A Distributed Approach to Diagnosis Candidate Generation

Nuno Cardoso and Rui Abreu

Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
nunopcardoso@gmail.com, rui@computer.org

Abstract. Generating diagnosis candidates for a set of failing transactions is an important challenge in the context of automatic fault localization of both software and hardware systems. 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 runtime overhead.

Keywords: Fault Localization, Minimal Hitting Set, Map-Reduce

1 Introduction

Detecting, localizing and correcting erroneous behavior in software, collectively known as debugging, is arguably one of the most expensive tasks in almost all software systems' life cycle [8]. While still being essentially a manual process, a wide set of techniques have been proposed in order to automate this process [1–3, 5]. With regard to automatic fault localization, which is the scope of this paper, two main problems exist:

- Finding sets of components (known as diagnostic candidates) that, by assuming their faultiness, would explain the observed erroneous behavior.
- Ranking the candidates according to their likelihood of being correct.

In this paper, we propose a Map-Reduce [4] approach, dubbed MHS^2[1], aimed at computing minimal diagnosis candidates in a parallel or even distributed fashion in order to broaden the search scope of current approaches.

This paper makes the following contributions:

[1] MHS^2 is an acronym for Map-reduce Heuristic-driven Search for Minimal Hitting Sets.

- We introduce an optimization to the sequential algorithm, which is able to prevent a large number of redundant calculations.
- We propose MHS^2, a novel Map-Reduce algorithm for dividing the minimal hitting set (MHS) problem across multiple CPUs.
- We provide an empirical evaluation of our approach, showing that MHS^2 efficiently scales with the number of processing units.

The remainder of this paper is organized as follows. In Section 2 we describe the problem as well as the sequential algorithm. In Section 3 we present our approach. Section 4 discusses the results obtained from synthetic experiments. Finally, in Section 5 we draw some conclusions about the paper.

2 Preliminaries

As previously stated, the first step when diagnosing a system is to generate valid diagnosis candidates, conceptually known as hitting sets. In scenarios where only one component is at fault this task is trivial. However, in the more usual case where multiple faults are responsible for erroneous behavior, the problem becomes exponentially more complex. In fact, for a particular system with M components there are 2^M possible (but not necessarily valid) candidates. To reduce the number of generated candidates only minimal candidates are considered.

Definition 1 (Minimal Valid Candidate/Minimal Hitting Set). *A candidate d is said to be both valid (i.e., a hitting set) and minimal if (1) every failed transaction involved a component from candidate d and (2) no valid candidate d' is contained in d.*

We use the hit spectra data structure to encode the involvement of components in pass/failed system transactions (a transaction can represent, for instance, a test case from a particular test suite).

Definition 2 (Hit Spectra). *Let S be a collection of sets s_i such that*

$$s_i = \{j \mid \textit{if component } j \textit{ participated in transaction } i\}$$

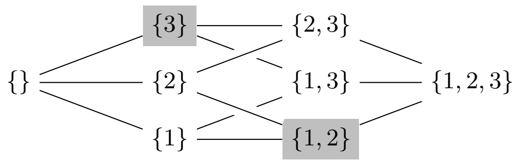
Let $N = |S|$ denote the total number of observed transactions and $M = |COMPS|$ denote the number of system components. A hit spectra [9] is a pair (A, e), where A is a $N \times M$ activity matrix *of the system and e the* error vector*, defined as*

$$A_{ij} = \begin{cases} 1, & \textit{if } j \in s_i \\ 0, & \textit{otherwise} \end{cases} \qquad e_i = \begin{cases} 1, & \textit{if transaction } i \textit{ failed} \\ 0, & \textit{otherwise} \end{cases}$$

As an example, consider the hit spectra in Figure 1a for which all 2^M possible candidates (i.e., the power set $\mathcal{P}_{COMPS}$) are presented in Figure 1b. For this particular example, two valid minimal candidates exist: $\{3\}$ and $\{1, 2\}$. Even though

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(a) Hit spectra example



(b) Hasse diagram of $\mathcal{P}_{\{1,2,3\}}$

Fig. 1: Running example

the candidate $\{1, 2, 3\}$ is also valid, it is not minimal as it can be subsumed by either $\{3\}$ or $\{1, 2\}$.

