Parallelized Hitting Set Computation for Model-Based Diagnosis

Dietmar Jannach[a], **Thomas Schmitz**[a], **Kostyantyn Shchekotykhin**[b]
[a]TU Dortmund, Germany
[b]Alpen-Adria Universität Klagenfurt, Austria
{firstname.lastname}@tu-dortmund.de, kostya@ifit.uni-klu.ac.at

Abstract

Model-Based Diagnosis techniques have been successfully applied to support a variety of fault-localization tasks both for hardware and software artifacts. In many applications, Reiter's hitting set algorithm has been used to determine the set of all diagnoses for a given problem. In order to construct the diagnoses with increasing cardinality, Reiter proposed a breadth-first search scheme in combination with different tree-pruning rules. Since many of today's computing devices have multi-core CPU architectures, we propose techniques to parallelize the construction of the tree to better utilize the computing resources without losing any diagnoses. Experimental evaluations using different benchmark problems show that parallelization can help to significantly reduce the required running times. Additional simulation experiments were performed to understand how the characteristics of the underlying problem structure impact the achieved performance gains.

Introduction

Model-Based Diagnosis (MBD) is a principled and domain-independent way of determining the possible reasons why a system does not behave as expected [de Kleer and Williams, 1987; Reiter, 1987]. MBD was initially applied to find faults in hardware artifacts like electronic circuits. Later on, due to its generality, the principle was applied to a number of different problems including the diagnosis of VHDL and Prolog programs, knowledge bases, ontologies, process descriptions and Java programs or spreadsheet programs [Friedrich, Stumptner, and Wotawa, 1999; Console, Friedrich, and Dupré, 1993; Felfernig et al., 2004; Friedrich and Shchekotykhin, 2005; Mateis et al., 2000; Jannach and Schmitz, 2014; Felfernig et al., 2009].

MBD approaches rely on an explicit description of the analyzed system. This includes the system's components, their interconnections, and their normal "behavior". Given some inputs and expected outputs for the system, a diagnosis task is initiated when there is a discrepancy between what is expected and observed. The task then consists in finding minimal subsets of the components, i.e., *diagnoses*, which, if assumed to be faulty, explain the observed outputs.

To focus the search, [Reiter, 1987] relied on *conflicts*, which are subsets of the system's components that expose an unexpected behavior given the inputs and the observations. The set of diagnoses for a system corresponds to the minimal *hitting set* of the conflicts. To determine the hitting sets, a breadth-first procedure was proposed leading to the construction of a hitting set tree (HS-tree), where the nodes are labeled with conflicts and the edges are labeled with elements of the conflicts. The breadth-first principle ensures that the generated diagnoses contain no superfluous elements. Techniques like conflict re-use and tree pruning help to further reduce the search effort.

The most costly operation during tree construction in many application settings is to check if a new node is a diagnosis for the problem and, if not, to calculate a new conflict for the node using a "theorem prover" call [Reiter, 1987]. When applying MBD, e.g., to diagnose a specification of a Constraint Satisfaction Problem (CSP), we would need to determine at each node if a relaxed version of the CSP – we assume some constraint definitions to be faulty – has a solution. If not, a method like QUICKXPLAIN [Junker, 2004] could be used to find a minimal conflict, which, in turn, requires a number of relaxations of the original CSP to be solved. In practice, the overall diagnosis running time mainly depends on how fast a consistency check can be done, since during this time, the HS algorithm has to wait and cannot expand further nodes. However, today's desktop computers and even mobile phones and tablets have multiple cores. Therefore, if the calculations are done sequentially, most of the processing power remains unused.

In this work, we address application scenarios like the ones mentioned above in which (a) the conflicts are generated "on-demand" and are not available before the diagnosis step and (b) where we are interested in finding *all* minimal diagnoses, which corresponds to a (worst case) scenario that is common in many of the applications mentioned above. In particular, we propose to parallelize the HS-tree construction process to better utilize the potential of modern computer architectures.

Reiter's Theory of Diagnosis

A formal characterization of model-based diagnosis based on first-order logic is given in [Reiter, 1987] and can be summarized as follows.

Definition 1. *(Diagnosable System) A diagnosable* system *is described as a pair* (SD, COMPONENTS) *where* SD *is a system description (a set of logical sentences) and* COMPONENTS *represents the system's components (a finite set of constants).*

The normal behavior of components is described by using a distinguished (usually negated) unary predicate AB(.), meaning "abnormal". A diagnosis problem arises when some observation $o \in$ OBS of the system's input-output behavior (again expressed as first-order sentences) deviates from the expected system behavior.

Definition 2. *(Diagnosis) Given a diagnosis problem* (SD, COMPONENTS, OBS)*, a diagnosis is a minimal set* $\Delta \subseteq$ COMPONENTS *such that* SD $\cup$ OBS $\cup \{AB(c) | c \in \Delta\} \cup \{\neg AB(c) | c \in$ COMPONENTS$\backslash\Delta\}$ *is consistent.*

In other words, a diagnosis is a minimal subset of the system's components, which, if assumed to be faulty (and thus behave abnormally) explain the system's behavior, i.e., are consistent with the observations.

