An Efficient Distributed Algorithm for Computing Minimal Hitting Sets

[†‡]**Nuno Cardoso** and [†‡]**Rui Abreu**
[†]Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
[‡]HASLab / INESC Tec
Campus de Gualtar
Braga, Portugal
nunopcardoso@gmail.com, rui@computer.org

Abstract

Computing minimal hitting sets for a collection of sets is an important problem in many domains (e.g., Spectrum-based Fault Localization). Being an NP-Hard problem, exhaustive algorithms are usually prohibitive for real-world, often large, problems. In practice, the usage of heuristic based approaches trade-off completeness for time efficiency. An example of such heuristic approaches is STACCATO, which was proposed in the context of reasoning-based fault localization. In this paper, we propose an efficient distributed algorithm, dubbed MHS^2, that renders the sequential search algorithm STACCATO suitable to distributed, Map-Reduce environments. The results show that MHS^2 scales to larger systems (when compared to STACCATO), while entailing either marginal or small run time overhead.

1 Introduction

Computing Minimal Hitting Sets (MHSs) for a collection of constraints is an important problem in many domains (e.g., DNA analysis [1], crew scheduling [2], model/reasoning-based fault diagnosis [3; 4; 5; 6; 7], Spectrum-based Fault Localization [8]). The computation of MHSs can be polynomially reduced to the set cover optimization problem [9], which is known to be NP-hard. Being an NP-hard problem, the usage of exhaustive search algorithms (e.g., [3; 10]), is prohibitive for most real-world problems. However, in most situations, near optimal solutions are often acceptable and approximation algorithms are used to solve this problem in a reasonable amount of time. In the particular case of Spectrum-based Fault Localization (SFL), which is the context of our work, the strict minimality constraint is normally relaxed[1] and heuristics are used to drive the search in order to increase the likelihood of finding the *best* minimal candidate for a particular problem instance [8].

In this paper, we propose a Map-Reduce [11] approach, dubbed MHS^2[2], aimed at computing MHSs in a parallel or even distributed fashion in order to broaden the search scope of current approaches.

This paper makes the following contributions:

- We describe the problem in the context of SFL and present a sequential algorithm to solve it.
- We propose 3 optimizations to the sequential algorithm, which prevent a large number of redundant calculations.
- We propose an approach for dividing the MHS problem across multiple CPUs.
- We provide an empirical evaluation of our approach, showing that:
  1. The proposed optimizations introduce a considerable performance improvement.
  2. MHS^2 efficiently scales with the number of processing units.

2 Preliminaries

In this section we formally define the MHS problem and explain its relation to SFL.

2.1 Minimal Hitting Set Problem

Definition 1 (Hitting Set)**.** *Given a set $U = \{1, 2, \ldots, M\}$ (called the universe) and a collection S of N non-empty sets whose union is equal to the universe (i.e., $\bigcup_{s \in S} s = U$), a set d is said to be a Hitting Set (HS) of S if and only if*

$$\mathrm{HS}(U, S, d) : d \subseteq U \wedge \forall_{s \in S} : d \cap s \neq \emptyset \qquad (1)$$

A consequence of Definition 1 is that U is a trivial HS of S since, by definition, as $\mathrm{HS}(U, S, U)$ always holds.

Definition 2 (Minimal Hitting Set)**.** *A set d is a Minimal Hitting Set (MHS) of S if and only if*

$$\mathrm{MHS}(U, S, d) : \mathrm{HS}(U, S, d) \wedge (\nexists d' \subset d : \mathrm{HS}(U, S, d')) \qquad (2)$$

i.e., d is a HS and no proper subset of d is a HS.

There may be several MHSs d_k for S, which constitute the MHS collection D. The MHS problem consists thereby in computing D for a particular pair (U, S).

As an example, consider the universe $U = \{1, 2, 3\}$ and $S = \{\{1, 2\}, \{1, 3\}\}$. For this particular example, two MHSs exist: $\{1\}$ and $\{2, 3\}$. Even though $\{1, 2, 3\}$ is also a HS, it is not minimal as it can be subsumed either by $\{1\}$ or $\{2, 3\}$.

[1] We use the term minimal in a more liberal way due to mentioned relaxation. A Hitting Set (HS) d is said to be minimal if no other calculated HS is a proper subset of d.

[2] MHS^2 is an acronym for Map-reduce Heuristic-driven Search for Minimal Hitting Sets. An implementation of MHS^2 is available at `https://github.com/npcardoso/MHS2`.

2.2 Spectrum-based Fault Localization

SFL approaches work by abstracting the run-time behavior of the system under analysis in terms of two general concepts: components and transactions. A component is an element of the system that, for diagnostic purposes, is considered to be atomic[3], whereas a transaction is a set of component activations that (1) share a common goal, and (2) the correctness of the output can be verified. A failed transaction (or more precisely the components involved in a failed transaction) represents a conflict. Informally, a conflict is a set of components that cannot be simultaneously healthy to explain the observed erroneous behavior. Computing minimal diagnosis candidates for a set conflicts is in fact equivalent to computing MHSs for (U, S), where the components in the system are identified by the elements of U and collection S encodes the conflict collection [3].

In the remainder of this paper, we capture a set of system observations in the so called hit spectra data structure [12].