Despite the advantage of only computing minimal candidates, the problem is known to be NP-hard [6]. Being an NP-hard problem, the usage of exhaustive search algorithms (e.g., [7, 12, 14]), is prohibitive for most real-world problems. In order to solve the candidate generation problem in a reasonable amount of time, approaches that relax the strict minimality[2] constraint have been proposed [2, 10, 5, 11, 13, 15]. In particular, Staccato [2], the sequential algorithm which serves as foundation for our distributed algorithm, uses the Ochiai heuristic [3] in order to increase the likelihood of finding the actual fault explanation, yielding significant efficiency gains.

In Algorithm 1 (discarding, for now, the highlighted lines) a simplified version of Staccato that captures its fundamental mechanics is presented[3]. The algorithm can perform two different tasks (lines 2–4 and 6–11), depending on whether or not the candidate d' is a hitting set. By definition, d' is a hitting set if all failing transaction in the spectra under analysis contain at least one component in d'. Due to the structure of the algorithm d' is a hitting set whenever e does not contain any errors (i.e., $\nexists e_i \in e : e_i = 1$). In the first case, where d' is a hitting set (lines 2–4), the algorithm checks if d' is minimal (line 2) with regard to the already discovered minimal hitting set collection D'. If d' is minimal, all hitting sets in D' subsumable by d' are purged (line 3) and d' is added to D' (line 4). In the second case, where d' is not a hitting set (lines 6–11), the algorithm composes candidates in the form of $d' + \{j\}$. The order in which the components $j \in R \subset COMPS$ are selected is determined by some arbitrary heuristic $\mathcal{H}$ (line 6). This heuristic is responsible for both driving the algorithm towards high potential candidates and reducing the amount of candidates that need to be checked. Such tasks are application dependent and, in the particular case of model/reasoning-based fault diagnosis (MBD), the Ochiai heuristic [2, 3] was shown to exhibit good accuracy levels. Whenever a component j is selected, a temporary (A', e') is created where all transactions s_i such that $j \in s_i$ as well as column j are omitted (function Strip, line 9). Finally, the algorithm makes a recursive call in order to explore (A', e') with candidate $d' + \{j\}$ (line 10).

[2] We use the term minimal in a more liberal way due to mentioned relaxation. A candidate d is said to be minimal if no other calculated candidate is contained in d.

[3] It is important to note that the cut-off conditions were removed for the sake of simplicity and any direct implementation 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 [2] for more information.

Algorithm 1 Staccato / MHS^2 map task

```
Inputs:
    Matrix $(A, e)$
    Partial minimal hitting set collection $D'$ (default: $\emptyset$)
    Candidate $d'$ (default: $\emptyset$)
Parameters:
    Ranking heuristic $\mathcal{H}$
    Branch level $L$
    Load division function Skip
Output:
    Minimal hitting set collection $D$
    Partial minimal hitting set collection $D'_k$

 1 if $\nexists e_i \in e : e_i = 1$ then                 # Minimality verification task
 2     if Minimal$(D', d')$ then
 3         $D' \leftarrow$ Purge_Subsumed$(D', d')$
 4         $D' \leftarrow D' \cup \{d'\}$
 5 else                                                    # Candidate compositions task
 6     $R \leftarrow$ Rank$(\mathcal{H}, A, e)$                # Heuristic ranking
 7     for $j \in R$ do
 8         if $\neg($Size$(d') + 1 = L \wedge$ Skip$())$ then  # Load Division
 9             $(A', e') \leftarrow$ Strip$(A, e, j)$
10             $D' \leftarrow$ Staccato$(A', e', D', d' + \{j\})$
11         $A \leftarrow$ Strip_Component$(A, j)$           # Optimization
12 return $D'$
```

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(a) After Strip$(A, e, 3)$

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(b) After Strip$(A, e, 2)$

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(c) After Strip$(A, e, 1)$

Fig. 2: Evolution of (A, e)

