Parallelized Hitting Set Computation for Model-Based Diagnosis

Dietmar Jannach[1] and **Thomas Schmitz**[1] and **Kostyantyn Shchekotykhin**[2]
[1]TU Dortmund, Germany, {firstname.lastname}@tu-dortmund.de
[2]Alpen-Adria Universität Klagenfurt, Austria, 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 show that the approach significantly reduces the required running times.

1 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 [1; 2]. 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 [3; 4; 5; 6; 7; 8; 9].

MBD approaches rely on an explicit description of the analyzed system. This includes the system's components, their interconnections and the components' 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 which, if assumed to be faulty, explain the observed outputs.

To focus the search, [2] 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 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 [2]. 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 [10] 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 have multiple cores. Therefore, if the calculations are done sequentially, most of the processing power remains unused.

In this work, we propose to parallelize the HS-tree construction process to better utilize the potential of modern computer architectures and develop two sound and complete algorithm variants that are based on parallel node expansion. To evaluate the possible gains obtained through the parallelization, we run a series of experiments in which we use benchmark problems from the diagnosis and the CSP community as well as synthetic problem instances.

1.1 Reiter's Theory of Diagnosis

A formal characterization of model-based diagnosis based on first-order logic is given by Reiter in [2] 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$\setminus\Delta\}$ *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 [2], 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 [11] 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 [11]. 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.

1.2 Reiter's HS-Tree Algorithm

In this section, we give an example of applying Reiter's tree construction algorithm and present a non-recursive breadth-first search procedure, which we will use as a basis for the presentation of our parallelization approaches.

Let $\{\{C1, C2, C3\}, \{C2, C4\}\}$ be a set of minimal conflicts in our sample diagnosis problem. The minimal hitting sets and thus diagnoses for the problem would be $\{\{C2\}, \{C1, C4\}, \{C3, C4\}\}$.

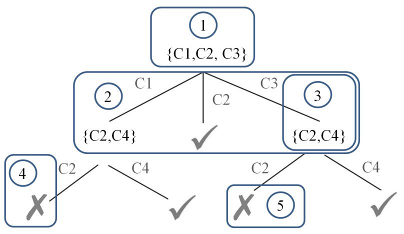

Figure 1: Example for HS-tree construction.

Figure 1 shows how the HS-tree is constructed. After the root node (①) is created, the first level of the tree is expanded (②), where {C2} is identified as a diagnosis and the corresponding node labeled with ✓. When examining the right-most node on the first level (③), the conflict {C2,C4} can be re-used as it has no overlap with the path label {C3}.

```
Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set $\Delta$ 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, $\Delta$, paths, conflicts, newNodes));
    nodesToExpand = tail(nodesToExpand) $\oplus$
                                            newNodes;
return $\Delta$;
```

Algorithm 1: DIAGNOSE(SD, COMPS, OBS): Main loop of the hitting set algorithm.

```
Input: An existingNode to expand,
       a conflict element $c \in$ COMPS,
       sets $\Delta$, paths, conflicts, newNodes
newPathLabel = existingNode.pathLabel $\cup$ {c};
if ($\nexists\, l \in \Delta : l \subseteq$ newPathLabel) $\wedge$
    CHECKANDADDPATH(paths, newPathLabel) then
    node = new Node(newPathLabel);
    if $\exists\, S \in$ conflicts : $S \cap$ newPathLabel $= \emptyset$ then
        node.conflict = S;
    else
        node.conflict = CHECKCONSISTENCY
                (SD, COMPS, OBS, node.pathLabel)
    if node.conflict $\neq \emptyset$ then
        newNodes = newNodes $\cup$ {node};
        conflicts = conflicts $\cup$ node.conflict;
    else
        $\Delta = \Delta \cup$ {node.pathLabel};
```

Algorithm 2: EXPAND(node, c, Δ, paths, conflicts, newNodes): Expansion logic.

At the next level, the node (④) with the path {C1,C2} is not expanded (marked with ✗) as it would be a superset of the known diagnosis {C2}. The same is true for the node with the path {C3,C2} (⑤).

Algorithm 1 shows the main loop of a breadth-first procedure using a list of open nodes to be expanded. Algorithm 2 presents the node expansion logic which includes Reiter's proposals for conflict re-use and tree pruning and the management of the lists of known conflicts, paths and diagnoses.

