algorithms

MDPI

Article

Tries-Based Parallel Solutions for Generating Perfect Crosswords Grids

Virginia Niculescu *,† and Robert Manuel Ştefănică †

Department of Computer Science, Faculty of Mathematics and Computer Science, "Babeş-Bolyai" University, 400084 Cluj-Napoca, Romania; robert.stefanica@stud.ubbcluj.ro

* Correspondence: virginia.niculescu@ubbcluj.ro

† These authors contributed equally to this work.

Abstract: A general crossword grid generation is considered an NP-complete problem and theoretically it could be a good candidate to be used by cryptography algorithms. In this article, we propose a new algorithm for generating perfect crosswords grids (with no black boxes) that relies on using tries data structures, which are very important for reducing the time for finding the solutions, and offers good opportunity for parallelisation, too. The algorithm uses a special tries representation and it is very efficient, but through parallelisation the performance is improved to a level that allows the solution to be obtained extremely fast. The experiments were conducted using a dictionary of almost 700,000 words, and the solutions were obtained using the parallelised version with an execution time in the order of minutes. We demonstrate here that finding a perfect crossword grid could be solved faster than has been estimated before, if we use tries as supporting data structures together with parallelisation. Still, if the size of the dictionary is increased by a lot (e.g., considering a set of dictionaries for different languages—not only for one), or through a generalisation to a 3D space or multidimensional spaces, then the problem still could be investigated for a possible usage in cryptography.

Keywords: crosswords; tries; parallel computation; cryptography

Citation: Niculescu, V.; Ştefănică, R.M. Tries-Based Parallel Solutions for Generating Perfect Crosswords Grids. *Algorithms* **2022**, *15*, 22. https://doi.org/10.3390/a15010022

Academic Editor: Charalampos Konstantopoulos

Received: 30 November 2021
Accepted: 6 January 2022
Published: 13 January 2022

1. Introduction

Crosswords are known as a form of intellectual game, and they are widespread throughout the world. The most common crosswords are generally presented on grids that do not necessarily have equal sides. Many of them have black cells because it is very difficult to create such grids otherwise.

In addition to the typical crosswords grids, there are some very special ones that we intend to refer to, such as:

Perfect Grids: crosswords grids with no black boxes, where the words are considered in two directions, from left to right and from up to down.

The computation used in the crossword domain follows different directions: generating new grids (without any clues), automatic clue generation, solving grids starting from clues, etc. Our focus is on the first one applied to perfect grids.

A general crossword grid generation is considered an NP-complete problem [1]. This is why we investigate if it could be a good candidate to be used by cryptography algorithms. As for prime number factorisation, the difficulty does not necessarily rely on the theoretical difficulty of problem solving, but on the time needed to practically solve the problem. The possible usage of the crosswords in cryptography was considered a long time ago [2].

There are different solutions—brute-force, backtracking based, heuristics, different constraint-satisfaction algorithms—but when we arrive at the time-complexity evaluation, we discover that it is very difficult to obtain concrete solutions for large dictionaries cases.

In order to reduce the execution time, we can try to reduce the searching space by using different approaches and heuristics, but we can also apply different parallelisation techniques.

Because of the problem's intrinsic nature, the solutions generally allow a high degree of parallelism. Still, it is possible that the reduced time complexity obtained only through parallelisation would not be enough, or at least if the degree of parallelism is highly bounded because of the concrete implementation architecture constraints, other optimisations should be investigated, too.

We intend to investigate the problem of generating all the perfect crosswords grids, for a given keyword, which is to be placed in the first column. This could be considered simpler than the finding a solution for a given grid with white and black cells, but it depends much more on the grid size and the given keyword. Additionally, without the black cells, the number of possible choices that should be investigated increases very much. On the other hand, it is possible to not obtain any solution, or to have many solutions for different grid sizes (the number of columns could be allowed to vary).

This special case was chosen in order to analyse it in the larger context of their possible usage in a public-key cryptography system [3]:

The public and private keys could be based on the words of a perfect grid:

- *Public key*
  - The word placed in the first column of the solution grid.
- *Private key*
  - The word in the ith ($0 \leq i$) row of the solution;
  - A preset combination of the solution rows.

If there are more solutions for the word to be placed in the first column, then the solution with the largest weight sum in terms of characters is considered.

In this paper, we propose a solution based on *tries* (digital trees or prefix trees), which are a special kind of tree used especially for storing strings, and where there is one node for every common prefix; the final strings are represented through the path from the root to the leaf nodes [4]. The usage of tries led to a very efficient way of choosing possible words that could be placed in the grid.

The solution offers good potential for parallelisation, too. The proposed parallel solution is a hybrid one based on multiprocessing (using MPI) and multithreading. The performance improvement through parallelisation is important and is proven by the speed-up seen in the concrete experiments.

Paper Organisation

Section 3 specifies the problem and its possible generalisation. This section also analyses some solutions to solve the "direct solutions" problem because they are derived from a classical direct approach. The first is the brute-force method, and the following ones are based on a backtracking approach for which different strategies for reducing the searching space are proposed. After this, our solution based on *trie* data structures is presented in Section 4. Several experiments were conducted and the results are presented in Section 5.

2. Related Work

Crossword puzzles have a long history and they have different particularities from one country to another. A general presentation of the history of crosswords, with a special focus on those based on the Romanian language, is given in [5].

There are two main approaches for generating crossword puzzles: the letter-by-letter instantiation approach and word-by-word instantiation approach. The letter-by-letter instantiation approach proceeds by repeatedly picking an empty cell from the grid and instantiating it from a given alphabet. The word-by-word instantiation approach is based on repeatedly placing a word into the grid.