To illustrate how Staccato works, recall the example in Figure 1a. In the outer call to the algorithm as $\exists e_i \in e : e_i = 1$ (i.e., candidate d' is not a hitting set), candidates in the form of $\emptyset + \{j\}$ are composed. Consider, for instance, that the result of the heuristic over (A, e) entails the ranking $(3, 2, 1)$. After composing the candidate $\{3\}$ and making the recursive call, the algorithm adds it to D' as $\nexists e_i \in e : e_i = 1$ (Figure 2a) and $D' = \emptyset$, yielding $D' = \{\{3\}\}$. After composing candidate $\{2\}$, a temporary (A', e') is created (Figure 2b). Following the same logic, the hitting sets $\{2, 1\}$ and $\{2, 3\}$ can be found but only $\{2, 1\}$ is minimal as $\{2, 3\}$ can be subsumed by $\{3\}$[4]. Finally the same procedure is repeated for component 1 (Figure 2c), however no new minimal hitting set is found. The result for this example would be the collection $D = \{\{1, 2\}, \{3\}\}$.

[4] Actually, Staccato would not compose candidate $\{2, 3\}$ due to an optimization that is a special case of the one proposed in this paper (see Section 3).

3 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 [4]. 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 an optimization that prevents the calculation of the same candidates in different orders (line 11), as it would be the case of candidates $\{1, 2\}$ and $\{2, 1\}$ in the example of the previous section. Additionally, it generalizes over the optimization proposed in [2], which would be able to ignore the calculation of $\{2, 3\}$ but not the redundant reevaluation of $\{1, 2\}$. The fundamental idea behind this optimization is that after analyzing all candidates that can be generated from a particular candidate d' (i.e., the recursive call), it is guaranteed that no more minimal candidates subsumable by d' will be found[5]. A consequence of this optimization is that, as the number of components in (A, e) is different for all components $j \in R$, the time that the recursive call takes to complete may vary substantially.

To parallelize the algorithm across *np* processes, 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. 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 calculate all candidates 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*. 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 minimal hitting set collections $D'_{k \in [1..np]}$ originating from the map task (Algorithm 1), is presented in Algorithm 2. The reducer works by merging all hitting sets in a list

[5] Visually, using a Hasse diagram (Figure 1b), this optimization can be represented by removing all unexplored edges touching nodes subsumable by d'. As every link represents an evaluation that would be made without any optimizations (several links to same node means that the set is evaluated multiple times), it becomes obvious the potential of such optimization.

Algorithm 2 MHS^2 reduce task

Inputs:
Partial minimal hitting set collections $D'_1, ..., D'_K$
Output:
Minimal hitting set collection D

```
1 $D \leftarrow \emptyset$
2 $D' \leftarrow$ SORT($\bigcup_{k=1}^{K} D'_k$)                # Hitting sets sorted by cardinality
3 for $d \in D'$ do
4     if MINIMAL($D, d$) then
5         $D \leftarrow D \cup \{d\}$
6 return $D$
```

ordered by cardinality. The ordered list is then iterated, adding all minimal hitting sets to D. As the hitting sets are inserted in an increasing cardinality order, it is not necessary to look for subsumable hitting sets (PURGE_SUBSUMED in Algorithm 1) in D.

4 Results

In order to assess the performance of our algorithm we implemented it in C++ using OpenMPI as the parallelization framework. All the benchmarks were conducted in a single computer with 2× Intel Xeon CPU X5570 @ 2.93GHz (4 cores each). Additionally, we generated several (A, e) by means of a Bernoulli distribution, parameterized by r (i.e., the probability that a component is involved in a row of A equals r) for which solutions have been computed with different parameters. In order to ease the comparison of results, *all* transactions in *all* generated cases fail[6]. For each set of parameters, we generated 50 inputs for each $r \in \{0.25, 0.5, 0.75\}$, and the results represent the average of the observed metrics. Both due to space constraints and the fact that the diagnosis efficiency of the algorithm has already been studied in [2], we only analyze the performance gains obtained from the parallelization.

4.1 Benchmark 1

In the first benchmark, we aimed at observing the behavior of MHS^2 for small scenarios ($N = 40, M = 40, L = 2$) where all minimal candidates can be calculated. In Figure 3, we observe both the speedup (defined as $Sup(np) = \frac{T_1}{T_{np}}$, where T_{np} is the time needed to solve the problem for np processes) and the

[6] 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 minimal hitting set (the empty set) whereas, for the second scenario, a potentially large number of minimal hitting sets may exist. As it is demonstrated in this section, the number of minimal hitting sets has a large impact in the algorithm's run-time.

