Parallel simulated annealing for the covering arrays construction problem

Himer Avila-George\textsuperscript{1}, Jose Torres-Jimenez\textsuperscript{2} and Vicente Hernández\textsuperscript{1}

\textsuperscript{1}Instituto de Instrumentación para Imagen Molecular (I3M). Centro mixto CSIC - Universitat Politècnica de València - CIEMAT, camino de Vera s/n, 46022 Valencia, Spain.
\textsuperscript{2}Information Technology Laboratory, CINVESTAV-TAMAUJLPAS, Km. 5.5 Carretera Victoria-Soto La Marina, 87130 Victoria Tamps., Mexico.

Abstract—A covering array (CA) is a combinatorial structure specified as a matrix of $N$ rows and $k$ columns over an alphabet on $v$ symbols such that for each set of $t$ columns every $t$-tuple of symbols is covered at least once. Given the values of $t$, $k$, and $v$, the optimal covering array construction problem (CAC) consists in constructing a $CA(N; t, k, v)$ with the minimum possible value of $N$. There are several reported methods to attend the CAC problem, among them are: direct methods, recursive methods, greedy methods, and metaheuristics methods. In this paper, three parallel approaches for simulated annealing, i.e. the independent, semi-independent and cooperative searches are applied to the CAC problem. The empirical evidence supported by statistical analysis indicate that the cooperative approach offers the best execution times and the same upper bounds than the independent and semi-independent approaches. Extensive experimentation was carried out, using 96 well-known benchmark instances, for assessing its performance with respect to the best-known bounds reported previously. The results show that cooperative approach attains 78 new bounds and equals the solutions for other 6 instances.

Keywords: Covering Array, Simulated Annealing, Parallel Computing

1. Introduction

A covering array, denoted by $CA(N; t, k, v)$ is a matrix $\mathcal{M}$ of size $N \times k$ which takes values from the set of symbols $\{0, 1, 2, \ldots, v-1\}$ (called the alphabet), and every submatrix of size $N \times t$ contains each tuple of symbols of size $t$ (or $t$-tuple), at least once. The value $N$ is the number of rows of $\mathcal{M}$, $k$ is the number of parameters, where each parameter can take $v$ values and the interaction degree between parameters is described by the strength $t$. Each combination of $t$ columns must cover all the $v^t$ $t$-tuples. Given that there are $\binom{k}{t}$ sets of $t$ columns in $\mathcal{M}$, the total number of $t$-tuples in $\mathcal{M}$ must be $v^t(\binom{k}{t})$. When a $t$-tuple is missing in a specific set of $t$ columns we will refer to it as a missing $t$-wise combination. Then, $\mathcal{M}$ is a covering array if the number of missing $t$-wise combinations is zero.

When a matrix has the minimum possible value of $N$ to be a $CA(N; t, k, v)$, the value $N$ is known as the Covering Array Number. The Covering Array Number is formally defined as (1):

$$CAN(t, k, v) = \min\{N : \exists CA(N; t, k, v)\}. \quad (1)$$

Given the values of $t$, $k$, and $v$, the optimal covering array construction problem (CAC) consists in constructing a $CA(N; t, k, v)$ such that the value of $N$ is minimized.

A major application of covering arrays (CAs) arises in software interaction testing, where a covering array can be used to represent an interaction test suite as follows. In a software test we have $k$ components or factors. Each of these has $v$ values or levels. A test suite is an $N \times k$ array where each row is a test case. Each column represents a component and a value in the column is the particular configuration. By mapping a software test problem to a covering array of strength $t$ we can guarantee that we have tested, at least once, all $t$-way combinations of component values. Thus, software testing costs can be substantially reduced by minimizing the number of test cases $N$ in the covering array. Please observe that software interaction testing is a black-box testing technique, thus it exhibits weaknesses that should be addressed by employing white-box testing techniques. For a detailed example of the use of covering arrays in software interaction testing the reader is referred to [1].

Because of the importance of the construction of (near) optimal covering arrays, 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: direct methods [2], [3], recursive methods [4], [5], [6], greedy methods [7], [8], [9], [10], [11], and metaheuristics methods [12], [13], [14], [15], [16], [17].

