A parallel algorithm for the verification of Covering Arrays

Himer Avila-George[1], Jose Torres-Jimenez[2], Vicente Hernández[1] and Nelson Rangel-Valdez[2]
[1]Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Valencia, Spain
[2]Laboratorio de Tecnologías de Información, CINVESTAV-Tamaulipas, Ciudad Victoria, Tamaulipas, Mexico

Abstract— *Covering Arrays (CAs) are combinatorial objects that, with a small number of cases, cover a certain level of interaction of a set of parameters. CAs have found application in a variety of fields where interactions among factors need to be identified; some of these fields are biology, agriculture, medicine, and software and hardware testing. In particular, a covering array is an $N \times k$ matrix over an alphabet v s.t. each $N \times k$ subset contains at least one time each combination from $\{0, 1, ..., v-1\}^t$, given a positive integer value t. The process of ensuring that a CA contains each of the v^t combinations is called verification of CA. When CAs have many variables or their strength is greater than 3, its verification is computationally very expensive. In this paper we present an algorithm for CA verification and its implementation details in sequential and parallel computing.*

Keywords: Covering Arrays, Parallel Computing Algorithms, Software Testing

1. Introduction

The experimental design is a key piece in the software development given that it allows to identify fails in the software before it begins to operate. A good strategy to test a software involves the generation of the whole set of cases that participate in its operation. However, testing all the possible cases of a program requires a great amount of time, which can be inadmissible even for small programs [1].

An alternative strategy to test a software is the use of Covering Arrays (CAs). A Covering array (CA) is a combinatorial object that, with a small number of cases, covers a certain level of interaction of a set of parameters. The meaning of level of interaction relates any subset of t parameters of a matrix to the set of the v^t different combinations derived from v different values. The confidence level of the testing using combinatorial objects as CA increases with the interaction level involved [2].

Covering arrays(CAs) have been object of study and application in different research areas. Cawse [3] used CAs in the material design, Hedayat *et al.* [4] used them in medicine and agriculture; in biology and industrial processes have also been used by Shasha *et al.* [5] and Pahdke [6]. CAs have been used in hardware testing [7] but significantly the area with the major application of these objects is in software testing [8], [9]. Next, Definition 1.1 is a formal definition of CA.

Definition 1.1 (Covering Array): Let N, t, k, v be positive integers with $t \leq k$. A covering array, denoted by $CA(N; t, k, v)$, of alphabet v, strength t, is an array $\mathcal{M}$ of size $N \times k$, where each element $m_{i,j}$ takes values from the set $S = \{0, 1, 2, ..., v-1\}$ and each subset of $\mathcal{M}$ of size $N \times t$ contains all the possible combinations derived from $\{0, 1, ..., v-1\}^t$ symbols. In the rest of this document a subset of size $N \times t$ will be known as a t-tuple and the initial t-tuple will be $\{1, 2, ..., t\}$.

To illustrate the CA approach applied to the design of software testing, consider the Web-based system example shown in Table 1, the example involves four parameters each with three possible values. A full experimental design ($t = 4$) should cover $3^4 = 81$ possibilities, however, if the interaction is relaxed to $t = 2$ (pair-wise), then the number of possible combinations is reduced to 9 test cases.

Table 1: Parameters of Web-based system example.

	Browser	**OS**	**DBMS**	**Connections**
0	Firefox	Windows 7	MySQL	ISDN
1	Chromium	Ubuntu 10.10	PostgreSQL	ADSL
2	Netscape	Red Hat 5	MaxDB	Cable

Figure 1 shows the CA corresponding to $CA(9; 2, 4, 3)$; given that its strength and alphabet are $t = 2$ and $v = 3$, respectively, the combinations that must appear at least once in each subset of size $N \times 2$ are $\{0, 0\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 0\}$, $\{1, 1\}$, $\{1, 2\}$, $\{2, 0\}$, $\{2, 1\}$, $\{2, 2\}$.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 2 & 2 & 2 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

Fig. 1: A combinatorial design, $CA(9; 2, 4, 3)$.

Finally, to make the mapping between the CA and the Web-based system, every possible value of each parameter in Table 1 is labeled by the row number. Table 2 shows the corresponding pair-wise test suite; each of its nine experiments is analogous to one row of the CA shown in Figure 1.

Table 2: Test-suite covering all 2-way interactions, $CA(9; 2, 4, 3)$.

