Algorithm xxx: Parallel Implementations for Computing the Minimum Distance of a Random Linear Code on Distributed-memory Architectures

GREGORIO QUINTANA-ORTÍ, FERNANDO HERNANDO, and FRANCISCO D. IGUAL

The minimum distance of a linear code is a key concept in information theory. Therefore, the time required by its computation is very important to many problems in this area. In this paper, we introduce a family of implementations of the Brouwer-Zimmermann algorithm for distributed-memory architectures for computing the minimum distance of a random linear code over $\mathbb{F}_2$. Both current commercial and public-domain software only work on either unicore architectures or shared-memory architectures, which are limited in the number of cores/processors employed in the computation. Our implementations focus on distributed-memory architectures, thus being able to employ hundreds or even thousands of cores in the computation of the minimum distance. Our experimental results show that our implementations are much faster, even up to several orders of magnitude, than current implementations widely used nowadays.

ACM Reference Format:
Gregorio Quintana-Ortí, Fernando Hernando, and Francisco D. Igual. 2020. Algorithm xxx: Parallel Implementations for Computing the Minimum Distance of a Random Linear Code on Distributed-memory Architectures. *ACM Trans. Math. Softw.* 0, 0, Article 0 (2020), 24 pages. https://doi.org/0

1 INTRODUCTION

In 1948, Claude Shannon published his seminal paper *"A Mathematical Theory of Communications"* [22], which is widely recognized as the foundation of the Information Theory field. As of today, this work is still considered a key reference in the area, as it describes the concept of information and how to measure it; actually, nowadays it is even more up-to-date than ever given the widespread development of network communications. As information is sensitive to be corrupted due to external factors –e.g. noise–, Coding Theory can be leveraged to detect and correct errors [16]. It is worthwhile mentioning that Coding Theory goes beyond error correction, as it can be used for many other purposes, namely: quantum computing [7], biological systems [1, 17], data compression [2, 15], cryptography [18, 19], network coding [13], or secret sharing [8, 21], among others.

For practical reasons, the most common codes employed are usually linear codes, i.e., vector subspaces C of dimension k within a vector space of dimension n. In addition to n and k, a crucial parameter to be considered is the Hamming minimum distance d of the vector subspace C, since it strongly determines error detection and error correction capabilities. In other words, if a corrupted word r is received, the most likely codeword that was sent is the closest to r in C, according to the mentioned metric. If the minimum distance of a linear code is d, up to $d-1$ errors can be detected and up to $\lfloor \frac{d-1}{2} \rfloor$ errors can be corrected. The knowledge of d is not only crucial to detect and correct errors, but also in the other applications mentioned above.

Authors' address: Gregorio Quintana-Ortí; Fernando Hernando; Francisco D. Igual.

0098-3500/2020/0-ART0 $15.00
https://doi.org/0

The scientific literature contains fast algorithms for computing –or bounding– the minimum distance of linear codes with a complex structure, such as the Reed-Solomon codes or the BCH codes [16]. In contrast, computing the minimum distance of random linear codes is an NP-hard problem. As of today, the fastest algorithm for computing this minimum distance is the so-called Brouwer-Zimmerman algorithm [24], which was described and updated by Grassl [9]. Nevertheless, there are some other algorithms that are faster in some special cases, such as when the dimension is small and the length is large. Hernando *et al.* [11] carried out a brief performance comparison of an implementation of Brouwer-Zimmermann and an exhaustive algorithm based on the Gray code for some linear codes with different dimensions and lengths. The commercial software MAGMA [4] contains an implementation of this algorithm over both small and large fields. The public-domain software GAP [3, 10] contains an implementation of this algorithm over $\mathbb{F}_2$ and $\mathbb{F}_3$. A family of implementations of this algorithm over $\mathbb{F}_2$ with much higher performances has been recently developed [11]. These implementations produced higher performances on both serial computers (unicore processors) and shared-memory architectures with multiple/multicore processors by exposing and efficiently exploiting thread-level and data-level parallelism by means of vector instructions.

After submitting this manuscript, Bouyukliev *et al.* [6] developed a modification of the Brouwer-Zimmermann algorithm that can accelerate the original algorithm in some cases by creating and processing additional systematic sets in order to enumerate a fewer number of codewords.

However, the computation cost of computing the distance of large random linear codes is really humongous even when employing shared-memory architectures with several cores. As the number of cores in these architectures is limited, a new approach is presented in this paper. We introduce several new efficient implementations, also over $\mathbb{F}_2$, that can be employed in distributed-memory architectures with hundreds (or even thousands) of cores. Experiments show that the new implementations are scalable and can compute the minimum distance of the random linear codes much faster than current optimized shared-memory implementations. In fact, the computation of the minimum distance of a random linear code with the public-domain GAP (Guava) software took 5 days, whereas the computation of the same distance took only 5 minutes with our new implementations.

This article is organized as follows: Section 2 introduces the necessary background in order to make this article self-contained. Section 3 describes several new algorithms and implementations for distributed-memory architectures. Section 4 presents the performances of the the new algorithms, and compares the results with current software. Section 5 contains the conclusions.

2 BACKGROUND

The objective of this section is twofold: First, to provide the necessary mathematical tools employed in the rest of the manuscript; second, to review the new fast implementations for computing the minimum distance of a random linear code introduced in [11].

Although a part of the mathematical background is described for $\mathbb{F}_q$, the algorithms are described and implemented over $\mathbb{F}_2$.

2.1 Mathematical Background

Let e be a prime number and $q = e^r$ a power of it. We denote by $\mathbb{F}_q$ the finite field with q elements. By definition, a linear code C is a vector subspace of $\mathbb{F}_q^n$. The dimension of C as a vector subspace is denoted by k and is referred as the dimension of the linear code. The encoding is done via the generator matrix, i.e., a $k \times n$ matrix denoted by G, whose rows form a base of C. After elementary row operations and columns permutations, any generator matrix can be written in the systematic form $G = (I_k \mid A)$, where I_k is the identity matrix of dimension k, and A is a $k \times (n-k)$ matrix. To

encode the information, it is only needed to multiply $(c_1, \ldots, c_k)(I_k \mid A) = (c_1, \ldots, c_k, c_{k+1}, \ldots, c_n)$, introducing $n - k$ new symbols, which eventually will help to detect and correct errors. In the decoding process, if a corrupted word is received, it is replaced by the closest codeword, in case it is unique. Therefore, a metric to measure the closeness is needed. In coding theory, the most common metric is the Hamming distance. Given two vectors $a, b \in \mathbb{F}_q$, the Hamming distance between $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ is:

$$\mathrm{d}(a, b) = \#\{i \mid a_i \neq b_i\}.$$

Then, the minimum distance of a linear code C is:

$$\mathrm{d}(C) = \min\{\mathrm{d}(a, b) \mid a, b \in C\}.$$

Since the code is linear, the minimum distance coincides with the minimum weight:

$$\mathrm{wt}(C) = \min\{\mathrm{wt}(a) \mid a \in C\},$$

where $\mathrm{wt}(a) = \mathrm{d}(a, 0)$.

For the sake of simplicity, in the following we will use d instead of $\mathrm{d}(C)$ will refer to either the minimum distance or the minimum weight. A linear code with minimum distance d can detect up to $d - 1$ errors and can correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors. Computing the parameter d for a random linear code is a NP-hard problem [23], and the corresponding decision problem is NP-complete.

To this day, the fastest algorithm for computing the minimum distance of a random linear code is the so-called Brouwer-Zimmerman algorithm [24], which was described and slightly modified in [9]. The method for $\mathbb{F}_2$ is outlined in Algorithm 1. The key components in this algorithm are the information sets, i.e, a subset of k indices $S = \{i_1, \ldots, i_k\} \subset \{1, \ldots, n\}$ such that the corresponding columns of the generator matrix G are linearly independent. There are several approaches for finding the maximum number of disjoint information sets. Despite its crucial role, the computational cost of this step is negligible compared with the rest of the algorithm. Assume that one has found $m - 1$ disjoint information sets S_j, $j = 1, \ldots, m - 1$. For each disjoint information set S_j, a matrix in systematic form $\Gamma_j = (I_k \mid A_j)$ can be obtained. In addition to that, there are $n - k(m - 1)$ columns in G that do not form an information set, i.e., the corresponding columns have rank strictly less than k, say k_m. After elementary row operations and column permutations, the following matrix can be obtained:

$$\Gamma_m = \begin{pmatrix} I_{k_m} & A \\ 0 & B \end{pmatrix}.$$

Once the matrices $\Gamma_1, \ldots, \Gamma_m$ have been computed, the process of enumerating codewords can proceed. This process is as follows: A lower bound L is initialized to one, and an upper bound U is initialized to $n - k + 1$. First, all the linear combinations of the form $c \cdot \Gamma_j$, $j = 1, \ldots, m$ where $\mathrm{wt}(c) = 1$ are generated. For each linear combination, if its weight is smaller than U, the upper bound U is updated to the new weight. After processing all the vectors c of weight one, the lower bound is increased in $m - 1$ units. Then, if $L \geq U$, the minimum weight is U and the algorithm stops. Otherwise, this same process is repeated for all vectors c such that $\mathrm{wt}(c) = 2$, then $\mathrm{wt}(c) = 3$, and so on, until $L \geq U$, in which case U is the minimum distance.