Finding all diagnoses can in theory be done by simply trying out all possible subsets of COMPONENTS and checking their consistency with the observations. In [Reiter, 1987], Reiter however proposes a more efficient procedure based on the concept of conflicts.

Definition 3. *(Conflict) A conflict for* (SD, COMPONENTS, OBS) *is a set* $\{c_1, ..., c_k\} \subseteq$ COMPONENTS *such that* SD $\cup$ OBS $\cup \{\neg AB(c_1), ..., \neg AB(c_k)\}$ *is inconsistent.*

A conflict thus corresponds to a subset of the components, which, if assumed to behave normally, are not consistent with the observations. A conflict c is considered to be *minimal*, if there exists no proper subset of c which is also a conflict.

Reiter finally shows that finding all diagnoses can be accomplished by finding the minimal hitting set of the given conflicts. Furthermore, he proposes a breadth-first search procedure and the construction of a corresponding hitting set tree (HS-Tree) to determine the minimal diagnoses. Later on, Greiner et al. in [Greiner, Smith, and Wilkerson, 1989] found a potential problem that can occur during the HS-tree construction in presence of non-minimal conflicts. To fix the problem, they proposed an extension to the algorithm which is based on a directed acyclic graph (DAG) instead of the HS-tree.

In this paper, we will use the original HS-tree variant to simplify the presentation of our parallelization approaches; the same scheme can however also be applied to the HS-DAG version proposed by [Greiner, Smith, and Wilkerson, 1989]. Note that in our experiments the additional tree-pruning step in case of non-minimal conflicts is not required, since our approach is based on QUICKXPLAIN – a conflict detection technique which guarantees to return only minimal conflicts.

Reiter's HS-Tree Algorithm

Figure 1 shows how an example HS-tree is constructed. After an unexpected behavior was detected, a first conflict, in this case {C1,C2,C3}, is computed and used as a label for the tree's root note (①). The tree is then expanded in left-to-right breadth first manner. Each new node is either labeled with a new conflict that is created on-demand (②,③), closed because of pruning rules (④,⑤) or represents a diagnosis (marked with ✓). The diagnoses finally correspond to the path labels of the nodes marked with ✓, i.e., {{C2}, {C1,C4}, {C3,C4}} in this example.

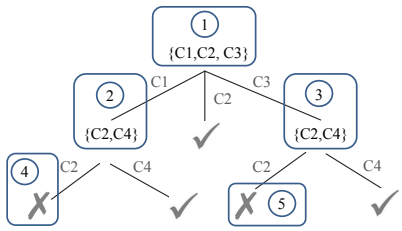

Figure 1: Example for HS-tree construction.

Algorithm 1 shows the main loop of a non-recursive implementation of a breadth-first procedure that maintains a list of open nodes to be expanded. Algorithm 2 contains the node expansion logic and includes mechanisms for conflict reuse, tree pruning and the management of the lists of known conflicts, paths and diagnoses. We use this implementation as a basis for illustrating our new parallelization schemes.

Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set Δ of diagnoses

$\Delta = \emptyset$; paths $= \emptyset$; conflicts $= \emptyset$; newNodes $= \emptyset$;
rootNode = CREATEROOTNODE(SD, COMPS, OBS);
nodesToExpand = $\langle$rootNode$\rangle$;
while *nodesToExpand* $\neq \langle\,\rangle$ **do**
  newNodes = $\langle\,\rangle$;
  node = head(nodesToExpand) ;
  foreach $c \in$ *node.conflict* **do**
    EXPAND(node, c, Δ, paths, conflicts, newNodes);
  end
  nodesToExpand = tail(nodesToExpand) $\oplus$ newNodes;
end
return Δ;

Algorithm 1: DIAGNOSE: Main loop.

Algorithm 1 takes a diagnosis problem (DP) instance as input and returns the set Δ of diagnoses. The DP is given as a tuple (SD, COMPS, OBS), where SD is the system description, COMPS the set of components that can potentially be faulty, and OBS a set of observations. The method CREATEROOTNODE creates the initial node, which is labeled with a conflict and an empty path label. Within the while loop, the first element of the list of open nodes NODESTOEXPAND is taken. The function EXPAND (Algorithm 2) is called for each element of the node's conflict and it adds new leaf nodes to be explored to a global list. These new nodes are then appended ($\oplus$) to the remaining list of open nodes in the main loop which continues until no more elements remain for expansion.