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 adds new leaf nodes to be explored to a global list. The loop 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

```
Input: The previously explored $paths$,
       the $newPathLabel$ to be explored
Result: Boolean stating if $newPathLabel$ was added to
        $paths$
if $\nexists\, l \in paths : l = newPathLabel$ then
    paths = paths ∪ newPathLabel;
    return true;
return false;
```

Algorithm 3: CHECKANDADDPATH(paths, newPathLabel): Check for a new path label.

```
Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set $\Delta$ of diagnoses
$\Delta = \emptyset$; $conflicts = \emptyset$;
rootNode = GETROOTNODE(SD, COMPS, OBS);
nodesToExpand = ⟨rootNode⟩;
while $nodesToExpand \neq \langle\rangle$ do
    $paths = \emptyset$; $newNodes = \emptyset$;
    for $node \in nodesToExpand$ do
        for $c \in node.conflict$ do
            threads.execute(EXPAND(node, c, $\Delta$, paths,
                    conflicts, newNodes));
    threads.await();
    nodesToExpand = newNodes;
return $\Delta$;
```

Algorithm 4: DIAGNOSELW(SD, COMPS, OBS): Level-wise parallelization approach.

already found diagnosis. The function CHECKANDADDPATH (Algorithm 3) then checks if the node was not already explored elsewhere in the tree. The function returns true if the new path label 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.

2 Parallelization Approaches

2.1 Level-Wise Parallelization

Our first parallelization approach examines all nodes *of one tree level* in parallel and proceeds with the next level once all elements of the level have been processed. Using this approach, the breadth-first character is maintained. However, some synchronization between threads is required, e.g., to avoid that a thread starts exploring a path which is already under examination by another thread.

Algorithm 4 shows how Algorithm 1 can be adapted to support this parallelization approach. Again, we maintain a list of open nodes to be expanded. The difference is that we run the expansion of all these nodes in parallel and collect all the nodes of the next level in the variable *newNodes*. Once the current level is finished, we overwrite the list *nodesToExpand* with the list containing the nodes of the next level. The Java-like API calls used in the pseudo-code in Algorithm 4 have to be interpreted as follows. The statement *threads.execute()* takes a function as a parameter and schedules it for execution in a pool of threads of a given size. Given, for example, a thread pool of size 2, the expansion of the first two nodes would be done in parallel and the next ones would be queued until one of the threads has finished. With this mechanism, we can ensure that the number of threads executed in parallel is less 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.

```
Input: A diagnosis problem (SD, COMPS, OBS)
Result: The set $\Delta$ of diagnoses
$\Delta = \emptyset$; conflicts = $\emptyset$; paths = $\emptyset$;
rootNode = GETROOTNODE(SD, COMPS, OBS);
nodesToExpand = ⟨rootNode⟩;
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, $\Delta$, paths,
                    conflicts, nodesToExpand));
    lastSize = size;
    wait();
    size = nodesToExpand.length();
return $\Delta$;
```

Algorithm 5: DIAGNOSEFP(SD, COMPS, OBS): Fully parallelized HS-tree construction.

To guarantee that the same path is not explored twice, we make sure that no two threads in parallel add a node with the same path label to the list of known paths. This can be achieved by declaring the function CHECKANDADDPATH as a "critical section", which means that no two threads can execute the function in parallel. Furthermore, we have to make the access to the global data structures (e.g., the already known conflicts or diagnoses) thread-safe, i.e., ensure that no two threads can simultanuously manipulate them[1].

2.2 Full Parallelization

In the level-wise parallelization approach, there can be situations where the computation of a conflict for a specific node takes particularly long. This however means that even if all other nodes of the current level are finished and many threads are idle, the HS-tree expansion cannot proceed before the level is fully completed. Algorithm 5 shows an algorithm variant that 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, it retrieves the recently added elements and adds 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. With the full parallelization approach, the search does not necessarily follow the breadth-first strategy anymore and non-minimal diagnoses are found

[1]Controlling such concurrency aspects is comparably simple in modern programming languages like Java, e.g., by using the `synchronized` keyword.

```
Input: An existingNode to expand, c ∈ 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
        $\Delta = \Delta \setminus d$;
notify();
```

Algorithm 6: EXPANDFP(node, c, Δ, paths, conflicts, nodesToExpand): Extended treatment of diagnosis supersets for the fully parallelized approach.

during the process. Therefore, whenever we find a new diagnosis d, we have to 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 the shared data structures (*nodesToExpand*, *conflicts*, and Δ), we make sure that the threads do not interfere with each other. The mutual exclusive section is marked with the `synchronized` keyword.