Definition 3 (Hit Spectra)**.** *Let N denote the total number of observed transactions and M denote the number of system's component. We define the hit spectra data structure as being the pair (A, e), where A is a $N \times M$* activity matrix *of the system and e the* error vector*, defined as*

$$A_{ij} = \begin{cases} 1, & \textit{if component } j \textit{ activated in transaction } i \\ 0, & \textit{otherwise} \end{cases} \tag{3}$$

$$e_i = \begin{cases} 1, & \textit{if transaction i failed} \\ 0, & \textit{otherwise} \end{cases} \tag{4}$$

3 Algorithm

A naïve, brute-force approach to compute the collection D of MHSs for S would be to iterate through all elements of the power set of U checking (1) whether it is a HS, and (2) (if it is a HS) whether it is minimal, i.e., not subsumed by any other set of lower cardinality (cardinality of a set d, $|d|$, is the number of elements in the set). Since many of the sets turn out not to be MHSs, this approach is extremely inefficient.

In this section we present MHS^2, a efficient sound and complete[4] depth-first search[5] algorithm to compute solutions to a MHS problem. This algorithm uses an heuristic to guide the search towards high potentials, yielding high efficiency and accuracy gains (in the case of large problems). Our algorithm (Algorithm 1) is based upon STACCATO [8] to which we propose 3 optimizations as well as a Map-Reduce parallelization approach.

3.1 STACCATO

In Algorithm 1 (ignoring, for now, the highlighted lines) a simplified version of STACCATO that captures its fundamental mechanics is presented[6]. The algorithm works in a divide and conquer fashion by, at each stage of its execution, performing one of two different tasks (lines 3–4, 8, and 12-13 or 15–18), depending on whether or not the set d is a HS.

Algorithm 1 STACCATO/MHS^2 map task

Inputs:
Spectra (A, e)
Set d (default: $\emptyset$)
Hitting set collection D (default: $\emptyset$)

Parameters:
Ranking heuristic $\mathcal{H}$
Branch level L
Load division function SKIP

Output:
Minimal hitting set collection D

```
1  if ∃i ∄j : e_i = 1 ∧ A_ij = 1 then                    # opt. 3
2      return D
3  if ∃i : e_i = 1 then                                  # divide task
4      R ← RANK(H, A, e)
5      for all (s, j) : (s, j) ∈ R ∧ s = 0 do            # opt. 2
6          R ← R \ {j}
7          A ← STRIP_COMPONENT(A, j)
8      for j ∈ R do
9          if SIZE(d') + 1 = L ∧ SKIP() then
10             A ← STRIP_COMPONENT(A, j)                 # opt. 1
11             continue
12         (A', e') ← STRIP(A, e, j)
13         D ← STACCATO(A', e', D, d ∪ {j})
14         A ← STRIP_COMPONENT(A, j)                     # opt. 1
15 else                                                  # conquer task
16     if ∄d' ∈ D : d' ⊆ d then
17         D ← D \ {d' | d' ∈ D ∧ d ⊆ d'}
18         D ← D ∪ {d}
19 return D
```

The first task, which is triggered whenever d is not a HS[7] (line 3), aims at dividing the initial problem in smaller subproblems. This goal is achieved by iterativelly selecting a component j from an heuristically ordered set of components R (lines 4 and 8) and creating a temporary spectra (A', e') where all rows A_i such that $A_{ij} = 1$ as well as column j are omitted (function STRIP, line 12). Finally, the algorithm makes a recursive call in order to solve the subproblem (A', e') with set $d \cup \{j\}$ (line 13).

The second task, which occurs whenever d is a HS (lines 7–9), aims at collecting HSs while guaranteeing that all HSs in D do not have a proper subset also contained in D. The first step in this task is to check if d is minimal (line 16) with regard to the already discovered MHS collection D. If d is minimal, all super-sets of d in D are purged (line 17) and, finally, d is added to D (line 18).

To illustrate how STACCATO works, consider the exam-

[3] In a software environment, a component can be for instance a statement, a function, a class, or a service.

[4] For small enough problems.

[5] The reason for using a depth-first search is related to the fact that large real-world problems generate search trees with a large branching factor, making a breadth-first search approach very costly.

[6] For simplicity, the cut-off conditions were omitted and, as a result, a direct implementation of the algorithm will not be suitable to large problems. Both the details regarding search heuristics and cut-off parameters are outside the scope of this paper (refer to [8] for more information - the algorithm was also explained in [13] as well as [14]).

[7] Due to the divide and conquer nature of the algorithm, d is a HS whenever all failing transactions are striped from the original spectra.

Algorithm 2 MHS^2 reduce task

Inputs:

Partial minimal hitting set collections $D'_1, ..., D'_K$

Output:

Minimal hitting set collection D

```
1 D ← ∅
2 D' ← SORT(⋃_{k=1}^{K} D'_k)
3 for d ∈ D' do
4     if MINIMAL(D, d) then
5         D ← D ∪ {d}
6 return D
```

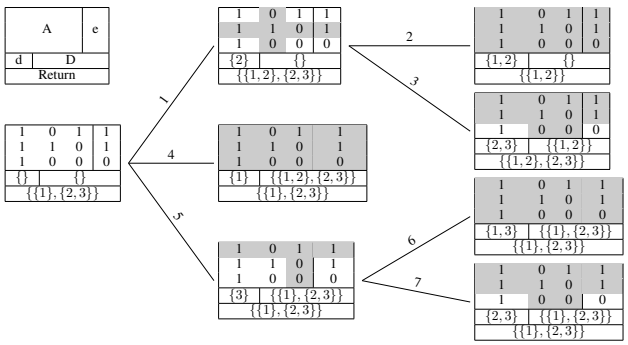

Figure 1: Example search tree

ple in Figure 1[8] which represents a possible search tree for STACCATO. Each node in the search tree represents a call to the function (all the parameters as well as the return value are encoded as a table). Gray spectra elements represent the portions of the original spectra filtered by STRIP function. Leaf nodes represent function calls for which d is a HS whereas intermediate nodes represent function calls for which d is not a HS.