The earliest work in the field was conducted by Mazlack [6], who used a letter-by-letter approach and treated the problem as a heuristic search problem. However, with limited processing power, only a small dictionary could be used and the solution was unable to leverage the power of the word-by-word instantiation approach.

Ginsberg et al. [7] focused on a word-by-word approach. The list of matching words for each slot is updated dynamically based on the slot positions already filled with letters. Meehan and Gray [8] compared a letter-by-letter approach against a word-by-word encoding and concluded that the latter is able to scale up to harder puzzles. Still, the algorithm that we propose here is a letter-by-letter algorithm that uses the verification of word prefixes, which proves to be extremely efficient.

Following these, the crossword generation was considered an example of a constraint satisfaction problem [9]. The variables consist of the empty grid positions to be filled up and the values are dictionary words and/or phrases that can be placed in these positions. In addition to the constraints induced by the shape of the grid, many others have to be set: the number of empty grid positions to be filled, the length of word patterns and the intersection between different "across" and "down" word patterns. The most important constraint when placing a word is that the word must belong to a given dictionary. This constraint means that a dictionary lookup is necessary, and if the constraint scope (the dictionary) is very large, the complexity of the problem remains very high. The solving of crossword puzzles is also applied in other interesting research areas of the AI community, such as cognitive computing [10].

Additionally, we may notice that solving a perfect crossword grid is connected to regex crossword problems, which are also known as NP-complete problems [11]. In a typical regular expression (regex) crossword puzzle, two non-empty lists $R_1, \ldots, R_n$ and $C_1, \ldots, C_m$ of regular expressions over some alphabet are given, and the goal is to fill in an $n \times m$ grid with letters from that alphabet such that the string formed by the ith row is in $L(R_i)$, and the string formed by the jth column is in $L(C_j)$, for all $1 \leq i \leq m$ and $1 \leq j \leq n$. It is known that determining whether a solution exists is an NP-complete problem. A regex crossword could be considered a more structured variant of finding a perfect crossword grid problem where, instead of having a dictionary, we have rules for creating words. The dictionary-based construction of such a perfect crossword grid imposes the need to search for words in the dictionary, and so choosing the correct method to choose possible words is essential in order to ensure a good performance.

3. Perfect Crosswords Grid Generation Problem

3.1. Problem Specification

The informal specification of the problem is as follows:

> Let D be a dictionary with N words and
> w a given word from D of length (number of characters) equal to n.
> It is necessary to compute all the perfect crossword grids with words from D
> that have on the first column the word w and number of columns equal to m
> —which is a number that could vary.
> If more than one solution is possible, then
> the solution that maximizes an optimisation Φ function
> (e.g., biggest characters weight sum) has to be chosen.

Because the specification of the problem requires the generation of all the possible grids, heuristics that try to increase the chances of arriving at a solution sooner are not considered, because all the possible solutions should be found. Rather, the specification leads to approaches that could eliminate, as quickly as possible, the choices that do not lead to any solution.

The reason for the analysis presented in this section is to emphasize some aspects of the involved time complexity of some direct classical solutions.

For the sake of simplicity, this analysis will consider the case of square grids.

Notations: we will denote by S_n the set formed of all the words of length n, and we will denote the cardinal of a set by $|.|$ (for example, the cardinal of set S is $|S|$).

3.2. Brute-Force Method

Even if it is well known that a brute-force solution is not efficient, we briefly describe it just to emphasize in what way a simple parallelisation is not enough.

A brute-force method considers the set S_n, where n is the length of the given word w. Let us imagine that this set has a cardinal equal to M_n. If the set S_n is not formed beforehand, we consider, as the first stage, the creation of the set S_n, and this comes with a time complexity equal to N, but also with a space complexity equal to $M_n \times n$.

The solution is created by setting the rest of the columns of the grid with all possible arrangements of words from S_n and then verified for each complete variant if, in the rows, words from the same set are obtained.

The time complexity of this solution is given by:

$$T^p s(N, M_n, n) = A_{M_n}^{n-1} \times (n \times N) = \frac{M_n!}{(M_n - n + 1)!} \times (n \times N)$$

where the factors $n \times N$ are due to the verification of rows.

The degree of parallelism of this variant is very high because, when the verification for each set of words is placed in the columns, the row verification is independent.

If we split the set S_n into p disjunctive sets, we may consider p processes that work in parallel, each having the responsibility of checking the cases when there is a word from their partition in the second column.

Considering an ideal case with shared memory (all the data are accessible from all processors), we arrive at a time complexity equal to

$$T^p(N, M, n, p) = A_{M_n}^{n-1} \times n \times N/p$$

The parallelisation could go further if $p > M_n$ and create new processes for all possible choices to be put in the third column, and this could continue.

Some questions could be raised:

- How big should p be in order to obtain a reasonable execution time?
- How big could p be in the condition of a given system architecture (shared or distributed memory)?

If we consider that for the Oxford English Dictionary (OED) it is stated that it includes 171,476 "active" words, and if, in addition to these words, we also allow derivative words, proper names, names of places, regionalisms, or technical words, etc., it is estimated that it is possible to achieve 750,000 words.

If we estimate that $M_n = 50{,}000$, we can use the same number of processors, but even if the required hardware is available, the resulting time complexity is still very high. On the other hand, it is certainly not possible to use 50,000 parallel processes in shared memory systems. So, in addition to the estimated time complexity $T^p(N, M.n, p)$, we have to add the cost for distributing the dictionary to all the processes (the row verification uses all the words). All these lead to the conclusion that this variant is not a valid one—it imposes too high a cost of computation.

3.3. Backtracking-Based Methods

A common way to reduce the search space in these kinds of problems is to use backtracking to reduce the verified combinations.