The EXPAND method determines the path label for the new node and checks if the new path label is not a superset of an already found diagnosis. The function CHECKANDADDPATH

Input: An *existingNode* to expand, a conflict element $c \in$ COMPS, the sets Δ, paths, conflicts, newNodes

```
newPathLabel = existingNode.pathLabel ∪ {c};
if (∄ l ∈ Δ : l ⊆ newPathLabel) ∧
   CHECKANDADDPATH(paths, newPathLabel) then
   node = new Node(newPathLabel);
   if ∃ S ∈ conflicts : S ∩ newPathLabel = ∅ then
      node.conflict = S;
   else
      node.conflict = CHECKCONSISTENCY
            (SD, COMPS, OBS, node.pathLabel)
   end
   if node.conflict ≠ ∅ then
      newNodes = newNodes ∪ {node};
      conflicts = conflicts ∪ node.conflict;
   else
      Δ = Δ ∪ {node.pathLabel};
   end
end
```

Algorithm 2: EXPAND: Node expansion logic.

Input: Already explored *paths*, the *newPathLabel* to be explored
Result: Flag indicating successful addition

```
if ∄ l ∈ paths : l = newPathLabel then
   paths = paths ∪ newPathLabel;
   return true;
end
return false;
```

Algorithm 3: CHECKANDADDPATH: Create a path label.

(Algorithm 3) then checks if the node was not already explored elsewhere in the tree. The function returns true if the set of labels corresponding to the new path was successfully inserted into the list of known paths. Otherwise, the list of known paths remains unchanged and the node is "closed". For new nodes, either an existing conflict is reused or a new one is created with a call to the consistency checker, which tests if the new node is a diagnosis or returns a minimal conflict otherwise. Depending on the outcome, a new node is added to the list NODESTOEXPAND or a diagnosis is stored. Note that Algorithm 2 has no return value but instead modifies the sets Δ, paths, conflicts, and newNodes, which were passed as parameters.

Parallelization Approaches

Level-Wise Parallelization In this scheme (Algorithm 4), we examine all nodes *of one tree level* in parallel, collect the new nodes in the variable *newNodes*, and proceed with the next level once all nodes are processed. The breadth-first character of the search strategy is thus maintained.

The Java-like API calls have to be interpreted as follows: *threads.execute()* takes a function as a parameter and schedules it for execution in a fixed-size thread pool. Given, e.g., a pool of size 2, the expansion of the first two nodes is in parallel and the next ones are queued until one of the threads

Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set Δ of diagnoses

```
Δ = ∅; conflicts = ∅;
rootNode = GETROOTNODE(SD, COMPS, OBS);
nodesToExpand = ⟨rootNode⟩;
while nodesToExpand ≠ ⟨⟩ do
   paths = ∅; newNodes = ∅;
   for node ∈ nodesToExpand do
      for c ∈ node.conflict do
         threads.execute(EXPAND(node, c, Δ, paths,
               conflicts, newNodes));
      end
   end
   threads.await();
   nodesToExpand = newNodes;
end
return Δ;
```

Algorithm 4: Level-wise parallelization approach.

finishes. This way we ensure that the number of threads executed in parallel is smaller than the number of hardware threads or CPUs. The statement *threads.await()* is used for synchronization and blocks the execution of the subsequent code until all scheduled threads are finished.

To guarantee that the same path is not explored twice, we declare the function CHECKANDADDPATH as a "critical section", which means that no two threads can execute the function in parallel. In addition, we make the access to the global data structures (e.g., the already known conflicts or diagnoses) thread-safe so that no two threads can simultanuously manipulate them.

Full Parallelization In the level-wise scheme, there can be situations where the computation of a conflict for a specific node takes particularly long. This however means that the HS-tree expansion cannot proceed even if all other nodes of the current level are finished and many threads are idle. We therefore propose Algorithm 5, which immediately schedules every expandable node for execution and avoids such potential CPU idle times at the end of each level.

The main loop of the algorithm is slightly different and basically monitors the list of nodes to expand. Whenever new entries in the list are observed, i.e., when the last observed list size is different from the current one, we retrieve the recently added elements and add them to the thread queue for execution. The algorithm returns the diagnoses when no new elements are added since the last check and no more threads are active. In order to not actively wait for new open nodes, the main loop sleeps using the function *wait()* until it is reawakened by one of the expansion threads through a *notify()* call.

With the full parallelization approach, the search does not necessarily follow the breadth-first strategy and non-minimal diagnoses are found during the process. Therefore, whenever we find a new diagnosis d, we check if the set of known diagnoses Δ contains supersets of d and remove them from Δ. The updated parts of the EXPAND method are listed in Algorithm 6. When updating shared data structures

Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set Δ of diagnoses

$\Delta = \emptyset$; conflicts = $\emptyset$; paths = $\emptyset$;
rootNode = GETROOTNODE(SD, COMPS, OBS);
nodesToExpand = $\langle$rootNode$\rangle$;
size = 1; lastSize = 0;
while $(size \neq lastSize) \vee (threads.activeThreads \neq 0)$ **do**
  for $i = 1$ **to** $size - lastSize$ **do**
    node = nodesToExpand.get[lastSize + i];
    for $c \in node.conflict$ **do**
      threads.execute(EXPANDFP(node, c, Δ, paths, conflicts, nodesToExpand));
    end
  end
  lastSize = size;
  wait();
  size = nodesToExpand.length();
end
return Δ;

Algorithm 5: Fully parallelized HS-tree construction.