In the outer call to the algorithm (the leftmost node), as $\exists i : e_i = 1$, the algorithm performs the divide task. After exploring the sub-tree starting with $d = \{2\}$ (calls 1–3), the algorithm yields the collection $D = \{\{1, 2\}, \{2, 3\}\}$.

We can see that at this point, if the execution were to be interrupted, $\{1, 2\}$ would be erroneously considered a MHS. However, after exploring the sub-tree starting with $d = \{1\}$ (call 4), the set $\{1, 2\}$ is removed yielding the collection $D = \{\{1\}, \{2, 3\}\}$.

The inspection of the sub-tree starting with $d = \{3\}$ (call 5–7) does not make further changes to collection D. On the one hand, the HS $\{1, 3\}$ is a proper super-set of $\{1\}$. On the other hand, the HS $\{2, 3\}$ is already contained in D.

As expected, the result for this example would be the collection $D = \{\{1\}, \{2, 3\}\}$.

3.2 MHS^2

In this section we propose MHS^2, our distributed MHS search algorithm. The proposed approach can be viewed as a Map-Reduce task [11]. The map task, presented in Algorithm 1 (now also including highlighted lines), consists of an adapted version of the sequential algorithm just outlined. In contrast to the original algorithm, we added 3 optimizations that prevent redundant calculations.

The first optimization (lines 10 and 14) prevents multiple examinations of the same set, as it was the case of $\{2, 3\}$ in the example from the previous section[9].

The second optimization (lines 5–7) preemptively filters components with heuristic score equal to 0. This optimization works under the assumption that the heuristic scores are non-negative and components with heuristic score equal to 0 are not members of any conflict set (and, consequently, guaranteed not to form non-minimal HSs). As this filtering process reduces the spectra size, the heuristic calculation overhead (which normally is $O(N \times M)$) is decreased.

The third optimization (lines 1–2) prevents the examination of branches that contain empty conflict sets. If a branch exhibits such property, any node in its sub tree will always contain at least one unsolved conflict thus guaranteeing no MHS will be found.

To parallelize the algorithm, we added a parameter L that sets the *split-level*, i.e., the number of calls in the stack minus 1 or $|d|$, at which the computation is divided among the processes. When a process of the distributed algorithm reaches the target level L, it uses a load division function (SKIP) in order to decide which elements of the ranking to skip or to analyze (line 9). The value of L implicitly controls the granularity of decision of the load division function at the cost of performing more redundant calculations. Implicitly, by setting a value $L > 0$, all processes redundantly compute all HS such that $|d| <= L$.

With regard to the load division function SKIP, we propose two different approaches. The first, referred to as *stride*, consists in assigning elements of the ranking R to processes in a cyclical fashion. Formally, a process $p_{k \in [1..np]}$ is assigned to an element $R_l \in R$ if $(l \mod np) = (k - 1)$.

The second approach, referred to as *random*, uses a pseudo-random generator in order to divide the computation. This random generator is then fed into an uniform distribution generator that assures that, over time, all p_k get assigned a similar number of elements in the ranking R although in random order (specially for large values of L). This method is aimed at obtaining a more even distribution of the problem across processes than *stride*[10]. A particularity of this approach is that the seed of the pseudo random generator must be shared across process in order to assure that no further communication is needed.

Finally, the reduce task, responsible for merging all partial MHS collections $D'_{k \in [1..np]}$ originating from the map task (Algorithm 1), is presented in Algorithm 2. The reducer works by merging all HSs in a list ordered by cardinality. The ordered list is then iterated, adding all MHSs to D. As the HSs are inserted in an increasing cardinality order, it is not necessary to look for subsumable HSs (line 17 in Algorithm 1) in D.

4 Results

To assess the performance of our algorithm we conducted two sets of benchmarks (all the benchmarks were conducted in a single computer with 2× Intel Xeon CPU X5570 @ 2.93GHz, 4 cores each). The first is focused on evaluating

[8] Note that for this example the order of node exploration (and consequently the shape of the tree) was selected for illustrative purposes.

[9] This optimization generalizes over the optimization proposed in [8], which would be able to ignore the calculation of $\{1, 2\}$ but not the redundant reevaluation of $\{2, 3\}$.

[10] A consequence of the proposed optimizations is that, as the size of (A', e') is different for all components $j \in R$, the time that the recursive call takes to complete may vary substantially.

the impact of each of the proposed optimizations. The second is focused on evaluating the algorithm's parallelization approach.

For our benchmarks, we generated several (A, e) by means of a Bernoulli process, with parameters M (number of transactions), N (number of components), and R (probability of component activation in each transaction, also known as activation rate). The presented results represent the average over 100 test cases for each combination (M, N, R). To ease the comparison of results, *all* transactions in *all* generated cases fail[11]. Both due to space constraints and the fact that the algorithm's diagnosis accuracy has already been studied in [8], we only analyze the computational gains introduced by our approach.