2.3 Discussion of Soundness and Completeness

Algorithm 1, the sequential version, is a direct implementation of Reiter's sound and complete HS-algorithm, i.e., all identified hitting sets are minimal and no diagnosis is missed.

Soundness: The *level-wise* algorithm retains the characteristics of Reiter's algorithm. The breadth-first strategy and the assumption that all found conflicts 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 diagnoses are subset minimal.

In the full parallel approach, the minimality of the diagnoses encountered during the search is not guaranteed. However, the special treatment in the EXPAND function ensures that no supersets of already found diagnoses are added and that supersets of a newly found diagnosis will be removed in a thread-safe manner.

Completeness: Similarly to the sequential version, both proposed parallelization variants systematically explore all possible candidates and we do not introduce any additional pruning strategies or node closing rules.

3 Empirical Evaluation

We conducted three types of experiments. First, we made tests with electric circuits from the DX Competition (DXC) synthetic track. Second, we diagnosed a number of CSP benchmark problems into which we artificially injected errors (*mutation testing*). Third, we used a suite of artificially created hitting set construction problems with varying characteristics. The required wall clock times for the diagnosis process was in all cases used as the performance measure.

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.

3.1 Diagnosing DXC Benchmark Problems

Dataset and Procedure For this first set of experiments, we selected the first five systems of the DX Competition 2011 Synthetic Track[2] (see Table 1). We selected these systems and limited the search depth of the last two (denoted with a *) to their actual number of faults to ensure termination of the sequential algorithm within a reasonable time frame. For every system, the competition specifies 20 scenarios with injected faults resulting in different faulty output values. We only used the system description and the given input and output values for the diagnosis process. The additionally available information about the injected faults was of course ignored.

All diagnosis algorithms examined in our experiments were implemented in Java and use the same code base. In our evaluations, we use Choco 2 as a constraint solver and QUICKXPLAIN for conflict detection. As the computation times required for conflict identification strongly depend on the order of the possibly faulty constraints, we shuffled the constraints for each test and repeated all tests 100 times. For all experiments, we report the wall clock times for the actual diagnosis task; the times needed for input and output are independent from the HS-tree construction scheme.

Table 1 shows the characteristics of each diagnosed system in terms of the number of constraints (#C) and the problem variables (#V). 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 and column ∅#D the average number of diagnoses 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 The averaged results when using a thread pool of size 4 are shown in Table 2. We first list the running times[3] for the sequential version (Abs. seq.) and then the improvements of level-wise (LW-P) and full parallelization (F-P). In this paper, all absolute running times are given in milliseconds. The improvements are measured in terms of *speedup* and *efficiency* [12]. The speedup S_p is computed as $= T_1/T_p$, where T_1 is the wall time when using 1 thread (i.e., 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 a theoretical optimum.

In all tests, both parallelization approaches outperform the sequential algorithm. On average, 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

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

[3]We used a standard desktop computer (Intel i7-3770K, 4 cores with Hyper-Threading, 16GB RAM) running Windows 7.

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.

System	Abs. seq. [ms]	LW-P S_4	LW-P E_4	F-P S_4	F-P E_4
74182	78	1.95	0.49	1.78	0.44
74L85	2090	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.

3.2 Diagnosing CSP Benchmark Problems

Data Sets and Procedure For the second set of experiments we assumed an application scenario, where a knowledge engineer made a mistake when modeling a CSP and the diagnosis task is to find these errors with the help of test cases. In the experiments, we used a number of CSP problem instances from the 2008 CP solver competition[4]. To be able to do a sufficient number of repetitions, we again picked instances with comparably small running times. The diagnosis problems were created as follows. We took the original CSP formulations and generated random solutions for them using the CSP solver. From each solution, we randomly picked about 10% of the variables and stored their value assignments which then served us 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 or equal" operator to a "less than" operator, which corresponds to a mutation-based approach in software testing. We then used the CSP solver to complete the partial test case using the modified constraint base. In cases where no solution could be found, the diagnosis task was to identify sets of possibly faulty constraints.

Table 3 shows the characteristics of the evaluated problems including the number of diagnoses (#D), the average diagnosis size (⌀|D|) and average time needed by the constraint solver to find a solution in milliseconds (⌀ST). In general, we selected CSPs which are quite diverse with respect to their size. In contrast to the previous experiments, the number and size of the conflicts and diagnoses is comparably small for most experiments.

Results The measurement results are given in Table 4. Again, we show the absolute running times for the sequential version (Abs. seq.) and report the relative improvements of the parallelized versions using 4 threads.