Experiments				
1	Firefox	Windows 7	MySQL	ISDN
2	Firefox	Ubuntu 10.10	PostgreSQL	ADSL
3	Firefox	Red Hat 5	MaxDB	Cable
4	Chromium	Windows 7	PostgreSQL	Cable
5	Chromium	Ubuntu 10.10	MaxDB	ISDN
6	Chromium	Red Hat 5	MySQL	ADSL
7	Netscape	Windows 7	MaxDB	ADSL
8	Netscape	Ubuntu 10.10	MySQL	Cable
9	Netscape	Red Hat 5	PostgreSQL	ISDN

When a CA contains the minimum possible number of rows, it is optimal and its size is called the *Covering Array Number* (CAN). The CAN is defined according to

$$CAN(t, k, v) = \min\{N : \exists\ CA(N; t, k, v)\}.$$

The trivial mathematical *lower bound* for a covering array is $v^t \leq CAN(t, k, v)$, however, this number is rarely achieved. Therefore determining achievable lower bounds is one of the main research lines for CAs.

The construction of $CAN(2, k, 2)$ can be efficiently done according with [10]; the same is possible for $CA(2, k, v)$ when the cardinality of the alphabet is $v = p^n$, where p is a prime number and n a positive integer value [11]. However, in the general case determining the *covering array number* is known to be a hard combinatorial problem [12], [13]. For the values of t and v that no efficient algorithm is known, we use approximated algorithms to construct them. Some of these approximated strategies must verify that the matrix they are building is a CA. If the matrix is of size $N \times k$ and the interaction is t, there are $\binom{k}{t}$ different combinations which implies a cost of $O(N \times \binom{k}{t})$ for the verification (when the matrix has $N \geq v^t$ rows, otherwise it will never be a CA and its verification is pointless). For small values of t and v the verification of CAs is overcame through the use of sequential approaches; however, when we try to construct CAs of moderate values of t, v and k, the time spent by those approaches is impractical. Then, the necessity for parallel algorithms to construct and verify CAs appears.

In this paper we propose an algorithm to verify that a given matrix is a CA. The algorithm is implemented following the sequential and parallel strategies. The performance of each paradigm is experimentally compared with the purpose of showing their particular limitations. The remaining of the paper is organized as follows. Section 2 reviews related work dedicated to the construction and verification of CAs. Section 3 formally presents the problem of verifying a CA and describes the approaches proposed in this paper to verify CAs. Section 4 presents the methodology followed to experimentally compare the approaches; this section ends with some discussion derived from the results obtained from the experiment. Section 5 presents the conclusions derived from the research presented in this paper.

2. Relevant related work

Because of the importance of the construction of (near) optimal CAs, much research has been carried out in developing effective methods for construct them. There are several reported methods for constructing these combinatorial models. Among them are: (a) algebraic methods, (b) recursive methods, (c) greedy methods, and d) meta-heuristics. In this section we describe the relevant related work to the construction of CAs.

Algebraic approaches construct CAs using predefined rules. Most algebraic approaches compute CAs directly by a mathematical function [14], [15]. The algorithms that use recursive constructions generate a number of small CAs and with them build CAs of greater size. These algorithms have been used in Augmented Annealing [16] and CTS [17] to construct CAs. The majority of commerce and open source test data generating tools use greedy algorithms for covering array construction (AETG [18], TCG [19], DDA [20], ACTS [12], [21], and IRPS [22]). Some meta-heuristics that have been used to solve the CA problem are: Simulated Annealing (SA) [23], [24], Tabu Search (TS) [25], [26] and Memetic Algorithms (MA) [27].

All the algorithms already presented that construct covering arrays must contain a certificate that the resulting matrix is indeed a CA. For general matrices, the certificate is given by the verification process of a CA; this process requires to test that the sub-matrices derived from the $\binom{k}{t}$ different t-tuples contain all the combinations of symbols found in $\{0,\ 1,\ ...,\ v-1\}^t$. Hence, the cost of verifying a CA is a time consuming task that is highly combinatorial and rapidly becomes impractical to be solved by sequential approaches (even for moderate values of t, k, v of the CA). It is there where lies the necessity of a parallel strategy that makes faster the verification process in larger CAs.

The next section presents sequential and parallel strategies proposed in this paper to verify a given matrix as a CA.

3. How to verify CAs

A matrix $\mathcal{M}$ of size $N \times k$ is a $CA(N; t, k, v)$ iff every t-tuple contains the set of combination of symbols described by $\{0, 1, ..., v-1\}^t$. We propose a strategy that uses two data structures called P and J, and two injections between the sets of t-tuples and combinations of symbols, and the set of integer numbers, to verify that $\mathcal{M}$ is a CA.

