from Table 3, in order to keep the elapsed times over an hour long when 14 nodes (504 CPU cores) are used. Here, in a multi-node environment, there is no difference between the average CPU frequency of the

Table 4 Elapsed times of single-node (36 cores) executions of Fiuncho for interaction orders between two and six

Order	Variants	Combinations	Elapsed time (s)
2	3755572	7052158645806	56261.70
3	28576	3888727096800	42539.30
4	2409	1399760565126	43176.00
5	561	454852770372	42594.70
6	223	159602946217	43103.40

Table 5 Elapsed time, in seconds, to complete an epistasis search both with MPI3SNP and Fiuncho, using a different number of nodes and CPU cores

Order	Variants	Combinations	Nodes	Cores	Fiuncho (s)	MPI3SNP (s)
3	3246	5694987980	1	1	1612.47	12868.42
3	8505	102498733260	1	18	2240.87	15312.84
3	10716	205033710860	1	36	2269.89	16046.58
3	13501	410062497750	2	72	2260.79	16085.50
3	17010	820134519120	4	144	2291.31	16186.12

different nodes since all of them use all the available CPU cores, and thus there is no need to include an adjusted measure of the speedup. Again, results show very good efficiencies except for the second-order interaction. This is due to the large input data for this interaction order, sizing over 29,386 MB and read sequentially, thus limiting the maximum speedup achievable.

5.4 Comparison with other software

Lastly, the performance of Fiuncho was compared with other exhaustive epistasis detection tools from the literature: MPI3SNP [4], MDR [3] and BitEpi [5]. To do this, we compared the elapsed times of all programs when looking for epistasis interactions of orders two to four. In order to keep the elapsed time constrained, multiple data sets were used containing a number of variants inversely proportional to the order of the epistasis search and the hardware resources used. The number of samples per variant, however, is 2048 for all data sets. Since MDR is considerably slower than the rest of the programs included, smaller data sizes were used for its evaluation.

Table 5 compares the elapsed times of Fiuncho and MPI3SNP, the tool previously developed by us. This program is limited to third-order searches, thus the evaluation only considers this interaction order. It implements MPI multiprocessing, so different scenarios were considered which include single-thread, single-node and multinode configurations. Both MPI3SNP and Fiuncho assign one MPI process per node, and create as many threads per process as cores available in each node. The results show that Fiuncho is significantly faster than MPI3SNP in all the evaluated scenarios.

Table 6 Elapsed time, in seconds, to complete an epistasis search both with BitEpi and Fiuncho, using different orders and number of CPU cores

Order	Variants	Combinations	Cores	Fiuncho (s)	BitEpi (s)
2	184865	17087441680	1	2491.07	18090.91
2	784314	307573833141	18	3582.01	22294.39
2	1109187	615147345891	36	3797.39	23365.74
3	3246	5694987980	1	1612.47	5417.15
3	8505	102498733260	18	2240.87	7474.48
3	10716	205033710860	36	2269.89	7564.48
4	464	1906472876	1	1514.61	2202.65
4	954	34296318126	18	2101.60	3239.25
4	1134	68539472001	36	2111.74	3246.72

The total workload between orders was kept as similar as possible

Table 7 Elapsed time, in seconds, to complete an epistasis search both with MDR and Fiuncho, using different orders and number of CPU cores

Order	Variants	Combinations	Cores	Fiuncho (s)	MDR (s)
2	9300	43240350	1	6.26	3571.77
2	39455	778328785	18	13.16	8656.46
2	55797	1556624706	36	16.04	10285.10
3	580	32350660	1	9.22	3204.88
3	1518	581842316	18	12.94	4710.82
3	1913	1164963436	36	13.34	6598.97
4	160	26294360	1	20.97	3767.28
4	328	473490550	18	29.04	4710.03
4	390	949173615	36	29.37	6491.59

The total workload between orders was kept as similar as possible

Table 6 compares the results of BitEpi with Fiuncho. BitEpi is a very novel program that only supports interaction orders between two and four, thus the evaluation is restricted to those orders. Additionally, BitEpi supports multithreading, therefore single-thread and multithread scenarios are used. BitEpi uses a substantially different association test with a time-complexity of $O(n)$, while the association test used in Fiuncho has a time-complexity of $O(3^n)$. This can be observed in the results as a shrinking difference between the elapsed times with the epistasis size. Despite this, Fiuncho is still faster in all configurations tested. Furthermore, BitEpi does not support multinode environments and can only exploit the hardware resources of a single node, while Fiuncho can use as many resources as available in order to reduce even further the elapsed time of the search.

