


# An investigation of distributed computing for combinatorial testing

Edmond La Chance | Sylvain Hallé 

Laboratoire d'informatique formelle, Université du Québec à Chicoutimi, Chicoutimi, Quebec, Canada

## Correspondence

Sylvain Hallé, Université du Québec à Chicoutimi, 555 boul. de l'Université, Chicoutimi, QC G7H 2B1, Canada.  
Email: [shalle@acm.org](mailto:shalle@acm.org)

## Funding information

Canada Research Chair on Software Specification, Testing and Verification; Natural Sciences and Engineering Research Council of Canada, Grant/Award Number: RGPIN-2016-05626

## Summary

Combinatorial test generation, also called  $t$ -way testing, is the process of generating sets of input parameters for a system under test, by considering interactions between values of multiple parameters. In order to decrease total testing time, there is an interest in techniques that generate smaller test suites. In our previous work, we used graph techniques to produce high-quality test suites. However, these techniques require a lot of computing power and memory, which is why this paper investigates distributed computing for  $t$ -way testing. We first introduce our distributed graph colouring method, with new algorithms for building the graph and for colouring it. Second, we present our distributed hypergraph vertex covering method and a new heuristic. Third, we show how to build a distributed IPOG algorithm by leveraging either graph colouring or hypergraph vertex covering as vertical growth algorithms. Finally, we test these new methods on a computer cluster and compare them to existing  $t$ -way testing tools.

## KEYWORDS

combinatorial testing, distributed computing, phi-way testing, test case generation

## 1 | INTRODUCTION

Testing computer software with various interactions of inputs is one of the most effective ways to test software known today. However, when the situation involves a considerable number of variables, each having different values, the number of different input-states a target program can reach grows exponentially. For this reason, exhaustive testing is not feasible, considering that some problems would take more than a lifetime to test.

In recent years, combinatorial testing has gathered increasing interest as a testing technique that can exercise interactions between input parameters in an efficient way. Instead of including the interactions of all values for every parameter, a combinatorial test suite only requires the presence of all interactions of values for each set of  $t$  parameters; this is why this testing technique is called ' $t$ -way testing'. A seminal study [1] has shown that combinatorial test suites can be very efficient at finding bugs in real-life systems since these bugs are typically caused by the interaction of a small number of parameters (four to six parameters). As a consequence, a well-designed combinatorial test suite can, in practice, have the same bug-finding power as an exhaustive enumeration of all parameter values, using a much smaller number of test cases.

In the past decade, various testing tools such as PICT [2], Jenny [3] or ACTS [4] have been developed to generate combinatorial test suites for a configurable number of parameters, strength and number of values for each parameter. In the general case, finding the *smallest* test suite for a given configuration is a computationally hard problem [5]; in some cases, only lower and upper bounds on the minimum test suite size are known. This is why most of the combinatorial test generation algorithms attempt to strike a balance between computation time and producing a test suite that lies 'reasonably' close to some (unknown) optimum.

This is an open access article under the terms of the [Creative Commons Attribution-NonCommercial-NoDerivs](https://creativecommons.org/licenses/by-nc-nd/4.0/) License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.

© 2023 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

However, existing tools, being bounded by the capabilities of one machine, have limitations on the amount of computing power and memory they can exercise. Recent graph-based techniques, such as graph colouring or hypergraph vertex covering, require even more processing power to produce test suites [6]. A few algorithms making use of parallelism have been presented in the recent past [7–9], but these techniques focus on making more processors contribute to the algorithm; they do not partition the problem on multiple servers. Therefore, such techniques cannot scale beyond the memory and processing power of a single server.

Consequently, even the most efficient implementations are bound to run into processing power and/or memory limitations. To leverage the power of distributed computing platforms, new algorithms must be designed. To this end, the present paper investigates the use of distributed graph-based algorithms to allow  $t$ -way test suite generation to be taken to clusters of multiple machines. First, in Section 2, we introduce the classical problem of  $t$ -way testing and present a number of use cases where the approach has been successfully used, followed by a more specific literature review of In-Parameter-Order algorithms.

Section 3 then deals with distributed algorithms and data structures for compressed graphs. Two methods are provided to solve a  $t$ -way testing scenario. The first method is to use graph colouring. We begin by proposing a faster way of building the graph with an algorithm named *Database Graph Construction*. The graph produced by this algorithm can then be coloured using a distributed algorithm called *Knights and Peasants* (K&P). Both of these algorithms are original contributions of this paper. The second method is to use hypergraph vertex covering. For this method, we have devised a heuristic called *Greedy picker* that accelerates the construction of the covering array. Equipped with these algorithms, we then revisit the classical IPOG algorithm and provide a new version that is distributed. We call this algorithm D-IPOG. We also adapt IPOG's enumeration algorithm into a distributed version.

Section 4 describes how these algorithms have been concretely implemented into TSPARK, an open source  $t$ -way testing tool that leverages Apache Spark to run on computer clusters. Experimental results on Compute Canada, an organization that provides computing clusters to academia, show that graph-based techniques are promising for certain types of problems. Section 5 summarizes our findings and highlights some of the limitations of the current tool that open the way to future improvements.

## 2 | STATE OF THE ART IN COMBINATORIAL TEST CASE GENERATION

In this section, we provide a brief overview of basic concepts in combinatorial test generation. We then show, through multiple use cases, how this technique has been successfully used in practice in a variety of fields. We then present a survey of existing IPOG-based algorithms. For a recent and thorough literature review of all techniques and algorithms for combinatorial testing, we recommend the work of Torres-Jimenez et al. [10] to the reader.

### 2.1 | Basic concepts and definitions

Combinatorial test generation is a testing technique used to detect unwanted behaviour in software. It considers a system under test (SUT) taking as input multiple parameters, each taking arbitrary values taken from a finite set, and producing a given output. It stems from the observation that complex software bugs can occur when the interaction between different parameters sends the processor on a bad code path. Knowing this, a programmer interested in proof-testing his code could try every possible interaction of parameters. However, while this is possible for small problems, it cannot be done for larger problems because of the exponential growth in interactions. To work around this reality, combinatorial testing aims not at testing every interaction of parameters, but rather  $t$ -interactions of parameters, with parameter  $t$  being defined as the *interaction strength*—hence the frequent use of ‘ $t$ -way testing’ to describe this technique.

The *covering array* is a mathematical object used to represent a test suite for interaction testing. A test suite is said to cover a SUT when every  $t$ -way interaction is tested at least once; however, this condition does not guarantee that the test suite is minimal. A notation for covering arrays is  $CA(N; t, n, v)$ , where  $N$  is the number of tests in the test suite,  $n$  is the number of parameters in the problem,  $v$  is the domain size (number of values for every parameter) and  $t$  is the interaction strength. When every parameter has the same domain size (the same number of different values), a covering array is said to be *uniform*. When this is not the case, a covering array is said to be *mixed*. When describing mixed covering arrays, a more precise notation is used to describe the different domain sizes:  $MCA(N; t, n, v_1, \dots, v_n)$ . For instance, in Ahmed et al. [11], the lexical analyser **flex** was tested, and the following MCA was found:  $MCA(N; t, 2^4 3^1 16^1 6^1)$ . This notation is space-saving because it groups parameters with the same domain sizes. In the example given, the SUT

is made of four parameters with two values, one parameter with three values, one parameter with 16 values and one parameter with six values.

The construction of covering arrays is centred on the concept of  $t$ -way interactions. A  $t$ -way interaction can be seen as the smallest unit of coverage: It represents an interaction between  $t$  valued parameters. To solve a given  $t$ -way problem, a test generator must cover all the  $t$ -way interactions generated by the problem. Coverage of interactions is accomplished by creating tests, with a single test case allowed to cover multiple interactions.

As an example, Figure 1 shows a  $t$ -way problem having three parameters ( $n=3$ ), pairwise interaction strength ( $t=2$ ) and two values per parameter ( $v=2$ ). In this particular problem, a  $t$ -way interaction is a pair of parameter values, since the interaction strength is 2. When a parameter is absent inside a given interaction, the star character (\*) is often used as a placeholder. These  $t$ -way interactions can be covered using the test suite seen on the right of the figure.

The  $t$ -way test generation (more generally called ‘combinatorial test case generation’) consists of generating a CA for given values of  $n$ ,  $t$  and  $v$ . This problem is separate from the *oracle problem*, which consists of determining whether a test is a success or a failure when it is actually executed on the SUT. A  $t$ -way test generator takes a description of a SUT and produces an optimized test suite.

Consider as an example an SUT that takes the form of a font dialogue box of the Windows operating system (Figure 2). When the user selects different parameter values and checkboxes, the text changes in the live preview; the goal is therefore to ensure that no error occurs when selecting different interactions of parameters. Using a software tool such as ACTS [12], a user can define the names of each parameter and the set of possible values, as in Figure 3. The tool then produces a covering array for a given interaction strength; each line of this array is a single test, which in this case defines the precise values of each parameter (font name, style, size and status of each checkbox).

Many studies have shown the effectiveness of  $t$ -way testing in practice. As reported by Torres-Jimenez et al. in their review [10], a series of studies [1, 13–16] done by the National Institute of Standards and Technology (NIST) found that interactions involving six parameters exposed most problems. We can mention a few fields where  $t$ -way testing has been successfully used: open source software [17], hardware Trojan detection [18], fire accident reconstruction [19], deep learning systems [20], biological systems [21] and web applications [22].

## 2.2 | In-Parameter-Order algorithms

The family of algorithms called *In-Parameter-Order* contains algorithms such as IPO [23] (which works for  $t=2$ ) and IPOG [12] (for any value of  $t$ ). This strategy is used in tools such as Jenny [3], ACTS, PICT [24, 25] and CAGEN [26]. It works by covering a  $t$ -way problem one parameter at a time. This method is favoured because it controls the growth of the algorithm; this makes it easier to fit the essential data structures into RAM. Another advantage of covering a system one parameter at a time is that the coverage can be paused and then resumed. For instance, when covering a

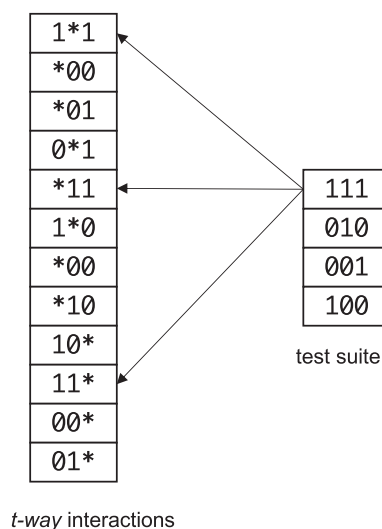


FIGURE 1 Covering array  $CA(4;2,3,2)$  and its associated  $t$ -way interactions.

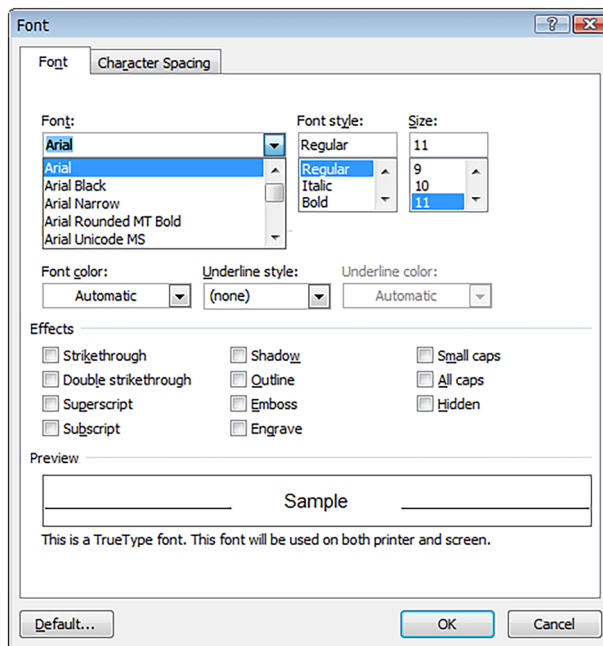


FIGURE 2 A font dialogue box. Each checkbox represents a parameter for a combinatorial test case generation problem.

Saved Parameters		
Parameter Name	Parameter Type	Parameter Value
Font	Enum	[Arial,Courier New,Times New
style	Enum	[Regular,Italic,Bold]
strikethrough	Boolean	[true,false]
double_strikethrough	Boolean	[true,false]
superscript	Boolean	[true,false]
subscript	Boolean	[true,false]
shadow	Boolean	[true,false]
outline	Boolean	[true,false]
emboss	Boolean	[true,false]
engrave	Boolean	[true,false]
smallcaps	Boolean	[true,false]
allcaps	Boolean	[true,false]
hidden	Boolean	[true,false]

FIGURE 3 Building the system in ACTS.

problem of 400 parameters, one can end the calculation after covering 300 parameters, write the current test suite to disk and then resume the computation by having the program load the test suite.

