Abstract

A diagnostic problem is a problem in which one is given a set of abnormal findings for some system, and must explain why those findings are present. Several expert systems have been developed for solving various diagnostic problems. One class of these expert systems uses association-based abduction as their inference process. Many of these systems use an approach called the set covering approach in their mechanism.

This paper develops some parallel algorithms for set covering. A brief introduction is given into the parsimonious set covering theory, and algorithms using one type of parsimony called irredundancy are developed. The paper also discusses several machine-independent parallel constructs that are used to express the parallel algorithms. The algorithms were tested on the Butterfly* parallel processor system. The paper presents some of the tests conducted and their analyses. Finally, the merits and limitations of the algorithms that were identified during the tests are presented.

1. Introduction

A diagnostic problem is a problem in which one is given a set of abnormal findings (manifestations) for some system, and must explain why those findings are present [18]. Examples of diagnostic problems include medical diagnosis, determining why a computer program failed, and finding out why a car does not start.

Several expert systems have been developed for solving various diagnostic problems. These can be broadly divided into two classes – systems using rule-based deduction, and systems that use association-based abduction ("abduction" refers to an inference process that constructs plausible explanations for observed facts) [17, 18]. Examples of these latter abductive systems include INTERNIST – used for diagnosing patients in internal medicine [15]; IDT – an intelligent hardware diagnostic tool for PDP 11/03 computers [22], and KMS.IIT – a domain independent expert system shell [16, 17].

Recently, a formal model of the inference method used in abductive diagnostic expert systems has been under study. This model, referred to as parsimonious covering theory [11, 12, 13, 18], is both being tested in practice and extended to include more complex diagnostic problems involving prior probabilities for causal relations, interacting disorders, multi-layer problems, and so forth [11, 12, 13]. Similar formal theories of diagnosis are also being developed by others [10, 19]. Because of the growing importance of this class of diagnostic expert systems and the suggestion that the methods they use may be applicable in other areas, and the large amounts of computation they involve, we investigated their suitability for a parallel MIMD computer. Parallel predicate covering algorithms have previously been proposed [9], but errors were found in these parallel algorithms [20]. The algorithms presented in [9] were used in part to motivate some of the algorithms in the research discussed in this paper.

Most of the earlier work done on set covering algorithms has been the development of sequential algorithms for set covering [10, 11, 12, 13, 17, 18]. This paper develops some parallel algorithms for set covering, and presents some tests that were conducted on these algorithms using the Butterfly parallel processor system. The algorithms are analyzed based on the test results, and an attempt is made to identify the merits and limitations of each of the algorithms. The paper also presents several machine-independent parallel constructs that were developed for expressing the parallel algorithms. More details of this work can be found in [21].

The paper is divided into several sections. First, the parallel constructs used for presenting the algorithms are briefly discussed. The next section describes some of the terminology used in discussing the algorithms. The different algorithms are then discussed. The next section discusses the implementation and testing of the algorithms on the Butterfly system. The algorithms are then analyzed for their performance based on the test results. Sample test data and test results are shown and discussed. The results of the research are then summarized, and the limitations and benefits derived from this research are discussed.

2. Parallel Constructs

Four parallel constructs are used to present the parallel set covering algorithms, three of which have not been applied significantly in the past. These constructs are used as machine-independent and can be implemented on any parallel
The first parallel construct is the \texttt{cobegin ... coend} construct, which is used widely for execution of several statements in parallel [3,6]. It states that the code between the keywords \texttt{cobegin} and \texttt{coend} can be executed in parallel or in any order, instead of the normal sequential order. The statements within the \texttt{cobegin-coend} block can be simple statements or compound statements consisting of parallel constructs defined in this section.

The second parallel construct is the \texttt{parallel for} loop. This construct is used to execute a piece of code for several values of some control variable whose values lie in a given set (or list). The construct is written as:

\begin{verbatim}
foreach <val> in <val-set> coexecute
B
coterminate;
\end{verbatim}

where, <val> is a free variable which presumably occurs in the statements in body B. <val-set> is a set of values that the free variable <val> can take, and B is the body which can contain any statement/s. The body B is executed for each of the values in the value-set. The statements within B can be sequential or parallel, and they will be executed accordingly. The order in which B is executed for different values of <val> is non-deterministic, and the body B may even be executed in parallel for different values of <val>. The \texttt{foreach} statement terminates when, for each of the values in the val-set, the body B has been executed, and control then passes to the statement following \texttt{coterminate}.