One way to limit the searching space is to see what limitations on the possible choices in the rows are produced when a new word is placed in the current column. If this produces an impossible choice in a row, it should be excluded.

In order to verify these situations, we may either manage a list of possible choices for each row, or use lists of possible prefixes of lengths equal to $1, 2, 3, ...n-1$. We will next analyse the solution based on lists of prefixes.

3.3.1. A Search Algorithm with Predefined List of Prefixes

Definition 1. *A group of l letters is an admissible l-prefix in the dictionary D if there is at least one word in D that starts with that group of letters.*

Let $w = (w_0, w_1, ..., w_{n-1})$ be the letters of the given word; the letters $w_0, w_1, ...w_{n-1}$ are 1-prefixes for the words in each row.

Going further, the solutions could consider:

- L_2—the set of all 2-prefixes;
- L_3—the set of all 3-prefixes;
- L_4—the set of all 4-prefixes;
  ...
- L_{n-1}—the set of all $n-1$-prefixes.

These lists of prefixes could be computed and stored in the first stage of the computation. These could be solved in a single traversal of the dictionary, but how these lists are represented could influence the final time complexity. This stage could be parallelized by applying a data decomposition of the dictionary.

We schematize and analyse a word-by-word backtracking solution that uses prefixes with a maximal length equal to l and stored into linear lists. We emphasize here only the solutions domain, the continuation conditions and the solution verification:

- The solutions are formed from the tuples $X_1, ..., X_{n-1}$, where X_j is a word from the set S_n to be put in the column j.
- The continuation conditions for the level k are:
  - If $k \leq l$ then all the obtained prefixes (on each row) should be admissible in the dictionary D (belongs to L_{k+1}).
    The prefix in row i is obtained in the following way:
    $x_{i,0}x_{i,1}x_{i,2}...x_{i,k}$, where $x_{i,j}$ is the letter i of the word X_j.
  - If $k > l$, then no other conditions are verified.
- Solution verification: verify that the words on each row belong to the dictionary D.

For a backtracking solution the time complexity is difficult to calculate rigorously, but we tried to make an estimation. We denote with M_i the average number of words from S_n, for which, when placed in the column $i+1$, the resulting $(i+1)$-prefixes are admissible; for each $k > l$, $M_k = M_n$. If we ignore the time complexity for the verification of the continuation conditions, we may approximate the number of possible choices that need to be verified as being $M_1 \times M_2 \times \cdots \times M_{n-1}$.

In order to verify that a word is admissible to be placed in the i-th column, we need $n \times l_i$ operations (l_i is the size of the list L_i), which verifies that the new created prefixes are admissible.

Parallelisation is possible based on the decomposition of the set of all words that could be placed in the second column (S_n with L_2 prefixes), and so the time complexity could be reduced by a factor equal to M_1.

3.3.2. Other Variations

Other variations could be considered, such as:

- Placing the words in columns until arriving at column l, and then changing the direction and placing the words in rows—depending on the already-computed prefixes;
- Placing the words alternatively in columns and rows;
- Variations of the previous alternatives.

Overall, the resulting time complexity is still very high. However, the final execution time drastically depends on how well organized the data structures are that help us to find the new admissible words, starting from some prefixes, and also how fast we may check if a prefix is admissible or not.

The prefix-based analysis inspired us to use tries to search for appropriate words. The next section describes this solution, which proves to be very efficient.

4. Tries-Based Solutions

A *trie* is an efficient information-re*trie*val, tree-type data structure, through which search complexities can be brought to the optimal limit. Trie data structures for computer searching were first described by René de la Briandais [12], and independently in 1960 by Edward Fredkin [13] who coined the term trie; the tries were originally created in order to find a compromise between space and time complexity. They are defined in "Dictionary of Algorithms and Data Structures" [14] as a tree for storing strings in which there is one node for every common prefix. By using tries, the searching time complexity could be very much improved with some penalty related to the storage requirements of tries.

A trie encodes a set of strings, represented by concatenating the characters of the edges along the path from the root node to each corresponding leaf. The common prefixes are collapsed such that each string prefix corresponds to a unique path.

An important point related to tries is that the number of child nodes of one particular node depends completely upon the total number of possible keys. Tries are often used to represent words in an alphabet, when each of the nodes represents a single letter of a word. The root node is empty and the children of the root represent the first letter of each possible prefix. For example, if we are representing the English alphabet, then the number of child nodes for one parent node is directly connected to the total number of possible letters. In the English alphabet, there are 26 letters, so the total number of possible child nodes for one node will be 26. An illustration for a subset of words is shown in Figure 1.

More formally, if we consider a set of n keys (characters) that are used to form a set of values (words) that form a dictionary, then a trie could be represented as a tree with an empty root, and each other node stores:

- An array of n pointers, one for each key;
- A bit indicating whether the value corresponding to the path between the root and the node is in the possible values set (the set of words from the dictionary).

The cost of looking up a value w of size $|w|$ is $O(|w|)$, if we consider that each pointer can be looked up in time $O(1)$.

A very important characteristic of the tries is the fact that the searching time is independent of the number of values represented in the trie [15].

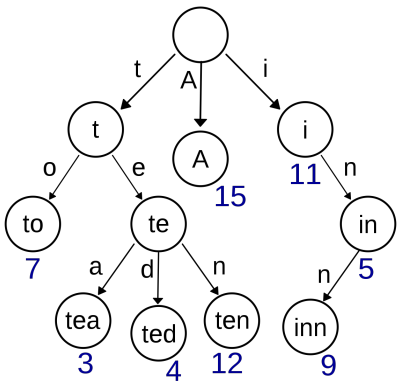

Figure 1. A trie for the values "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

4.1. The Trie Solution

The problem specification stipulates that the size of the grid should be equal to $n \times m$, where n is the size of the given word that should be placed in the first column, and m is the number of columns of the optimal solution. If the optimisation function Φ is defined to maximize the characters' weight sums, then it is very probable that an optimal solution will be obtained for big values of m, and so an approach could start for the maximal value of m, which decreases successively.