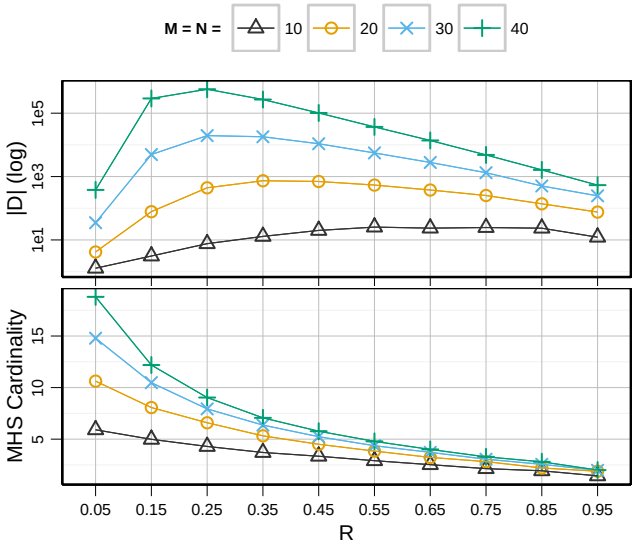

Figure 2: (M, N, R) parameters' impact

To better understand how the parameters affect the problem's solution, consider the following observations (Figure 2):

1. For the same (M, N) parameter values:
   (a) The average MHS cardinality for problems generated with smaller R values is larger than the average MHS cardinality for problems generated with larger R values (i.e., the MHS cardinality is negatively correlated with the R value).
   (b) The average solution size (i.e., $|D|$) was minimal for problems generated with $R = 0.05$.
   (c) $|D|$ was **not** maximal for problems generated with $R = 0.95$.
2. Problems generated with larger (M, N) values have the maximal $|D|$ value for smaller R values than problems generated with smaller (M, N) values (i.e., the value of R for maximal $|D|$ is negatively correlated with (M, N)).

[11]To illustrate the potential problems of having successful transactions in the test cases, consider the extreme case of a set of test cases with no failures versus a set of test cases with no nominal transactions. For the first scenario, all test cases only have one MHS (the empty set) whereas, for the second scenario, a potentially large number of MHSs may exist, thus having a large impact on the algorithms' run-time.

4.1 Optimizations

Small problems

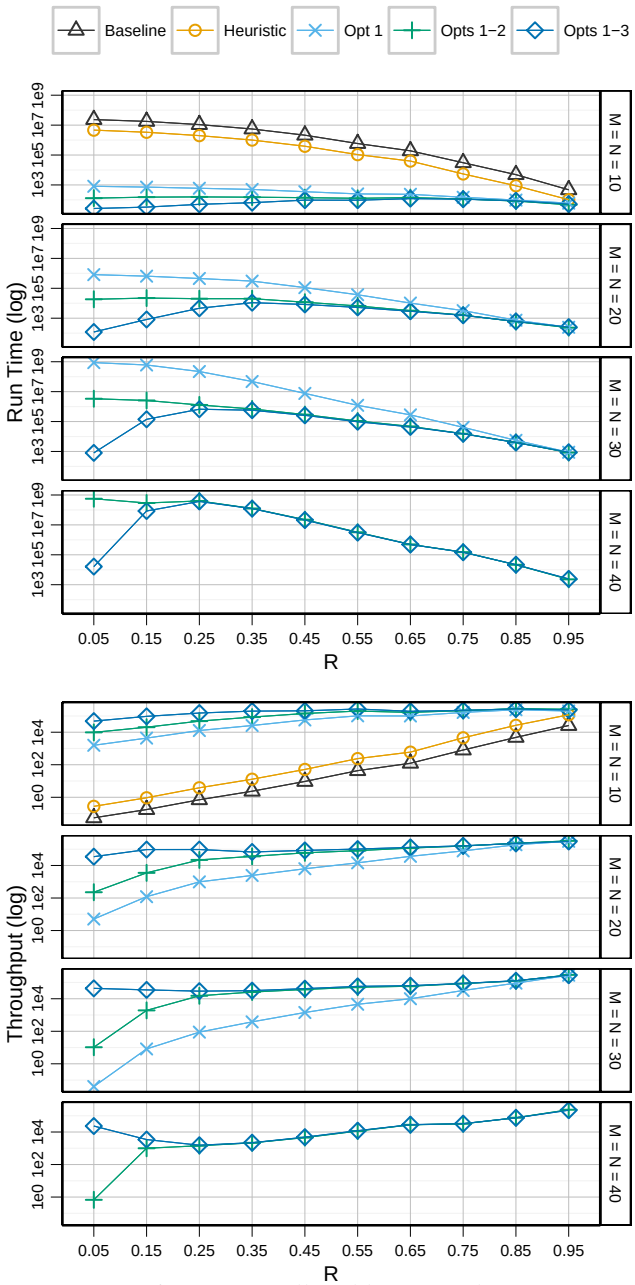

Figure 3: Small problems' results

The first benchmark in this section is aimed at evaluating the impact of each optimization for small problems, for which all MHSs can be calculated ($M = N \in \{10, 20, 30, 40\}$, and $R \in \{0.05, 0.15, ..., 0.95\}$).

In Figure 3, we observe the run-times/throughputs[12] for 5 different implementations of the algorithm with a decreasing number of features[13]. At one end of the scale, *Opts 1–3* represents an implementation with both all of the proposed

[12]The throughput metric is calculated as the number of generated MHSs divided by the total run-time and is measured in MHSs/sec. When calculating this metric, we took care to discard all non-minimal HSs.