Overall, runtime improvements could be achieved for all problem instances. For some problems, the improvements are very strong (with a running time reduction of over 50% with 4 threads), whereas for others the improvements are modest. Again, the full parallelization mode is not consistently better than the level-wise mode. Often the differences are small and not significant. For the small problems which had average solving times of below 1ms like MKNAP-1-5 or PRIMES, we artificially added 10ms waiting time for each solver call to simulate more complex constraint propagations. Otherwise, only the level-wise parallelization leads to a slight performance increase due to the synchronization overhead of full parallelization.

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.

3.3 An evaluation based on a systematic variation of problem characteristics

Procedure To better understand in which way the problem characteristics actually influence the potential 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 desired characteristics. The generator first creates the given number of dummy components and then uses these components to generate the requested number of conflicts. In order 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 a probability value to each component[5].

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 in fact 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 milliseconds in Table 5). Note that in our algorithm the consistency check is only called if no previously found conflict can be reused for the current node, so that the artificial waiting time only applies to cases in which a new conflict has to be determined.

Again, each experiment was repeated 100 times with different variations of the problem settings to factor out random effects. The number of diagnoses #D is thus an average

[4]http://www.cril.univ-artois.fr/CPAI08/

[5]Of course, other probability distributions could be used in the generation process, e.g., to reflect specifics of a certain application domain.

Scenario	#C	#V	#F	#D	⌀ |D|	⌀ST
aim-50-1-6-3	130	100	5	12	3	< 1
c8	523	239	8	4	6.25	< 1
costasArray-13	87	88	2	2	2.5	~ 3
domino-100-100	100	100	3	81	2	< 1
e0ddr1-10-by-5-8	265	50	17	15	4	~ 35
fischer-1-1-fair	320	343	9	2006	2.98	~ 10
graceful–K3-P2	60	15	4	117	2.94	~ 1.5
graph2	2245	400	14	72	3	~ 140
mknap-1-5	7	39	1	2	1	< 1
primes-15-20-3-1	20	100	3	2	1	< 1
queens-8	28	8	15	9	10.9	< 1
series-13	156	25	2	3	1.3	~ 150

Table 3: Characteristics of selected problem settings.

Scenario	Abs. seq. [ms]	LW-P S_4	LW-P E_4	F-P S_4	F-P E_4
aim-50-1-6-3	5,245	2.09	0.52	2.32	0.58
c8	1,685	1.08	0.27	1.22	0.30
costasArray-13	7,367	1.61	0.40	1.79	0.45
domino-100-100	8,628	1.68	0.42	1.77	0.44
e0ddr1-10-by-5-8	5,875	2.34	0.59	2.31	0.58
fischer-1-1-fair	422,559	1.17	0.29	1.18	0.29
graceful–K3-P2	3,480	2.29	0.57	2.30	0.58
graph-2	94,398	1.87	0.47	1.72	0.43
mknap-1-5	383	1.26	0.31	1.30	0.33
primes-15-20-3-1	323	1.31	0.33	1.29	0.32
queens-8	4,824	1.72	0.43	2.10	0.52
series-13	7,432	1.82	0.46	1.48	0.37

Table 4: Results for CSP benchmarks.

as well. All algorithms had however to solve identical sets of problems and thus returned identical sets of diagnoses. We limited the search depth to 4 for all experiments and report the average running times.

Results *Varying computation times:* In the first set of experiments (Table 5), we varied the assumed conflict computation times for a quite small diagnosis problem using 4 parallel threads. The first row with assumed zero computation times shows how long the HS-tree construction alone needs. Parallelization does not pay off for this case and expanding the tree in full-parallel mode leads to an overhead of about 50ms. However, as soon as the average running time for the consistency check is assumed to be 1ms, parallelization helps to reduce the running times by two thirds (level-wise) and even three fourth (full parallel). Further increasing the assumed computation time does not lead to better relative improvements.

Varying conflict sizes: The average conflict size directly impacts the breadth of the HS-tree. Next, we therefore varied the average size of the conflicts. Our assumption is that larger conflicts and correspondingly broader HS-trees are better suited for parallel processing. The results show that broader conflicts in fact lead to even higher performance speedups of about 4. The full parallel version is always slightly more efficient than the level-wise parallel version. Increasing the conflict sizes further than 9 does not lead to additional improvements with 4 parallel threads.