IPOG is the generalized version of IPO that supports  $t$ -way testing [12]; Algorithm 1 details its operation. IPOG first starts by generating a test suite for the first  $t$  parameters (out of  $n$ ). Then, it generates the interactions needed to cover the next parameter using its enumeration algorithm. In the second iteration, this would be the  $(t+1)$ th parameter. Then, the test suite is horizontally extended. Every test is extended using the value that covers the most interactions. This phase of the algorithm is called *horizontal growth*. After horizontal growth, there are usually interactions remaining to cover. To cover these, the algorithm enters a phase called *vertical growth*. IPOG's vertical growth covers the remaining interactions one at a time. First, it tries to fit an interaction inside a test with available room. If it cannot, it creates a new test and places the interaction inside.

For an IPOG algorithm to perform these operations in an efficient manner, there needs to be a fast way to check whether or not an interaction is already covered by a test. The ACTS implementation of IPOG leverages the speed of a bitmap data structure for this. Using this data structure, a single bit represents the coverage status of an

$t$ -way interaction. To access the right bitmap, a formula is used to calculate an index in the array of pointers to bitmaps.

A variant called Modified IPOG (MIPOG) [27] modifies the horizontal growth's search process by also considering don't-care values inside a test when optimizing its last parameter—a job that IPOG leaves to vertical growth. As for its vertical growth tests, MIPOG spends more effort optimizing, as it produces a test by trying to combine the maximum number of uncovered  $t$ -way interactions every time. Also, MIPOG's vertical growth does not consider don't-care values since they are already filled by its horizontal growth phase.

A parallel version of MIPOG called MC-MIPOG also exists [8]. In MC-MIPOG, a driver thread handles the creation, synchronization and deletion of all its worker threads. When covering a new parameter  $P_i$  with four values, the driver thread creates four worker threads called combinatorial threads. Inside these threads, we enumerate the yet uncovered  $t$ -way interactions for that value of  $P_i$  only. So Thread 1 would enumerate the uncovered  $t$ -way interactions related to  $P_i = 1$  (if the values of  $P_i$  are 1, 2, 3, 4). Then, we delete these threads and recreate them as 'horizontal extension' threads. Every such thread accesses the interactions previously created by its cousin thread. Then, for each test of the test suite, the threads report on the number of tuples that would be covered by the test. The driver program then makes its greedy choice using this data, following the MIPOG algorithm.

For its vertical growth phase, MC-MIPOG also spawns the same number of threads (one for each value), but this time, the  $t$ -way combining method is used with the local interactions owned by these threads and creates a local test suite. These partial test suites are then merged with the main test suite. Recently, algorithmic enhancements to IPOG's have been found [28]. By simultaneously computing the coverage gain for same-prefix tuples during horizontal growth, an important speedup is obtained. Other implementation-level optimizations are also introduced, such as generating specialized code for different values of  $t$  at compile time. These techniques are available in a new tool called CAGEN [26].

---

**Algorithm 1: IPOG algorithm.**


---

```

input  :  $t$ : the interaction strength
           $n$ : the number of parameters,  $v$ : the domain size
output  $ts$ : the test suite
:
1  Generate the exhaustive test suite for the first  $t$  parameters. It will be used as the initial test suite
2   $ts \leftarrow \text{GenerateInteractions}(t,t)$ 
3  for  $i = t+1; i \leq n; i++$  do
4  |   Let  $\pi$  be the  $t$ -way interactions involving parameter  $P_i$  and  $t - 1$  other parameters among the
      |   parameters already covered
5  |   Horizontal Growth Algorithm:
6  |   for every test in  $ts$  do
7  |   |   Choose the value  $v_i$  of  $P_i$  that covers the most  $t$ -way interactions.
8  |   |   Remove the  $t$ -way interactions covered by this value
9  |   end
10 |   Vertical Growth algorithm:
11 |   for every  $t \leftarrow$  interactions do
12 |   |   if There is room inside an existing test then
13 |   |   |   Modify existing test to include  $t$ 
14 |   |   end
15 |   |   else
16 |   |   |   Add a new test that only contains  $t$ 
17 |   |   |    $ts+ = \text{newTest}$ 
18 |   |   end
19 |   end
20 end
21 return  $ts$ 

```

---

### 3 | DISTRIBUTED ALGORITHMS FOR COMBINATORIAL TEST GENERATION

In this section, we describe distributed algorithms to solve the  $t$ -way test suite generation problem. First, in Section 3.1, we describe an algorithm to generate the search space of all  $t$ -way interactions in a distributed manner, by partitioning

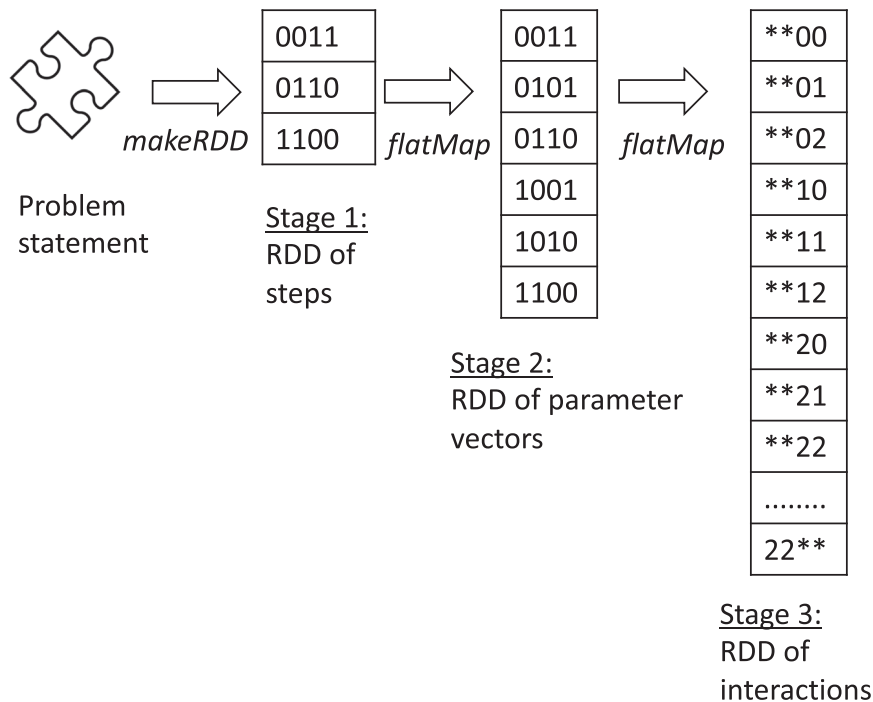


FIGURE 4 Distributed generation of  $t$ -way interactions for  $t=2, n=4, v=3$

the set of interactions to be produced across multiple machines. In Sections 3.2 and 3.3, we revisit two reductions of the  $t$ -way test suite generation into graph problems and provide efficient distributed algorithms to solve them. In Section 3.2, we present an original distributed graph colouring algorithm, while in Section 3.3, we do the same for the reduction to hypergraph vertex covering. In Section 3.4, we revisit the original IPOG algorithm and show how each of its three main phases can be replaced with distributed algorithms, thereby forming the basis for a distributed variant we call D-IPOG. Finally, in Section 3.5, we compare our methods to existing parallel or distributed approaches in the literature.

We shall first mention that the distributed algorithms that are described in this section are implemented using computation primitives that are variants of those offered by the MapReduce programming model: `map` and `reduce` [29]. MapReduce improves upon the Message Passing Interface (MPI) standard [30] by providing fault tolerance, meaning that a given `map` or `reduce` task can be automatically retried when failure is detected. The initial implementations of MapReduce are Google's MapReduce [31] and Hadoop's MapReduce [32]. Since then, new frameworks with performance improvements such as memory-only execution have appeared: Apache Spark [33, 34], Apache Flink [35] and Apache Tez [36]. Such frameworks are leveraged by distributed databases or data warehouses (e.g., Apache Hive [37]) to implement SQL queries [38]. The computation primitives used in this section use Apache Spark's syntax. We refer the reader to Spark's RDD Programming guide,<sup>1</sup> a short document which is sufficient to fully understand any algorithm presented here.

### 3.1 | Distributed generation of interactions

A first step, which is prerequisite to any distributed test case construction procedure, is to generate the search space, which consists of  $t$ -way interactions.

We have adapted IPOG's [12] generation algorithm into a three stage algorithm which can be seen in Figure 4. In Stage 1, we generate a number of 'steps'. A step is a parameter vector eventually enumerated by the algorithm; it is used as an execution boundary in the distributed generation algorithm. With this, a distributed task will stop generating parameter vectors using either the boundary or the usual exit condition of the algorithm. To generate the

<sup>1</sup><https://spark.apache.org/docs/latest/rdd-programming-guide.html>.



steps, we start with a string that contains a number of zeroes equal to the number of parameters of the problem. Then, we place inside a number of 1s equal to the interaction strength, and we shift them left until we can't, every time creating a step in the process. In Stage 2, we map the 'steps' into a number of parameter vectors. The value of the step is used as a boundary. Finally, in Stage 3, a parameter vector is transformed into several  $t$ -way interactions. For every transformation stage, we use the 'flatMap' primitive to transform steps into parameter vectors and parameter vectors into interactions.

### 3.2 | Distributed $t$ -way through graph colouring

In past work, a reduction of  $t$ -way test suite generation to graph colouring has been presented [6]. Since graphs naturally lend themselves to distribution, it is reasonable to expect that the reduction of the  $t$ -way problem to a graph structure could provide good results. The reduction works as follows: For given values of  $t, v$  and  $n$ , one first builds a graph  $G$  whose vertices consist of all the parameter interactions, using, for example, the technique described in Section 3.1. From this set of vertices, the adjacency relationship is created such that two vertices are linked with an edge if they assign conflicting values to the same parameter. Figure 5 shows an example of a graph with three parameters  $a, b$  and  $c$ , each taking values 0 or 1, and an interaction strength of  $t = 2$ .

We recall that a colouring of a graph is an assignment of each vertex to a colour, such that no pair of adjacent vertices is given the same colour. Given a colouring of  $G$ , one can build a test suite by creating test cases as the union of all vertices assigned to the same colour. In the previous example, the test case corresponding to the colour yellow would correspond to the assignment  $a = 0, b = 0$  and  $c = 0$ .

**Theorem 1 From Hallé et al. [6]** *Let  $T$  be a test suite built from a valid colouring of the graph  $G$  constructed as described. Then,  $T$  is a valid  $t$ -way test suite.*

Thus, a first possible way of solving the  $t$ -way problem in a distributed manner is by directly attempting to compute a colouring for  $G$ .

In a distributed graph colouring algorithm, every graph vertex is a computation unit. And a vertex has one goal: to obtain a valid colour. When every vertex has been coloured, the graph colouring is a success. However, for the graph colouring to be correct, the colour a vertex chooses must be different from the colours of its neighbours. When several

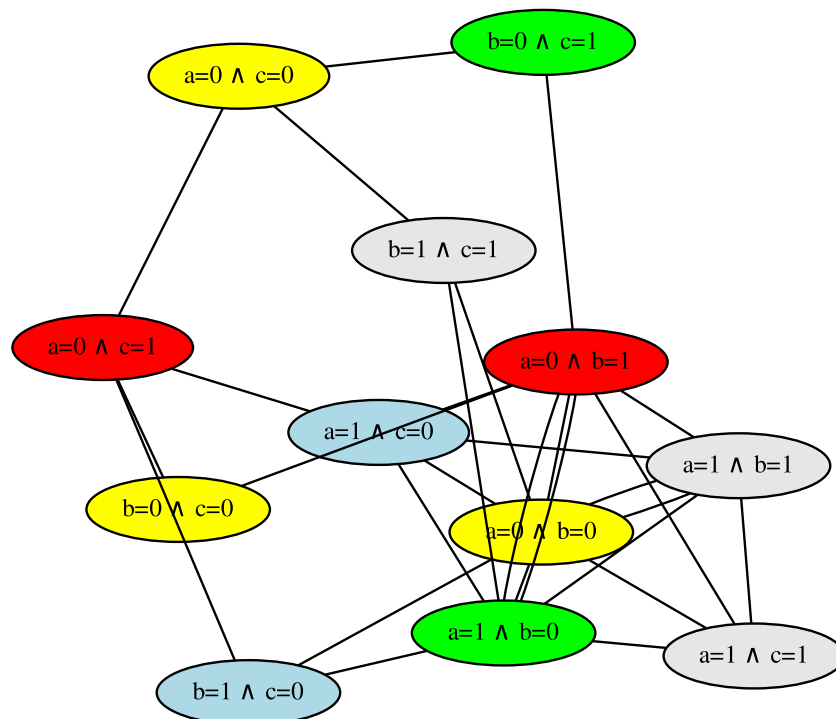


FIGURE 5 An example of a graph built by the reduction of  $t$ -way to graph colouring [6].

vertices try to choose their lowest available colour (when colours are described using integers), their choices can collide, and the iteration will not produce a valid graph colouring. Therefore, the simplest solution is to assign priorities which will be used by the vertices to organize. This approach is a classic way to solve symmetry breaking problems [39–41]. However, such an algorithm will suffer from priority congestion when the graph is dense. A given vertex, being linked to so many others, will have to wait for many distributed iterations to colour itself, because many vertices in the neighbourhood have priority over it.