The third construct used to present the algorithms is the \texttt{with} construct, which is used for implementing critical regions while wanting to use shared memory resources. This construct is written as:

\begin{verbatim}
with <var-list> do <stmt>
\end{verbatim}

where <var-list> is a list of one or more shared variables separated by semicolon, that are needed to execute <stmt>. The shared variables must be declared as \texttt{shared-var} in the declaration. The <stmt> can be any statement including a parallel statement, which may use the shared variables in <var-list> in addition to non-shared variables. A process executing a \texttt{with} statement first makes sure that no other process is using the same variable(s) that are declared in the \texttt{with} statement, then takes control of that (those) variable(s), and bars other processes from accessing that (those) variable(s) until the time it has completed execution of the statement within the \texttt{with} construct. There is a potential for creating a deadlock when using this construct, since the shared variables are accessed in the order in which they appear, and processes which have accessed only a part of the <var-list> will bar other processes from accessing those variables while waiting for access to the remaining variables in the <var-list>. However, consistent ordering of shared variables when they appear in multiple \texttt{with} statements will avoid creating deadlocks.

The fourth construct used to present the parallel algorithms is the \texttt{terminate-processes} procedure. This procedure is used to terminate all parallel processes that were spawned by some parent process and are still currently running. This procedure can be called from any one of the parallel processes and it results in the termination of all other sibling processes that were activated by the same parent. The child process calling this procedure then retains control and can then return to the parent process with an appropriate code to indicate abnormal termination of other sibling processes. This construct is used when several child processes are started for doing some task, and one of them completes performing the task.

A reviewer pointed out that the parallel constructs presented above are all subsumed by two multiprocess primitives: Future and Store-conditional, as given in [4]. We thank the reviewer for pointing out this information; however, when this research was done, one of the objectives was to present the algorithms developed using language-independent and machine-independent parallel constructs. As a result, no existing parallel primitives of any language were adopted for presenting the algorithms, and the language-independent constructs described in this paper were formulated. These constructs were developed such that they can be implemented using other parallel primitives provided in existing parallel languages, and so, there may be several existing languages that have parallel primitives that subsume the above constructs.

3. Set Covering Algorithms

The parallel set covering algorithms discussed in this paper use two different methods. The first three algorithms present a state-space search technique for forming diagnostic solutions. The fourth algorithm uses the generalized set covering approach of representing solutions in terms of their generators. Thus, this algorithm forms solutions in clusters instead of forming them individually as is done by the first three algorithms. Before presenting the algorithms, some concepts used in the algorithms are informally explained below.

3.1. Terms and Concepts Used in the Algorithms

A diagnostic problem is one in which a set of manifestations (symptoms) is given, and explanations in terms of causality need to be formed out of a set of given disorders, according to some given criterion.

A set of disorders, \( D_t \), is said to cover a set of manifestations, \( M_j \), if, for every manifestation in \( M_j \), there is at least one disorder in \( D_t \) that causes it [11, 12, 13, 18].

The type-4 diagnostic problem is one of finding covers formed by disorders in the given set \( D \), for the manifestation-set \( M^* \), using the causal relation \( C \). Given a set \( D \) of disorders and a manifestation-set (set of symptoms) \( M^* \), there may potentially be a very large number of covers that can be constructed for \( M^* \) using the disorders in \( D \). In order to constrain the number of covers that are generated
for $M^*$, the covering algorithms, in general, use some criterion that decides on what covers will be generated. The Principle of Parsimony, or the Occam's razor principle is used by the algorithms in generating covers [18]. This principle states that simple solutions are preferred to complex solutions. Different criteria are used by covering algorithms to determine the simplicity of a solution. The algorithms in this paper use the criterion of irredundancy for generating covers. A cover $E$ is an irredundant cover for a manifestation-set $M^*$, if no proper subset of $E$ explains the presence of every manifestation in $M^*$. A covering algorithm which generates all and only the irredundant covers for a diagnostic problem is called an irredundant set covering algorithm. Other criteria of parsimony include minimal cardinality covering, relevant covering, etc.