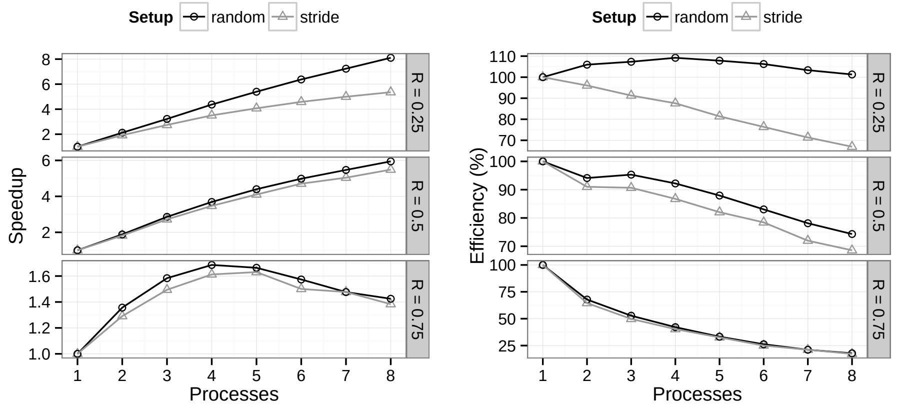


Fig. 3: Small scenarios speedup (left) and efficiency (right)

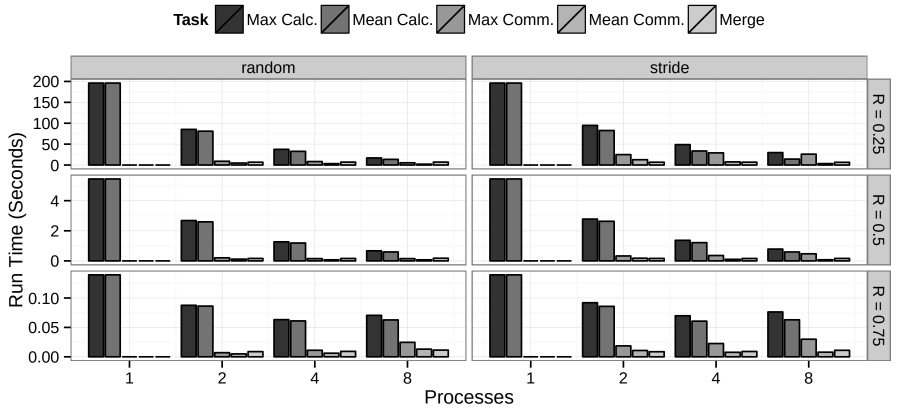


Fig. 4: Small scenarios time distribution

efficiency (defined as $Ef(np) = \frac{Sup(np)}{np}$) for both the *stride* and *random* load distribution functions.

The analysis of Figure 3 shows different speedup/efficiency patterns for r values. Despite, *random* consistently outperforms *stride*. Additionally, for $r = 0.75$ and in contrast to $r \in \{0.25, 0.5\}$, the speedup/efficiency is low (note the differences in y-axis scales). The observed speedup/efficiency patterns can be explained by analyzing Figure 4 where the total runtime is divided amongst the composing tasks (calculation, communication, and the merging of results). Additionally the maximum times for calculation and communication are shown to compare them with the respective mean values. First, it is important to note that the maximum runtime for different values of r varies substantially: ≈ 200 seconds for $r = 0.25$ vs. > 0.2 seconds for $r = 0.75$. This difference in runtimes exists due to the fact that for higher values of r, both the size and amount of minimal candidates tends to be smaller and, due to the optimization proposed