This is a problem for performance. Reducing the number of iterations used by a distributed algorithm is always key, since distributed iterations, often suffering from network latencies, are very slow [42]. A solution to this problem is seen with the multi-trials technique [43]. This technique, when applied to graph colouring, will see a vertex attempt to choose several colours in the same communication round. This number is managed by the algorithm by considering a maximum number of colours, which is often bounded by the graph's maximum degree  $\Delta$ . When this number is allowed to be high, the number of communication rounds can be cut down significantly. For instance, by allowing up to  $O(\Delta^2)$  colours, it has been shown that a graph colouring can be completed in  $O(\log^* n)$  steps [44]. However, if the aim is to produce a small graph colouring, then a classic symmetry breaking strategy is preferred [41].

For this reason, out of concern for solution quality, we elect not to use a graph colouring algorithm like multi-trials. Instead, we propose a solution that uses two algorithms, with one for each graph type. For sparse graphs, we employ a distributed graph colouring algorithm. These graphs do not have much priority congestion and are therefore perfectly suited to distributed decision-making. Our original algorithm, K&P, uses a static symmetry breaking system which has its vertices choose their lowest possible colour at every iteration. For dense graphs, which have a lot of priority congestion, we revert to a single-threaded algorithm.

The following section details our graph colouring strategies. We will first present a faster algorithm called *Database Graph Construction* to build the graph for a  $t$ -way problem. Then, we will detail the K&P algorithm along with our graph compression strategy.

### 3.2.1 | Efficient graph generation

An important aspect of the colouring approach is the efficient generation of the graph. A naïve way of generating the graph is to enumerate all  $t$ -way interactions and link these interactions according to their compatibility; this was the method used in the original work presenting the reduction [6]. Such an algorithm has a complexity of  $\frac{1}{2} \cdot \binom{n}{t} v^t n$ . In this section, we present a faster algorithm called *Database Graph Construction*, which works by having every  $t$ -way interaction build its own adjacency matrix directly, using two databases and set operations.

First, we assign a unique number to every  $t$ -way interaction (this number is the vertex id). Then, we build a database for values and a database of 'stars' (stars are don't-care values represented by a \* symbol). This database can be implemented with an array of arrays of sets or a dictionary data structure. The database of stars can be implemented using an array of sets, or a dictionary. It contains an entry for every parameter value, and these entries map to a set that contains vertex ids.

For instance, let us have three two-valued parameters and a  $t$ -way interaction like  $P_0 = 1, P_2 = 0$  with a unique id of 2. In the database, at  $P_0 = 1$  and  $P_2 = 0$ , we would insert 2 into the set. Also, since this interaction does not contain a parameter  $P_1$ , we insert its id into the database of stars at parameter  $P_1$ . When every  $t$ -way interaction is loaded onto the databases in this manner, we run Algorithm 2. Running this algorithm creates the adjacency vector of a  $t$ -way interaction.

Algorithm 2 works in the following way. We create a list of invalid interactions for every parameter by doing the following: We take the set associated with the current parameter value in *database*, and we also take the set associated with the current parameter in *stars*. We perform the union of these two sets, and we compute the complement to output the set of invalid interactions for this parameter only. We repeat this process for every parameter, we merge the sets, and we output the adjacency vector.

It takes a negligible time to build the database, and Algorithm 2 runs for every  $t$ -way interaction. The algorithm loop is run  $t$  times, and inside the loop, three set operations are performed, with each taking  $O(n)$  time. The set operations are all bounded with the vertex *id*, which means that Vertex 10 always works with a set that can contain the elements 0...9. Therefore, the time complexity of this algorithm is  $\frac{1}{2} \cdot \binom{n}{t} v^t t n$ , which represents an important decrease compared to the original naïve algorithm. Moreover, such an algorithm has been empirically tested and shown to provide improved performance when used with data structures such as bit sets or RoaringBitMaps [45–47]. This is explained by the fact that most of the work is done by branchless set operators, which implies that the process does not suffer from execution pipeline stalls [48, 49].



**Algorithm 2:** Database Graph Construction.

---

```

input : interaction, database, stars
output : avoidList: a RoaringBitmap or BitSet
1 avoidList  $\leftarrow$  new RoaringBitmap()
2  $i \leftarrow 0$  The parameter counter
3 for every parameter  $p$  in this interaction do
4   if  $p \neq *$  then
5     paramVal  $\leftarrow$  paramVal - '0'
6     listSame  $\leftarrow$  database[ $i$ ][paramVal]
7     listUndecided  $\leftarrow$  stars[ $i$ ]
8     listAvoid  $\leftarrow$  FLIP(listSame  $\cup$  listUndecided)
9     avoidList  $\cup$  listAvoid
10  end
11   $i \leftarrow i + 1$ 
12 end
13 return avoidList

```

---

### 3.2.2 | Generating test suites with distributed graph colouring

The graph colouring process works as follows: we build and colour the graph chunk by chunk from the Resilient Distributed Dataset (RDD) containing all the  $t$ -way interactions. The graph is built and coloured this way to minimize total RAM usage on the cluster. In addition, the RDD of interactions is randomly permuted beforehand, which guarantees a different colouring every time.

We begin with the data structures used and how the graph is stored using a single RDD. The RDD stores vertices using pairs of the following type: [**id**, **Adjvector**]. The id of a vertex is a long integer, and the Adjvector is a data structure that contains the adjacency information between vertex id and Vertex 0,1,2...id-1. We only build the adjacency data between a vertex and vertices which have a lower id; a useful optimization that cuts the final graph size in half.

To represent an adjacency vector, several data structures are interesting. First, we have the bit set data structure. With a bit set or bit map, we use individual bits to indicate adjacency with another vertex. For instance, if bits 0 and 5 are set, then this vertex has an edge to Vertices 0 and 5. We can also use a compressed bit set like a RoaringBitMap (see Section 3.2.1).

We use the Database Graph Construction algorithm to build the graph. Not only does this algorithm offer an algorithmic speedup, it also does away with more expensive distributed computing primitives like **reduceByKey**, as it only uses **flatMap** to build the graph.

### 3.2.3 | The knights and peasants algorithm

We now describe a distributed graph colouring algorithm called K&P. In this algorithm a graph node can either be a *knight* or a *peasant*. The goal of each peasant node is to eventually become a knight; and when this is the case for all nodes, the algorithm terminates with a valid colouring of the graph. At the beginning of the algorithm, we start with a graph that contains only peasants. Then, these peasants are assigned a numerical value that acts as a priority level; this value is called a *tiebreaker*. This tiebreaker value stays the same for the duration of the algorithm; this symmetry breaking mechanic is used in classical distributed algorithms such as Lamport's Bakery Algorithm [39]. Tiebreakers can be generated by shuffling a sequence of numbers. One can also incorporate the degree of the vertex by giving higher degree vertices better tiebreakers to prioritize their colouring, much like the DSatur algorithm [50].

At each iteration of the algorithm, adjacent peasants communicate their tiebreaker value; a peasant that has the lowest tiebreaker value out of its neighbourhood of peasants can become a knight. When this happens, the peasant takes on a colour. This colour is chosen by looking at the colour of adjacent knights and picking the lowest colour number not used by any of them. When a peasant becomes a knight, it stops all peasant activities and only serves to choose colours for future knights.

Figure 6 shows how the Petersen graph is coloured by this algorithm. In the first iteration (S1), three peasants become knights. The top node, with a tiebreaker value of 6, has the smallest tiebreaker out of all its neighbours ( $\{7,10,11\}$ ) and becomes a knight. The same happens with two other nodes (1 and 3). When a node becomes a knight,

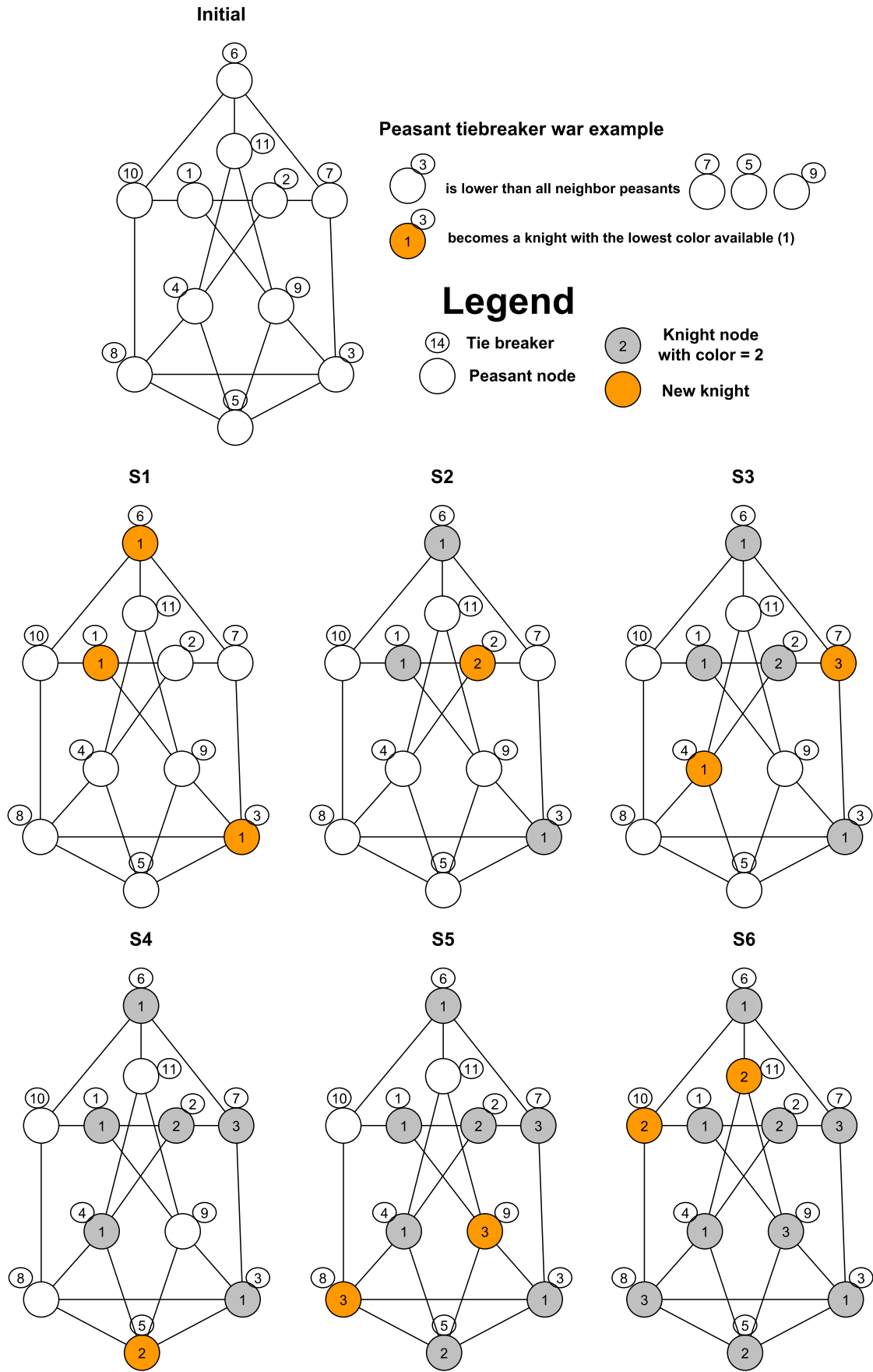


FIGURE 6 An illustration of the K&P distributed colouring algorithm on the Petersen graph.

it creates a list of all adjacent knights' colours; when the list is empty, a node takes Colour 1. In the second iteration, a single new node becomes a knight: the peasant with Tiebreaker 2, which has the lowest tiebreaker in its neighbourhood. It then builds the list of adjacent colours by looking at adjacent knights, which this time contains the value 1; therefore, it assumes Colour 2. The process repeats, and the algorithm finishes after six iterations.