In this paper, we aim to develop an improved implementation of a simulated annealing algorithm for constructing covering arrays. Simulated annealing algorithm is a general-purpose stochastic optimization technique that has proved to be an effective tool for approximating globally optimal solutions to many optimization problems. However, one of the major drawbacks of the technique is its very slow convergence. To address this drawback, we propose three parallel simulated annealing approaches to solve the
CAC problem. The objective is to find the best bounds to some ternary covering arrays by using parallelism. To our knowledge the application of parallel simulated annealing to the CAC problem has not been reported in the literature.

A parallel algorithm for the verification of covering arrays is presented in [18]. The methods of parallelization of simulated annealing are discussed in [19], [20], [21], [22], [23].

Contrary to existing simulated annealing implementations [12], [13], our algorithm has the merit of improving two key features that have a great impact on its performance: an efficient heuristic to generate good quality initial solutions and a compound neighborhood function which combines two carefully designed neighborhood relations.

The remainder of this paper is organized as follows: Section 2 describes the components of our sequential annealing algorithm, in Section 3 three parallel simulated annealing approaches are discussed, Section 4 describes the experimental results, and finally, Section 5 presents the conclusions derived from the research presented in this paper.

2. Sequential simulated annealing

In this section, we briefly review simulated annealing (SA) algorithm and propose an implementation to solve the CAC problem.

SA is a randomized local search method based on the simulation of annealing of metal. The acceptance probability of a trial solution is given by (2), where $T$ is the temperature of the system, $\Delta C$ is the difference of the costs between the trial and the current solutions (the cost change due to the perturbation), (2) means that the trial solution is accepted by nonzero probability $e^{-\Delta C/T}$ even though the solution deteriorates (uphill move). In this case, because the cost of the solution increases, we call this update on the solution an uphill move.

$$
(P) = \begin{cases} 
1 & \text{if } \Delta C < 0 \\
\frac{1}{e^{\Delta C/T}} & \text{otherwise} 
\end{cases} 
$$

(2)

Uphill moves enable the system to escape from the local minima; without them, the system would be trapped into a local minimum. Too high of a probability for the occurrence of uphill moves, however, prevents the system from converging. In SA, the probability is controlled by temperature in such a manner that at the beginning of the procedure the temperature is sufficiently high, in which a high probability is available, and as the calculation proceeds the temperature is gradually decreased, lowering the probability [24].

In order to use the simulated annealing metaheuristic for a combinatorial optimization problem in particular, there are a number of decisions to be taken. Johnson et al. [25] classified these decisions as follows:

- Problem-specific choices:
  - The problem must be clearly formulated, so that the set of feasible solutions is defined.
  - The neighborhood of any solution must also be defined as well as a way of determining the value of the objective to be minimized.
  - An initial solution must also be generated.

  - Generic choices:
    - Initial Temperature
    - Cooling schedule, a temperature function, $T(t)$, to determine how the temperature is to be changed
    - Maximum neighboring solutions per temperature, viz. the number of iterations, $N(t)$, to be performed at each temperature
    - A stopping criterion to terminate the algorithm

The following paragraphs will describe each of the components of the implementation of our SA. The description is done given the matrix representation of a covering array.

2.1 Internal representation

A covering array can be represented as a matrix $M$ of size $N \times k$, where the columns are the parameters and the rows are the cases of the test set that is constructed. Each cell $m_{i,j}$ in the array accepts values from the set $\{1, 2, \ldots, v_j\}$ where $v_j$ is the cardinality of the alphabet of $j$-th column. The size of the search space $M$ is then given by (3):

$$
|M| = v^Nk. 
$$

(3)

2.2 Initial solution

The initial solution $M$ is constructed by generating $M$ as a matrix with maximum Hamming distance. The Hamming distance $d(x, y)$ between two rows $x, y \in M$ is the number of elements in which they differ. Let $r_i$ be a row of the matrix $M$. To generate a random matrix $M$ of maximum Hamming distance the following steps are performed:

1) Generate the first row $r_1$ at random.
2) Generate $s$ rows $c_1, c_2, \ldots, c_s$ at random, which will be candidate rows.
3) Select the candidate row $c_i$ that maximizes the Hamming distance according to

$$
g(r_i) = \sum_{s=1}^{i-1} \sum_{v=1}^{k} d(m_{s,v}, m_{i,v}),
$$

where $d(m_{s,v}, m_{i,v}) = \begin{cases} 
1 & \text{if } m_{s,v} \neq m_{i,v} \\
0 & \text{Otherwise} 
\end{cases}$

and added to the $i$-th row of the matrix $M$.
4) Repeat from step 2 until $M$ is completed.