Covering algorithms can also be classified on the basis of how they get their $M^*$ set. If the algorithm is given the full set of manifestations before the beginning of problem-solving, such a set covering algorithm is referred to as a closed set covering algorithm. If manifestations are added throughout the execution of the algorithm, such a set covering algorithm is called an open set covering algorithm. In this class of algorithms, the set of covers formed will be incrementally changed to accommodate new manifestations, as and when they are given to the algorithm. In case of a closed set covering algorithm, if new manifestations are added to $M^*$, the algorithm needs to re-generate the irredundant covers from the start. Thus, closed set covering algorithms are less powerful when compared to open set covering algorithms, but are less complex too. The algorithms considered in this paper are all closed set covering algorithms.

The fourth algorithm, which uses the generator-set approach for forming the irredundant covers, generates the irredundant covers in the form of their generators as already mentioned. A generator $G_I$ is a set of non-empty pairwise-disjoint sets of disorders. Thus, if $g_1, g_2, \ldots, g_n$ are non-empty pairwise-disjoint subsets of $D$, the set of disorders in a diagnostic problem, then $G_I = \{g_1, g_2, \ldots, g_n\}$ is a generator. The class generated by $G_I$, designated $[G_I] = \{\{d_1, d_2, \ldots, d_r\}; d_i \in g_i, 1 \leq i \leq n\}$. The individual irredundant covers that can be generated from a generator are elements of its class.

$G = \{G_1, G_2, \ldots, G_N\}$ is a generator-set if each $G_i \in G$ is a generator and $[G_I] \cap [G_J] = \emptyset$ if $I \neq J$. The class generated by $G$ is $[G] = [G_1] \cup [G_2] \cup \cdots \cup [G_N]$.

The fourth algorithm presented in this paper forms a generator-set, whose class consists of all irredundant covers for the diagnostic problem, and hence, is called the “Gen-set4” algorithm since it deals with diagnostic problems of type 4.

3.2. The Bidirectional Breadth-First Search Approach

In this section, algorithms motivated by Kornfeld’s “Combinatorially Implosive Algorithms” are presented [9]. In his paper, Kornfeld develops two algorithms called the “Top-down” and the “Bottom-up” algorithms, which are for the predicate covering problem. Their modified version, which is applicable for the set covering problem, is presented in this paper. The algorithms in this paper have been written using the parallel constructs described earlier, and thus differ from the ones presented in [9], where the parallel algorithms were presented in the ETHER language [8].

3.2.1. The Top-down Approach

This approach generates irredundant covers by taking some large set which is called the start set, tests whether the start set is itself a cover, and if so, generates all proper subsets of the start set with one element less, and tests whether each of these are covers. If any one of them is a cover, all such subsets which cover the initial set of manifestations are then taken as the start sets and the above approach is applied to each of these sets recursively. At some stage, for some start set, every subset of this set with one element less will not be a cover for the set of manifestations. Such a start set is asserted as an irredundant cover, and any previously asserted set that is a superset of this start set is discarded from the set of solutions. Since at each step, all subsets that are covers are treated as start sets, this algorithm generates all irredundant covers.

The Top-down algorithm is called initially with the set of all disorders which cover some manifestation in the set of known manifestations $M^*$, chosen from the set of disorders with which covers are to be formed. The set of known manifestations $M^*$, is also supplied to the algorithm. The algorithm returns the set of all irredundant covers for $M^*$ that can be formed out of disorders in the original set of disorders. Since any irredundant cover for a set of manifestations can have at most as many elements as the number of manifestations, the Top-down algorithm implemented in this research tests the irredundancy criterion with only covers having same or less number of elements than the number of manifestations.

3.2.2. The Bottom-up Approach

This approach generates irredundant covers by adding new elements one at a time from the set of disorders scope, (initially the set of all disorders which cover at least one manifestation in the set of known manifestations, chosen from the set of disorders with which irredundant covers are desired), to a given set that is currently not a cover (initially, this set is the empty set). This continues until the set obtained by adding some new element to a previous non-cover becomes a cover at some stage. This newly generated set either has some subset which was already found as an irredundant cover, or has no such subsets. In the former case, the newly generated set is discarded, and in the latter case, it is reported as part of the result found until that time and all supersets of this set which were previously reported as irredundant covers are discarded.

As with the Top-down algorithm, the Bottom-up algorithm implemented in this research stops forming supersets if the non-cover set with which it started contained as many elements as they are in the set of manifestations, since
adding more disorders to this set will result in only redundant covers and not irredundant ones.

3.2.4. The Bidirectional Breadth-First Search Algorithm

Kornfeld observed that:

(1) Every subset of a non-cover set is also a non-cover set.
(2) Every proper superset of an irredundant cover set is a redundant cover (i.e., a cover that is not irredundant).