In this algorithm, the structure of the graph (dense or sparse) and tiebreaker setup influences the number of iterations needed to reach a colouring. It can be reasoned that the algorithm is more efficient at colouring sparse graphs than dense graphs: Dense graphs will produce neighbourhoods with many edges and create tiebreaker traffic. On the other hand, a sparse graph has little tiebreaker traffic, and the graph will be coloured in much fewer iterations.

---

**Algorithm 3: Knights and Peasants.**


---

```

input : colors: array of colors, graph: RDD[Long, RoaringBitmap], currentMaxColor
output : currentMaxColor

1 remaining ← graph.count()
2 while true do
3   if remaining ≤ 0 then
4     | break
5   end
6   colorsbcast ← sc.broadcast(colors)
7   colorsRDD ← graph.flatMap elem ⇒
8     begin
9     | neighborColors ← new Array[Int](currentMaxColor+1)
10    | betterPeasant ← false
11    | c ← 0
12    | if colors.value(elem.id) = 0 then
13    | | for every neighbor in elem.adjlist do
14    | | | neighborColor ← colors.value(neighbor)
15    | | | if neighborColor is 0 (peasant) then
16    | | | | betterPeasant ← true
17    | | | | break
18    | | | end
19    | | | end
20    | | | if betterPeasant is false then
21    | | | | c ← color(neighborColors)
22    | | | | end
23    | | | end
24    | | if c ≠ 0 then
25    | | | Array(thisId, c)
26    | | else
27    | | | Array()
28    | | end
29    | end
30  results ← colorsRDD.collect()
31  results.foreach elem ⇒ begin
32  | colors(elem.id) ← elem.color
33  | remaining ← remaining-1
34  | if elem.color ≥ currentMaxColor then
35  | | currentMaxColor ← elem.color
36  | end
37  end
38 end
39 return currentMaxColor

```

---

Now, we can comment the pseudocode of Algorithm 3. We start by entering a loop that ends when all vertices have been coloured (Line 3). We broadcast the colours and use **flatMap** to transform a graph of **[Long, RoaringBitmap]** into a **[Long, Int]** RDD. By doing so, we are reusing the RDD to return the colours of the graph. Then, at Line 30, we collect this RDD, update our colours array, decrease the number of remaining vertices and update the *currentMaxColor* variable.

Let us now describe what happens inside the **flatMap** primitive (Lines 8–29). We create an array of the colours of the neighbours of this vertex at Line 9, using the *currentMaxColor* variable. At Line 12, we look at the colour of this vertex. If the colour is 0, then this vertex is a *peasant* and not a knight. Therefore, this vertex can perform work. We iterate on the adjacency list of this vertex, and if we find a peasant in it, we set the *betterPeasant* variable to *true*. At Line 20, we test the *betterPeasant* variable to see if we are to become a knight. If true, we pick the first available colour in the neighbourhood using the *color* function. Finally, at Line 24, if the vertex becomes a knight, we return a single element array that contains its id and colour. Otherwise, we return an empty array.

### 3.3 | Generating test suites with distributed hypergraph covering

The second reduction of the  $t$ -way test suite generation problem to graphs proposed by Hallé et al. [6] involves the generation of a hypergraph  $H$  and the computation of a vertex covering for this hypergraph. More precisely, for given values of  $n, t$  and  $v$ , a set of  $v^n$  vertices is created for each combination of values of all  $n$  parameters; these vertices represent all the possible test cases. Each interaction of  $t$  parameters corresponds to one hyperedge of the graph, linking together all test cases (i.e., all vertices) that cover this interaction. Figure 7 gives an example for the case  $t=2, n=3, v=2$ . In this example, each hyperedge is identified with a number and each links exactly two vertices; for instance, hyperedge 0 corresponds to the interaction  $a=0, b=0$ .

We recall that a hypergraph vertex covering is a subset of the vertices of  $H$  such that each hyperedge has at least one of its vertices included in that subset. The coloured (non-white) nodes in Figure 7 represent a possible covering for this particular hypergraph.

**Theorem 2 From Hallé et al. [6]** *Let  $T$  be a vertex covering of the hypergraph  $H$  built as described. Then,  $T$  is a valid  $t$ -way test suite.*

Thus, a second possible way to distribute the calculation of a  $t$ -way test suite is by solving the hypergraph vertex covering problem in a distributed way. This is possible by implementing a distributed version of a greedy algorithm that selects at every iteration the vertex that covers the most hyperedges. To do this, we have to analyse the frequency of the vertices,

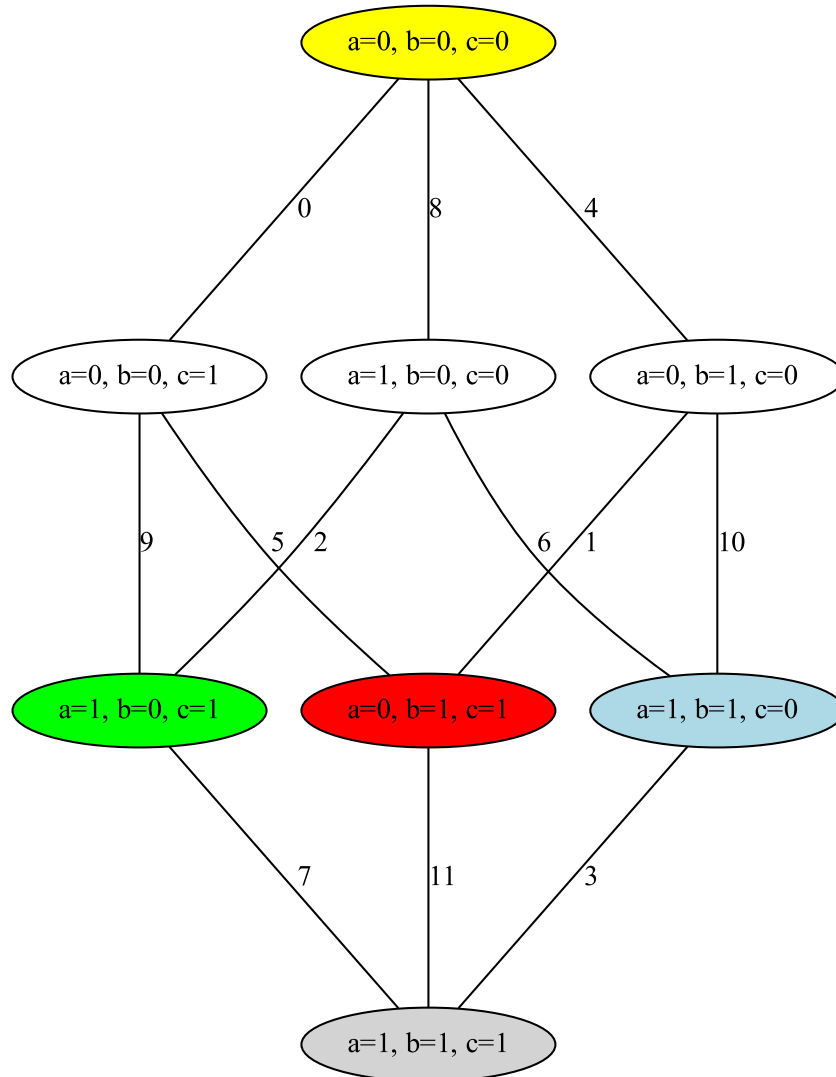


FIGURE 7 The hypergraph obtained from the  $t$ -way problem where  $t=2, n=3, v=2$

so we use the `flatMap` and `reduceByKey` primitives to implement a distributed count. Then, the driver program selects the most valuable test (vertex) and covers the interactions (hyperedges) covered by it using a `filter` primitive.

### 3.3.1 | The greedy picker heuristic

---

#### Algorithm 4: Greedy test picker.

---

```

input  : list, a list of the best tests
output : chosenTests, a list of chosen tests

1 Let sizeOfTests be the size of a test, in length of characters. Let differenceScore be a function that compares two
  tests and returns a score of difference
2 if list contains only one test then
3   | return list(0)
4 end
5 chosenTests += list(0)
6 for every element i of the list, starting from the second element do
7   | thisTest ← list(i)
8   | for every element j of the chosenTests do
9     | oneofthechosen ← chosenTests(j)
10    | difference ← differenceScore(thisTest, oneofthechosen)
11    | diff ← difference/sizeOfTests * 100
12    | if diff < 40.0 then
13      | found ← false
14      | return
15    | end
16  | end
17  | if found = true then
18    | chosenTests += thisTest
19  | end
20 end
21 return chosenTests

```

---

To perform fewer distributed iterations, we use a heuristic named *greedy picker* to select several tests in one iteration, instead of one.

We describe this algorithm using Algorithm 4. First of all, the algorithm receives a parameter called *list*—a list that contains every test with the maximum count. We select the first test in this list. Then, at Lines 6–19 we iterate with two loops over the list of tests, and we pick a test *t* when *t* has an adequate difference compared to all the chosen tests. Thus, the second test is picked when it is sufficiently different from the first, and the third test is picked when it passes the difference test with the first test and the second test. The *differenceScore* function returns a number equal to the number of times a parameter value differs in the interactions.

Figure 8 shows the execution of the hypergraph covering algorithm on a small *t*-way problem:  $t = 2, n = 3, v = 2$ . This example was generated by logging an actual execution of the algorithm. For demonstration purposes, we use only two partitions, but this number can be different. First, at the initial state, every hyperedge is a *t*-way interaction. At Step 1, we use the `mapPartitions` primitive to generate the hyperedge’s vertices—the tests—by enumerating them on-the-fly. These new tests are counted locally using a hash table. Finally, we turn these hash tables back into arrays and use the `reduceByKey` primitive to perform the final aggregation phase.

In Step 2, `reduceByKey` shuffles the keys to the proper partition using the modulo operator on the value provided by a JVM object’s `hashCode()`. Then, the best tests from each partition are sent back to the driver program. There, a diverse subset of the best tests is chosen by the greedy picker algorithm (Algorithm 4). The greedy picker algorithm receives a list of the best tests (highest number of votes) and does the following: It selects the first test and puts it in a list called ‘chosen tests’. Then, it iterates through the rest of the list and if a given test is at least 60% different from all the previous chosen tests, it is added onto the list of chosen tests. Otherwise, the test is skipped. The 60% value was calibrated through experimentation; it yielded both speed and quality in our tests. Finally, using the list of chosen tests, we delete hyperedges. The algorithm ends when all the interactions have been covered.

### 3.3.2 | The full algorithm

We can now describe Algorithm 5, the main algorithm of the hypergraph vertex cover. The algorithm expects two parameters: the RDD of *t*-way interactions and a *vstep* parameter that limits the number of tests we can pick from in one iteration. Setting *vstep* at 1 will not allow the algorithm to use the *greedy picker* algorithm.



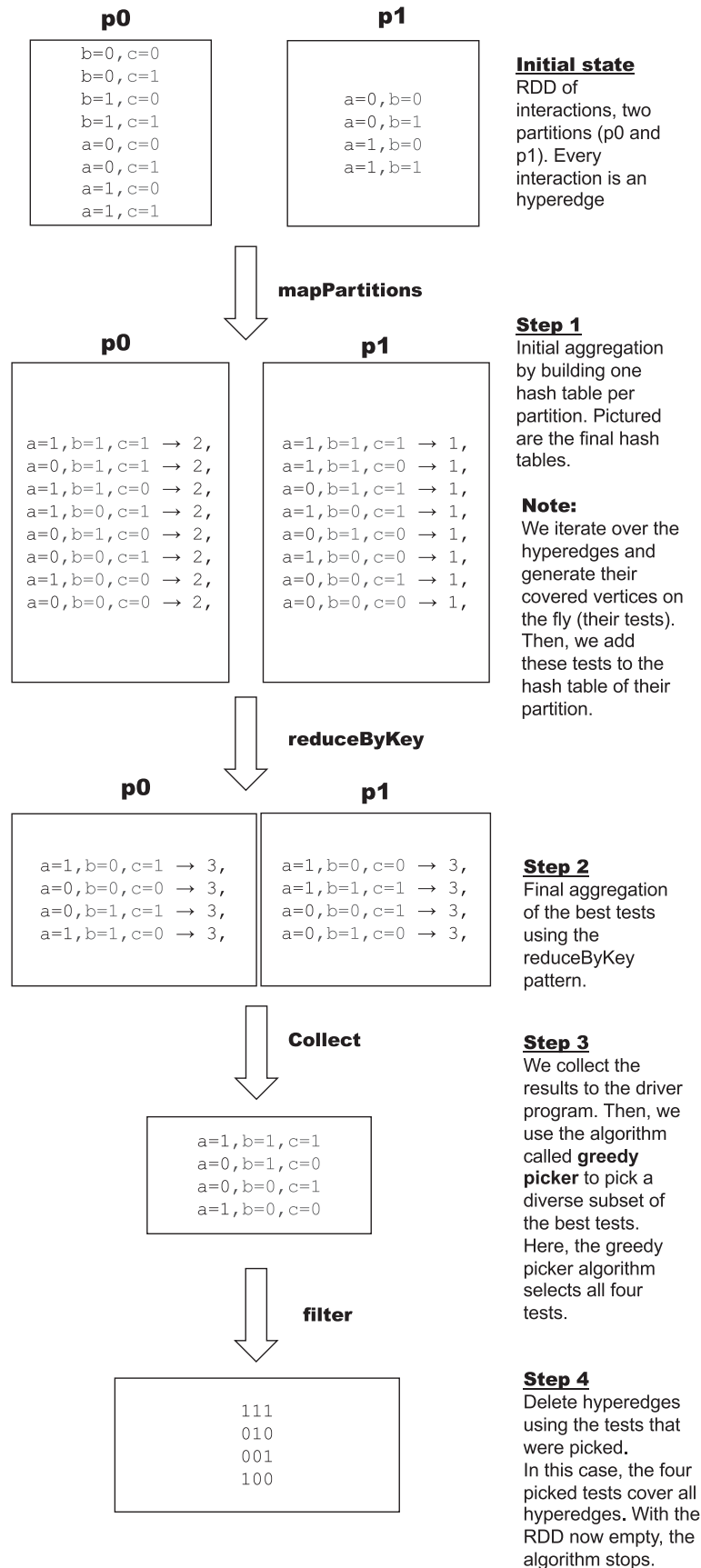


FIGURE 8 Hypergraph covering example with CA(4;2;3;2)