The proposed solution follows a letter-by-letter approach, and the design analysis considers a specified size $n \times m$ of the grid.

Data representation: The words of the dictionary are represented using a complex data structure that contains max_size tries; the number max_size is equal to the maximum size of the words from the dictionary (e.g., $max\_size = 16$ for our dictionary). We have words that contains 1, 2, ..., max_size letters. This way, we have words represented only as full paths—from root to leaves—in the tries.

The roots of the corresponding tries are stored in an array of lengths, max_size—$Tries[1..max\_size]$; so, $Tries[n]$ is the root of the trie that stores the words with lengths equal to n.

The decision of using max_size tries increases the space complexity, but facilitates a very good time complexity, since in order to fill a crossword grid, as specified in the problem specification, we need words of specific lengths. These tries could be created one time and stored, potentially as the first step of the algorithm. The time complexity of creating these tries is not very high since a single dictionary traversal is required; for each new word, the path corresponding to its letters is followed and new nodes (corresponding to the contained letters) are added when necessary. So, the time complexity is $O(N)$ for this operation.

For each trie, the first level contains an array of 26 roots for the words starting with each alphabet letter. We propose a special trie representation, in which each node of a trie contains:

- A letter—*letter*;
- An array with a length equal to 26—*array_link*—for the children nodes (one for each letter);
- A number—*code*—which is obtained from a binary representation with 26 digits that reflects the possible continuations:
  - 0 on the position i means that there is not a subtrie corresponding the ith letter;
  - 1 on the position i means that there is a subtrie corresponding the ith letter.

Example: if for one particular node there are subtries that continue only with the letters A and D, the *code* will be 37748736, which has the following binary representation: 10010000000000000000000000.

This binary code is very important in the fast verification of the possible letters that could be placed in a new position.

For the node that represents the root of such a tree, the attribute *letter* is empty.

The words are obtained from the leaf nodes and correspond to the letters on the path from root to them.

In order to fill the crossword grid, three matrices ($M1$, $M2$ and $M3$), which all have the same size as the grid ($n \times m$), are used:

- $M1$ memorizes pointers to the nodes from $Tries[m]$ that correspond to each letter of the words written horizontally (on rows)s;
- $M2$ memorizes pointers to the nodes from $Tries[n]$ that correspond to each letter of the words written vertically (in columns);
- $M3$ memorizes each position the index of the last letter put into the grid—$M3[i, j].index$, and the binary code of the possible letters that could be placed on that position—$M3[i, j].code$—this code is obtained through the intersection (and operation) of the binary codes stored in the corresponding pointed nodes of the first

two matrices. The type of the $M3$ elements is defined by a data structure with two fields—$(index, code)$.

In addition, a position that indicates the current position in the grid completion is used:

- *position*—a variable of type $pair = [0..n] \times [0..m]$ ($[i, j]$).

The algorithm: The algorithm is in essence a backtracking algorithm that fills the cells step by step in columns, starting with the first which is known. So, it is a letter-by-letter algorithm that relies on prefix verification.

The $M1$ and $M2$ matrices are first initialized with the pointers to the nodes that correspond to the letters of the given word that should appear on the first column. In $M2$, the nodes (from $Tries[n]$) follow a path that represents the given word, and in $M1$ we will have pointers to the nodes from $Tries[m]$ that correspond to the specified first letters in the rows.

The letter in a new position $[i, j]$ is chosen from the letters obtained by the intersection of the set of the possible letters that allow horizontal words (it creates possible prefixes in rows)—given from the code stored in the node $M1[i, j-1]$—and the set of possible letters that allow vertical words (creates possible prefixes in columns)—given from the code stored in the node $M2[i-1, j]$. The intersection is obtained by applying the *bit_and* operation (denoted by &&) to the codes $M1[i, j-1].code$ *and* $M2[i-1, j].code$, and it is stored in the $M3[i, j].code$. The letters from this intersection are considered in order and the index of the current chosen letter is stored into $M3[i, j].index$.

For the first letter in a column j the set of possible letters is given by $M1[0, j-1].code$; no intersection operation is needed.

If the result of the && operation between the codes of the nodes stored into $M1[i-1, j]$ and $M2[i, j-1]$ is zero (the intersection is empty), then a step back is executed and the letter in the position $[i-1, j-1], if\ i \geq 1$ is changed by taking the next possible letter in the corresponding intersection set. Additionally, when all the possible letters from the intersection were verified, and moving forward is not possible, a step back is executed again.

So, the BACK_STEP function is defined as:

---

procedure BACK_STEP( )

$$position = \begin{cases} (position.i - 1, position.j), & \text{if } position.i > 0 \\ (n - 1, position.j - 1), & \text{if } position.i = 0 \end{cases}$$

The function SET_LETTER has the responsibility of setting a new letter in the current position.

---

function SET_LETTER( )
  $M3[position].index =$ FIRST_INDEX$(M3[position].code)$
  if $(M3[position].index \neq 0)$ **then**
    $M1[position] = M1[\text{LEFT}(position)].array\_link[M3[position].index]$
    if $position.i > 0$ **then**
      $M2[position] = M2[\text{UP}(position)].array\_link[M3[position].index]$
    else
      ▷ the root node corresponding to $M3[position].index$ letter in **Tries**[**m**]
      $M2[position] =$ **Tries**[**m**]$.array\_link[M3[position].index]$
    return *true*
  else
    return *false*

The function FIRST_INDEX returns the position of the first bit equal to 1 in the binary representation of $M3[position].code$ and set it to 0; setting this bit to 0 is needed in order to emphasize that it has been verified. Additionally, in this way the next call of the function FIRST_INDEX will return the position of the next bit equal to 1.