Lastly, Table 7 compares the elapsed time of MDR with Fiuncho, using a more limited number of variants than previous comparisons. MDR is a relatively old

program written in Java, but we decided to include it due to its relevance in the field. It implements an epistasis search supporting interactions of any order, although its elapsed time quickly becomes prohibiting even with a reduced input size, so we decided to keep the interaction orders between two and four. MDR supports multithreading, so single-thread and multithread scenarios were considered in this evaluation. Results show a massive difference in elapsed times, with an average speedup of 358 of Fiuncho over MDR. This speedup could be increased even further if we considered multinode scenarios for larger data sets, something that MDR does not support, unlike Fiuncho.

6 Conclusions

This paper presents Fiuncho, an epistasis detection program written in C++, with MPI directives and multithread support, that can be executed in CPU clusters. It supports any interaction order, and implements an association testing method based on the Mutual Information metric that has been proven to perform well in epistasis detection [2]. Fiuncho includes explicit SIMD implementations of the association test calculations to exploit the full computational capabilities of *x86_64* processors.

Fiuncho exhibits exceptional performance, with a parallel strategy that balances the workload remarkably well, obtaining computational efficiencies close to an ideal growth with the hardware resources provided. When compared to existing epistasis detection software, Fiuncho offers support for a wider scope of application with no limit on the target epistasis size, and performs the fastest of all programs considered in this study. For example, on average, Fiuncho is seven times faster than its predecessor, MPI3SNP [4], three times faster than BitEpi [5] and 358 times faster than MDR [3]. Moreover, the speedups over BitEpi and MDR could be multiplied if larger experiments on multinode environments were considered, as they are restricted to the hardware resources available in a single node.

The main limitation of Fiuncho is its computational complexity, which makes its cost prohibitive for large-scale studies and high interaction orders. For this reason, future work should focus on improving the exhaustive strategy so that its computational complexity can be reduced while not losing the epistasis detection capabilities characteristic of these methods.

Fiuncho is distributed as open-source software, available to all the scientific community in its Github repository[1].

Acknowledgements We would like to thank Supercomputación Castilla y León (SCAYLE), for providing us access to their computing resources.

Author contributions Conceptualization: all authors; Investigation: CP-F; Software: CP-F; Supervision: JG-D and MJM; Visualization: all authors; Writing—original draft: CP-F; Writing—review & editing: all authors.

[1] https://github.com/UDC-GAC/fiuncho

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 / AEI / 10.13039/501100011033), the Xunta de Galicia and FEDER funds of the EU (CITIC-Centro de Investigación de Galicia accreditation 2019–2022, Grant no. ED431G 2019/01), Consolidation Program of Competitive Research (Grant no. ED431C 2021/30), and the FPU Program of the Ministry of Education of Spain (Grant no. FPU16/01333).

Availability of data and materials Not applicable.

Code availability The source code is distributed as open-source software, available in the repository https://github.com/UDC-GAC/fiuncho.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.the authors have no competing interests to declare that are relevant to the content of this article.

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication All authors have reviewed the study and consented to its publication.