Input: An existingNode to expand, c $\in$ COMPS, sets Δ, paths, conflicts, nodesToExpand

... % same as in previous version
if $(\nexists\, l \in \Delta : l \subseteq newPathLabel) \wedge$ CHECKANDADDPATH(*paths, newPathLabel*) **then**
  ... % Obtain a conflict
  synchronized
    if $node.conflict \neq \emptyset$ **then**
      nodesToExpand = nodesToExpand $\cup$ {node};
      conflicts = conflicts $\cup$ node.conflict;
    else if $\nexists\, d \in \Delta : d \subseteq newPathLabel$ **then**
      $\Delta = \Delta \cup$ {node.pathLabel};
      for $d \in \Delta : d \supseteq newPathLabel$ **do**
        $\Delta = \Delta \setminus d$;
      end
    end
  end
end
notify();

Algorithm 6: Node expansion of the full parallelization.

(*nodesToExpand*, *conflicts*, Δ), we ensure that the threads do not interfere with each other. The mutually exclusive section is marked with the `synchronized` keyword.

Discussion of Soundness and Completeness Algorithm 1, the sequential version, is a direct implementation of Reiter's sound and complete HS-algorithm. That is, each hitting set found by the algorithm is minimal (soundness) and the algorithm finds all minimal hitting sets (completeness). Consequently, every diagnosis will be found by Algorithm 1.

Soundness: The *level-wise* algorithm retains the soundness property of Reiter's algorithm. It processes the nodes level-wise and, thus, maintains the breadth-first strategy of the original algorithm. The assumption that all conflicts returned by a conflict computation algorithm are minimal ensures that all identified hitting sets are diagnoses. The pruning rule that removes supersets of already found diagnoses can be applied as before due to the level-wise construction of the tree and guarantees that all found hitting sets are subset minimal.

In the *full parallel* approach, the minimality of the hitting sets encountered during the search is not guaranteed, since the algorithm schedules a node for processing immediately after its generation. The special treatment in the EXPANDFP function ensures that no supersets of already found hitting sets are added and that supersets of a newly found hitting set will be removed in a thread-safe manner. Consequently, the algorithm is sound in cases where it is applied to compute all possible diagnoses. In such scenarios, the algorithm will stop only if no further hitting set exists and, because of the logic of EXPANDFP, all returned hitting sets are minimal. Therefore, every element of the returned set corresponds to a diagnosis.

However, if the algorithm is applied to compute one single diagnosis, the returned hitting set of the minimal conflict sets might not be minimal, and, therefore, the output of the algorithm might not be a diagnosis. In this case the returned hitting set can be minimized through the application of an algorithm like INV-QUICKXPLAIN [Shchekotykhin et al., 2014]. Given a hitting set H for a diagnosis problem, the algorithm is capable of computing a *minimal* hitting set $\Delta \subseteq H$ requiring only $O(|\Delta| + |\Delta| \log(|H|/|\Delta|)))$ calls to a solver, where the first part, $|\Delta|$, reflects the computational costs of determining whether or not H is minimal and the second part the costs of the minimization.

Completeness: Similar to the sequential version, both proposed parallelization variants systematically explore all possible candidates. Since we do not introduce any additional pruning strategies or node closing rules none of the minimal hitting sets are removed from the resulting set.

Empirical Evaluation

In this section, we report the results of a comprehensive empirical evaluation of our parallelization approaches.

Diagnosing DXC Benchmark Problems

Dataset and Procedure For these experiments, we selected the first five systems of the DX Competition 2011 Synthetic Track[1] (see Table 1). For each system, the competition specifies 20 scenarios with injected faults resulting in faulty output values. We used the system description and the given input and output values for the diagnosis process.

The diagnosis algorithms were implemented in Java using Choco 2 as a constraint solver and QUICKXPLAIN for conflict detection. As the computation times required to find a conflict strongly depend on the order of the possibly faulty constraints, we shuffled the constraints for each test and repeated all tests 100 times. We report the wall clock times for the actual diagnosis task; the times required for input/output are independent from the HS-tree construction scheme.

Table 1 shows the characteristics of the systems in terms of the number of constraints (#C) and the problem variables

[1] https://sites.google.com/site/dxcompetition2011/

System	#C	#V	#F	#D	∅#D	∅|D|
74182	21	28	4 - 5	30 - 300	139	4.66
74L85	35	44	1 - 3	1 - 215	66.4	3.13
74283	38	45	2 - 4	180 - 4,991	1,232.7	4.42
74181*	67	79	3 - 6	10 - 3,828	877.8	4.53
c432*	162	196	2 - 5	1 - 6,944	1,069.3	3.38

Table 1: Characteristics of selected DXC benchmarks.