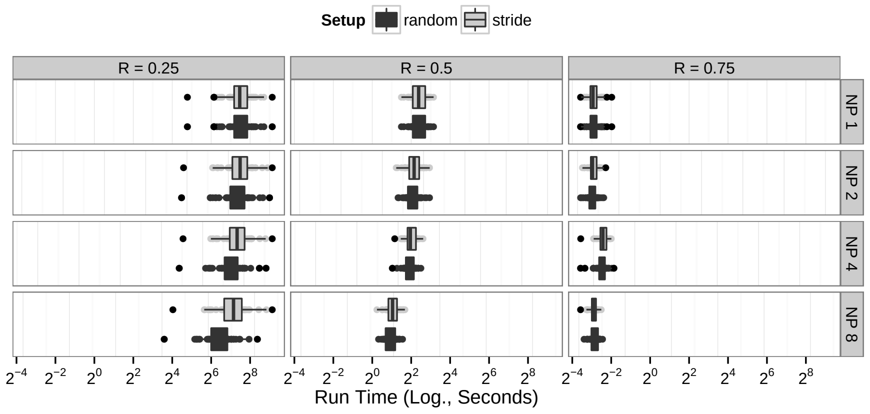


Fig. 5: Small scenarios runtime

in this paper, the number of candidates analyzed is also smaller. On the one hand, when the runtime is smaller, the parallelization overheads represent a higher relative impact in the performance (in extreme cases, the runtime can even increase with the number of processes). On the other hand, when both the cardinality and the amount of minimal hitting sets increase (small values of r) the parallelization overhead becomes almost insignificant. In such cases a larger efficiency difference between *stride* and *random* is observed due to a better load division by the *random* mechanism.

In Figure 4 this is visible when comparing the time bars for the maximum and mean calculation and communication times (actually, as the communication time includes waiting periods, it is dependent on the calculation time). For the *random* case, the maximum and mean calculation times are almost equal thus reducing waiting times. In contrast, in the *stride* case, the maximum and mean calculation times are uneven and, as a consequence, the communication overhead becomes higher: average ≈ 7 seconds for *random* vs. ≈ 28 seconds for *stride*. In scenarios where a large number of hitting sets exist and due to the fact of the function PURGE_SUBSUMED having a complexity greater than $O(n)$, the efficiency of the *random* load division can be greater than 100%. While good results are also observable for $r = 0.5$, the improvement is less significant.

Table 1: Small scenarios means and standard deviations

	R=0.25				R=0.5				R=0.75			
	Random		Stride		Random		Stride		Random		Stride	
NP	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd	mean	sd
1	196.08	94.42	196.08	94.42	5.44	1.43	5.44	1.43	0.14	0.03	0.14	0.03
2	92.54	43.47	102.08	49.16	2.89	0.68	2.99	0.74	0.10	0.01	0.10	0.02
4	44.89	21.77	55.96	27.85	1.47	0.28	1.56	0.31	0.08	0.01	0.08	0.01
8	24.20	11.23	36.61	18.36	0.91	0.16	0.99	0.18	0.10	0.01	0.09	0.01

Finally, in Figure 5 the runtime distribution of the tests is plotted and in Table 1 both the means and standard deviations are presented. Both the figure and the table show that for the same generation parameters (r, M, N and L) the runtime exhibits a considerable variance (note that the x-axis has a logarithmic scale). It is important to note that, while the value of r has an important effect

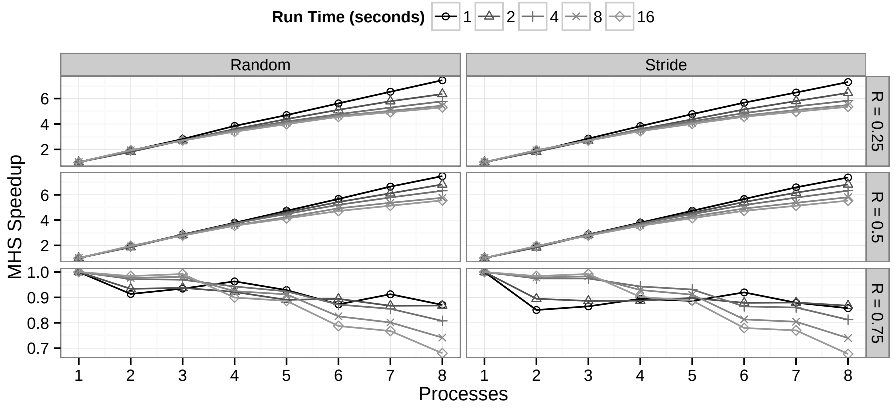


Fig. 6: Big scenarios MHS speedup

on the performance, the real key for efficiency is the problem complexity (i.e., the time needed to solve the problem). For complex enough (but still calculable) problems, the efficiency of the algorithm would increase asymptotically to 100% (or even past 100%) as the polynomial parallelization overhead would eventually be overwhelmed by the exponential complexity of the problem.