2.2 Improved algorithms

Hernando *et al.* [11] introduced several new algorithms and implementations that were faster than both current commercial software and public-domain software, including accelerated versions of both the brute-force algorithm and the Brouwer-Zimmermann algorithm. These algorithms were designed for both unicore systems and shared-memory architectures (multicore and multiprocessors). Since the new parallelizations for distributed-memory computers are based on and make

Algorithm 1 Minimum weight algorithm for a linear code C

Require: A generator matrix G of the linear code C with parameters $[n, k, d]$.
Ensure: The minimum weight d of C.

1: $L := 1; U := n - k + 1;$
2: $g := 1;$
3: **while** $g \leq k$ and $L < U$ **do**
4: **for** $j = 1, \ldots, m$ **do**
5: $U := \min\{U, \min\{\mathrm{wt}(c\Gamma_j) : c \in \mathbb{F}_2^k \mid \mathrm{wt}(c) = g\}\}$;
6: **end for**
7: $L := (m-1)(g+1) + \max\{0, g+1-k+k_m\}$;
8: $g := g + 1;$
9: **end while**
10: **return** $U;$

heavy use of these algorithms, we proceed first with a brief description of these serial versions. The reader can refer to [11] for an in-depth description and detailed analysis.

The focus of all the algorithms is the generation of all the linear combinations since it is the most compute-intensive part. The next descriptions and algorithms do not show the updates of the lower and upper bounds (L and U, respectively), nor the termination condition to simplify the notation. All the algorithms stop as soon as the lower bound is equal to or larger than the upper bound after processing a Γ matrix.

A common first step is the computation of the Γ matrices out of the generator matrix G. As the cost of this part is usually much cheaper than the rest of the algorithm, and its results can greatly affect the overall computational cost of the algorithm, several random permutations are applied to the original generator G in order to find one permutation with both the largest number of full-rank Γ matrices and the largest rank in the last Γ matrix. Lisoněk *et al.* [14] described how to perform a more clever election of the Γ matrices.

Once the Γ matrices have been computed, the basic goal of the Brouwer-Zimmermann algorithm is simple: For every Γ matrix, the additions of all the combinations of its rows taken one at a time must be computed, and then the minimum of the weights of those additions must be computed. This process is repeated then taking successively two rows at a time, then taking three rows at a time, etc. After processing each Γ matrix in each of these stages, the lower and upper bounds are checked, and this iterative process finishes as soon as the lower bound is equal to or larger than the upper bound.

2.2.1 Serial algorithms . All the algorithms presented hereafter focus on how to generate the different codewords, that is, line 5 in Algorithm 1, with as few additions as possible. All of them enumerate exactly the same number of codewords, which depends on the maximum number of linear combinations g generated. The number of linear combinations enumerated is:

$$m \sum_{j=1}^{g} \binom{k}{j},$$

where m is the number of Γ matrices. This number depends on the ranks of Γ matrices. On the other hand, the number of additions carried out in each algorithm is completely different. In [11], we describe in detail the cost in terms of additions of the code within the innermost for loop. Further interesting references for algorithms that minimize the number of additions are [12] as a general reference and [5] for linear codes.

Basic algorithm. This is an straightforward implementation of the Brouwer-Zimmermann algorithm. Let us say that a Γ matrix has k rows, then the basic algorithm generates all the combinations of the k rows taken with an increasing number of rows. For every generated combination, the rows of this combination are added, and the overall minimum weight is updated. This method is outlined in Algorithm 2. The `Get_first_combination()` function returns true and the row indices of the first combination, such as $(0, 1, 2, \ldots, g-1)$. In this one and the next algorithms a combination is represented as a sequence of row indices, where the first row index is zero. The `Get_next_combination( c )` function receives a combination c and returns both true and the next one if there is one. The `Process_combination( c, Γ )` function computes the weight of the addition of the rows of Γ with indices in c. Besides, it updates the lower and upper bounds if needed. Though in this algorithm the order in which the combinations are generated is not important, the lexicographical order was employed to reduce the number of cache misses.

Algorithm 2 Basic algorithm

Require: A generator matrix G of the linear code C with parameters $[n, k, d]$.
Ensure: The minimum weight of C, i.e., d.

```
1:  Beginning of Algorithm
2:  [ Γ_j ] = Compute_gamma_matrices( G );
3:  for g = 1, 2, ... do
4:      for every Γ matrix (k × n) of G do
5:          // Process all combinations of the k rows of Γ taken g at a time:
6:          ( done, c ) = Get_first_combination();
7:          while ( ! done ) do
8:              Process_combination( c, Γ );
9:              ( done, c ) = Get_next_combination( c );
10:         end while
11:     end for
12: end for
13: End of Algorithm
```

Optimized algorithm. In the lexicographical order, the only difference between each combination and the next one (or the previous one) is usually the last element; hence, this algorithm reduces the number of additions by saving and reusing the addition of the first $g-1$ rows. The outline of this algorithm is very similar to the previous one. The main difference is that combinations are generated with $g-1$ rows instead of g rows. Therefore, the new `Process_combination( c, Γ )` function must perform the following two tasks: First, it adds and saves the combination with the $g-1$ rows with indices in c. Then, it builds all the combinations with g rows by adding the corresponding rows to the previous addition, thus saving a considerable number of row additions (compute operations) and row accesses (memory operations).

Stack-based algorithm. The goal of this algorithm is to further reduce the number of additions needed to compute the addition of the $g-1$ rows, performed in each iteration of the `while` loop, by using a stack with $g-1$ vectors of dimension n and the lexicographical order. The stack, which only requires a few KB, stores data progressively. When the combination $c = (c_1, c_2, \ldots, c_{g-1})$, where c_i is a row index, is being processed, the stack contains the following elements (incremental additions): row c_1, the addition of rows (c_1, c_2), the addition of rows (c_1, c_2, c_3), ..., and finally the addition of rows $(c_1, c_2, c_3, \cdots c_{g-1})$. The main savings of this algorithm are obtained in the computation of

the addition of the $g-1$ rows because the top of the stack contains that information. Then, when computing the next combination of $g-1$ rows, the contents of the stack must be rebuilt from the left-most index that has changed between the current combination and the next one.

Algorithm with saved additions. The key to this algorithm is the efficient composition of combinations with up to s elements to build combinations with larger number of elements. If $g = a + b$ with numbers a and b such that $0 < a \leq s, 0 < b \leq s$, the addition of the rows of the combination c with indices $(c_1, c_2, \ldots, c_a, c_{a+1} \ldots, c_g)$ can be computed as the addition of the rows of the combination $(c_1, c_2, \ldots, c_a)$ (called left combination) and the combination $(c_{a+1} \ldots, c_g)$ (called right combination). In this way, with just one addition the desired result can be obtained if the additions of the combinations with up to at least $\max(a, b)$ rows have been previously saved. Therefore, if $g = a+b$, to obtain the combinations of k rows taken g at a time, the combinations of k rows taken a at a time (left combinations) and the combinations of k rows taken b at a time (right combinations) must be composed. However, not all those combinations have to be processed since there are some restrictions. These restrictions to the combinations must be applied efficiently to accelerate this algorithm since otherwise an important part of the performance gains could be lost.

The outline of the method is very different to the previous ones since it can be implemented as a recursive algorithm (see [11] for further details). The data structure that stores the saved additions of the combinations of the rows of every Γ matrix must be built in an efficient way. Otherwise, the algorithm could underperform for matrices that finish after only a few generators. For every Γ matrix, this data structure contains several levels ($l = 1, \ldots, s$), where level l contains all the combinations of the k rows of the Γ matrix taken l at a time. The way to do it in an efficient way is to use the previous levels of the data structure to build the current level. A similar approach proposed by Bouyukliev [5] further reduces the number of additions to one. This method is even faster, but its execution for larger number of rows is limited by the available memory. In contrast, our implementation performs some more additions in some cases, but it works for any number of rows (since it does not exceed the available total memory).

Algorithm with saved additions and unrolling. This algorithm reduces the number of memory accesses (not additions) by processing several left combinations at the same time (called *unrolling*). To do that, right combinations will be reused when brought from main memory. For instance, by processing two left combinations at the same time, the number of data being accessed can be nearly halved since each accessed right combination is used twice (one time for every one of the two left combinations), thus doubling the ratio of vector additions to vector accesses. However, this technique is more effective when the two left combinations must be composed with the same subsets. To achieve that, the right-most element of the two left combinations must be the same. As in the lexicographical order the right-most index always changes, a variant was employed.

Vectorization and other implementation details. The main advantage of hardware vector instructions is to be able to process many elements simultaneously by using data stored in large vector registers. Although the length n is usually smaller than a few hundreds (very small compared with the size of modern vector registers), an efficient vectorization was achieved. Four-byte integers were employed to store data, thus packing 32 elements into each integer and hence leveraging vector instructions to boost performance on modern and legacy computing architectures.

2.2.2 Parallel algorithms for shared-memory architectures . The parallelization of the basic, optimized, and stack-based algorithms do not render good results because of the restrictions of the loop sizes and the size of the critical regions in comparison with the amount of work that can be simultaneously executed. See Hernando *et al.* [11] for more details.

In contrast, the parallelization of the two algorithms with saved additions is much easier and more effective. However, to create a large-grain parallelism, it must be parallelized only for the first level of the recursion. Though we used a small critical region for updating the overall minimum weight (a reduction operation), the impact of this critical region was minimized because of the small computational cost of the operation and by making every thread work with local variables throughout its execution, and by updating the global variables just once at the end. In the parallelized codes, OpenMP [20] was employed.