[13]The test subjects were removed from the benchmark when the execution of individual test cases exceeded 1000 seconds.

optimizations[14] as well as the heuristic[15]. At the other end, *Baseline* represents an implementation with no optimizations as well as no heuristic (i.e., random ranking). Even though the performance of STACCATO (the algorithm that served as starting point for our research) was not directly considered, it should be bounded by the performances of *Heuristic* (in a worst case) and of *Opt 1* (in the best case).

The analysis of the results shows that the heuristic by itself introduces a significant performance improvement over *Baseline* for $M = N = 10$ ($5\times$ faster on average, note the log scale). Such results present a strong evidence that using an heuristic to drive the search not only improves the quality of the computed MHSs (as shown in [8]) but also improves the computational efficiency of the algorithm. For such small cases, and in comparison to the *Heuristic* performance, all of the optimizations showed an improved performance of at least $1.5\times$ (*Opt 1* for $R = 0.95$), at most $170000\times$ (*Opts 1–3* for $R = 0.05$), and on average $34000\times$, $8000\times$, and $1700\times$ for *Opts 1–3*, *Opts 1–2*, and *Opt 1* respectively.

Analyzing the remaining tests cases ($M = N \in \{20, 30, 40\}$), we can see that, with the increase of the problem size, the relative contribution of each optimization becomes more significant (for $M = N \in \{10, 20, 30\}, R = 0.05$, *Opts 1–3* performs $30\times$, $7000\times$, and $1000000\times$ faster than *Opt 1*, respectively). We also observe that, on average and for all combinations of (M, N, R), *Opts 1–3* was the algorithm with the best performance, implying that the computational savings introduced by optimization 3 should, on average, outweigh its overhead.

It also worth understanding how each optimized implementation improves over *Heuristic*. A closer inspection of the results reveals the following patterns emerge:

1. All algorithms have similar performances for large R values.
2. The optimizations are more effective for smaller values of R.
3. Optimizations 1 and 2 have a considerable effect for all R values whereas optimization 3 is only effective for small R values.

Pattern 1 can be explained by recalling that MHS^2 is a divide and conquer algorithm and the observations made about Figure 2. As the average MHS cardinality is negatively correlated with the R value, we see that for problems with large R values, the algorithms' search trees are shallow and, consequently, the algorithms spend most of their time performing the conquer task[16]. Given that all the optimizations focus on improving the search tree exploration, for shallow trees the performance impact is less significant.

Conversely, pattern 2 can be explained using the complimentary argument. For small R values, as the search trees become deeper, the non-optimized algorithms perform a large amount of unnecessary divide tasks, thus leaving (exponentially) more room for improvements.

To explain pattern 3, we shall look at the optimizations individually:

- Optimization 1 prevents the exploration of paths composed of the same components although in different orders. Such inefficiencies occur whenever the MHSs are composed of more than 1 element.
- Optimization 2 reduces the overhead associated with the heuristic calculation. The overhead reduction also occurs whenever the MHSs are composed of more than 1 element.
- Optimization 3 performs a look-ahead verification to assess whether or not the current sub-tree is a dead-end (i.e., no MHSs will be generated in such sub-tree) and terminate the exploration if a dead-end is reached. The impact of this optimization is contingent on how far ahead it detects the dead-end which, in turn, is dependent on the deepness of the search tree. As the deepness of the search tree is negatively correlated with R, this optimization is more effective for small R values.

Large problems

The second benchmark is aimed at evaluating the impact of each optimization for large problems where it is impractical to calculate all MHSs ($M = N = 10^3$, and $R \in \{0.05, \cdots, 0.95\}$). In all of the following test cases a time based cut-off of 30 seconds was enforced.

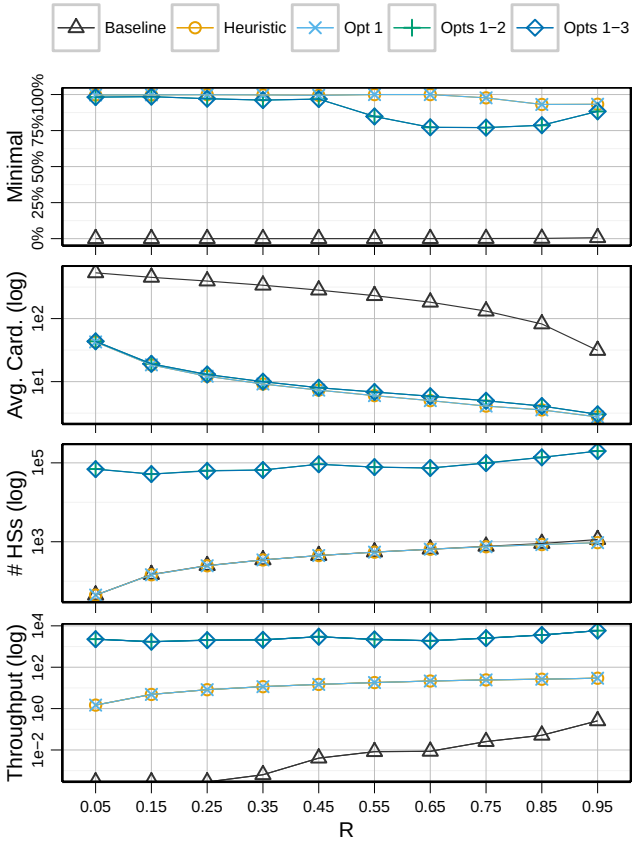

Figure 4: Large problems' results

The first two plots in Figure 4 present both the percentage of HSs that were minimal (MHS%) as well as the average HS cardinality for each of the 5 implementations. For large problems, we observe that the heuristic plays an important role in assuring that the computed HSs are in fact minimal (98% vs. 0% minimality for *Heuristic* and *Baseline*, on av-

[14]The numbers of the optimizations are presented in Algorithm 1.