Let $\mathcal{C} = \{c_1, c_2, ..., c_{\binom{k}{t}}\}$ be the set of the different t-tuples. A t-tuple $c_i = \{c_{i,1}, c_{i,2}, ..., c_{i,t}\}$ is formed by t numbers, each number $c_{i,1}$ denotes a column of matrix $\mathcal{M}$. The set $\mathcal{C}$ can be managed using an injective function $f(c_i) : \mathcal{C} \rightarrow \mathcal{I}$ between $\mathcal{C}$ and the integer numbers, this function is defined in Equation 1.

$$f(c_i) = \sum_{j=1}^{t} \binom{c_{i,j} - 1}{i + 1} \tag{1}$$

Now, let $\mathcal{W} = \{w_1, w_2, ..., w_{v^t}\}$ be the set of the different combination of symbols, where $w_i \in \{0, 1, ..., v-1\}^t$. The injective function $g(w_i) : \mathcal{W} \rightarrow \mathcal{I}$ is defined as done in Equation 2. The function $g(w_i)$ is equivalent to the transformation of a v-ary number to the decimal system.

$$g(w_i) = \sum_{j=1}^{t} w_{i,j} \cdot v^{t-i} \quad (2)$$

The use of the injections represents an efficient method to manipulate the information that will be stored in the data structures P and J used in the verification process of $\mathcal{M}$ as a CA. The matrix P is of size $\binom{k}{t} \times v^t$ and it counts the number of times that each combination appears in $\mathcal{M}$ in the different t-tuples. Each row of P represents a different t-tuple, while each column contains a different combination of symbols. The management of the cells $p_{i,j} \in P$ is done through the functions $f(c_i)$ and $g(w_j)$; while $f(c_i)$ retrieves the row related with the t-tuple c_i, the function $g(w_i)$ returns the column that corresponds to the combination of symbols w_i. The vector J is of size t and it helps in the enumeration of all the t-tuples $c_i \in \mathcal{C}$.

Table 3 shows an example of the use of the function $g(w_j)$ for the Covering Array $CA(9; 2, 4, 3)$ (shown in Figure 1). Column 1 shows the different combination of symbols. Column 2 contains the operation from which the equivalence is derived. Column 3 presents the integer number associated with that combination.

Table 3: Mapping of the set $\mathcal{W}$ to the set of integers using the function $g(w_j)$ in $CA(9; 2, 4, 3)$ shown in Figure 1.

$\mathcal{W}$	$g(w_i)$	$\mathcal{I}$
w_1 ={0,0}	$0 \cdot 3^1 + 0 \cdot 3^0$	0
w_2 ={0,1}	$0 \cdot 3^1 + 1 \cdot 3^0$	1
w_3 ={0,2}	$0 \cdot 3^1 + 2 \cdot 3^0$	2
w_4 ={1,0}	$1 \cdot 3^1 + 0 \cdot 3^0$	3
w_5 ={1,1}	$1 \cdot 3^1 + 1 \cdot 3^0$	4
w_6 ={1,2}	$1 \cdot 3^1 + 2 \cdot 3^0$	5
w_7 ={2,0}	$2 \cdot 3^1 + 0 \cdot 3^0$	6
w_8 ={2,1}	$2 \cdot 3^1 + 1 \cdot 3^0$	7
w_9 ={2,2}	$2 \cdot 3^1 + 2 \cdot 3^0$	8

The matrix P is initialized to zero. The construction of matrix P is direct from the definitions of $f(c_i)$ and $g(w_j)$; it counts the number of times that a combination of symbols $w_j \in \mathcal{W}$ appears in each subset of columns corresponding to a t-tuple c_i, and increases the value of the cell $p_{f(c_i),g(w_j)} \in P$ in that number.

Table 4b shows the use of injective function $f(c_i)$. Table 4b presents the matrix P of $CA(9; 2, 4, 3)$. The different combination of symbols $w_j \in \mathcal{W}$ are in the first rows. The number appearing in each cell referenced by a pair (c_i, w_j) is the number of times that combination w_j appears in the set of columns c_i of the matrix $CA(9; 2, 4, 3)$.

In summary, to determine if a matrix $\mathcal{M}$ is or not a CA the number of different combination of symbols per t-tuple is counted using the matrix P. The matrix $\mathcal{M}$ will be a CA iff the matrix P contains no zero in it.

Several approaches can be followed to implement this strategy to verify a CA. The traditional one is the sequential algorithm (one instruction at a time), other approach is parallel computing. These strategies use the data structures described in this section and are discussed in the following subsections.

3.1 Sequential Algorithm (SA) to verify CAs

The Sequential Algorithm (SA) takes as input a matrix $\mathcal{M}$ and the parameters N, k, v, t that describe the CA that $\mathcal{M}$ can be. Also, the algorithm requires the sets $\mathcal{C}$ and $\mathcal{W}$ and, without lost of generality, the values $\mathcal{K}_l$ and $\mathcal{K}_u$ that represent the first and last t-tuple to be verified (which for SA are $\mathcal{K}_l = 1, \mathcal{K}_u = \binom{k}{t}$). SA outputs the total number of missing combinations in the matrix $\mathcal{M}$ to be a CA. The algorithm first counts for each different t-tuple c_i the times that a combination $w_j \in \mathcal{W}$ is found in the columns of $\mathcal{M}$ corresponding to c_i. After that, SA calculates the missing combinations $w_j \in \mathcal{W}$ in c_i. Finally, the algorithm transforms c_i into c_{i+1}, i.e. it determines the next t-tuple to be evaluated.