3 NEW ALGORITHMS AND IMPLEMENTATIONS FOR DISTRIBUTED-MEMORY ARCHITECTURES

This section describes several new algorithms for distributed-memory architectures. Two families of algorithms have been developed: The first family employs a *dynamic* distribution of tasks, whereas the second family employs a *static* distribution of tasks. Each family comprises four different algorithms.

Algorithm 3 Distributed algorithm

Require: A generator matrix G of the linear code C with parameters $[n, k, d]$.
Require: A prefix size p (integer value).
Require: The number of processes to employ P (integer value).
Ensure: The minimum weight of C, i.e., d.

1: **Beginning of Algorithm**
2: i = Process_identifier(); // Number between 0 and P-1.
3: [Γ_j] = Compute_gamma_matrices(G);
4: **for** $g = 1, 2, \ldots$ **do**
5: **for** every Γ matrix $(k \times n)$ of G **do**
6: // Process all combinations of the k rows of Γ taken g at a time:
7: Process_all_combinations(Γ, g, p, i, P);
8: **end for**
9: **end for**
10: **End of Algorithm**

3.1 Distributed-memory algorithms outline

The main outline of all the distributed algorithms is common, and can be found in Algorithm 3. Although the structure is similar to the Brouwer-Zimmermann algorithm (see Algorithm 1) and the serial algorithm (see Algorithm 2), there are some subtle differences that need to be described. As usual in distributed-memory programming, in this distributed algorithm all processes execute the algorithm from the beginning. Then, inside the `Process_all_combinations` function both the distribution among processes and the parallel computation of all the combinations of the rows of Γ taken g at a time are performed. This distribution of combinations among processes can be performed in a dynamic or in a static way, thus generating two different families of algorithms. The `Process_all_combinations` function will be described later for both families of algorithms.

Note that this algorithm requires two new parameters: p is the so-called prefix size, and P is the number of processes to be employed. These two parameters strongly determine the parallelization of the computational process.

Usually, the prefix size p must be smaller than g. Although inside the innermost `for` loop combinations must contain g rows of Γ, in the computational process the basic task to be assigned

to processes contains only p rows (assuming $p \leq g$). If only one different combination of g rows was assigned to every process, the overhead generated by a too small computational task would make the parallelization very slow and thus useless. In contrast, by assigning p rows to every process, we can increase the size of the computational task assigned to every process. Once a process receives a combination c of p elements, it must generate and process all the combinations with g elements starting with c. Therefore, p determines the task size of the distributed algorithms, where each one can be assigned to a different process in a distributed implementation. A task or combination c with p elements is also called a *prefix*, since the combination c will be then employed to generate all the combinations with g elements starting with c.

It is important to study in detail the effect of the prefix size on the number of tasks and on the computational cost of tasks.

3.1.1 Impact of the prefix size on the number of tasks. First, the effect of the prefix size on the number of tasks is explored. The number of tasks is $\binom{k-(g-p)}{p}$ for every value of g being processed (every iteration of the `For g` loop). In fact, when working with usual values of k and g (k is usually smaller than a few hundreds and g is usually smaller than 20), the smaller the prefix size, the fewer tasks will be generated. With those values, just by increasing the prefix size by one, the number of prefixes (and tasks) to be processed is increased by about one order of magnitude.

3.1.2 Impact of the prefix size on the computational cost of tasks. A prefix size p requires that in every task only the proper combinations of $g - p$ elements are composed with the prefix. Therefore, the smaller the prefix size, the larger the computational cost of tasks will be, and vice versa.

It is interesting to note that the cost of processing a prefix is very heterogeneous, and it strongly depends on the right-most element in the prefix, since it determines the number of combinations with g elements to be generated starting with the prefix (or combination with p elements). Obviously, the smaller the right-most element of the prefix, the more combinations with g elements must be generated and processed, and the larger the right-most element of the prefix, the fewer combinations with g elements must be generated and processed. For instance, if $k = 50$, $g = 5$, and $p = 3$, the cost of processing the prefix $(0, 1, 2)$ is much larger than the cost of processing the prefix $(0, 1, 47)$, since the first prefix requires a lot of combinations with 5 elements to be generated and processed, whereas the second prefix only requires one combination with 5 elements to be generated and processed: $(0, 1, 47, 48, 49)$.

The advantage of distributing prefixes is that every prefix can be processed in parallel, but its main disadvantage is the extremely wide range of the computational cost of processing prefixes. Some prefixes require a lot of time, whereas other prefixes are almost instantaneous.

3.1.3 Orderings in combination generation. The distributed algorithm shown in Algorithm 3 can employ any order to generate the combinations with p rows inside the `Process_all_combinations` function. Nevertheless, in our implementation we have employed two orders: the *lexicographical order* and the *left-lexicographical order*, a variant of the first one:

- In the lexicographical order, the right-most element is the one that changes most. For instance, if $k = 6$, $g = 4$, $p = 3$, the prefixes are generated in the following order: $(0, 1, 2)$, $(0, 1, 3)$, $(0, 1, 4)$, $(0, 2, 3)$, $(0, 2, 4)$, $(0, 3, 4)$, $(1, 2, 3)$, $(1, 2, 4)$, $(1, 3, 4)$, and $(2, 3, 4)$.
  With this ordering, the computational cost of the prefixes is usually (although not always) decreasing, as it depends on the right-most element of the prefix. For instance, in the previous example the prefix $(0, 1, 4)$ appears before the prefix $(0, 2, 3)$, but the cost of processing the latter is higher than the cost of processing the former one.

- In the left-lexicographical order, the left-most element is the one that changes most. For instance, if $k = 6$, $g = 4$, $p = 3$, the prefixes are generated in the following order: $(0, 1, 2)$, $(0, 1, 3)$, $(0, 2, 3)$, $(1, 2, 3)$, $(0, 1, 4)$, $(0, 2, 4)$, $(1, 2, 4)$, $(0, 3, 4)$, $(1, 3, 4)$, and $(2, 3, 4)$.
  As can be seen, in this ordering the right-most element of each prefix is always the same as or larger than the previous one. Therefore, the first prefixes are always much more expensive than the last ones. This might be an advantage when scheduling prefixes to avoid that expensive prefixes arise in the final stages of the algorithm, thus slowing some processes and unbalancing the load.

3.2 Dynamic algorithms

Both dynamic algorithms and static algorithms share general outline depicted the Algorithm 3 presented above. However, both families of algorithms implement the the `Process_all_combinations` function in a different way. Therefore, only this function is presented when describing both families. Algorithm 4 depicts the implementation of the function for the dynamic family.

Algorithm 4 PROCESS_ALL_COMBINATIONS(Γ, g, p, i, P)

Require: *Gamma*: A gamma matrix.
Require: g: The number of elements of Γ taken at a time.
Require: p: The prefix size.
Require: i: The process identifier.
Require: P: The number of processes.

```
1:  Beginning of Algorithm
2:  if i = 0 then
3:      // Coordinator process.
4:      ( done, c ) = Get_first_combination_with_p_elements();
5:      while ( ! done ) do
6:          Receive request of prefix from any process.
7:          Send prefix c to the process that just asked for it.
8:          ( done, c ) = Get_next_combination_with_p_elements( c );
9:      end while
10:     for all the rest of processes do
11:         Receive request of new prefix and previous result from any process.
12:         Send a poisonous prefix c to the process that just asked for it.
13:     end for
14: else
15:     // Rest of processes.
16:     Send request of new prefix to process 0.
17:     Receive prefix c from process 0.
18:     while ( c is not poisonous ) do
19:         Process_prefix( c, Γ, g );
20:         Send request of new prefix and previous result to process 0.
21:         Receive prefix c from process 0.
22:     end while
23: end if
24: End of Algorithm
```

As can be seen, the `while` loop of the process 0 in the distributed algorithm generates and processes all the combinations of k rows taken p at a time, instead of g at a time.

The `Get_first_combination_with_. . .` function and the `Get_next_combination_with_. . .` function are similar to those with similar names in previous algorithms, but note that in this case they process combinations with p elements.

Another difference lies in the method employed to process combinations, called `Process_prefix`. This method computes the addition of all the combinations with g rows of Γ starting with the received combination c with p rows. To improve performances, the addition of the p rows in the received combination c (prefix) must be computed first. Then, the proper additions with $g - p$ rows must be computed and added to the previous addition. (Obviously, when $p > g$, no parallel work is generated since the prefix size is larger than the number of elements in a combination. In this case, no prefix distribution is performed, and prefix replication is done instead.)

For example, if $g = 5$, $p = 3$, the lexicographical order is employed, and the combination $(0, 1, 2)$ is assigned; this method must compute all the combinations with 5 elements starting with $(0, 1, 2)$, that is, $(0, 1, 2, 3, 4)$, $(0, 1, 2, 3, 5)$, $(0, 1, 2, 3, 6)$, . . ., $(0, 1, 2, 4, 5)$, $(0, 1, 2, 4, 6)$, . . ., etc. To save work, first the addition of the rows $(0, 1, 2)$ must be computed, and then the addition of all the combinations with $g - p = 2$ rows starting at least with the index row 3 (the right-most element of $(0, 1, 2)$ plus one) must be computed.