References

1. Génin E (2020) Missing heritability of complex diseases: case solved? Hum Genet 139(1):103–113. https://doi.org/10.1007/s00439-019-02034-4
2. Ponte-Fernandez C, Gonzalez-Dominguez J, Carvajal-Rodriguez A, Martin MJ (2020) Evaluation of existing methods for high-order epistasis detection. IEEE/ACM Trans Comput Biol Bioinf. https://doi.org/10.1109/TCBB.2020.3030312
3. Hahn LW, Ritchie MD, Moore JH (2003) Multifactor dimensionality reduction software for detecting gene-gene and gene-environment interactions. Bioinformatics 19(3):376–382. https://doi.org/10.1093/bioinformatics/btf869
4. Ponte-Fernández C, González-Domínguez J, Martín MJ (2020) Fast search of third-order epistatic interactions on CPU and GPU clusters. Int J High Perform Comput Appl 34(1):20–29. https://doi.org/10.1177/1094342019852128
5. Bayat A, Hosking B, Jain Y, Hosking C, Kodikara M, Reti D, Twine NA, Bauer DC (2021) Fast and accurate exhaustive higher-order epistasis search with BitEpi. Sci Rep 11(1):1–12. https://doi.org/10.1038/s41598-021-94959-y
6. Campos R, Marques D, Santander-Jiménez S, Sousa L, Ilic A (2020)Heterogeneous CPU+iGPU processing for efficient epistasis detection. In: European conference on parallel processing, pp 613–628. Springer, Berlin. https://doi.org/10.1007/978-3-030-57675-2_38

7. Ponte-Fernández C, González-Domínguez J, Martín MJ (2022) A SIMD algorithm for the detection of epistatic interactions of any order. Futur Gener Comput Syst 132:108–123. https://doi.org/10.1016/j.future.2022.02.009
8. Wienbrandt L, Kässens JC, Ellinghaus D (2021) SNPInt-GPU: tool for epistasis testing with multiple methods and GPU acceleration. In: Wong, K.-C. (ed.) Epistasis: methods and protocols, pp 17–35. Springer, New York. https://doi.org/10.1007/978-1-0716-0947-7_2
9. Nobre R, Ilic A, Santander-Jiménez S, Sousa L (2020) Exploring the binary precision capabilities of tensor cores for epistasis detection. In: 2020 IEEE international parallel and distributed processing symposium (IPDPS), IEEE, pp 338–347. https://doi.org/10.1109/IPDPS47924.2020.00043
10. González-Domínguez J, Wienbrandt L, Kässens JC, Ellinghaus D, Schimmler M, Schmidt B (2015) Parallelizing epistasis detection in GWAS on FPGA and GPU-accelerated computing systems. IEEE/ACM Trans Comput Biol Bioinf 12(5):982–994. https://doi.org/10.1109/TCBB.2015.2389958
11. Kässens JC, Wienbrandt L, González-Domínguez J, Schmidt B, Schimmler M (2015) High-speed exhaustive 3-locus interaction epistasis analysis on FPGAs. J Comput Sci 9:131–136. https://doi.org/10.1016/j.jocs.2015.04.030
12. Ribeiro G, Neves N, Santander-Jiménez S, Ilic A (2021) HEDAcc: FPGA-based accelerator for high-order epistasis detection. In: 2021 IEEE 29th annual international symposium on field-programmable custom computing machines (FCCM), IEEE, pp 124–132. https://doi.org/10.1109/FCCM51124.2021.00022
13. Nobre R, Ilic A, Santander-Jiménez S, Sousa L (2021) Fourth-order exhaustive epistasis detection for the xPU Era. In: 50th international conference on parallel processing, pp 1–10. https://doi.org/10.1145/3472456.3472509
14. Nobre R, Santander-Jiménez S, Sousa L, Ilic A (2020) Accelerating 3-way epistasis detection with CPU+GPU processing. In: Workshop on job scheduling strategies for parallel processing, pp 106–126. Springer. https://doi.org/10.1007/978-3-030-63171-0_6
15. Wienbrandt L, Kässens JC, Hübenthal M, Ellinghaus D (2019) 1000× faster than PLINK: combined FPGA and GPU accelerators for logistic regression-based detection of epistasis. J Comput Sci 30:183–193. https://doi.org/10.1016/j.jocs.2018.12.013
16. Wan X, Yang C, Yang Q, Xue H, Fan X, Tang NL, Yu W (2010) BOOST: a fast approach to detecting gene-gene interactions in genome-wide case-control studies. Am J Hum Genet 87(3):325–340. https://doi.org/10.1016/j.ajhg.2010.07.021
17. Corporation I (2020) Second generation intel xeon scalable processors specification update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/2nd-gen-xeon-scalable-spec-update.pdf. Accessed 7 Nov 2020