An example is shown in Fig. 1. The number of symbols different between rows $r_1$ and $c_1$ are 4 and between $r_2$ and $c_1$ are 3 summing up 7. Then, the hamming distance for the candidate row $c_1$ is 7.
2.3 Evaluation function

The evaluation function $C(M)$ is used to estimate the goodness of a candidate solution. Previously reported meta-heuristic algorithms for constructing covering arrays have commonly evaluated the quality of a potential solution (covering array) as the number of combination of symbols missing in the matrix $M$ [13], [14], [15]. Then, the expected solution will be zero missing. In the proposed SA implementation this evaluation function definition was used. Its computational complexity is equivalent to (4):

$$O\left( N\left(\frac{k^t}{t}\right)\right).$$

(4)

2.4 Neighborhood function

Given that SA is based on Local Search (LS) then a neighborhood function must be defined. The main objective of the neighborhood function is to identify the set of potential solutions which can be reached from the current solution in a local search (LS) algorithm. In case two or more neighborhoods present complementary characteristics, it is then possible and interesting to create more powerful compound neighborhoods. The advantage of such an approach is well documented in [26]. Following this idea, and based on the results of our preliminary experimentations, a neighborhood structure composed by two different functions is proposed for this SA algorithm implementation.

Two neighborhood functions were implemented to guide the local search of our SA algorithm. The neighborhood function $N_1(s)$ makes a random search of a missing $t$-tuple, then tries by setting the $j$-th combination of symbols in every row of $M$. The neighborhood function $N_2(s)$ randomly chooses a position $i,j$ of the matrix $M$ and makes all possible changes of symbol. During the search process a combination of both $N_1(s)$ and $N_2(s)$ neighborhood functions is employed by our SA algorithm. The former is applied with probability $\rho$, while the latter is employed at a $(1-\rho)$ rate. This combined neighborhood function $N_3(s, x)$ is defined in (5), where $x$ is a random number in the interval $[0, 1)$.

$$N_3(s, x) = \begin{cases} N_1(s) & \text{if } x \leq \rho \\ N_2(s) & \text{if } x > \rho \end{cases}$$

(5)

2.5 Cooling schedule

The cooling schedule determines the degree of uphill movement permitted during the search and is thus critical to the SA algorithm’s performance. The parameters that define a cooling schedule are: an initial temperature, a final temperature or a stopping criterion, the maximum number of neighboring solutions that can be generated at each temperature, and a rule for decrementing the temperature. The literature offers a number of different cooling schedules, see for instance [19], [27]. In our SA implementation we preferred a geometrical cooling scheme mainly for its simplicity. It starts at an initial temperature $T_1$ which is decremented at each round by a factor $\alpha$ using the relation shows in (6). For each temperature, the maximum number of visited neighboring solutions is $L$. It depends directly on the parameters $(N, k, \nu)$ of the studied covering array. This is because more moves are required for covering arrays with alphabets of greater cardinality.

$$T_k = \alpha T_{k-1}$$

(6)

2.6 Termination condition

The stop criterion for our SA is either when the current temperature reaches $T_j$, when it ceases to make progress, or when a valid covering array is found. In the proposed implementation a lack of progress exists if after $\phi$ (frozen factor) consecutive temperature decrements the best-so-far solution is not improved.

2.7 Simulated annealing pseudocode

The Algorithm 1 presents the SA heuristic as described above. The meaning of the three functions is obvious: INITIALIZE computes a start solution and initial values of the parameters $T$ and $L$; GENERATE selects a solution from the neighborhood of the current solution, using the neighborhood function $N_3(s, x)$; CALCULATE_CONTROL computes a new value for the parameter $T$ (cooling schedule) and the number of consecutive temperature decrements with no improvement in the solution.

Unlike the classical method which takes as a solution to the problem, the last value obtained in the annealing chain. We memorize the best solution found during the whole annealing process (see lines 3 and 12).