The pseudo-code for SA is presented in Algorithm 1. For each different t-tuple (lines 5 to 21) the algorithm performs the following actions: counts the expected number of times a combination w_j appears in the set of columns indicated by J (lines 6 to 11, where the combination w_j is the one appearing in $\mathcal{M}_{n,J}$, i.e. in row n and t-tuple J); then, the counter $covered$ is increased in the number of different combinations with a number of repetitions greater than zero (lines 10 and 11). After that, the algorithm calculates the number of missing combinations (line 12). The last step of each iteration of the algorithm is the calculation of the next t-tuple to be analyzed (lines 13 to 21). The algorithm ends when all the t-tuples have been analyzed (line 5).

3.2 Parallel Approach (PA) to verify CAs

The parallel strategy to verify CAs is simple. It involves a block distribution model of the set of t-tuples. The set $\mathcal{C}$ is divided into n blocks, where n is the processors number; the size of block $\mathcal{B}$ is equal to $\lceil \frac{C}{n} \rceil$. The block distribution model maintains the simplicity in the code; this model allows the assignment of each block to a different core such that SA can be applied to verify the blocks.

To make the distribution of work, it is necessary to calculate the initial t-tuple f for each core according to its ID (denoted by $rank$), where $F = rank \cdot \mathcal{B}$. Therefore it is necessary a method to convert the scalar F to the equivalent t-tuple $c_i \in \mathcal{C}$. The sequential generation of each t-tuple c_i previous to c_F can be a time consuming task. There is where lies the main contribution of our parallel approach; its simplicity is combined with a clever strategy for computing the initial t-tuple of each block.

Table 4: Example of the matrix P resulting from $CA(9; 2, 4, 3)$ presented in Figure 1.

(a) Applying $f(c_i)$.

c_i index	c_i t-tuple	$f(c_i)$
c_1	$\{1,2\}$	0
c_2	$\{1,3\}$	1
c_3	$\{1,4\}$	3
c_4	$\{2,3\}$	2
c_5	$\{2,4\}$	4
c_6	$\{3,4\}$	5

(b) Matrix P.

	$g(w_j)$								
$f(c_i)$	{0,0}	{0,1}	{0,2}	{1,0}	{1,1}	{1,2}	{2,0}	{2,1}	{2,2}
0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1

Algorithm 1: SA, sequential algorithm to verify a CA.

```
1  t_wise(M_{N,k}, v, t, K_l, K_u)
2  begin
3      Miss ← 0, iMax ← t − 1, P ← 0
       /* Get initial t-tuple                                   */
4      J ← getInitialTuple(k, t, c_{K_l})
5      while J_t ≤ k and f(J) ≤ K_u do
6          covered ← 0
7          for n ← 1 to N do
8              P_{f(J),g(M_{n,J})} ← P_{f(J),g(M_{n,J})} + 1
9          for j ← 1 to v^t do
10             if P_{f(J),j} > 0 then
11                 covered ← covered + 1
12         Miss ← Miss + v^t − covered
           /* Calculates the next t-tuple                       */
13         J_t ← J_t + 1
14         if J_t > k and iMax > 0 then
15             J_{iMax} ← J_{iMax} + 1
16             for i ← iMax + 1 to t do
17                 J_i = J_{i−1} + 1
18             if J_{iMax} > k − t + iMax then
19                 iMax ← iMax − 1
20             else
21                 iMax = t
22     return Miss
```

We propose the Algorithm 2 as a method that generates c_F, according to a lexicographical, without generating its previous t-tuples c_i, where $i < F$. To explain the purpose of the Algorithm 2, lets consider the $CA(9; 2, 4, 3)$ shown in Figure 1. This CA has as set $\mathcal{C}$ the elements found in column 1 of Table 4a. The algorithm `getInitialTuple` with input $k = 4,\ t = 2,\ F = 3$ must return $J = \{1, 4\}$, i.e. the values of the t-tuple c_3. The Algorithm 2 is optimized to find the vector $J = \{J_1, J_2, ..., J_t\}$ that corresponds to F. The value J_i is calculated according to

$$J_i = \min_{j \geq 1} \left\{ \Delta_i \leq \sum_{l=J_{i-1}+1}^{j} \binom{k-l}{t-i} \right\}$$

where

$$\Delta_i = F - \sum_{m=1}^{i-1} \sum_{l=J_{m-1}+1}^{J_m-1} \binom{k-l}{t-m}$$

and

$$J_0 = 0.$$

Algorithm 2: Get initial t-tuple to PA.