Although the right-most element in a combination of g rows can be up to $k - 1$, the right-most element of a prefix with p rows ($p < g$) must be smaller than or equal to $k - 1 - (g - p)$, because no valid combination of k elements taken g at a time can be formed with a value larger than $k - 1 - (g - p)$ in the position p. For instance, if $k = 6$, $g = 4$, $p = 3$, the right-most element of valid prefixes must be smaller than or equal to 4, since obviously no valid combination of 6 (k) elements taken 4 (g) at a time can be formed starting with a prefix such as $(0, 1, 5)$.

Note that the processing of a prefix within the `Process_prefix` method is performed by only one process, and therefore it can be performed by using any of the previous serial or shared-memory algorithms, thus making the code more modular. In the case of the algorithms with saved additions special care must be applied since combinations must be saved up to size s. The code that is executed inside the `Process_prefix` method to process a prefix is called the *node engine* because it is executed by only one process, and thus it is executed inside one node of a distributed-memory machine. The order in which the combinations with $g - p$ rows are generated inside the `Process_prefix` method is not important to the distributed algorithm, and it is determined by the node engine (the serial or the shared-memory algorithm).

All the dynamic algorithms work in a similar way: They apply the one-master-n-workers model. As can be seen, one process in the application is the coordinator process, and the remaining processes are workers.

The coordinator process generates the prefixes (combinations with p elements) in a certain order and assigns them to the worker processes under request. When one worker has finished the processing of its current prefix, it sends the minimum distance computed for that prefix, and waits for the next prefix. The sending of the result tells the coordinator process that it has finished, and therefore it is ready to accept another prefix. When the coordinator receives a result from a worker, it updates its global minimum distance, and then sends the next prefix to that worker. When there are no more prefixes, it sends a special message (*a poisonous task*) to indicate the finishing condition.

When all the prefixes with p elements for the combinations with g elements are done, the `Process_all_combination` method finishes. Then, the current iteration of the `For` g loop (line 3 of Algorithm 3) finishes, and the next iteration starts if the values of the lower and upper bounds allow it.

We propose four different algorithms in this family:

- `D-Lex`: Distributed algorithm with a dynamic scheduling of tasks, in which the prefixes are generated in the lexicographical order.
- `D-Lex-2cm`: Same as the previous one, but two prefixes are assigned at the same time (within the same message). This technique reduces the communication cost (latencies) of the previous algorithm, that could be very effective in networks with high latencies. Targeting load balancing, one prefix is picked up from the beginning of the ordering and the other one is picked up from the end of the ordering, since the first prefixes are usually more computationally expensive.
- `D-Lle`: Distributed algorithm with a dynamic scheduling of tasks, in which the prefixes are generated in the left-lexicographical order.
- `D-Lle-2cm`: Same as the previous one, but two prefixes are assigned at the same time (within the same message). The rationale has just been described above.

3.3 Static algorithms

As previously told, the main algorithm outline has already been shown before, and only the `Process_all_combinations` function remains to be presented. Algorithm 5 shows that function in this case.

Algorithm 5 PROCESS_ALL_COMBINATIONS(Γ, g, p, i, P)

Require: *Gamma*: A gamma matrix.
Require: g: The number of elements of Γ taken at a time.
Require: p: The prefix size.
Require: i: The process identifier.
Require: P: The number of processes.

```
1: Beginning of Algorithm
2: ( done, c ) = Get_first_combination_with_p_elements_for_process( i );
3: while ( ! done ) do
4:     Process_prefix( c, Γ, g );
5:     ( done, c ) = Get_next_combination_with_p_elements_for_process( i, P, c );
6: end while
7: End of Algorithm
```

All the algorithms from the static family work in a similar way. In this family, all the processes in the application are peer, and therefore no coordination role is necessary. The distribution of the tasks is made in a static way. By calling to the `Get_first_combination_with_`p`_elements_for_process` function, and the `Get_next_combination_with_`p`_elements_for_process` function each process works on different prefixes, but the prefixes processed by every process are exactly the same ones in any run with the same parameters.

For example, if $g = 5$, $p = 3$, $P = 2$, the lexicographical order is employed, and the static cycling data distribution is employed, the calls to `Get_first_combination_with_`p`_elements_for_process` function made by process 0 and process 1 return the combinations $(0, 1, 2)$ and $(0, 1, 3)$, respectively. In this same case, the first calls to `Get_next_combination_with_`p`_elements_for_process` made by process 0 and process 1 return the combinations $(0, 1, 4)$ and $(0, 1, 5)$, respectively.

The cyclic distribution (or a similar variant) has been employed because it provides a better load balancing than the block distribution since the most expensive prefixes are usually the first ones.

One important difference between this family and the previous one is the handling of the prefix size. The dynamic family employs an absolute prefix size, whereas the static family employs a so-called relative prefix size since the actual prefix size is: $g - p$, where p is the given prefix size. To achieve this, the only easy change to the previous algorithm is to modify the `Get_first_combination_with_. . .` function and the `Get_next_combination_with_. . .` function to process combinations with $g-p$ elements, instead of p elements. The reason to use a relative prefix in the static algorithms is that the static scheduling requires a larger number of tasks to achieve a good load balancing of the workload across the processes. Since there must be many tasks and they cannot be too small, the actual prefix size p should increase when g increases (in every iteration of the `For` g loop). The use of relative prefix sizes achieves this, thus making that some values of the relative prefix size work fine on a wide range of g values, and on a wide range of linear codes.

Next, the similarities and differences among the four algorithms of this family are described:

- `S-Lex`: Distributed algorithm with static scheduling of tasks that employs the cyclic distribution of tasks generated using the lexicographical order.
  For example, the following table shows the distribution of prefixes for $k = 11$, $g = 4$, $p = 3$, and three processes. Recall that the right-most element in those prefixes must be 9, because no valid combination of 11 (k) elements taken 4 (g) can be formed with the value 10 in the third position.

Process 0	Process 1	Process 2
(0, 1, 2)	(0, 1, 3)	(0, 1, 4)
(0, 1, 5)	(0, 1, 6)	(0, 1, 7)
(0, 1, 8)	(0, 1, 9)	(0, 2, 3)
(0, 2, 4)	(0, 2, 5)	(0, 2, 6)
⋮	⋮	⋮

- `S-Lex-Snc`: Same as the previous one, but a variant of the cyclic distribution, called *snake cyclic*, is employed. In the usual cyclic distribution, the right-most elements of the prefixes assigned to the i-th process are very often (but not always) smaller than the right-most elements of the prefixes assigned to the $(i + 1)$-th process, which can unbalance the load by assigning more work to the first processes. You can compare the right-most elements of every two consecutive columns in the above example. The snake variant of the cyclic distribution tries to break this frequent event by using the usual cyclic distribution in half of the cases (the odd rows of the table), and then reversing the usual cyclic distribution in the other half (the even rows of the table). In this way, the right-most elements of the prefixes assigned to a process are not so often smaller that those assigned to the next process.
  For example, the following table shows the distribution of prefixes for $k = 11$, $g = 4$, $p = 3$, and three processes.

Process 0	Process 1	Process 2
(0, 1, 2)	(0, 1, 3)	(0, 1, 4)
(0, 1, 7)	(0, 1, 6)	(0, 1, 5)
(0, 1, 8)	(0, 1, 9)	(0, 2, 3)
(0, 2, 6)	(0, 2, 5)	(0, 2, 4)
⋮	⋮	⋮

- `S-Lle`: Distributed algorithm with static scheduling of tasks that employs the cyclic distribution of tasks generated using the left-lexicographical order.

The combination of the cyclic distribution and this new ordering achieves the following two goals: First, it ensures that the most expensive prefixes are processed at the beginning. Second, it ensures that the right-most elements of the prefixes assigned to a process are very similar (usually the same) to those assigned to the next process.
For example, the following table shows the distribution of prefixes for $k = 12$, $g = 3$, $p = 2$, and three processes.

Process 0	Process 1	Process 2
(0, 1, 2)	(0, 1, 3)	(0, 2, 3)
(1, 2, 3)	(0, 1, 4)	(0, 2, 4)
(1, 2, 4)	(0, 3, 4)	(1, 3, 4)
(2, 3, 4)	(0, 1, 5)	(0, 2, 5)
⋮	⋮	⋮

- `S-Lle-Snc`: Same as the previous one, but the snake cyclic distribution of tasks is employed. Although in the previous algorithm the right-most elements in the prefixes assigned to a process are often the same as those assigned to the next process, in the few remaining cases the right-most elements assigned to a process are smaller (by one) than those assigned to the next process. The snake cyclic distribution tries to avoid that fact by reversing the ordering in half of the cases (the even rows of the table). Thus, the right-most elements assigned to a process will be often the same as those assigned to the next process, and in the few remaining cases the right-most elements assigned to a process will be smaller (by one) or larger (by one) than those assigned to the next process.
For example, the following table shows the distribution of prefixes for $k = 12$, $g = 3$, $p = 2$, and three processes.