There is always a left position since we have a word in the first column, and the UP function will be called only when $position.i > 0$.

function UP($position$)

$$up\_position = \begin{cases} (position.i - 1, position.j), & \text{if } position.i > 0 \\ (n - 1, position.j) - 1, & \text{if } position.i = 0 \end{cases}$$

return $up\_position$

function LEFT($position$)

$$left\_position = \begin{cases} (position.i, position.j - 1), & \text{if } position.i > 0 \wedge position.j > 0 \end{cases}$$

return $left\_position$

The procedure *MOVE_NEXT* moves the current position to the next cell in the grid, and sets the corresponding value of the *code* attribute in $M3$:

procedure MOVE_NEXT( )

$$position = \begin{cases} (position.i + 1, position.j), & \text{if } position.i < n - 1 \\ (0, position.j + 1), & \text{if } position.i = n - 1 \end{cases}$$

if position.i>0 **then**

$M3[position].code = M1[\text{LEFT}(position)].code$ && $M2[\text{UP}(position)].code$

else

$M3[position].code = M1[\text{LEFT}(position)].code$

Using these functions, we may define the scheme of the overall algorithm—Algorithm 1.

The algorithm was parameterized with the start position in order to allow its usage for the parallel implementation, for the sequential case $start\_pos = [0, 1]$.

Algorithm 1 Tries_Crosswords_Seq(Sol_List, Optimal_Sol, start_pos)

▷ **Sol_List**—the list of the resulted solution
▷ **Optimal_Sol**—store the optimal solution
@read the dictionary and create the tries—**Trie**[**n**] and **Trie**[**m**]
▷ matrices initialisation
@set the nodes from **Tries**[**m**] in **M1** corresponding to the given word (first column)
@set the nodes from **Tries**[**n**] in **M2** corresponding to the given word (first column)
$position = \text{UP}(\textbf{start\_pos})$
MOVE_NEXT()
repeat
  $letter\_found = \text{SET\_LETTER}\ (position)$
  if $letter\_found$ **then**
    if $(position = [n - 1, m - 1]$ **then** ▷ if solution
      @save the solution into the list of solutions—**Sol_List**
      if optimal_solution() **then**
        @save the solution as optimal—**Optimal_Sol**
    else
      MOVE_NEXT()
  else
    BACK_STEP()
until $M3[\textbf{start\_pos}].code = 0$

The most important characteristics and benefits of the algorithm are:

- The usage of tries to find the possible prefixes;
- The codes attached to each node of the tries (which reflect the possible continuation); using the codes in the nodes of the tries facilitates a very fast verification of the possible paths to solutions using the *and* operation on bits;
- The codes saved in the cells of the $M3$ matrix, as an intersection of the corresponding cells in $M1$ and $M2$ matrices.

The performance of this algorithm depends on the size of the grid, but most importantly on the number of words of a certain size.

4.2. Parallel Implementation

The algorithm has good potential for parallelisation, and we have developed a hybrid parallel implementation using an MPI (Message Passing Interface) and multithreading. We have chosen this hybrid parallelisation in order to allow using distributed memory and not only shared memory architectures.

Let P be the number of MPI processes and each of these uses a number of T threads (a thread pool of size T). Each process will create the necessary tries from the dictionary file. The space complexity of these structures is not very high, and so, it is worth carrying out this duplication.

The first parallelisation that we may identify relies on the observation that there are several possibilities to set the cell value in the position $(0,1)$; these possible letters are given by the bit code of the node stored in $M1[0,0]$. Still, in order to allow a parallelisation control independent of the given word that should be placed in the first column, the entire set of alphabet letters is distributed throughout the processes. An exemplification of this is given in Figure 2. In this way, the responsibility of one process is to find the solutions that have one of the letters assigned to said process in the position $[0,1)$.

Process No.	0	1	2	3
Letter to be placed on position [0,1]	A,E,I,M,Q,U,Y	B,F,J,N,R,V,Z	C,G,K,O,S,W	D,H,L,P,T,X

Figure 2. The distribution of the letters to be placed in position $(0,1)$ for the case of 4 processes.

Since the proposed solution is a hybrid one (multiprocessing combined with multithreading), each process will use a thread pool of size T able to execute the associated tasks that lead to the solution finding.

For the parallelisation at the threads level, the tasks are defined based on the construction of a list of pairs of letters $(L1, L2)$, where $L1$ is a potential letter to be placed in the position $(0,1)$ and $L2$ in the position $(1,1)$ (Procedure PAIR_TASK). In this way, the maximal parallelisation degree increases to an adequate value—26^2. Each task will define and use its own matrices $M1, M2$ and $M3$.

More concretely, if we consider the case of four processes with the letter distribution described in Figure 2, the first process will create 7×26 tasks that correspond to all the pairs that are formed by the Cartesian product between the following two sets: $\{A, E, I, M, Q, U, Y\}$ and $\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$.

The program executed by each MPI process is described by Algorithm 2.