These two properties can be used to reduce the work done by the two algorithms by having them execute in parallel and pass information to each other. As the two algorithms execute in parallel, they produce irredundant covers and other non-cover sets. If these sets are made available to the other algorithm, they can be used by the other algorithm for eliminating testing of several sets which may have irredundant covers as their subsets or non-cover sets as their supersets. Thus, the algorithms will reduce work on testing for covers, and replace this testing by testing for subset and superset relations in many instances. This testing is computationally less expensive than testing for covers. These observations were used by Kornfeld in developing the "combinatorially implosive algorithms" for predicate covering [9].

The parallel versions of Top-down and Bottom-up algorithms discussed in this paper share information about irredundant covers and non-cover sets which do not have a superset also as a non-cover. This compromise of storing only maximal non-cover sets and irredundant covers instead of all covers and non-covers as is done in Kornfeld's algorithms reduces the memory needed to store the global lists.

3.2.4. Parallel Breadth-First Search Algorithms

The sequential breadth-first search algorithms discussed earlier have lots of parallelism inherent in them. The parallel implementation of the Top-down algorithm, PTOPDOWN, processes all subsets of the start set in parallel, both for testing for covers and for recursive calls. Similarly, the parallel version of the Bottom-up approach, PBOTTOMUP, processes all supersets of the given non-cover set in parallel. In addition, when these algorithms search the global lists of irredundant covers (Worklist), and maximal non-covers (Notworklist) for subsets and supersets, these operations are performed in parallel for each element of the global lists. More details of the parallel versions are omitted due to lack of space. The two functions PTOPDOWN and PBOTTOMUP are called in parallel with the same global lists Worklist and Notworklist, for implementing the bidirectional algorithm.

3.3. The Generator Set Algorithm

This algorithm uses the generator-set approach for forming irredundant covers. Instead of generating individual covers, this algorithm generates a generator-set for all the irredundant covers for the diagnostic problem. This algorithm accepts a set of disorders, scope, and the set of manifestations to be covered, manif, as input parameters and generates a generator-set for all the irredundant covers for manif. The algorithm first checks for the trivial cases where either no manifestation is present or no solution is possible with the given disorders, and terminates with an appropriate reply in such a case. If the diagnostic problem does have some irredundant covers, the algorithm uses the divide-and-conquer (problem reduction) strategy to generate the generator-set. The generator-set whose class consists of all irredundant covers can be split into two parts. Let \( d \) be some disorder in scope such that it is one of the disorders that cover the maximum number of manifestations in manif, and let \( f \) be a set consisting of all disorders in scope that cover the same set of manifestations in manif as \( d \) does. Then, the set of all irredundant covers can be partitioned into two sets, one in which every irredundant cover contains one element from the set \( f \), the other in which no irredundant cover contains any element of \( f \). Corresponding to this partition, the set of generators can also be partitioned into two sets: one set of generators have \( f \) as one of their disjoint sets, the other set of generators do not have \( f \) as any of their elements.

Thus, two generator-sets result, the union of which gives the resultant generator-set whose class consists of all irredundant covers. This is the basic idea behind the operation of the Genset4 algorithm.

The parallel version of the Genset4 algorithm is the PGenset4 algorithm. The generation of the two partitioned generator-sets is done in parallel in PGenset4. By slightly modifying the details of the Genset4 algorithm, we were able to exploit more parallelism in the PGenset4 algorithm even though Genset4 algorithm as such is inherently sequential. Details are omitted due to lack of space.

4. Implementation and Testing

The algorithms discussed in this paper were implemented on a Butterfly parallel processor system, which is a shared memory MIMD machine [1,2]. It consists of 112 MC68000 processors, each with 1 MByte of memory associated with it. Any processor can access memory associated with any other processor by means of a Butterfly switch.

One approach to program the Butterfly Parallel Processor System is the Uniform System approach. The Uniform System treats processors as a group of identical workers, each able to do any task. A program is viewed as consisting of two parts: a set of subroutines performing various application tasks; and one or more "generators" that identify the next task to be performed. The generator discussed in the context of the Butterfly system is referred to, henceforth, as B-generator to avoid confusion with the generator defined in an earlier section. A B-Generator G takes as parameters a worker procedure (W) and a description of data upon which work is to be done. It then calls the worker procedure W, specifying subsets of the data to work upon, as often as necessary to complete the work. When a B-generator is invoked, the system broadcasts the B-generator and the worker procedure to the available processors, which start generating and executing tasks. When the last task is completed, the system returns to the original caller.