Process 0	Process 1	Process 2
(0, 1, 2)	(0, 1, 3)	(0, 2, 3)
(0, 2, 4)	(0, 1, 4)	(1, 2, 3)
(1, 2, 4)	(0, 3, 4)	(1, 3, 4)
(0, 2, 5)	(0, 1, 5)	(2, 3, 4)
⋮	⋮	⋮

3.4 Comparison of the dynamic algorithms and the static algorithms

3.4.1 Communication cost. In the dynamic algorithms, the coordinator must send every prefix to a worker under request. Then, after being processed by the worker, the worker must send the result back to the coordinator. Although the amount of data is not very large since the prefix comprises only p indices (integer values) and the result is just one value (an integer), the number of point-to-point communication is considerable. The communication cost of the dynamic algorithms depends on the number of tasks, which strongly depends on the prefix size (p). In fact, the communication cost is $2(\binom{k-(g-p)}{p})$ point-to-point operations for every value of g being processed (every iteration of the `For` g loop). The dynamic algorithms with suffix `2cm` reduce this communication cost by sending two combinations per message. Obviously, a large value of p would greatly increase the communication cost by creating many tasks, and therefore many point-to-point communications. In fact, when working with usual values of k and g (such as those in the experimental section), just by increasing the prefix size by one, the number of prefixes is increased by about one order of magnitude, and therefore the communication cost increases by the same order.

In contrast, the communication cost of the static algorithms is much smaller, since they do not send each prefix and do not receive each result. The assignment of tasks is static and requires no

communication at all, and the computation of the global result requires one collective reduction operation (no point-to-point communications at all) after processing all the prefixes for every value of g being processed (every iteration of the `For` g loop). The goal of the collective reduction operation is to compute the minimum distance of the distances computed by all the processes. Actually, the number of global reduction operations per iteration of the `For` g loop is two (instead of one) to avoid uninitialized distances. Nevertheless, the cost is much smaller than that of the dynamic algorithms. Note that the communication cost of the static algorithms does not depend on k, g, nor p, and it only depends on the number of processes logarithmically and the total number of iterations of the `For` g loop.

3.4.2 Number of tasks. Now the effect of the number of tasks on both distributed families is explored. In the dynamic family, as said, a large number of tasks increases the communication cost. Therefore, a large prefix size will greatly increase the number of tasks and thus the communication costs. Nevertheless, despite the dynamic nature of the scheduling, a too low number of tasks could unbalance the load. To guarantee a good load balancing across all the processes, the number of tasks should be at least several times larger than the number of processes being employed. If the number of tasks is too small (and therefore the variability is very large), several processes could be processing a prefix with a large computational cost, while others could have already finished all their work. Therefore, a balance must be found between the communication costs and the load balancing, since the reduction of the communication cost requires few tasks, whereas a better load balancing requires a large number of tasks.

In contrast, in the static family, since the communication cost does not depend on the number of tasks, a large number of tasks can render better performances because a large number of tasks with smaller computational costs can be more evenly distributed among the processes.

The dynamic algorithms employ one process, the coordinator process, to assign the work to be done and to gather the results. This can be a disadvantage when employing a low number of process because the coordinator process is not really processing combinations. In contrast, the static algorithms employ all the processes to work on combinations.

Table 1 summarizes these findings by linking the prefix size with the communication cost and the load balancing.

Table 1. Relationship between the actual prefix size, communication costs, and load balancing

Prefix size	No. of tasks	Task size	Dynamic algs.	Static algs.
Small	Few	Large	Low comm. cost Bad load balancing	Fixed comm. cost Bad load balancing
Medium	Medium	Medium	Medium comm. cost Good load balancing	Fixed comm. cost Bad load balancing
Large	Many	Small	High comm. cost Good load balancing	Fixed comm. cost Good load balancing

4 PERFORMANCE ANALYSIS

4.1 Experimental setup

The experiments reported in this article were performed on the following two computing platforms:

- ua: This is a multicomputer in which each node contained two Intel Xeon® CPU X5560 processors running at 2.8 GHz, with 12 cores and 48 GiB of RAM in total.

The nodes were connected with an Infiniband 4X QDR network. This network is capable of supporting 40 Gb/s signaling rate, with a peak data rate of 32 Gb/s in each direction.
The OS of each node was GNU/Linux (Version 3.10.0-514.21.1.el7.x86_64). OpenMPI 1.4.3 was employed to compile (the `mpicc` compiler) and to deploy the implementations on the cluster (the `mpirun` tool).

- `skx`: This is a subset of compute nodes of the Stampede2 supercomputer at Texas Advanced Computing Center. Each node contained two Intel Xeon® CPU Platinum 8160 ("Skylake") running at 2.1 GHz with 48 cores and 192 GB in total.
  The nodes were connected with a 100 Gb/s Intel Omni-Path (OPA) network with a fat tree topology employing six core switches. There is one leaf switch for each 28-node half rack, each with 20 leaf-to-core uplinks (28/20 oversubscription).
  The OS of each node was GNU/Linux (Version 3.10.0-957.5.1.el7.x86_64). Intel MPI from Intel C compiler Version 18.0.2 20180210 was employed.

In this experimental study we have usually employed the two following linear codes with parameters $[n,k,d]$: The first one had parameters [150,77,17] and was called `mat015`; the second one had parameters [232,51,61] and was called `mat023`. These two different linear codes were chosen because the computational costs, the dimensions k, and the lengths n were very different. First, the computational cost of computing the minimum distance of `mat023` is about one order of magnitude larger than that of `mat015`. Second, the dimension k of `mat015` is larger than that of `mat023`, which is a critical factor since th number of parallel tasks depends on this value. Third, the length n of `mat023` is much larger than that of `mat015`, which can affect the vectorization and other aspects of the different implementations. Usually, the left plot shows the results for `mat015`, whereas the right plot shows the results for `mat023`.

In the following, we report experimental results for the assessment of the following distributed algorithms:

- `D-Lex`: Distributed algorithm with a dynamic scheduling of tasks generated using the lexicographical order.
- `D-Lex-2cm`: Same as the previous one, but two tasks are assigned at the same time.
- `D-Lle`: Distributed algorithm with a dynamic scheduling of tasks generated using the left-lexicographical order.
- `D-Lle-2cm`: Same as the previous one, but two tasks are assigned at the same time.
- `S-Lex`: Distributed algorithm with a static cyclic scheduling of tasks generated using the lexicographical order.
- `S-Lex-Snc`: Same as the previous one, but the snake cyclic distribution of tasks is employed.
- `S-Lle`: Distributed algorithm with a static cyclic scheduling of tasks generated using the left-lexicographical order.
- `S-Lle-Snc`: Same as the previous one, but the snake cyclic distribution of tasks is employed.

4.2 Impact of node engine

The first round of experiments includes two versions of the node engines previously described: node engines with scalar (non-vectorized) codes, and node engines with vectorized codes. Figure 1 reports the times spent by the different node engines on `ua` to compute the minimum distance of both linear codes when using the algorithm `D-Lex` with prefix 3 and 1 thread per process on 10 nodes (120 cores), including scalar (`Sca`) and vectorized (`Vec`) versions. Results for other configurations (distributed algorithms, prefixes, number of threads per process, etc.) were observed to be similar.

The obtained results clearly show that the node engine employed within the distributed algorithm can dramatically affect performance. When comparing the vectorized codes of the saved variants

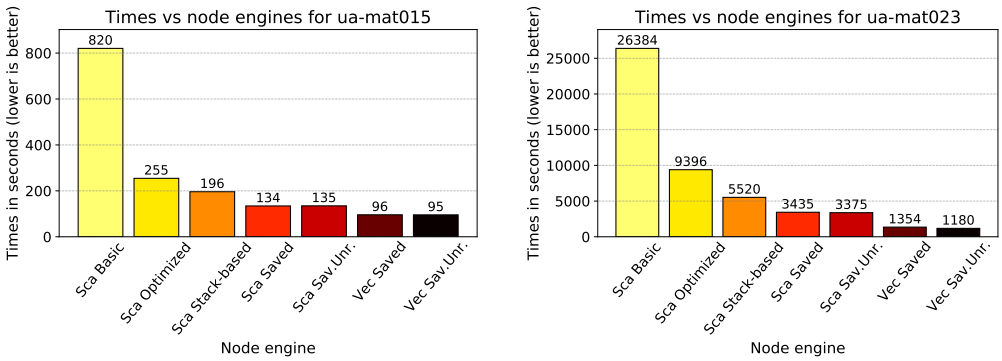

Fig. 1. Times (in seconds) of several node engines on ua for the algorithm D-Lex with prefix 3 and 1 thread per process on the two linear codes. Sca means scalar codes (non-vectorized), whereas Vec means vectorized codes. To improve legibility, the time in seconds is shown on top of all bars.

(Vec Saved and Vec Saved Unrolled) and the scalar codes of the saved variants (Sca Saved and Sca Saved Unrolled), the vectorized codes are about 1.4 times as fast as the scalar codes for the mat015 linear code, and the vectorized codes are about 2.5 times as fast as the scalar codes for the mat023 linear code. The main reason behind the larger impact on the mat023 linear code might be its larger length ($n = 232$ versus $n = 150$). On the other side, the unrolling only seems effective for the mat023 linear code, which can be caused by the larger computational cost of this linear code.

Unless otherwise stated, from now on, the node engine with saved additions and vectorization will been employed in the remaining experiments.

4.3 Impact of the prefix size

Figure 2 reports the times spent by different distributed implementations on ua to compute the minimum distance versus the prefix size, using one thread per process. Each plot shows four lines: Two different distributed algorithms (D-Lex Vec and S-Lex Vec), and two different node configurations (5 and 15 nodes, that is, 60 and 180 cores). Prefix sizes are absolute for the dynamic algorithms, and relative the static algorithms. The D-Lex Vec name means the distributed algorithm D-Lex and the vectorized Saved node engine. Analogously, the S-Lex Vec name means the distributed algorithm S-Lex and the vectorized Saved node engine. Similar results were obtained for the remaining distributed algorithms.