In this algorithm, we reuse the same RDD in a loop, and we can do this for many iterations. For this reason, we use Apache Spark's `localCheckpoint` to reset the DAG of the RDD; otherwise, we would get a `StackOverflow` error. We do this at Line 11. At Line 14 is our *exit condition*; when the RDD is empty we can exit. At Lines 18–32, we perform a local aggregation using hash tables. We generate the list of possible tests from an interaction, and we enter the results into the hash table. At Lines 34–39, we perform the final aggregation using the *reduceByKey* primitive, and we select a certain number of the best tests. Our default value is to allow this number to be quite large, at 1% of the number of total interactions. Then, we enter these tests into the *greedy picker* algorithm, and we get a set of diverse tests in return. We delete hyperedges from the RDD using these tests and we update the test suite variable *chosenTests*.

---

**Algorithm 5:** Hypergraph Vertex Cover.
 

---

```

input : rdd: an RDD of  $t$ -way interactions, vstep: the covering speed
output : tests: a list of tests

1 chosenTests  $\leftarrow$  []
2 currentRDD  $\leftarrow$  rdd
3 counter  $\leftarrow$  0
4 if vstep = -1 then
5   | maxPicks  $\leftarrow$  rdd.size / 100
6 else
7   | maxPicks  $\leftarrow$  vstep
8 end
9 while true do
10  | counter  $\leftarrow$  counter+1
11  | if counter %3 == 0 then
12  |   | currentRDD.localCheckpoint()
13  | end
14  | if currentRDD.isEmpty() then
15  |   | break
16  | end
17
18  | hashTablesRDD  $\leftarrow$  currentRDD.mapPartitions( partition =>
19  | begin
20  |   | hashmap  $\leftarrow$  new
21  |   |   | hashmap(String, Int)
22  |   |   | partition.foreach(interaction =>
23  |   |   | begin
24  |   |   |   | list  $\leftarrow$  interactionToTests(interaction)
25  |   |   |   | for each key in list do
26  |   |   |   |   | if key exists then
27  |   |   |   |   |   | hashmap(key)  $\leftarrow$  hashmap(key)+1
28  |   |   |   |   | else
29  |   |   |   |   |   | hashmap(key)  $\leftarrow$  1
30  |   |   |   |   | end
31  |   |   |   | end
32  |   |   | end
33  |   |   | return hashmap
34  |   | end
35  |   | testsCounts  $\leftarrow$  hashTablesRDD.flatMap( hashtable => hashtable.toArray)
36  |   | testsCounts  $\leftarrow$  testsCounts.reduceByKey( (a,b) => a+b))
37  |   | bestTests  $\leftarrow$  selectBestTests(testCounts, maxPicks)
38  |   | diverseTests  $\leftarrow$  greedyPicker(bestTests)
39  |   | currentRDD  $\leftarrow$  coverInteractions(currentRDD, diverseTests)
40  |   | chosenTests  $\cup$  diverseTests
41  | end
42 return chosenTests

```

---

### 3.3.3 | Updating the distributed state

When Greedy picker chooses tests, these tests are then used to remove  $t$ -way interactions from the state—which is contained in a RDD. Our previous approach was to broadcast the chosen tests and use string comparisons to see whether or not an interaction was covered or not. Now, our approach is to use set manipulations, just like in Algorithm 2. We

first build a database using the set of tests that are to be deleted. Then, every individual interaction performs set manipulations to build the list of invalid tests. After doing this, if the list is not empty, which means that there is a test that covers this interaction, we delete the interaction from the RDD using the `filter` primitive.

### 3.4 | Generating test suites with distributed IPOG (D-IPOG)

So far, we considered distributed graph algorithms that attempt to directly solve the  $t$ -way problem in a distributed way, by devising distributed versions of algorithms operating on two possible graph-based reductions of the problem. An alternate way of distributing computation is by introducing distribution in the operation of an existing algorithm. We consider in this section the case of IPOG.

As we already mentioned, the IPOG algorithm is a greedy algorithm that generates a covering array to a  $t$ -way problem by covering one parameter at a time. There are several ingredients needed for an IPOG algorithm: a routine that enumerates interactions for the current parameter only, a routine that adds an optimized column of values to the test suite (the *horizontal growth* phase) and a routine called *vertical growth* that adds tests to cover the interactions that were not covered by the horizontal growth phase.

Thus, to create a D-IPOG algorithm, all three of these parts must be distributed. As of now, we have shown that the enumeration of interactions can be distributed (Section 3.1), and we have also shown that our distributed graph-based algorithms (Sections 3.2 and 3.3). By distributing the horizontal growth phase and reusing these graph algorithms, we can create such an algorithm. We will now proceed to the description of the distributed horizontal growth algorithm.

#### 3.4.1 | Distributed horizontal growth

Horizontal growth is considered to be one of the most important parts of the IPOG algorithm. This is because adding an optimized column of values to an existing test suite is usually what covers the most  $t$ -way interactions. Single-threaded algorithms such as those found in IPOG or in a routine like BestExtendCA [51] are all effective at doing this. The general logic of these methods is the following:

1. Try all possible values for parameter  $j$  of test  $i$ . Consult global state every time to see how many interactions are covered
2. Pick the value that covers the most interactions. Remove these interactions from the state. Go to the next test and do the same thing
3. If no value covers any interactions, add a *don't-care* value

When thinking of distributing this horizontal growth algorithm, one can notice that this algorithm performs an iteration for every test in its test suite. This is not a problem for a single-threaded algorithm, but it is a major problem for a distributed algorithm: doing a hundred thousand *distributed* iterations is far from ideal. Therefore, we can adjust this algorithm for the distributed world by making it improve *several tests* at every iteration. This might come at the cost of some solution quality however, because we are no longer making the same quality of greedy decisions. Our code's default behaviour is to extend the test suite in 100 iterations ( $hstep = 100$ ).

Figure 9 shows the execution of the horizontal growth algorithm for the fourth parameter of the problem  $t = 2, n = 4, v = 2$ . First, we start with the  $t$ -way interactions from the enumerator. Then, we extend these tests using four iterations of the algorithm. At every iteration, we modify the test suite and remove covered interactions. After horizontal growth, there are two interactions remaining. We add any incomplete tests to the interactions and send everything to the vertical growth algorithm. The graph colouring algorithm colours the graph using two colours and returns a test suite that is merged with the existing one.

Algorithm 6 details how the horizontal growth algorithm works. At Line 11, the algorithm enters a loop that exits once all the tests have been covered. We select a number of tests according to the covering speed. Then, these tests are broadcasted to the cluster (Line 19). At Line 20, we perform our local aggregation using a hash table. This time, we are using a hash table in which the key is *composite*. The key tracks the id of the test and its version (using the additional value) using a type of [Int,Char]. So, for instance, if our third test is  $a = 0, b = 0$ , then our key will be [3, '0'] or [3, '1']. Every  $t$ -way interaction of the RDD then submits a vote to the hash table when it can be covered by a version of one of the tests. At Line 21, we perform the final aggregation, and we obtain the counts for every test version. At Line 22, we use the cluster to choose the best version. To do this, we adjust the key using `map` and perform the full aggregation using `reduceByKey`. At Line 31, we grow our test chunk by adding the best parameter we have calculated for each.

At Lines 32–33, we take any test that did not grow during this iteration of horizontal growth. To do this, we compare our test chunk against the *bestVersions* tests. If a test is missing in *bestVersions*, it means that the test was not able to cover any interaction during the initial hash table aggregation. Therefore, we keep the test and add a new parameter with a don't-care value (\*) to indicate the presence of available space. By doing this, future iterations of graph colouring will try to recover that space.

---

**Algorithm 6:** Horizontal Growth.
 

---

```

input : rdd the RDD of  $t$ -way interactions, tests the test suite,
         hstep the covering speed, sc the SparkContext
output : finalTests, and uncoveredInteractions, the RDD of uncovered interactions

1 finalTests  $\leftarrow$  []
2 currentRDD  $\leftarrow$  rdd
3  $i \leftarrow 0$ 
4  $m \leftarrow$  tests.size/100
5 if  $m < 1$  then
6   |  $m \leftarrow 1$ 
7 end
8 if hstep  $\neq -1$  then
9   |  $m \leftarrow$  hstep
10 end
11 while true do
12   | if  $i >$  tests.length then
13     | break
14   | end
15   | someTests  $\leftarrow$  pickM(tests, m, i)
16   | if someTests.size  $<$  m then
17     |  $m \leftarrow$  someTests.size
18   | end
19   | testsbcast  $\leftarrow$  sc.broadcast(someTests)
20   | hashTablesRDD  $\leftarrow$  initialHashAggregation(testsbcast, currentRDD)
21   | testsCounts  $\leftarrow$  hashTablesRDD.flatMap( hashtable  $\Rightarrow$  hashtable.toArray).reduceByKey( (a,b)
     |  $\Rightarrow$  a+b)
22   | bestVersionsRDD  $\leftarrow$  testCounts.map( e  $\Rightarrow$  (e.test, (e.version, e.votes))).reduceByKey( (a,b)  $\Rightarrow$ 
23   | begin
24   |   | if a.votes  $>$  b.votes then
25     |   | return a
26     |   | else
27     |   | return b
28     |   | end
29   | end
30   | bestVersions  $\leftarrow$  bestVersionsRDD.collect()
31   | grownTests  $\leftarrow$  addBestParamValue(tests, bestVersions)
32   | didNotGrow  $\leftarrow$  findDNG(tests, bestVersions)
33   | grownTests  $\leftarrow$  grownTests  $\cup$  addStar(didNotGrow)
34   | currentRDD  $\leftarrow$  progressivefilter(currentRDD, grownTests)
35   | finalTests  $\leftarrow$  finalTests  $\cup$  grownTests
36   | currentRDD.localCheckpoint()
37   |  $i \leftarrow i+m$ 
38 end
39 return finalTests, currentRDD

```

---

### 3.4.2 | Notes on vertical growth

The vertical growth algorithm handles the  $t$ -way interactions not covered by horizontal growth. It does so by adding new tests, but it can also reuse the 'don't-care values' inside the existing tests. We therefore take the tests that contain these values, along with the uncovered interactions, and invoke a vertical growth algorithm.