4.2 Benchmark 2

The second benchmark is aimed at observing the behavior of MHS^2 for realistic scenarios ($N = 100, M = 10000, L = 2$) where it is impractical to calculate all minimal candidates. In all the following scenarios a time based cut-off was implemented and we define the metric $CSup(np) = \frac{|D_1|}{|D_{np}|}$, where $|D_{np}|$ is the number of candidates generated using np processes for the same time, henceforward referred to as *MHS speedup*. While higher values of this metric do not necessarily mean higher diagnosis capabilities[7], in most cases they are positively correlated, due to the usage of the heuristic ranking function (see [2]).

Figures 6 and 7 show the results of computing candidates for big problems for runtimes $rt \in \{1, 2, 4, 8, 16\}$ and $np \in [1..8]$ (entailing a total runtime of $rt \times np$). It is clear that, for big problems, there is no significant difference in terms of the amount of generated candidates between *random* and *stride*. Figure 6 shows that for $r \in \{0.25, 0.5\}$ the MHS speedup scales well with the number of processes, however as the time increases, it becomes harder to find new minimal candidates (Figure 7). Regarding $r = 0.75$ we see that the MHS speedup pattern is not as smooth as for the former cases. Additionally it is important to note that, in contrast to the cases $r \in \{0.25, 0.5\}$, for $r = 0.75$, the number of candidates generated by $np = 8$ is smaller than for all other cases (Figure 7). This is due to the fact that for higher values of r both the cardinality

[7] As an example consider a set of failures for which the true explanation would be $d = \{1\}$ and two diagnostic collections $D^1 = \{\{1\}\}$ and $D^2 = \{\{1, 2\}, \{1, 3\}\}$. While $|D^2| > |D^1|$, D^1 has better diagnostic capabilities than D^2 as $d \in D^1 \wedge d \notin D^2$.