As can be seen in Figure 2, for the dynamic algorithm D-Lex the prefix size with the best performances is 3 for the mat015 linear code, and 4 for the mat023 linear code. For this dynamic algorithm, performances drop very quickly as the prefix size increases. This is due to the fact that for the dynamic algorithms the number of parallel tasks generated, assigned and then recollected among the processes is $2(\binom{k-(g-p)}{p})$, where g is the number of rows in the combinations, and p is the prefix size. Therefore, as the prefix increases in one unit, the number of tasks to be processed increases in nearly one order of magnitude, which correspondingly increases the communications costs.

As observed in Figure 2, for the static algorithm S-Lex the relative prefix size with the best performances is 4 for mat015, and 6 for mat023. Note that the performances of this algorithm are not so affected by the prefix sizes, and the range of optimal prefix sizes is much larger. The reason is that the communication cost of this algorithm is much smaller and therefore having a larger number of tasks does not usually harm performances so much.

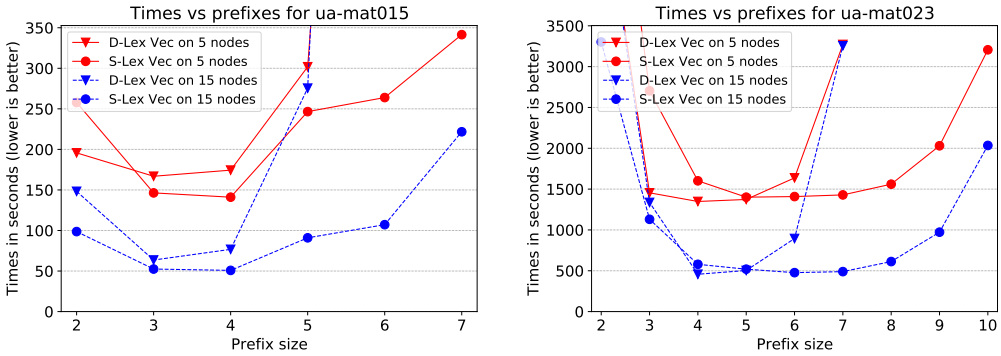

Fig. 2. Times (in seconds) of two distributed algorithms on ua for several prefix sizes on the two linear codes.

4.4 Number of threads per process

Figure 3 reports the times spent by the distributed implementations on ua to compute the minimum distance versus the number of threads per process. The prefix sizes have been obtained from the previous experiment: The dynamic algorithms employ 3 for `mat015`, and 4 for `mat023`, whereas the static algorithms employ 4 for `mat015`, and 6 for `mat023`. Each plot shows four lines: Two different distributed algorithms (`D-Lex Vec` and `S-Lex Vec`), and two different node configurations (5 and 15 nodes, that is, 60 and 180 cores). Similar results were obtained for the other distributed algorithms.

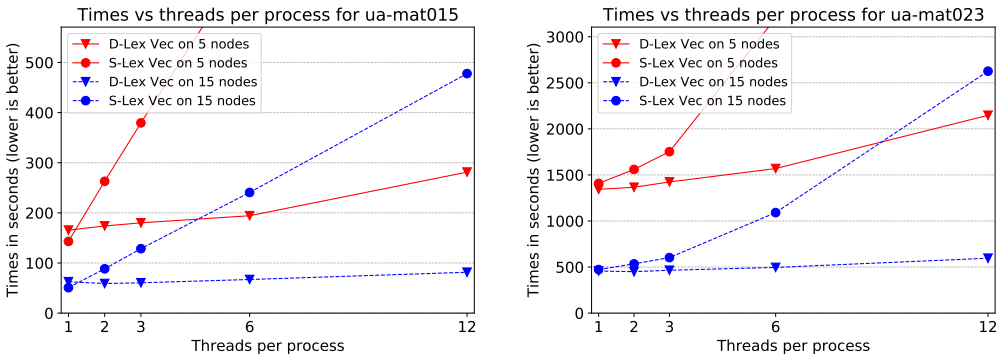

Fig. 3. Times (in seconds) for different number of processes per node and nodes on ua for the two linear codes.

To efficiently employ all the cores in every node, when increasing the number of threads per process, a proportional reduction in the number of total processes must be applied. If each node has 12 cores, l is the number of nodes being used, and t is the number of threads being deployed by each process, then the number of processes must be: $p = (l \cdot 12)/t$. When the number of threads per process is increased, the communication cost is usually reduced since there are fewer processes communicating among themselves. In contrast, the computing power of each process is increased since each process has several threads and therefore several cores to process tasks. This larger computational power per process requires larger tasks, which can only be achieved by

generating fewer tasks, which can unbalance the load. Therefore, a balance must be found between the communication cost and the number of tasks.

As can be observed in Figure 3, for the dynamic algorithm `D-Lex` the optimal number of threads per process is 1 when using 5 nodes, and about 2 or 3 when using 15 nodes. In the first case (5 nodes) the number of total processes is not so high, and thus the coordinator process can keep up with the requests. However, in the second case (15 nodes) the number of total processes is much higher, and thus the burden on the coordinator process can reduce performances. In this case, 2 or 3 threads per process are optimal, and achieve a good balance between the communication cost and the task size. On the other hand, for the static algorithm `S-Lex` the optimal number of threads per process is 1, since the communication cost of this type of algorithms is very small, and they require many tasks to effectively balance the load.

4.5 Distributed-memory algorithms: performance comparison

Figure 4 compares the proposed distributed implementations on ua. The prefix sizes and the numbers of threads per process employed in these experiments are the optimal values obtained in the above experiments. For the dynamic algorithms, the prefix size employed by the dynamic algorithms is 3 for `mat015`, and 4 for `mat023`, and the number of threads per process is 2 in both cases. For the static algorithms, the prefix size employed by the dynamic algorithms is 4 for `mat015`, and 6 for `mat023`, and the number of threads per process is 1 in both cases. Each plot contains two blocks of bars: one for 5 nodes and the other one for 15 nodes. Each block shows the performances of the eight distributed algorithms. In all cases, the vectorized `Sav` node engine has been employed.

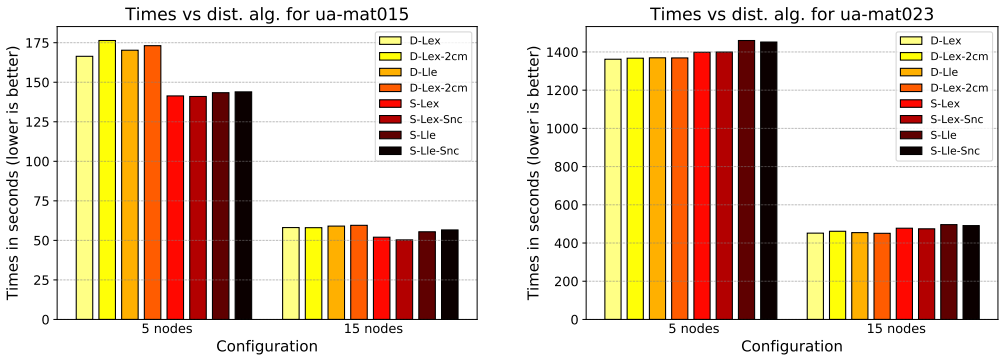

Fig. 4. Times (in seconds) for several distributed algorithms and nodes on ua for the two linear codes.

When comparing the dynamic algorithms with the static algorithms, the static algorithms clearly outperform the dynamic algorithms on the `mat015` linear code. This improvement is larger when the number of nodes is smaller. In contrast, on the `mat023` linear code dynamic algorithms are slightly faster. The reason of this performance difference in the two linear codes might be that the computation of the linear distance of `mat015` requires the generation of a much larger number of tasks than the computation of the linear distance of `mat023`. Recall that the number of tasks generated by the dynamic algorithms is $2(\binom{k-(g-p)}{p})$, and that $k = 77$ in `mat015`, whereas $k = 51$ in `mat023`. A large number of tasks (`mat015`) allows the static algorithms to balance the load more evenly, while simultaneously taking advantage of their lower communication cost. Moreover, if the number of cores is not so large (such as in the 5-node configuration), performances of the static algorithms increase because the load balancing of the static algorithms improves with fewer

processes and because the static algorithms employ one more process to perform computations. In contrast, a smaller number of tasks (`mat023`) allows the dynamic algorithms to balance the load more evenly than the static algorithms while simultaneously reducing the communication cost. Therefore, the static algorithms seem to require a large number of tasks to balance the load evenly on all the cores, which is a bit difficult when k is small.

When comparing the four dynamic algorithms, performances are similar. The `mat015` linear case (the one with the shortest computational cost), and the 5-node configuration, performances of the `D-Lex` are the better. In the other cases, performances are very similar. When comparing the four static algorithms, the performances of the two `D-Lex` variants (lexicographical order) are slightly better than those of the two `D-Lle` variants (left-lexicographical order).