System	Abs. seq. [ms]	LW-P		F-P	
		$\mathbf{S}_4$	$\mathbf{E}_4$	$\mathbf{S}_4$	$\mathbf{E}_4$
74182	78	1.95	0.49	1.78	0.44
74L85	209	2.00	0.50	2.08	0.52
74283	152,835	1.52	0.38	2.32	0.58
74181*	14,534	1.79	0.45	3.44	0.86
c432*	64,150	1.28	0.32	2.86	0.71

Table 2: Observed performance gains for DXC benchmarks.

(#V)[2]. The numbers of the injected faults (#F) and the calculated diagnoses per setting vary strongly because of the different scenarios for each system. Column #D contains the range of the identified diagnoses for a system. Columns ∅#D and $\emptyset|D|$ indicate the average number of diagnoses and their average cardinality over all scenarios. As can be seen, the search tree for the diagnosis can become extremely broad with up to 6,944 diagnoses with an average diagnosis size of only 3.38 for the system c432.

Results Table 2 shows the averaged results over all scenarios when using a thread pool of size 4. We first list the running times[3] in milliseconds for the sequential version (Abs. seq.) and then the improvements of the level-wise (LW-P) and full parallelization (F-P) in terms of *speedup* and *efficiency*. Speedup S_p is computed as $= T_1/T_p$, where T_1 is the wall time when using 1 thread (the sequential algorithm) and T_p the wall time when p parallel threads are used. The efficiency E_p is defined as S_p/p and compares the speedup with the theoretical optimum.

In all tests, both parallelization approaches outperform the sequential algorithm. Furthermore, the differences between the sequential algorithm and one of the parallel approaches were statistically significant ($p < 0.05$) in 96 of the 100 tested scenarios. In most scenarios, the full-parallel variant was more efficient than the level-wise parallelization and the speedups range from 1.78 to 3.44 (i.e., up to a reduction of running times of more than 70%). Only in the very small scenario the level-wise parallelization was slightly faster due to its limited synchronization overhead.

Diagnosing Constraint Satisfaction Problems

Data Sets and Procedure In the next set of experiments, we used a number of CSP instances from the 2008 CP solver competition[4] in which we injected faults. We first generated a random solution using the original CSP formulations. From each solution, we randomly picked about 10% of the variables and stored their value assignments which then served as test cases. These stored variable assignments correspond to the *expected outcomes* when all constraints are formulated correctly. Next, we manually inserted errors (mutations) in the constraint problem formulations, e.g., by changing a "less than" operator to a "more than" operator. The diagnosis task then consists of identifying the possibly faulty constraints using the partial test cases.

In addition to the benchmark CSPs we converted a number of spreadsheet diagnosis problems from [Jannach and Schmitz, 2014] to CSPs to test the performance gains on realistic application settings. Table 3 shows the problems characteristics including the number of diagnoses (#D) and the average diagnosis size (∅|D|). In general, we selected CSPs which are quite diverse with respect to their size.

Scenario	#C	#V	#F	#D	∅ |D|
costasArray-13	87	88	2	2	2.5
e0ddr1-10-by-5-8	265	50	17	15	4
fischer-1-1-fair	320	343	9	2006	2.98
graceful–K3-P2	60	15	4	117	2.94
graph2	2245	400	14	72	3
series-13	156	25	2	3	1.3
hospital payment	38	75	4	40	4
course planning	457	583	2	3024	2
preservation model	701	803	1	22	1
revenue calculation	93	154	4	1452	3

Table 3: Characteristics of selected problem settings.

Results The measurement results are given in Table 4. Improvements could be achieved for all problem instances. For some problems, the improvements are very strong (with a running time reduction of over 50%), whereas for others the improvements are modest. Again, the full parallelization mode is not consistently better than the level-wise mode and in several cases the differences are small. For all problem instances the differences between the parallel algorithms and the sequential version were statistically significant.

The observed results indicate that the performance gains depend on a number of factors including the size of the conflicts, the computation times for conflict detection and the problem structure itself.

Systematic variation of problem characteristics

Procedure To better understand in which way the problem characteristics influence the performance gains, we used a suite of artificially created hitting set construction problems with the following varying parameters: number of components (#Cp), number of conflicts (#Cf), average size of conflicts (∅|Cf|). Given these parameters, we used a problem generator which produces a set of minimal conflicts with the

[2]For systems marked with *, the search depth was limited to their actual number of faults to ensure termination of the sequential algorithm within a reasonable time frame.

[3]We used a standard laptop computer (Intel i7-3632QM, 4 cores with Hyper-Threading, 16GB RAM) running Windows 8.

[4]See http://www.cril.univ-artois.fr/CPAI08/. To be able to do a sufficient number of repetitions, we again picked instances with comparably small running times.