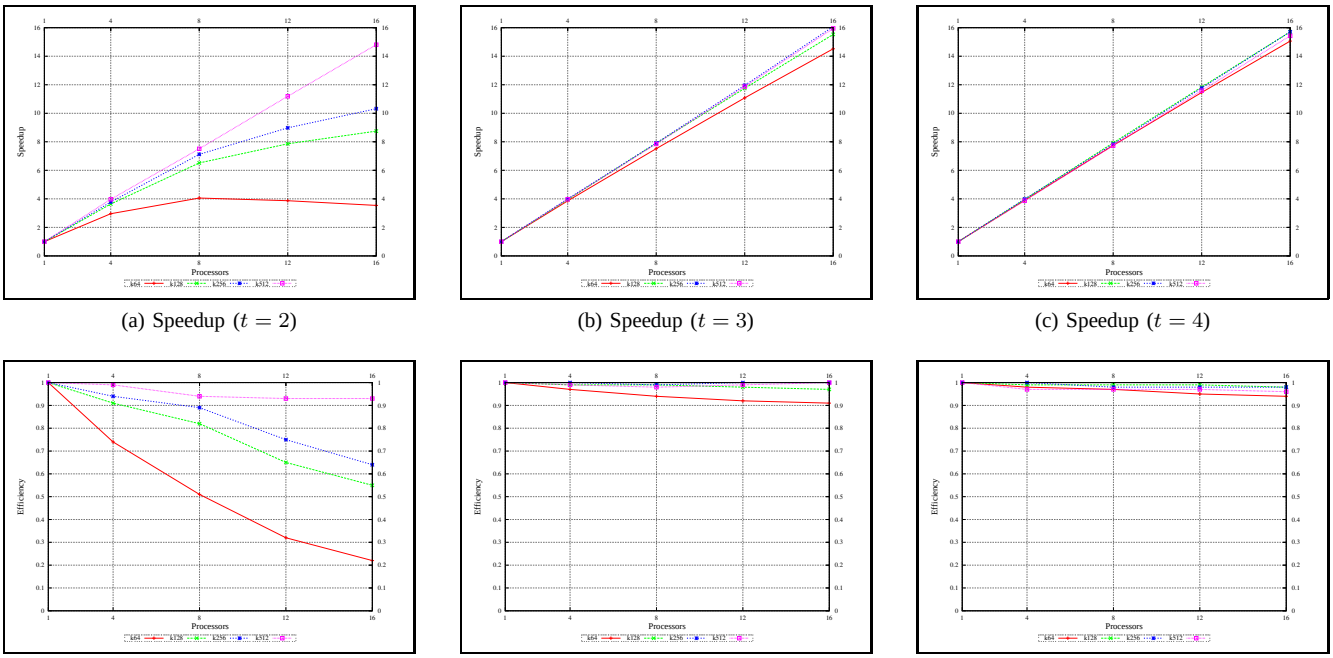
```
1  getInitialTuple(k, t, c_i)
   Output: Initial t-tuple each core
2  begin
3      Θ ← i, iK ← 1, iT ← 1
4      kint ← C(k−iK, t−iT)
5      for i ← 1 to t do
6          while Θ > kint do
7              Θ ← Θ − kint
8              iK ← iK + 1
9              kint ← C(k−iK, t−iT)
10         J_i ← iK
11         iK ← iK + 1
12         iT ← iT + 1
13         kint ← C(k−iK, t−iT)
14     return J
```

In conclusion, the Algorithm 2 only requires the computation of $O(t \times k)$ binomials to compute the n initial t-tuples of the PA. This represents a great improvement in contrast with the naive approach that would require the generation of all the $\binom{k}{t}$ t-tuples, as done in the SA. The next section presents the experimental results of using SA and PA approaches proposed in this paper applied to the verification of CAs.

4. Experimental design to compare the different approaches to verify a CA

This section presents an experimental design and results derived from testing the approaches described in the last section. The purpose of the experiment is to show the performance of each approach and the limitations that one can find on them. The two approaches were implemented in *C* language. SA was compiled with *gcc -O3* and PA was compiled with *mpicc -O3*. SA and PA were run in the Tirant supercomputer belonging to the Spanish Supercomputing Network [28].

In order to show the performance of the verification of CAs algorithm, two experiments were developed. The first experiment uses a benchmark that contains 16 binary CAs, the second experiment uses a benchmark that contains 16 ternary CAs. Each benchmark was created using the following parameters: $t = \{2, 3, 4\}$, $k = \{64,\ 128,\ 256,\ 512\}$. We developed a specific implementation of Galois algorithm [11], enhanced by the use of logarithmic tables [29].