Figure 5 compares the distributed implementations on `skx`. The prefix sizes for the dynamic and the static algorithms employed in these experiments are 5 and 6, respectively, since they are optimal or very close to optimal. In both cases, the numbers of threads per process is two. In this case, we only assessed the `mat023` linear code since it is more expensive and many cores are going to be employed. The plot contains four blocks of bars for several number of nodes: 8 nodes (384 cores), 16 nodes (768 cores), 32 nodes (1536 cores), and 64 nodes (3072 cores). Each block shows the performances of the eight distributed algorithms. In all cases, the vectorized `Sav` node engine has been employed.

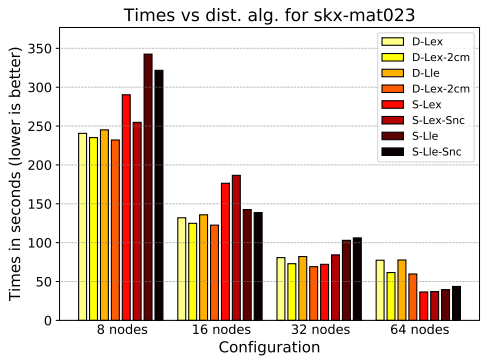

Fig. 5. Times (in seconds) for several distributed algorithms and nodes on `skx` for `mat023`.

As may be seen in Figure 5, when comparing the dynamic algorithms with the static algorithms, the dynamic algorithms clearly outperform the dynamic algorithms when employing 8, 16, and 32 nodes. When employing 64 nodes, the static ones are very competitive with respect to the dynamic ones. This can be due to the fact that the dynamic algorithm is based on a central process (the coordinator) that distributes the work and gathers the results. As the number of processes grow, this central coordinator may become the bottleneck of the application. On the other hand, static algorithms are not based on a central coordinator and can work independently, but the distribution of the work might not be so efficient.

When comparing the four dynamic algorithms, performances of the two algorithms that assigns two tasks at a time (`D-Lex-2cm` and `D-Lle-2cm`) are always better. This fact can support the previous hypothesis of the bottleneck in the dynamic algorithms. When comparing the four static algorithms, the performances of the new algorithms based on the left-lexicographical order and the snake cyclic (`S-Lex-Snc`, `S-Lle`, and `S-Lle-Snc`) are much better for 8, 16, and 32 nodes than the basic static algorithm (`S-Lex`).

4.6 Scalability

To measure the scalability of our implementations, Figure 6 shows the speedups obtained by several configurations to compute the minimum distance of both linear codes on ua. The prefix sizes and the numbers of threads per process employed in these experiments are the optimal values obtained in the above experiments. Recall that the speedup is the number of times that the parallel algorithm is as fast as the serial (one core) algorithm. Obviously, all the number of cores assessed in this experiment were multiple of 12 (the number of cores per node).

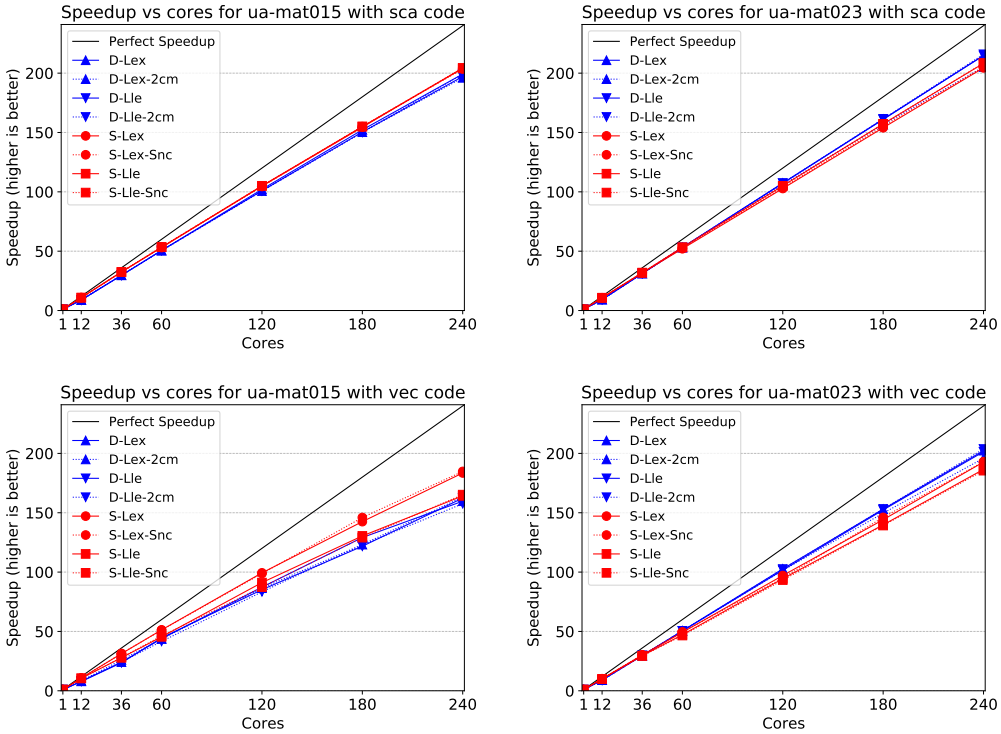

Fig. 6. Speedups of configurations with several number of nodes (and cores) for the two linear codes. on the ua multicomputer. The top row shows the results for the scalar node engine Sav; the bottom row shows the results for the vectorized Sav node engine.

As may be observed in Figure 6, the static algorithms are faster on the `mat015` linear code, whereas the dynamic algorithms are faster on the `mat023` linear code. As was commented, this might be related to the number of parallel tasks generated: the static algorithms require many tasks to balance the load, and the number of tasks greatly depend on the dimension k. Note that the speedups on the `mat015` linear code are smaller than those on the `mat023` linear code. The reason is that the computational cost of the `mat015` linear code is about one order of magnitude smaller than the computational cost of the `mat023` linear code. Note that for the `mat015` linear code the total time on 240 cores is about 44 seconds, which is very small in comparison with the total number of cores. The speedups for the scalar codes (top row) are slightly larger that the speedups of the vectorized codes (bottom row) since the vectorized codes are much more efficient on one core. Note that the speedups achieved are remarkable and can be up to about 200 when employing 240 cores.

Next, we measure the scalability of our implementations on the skx multicomputer to assess a larger number of cores. Figure 7 shows the speedups obtained by several configurations to compute the minimum distance on the skx multicomputer. This figure shows the results for the mat023 linear code (left side) since it is the most expensive one of both previous linear codes. Besides, it also shows the results for the mat020 linear code with parameters [235,51,64] (right side), since this requires twice the total time of the previous one. As many cores (several thousands) are going to be employed, more expensive linear codes must tested. The prefix sizes for the dynamic and the static algorithms employed in these experiments are 5 and 6, respectively, since they are optimal or very close to optimal. In both cases, the numbers of threads per process is two. Obviously, all the number of cores assessed in this experiment were multiple of 48 (the number of cores per node).

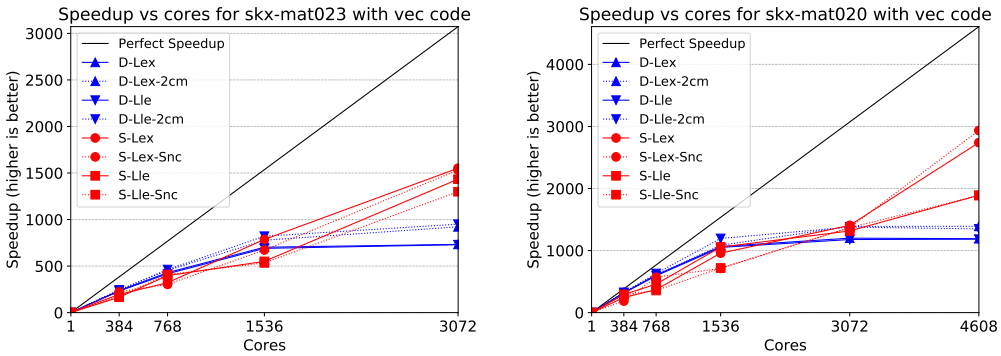

Fig. 7. Speedups of configurations with several number of nodes (and cores) for the mat023 linear code (left) and mat020, a linear code with parameters [235,51,64] (right) on skx. The second linear code requires about twice the cost of the first one.

As can be easily observed in Figure 7, the dynamic algorithms are usually faster when employing up to 1536 cores. However, when employing 3072 cores or more, the static algorithms clearly outperform the dynamic ones.

Note that the efficiency (speedup divided by the number of cores) is smaller on the skx multicomputer than on the ua one. The reason is that the number of cores employed on the skx is more than one order of magnitude higher whereas similar linear codes are employed. Moreover, the total run time when employing so many cores on the skx is very short. For instance, when employing 3072 cores, the total runtime of processing the mat023 linear code is about 34 seconds, which is a very short execution time. However, when employing more expensive linear codes such as mat020, speedups and efficiencies grow higher when comparing the same algorithms and same number of cores (such as 1536). Note that the mat020 linear code obtains higher speedups than the mat023 since its computational cost is twice higher. In any case, note that the speedups achieved are remarkable, and they grow as the computational cost grows.

4.7 Comparison with state-of-the-art software implementations

Now we compare the performances of the new distributed algorithms in the ua cluster with the performances of both commercial and public-domain software published by Hernando *et al.* [11]. In this paper the times were obtained in the cplex server, which is a computer based on AMD processors. It contained an AMD Opteron™ Processor 6128 (2.0 GHz), with 8 cores (though only 6 were used to let other users work). Its OS was GNU/Linux (Version 3.13.0-68-generic). Gcc compiler (version 4.8.4) was employed.