[15]All the benchmarks including an heuristic were conducted using the Ochiai heuristic (see [8]).

[16]Intuitively and simplifying the problem, performing the conquer task increases the throughput, whereas performing the divide task decreases the throughput.

erage). Even though the MHS% of *Opts 1–2* and *Opts 1–3* are lower when compared to *Opt 1*, we can see that the average HS cardinality of the former implementations is comparable to cardinality of the later, implying that the number of extraneous elements in the non-minimal HSs is small (specially when compared to *Baseline*).

Despite the lower MHS percentage of the fully optimized algorithm, in absolute terms it computes a significantly larger amount of HSs than *Opt 1*. The remainder of Figure 4 presents the number of HSs as well as the throughput for all implementations. While *Baseline*, *Heuristic*, and *Opt 1* compute 500 HSs on average, the remaining implementations compute 93000 HSs on average (186× better). Taking into account the number of computed HSs, despite the lower MHS percentage, the absolute number of MHS for optimizations 2 and 3 is effectively higher than the number of MHS calculated by the other approaches.

Finally, it is interesting to note that even though we increased the problem size by 25× factor, the throughput of MHS^2 is comparable to the throughput observed in Section 4.1. The practical implications of such observation is that the fully optimized MHS^2 efficiently scales to large problems whereas STACCATO does not (since STACCATO's performance should be bounded by the performances of *Heuristic* and *Opt 1*).

4.2 Parallelization

Small Problems

The first parallelization benchmark, is aimed at observing the behavior of MHS^2 for small problems ($M = N = 40$, $R \in \{0.25, 0.5, 0.75\}$).

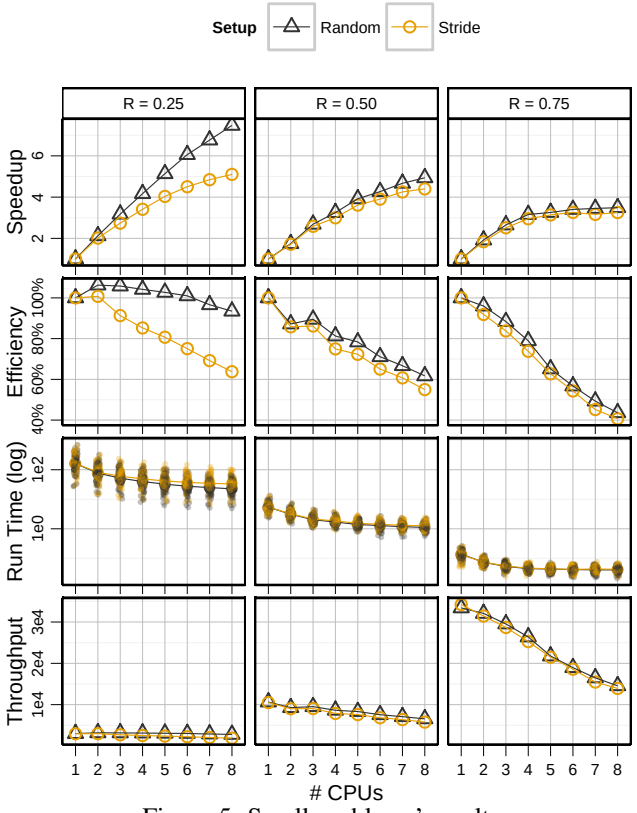

Figure 5: Small problems' results

In Figure 5, we observe both the speedup (defined as $\text{SUP}(np) = \frac{T_1}{T_{np}}$, where T_{np} is the time needed to solve the problem for np processes) and the efficiency (defined as $\text{EF}(np) = \frac{\text{SUP}(np)}{np}$) for both the *Stride* and *Random* load distribution functions. Also, Figure 5, shows both the run-times [17] as well as the throughput for both load distribution functions.

The analysis of Figure 5 shows different speedup/efficiency patterns for different values of R. This difference is due to the large difference in run-time for different values of R: $\approx$ 200, 5.7 and 0.1 seconds for $R = \{0.25, 0.5, 0.75\}$, respectively. On the one hand, when the run-time is smaller, the parallelization overhead has a higher relative impact in the performance (in extreme cases, the run-time can even increase with the number of processes). On the other hand, when both the cardinality and the amount of MHSs increase (small values of R) the parallelization overhead becomes almost insignificant.

For $R = 0.25$ and in contrast to $R \in \{0.5, 0.75\}$, the speedup/efficiency of *Random* is much superior to the performance of *Stride*. The reason of such efficiency difference is that the *cool-down period* (i.e., the period in which at least one process is idle and another is active) of *Stride* was longer than the one observed for *Random*. The smaller cool-down period of *Random* shows that the usage of a stochastic approach evenly divides an unbalanced search tree, leading to a better load division.

Finally, we observe that *Random* experiences super linear speedups (i.e, efficiency above 100%). This pattern emerges due to the fact that the complexity of the operations performed on D (lines 16 – 17 in Algorithm 1)) are not linear with the number of elements stored in it. As a consequence, by reducing size of D to $\frac{1}{np}$ effectively reduces the cost of the operations by a factor greater than np. The corollary of such observation is that a better data structure for encoding D is likely to exist[18]

Large Problems

The second benchmark is aimed at observing the behavior of MHS^2 for large problems ($M = N = 10^3$, and $R \in \{0.25, 0.5, 0.75\}$).