Algorithm 2 Tries_Crosswords_Parallel → MPI program

Load_dictionary— ▷ each process read the dictionary and create the tries
Create_thread_pool— ▷ each process creates a thread-pool with T threads
Sol_List— ▷ create an empty list where solutions will be placed
Optimal_Solution— ▷ set a variable where the optimal solution will be placed
Letter_distribution— ▷ balanced distribution of the set A over the P processes
▷ (A contains all 26 letters of the alphabet)
▷ each process i will have a subset A_i of A ($A_i \in A$)
▷ for each pair of $A_i \times A$ a task is created and submitted to the thread-pool
for each pair $(L1, L2)$ of the Cartesian product $A_i \times A$ **do**
@submit **PAIR_TASK** $(L1, L2, Sol\_List, Optimal\_Solution, [1,1]))$ to the thread pool
@wait for all tasks to finalize
@aggregate all the solutions and the optimal solution in the process with $ID = 0$

The final aggregation of all found solutions and also the optimal solution, which it is obtained through a *reduce*-type operation, is carried out in the first MPI process ($ID = 0$). This means that after all the tasks are executed, each process sends to the process with a rank equal to 0 the value of the optimal function obtained for the solutions it found. The process 0 computes the global optimal value and sends it back to all the other processes. The processes that have optimal solutions save their optimal solutions.

---

procedure PAIR_TASK()($L1, L2, Sol\_List, Optimal\_Sol, start\_position$)
  if $L1$ eligible to be placed on $M3[0,1]$ **then**
    @set the values $M1[0,1], M2[0,1], M3[0,1]$ to correspond to letter $L1$
  else
    EXIT
  if $L2$ eligible to be placed on $M3[1,1]$ **then**
    @set the values $M1[1,1], M2[1,1], M3[1,1]$ to correspond to letter $L2$
  else
    EXIT
  @call **Tries_Crosswords_Seq**($Sol\_List, Optimal\_Sol, start\_position$)

Analysis

The decision to distribute the letters between the processes was based on the fact that we intended to define a parallel algorithm for which the degree of parallelism could be controlled independently on the given word to be placed in the first column. The value of $M1[0,0].code$ gives, more precisely, the letters that could be placed in the position $M2[0,1]$, but these are value dependent, and so not appropriate for the MPI process definition.

Even if, theoretically, the algorithm allows us to define 26 MPI processes, it is more efficient from the cost point of view to define fewer processes (to decrease the probability of having processes that do not have effective tasks to execute). A good average would be to assign 3–4 letters per process, and also to use a *cyclic distribution* for assigning letters to processes.

The presented solution leads to a multiprocessing parallelisation degree equal to 26, and the hybrid degree of parallelism (through multiprocessing and multiprogramming) is 26×26. The degree of paralelism could be improved if we consider pairs of letters distributed to the processes instead of simple letters. Using this variant, each process (from p processes) will receive a list that contains $676/p$ of pairs of two letters. In this case, a task created by a process will be defined by three letters—the first two are given from a pair distributed to the process and they are used for setting the positions $(0,1)$ and $(1,1)$, and the third is one from the entire alphabet and is supposed to be placed in the position $(2,1)$. This would allow a degree of parallelism bounded by 676×26. This idea could be generalized to tuples of k letters, and the obtained degree of parallelism would be bounded by 26^{k+1}. So, the number of MPI processes could be increased by defining a more general algorithm that distributes tuples of letters and not just single letters to the processes.

The reason for this generalisation is the possible need to engage many more processes in the computation, which is very plausible when using big clusters.

5. Analysis and Experiments

In order to evaluate the performance of the proposed solutions, several experiments have been conducted, for both sequential and parallel solutions.

For the experiments, a Romanian dictionary with 610767 words was used. The dictionary does not use diacritics.

The concrete specification of the initial problem is defined through the following function used for searching the optimal solution:

- if more than one perfect grid is found, then the one that maximizes the following formula is chosen:

$$V = \sum_{i}^{n} V(w_i),$$

where w_i is a word in a row, and

$$V(w_i) = \sum_{j=1}^{m} code_{ASCII}(c_i^j),$$

with c_i^j being the j letter of the word w_i.

The experiments were conducted on an HPC cluster—Koty (http://hpc.cs.ubbcluj.ro, accessed on 20 November 2021). A node has the following characteristics: two Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz, each with 10 cores, and 128 GB RAM memory per node. All the cluster nodes are connected using a fast network of 56 Gb/s (Infiniband Mellanox FDR switch SX6512 with 216 ports, 1:1 subscription rate).

For the parallel implementation, the tests were conducted using $P = 8$ MPI processes each using $T = 32$ threads, and each process was executed on a different node of the cluster.

Seven experiments have been conducted for seven different given words, and the resulting grids obtained through these experiments are given in Appendix A.

The execution times were as follows:

- **Experiment 1**
  The given word "ABACA".
  The sequential execution time $T_1^s = 6.1$ s.
  The sequential execution time $T_1^p = 0.9$ s.

- **Experiment 2**
  The given word "XEROFAG".
  The execution time $T_2^s = 9.1$ s.
  The execution time $T_2^p = 3.5$ s.

- **Experiment 3**
  The given word "ABERANT".
  The execution time $T_3^s = 1199$ s.
  The execution time $T_3^p = 95.1$ s.

- **Experiment 4**
  The given word "INSECABIL".
  The execution time $T_4^s = 648.4$ s.
  The execution time $T_4^p = 350.4$ s.

- **Experiment 5**
  The given word "ABATORIZA".
  The execution time $T_5^s = 4191s = 69.85$ min.
  The execution time $T_5^p = 308.5s$.

- **Experiment 6**
  The given word "ABATORIZAT".
  The execution time $T_6^s = 4834s = 80.56$ min.
  The execution time $T_6^p = 427.6$ s.

- **Experiment 7**
  The given word "ZACAMANT".
  The execution time $T_7^s = 2131s = 35.51$ min.
  The execution time $T_7^p = 383.9$ s.

The execution times of the parallel and sequential executions for each experiment are depicted in Figure 3, and the resulting speed-up obtained through parallelisation is shown in Figure 4.