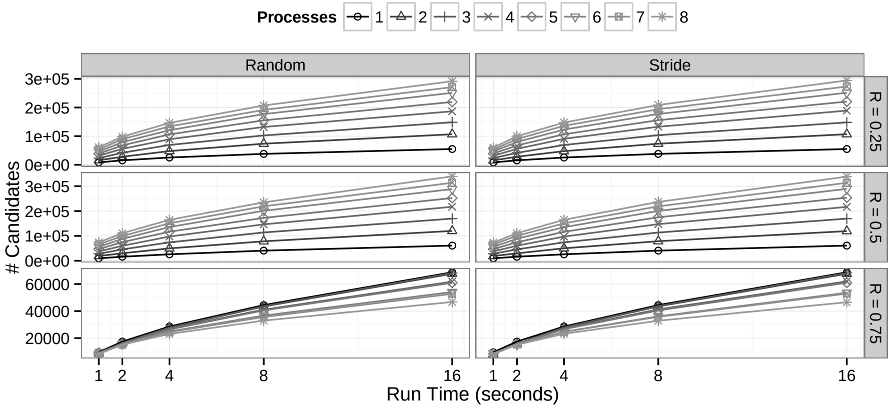


Fig. 7: Big scenarios number of candidates

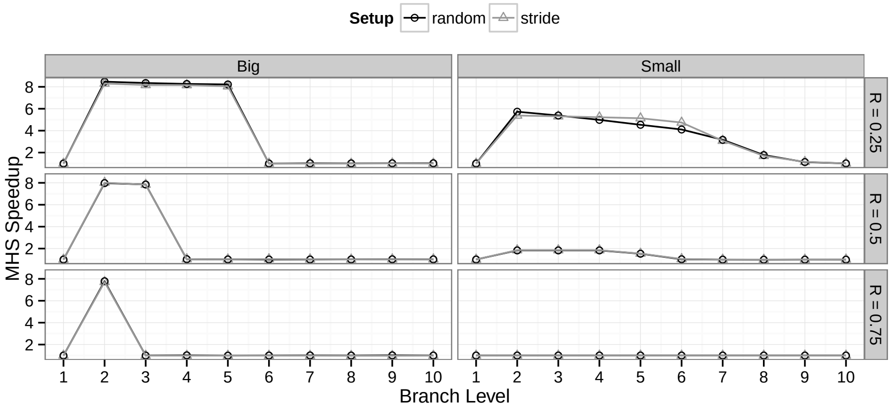


Fig. 8: Level parameter impact

and the number of minimal candidates becomes smaller, enabling the algorithm to explore a larger percentage of the search tree. As a consequence, and due to the limitations of an heuristic search, it happens that some of the candidates found first are subsumable by candidates found later, reducing the candidate collection size over time.

4.3 Benchmark 3

The final benchmark is aimed at observing the influence of parameter L in the number of generated candidates. In this benchmark we calculated candidates for both big and small problems using $np = 8$ and $rt = 10$ and $L \in [1..10]$. The analysis of Figure 8 reveals the great impact L has on the number of generated candidates. In the conducted experiments no MHS speedup lesser than 1 was observed, meaning that it should be a sensible choice to set L to be greater than

1. Optimal selection of values for L yielded an eightfold performance improvement for all the big scenarios. In the small scenarios, this improvement is still observable but with lesser magnitude. For the small scenarios with $r = 0.75$, L play no apparent role as the MHS speedup is always 1. This is due to the fact that all minimal candidates were always calculable within the defined time frame. A closer inspection of the data revealed an isolated threefold speedup peak at $L = 2$.

Another interesting pattern is the correlation between the Bernoulli distribution parameter r and the number of near-optimal values for L. This can be explained by using the argument relating r and the candidate size. The average candidate sizes (in optimal conditions) for $r \in \{0.25, 0.5, 0.75\}$ were $\{8, 6, 3\}$ for the small scenarios and $\{6, 4, 3\}$ for the large scenarios. If we observe, for each plot, the last value of L for which the performance is near optimal we see that it matches the average candidate size minus 1. Even though several levels may be near optimal, it is better to use the smaller still near optimal value for L ($L = 2$ for all the conducted experiments) as it implies less redundant calculation with an acceptable level of granularity for load division.

As a final note, although all benchmarks were conducted within a single host, implying low communication latencies, we expect that the algorithm is able to efficiently perform in distributed environments. In the conducted experiments, the communication sizes (using a non-compressed binary stream) were bounded by a 3.77 megabytes maximum.

5 Conclusions

In this paper, we proposed a distributed algorithm aimed at computing diagnosis candidates for a set of failing observations, dubbed MHS^2. This algorithm is not only more efficient in single CPU scenarios than the existent STACCATO algorithm but also is able to efficiently use the processing power of multiple CPUs to calculate minimal diagnosis candidates.

The results showed that, specially for large problems, the algorithm is able to 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 “actual” set of failing components.

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. Additionally, it would be interesting to study the performance in massively parallel computing Hadoop-based infrastructures.

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, and by the scholarship number SFRH/BD/79368/2011 from Fundação para a Ciência e Tecnologia (FCT).

References

1. Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, November 2009.
2. Rui Abreu and Arjan &GemundVanvan 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.
3. Rui Abreu, Peter Zoeteweij, and Arjan &GemundVanvan Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques*, TAICPART'07, pages 89–98, 2007.
4. 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.
5. Alexander Feldman, Gregory Provan, and Arjan &GemundVanvan 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. Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.
7. 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.
8. B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.
9. 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.
10. Johan &KleerDede Kleer and Brian C. Williams. Readings in model-based diagnosis. In *Readings in model-based diagnosis*, chapter Diagnosing multiple faults, pages 100–117. 1992.
11. 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.
12. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence.*, 32(1):57–95, 1987.
13. 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.
14. Franz Wotawa. A variant of Reiter's hitting-set algorithm. *Information Processing Letters*, 79(1):45–51, 2001.
15. 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.