(a) Speedup ($t = 2$) (b) Speedup ($t = 3$) (c) Speedup ($t = 4$)

(d) Efficiency ($t = 2$) (e) Efficiency ($t = 3$) (f) Efficiency ($t = 4$)

Fig. 2: Performance analysis when $t = \{2, 3, 4\}$ $k = \{64,\ 128,\ 256,\ 512\}$ and $v = 2$.

We use this implementation to generate the 32 CAs. For simplicity, the only multi-core approach considered during the comparison was PA having a maximum number of cores of 16 and the maximum time was 48 hours.

The comparison was based on the speedup ($\mathcal{S}$) and the efficiency ($\mathcal{E}$) on a parallel architecture against a single core case. We define speedup as the factor by which the execution time for the application changes, $\mathcal{S}(n) = \frac{\tau(s)}{\tau(n)}$, where τ_s is execution time on a single processor and τ_n is execution time on a multiprocessor. Efficiency gives fraction of time that processors are being used on computation, ($\mathcal{E} = \frac{\mathcal{S}(n)}{n}$).

4.1 Binary benchmark

This subsection presents a comparison between the SA and PA approaches using a binary benchmark with the purpose of studying the variation in the performance between a single core approach and an approach with more than one core. Table 5 shows the time (in seconds) spent by SA and PA to verify the experimental CAs. The first four columns describe the test case, the rest of the columns show the time spent by SA and PA, respectively, to verify each instance.

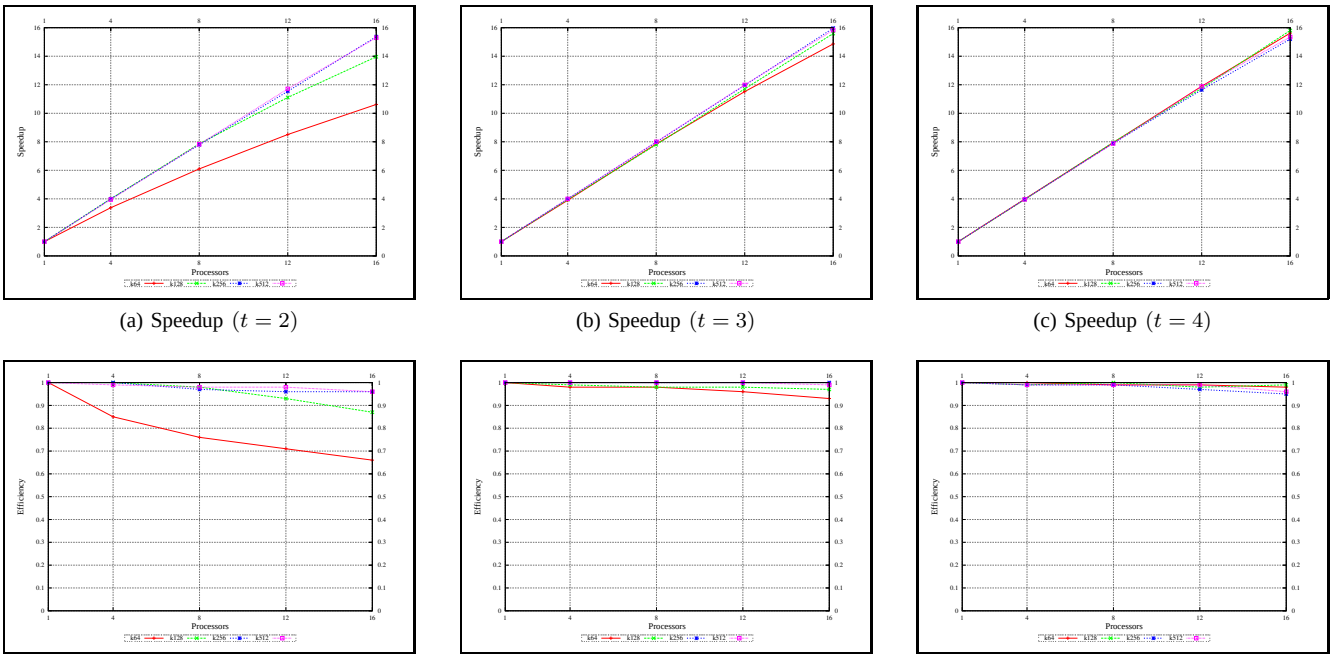
Figures 2a and 2d shows the performance achieved for the instances of strength $t = 2$; it is possible to observe that for almost all the cases the speedup is almost linear up to 4 cores, and after that it drops considerably. Figures 2b and 2e shows the performance for the instances where $t = 3$. A linear speedup up to 8 cores, and after that it drops slightly. Figures 2c and 2f shows the performance for the instances where $t = 4$. The performance shows a better behavior in the speedup and the efficiency than the ones observed for the cases where $t = \{2, 3\}$, i.e. the speedup is almost linear and the efficiency is almost 100% in all the instances showed.