Figure 6 shows the minimality percentage, the number of HSs, and the throughput for large problems when using time based cutoffs of $1, 2, 4, 8$, and 16 seconds per process with a varying number of processes (entailing a total CPU time of approximately $rt_{total} \times np$, where rt_{total} is the total run-time, including both map and reduce tasks, which must be greater than the cutoff value). It appears that, for large problems, there is no significant difference in terms of the amount of generated MHSs between *Random* and *Stride*. Performing a two-tailed T-test we determined, with a 99% confidence interval, that both approaches should have, on average, equal throughputs.

Regarding the minimality percentage of both approaches,

[17] In this plot, each cluster is composed of 1 data point per test case (100 data points for each load distribution function). The horizontal displacement inside each cluster was only added to improve the visualization of the results.

[18] In our implementation we make use of a trie to encode D. We envision this data structure to function similarly to an hashtable: the *hashtrie* data structure should be composed of multiple tries which are used to divide the load according to an HS hashing function. After all additions are made to D, the individual tries must be merged to remove non-minimal HSs.

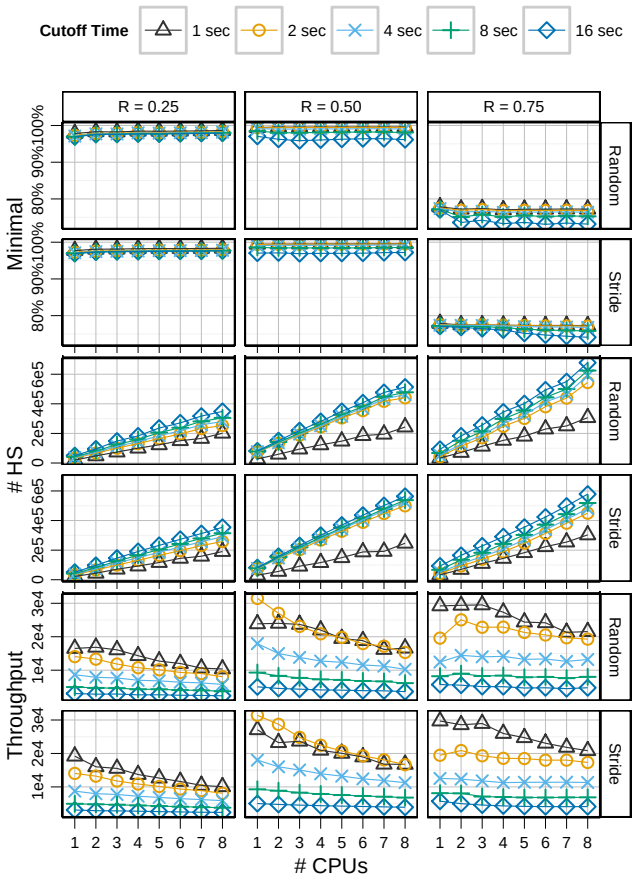

Figure 6: Large problems' results

we observed similar results to those presented in Section 4.1 even though different cutoff values were used: for $R = \{0.25, 0.5\}$, 97% minimality while for $R = 0.75$ the percentage decreases to around 75%. Also, the throughput is consistent with all previous benchmarks.

Figure 7 shows the number of MHSs and throughput after applying a data transformation. In spite of using the number of processes as the x-axis, we use the total algorithm run time. Each data point is represented by a number, which encodes the number of used CPUs. We can see that, as expected, running the algorithm for 8 seconds on 8 CPUs accounts approximately for the same total run-time as running the algorithm for 16 seconds on 4 CPUs.

Using this plot we can easily see that for a given total calculation time, it is always preferable to divide the task among the maximum possible number of CPUs (eventually, given enough CPUs, this trend should hit a maximum). This patterns can again be explained by the trie non-optimally argument presented before.

4.3 Related Work

Several approaches to solve the MHS problems have been proposed. In [3], the authors proposed a breath-first search algorithm that uses the so called HS-trees and, in [10; 15], some improvements over the base algorithm have been suggested. In [7] a method using set-enumeration trees to derive all MHSs in the context of model-based diagnosis is presented. All of the above algorithms make use of a constraint solver to check whether or not a set d is a HS, thus not requiring an explicit conflict set availability. While sound and complete, such algorithms do not gracefully scale to large real-world problems.

In [5], the authors propose a stochastic search algorithm, that starts with a HS d for (U, S) and iteratively removes elements from d while guaranteeing that the resulting set still is a HS. In [16; 17; 18; 19] several genetic algorithms to compute MHSs are proposed. While scalable to large problems, these algorithms do not guarantee soundness nor completeness.

As shown in the previous sections, our heuristic-driven depth-first search algorithm is able to guarantee soundness and completeness in small problems as also efficiently scale to large problems while guaranteeing a high minimality percentage (above 75%).

5 Conclusions

In this paper, we proposed an optimized and distributed version of STACCATO, dubbed MHS^2, for computing Minimal Hitting Sets/minimal diagnostic candidates. This algorithm is not only more efficient in single CPU scenarios than the original sequential algorithm but is also able to efficiently use the processing power of multiple CPUs to calculate MHSs.

The results showed that, the proposed optimizations have a large impact on the algorithm's performance and also that the algorithm is able to horizontally scale with negligible overhead. The usage of parallel processing power enables the exploration of a larger number of potential candidates, increasing the likelihood of actually finding the "best" HS for a particular instance of the problem.