Scenario	Abs. seq. [ms]	LW-P S_4	LW-P E_4	F-P S_4	F-P E_4
costasArray-13	9,640	1.70	0.42	1.88	0.47
e0ddr1-10-by-5-8	7,461	2.42	0.61	2.54	0.63
fischer-1-1-fair	461,162	1.09	0.27	1.12	0.28
graceful–K3-P2	4,019	2.47	0.62	2.64	0.66
graph-2	118,094	1.98	0.49	1.99	0.50
series-13	8,042	1.79	0.45	1.73	0.43
hospital payment	3,563	1.49	0.37	1.63	0.41
course planning	31,622	2.17	0.54	2.19	0.55
preservation model	478	1.31	0.33	1.31	0.33
revenue calculation	1,824	1.32	0.33	1.32	0.33

Table 4: Results for CSP benchmarks and spreadsheets.

desired characteristics. The generator first creates the given number of components and then uses these components to generate the requested number of conflicts. To obtain more realistic settings, not all generated conflicts were of equal size but rather varied according to a Gaussian distribution with the desired size as a mean. Similarly, not all components should be equally likely to be part of a conflict and we again used a Gaussian distribution to assign component failure probabilities.

Since the conflicts are all known in advance, the functionality of the conflict detection algorithm within the consistency check call is reduced to returning one suitable conflict upon request. Since zero computation times are however unrealistic and our assumption is that this is actually the most costly part of the diagnosis process, we varied the assumed conflict computation times to analyze their effect on the relative performance gains. These computation times were simulated by adding artificial active *waiting times* (Wt) inside the consistency check (shown in ms in Table 5). Note that the consistency check is only called if no conflict can be reused for the current node; the artificial waiting time only applies to cases in which a new conflict has to be determined.

Each experiment was repeated 100 times on different variations of each problem setting to factor out random effects. The number of diagnoses #D is thus an average as well. All algorithms had however to solve identical sets of problems and thus returned identical diagnoses. We limited the search depth to 4 for all experiments to speed up the benchmark process. The average running times are reported in Table 5.

Results *Varying computation times:* First, we varied the assumed conflict computation times for a quite small diagnosis problem using 4 parallel threads (Table 5). The first row with assumed zero computation times shows how long the HS-tree construction alone needs. The improvements of the parallelization are modest for this case because of the overhead of thread creation. However, as soon as the average running time for the consistency check is assumed to be 1ms, both parallelization approaches result in a speedup of more than 3. Further increasing the assumed computation time does not lead to better relative improvements.

Varying conflict sizes: The average conflict size impacts the breadth of the HS-tree. Next, we therefore varied the average conflict size. Our hypothesis was that larger conflicts

#Cp, #Cf, ∅|Cf|	#D	Wt [ms]	Seq. [ms]	LW-P S_4	LW-P E_4	F-P S_4	F-P E_4
Varying computation times Wt							
50, 5, 4	25	**0**	23	2.26	0.56	2.58	0.64
50, 5, 4	25	**1**	466	3.09	0.77	3.11	0.78
50, 5, 4	25	**10**	483	2.98	0.75	3.10	0.77
50, 5, 4	25	**100**	3,223	2.83	0.71	2.83	0.71
Varying conflict sizes							
50, 5, **6**	99	10	1,672	3.62	0.91	3.68	0.92
50, 5, **9**	214	10	3,531	3.80	0.95	3.83	0.96
50, 5, **12**	278	10	4,605	3.83	0.96	3.88	0.97
Varying numbers of components							
50, 10, 9	201	10	3,516	3.79	0.95	3.77	0.94
75, 10, 9	105	10	2,223	3.52	0.88	3.29	0.82
100, 10, 9	97	10	2,419	3.13	0.78	3.45	0.86
#Cp, #Cf, ∅|Cf|	#D	Wt [ms]	Seq. [ms]	LW-P S_8	LW-P E_8	F-P S_8	F-P E_8
Adding more threads (8 instead of 4)							
50, 5, **6**	99	10	1,672	6.40	0.80	6.50	0.81
50, 5, **9**	214	10	3,531	7.10	0.89	7.15	0.89
50, 5, **12**	278	10	4,605	7.25	0.91	7.27	0.91

Table 5: Simulation results.

and correspondingly broader HS-trees are better suited for parallel processing. The results shown in Table 5 confirm this assumption. The full parallel version is always slightly more efficient than the level-wise parallel version.

Adding more threads: For larger conflicts, adding more additional threads leads to further improvements. Using 8 threads results in improvements of up to 7.27 (corresponding to a running time reduction of over 85%) for these larger conflict sizes, as here even higher levels of parallelization can be achieved.

Adding more components: Finally, we varied the problem complexity by adding more components that can potentially be faulty. Since we left the number and size of the conflicts unchanged, fewer diagnoses were found when restricting the search level, e.g., to 4, as done in this experiment. As a result, the relative performance gains were lower than when there are fewer components (constraints).

Discussion: The simulation experiments demonstrate the advantages of parallelization. In all tests both parallelization approaches were statistically significantly faster than the sequential algorithm. The results also confirm that the performance gains can depend on different characteristics of the underlying problem. The additional gains of not waiting at the end of each search level for all worker threads to be finished typically leads to small further improvements.