Table 5: Comparison of the performance of SA and PA. This table shows the time (in seconds) spent by each approach when verifying the chosen benchmark.

N	t	k	v	SA	PA (16 cores)
64	2	64	2	0.000382	0.000108
64	3	64	2	0.018693	0.001287
64	4	64	2	0.662790	0.044016
64	5	64	2	14.490000	0.911106
128	2	128	2	0.001592	0.000182
128	3	128	2	0.159930	0.010300
128	4	128	2	12.760802	0.812173
128	5	128	2	709.999744	47.536213
256	2	256	2	0.005984	0.000580
256	3	256	2	1.276129	0.079378
256	4	256	2	206.547729	13.151242
256	5	256	2	27240.528490	1753.620271
512	2	512	2	0.024898	0.001681
512	3	512	2	10.938686	0.685710
512	4	512	2	3568.107378	231.240136
512	5	512	2	898675.740682	69418.805718

4.2 Ternary benchmark

This subsection presents a comparison between the SA and PA approaches using a benchmark that contains 16 ternary CAs. The Table 6 shows the time (in seconds) spent by SA and PA to verify the experimental CAs. The first four columns describe the test case, the rest of the columns show



(a) Speedup $(t = 2)$ (b) Speedup $(t = 3)$ (c) Speedup $(t = 4)$

(d) Efficiency $(t = 2)$ (e) Efficiency $(t = 3)$ (f) Efficiency $(t = 4)$

Fig. 3: Performance analysis when $t = \{2, 3, 4\}$, $k = \{64,\ 128,\ 256,\ 512\}$ and $v = 3$.

the time spent by SA and PA, respectively, to verify each instance. A cell in the table with the expression **n.a.** denotes that this experiment does not end at the maximum available time (48 hours).

Table 6: Comparison of the performance of SA and PA. This table shows the time (in seconds) spent by each approach when verifying the chosen benchmark.

N	t	k	v	SA	PA (16 cores)
64	2	64	3	0.000430	0.000255
64	3	64	3	0.104173	0.009200
64	4	64	3	2.233600	0.143011
64	5	64	3	35.201100	2.217308
128	2	128	3	0.007564	0.000590
128	3	128	3	1.109700	0.071200
128	4	128	3	68.076500	4.308390
128	5	128	3	2134.342164	135.955665
256	2	256	3	0.026556	0.001728
256	3	256	3	9.335492	0.584722
256	4	256	3	1895.894700	124.724641
256	5	256	3	132781.989216	9024.639374
512	2	512	3	0.112200	0.007339
512	3	512	3	80.039744	5.056195
512	4	512	3	41956.982600	2733.268051
512	5	512	3	n.a.	n.a.

To illustrate the scalability of our parallel algorithm we use the cases when $t = \{2, 3, 4\}$, $k = \{64, 128, 256, 512\}$ and $processors = \{4, 8, 12, 16\}$. The Figure 3 shows that the acceleration is almost linear and the efficiency is about 95% when $v = 3$. Therefore we can conclude that our parallel algorithm scales very well.

5. Conclusions

This paper presents sequential and parallel approaches to solve the problem of verifying CAs. The approaches were used to verify a benchmark formed by 32 matrices which were proposed to be verified as CAs. The CAs to be verified have alphabets $v = \{2, 3\}$, strengths $t = \{2, 3, 4, 5\}$ and number of columns $k = \{64, 128, 256, 512\}$.

The results of the two experiments presented showed SA as a good option when $t \leq 3$; however, this approach consumed a lot of time when $t \geq 4$ and $k > 128$ which made it impractical to use.

We proposed an optimized algorithm to calculate the initial t-tuple, its computational cost is $O(t \times k)$. PA approach worked well in the cases when $t > 2$. In general, the results showed that the partition strategy proposed for PA was efficient for the CA verification problem due to it required an almost null communication; this fact is observed in the linear speedup seen experimentally. Therefore we can conclude that our parallel algorithm scales very well.

Finally, given that the results gave birth to new CAs, another contribution of this paper is a new set of CAs which have been concentrated in the repository available at `http://www.tamps.cinvestav.mx/~jtj/CA.php`. In this web site it is possible to download the CAs, with better upper bounds than the ones given at the NIST Web site [30]. The current best upper bounds can be found in [31].

Acknowledgments

The authors thankfully acknowledge the computer resources and assistance provided by Spanish Supercomputing Network (TIRANT-UV). This research work was partially funded by the following projects: CONACyT 58554, Calculo de Covering Arrays; 51623 Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas.

References

[1] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03)*. IEEE Computer Society, 2003, pp. 394–405.