Future work would include the analysis of the algorithm's performance with a larger set of computation resources as also the analysis of its performance under a wider set of conditions.

Acknowledgements

We would like to thank Lígia Massena, André Silva and José Carlos de Campos for the useful discussions during the development of our work. This material is based upon work supported by the National Science Foundation under Grant No. CNS 1116848, by the scholarship number SFRH/BD/79368/2011 from Fundação para a Ciência e Tecnologia (FCT), and by the ERDF through the Programme COMPETE, the Portuguese Government through FCT - Foundation for Science and Technology, project reference FCOMP-01-0124-FEDER-020484.

References

[1] D. P. Ruchkys and S. W. Song. A parallel approximation hitting set algorithm for gene expression analysis. In *Symposium on Computer Architecture and High Performance Computing*, pages 75–81, 2002.

[2] J. Rubin. *A Technique for the Solution of Massive Set Covering Problems, with Application to Airline Crew Scheduling*. 1973.

[3] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence.*, 32(1):57–95, 1987.

[4] Johan de Kleer and Brian C. Williams. Readings in model-based diagnosis. In *Readings in model-based diagnosis*, chapter Diagnosing multiple faults, pages 100–117. 1992.

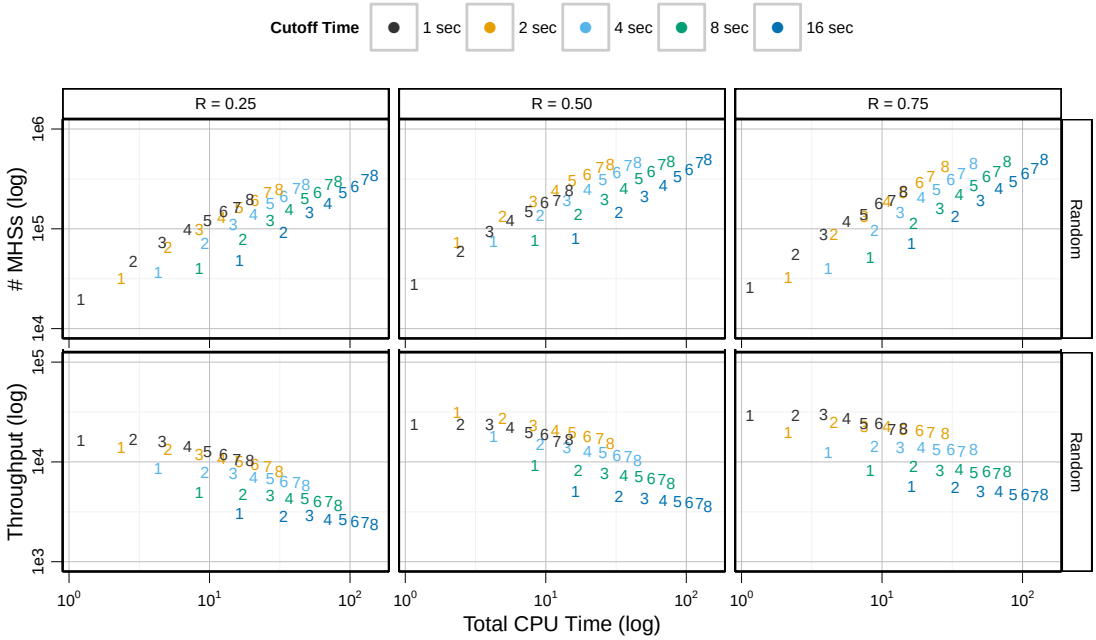

Figure 7: Large problems' results (with x-axis transformation)

[5] Alexander Feldman, Gregory Provan, and Arjan J. C. van Gemund. Computing minimal diagnoses by greedy stochastic search. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*, AAAI'08, pages 911–918, 2008.

[6] Ingo Pill and Thomas Quaritsch. Optimizations for the boolean approach to computing minimal hitting sets. In *European Conference on Artificial Intelligence*, ECAI'12, pages 648–653, 2012.

[7] Xiangfu Zhao and Dantong Ouyang. Improved algorithms for deriving all minimal conflict sets in model-based diagnosis. In *International Conference on Intelligent Computing*, ICIC'07, pages 157–166, 2007.

[8] Rui Abreu and Arjan J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Proceedings of the 8th Symposium on Abstraction, Reformulation, and Approximation*, SARA'09, 2009.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.

[10] Franz Wotawa. A variant of Reiter's hitting-set algorithm. *Information Processing Letters*, 79(1):45–51, 2001.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 137–150, 2004.

[12] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE'98, pages 83–90, 1998.

[13] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artificial Intelligence*, 174(18):1481–1497, 2010.

[14] Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, November 2009.

[15] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[16] Staal A. Vinterbo and Aleksander Øhrn. Minimal approximate hitting sets and rule templates. *Int. J. Approx. Reasoning*, 2000.

[17] Lin Li and Jiang Yunfei. Computing minimal hitting sets with genetic algorithm. Technical report, DTIC Document, 2002.

[18] Uwe Aickelin and Kathryn A Dowsland. An indirect genetic algorithm for a nurse-scheduling problem. *Computers & Operations Research*, 2004.

[19] Wen-Chih Huang, Cheng-Yan Kao, and Jorng-Tzong Horng. A genetic algorithm approach for set covering problems. In *International Conference on Evolutionary Computation*, 1994.