### 3.4.3 | Relation to previous work

Prior to this algorithm is an approach called MC-MIPOG [8], which is a parallel version of MIPOG [27]. We refer the reader to Section 2.2 for more details about MC-MIPOG. The main difference about MC-MIPOG and our approach

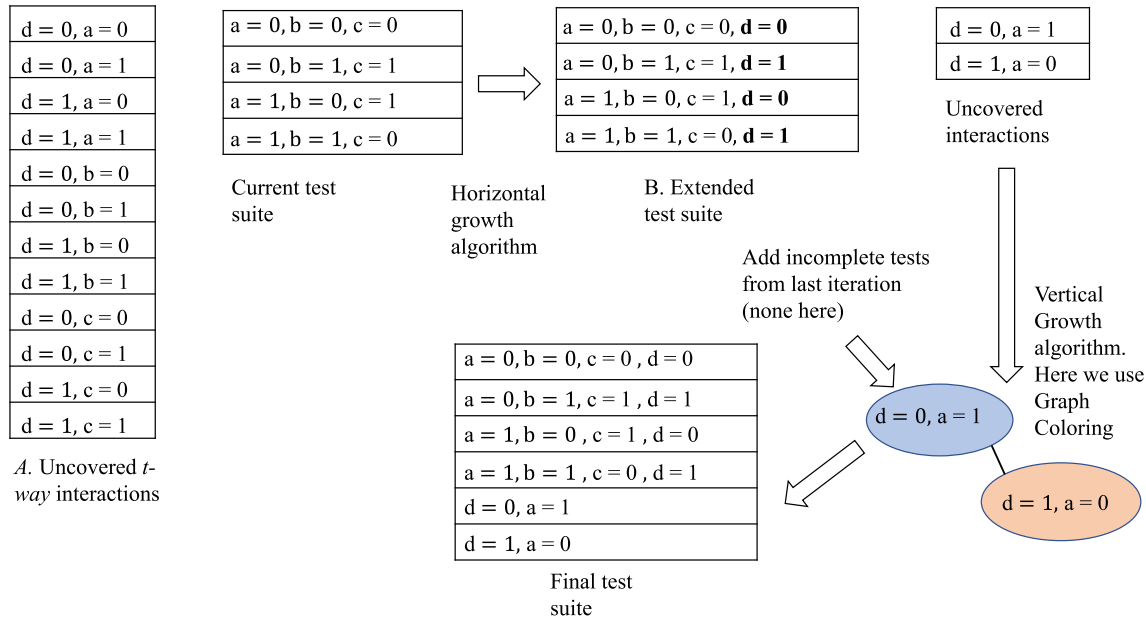


FIGURE 9 Horizontal growth, fourth parameter.

(beside the significant difference between parallel and distributed) is that MC-MIPOG performs more work during its horizontal growth and less work during the vertical growth. During horizontal growth, if a given test contains *don't-care values*, MC-MIPOG tries to fill the space with uncovered interactions. However, during vertical growth, MC-MIPOG classifies the remaining interactions into a number of partitions equal to the current parameter's domain size and then launches its vertical growth algorithm on each partition. The partial test suites are then merged.

### 3.5 | Discussion

This section presented the broad lines of two reductions to graph-based problems to generate combinatorial test suites, assuming no constraints on test cases and uniform covering arrays. It is worth noting that previous work on the topic [6] have shown how these two reductions can actually handle a more general problem called  $\Phi$ -way test case generation, in which each parameter can have an arbitrary domain size (i.e., mixed covering arrays), and where constraints can be expressed which must either apply on every test case (*universal constraints*) or for at least one test case (*existential constraints*). The handling of this more general problem requires straightforward modifications in the first step of the procedure, which generates the graph or hypergraph for a given test case generation problem. However, once this graph or hypergraph is obtained, the distributed colouring and vertex cover algorithms presented here can be applied directly. While this topic is not the main focus of our paper, it is relevant to mention in the context of our discussion.

The use of parallel techniques to generate covering arrays has been previously explored by researchers. One such approach is the application of the Simulated Annealing (SA) metaheuristic on a GPU, as reported by Mercan et al. [52]. The results demonstrated that the parallelization of certain computationally intensive operations (checking constraints and calculating the fitness) resulted in significant improvements in speed, with up to a 13-fold increase in speed when the solution space is larger, at which point the overhead of preparing a GPU computation is very little compared to the gain in processing power. A cluster computation is similar in that it offers even more processing power (and memory) but has a lot more overhead than the GPU because of network latencies.

In Calvagna et al. [53], the proposed method is similar to graph colouring, but without the transformation into a graph. Instead of a graph colouring order, the approach involves finding a merge sequence that produces a small test suite. To find such a merge sequence, multiple instances of the algorithm are run on multiple machines using grid computing [54], and the smallest test suite is retained. This approach differs significantly from the proposed distributed graph colouring method presented in this work, which involves distributing the graph data structure, graph building process and graph colouring across multiple computers. In Calvagna et al. [55], the same method as previously described is employed, but the problem space is distributed across a cluster using MPI [56] primitives. This is achieved by randomly dividing the  $t$ -way interactions into  $N$  partitions using a modulo operation. Within each of these partitions, a local incomplete test suite is assembled by merging compatible  $t$ -way interactions. These incomplete test suites



TABLE 1 Test suite sizes on sparse graphs.

$n$	K&P	ACTS	CAGEN	D-IPOG CLUSTER	D-IPOG PARALLEL
100	<b>13</b>	16	16	19	17
200	<b>15</b>	18	18	20	18
400	<b>16</b>	20	20	21	20
800	<b>18</b>	22	22	25	25
1600	<b>19</b>	24	24	28	28
3200	<b>21</b>	26	26	32	31

Note:  $t = 2$ ,  $v = 2$ . The bold cell on each line corresponds to the smallest (i.e., best) result obtained for this particular value of  $n$ .

are then merged to form a valid test suite. This approach differs from our distributed graph colouring method in that it involves colouring multiple subgraphs using a single-threaded algorithm and then merging them into a single final graph. In contrast, our method involves a single graph that is coloured using the distributed K&P algorithm.

## 4 | IMPLEMENTATION AND EXPERIMENTS

In this section, we proceed to describe an implementation of these algorithms into TSPARK, a distributed test case generator that uses the Apache Spark cluster computing framework as our distributed technology of choice. The source code for TSPARK, along with the shell script we used for running TSPARK on a cluster are all publicly available.<sup>2</sup>

TSPARK ships as a stand-alone JAR file of about 140 MB, which contains a working installation of Apache Spark 3.0.0. This JAR can then be tested on any Apache Spark cluster using the `spark-submit`<sup>3</sup> command when the cluster is live. Otherwise, TSPARK can be used locally by invoking the `java-jar tspark.jar` command. Used in this way, TSPARK detects that it is being used locally and creates a virtual cluster for the duration of the computation. Our implementation of TSPARK in Scala contains around 10,000 lines of code, and its repository also includes the script we used to run the experiments on the SLURM cluster manager [57]. TSPARK is currently in the alpha stage of development and as such, it does not offer all of the features that a more mature tool would have. At this time, TSPARK is only capable of generating solutions for problems that have a uniform number of values per parameter. We plan to expand the capabilities of TSPARK in the future.

### 4.1 | Experimental setup

We have designed our experiments around one central idea: to compare the performance of our distributed algorithms to existing high-performance tools, namely, ACTS and CAGEN. These tools all use the IPOG algorithm or variants of it. Concerning the D-IPOG implementations introduced in this paper, we study two variants: D-IPOG CLUSTER is the Apache Spark implementation that runs on the cluster, while D-IPOG PARALLEL is an implementation that runs on a single machine and uses Scala Parallel Collections [58] instead of Apache Spark.

To produce the test suites of TSPARK, we have used the computer clusters provided by Compute Canada,<sup>4</sup> a distributed cluster infrastructure for research. The cluster we have used is called Niagara and its technical specs are detailed in Compute Canada's user documentation.<sup>5</sup> This cluster is shared with other researchers; hence, we used only a fraction of its processing power. When running our jobs, we used 20 full nodes, which represents 800 processors and 4 TB of RAM. Since ACTS, CAGEN and D-IPOG PARALLEL are not distributed, their results were produced using a recent workstation computer equipped with an eight-core AMD Ryzen 7 5800X processor and 16 GB of RAM. For ACTS and D-IPOG PARALLEL, the JVM flag `-Xmx` has been used; it enables a process to access more memory than the default values.

In a first set of experiments, we compare the performance of graph colouring to IPOG implementations on sparse graphs (Tables 1 and 2). Sparse graphs are created by  $t$ -way problems in which the number of parameters  $n$  is large compared to  $t$  and  $v$ . Otherwise, when the values of  $t$  and  $v$  are larger, there are more conflicts (edges) between the interactions, and the resulting graph is denser. To comprehend the amount of memory it takes to store a graph, we calculate

<sup>2</sup><https://github.com/mitchi/TSPARK>.

<sup>3</sup><https://spark.apache.org/docs/latest/submitting-applications.html>.

<sup>4</sup><https://computecanada.ca>.

<sup>5</sup>The Niagara cluster: <https://docs.computecanada.ca/wiki/Niagara>.

TABLE 2 Generation times on sparse graphs.

$n$	K&P	ACTS	CAGEN	D-IPOG CLUSTER	D-IPOG PARALLEL
100	58 s	0,s	0,s	431 s	3 s
200	116 s	0,s	0 s	960 s	5 s
400	253 s	0,s	0 s	2066 s	10 s
800	747 s	1,s	0 s	4602 s	25 s
1600	3358 s	5,s	0 s	10,381 s	80 s
3200	23,278 s	34,s	1 s	23,494 s	364 s

Note:  $t=2, v=2$ .

TABLE 3 Additional information.

$n$	Number of vertices	Graph size	Percentage of iterations made by K&P
100	19,800	0.02 GB	3.6%
200	79,600	0.3 GB	1.90%
400	319,200	6.3 GB	0.97%
800	1,278,400	102 GB	0.50%
1600	5,116,800	1.6 TB	0.288%
3200	20,473,600	26 TB	0.15%

Note: Graph size is in gigabytes (GB) or terabytes (TB).

TABLE 4 Test suite sizes on dense graphs.

$n$	Number of vertices	Hypergraph	ACTS	CAGEN	D-IPOG CLUSTER	D-IPOG PARALLEL
8	131,072	23,255	<b>16,384</b>	<b>16,384</b>	26,422	24,921
9	589,824	<b>33,462</b>	39,296	38,424	39,347	38,021
10	1,966,080	<b>41,387</b>	51,636	48,480	50,879	49,734
11	5,406,720	<b>49,059</b>	58,952	59,048	62,049	60,977
12	12,976,128	TL	<b>65,882</b>	68,927	72,533	71,705
13	28,114,944	TL	<b>72,941</b>	78,226	82,472	81,723
14	56,229,888	TL	<b>81,412</b>	86,926	91,819	91,179
15	105,431,040	TL	<b>88,885</b>	95,108	100,586	100,091
16	187,432,960	TL	<b>95,700</b>	102,630	108,879	108,533
17	318,636,032	TL	<b>102,430</b>	109,668	116,692	ML

Note:  $t=7, v=4$ . The bold cell on each line corresponds to the smallest (i.e., best) result obtained for this particular value of  $n$ . Abbreviations: ML, memory limit; TL, time limit (24 h).

the memory it takes using half of the graph matrix (the smallest amount of graph data that can colour a graph) and a bit set data structure. We use the following formula to calculate the size in gigabytes:  $\binom{n}{t}v^t/1000^3/8/2$ .

In all experiments, D-IPOG used a standard setting of a maximum of 100 distributed iterations to perform its horizontal growth and used a single-threaded graph colouring algorithm to perform its vertical growth. Random tiebreakers were used for K&P.

In the second set of experiments, we compare the performance of hypergraph vertex cover to IPOG algorithms on dense graphs (Tables 4 and 5). In the third set of experiments (Table 6), we compare D-IPOG to MC-MIPOG [8], a parallel MIPOG algorithm. We also include ACTS and CAGEN for comparison. As MC-MIPOG is not available, we could only compare test suite sizes. Finally, we run a last set of experiment with Hypergraph, ACTS, CAGEN and D-IPOG-Parallel to investigate the value of extending a test suite produced by Hypergraph with a faster IPOG-based tool (Table 7).

TABLE 5 Generation times on dense graphs.

$n$	Number of vertices	Hypergraph	ACTS	CAGEN	D-IPOG CLUSTER	D-IPOG PARALLEL
8	131,072	162 s	0 s	0 s	63 s	3 s
9	589,824	259 s	3 s	0 s	146 s	11 s
10	1,966,080	601 s	6 s	0 s	266 s	29 s
11	5,406,720	5540 s	10 s	1 s	548 s	66 s
12	12,976,128	TL	16 s	2 s	1060 s	141 s
13	28,114,944	TL	33 s	4 s	1974 s	291 s
14	56,229,888	TL	63 s	8 s	2623 s	555 s
15	105,431,040	TL	131 s	15 s	4589 s	1046 s
16	187,432,960	TL	280 s	30 s	8438 s	1885 s
17	318,636,032	TL	483 s	60 s	14,015 s	ML

Note:  $t = 7, v = 4$ .

Abbreviations: ML, memory limit; TL, time limit (24 h).

TABLE 6 Size comparison with MC-MIPOG.

$t$	MC-MIPOG	ACTS	CAGEN	D-IPOG
2	45	48	45	50
3	281	308	305	330
4	1643	1843	1702	1986
5	8169	10,119	10,228	10,659
6	45,168	50,920	50,687	53,296
7	186,664	232,142	244,290	242,444

Note:  $t = 2$  to 7; 10 parameters and five values.