In the next section, it is presented three parallel SA approaches for solving the CAC problem.
3. Parallel simulated annealing

Parallelization is recognized like a powerful strategy to increase algorithms efficiency; however, SA parallelization is a hard task because it is essentially a sequential process.

In evaluating performance of a parallel simulated annealing (PSA), it needs to consider solution quality as well as execution speed. The execution speed may be quantified in terms of speed-up ($S$) and efficiency ($E$). The $S$ is defined as the ratio of the execution time (on one processor) by the sequential SA to that by the PSA (on $P$ processors) for an equivalent solution quality. In the ideal case, $S$ would be equal to $P$. The $E$ is defined as the ratio of the actual $S$ to the ideal $S^*(P)$.

Next, we propose three parallel implementations of the SA algorithm described in Section 2. For these cases, let $P$ denote the number of processors and $L$ the length of Markov chain.

3.1 Independent search approach

A common technique to parallelizing SA is the independent search approach (IS) [19], [21], [28]. In this approach each processor independently perturbs the configuration, evaluates the cost, and decides on the perturbation. The processors $P_i, i = 0, 1, \ldots, P - 1$, carry out the independent annealing searches using the same initial solution and cooling schedule as in the sequential algorithm. At each temperature $P_i$ executes $N \times k \times v^2$ annealing steps. When each processor finishes, it sends its results to processor $P_0$. Finally, processor $P_0$ chooses the final solution among the local solutions.

3.2 Semi-independent search approach

Our semi-independent search approach (SS) is an implementation of the division algorithm [19]. In the division algorithm, parallelism is obtained by dividing the effort of generating a Markov chain over the available processors. A Markov chain is divided into $P$ sub-chains of the length $[L/P]$. In this approach, the processors exchange local information including intermediate solutions and their costs. Then, each processor restarts from the best intermediate ones.

Compared to the IS, communication overhead in this SS approach would be increased. However, each processor can utilize the information from other processors such that the decrease in computations and idle times can be greater than the increase in communication overhead. For instance, a certain processor which is trapped in an inferior solution can recognize its state by comparing it with others and may accelerate the annealing procedure. That is, processors may collectively converge to a better solution.

3.3 Cooperative search approach

In order to improve the performance of the SS approach, we propose the cooperative search approach (CS), it uses asynchronous communication among processors accessing the global state to eliminate the idle times. Each processor follows a separate search path, accesses the global state which consists of the current best solution and its cost whenever it finished a Markov subchain and updates the state if necessary. Once a processor gets the global state, it proceeds to the next Markov subchain with any delay.

Unlike SS, CS having the following characteristics:

- Idle times can be reduced since asynchronous communications overlap a part of the computation.
- Less communication overhead, an isolated access to the global state is needed by each processor at the end of each Markov subchain.
- The probability of being trapped in a local optimum can be smaller. This is because not all the processors start from the same state in each Markov subchain.

4. Experimental results

This section presents an experimental design and results derived from testing the parallel IS, SS, and CS algorithms described in Section 3. In order to show the performance of these approaches, two experiments were developed. The first experiment had as purpose to compare the three approaches in terms of parallel execution time. Among the three approaches, the CS approach was the fastest. The second experiment evaluated the quality solutions of CS approach over a new benchmark proposed in this paper. The results were compared against the best-known solutions reported in the literature [29] to construct covering arrays. The three parallel approaches were implemented using C

---

**Algorithm 1** Sequential SA approach for the CAC problem.

```plaintext
1: function SA()
2:   INITIALIZE($M$, $T$, $L$
3:   $M^*$ ← $M$
4:   repeat
5:     for $i$ ← 1 to $L$
6:       $M_i$ ← GENERATE($M$
7:       $\Delta C$ ← $C(M_i)$-$C(M$
8:       $x$ ← random $\succ x$ in the range $[0,1)$
9:       if $\Delta C < 0$ or $e^{-\Delta C/T} > x$ then
10:          $M$ ← $M_i$
11:          if $C(M) < C(M^*)$ then
12:             $M^*$ ← $M$
13:             end if
14:       end if
15:   end for
16:   CALCULATE_CONTROL($T$, $\phi$
17: until stop_criterion
18: end function
```
language and the message passing interface (MPI) library. The implementations were run on the Tirant supercomputer.\footnotemark\footnotetext{The Tirant supercomputer: http://www.uv.es/siuv/cas/zcalculo/ces/informa.wiki} Tirant comprises 256 JS20 compute nodes (blades) and 5 p515 servers. Every blade has two IBM Power4 processors at 2.0 GHz running Linux operating system with 4 GB of memory RAM and 36 GB of local disk storage. All the servers provide a total of nearly 10 TB of disk storage accessible from every blade through GPFS (Global Parallel File System). Tirant has in total 512 processes, 1 TB of memory RAM and 19 TB of disk storage. The networks that interconnect the Tirant are:

1) Myrinet Network: High bandwidth network used by parallel applications communications.
2) Gigabit Network: Ethernet network used by the blades to mount remotely their root file system from the servers and the network over which GPFS works.

The following parameters were used for all SA implementations:

1) Initial temperature \( T_1 = 4.0 \)
2) Final temperature \( T_f = 1.0E - 10 \)
3) Cooling factor \( \alpha = 0.99 \)
4) Maximum neighboring solutions per temperature \( L = N \times k \times v^2 \)
5) Frozen factor \( \phi = 11 \)
6) The neighborhood functions \( N_1 \) and \( N_2 \) are applied with a probability of \( p_1 = 0.3 \) and \( p_2 = 0.7 \) respectively.

4.1 Comparison of algorithms

To test the performance of the IS, SS, and CS approaches, we propose the construction of a covering array with \( N = 80, t = 3, k = 22, \) and \( v = 3 \). Each approach was executed 31 times (for provide statistical validity to experiment) using \( P = \{4, 8, 16, 32\} \).

The performance of the algorithms has been compared based on median speed-up as a function of the number of processors, the results are shown in Fig. 2.

The IS approach, had difficulty in handling the large problem instances, it does not scale. The SS approach provides reasonable results, however, because it is a synchronous algorithm, the idle and communication times are inevitable. The CS approach is who offers the best results, it reduces the execution time of the SS approach by employing asynchronous information exchange.

In the next subsection, it is presented the second experiment of this work, the purpose is to measure the performance of the CS algorithm against the best-known solutions reported in the literature.

4.2 The cooperative search approach against the state-of-the-art procedures

The purpose of this experiment is to carry out a performance comparison of the bounds achieved by the CS approach with respect to the best-known solutions reported in the literature [29], which were produced using the following state-of-the-art procedures: orthogonal array construction, Roux type constructions, doubling constructions, algebraic constructions, Deterministic Density Algorithm (DDA), Tabu Search and IPOG-F.

For this experiment we have fixed the maximum computational time expended by our PSA for constructing a covering array to 72 hours and 50 processors. We create a new benchmark composed by 96 covering arrays with strength \( t = 4 \), degree \( 5 \leq k \leq 100 \), and order \( v = 3 \). The detailed results produced by this experiment are listed in Table 1. The first two columns in this table indicate the strength \( t \) and the degree \( k \) of the selected instances. Next two columns show, in terms of the size \( N \) of the covering arrays, the best-known solution reported in the literature and the improved bounds produced by the CS.

The analysis of the data presented in the Table 1 lead us to the following observation. The solutions quality attained by the CS approach is very competitive with respect to that produced by the state-of-the-art procedures summarized in column 3 (\( \theta \)). In fact, it is able to improve on 78 previous best-known solutions.

5. Conclusions

The long execution time of SA due to its sequential nature hinders its application to realistically sized problems, in this case, the CAC problem when \( t > 3, 5 \leq k \leq 100, \) and \( v = 3 \). A more efficient way to reduce execution time and make the SA a more promising method is to parallelize
We have verified all covering arrays described in this paper using the tool presented in [29]. β represents the best-known solution reported in the literature [29]. The quality of the solutions attained by the CS approach is very competitive with respect to that produced by the state-of-the-art procedures, in fact, it is able to improve on 78 previous best-known solutions and equals the solutions for other 7 instances.

Finally, the covering arrays are available in CINVESTAV Covering Array Repository (CAR), which is available under request at http://www.tamps.cinvestav.mx/~jtj/CA.php. We have verified all covering arrays described in this paper using the tool presented in [30].

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