Although the computational power of the cores in the cplex server is not exactly the same as that of the cores in the ua cluster, the processors were released in similar dates: The processor in the cplex server was launched in the first quarter of 2010, whereas the processor in the nodes of the ua cluster was launched in the first quarter of 2009. Therefore, the processors are of similar generations. We could not assess MAGMA in the same machine, the ua cluster, because it is a commercial software and we do not have a license for it.

Two of the most-common implementations currently available were assessed:

- MAGMA [4]: It is a commercial software package focused on computations in algebra, algebraic geometry, algebraic combinatorics, etc. Version 2.22-3 was employed in those experiments. In the cplex server, vectorization could not be employed since MAGMA only implements this feature on modern processors with AVX support. MAGMA was assessed on one core as well as 6 cores since this software is parallelized.
- GUAVA [3, 10]: GAP (*Groups, Algorithms, Programming*) is a public-domain software environment for working on computational group theory and computational discrete algebra. It contains a package named GUAVA that can compute the minimum distance of linear codes. GUAVA Version 3.12 within GAP Version 4.7.8 was employed in those experiments. GUAVA does not implement any vectorization, and it only works on one core since the software is not parallelized.

In contrast, our implementations can use hardware vector instructions both on old processors (SSE) and modern processors (AVX), both from Intel and AMD. Furthermore, our implementations can employ any number of cores inside a node.

Table 2 compares the times required by the commercial software MAGMA, the times required by the public-domain software GUAVA, and the times required by the new algorithms for distributed-memory architectures to compute the minimum distance of both linear codes. The vectorized D-Lex algorithm with prefix size 4 and two threads per process was employed on 24 nodes (288 cores). In this table the new distributed algorithms exceedingly reduce the time by making many computers (24 computers with 12 cores each one) cooperate to compute the minimum distance. Thus, large processing times in commercial and public-domain software can be significantly reduced. For instance, computing the distance of the mat023 linear code with the public-domain software GUAVA required about 5 days and 7 hours, whereas employing our new software required a bit less than 5 minutes.

Code	MAGMA 1 core cplex	GUAVA 1 core cplex	MAGMA 6 cores cplex	New alg. 288 cores ua
mat015	53,052.9	40,804.3	9,562.8	38.9
mat023	503,984.2	456,413.2	85,341.1	282.6

Table 2. Times (in seconds) for the commercial software MAGMA, the public-domain software GUAVA, and the new distributed algorithms for computing the distance of both linear codes.

5 CONCLUSIONS

In this paper, we have introduced several new implementations of the Brouwer-Zimmermann algorithm for computing the minimum distance of a random linear code over $\mathbb{F}_2$ on distributed-memory architectures. Both state-of-the-art commercial and public-domain software can only be

employed on either unicore architectures or shared-memory architectures, which have a strong bottleneck in the number of cores/processors employed in the computation. In contrast, our family of implementations focuses on distributed-memory architectures, which are well known because of its scalability and being able to comprise hundreds or even thousands of cores. In the experimental results we show that our implementations are much faster, even up to several orders of magnitude, than current implementations widely used nowadays because of its capability of employing these scalable architectures. For a particular linear code the time to compute the minimum distance has dropped from about 11 hours and 2.5 hours (in public domain and commerical software, respectively) to half a minute with our code on a distributed-memory machine with 288 cores. For another particular linear code the time to compute the minimum distance has dropped from about 5 days and 1 day (in public domain and commerical software, respectively) to five minutes with our code on a distributed-memory machine with 288 cores.

Future work in this area will investigate the development of specific new algorithms and implementations for new architectures such GPGPUs (General-Purpose Graphic Processing Units). Another interesting line of future work is the adaptation of these fast implementations to other finite fields.

ACKNOWLEDGEMENTS

The authors would like to thank the University of Alicante for granting access to the ua cluster. They also want to thank Javier Navarrete for his assistance and support when working on this machine. The authors would also like to thank Robert A. van de Geijn from the University of Texas at Austin for granting access to the skx cluster.

Quintana-Ortí was supported by the Spanish Ministry of Science, Innovation and Universities under Grant RTI2018-098156-B-C54 co-financed by FEDER funds.

Hernando was supported by the Spanish Ministry of Science, Innovation and Universities under Grants PGC2018-096446-B-C21 and PGC2018-096446-B-C22, and by University Jaume I under Grant PB1-1B2018-10.

Igual was supported by Grants PID2021-126576NB-I00 and RTI2018-B-I00, funded by MCIN /AEI/ 10.13039/501100011033 and by "ERDF A way of making Europe", and the Spanish CM (S2018/TCS-4423). This work has been supported by the Madrid Government (Comunidad de Madrid, Spain) under the Multiannual Agreement with Complutense University in the line Program to Stimulate Research for Young Doctors in the context of the V PRICIT (Regional Programme of Research and Technological Innovation).

REFERENCES

[1] Christoph Adami, *Information theory in molecular biology*, Physics of Life Reviews **1** (2004), no. 1, 3 – 22.

[2] T. Ancheta, *Syndrome-source-coding and its universal generalization*, IEEE Transactions on Information Theory **22** (1976), no. 4, 432–436.

[3] R. Baart, T. Boothby, J. Cramwinckel, J. Fields, D. Joyner, R. Miller, E. Minkes, E.and Roijackers, L. Ruscio, and C. Tjhai, *Guava, a gap package for computing with error-correcting codes, version 3.12*, 2012.

[4] Wieb Bosma, John Cannon, and Catherine Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), no. 3-4, 235–265, Computational algebra and number theory (London, 1993). MR 1484478

[5] Iliya Bouyukliev and Valentin Bakoev, *A method for efficiently computing the number of codewords of fixed weights in linear codes*, Discrete Applied Mathematics **156** (2008), no. 15, 2986–3004.

[6] Stefka Bouyuklieva and Iliya Bouyukliev, *An extension of the brouwer–zimmermann algorithm for calculating the minimum weight of a linear code*, Mathematics **9** (2021), no. 19.

[7] A. Chowdhury and B. S. Rajan, *Quantum error correction via codes over gf(2)*, 2009 IEEE International Symposium on Information Theory, June 2009, pp. 789–793.

[8] O. Geil, S. Martin, R. Matsumoto, D. Ruano, and Y. Luo, *Relative generalized hamming weights of one-point algebraic geometric codes*, IEEE Transactions on Information Theory **60** (2014), no. 10, 5938–5949.

[9] Markus Grassl, *Searching for linear codes with large minimum distance*, Discovering mathematics with Magma, Algorithms Comput. Math., vol. 19, Springer, Berlin, 2006, pp. 287–313. MR 2278933
[10] The GAP Group, *Gap – groups, algorithms, and programming, version 4.7.8*, 2015.
[11] Fernando Hernando, Francisco D. Igual, and Gregorio Quintana-Ortí, *Algorithm 994: Fast implementations of the brouwer-zimmermann algorithm for the computation of the minimum distance of a random linear code*, ACM Trans. Math. Softw. **45** (2019), no. 2, 23:1–23:28.
[12] Donald L. Kreher and Douglas Robert Stinson, *Combinatorial algorithms: generation, enumeration, and search*, CRC Press, 1999.
[13] Shuo-Yen Robert Li, Raymond W. Yeung, and Ning Cai, *Linear network coding*, IEEE Trans. Inform. Theory **49** (2003), no. 2, 371–381. MR 1966785
[14] Petr Lisoněk and Layla Trummer, *Algorithms for the minimum weight of linear codes*, Advances in Mathematics of Communications **10** (2016), no. 1, 195–207.
[15] A. D. Liveris, Zixiang Xiong, and C. N. Georghiades, *Compression of binary sources with side information at the decoder using ldpc codes*, IEEE Communications Letters **6** (2002), no. 10, 440–442.
[16] F. J. MacWilliams and Neil J. A. Sloane, *The theory of error-correcting codes*, North-Holland Mathematical Library, no. 16, North-Holland Pub. Co., 1977.
[17] Elebeoba E. May, Mladen A. Vouk, Donald L. Bitzer, and David I. Rosnick, *A coding theory framework for genetic sequence analysis*, 2002.
[18] R. J. McEliece, *A Public-Key Cryptosystem Based On Algebraic Coding Theory*, Deep Space Network Progress Report **44** (1978), 114–116.
[19] H. Niederreiter, *Knapsack-type cryptosystems and algebraic coding theory*, Problems Control Inform. Theory/Problemy Upravlen. Teor. Inform. **15** (1986), no. 2, 159–166. MR 851173
[20] OpenMP Architecture Review Board, *OpenMP application program interface version 3.0*, May 2008.
[21] Adi Shamir, *How to share a secret*, Comm. ACM **22** (1979), no. 11, 612–613. MR 549252
[22] Claude Elwood Shannon, *A mathematical theory of communication*, The Bell System Technical Journal **27** (1948), no. 3, 379–423.
[23] Alexander Vardy, *The intractability of computing the minimum distance of a code*, IEEE Trans. Inform. Theory **43** (1997), no. 6, 1757–1766. MR 1481035
[24] K.H. Zimmermann, *Integral hecke modules, integral generalized reed-muller codes, and linear codes*, Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, Techn. Univ. Hamburg-Harburg, 1996.