Adding more threads: For larger conflicts, adding more additional worker threads leads to a further improvement. Using 8 threads results in improvements of up to 7.14 (corresponding to a running time reduction of over 85%) for these larger conflict sizes, as here even higher levels of parallelization can be achieved. One reason why we could not achieve the theoretical efficiency optimum of 1 when using 8 threads is that the processor used in the experiments had only 4 cores with hyper-threading.

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 constraints.

Discussion: The simulation experiments clearly demonstrate the advantage of parallelizing the HS-tree construction. The results also confirm that the obtainable 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.

#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	30	**0**	22	1.04	0.26	0.39	0.10
50, 5, 4	30	**1**	299	3.03	0.76	4.17	1.04
50, 5, 4	30	**10**	434	2.78	0.69	3.03	0.76
50, 5, 4	30	**100**	3,103	2.78	0.69	2.78	0.69
Varying conflict sizes							
50, 5, **6**	110	10	1,662	3.57	0.89	4.17	1.04
50, 5, **9**	216	10	3,214	3.70	0.93	4.35	1.09
50, 5, **12**	233	10	3,498	3.70	0.93	4.00	1.00
Varying numbers of components							
50, 10, 9	162	10	2,635	3.57	0.89	3.85	0.96
75, 10, 9	141	10	2,574	3.03	0.76	2.86	0.71
100, 10, 9	92	10	2,455	2.56	0.64	2.78	0.69
#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**	110	10	1,668	6.67	0.83	6.25	0.78
50, 5, **9**	223	10	3,311	6.25	0.78	7.14	0.89
50, 5, **12**	216	10	3,526	5.88	0.74	6.25	0.78

Table 5: Observed performance gains (CPU time) for simulated problem instances. (LW-P: Level-wise parallelization, F-P: Full parallelization)

Due to the nature of our parallelization approach, redundant calculations can 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 overhead caused by thread synchronization.

4 Relation to Previous Work

The computation of minimal hitting sets for a collection of conflicts is an NP-hard problem in MBD [13]. Moreover, deciding if an additional diagnosis exists when conflicts are computed on demand is NP-complete even for propositional Horn theories [14]. Therefore, a number of approaches were proposed for finding diagnoses more efficiently than with Reiter's proposal. These approaches can be divided into exhaustive and approximate ones. The former perform a sound and complete search for all minimal diagnoses whereas the latter 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 for example be based on stochastic search techniques like genetic algorithms [15] or greedy stochastic search [16]. The greedy method proposed in [16], 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 et al. 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 [17], for example, Abreu et al. 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 probability that some of the diagnoses will not be found. In the approach of Abreu et al. the "aggressiveness" of the heuristics can be varied by input parameters depending on the application goals. Recently, Abreu et al. suggested a distributed version of the STACCATO algorithm [18], which is based on the Map-Reduce scheme [19] 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 [20]. 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" [21].

Exhaustive approaches are often based on HS-trees like the work of Wotawa [22] – a tree construction algorithm that reduces the number of pruning steps in presence of non-minimal conflicts. 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 [23]. Stern et al. in [24], for example, suggest a method that explores the duality between conflicts and diagnoses and uses this symmetry to guide the search. Other approaches exploit the structure of the underlying problem, which can be hierarchical [25], tree-structured [26], or distributed [27]. Parallel exhaustive algorithms – see [28] 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, one can encode the diagnosis problem as a SAT problem [29] or a CSP [30]. A recent comparison of diagnostic methods made by Nica et al. in [31] showed the superiority of the SAT encoding over the methods described in [2; 22; 30]. We plan to execute a similar study using *concurrent* SAT and CSP methods in our future work.

5 Summary

We propose to parallelize Reiter's hitting set algorithm to speed up the computation of diagnoses. In contrast to many heuristic or stochastic approaches, our parallel algorithms are able to find all minimal diagnoses for a given problem. At the same time, the application of the parallelization scheme is independent of the underlying problem structure. Overall, our experimental evaluation showed that significant performance improvements can be obtained through parallelization.

In our future work, we plan to further 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 plan to 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

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

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

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

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

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

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

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

[8] D Jannach and T Schmitz. Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Automated Software Engineering*, Online February 2014(in press), 2014.

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

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

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

[12] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, Mar 1989.

[13] M R Garey and D S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[14] T Eiter and G Gottlob. The Complexity of Logic-Based Abduction. *Journal of the ACM (JACM)*, 42(1):3–42, 1995.

[15] Lin Li and Jiang Yunfei. Computing Minimal Hitting Sets with Genetic Algorithm. In *DX'02 Workshop*, pages 1–4, 2002.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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