Redundant calculations can however still occur, in particular when the conflicts for new nodes are determined in parallel and two worker threads return the same conflict. Although without parallelization the computing resources would have been left unused anyway, redundant calculations can lead to overall longer computation times for very small problems because of the thread synchronization overheads.

Relation to Previous Work

Over the years, a number of approaches were proposed for finding diagnoses more efficiently than with Reiter's proposal, which can be divided into exhaustive and approximate ones. The former perform a sound and complete search for all minimal diagnoses; the latter often improve the computational efficiency in exchange for completeness, i.e., they for example search for only one or a small set of diagnoses.

Approximate approaches can, e.g., be based on stochastic search techniques like genetic algorithms [Li and Yunfei, 2002] or greedy stochastic search [Feldman, Provan, and van Gemund, 2010]. The greedy method proposed in [Feldman, Provan, and van Gemund, 2010], for example, uses a two-step approach. In the first phase, a random and possibly non-minimal diagnosis is determined, which is then minimized in the second step by repeatedly applying random modifications. In the approach of Li and Yunfei the genetic algorithm takes a number of conflict sets as input and generates a set of bit-vectors (chromosomes), where every bit encodes a truth value of an atom over the AB(.) predicate. In each iteration the algorithm applies genetic operations, such as mutation, crossover, etc., to obtain new chromosomes. Then, all obtained bit-vectors are evaluated by a "hitting set" fitting function which eliminates bad candidates. The algorithm stops after a predefined number of iterations and returns the best diagnosis. In general, such approximate approaches are not directly comparable with ours since they are incomplete. Our goal in contrast is to improve the performance while at the same time maintaining the completeness property.

Another way of finding approximate solutions is to use heuristic search approaches. In [Abreu and van Gemund, 2009], for example, Abreu and van Gemund suggest the STACCATO algorithm which applies a number of heuristics for pruning the search space. More "aggressive" pruning techniques result in better performance of the search algorithms. However, they also increase the likelihood that some of the diagnoses will not be found. In this approach the "aggressiveness" of the heuristics can be varied through input parameters depending on the application goals. Later on, Cardoso and Abreu suggested a distributed version of the STACCATO algorithm [Cardoso and Abreu, 2013], which is based on the Map-Reduce scheme [Dean and Ghemawat, 2008] and can therefore be executed on a cluster of servers. Other more recent algorithms focus on the efficient computation of one or more minimum cardinality (*minc*) diagnoses [de Kleer, 2011]. Both in the distributed approach and in the minimum cardinality scenario, the assumption is that the (possibly incomplete) set of conflicts is already available as an input at the beginning of the hitting-set construction process. In the application scenarios that we address with our work, finding the conflicts is considered to be the computationally expensive part and we do not assume to know all/some minimal conflicts in advance but rather to compute them "on-demand" [Pill, Quaritsch, and Wotawa, 2011].

Exhaustive approaches are often based on HS-trees and, e.g., use a tree construction algorithm that reduces the number of pruning steps in presence of non-minimal conflicts [Wotawa, 2001]. Alternatively, one can use methods that compute diagnoses without the explicit computation of conflict sets, i.e., by solving a problem dual to minimal hitting sets [Satoh and Uno, 2005]. Stern et al., for example, suggest a method that explores the duality between conflicts and diagnoses and uses this symmetry to guide the search [Stern et al., 2012]. Other approaches exploit the structure of the underlying problem, which can be hierarchical [Autio and Reiter, 1998], tree-structured [Stumptner and Wotawa, 2001], or distributed [Wotawa and Pill, 2013]. Parallel exhaustive algorithms – see [Burns et al., 2010] for an overview – cannot be efficiently applied in our approach. Like most search algorithms they assume that the main search overhead is due to the simultaneous expansion of the same node by parallel threads. The extra effort caused by the generation of nodes is ignored as this operation can be done fast. The latter is not the case in our approach since the generation of conflicts, i.e., nodes of the HS-tree, is time consuming.

Finally, some diagnosis problems – but not all, like ontology debugging [Friedrich and Shchekotykhin, 2005] – can be encoded as SAT problems [Metodi et al., 2012] or CSPs [Nica and Wotawa, 2012; Nica et al., 2013] allowing multi-threaded solvers to compute diagnoses in parallel. To ensure minimality of the diagnoses, such algorithms implement an iterative deepening strategy and increase the cardinality of the diagnoses to search for in each iteration. This parallelization approach roughly corresponds to our level-wise one and a performance comparison is part of our future work.

Summary

We propose to parallelize Reiter's algorithm to speed up the computation of diagnoses. In contrast to many heuristic or stochastic approaches, our parallel algorithms are designed for scenarios where all minimal diagnoses are needed. At the same time, the parallelization schemes are independent of the underlying problem structure and encoding. Our experimental evaluation showed that significant performance improvements can be obtained through parallelization.