Remark 1. *It can be seen that for the Experiment 2—word "XEROFAG"—the solution was found very fast because the first letter X restrains the searching space; there are only a few words that start with X.*

Experiments 5 and 6 had long sequential execution times since in these cases the searching space is quite large—there are many words starting with the letter A. They also emphasize the need for parallelisation. The parallel implementation manages to obtain the solution in much less time.

There is a correspondence between the amount of sequential computation and the level of speed-up through parallelisation: when the sequential computation was high the speed-up obtained through parallelisation was also high.

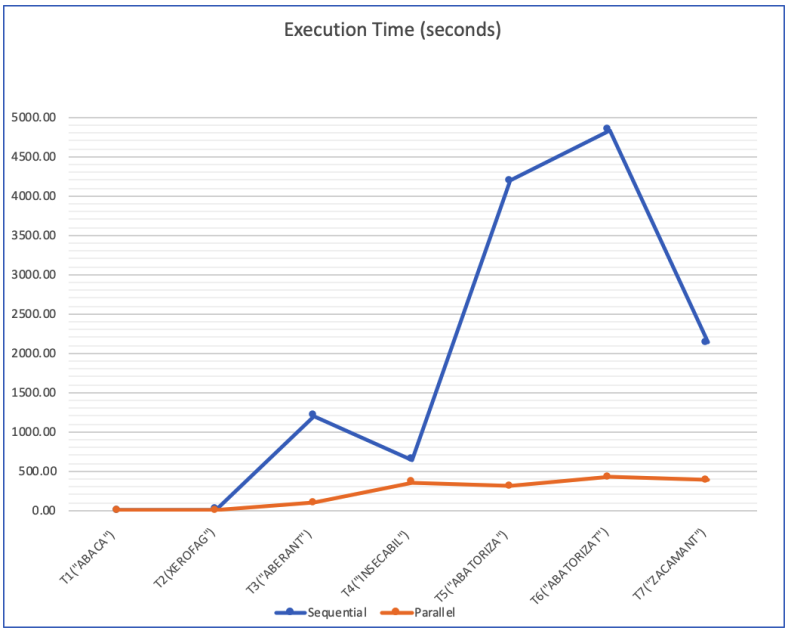

Figure 3. The execution time of parallel and sequential executions for each experiment (time expressed in seconds).

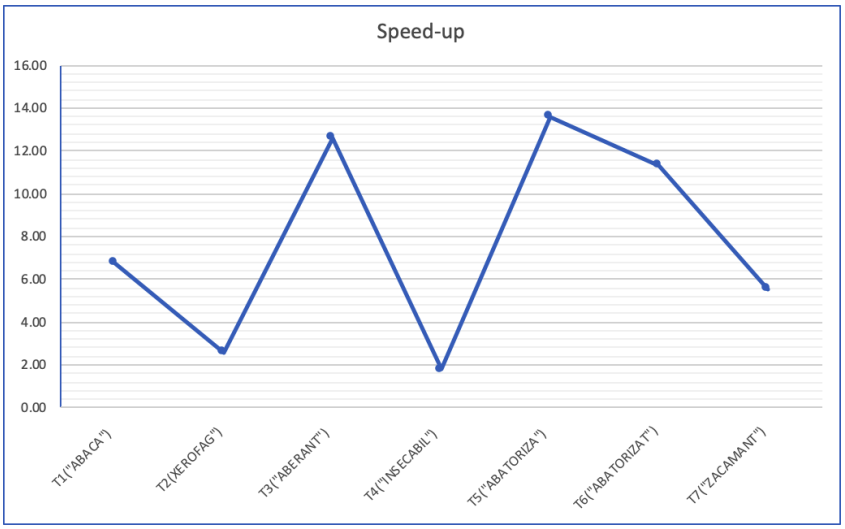

Figure 4. The speed-up obtained through parallelisation for each experiment.

6. Conclusions and Further Work

The presented computation solution based on tries dramatically reduces the time complexity of the solution for determining perfect crossword grids that start with a given keyword placed in the first column. With this solution, the usefulness of *tries* is enforced, proving once more to be an exceptional conceptual and practical tool. This method has unique advantages due to the new proposed representation of the tries that use binary codes attached to each node (which reflect the possible continuation); these codes facilitate a very fast verification of the possible paths to follow.

The solution offers a good parallelisation opportunity, and our experiments proved that very good speed-ups could be achieved through this parallelisation. The parallel solution could be easily adapted such that the maximum degree of parallelism can be increased; this depends on the size of the tuple of letters that each one of the processes considers to be fixed on the first cells that should be filled. This is also a very good premise for achieving a good scalability.

The initial analysis of the proposed problem emphasized a very high level of complexity that qualifies the problem for use in cryptography systems. Without a "smart" solution, such as the proposed one that uses tries, the complexity of the direct approaches for computing a perfect crossword grid would have been enough in order to qualify the problem for such systems.

We demonstrated here that finding a perfect crossword grid could be solved faster than it has been estimated before if we use tries to help data structures.Still, if the size of the dictionary is increased by a lot (e.g., considering a set of dictionaries for different languages—not only for one), also allowing the public key to be placed in any column in the grid, the execution time may increase, such that the problem is still eligible for use in cryptography. In addition, the problem could be generalized to a 3D space or even to bigger multidimensional spaces—even if they are not so intuitive.

In addition, crossword puzzles provide an interesting test bed for the evaluation of constraint satisfaction algorithms [16]. Constraint satisfaction problems (CSP) are among the most difficult problems for a computer to solve, but also there are some methods that make solving them relatively fast. Reversely, a good solving method for crosswords could provide good insights for generalisation that could be applied to CSP problems.