[2] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[3] J. N. Cawse, *Experimental Design for Combinatorial and High Throughput Materials Development*, J. N. Cawse, Ed. John Wiley & Sons, Inc., 2003.

[4] A. Hedayat, N. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*, A. Hedayat, N. Sloane, and J. Stufken, Eds. Springer-Verlag, 1999.

[5] D. E. Shasha, A. Y. Kouranov, L. V. Lejay, M. F. Chou, and G. M. Coruzzi, "Using combinatorial design to study regulation by multiple input signals: A tool for parsimony in the post-genomics era," *Plant Physiol*, vol. 127, no. 4, pp. 1590–1594, 2001.

[6] M. S. Phadke, *Quality Engineering Using Robust Design*, M. S. Phadke, Ed. Prentice Hall PTR, 1995.

[7] K. K. Vadde and V. R. Syrotiuk, "Factor interaction on service delivery in mobile ad hoc networks," *IEEE J Sel Area Comm*, vol. 22, no. 7, pp. 1335 – 1346, 2004.

[8] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generation and code coverage," in *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*. West, 1998, pp. 503–513.

[9] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE T Software Eng*, vol. 32, no. 1, pp. 20–34, 2006.

[10] D. J. Kleitman and J. Spencer, "Families of k-independent sets," *Discrete Math*, vol. 6, no. 3, pp. 255–262, 1973.

[11] K. A. Bush, "Orthogonal arrays of index unity," *Ann Math Stat*, vol. 23, no. 3, pp. 426–434, 1952.

[12] Y. Lei and K. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, ser. HASE '98. IEEE Computer Society, 1998, pp. 254–261.

[13] C. J. Colbourn, "Combinatorial aspects of covering arrays," *Le Matematiche*, vol. 58, pp. 121–167, 2004.

[14] M. Chateauneuf and D. L. Kreher, "On the state of strength-three covering arrays," *J Comb Des*, vol. 10, no. 4, pp. 217–238, 2002.

[15] K. Meagher and B. Stevens, "Group construction of covering arrays," *Journal of Combinatorial Designs*, vol. 13, no. 1, pp. 70–77, 2005.

[16] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Constructing strength three covering arrays with augmented annealing," *Discrete Math*, vol. 308, no. 13, pp. 2709–2722, 2008.

[17] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1-3, pp. 149 – 156, 2004.

[18] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 1996.

[19] Y. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *Aerospace Conference Proceedings*. IEEE, 2000, vol. 1, pp. 431–437.

[20] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Softw Test Verif Rel*, vol. 17, no. 3, pp. 159–182, 2007.

[21] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for t-way software testing," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '07)*. IEEE, 2007, pp. 549–556.

[22] M. Younis, K. Zamli, and N. Mat Isa, "IRPS: An Efficient Test Data Generation Strategy for Pairwise Testing," in *Knowledge-Based Intelligent Information and Engineering Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5177, pp. 493–500.

[23] J. Torres-Jimenez and E. Rodriguez-Tello, "Simulated annealing for constructing binary covering arrays of variable strength," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[24] J. Martinez-Pena, J. Torres-Jimenez, N. Rangel-Valdez, and H. Avila-George, "A heuristic approach for constructing ternary covering arrays using trinomial coefficients," in *IBERAMIA*, ser. Lecture Notes in Computer Science, Á. F. K. Morales and G. R. Simari, Eds., vol. 6433. Springer, 2010, pp. 572–581.

[25] T. Berling and P. Runeson, "Efficient evaluation of multifactor dependent system performance using fractional factorial design," *IEEE T Software Eng*, vol. 29, no. 9, pp. 769–781, 2003.

[26] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of variable strength using a tabu search approach," in *COCOA (1)*, ser. Lecture Notes in Computer Science, W. Wu and O. Daescu, Eds., vol. 6508. Springer, 2010, pp. 51–64.

[27] E. Rodriguez-Tello and J. Torres-Jimenez, "Memetic algorithms for constructing binary covering arrays of strength three," in *Artificial Evolution 2009*. Springer, 2009, vol. 5975, pp. 86–97.

[28] Tirant supercomputer, "Spanish Supercomputing Network," URL: http://www.uv.es/siuv/cas/zcalculo/res/descripcion.wiki. Accessed 20 April 2011.

[29] J. Torres-Jimenez, N. Rangel-Valdez, A. L. Gonzalez-Hernandez, and H. Avila-George, "Construction of logarithm tables for Galois Fields," *International Journal of Mathematical Education in Science and Technology*, vol. 42, no. 1, pp. 91–102, 2010.

[30] National Institute of Standards and Technology, "NIST covering array tables," URL: http://math.nist.gov/coveringarrays/. Accessed 20 April 2011.

[31] C. J. Colbourn, "Covering array tables for t=2,3,4,5,6," URL: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html. Accessed 20 April 2011.