In our future work, we plan to investigate if there are certain problem characteristics which favor the usage of one or the other parallelization scheme. In addition we plan to do an evaluation regarding the additional memory requirements that are caused by the parallelization of the tree construction.

Regarding algorithmic enhancements, we furthermore will investigate how information about the underlying problem structure can be exploited to achieve a better distribution of the work on the parallel threads and to thereby avoid duplicate computations. In addition, messages between threads could be used to inform a thread in case the currently processed node has become irrelevant and can be closed or when newly found conflicts can potentially be reused. Furthermore, we plan to explore the usage of parallel solving schemes for the dual problem.

Acknowledgements

This work was supported by the EU through the programme "Europäischer Fonds für regionale Entwicklung - Investition in unsere Zukunft" under contract number 300251802.

References

Abreu, R., and van Gemund, A. J. C. 2009. A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In *SARA '09*, 2–9.

Autio, K., and Reiter, R. 1998. Structural Abstraction in Model-Based Diagnosis. In *ECAI '98*, 269–273.

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-First Heuristic Search for Multicore Machines. *JAIR* 39:689–743.

Cardoso, N., and Abreu, R. 2013. A Distributed Approach to Diagnosis Candidate Generation. In *EPIA '13*, 175–186.

Console, L.; Friedrich, G.; and Dupré, D. T. 1993. Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In *IJCAI '93*, 1494–1501.

de Kleer, J., and Williams, B. C. 1987. Diagnosing Multiple Faults. *Artificial Intelligence* 32(1):97–130.

de Kleer, J. 2011. Hitting set algorithms for model-based diagnosis. In *DX Workshop '11*, 100–105.

Dean, J., and Ghemawat, S. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM* 51(1):107–113.

Feldman, A.; Provan, G.; and van Gemund, A. 2010. Approximate Model-Based Diagnosis Using Greedy Stochastic Search. *Journal of Artifcial Intelligence Research* 38:371–413.

Felfernig, A.; Friedrich, G.; Jannach, D.; and Stumptner, M. 2004. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence* 152(2):213–234.

Felfernig, A.; Friedrich, G.; Isak, K.; Shchekotykhin, K. M.; Teppan, E.; and Jannach, D. 2009. Automated debugging of recommender user interface descriptions. *Applied Intelligence* 31(1):1–14.

Friedrich, G., and Shchekotykhin, K. M. 2005. A General Diagnosis Method for Ontologies. In *ISWC '05*, 232–246.

Friedrich, G.; Stumptner, M.; and Wotawa, F. 1999. Model-Based Diagnosis of Hardware Designs. *Artificial Intelligence* 111(1-2):3–39.

Greiner, R.; Smith, B. A.; and Wilkerson, R. W. 1989. A Correction to the Algorithm in Reiter's Theory of Diagnosis. *Artificial Intelligence* 41(1):79–88.

Jannach, D., and Schmitz, T. 2014. Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Automated Software Engineering* February 2014 (published online).

Junker, U. 2004. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In *AAAI '04*, 167–172.

Li, L., and Yunfei, J. 2002. Computing Minimal Hitting Sets with Genetic Algorithm. In *DX Workshop '02*, 1–4.

Mateis, C.; Stumptner, M.; Wieland, D.; and Wotawa, F. 2000. Model-Based Debugging of Java Programs. In *AADEBUG '00*.

Metodi, A.; Stern, R.; Kalech, M.; and Codish, M. 2012. Compiling Model-Based Diagnosis to Boolean Satisfaction. In *AAAI' 12*, 793–799.

Nica, I., and Wotawa, F. 2012. ConDiag - computing minimal diagnoses using a constraint solver. In *DX Workshop '12*, 185–191.

Nica, I.; Pill, I.; Quaritsch, T.; and Wotawa, F. 2013. The route to success: a performance comparison of diagnosis algorithms. In *IJCAI '13*, 1039–1045.

Pill, I.; Quaritsch, T.; and Wotawa, F. 2011. From conflicts to diagnoses: An empirical evaluation of minimal hitting set algorithms. In *DX Workshop '11*, 203–211.

Reiter, R. 1987. A Theory of Diagnosis from First Principles. *Artificial Intelligence* 32(1):57–95.

Satoh, K., and Uno, T. 2005. Enumerating Minimally Revised Specifications Using Dualization. In *JSAI Workshops '05*, 182–189.

Shchekotykhin, K. M.; Friedrich, G.; Rodler, P.; and Fleiss, P. 2014. Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation. In *ECAI 2014*, 813–818.

Stern, R.; Kalech, M.; Feldman, A.; and Provan, G. 2012. Exploring the Duality in Conflict-Directed Model-Based Diagnosis. In *AAAI' 12*, 828–834.

Stumptner, M., and Wotawa, F. 2001. Diagnosing Tree-Structured Systems. *Artificial Intelligence* 127(1):1–29.

Wotawa, F., and Pill, I. 2013. On classification and modeling issues in distributed model-based diagnosis. *AI Communications* 26(1):133–143.

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