Author Contributions: Investigation, R.M.Ş. and V.N.; conceptualization, V.N. and R.M.Ş.; methodology, V.N. and R.M.Ş.; software, R.M.Ş.; validation, R.M.Ş. and V.N.; formal analysis, V.N.; writing—original draft preparation, V.N.; writing—review and editing, V.N.; visualisation, V.N.; supervision, V.N.; project administration, V.N.; funding acquisition, V.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by SC Robert Bosch SRL through the Partnership Agreement to support the 'High Performance Computing and Big Data Analytics' Master Program within Department of Computer Science, Faculty of Mathematics and Computer Science, "Babeş-Bolyai" University.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Experiments Results

The following grids were resulted in the specified experiments.

Experiment A1
The given word "ABACA"

A	S	P	R	U
B	U	T	O	N
A	S	U	Z	I
C	U	S	U	T
A	R	I	L	I

Experiment A2
The given word "XEROFAG"

X	E	R	O	F	A	G
E	T	U	V	E	Z	I
R	U	T	I	S	O	R
O	V	I	L	I	T	A
F	E	S	I	L	O	R
A	Z	O	T	O	S	I
G	I	R	A	R	I	I

Experiment A3
The given word "ABERANT"

A	T	E	S	T	A	T
B	U	T	U	R	I	I
E	T	A	T	I	S	T
R	U	T	I	S	O	R
A	R	I	M	A	R	E
N	A	S	I	T	U	L
T	U	T	I	I	L	E

Experiment A4
The given word "INSECABIL"

I	N	A	U	R	A	T	A
N	E	G	R	E	S	I	T
S	C	I	A	T	I	C	I
E	R	O	T	O	M	A	N
C	O	T	U	R	I	L	E
A	M	A	R	I	L	O	R
B	A	S	I	C	A	S	I
I	N	E	L	U	S	U	L
L	A	M	E	L	E	L	E

Experiment A5
The given word "ABATORIZA"

A	M	U	T	E	S	C
B	U	N	I	C	U	L
A	G	I	T	A	T	A
T	I	T	I	R	I	I
O	T	A	R	I	T	I
R	O	T	I	S	O	R
I	R	I	T	A	R	I
Z	U	L	U	S	U	L
A	L	E	L	E	L	E

Experiment A6
The given word "ABATORIZAT"

A	N	I	M	A	T
B	E	L	I	T	E
A	P	A	T	O	S
T	O	T	I	M	I
O	M	I	Z	I	T
R	E	V	A	Z	U
I	N	I	S	O	R
Z	I	L	E	R	I
A	T	O	S	U	L
T	E	R	I	L	E

Experiment A7
The given word "ZACAMANT"

Z	A	C	A	M	A	N	T
A	P	U	L	I	L	O	R
C	U	S	U	T	U	R	I
A	L	U	M	I	N	O	S
M	I	T	I	T	I	C	A
A	L	U	N	I	L	O	R
N	O	R	O	C	O	S	I
T	R	I	S	A	R	I	I

References

1. Garey, M.R.; Johnson, D.S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*; W. H. Freeman Co.: New York, NY, USA, 1990.
2. Fleming, V. Mystery of the D-day crosswords, Part 1. *Daily Record (Little Rock)*, 2008. Retrieved: 7 June 2010.
3. Stallings, W. *Cryptography and Network Security: Principles and Practice*, 7th ed.; Pearson Education: London, UK, 2017.
4. Brass, P. *Advanced Data Structures*; Cambridge University Press: Cambridge, UK, 2008.
5. Cioban, V.; Niculescu, V.; Prejmerean, V. *Crosswords Generator*; Zilele Acdemice Clujene. 2019; pp. 1–8.
6. Mazlack, L.J. Computer Construction of Crossword Puzzles Using Precedence Relationships. *Artif. Intell.* **1976**, *7*, 1–19. [CrossRef]
7. Ginsberg, M.L.; Frank, M.; Halpin, M.P.; Torrance, M.C. Search Lessons Learned from Crossword Puzzles. In Proceedings of the Eighth National Conference on Artificial Intelligence, Boston, MA, USA, 29 July–3 August 1990; pp. 210–215.
8. Meehan, G.; Gray, P. Constructing Crossword Grids: Use of Heuristics vs. Constraints. In Proceedings of the Expert Systems 97: Research and Development in Expert Systems XIV, SGES, Cambridge, UK, 12 December 1997; pp. 159–174.
9. Anbulagan; Botea, A. Crossword Puzzles as a Constraint Problem. In Proceedings of the Principles and Practice of Constraint Programming, Sydney, Australia, 14–18 September 2008; pp. 550–554.
10. Manzini, T.; Ellis, S.; Hendler, J.A. A Play on Words: Using Cognitive Computing as a Basis for AI Solvers in Word Puzzles. *J. Artif. Gen. Intell.* **2015**, *6*, 111–129. [CrossRef]

11. Fenner, S.A. The complexity of some regex crossword problems. *arXiv* **2014**, arXiv:1411.5437.
12. De La Briandais, R. File Searching Using Variable Length Keys. In Proceedings of the Western Joint Computer Conference, Association for Computing Machinery, IRE-AIEE-ACM '59 (Western), New York, NY, USA, 3–5 March 1959; pp. 295–298. [CrossRef]
13. Fredkin, E. Trie Memory. *Commun. ACM* **1960**, *3*, 490–499. [CrossRef]
14. Black, P.E. Dictionary of Algorithms and Data Structures. Available online: https://xlinux.nist.gov/dads/ (accessed on 30 June 2021).
15. Knuth, D.E. *The Art of Computer Programming*; Addison-Wesley: Reading, MA, USA, 1973.
16. Connor, J.; Duchi, J.; Bruce, I. *Crossword Puzzles and Constraint Satisfaction*; Technical Report; Stanford University: Stanford, CA, USA, 2005.