## 4.2 | Results

Two aspects of the performance were considered: the ability of a tool to generate *small* test suites for a given problem, and the *computing time* required to produce such a test suite. The results presented in Tables 1–5 indicate that while graph colouring and hypergraph vertex covering produce better results (i.e., smaller test suites) on very sparse and dense graphs (15% to 20% better), their computation time using a cluster setup is not competitive with the IPOG algorithms. The D-IPOG algorithm is also slower. The major drawback that the distributed algorithms suffer from is the time it takes to perform a *distributed iteration*. Serialization/deserialization of data structures, network transfer times and framework-related slowdowns are some of the suspected culprits.

Apache Spark was made for processing batches of data, so it might not be adapted to algorithms which require thousands of iterations. We can observe this by looking at the time it takes D-IPOG to compute  $n = 17, t = 7, v = 4$  (Table 4) and  $n = 3200, t = 2, v = 2$  (Table 1). While the first problem represents a much bigger graph, D-IPOG is able to process it in less time because there are fewer distributed iterations to perform overall.

To investigate this further, we took our code and replaced the Apache Spark calls with code using Scala's Parallel Collections [58]. Doing this only took a few modifications. Then, we ran the tests again, and we obtained the tests called D-IPOG PARALLEL. We can see in Table 4 that D-IPOG PARALLEL, with eight cores, is faster than the cluster on every single result, except  $n = 17$  because of an **OutOfMemory** error thrown by the Java Virtual Machine. These results indicate that while the cluster handles large IPOG problems without too much inconvenience, there is a large constant time associated with each distributed iteration.

Nevertheless, these results also demonstrate positive findings. First, the sparse graphs results (Table 1) show that graph colouring produces better results than IPOG on this kind of graph. For comparison purposes with the best known results, Colbourn [59] reports results of 15 for  $n = 3003$  and 16 for  $n = 6435$ . Having access to an algorithm that produces smaller test suites for sparse graphs can be valuable, because  $t$ -way testing has been proven effective at lower interaction strengths. A valid criticism of graph colouring is that it requires a lot of power and storage to create a graph, store its data structures and colour it. To remedy this, we propose building the graph using a faster algorithm (Database Graph Construction) and using a data structure with integer compression to store the graph

(RoaringBitmap for instance). Also, in Table 3, we can see the percentage of iterations made by the algorithm to colour the graph. This percentage is calculated by dividing the number of distributed iterations with the number of vertices in the graph. We can notice that as the graph becomes sparser, the algorithm becomes more effective.

Additionally, the dense graphs results (Table 4) show that hypergraph vertex cover produces better test suite sizes than IPOG. Again, solving an hypergraph vertex covering problem requires more power, and so a distributed approach is interesting. We diminish the problem of distributed iterations by using the Greedy Picker heuristic, which selects several tests in the same iteration. In the  $t=7, n=10, v=4$  problem, we noticed in our tests that the use of this heuristic did not even decrease solution quality. Regardless, hypergraph vertex cover eventually becomes unusable because of the combinatorial explosion associated with a sparser problem.

The D-IPOG algorithm gets the worst results in most cases, but it is not too far from CAGEN on dense graphs (7% difference for  $n=17$ ). This is easily explained by the fact that when D-IPOG computes  $n=17$  from the  $n=16$  test suite, it is only doing 100 iterations to do it, while CAGEN does 102,630 iterations. We also compare the results of D-IPOG to those of MC-MIPOG [8] in Table 6. As we can see, D-IPOG's results are comparable to ACTS and CAGEN but not to MC-MIPOG, because of the algorithm difference. A comparison on running times would be interesting but cannot be done because MC-MIPOG is not available.

We also investigated the merits of using an algorithm like hypergraph vertex cover to create a test suite that an IPOG-based tool would then extend. The results of this short experiment can be seen in Table 7. Each tool was seeded with a test suite of 41,396 tests produced by Hypergraph with the  $n=10, t=7, v=4$  problem. Initially, our hypothesis was that starting with a good test suite would be advantageous, especially when that test suite is 20% smaller than the equivalent ACTS test suite at  $n=10$ . As we can see, this does not seem like a useful strategy for ACTS, as it only improves one result, and its end result, for  $n=16$ , is 6% worse. CAGEN produces nearly identical results and D-IPOG only improves its results by a small margin.

### 4.3 | Discussion

The experimental results presented above lead to mixed conclusions. On the one hand, they reveal the relatively good standing of the graph-based reductions introduced by Hallé et al. [6] (and especially the hypergraph reduction) to generate test suites that are of size comparable or smaller to state-of-the-art tools. Unfortunately, they do so at the price of much larger resource consumption, especially in terms of computation time, due to the number of distributed iterations they require and the seemingly important overhead caused by these iterations. Distributed variants of IPOG, using graph algorithms under the hood for their expansion phases, fare better in this respect.

However, we also argue that sheer running time is not the only measure by which each tool must be compared and that this should be balanced with the quality (i.e., size) of the solutions that are generated. After all, the whole point of generating a test suite is to eventually *run* it on a SUT; the additional time taken to generate a solution may well be compensated by the fact that running a smaller test suite takes less time afterwards.

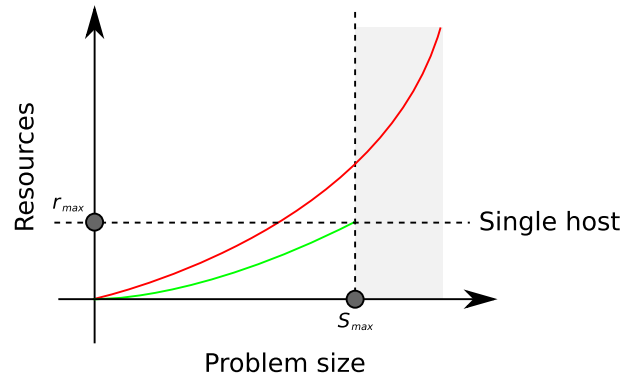
There exist multiple cost models for test input generation; we follow Garvin et al. and concentrate on a derived metric called the amortized *time per test* [60]. Suppose that two tools  $A$  and  $B$  produce a test suite of size  $s_A$  and  $s_B$  in time  $t_A$  and  $t_B$ , respectively, with  $s_A > s_B$  and  $t_A < t_B$ . In other words,  $A$  is faster than  $B$  but produces a larger test suite. The time per test between  $A$  and  $B$ , noted  $\tau(A, B)$ , is defined as

$$\tau(A, B) \triangleq \frac{t_B - t_A}{s_A - s_B}.$$

TABLE 7 Extending a test suite of 41,396 tests produced by Hypergraph.

$n$	ACTS	ACTS+	CAGEN	CAGEN+	D-IPOG	D-IPOG+
11	58,952	57,163	59,048	581,814	60,977	58,898
12	65,882	67,902	68,927	68,887	71,705	70,019
13	72,941	77,260	78,226	78,266	81,723	80,249
14	81,412	85,907	86,926	86,897	91,179	89,922
15	88,885	93,928	95,108	95,077	100,091	98,921
16	95,700	101,481	102,630	102,650	108,533	107,467
17	102,430	108,525	109,668	109,736	ML	ML

Note: The + character signifies that the tool is used in extend mode.  
Abbreviation: ML, memory limit.



**FIGURE 10** A plot showing the intrinsic limitation of an algorithm that does not scale beyond a single machine (green line) versus a distributed algorithm (red line)

The result, expressed in units of time per test, corresponds to the minimum duration of a test case, such that  $B$ 's slower generation time is compensated by the fact that the test suite takes a shorter time to run (what Garvin *et al.* call the 'break-even point'). For example, if  $A$  takes 3 s to generate a test suite of size 20, and  $B$  takes 10 s to generate a test suite of size 10, then the total time taken by  $B$  to generate a test suite *and* run it becomes shorter than  $A$  as soon as a single test takes more than  $\tau(A, B) = 0.7$  s to run. As one can see, this metric takes into account the whole generation and execution process and not just test suite generation time.

Equipped with such a metric, we can now revisit the results described earlier and examine them in a different light. Let us take the problem where  $n = 10$ ,  $t = 7$ ,  $v = 4$ . Hypergraph takes ten minutes to generate a test suite, while IPOG takes a mere 6 s. However, Hypergraph creates a suite that contains 10,000 fewer test cases. In this case,  $\tau(\text{IPOG}, \text{D-Hypergraph}) \approx 0.058$ ; this means that as soon as a test case takes on average more than 58 ms to run, the longer waiting time for Hypergraph is completely paid off by the shorter execution of the resulting test suite. This time can even be amortized further if the test suite is executed more than once, which is typically the case.

Finally, we shall stress a final, more theoretical point: any algorithm that is not designed to scale beyond a single machine is bound to reach a point where larger problems cannot be handled. This is illustrated by the green line in the plot of Figure 10, where more resources are consumed for larger problem sizes, until the resource limit of a single host (time and/or memory),  $r_{\max}$ , is reached;  $s_{\max}$  is therefore the largest problem that can be handled by the algorithm. In contrast, the red line shows the resource consumption of another, distributed algorithm. Despite showing a lower performance (i.e., a higher resource consumption), it becomes the only available solution for problems of size greater than  $s_{\max}$ . What our experiments reveal, in the current state of things, is that this 'final frontier' is further away than we expected.

## 5 | CONCLUSION

In this paper, we investigated using distributed computing for combinatorial (' $t$ -way') test suite generation. In our previous work, we saw the promise of transforming a  $t$ -way problem instance into a graph problem for generating small test suites. However, in order to leverage the algorithms that solve these graph problems, a lot of computing power and memory is required; this motivated the development of distributed versions of graph algorithms to solve the problem, and so we turned our attention to distributed computing. Although distributed computing is mainly used to transform and analyse massive amounts of business or user data, it can also be used for any other massive calculation, as it allows one to leverage thousands of CPU cores and terabytes of RAM.

We first turned our attention on the reduction of  $t$ -way test suite generation to graph colouring. We introduced a faster algorithm called Database Graph Construction to build the graph from a  $t$ -way problem followed by the K&P algorithm to colour it. These two algorithms are both original contributions. Second, we presented our approach to handle the reduction to hypergraph vertex covering in a distributed manner. To further accelerate this algorithm, we introduced a heuristic called Greedy Picker. In our experimental results, we observed that this heuristic quickened the computation while maintaining the same solution quality. Finally, we introduced a distributed version of the classical IPOG algorithm we call D-IPOG—a hybrid algorithm made from a distributed horizontal growth routine combined with either graph colouring or hypergraph vertex colouring as the vertical growth routine.



These algorithms were implemented in a publicly available tool called TSPARK and tested on a computer cluster provided by Compute Canada. Our experimental results indicate that these distributed algorithms cannot compete with the performance of the single-threaded IPOG implementations just yet. These findings go against the widespread belief that it suffices to distribute an algorithm to automatically improve its scalability. What we observed is that generating  $t$ -way test suites requires a large number of iterations, which, in a distributed setting, incur a high communication cost. Thus the paper puts the finger on an important problem that needs to be addressed in the future for  $t$ -way testing to be efficiently distributed. It should be understood that any improvement towards quicker communication will greatly benefit these techniques, as this is the biggest bottleneck, not sheer computing power or memory size.

These contributions, both at the theoretical and technical level, open the way to multiple avenues of research. First, the paper identified two subclasses of  $t$ -way problems (leading to either ‘sparse’ or ‘dense’ graphs) that are better suited to two graph reductions proposed in past literature. Experiments also hinted at the fact that these reductions produce test suites of smaller size than IPOG variants for these types of problems—but at the price of a steep overhead in CPU time. This nevertheless opens the way for future improvements by suggesting an adaptive approach where algorithms could be tailored specifically to particular ranges of values for  $t$ ,  $n$  and  $v$ .

Second, to improve iteration speed, our algorithms could be implemented on high-end graphic cards, as shown by the results of Mercan et al. [52] in which a gain of processing power was achieved while retaining significant iteration speed. Finally, it would be promising to leverage SIMD routines to accelerate some of our algorithms. For instance, the Database Graph Construction algorithm, currently implemented using Scala, applies certain logical operations (OR and NOT) with 64 bits instructions. Using SIMD programming would allow for 256 or 512 bits instructions.

## ACKNOWLEDGEMENTS

This work was supported by the Canada Research Chair on Software Specification, Testing and Verification and by the Natural Sciences and Engineering Research Council of Canada (Discovery Grant RGPIN-2016-05626).

## DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in TSPARK at <https://github.com/mitchi/TSPARK>.

## ORCID

Sylvain Hallé  <https://orcid.org/0000-0002-4406-6154>

## REFERENCES

- Kuhn D, Wallace D, Gallo A. Software fault interactions and implications for software testing. *IEEE Trans Softw Eng*. 2004;30(6):418–21. <https://doi.org/10.1109/TSE.2004.24>
- McCaffrey J. Pairwise testing with QICT. *MSDN Magazine*. 2009;29(12):28–35.
- Jenkins B. Jenny 2005, 2005; 2019. <https://burtleburtle.net/bob/math/jenny.html>; Accessed March 6, 2019.
- Yu L, Lei Y, Kacker R, Kuhn DR. ACTS: a combinatorial test generation tool. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, March 18–22, 2013*. Luxembourg: IEEE Computer Society; 2013. p. 370–5. <https://doi.org/10.1109/ICST.2013.52>
- Kampel L, Simos DE. A survey on the state of the art of complexity problems for covering arrays. *Theor Comput Sci*. 2019;800:107–24.
- Hallé S, Chance EL, Gaboury S. Graph methods for generating test cases with universal and existential constraints. In: El-Fakih K, Barlas GD, Yevtushenko N, editors. *Testing Software and Systems—27th IFIP WG 6.1 International Conference, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, November 23–25, 2015, Proceedings, Lecture Notes in Computer Science, vol. 9447*. Springer: Cham; 2015. p. 55–70. [https://doi.org/10.1007/978-3-319-25945-1\\_4](https://doi.org/10.1007/978-3-319-25945-1_4)
- Avila-George H, Torres-Jimenez J, Hernández V. Parallel simulated annealing for the covering arrays construction problem. In: Arabnia HR, Ishii H, Ito M, Joe K, Nishikawa H, Gravvanis GA, Solo AMG, editors. *PDPTA*. Las Vegas: CSREA Press; 2012. p. 522–8.
- Younis MI, Zamli KZ. MC-MIPOG: a parallel  $t$ -way test generation strategy for multicore systems. *ETRI J*. 2010;32(1):73–83.
- Klaib MF. A parallel tree based strategy for 3-way interaction testing. *Procedia Comput Sci*. 2015;65:377–84.
- Torres-Jimenez J, Izquierdo-Marquez I, Avila-George H. Methods to construct uniform covering arrays. *IEEE Access*. 2019;7:42774–97. <https://doi.org/10.1109/ACCESS.2019.2907057>
- Ahmed BS, Zamli KZ, Lim CP. Constructing a  $t$ -way interaction test suite using the particle swarm optimization approach. *Int J Innov Comput Inf Control*. 2012;8(1):431–52.
- Lei Y, Kacker R, Kuhn DR, Okum V, Lawrence J. IPOG: a general strategy for  $t$ -way software testing. In *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), March 26–29, 2007*. Tucson, AZ: EE Computer Society; 2007. p. 549–56. <https://doi.org/10.1109/ECBS.2007.47>
- Wallace DR, Kuhn DR. Failure modes in medical device software: an analysis of 15 years of recall data. *Int J Reliab Qual Saf Eng*. 2001;8(04):351–71.
- Kuhn DR, Reilly MJ. An investigation of the applicability of design of experiments to software testing. In *Proceedings 27th Annual NASA Goddard/IEEE Software Engineering Workshop*. Los Alamitos: IEEE; 2002. p. 91–5.
- Kuhn DR, Okum V. Pseudo-exhaustive testing for software. In *2006 30th Annual IEEE/NASA Software Engineering Workshop*. Los Alamitos: IEEE; 2006. p. 153–8.

16. Yuan X, Cohen MB, Memon AM. GUI interaction testing: incorporating event context. *IEEE Trans Softw Eng.* 2010;37(4):559–74.
17. Choi E, Mizuno O, Hu Y. Code coverage analysis of combinatorial testing. In: Lichter H, Fögen K, Sunetnanta T, Anwar T, Yamashita A, Moonen L, Mens T, Tahir A, Sureka A, editors. *Joint Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2016) and 1st International Workshop on Technical Debt Analytics (TDA 2016) co-located with the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016), Hamilton, New Zealand, December 6, 2016, CEUR Workshop Proceedings*, vol. 1771. Aachen: CEUR; 2016. p. 43–9. <https://ceur-ws.org/Vol-1771/paper7.pdf>
18. Kitsos P, Simos DE, Torres-Jimenez J, Voyiatzis AG. Exciting FPGA cryptographic Trojans using combinatorial testing. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015*. Gaithersbury, MD: IEEE Computer Society; 2015. p. 69–76. <https://doi.org/10.1109/ISSRE.2015.7381800>
19. Yang P, Tan X, Sun H, Chen D, Li C. Fire accident reconstruction based on LES field model by using orthogonal experimental design method. *Adv Eng Softw.* 2011;42(11):954–62.
20. Ma L, Zhang F, Xue M, Li B, Liu Y, Zhao J, Wang Y. Combinatorial testing for deep learning systems, 2018. CoRR, abs/1806.07723. <https://arxiv.org/abs/1806.07723>
21. Shasha DE, Kouranov AY, Lejay LV, Chou MF, Coruzzi GM. Using combinatorial design to study regulation by multiple input signals. a tool for parsimony in the post-genomics era. *Plant Physiol.* 2001;127(4):1590–4. <https://doi.org/10.1104/pp.010683>
22. Qi XF, Wang ZY, Mao JQ, Wang P. Automated testing of web applications using combinatorial strategies. *J Comput Sci Technol.* 2017;32(1):199–210. <https://doi.org/10.1007/s11390-017-1699-x>
23. Lei Y, Tai K. In-parameter-order: a test generation strategy for pairwise testing. In *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE'98), November, 13–14, 1998, Proceedings*. Washington, DC: IEEE Computer Society; 1998. p. 254–61. <https://doi.org/10.1109/HASE.1998.731623>
24. Microsoft. Pairwise independent combinatorial tool, 2019. <https://github.com/Microsoft/pict>, Accessed March 7, 2019.
25. Czerwonka J. Pairwise testing in the real world: practical extensions to test-case scenarios, 2008. <https://msdn.microsoft.com/en-us/library/cc150619.aspx>, Accessed August 9, 2022.
26. Wagner M, Kleine K, Simos DE, Kuhn R, Kacker R. CAGEN: a fast combinatorial test generation tool with support for constraints and higher-index arrays. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Los Alamitos: IEEE; 2020. p. 191–200.
27. Younis MI, Zamli KZ. MIPOG—an efficient t-way minimization strategy for combinatorial testing. *Int J Comput Theory Eng.* 2011;3(3):388.
28. Kleine K, Simos DE. An efficient design and implementation of the in-parameter-order algorithm. *Math Comput Sci.* 2018;12(1):51–67.
29. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun ACM.* 2008;51(1):107–13. <https://doi.org/10.1145/1327452.1327492>
30. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 1996;22(6):789–828.
31. Dean J, Ghemawat S. MapReduce: a flexible data processing tool. *Commun ACM.* 2010;53(1):72–7.
32. Dittrich J, Quiané-Ruiz JA. Efficient big data processing in hadoop mapreduce. *Proc VLDB Endow.* 2012;5(12):2014–5.
33. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, et al. Apache spark: a unified engine for big data processing. *Commun ACM.* 2016;59(11):56–65. <https://doi.org/10.1145/2934664>
34. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation USENIX Association*. Berkeley: USENIX; 2012. p. 2.
35. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: stream and batch processing in a single engine. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. Los Alamitos: IEEE; 2015. p. 4.
36. Saha B, Shah H, Seth S, Vijayaraghavan G, Murthy A, Curino C. Apache Tez: a unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. New York City: ACM; 2015. p. 1357–69.
37. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Zhang N, et al. Hive—a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE)*. Long Beach, CA: IEEE; 2010. p. 996–1005. <https://doi.org/10.1109/ICDE.2010.5447738>
38. Camacho-Rodríguez J, Chauhan A, Gates A, Koifman E, O'Malley O, Garg V, et al. Apache Hive: from MapReduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD'19*. New York: Association for Computing Machinery; 2019. p. 1773–86. <https://doi.org/10.1145/3299869.3314045>
39. Lamport LA. New solution of Dijkstra's concurrent programming problem. *Commun ACM.* 1974;17(8):453–5. <https://doi.org/10.1145/361082.361093>
40. Cole R, Vishkin U. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf Control.* 1986;70(1):32–53.
41. Schneider J, Wattenhofer R. A log-star distributed maximal independent set algorithm for growth-bounded graphs. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing*. New York City: Association for Computing Machinery; 2008. p. 35–44.
42. Barenboim L, Elkin M. Distributed graph coloring: fundamentals and recent developments, *Synthesis Lectures on Distributed Computing Theory*, vol. 4, 2013; 1–71. <https://doi.org/10.2200/S00520ED1V01Y201307DCT011>
43. Schneider J, Wattenhofer R. A new technique for distributed symmetry breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. New York City: Association for Computing Machinery; 2010. p. 257–66.
44. Linial N. Locality in distributed graph algorithms. *SIAM J Comput.* 1992;21(1):193–201.
45. Chambi S, Lemire D, Godin R, Boukhalfa K, Allen CR, Yang F. Optimizing Druid with roaring bitmaps. In: Desai E, Desai BC, Toyama M, Bernardino J, editors. *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, July 11–13, 2016*. ACM: Montreal, QC; 2016. p. 77–86. <https://doi.org/10.1145/2938503.2938515>
46. Chambi S, Lemire D, Kaser O, Godin R. Better bitmap performance with roaring bitmaps. *Softw: Pract Exper.* 2016;46(5):709–19.
47. Lemire D, Kaser O, Kurz N, Deri L, O'Hara C, Saint-Jacques F, Ssi-Yan-Kai G. Roaring bitmaps: implementation of an optimized software library. *Softw: Pract Exper.* 2018;48(4):867–95. <https://doi.org/10.1002/spe.2560>
48. Smith JE. Retrospective: a study of branch prediction strategies. In: Sohi GS, editor. *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. New York City: ACM; 1998. p. 22–3. <https://doi.org/10.1145/285930.285940>

49. Kapoor R. Avoiding the cost of branch misprediction, 2009. <https://software.intel.com/content/www/us/en/develop/articles/avoiding-the-cost-of-branch-misprediction.html>, Accessed July 13, 2020.
50. Brélaz D. New methods to color the vertices of a graph. *Commun ACM*. 1979;22(4):251–6.
51. Torres-Jimenez J, Avila-George H, Izquierdo-Marquez I. A two-stage algorithm for combinatorial testing. *Optim Lett*. 2017;11(3):457–69. <https://doi.org/10.1007/s11590-016-1012-x>
52. Mercan H, Yilmaz C, Kaya K. Chip: a configurable hybrid parallel covering array constructor. *IEEE Trans Software Eng*. 2019;45(12):1270–91. <https://doi.org/10.1109/TSE.2018.2837759>
53. Calvagna A, Gargantini A, Tramontana E. Building t-wise combinatorial interaction test suites by means of grid computing. In *2009 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*. Los Alamitos: IEEE; 2009. p. 213–18.
54. Jacob B, Brown M, Fukui K, Trivedi N. Introduction to grid computing. New York City: IBM Redbooks; 2005. p. 3–6.
55. Calvagna A, Pappalardo G, Tramontana E. A novel approach to effective parallel computing of t-wise covering arrays. In *IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, vol. 2012. Los Alamitos: IEEE; 2012. p. 149–53.
56. MPI Forum. MPI: a Message-Passing Interface Standard Version 3.1. <https://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>, 06 2015. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.
57. Yoo AB, Jette MA, Grondona M. SLURM: simple Linux utility for resource management. In: Feitelson DG, Rudolph L, Schwiegelshohn U, editors. *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, June 24, 2003, Revised Papers, Lecture Notes in Computer Science*, vol. 2862. Springer: Seattle, WA; 2003. p. 44–60. [https://doi.org/10.1007/10968987\\_3](https://doi.org/10.1007/10968987_3)
58. Prokopec A, Bagwell P, Rompf T, Odersky M. A generic parallel collection framework. In: Jeannot E, Namyst R, Roman J, editors. *Euro-Par 2011 Parallel Processing—17th International Conference, Euro-Par 2011, Bordeaux, France, August 29–September 2, 2011, Proceedings, Part II, Lecture Notes in Computer Science*, vol. 6853. Berlin, Heidelberg: Springer; 2011. p. 136–47. [https://doi.org/10.1007/978-3-642-23397-5\\_14](https://doi.org/10.1007/978-3-642-23397-5_14)
59. Colbourn C. Covering array tables for  $t = 1, 2, 3, 4, 5, 6$ , 2019. <https://www.public.asu.edu/ccolbou/src/tabby/catable.html>, Accessed June 27, 2019.
60. Garvin BJ, Cohen MB, Dwyer MB. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empir Softw Eng*. 2011;16(1):61–102. <https://doi.org/10.1007/s10664-010-9135-7>

**How to cite this article:** La Chance E, Hallé S. An investigation of distributed computing for combinatorial testing. *Softw Test Verif Reliab*. 2023;e1842. <https://doi.org/10.1002/stvr.1842>