The parallel processes share data by writing it into the shared memory. In addition, they use the process-private
memory for computations that are not shared. Thus, all heap and stack variables in C are process-private, while globally shared data-structures are stored in the Uniform System allocated shared storage.

A B-generator, GenOnList, was written for implementing the parallel constructs. The cobegin-coend construct was implemented using the GenOnList procedure, and the system provided generator, GenOnIndex. The foreach construct was implemented using the GenOnList procedure. The with parallel construct was implemented in a number of ways. If the shared list consisted of a single variable, then a suitable system-provided atomic operation (such as the Atomic Add operation) was used. If the shared memory is not a single variable, then a critical section is created using the LOCK and the UNLOCK operations of the operating system. This problem is the same as the classical Readers-Writers problem encountered in operating systems[5, 7, 14], and is solved in a similar manner. The terminate-processes construct is encountered within the foreach construct, and was implemented by writing an abortable GenOnList routine.

The parallel algorithms discussed earlier were tested with different test data cases, and measurements of their execution time were taken with different numbers of processors. The ratio of the execution time of an algorithm on one processor (in cases where the algorithm could not be run on one processor, the time on the smallest number of processors on which the algorithm is run is taken as the base to compute the ratio) to the execution time on \( n \) processors is termed the speedup of the algorithm, where \( n \) is the number of processors for which the speedup is computed. The speedup of an algorithm is a good measure of the parallelism present in the algorithm. Another measure of the parallelism is the utilization of processors, which is the ratio of the time taken with 1 processor (the note in the speedup definition applies here too) to the product of \( n \) with the time taken with \( n \) processors, where \( n \) is the number of processors for which the utilization is computed. The utilization gives an idea about how efficiently the multiple processors were used by the algorithm. The execution time was measured in seconds by an utility program called TimeTest, provided in the Uniform System Library.

The test data sets were produced using a random data generating program, which uses a random number generator to generate the number of causal links that a manifestation has, and the disorders which cause the manifestation.

The programs were written in Butterfly C, and were executed on a 112-node Butterfly system. Several problems were encountered with the Butterfly system during the testing phase, due to the presence of system bugs. The testing of algorithm PBOTTOMUP was aborted at the initial stages when it was found that this algorithm is very inefficient, and does more work than necessary for finding the irredundant covers. A modified Bottom-up algorithm, which generates irredundant covers in a more efficient manner, was formulated, and its testing has been left for future research. Some of the problems encountered due to system faults could not be circumvented, and as a result, tests could be conducted only on small data cases. The results from the tests that were conducted, combined with the causes of the problems that were encountered, were used to study the algorithms, and hypothesize on their merits and limitations.

5. Performance Analysis of the Algorithms

In this section, results of two test cases are presented and analyzed – one for the algorithm PTOPDOWN, and the other for the algorithm PGENSET4.

The first sample test is on algorithm PTOPDOWN. The test data has 4 manifestations to be covered, and was generated with a maximum of 8 disorders, and each of the manifestations has 4\times3 causal links associated with it. The causal links that were generated, and the readings of execution time along with the speedup and utilization figures, are given in figure 1.

As seen in figure 1, the problem has very few irredundant covers, and hence, the work done by PTOPDOWN in finding these covers is also very small. As a result, the utilization of the processors decreases rapidly as the number of processors is increased. The speedup increases very well from 1 processor to 2 processors, but increases at a much slower pace, remains constant, then finally decreases, when more than 2 processors are used. This is as expected since if the problem is small, use of many processors only increases the overhead, and does not help in doing useful work.

The second sample test case presented is for the PGENSET4 algorithm. In this case, there are 5 manifestations to be covered in the diagnostic problem, and a maximum of 15 disorders were used to generate the test data. Each of the manifestations has 5\times3 causal links associated with it. The causal relationship between manifestations and disorders, as well as the execution time, speedup and utilization figures for this test case are presented in figure 2.

As can be seen in figure 2, the speedup achieved is not very good as the number of processors is increased. This shows that algorithm PGENSET4 does not have much parallelism, and hence is unable to exploit additional processors for executing parallel processes. This point will be further discussed later.

A total of 10 successful experiments were conducted on the four algorithms, and the algorithms were analyzed based on the results of these experiments and on the behaviour of the algorithms on other unsuccessful experiments.

From the tests, the algorithms PTOPDOWN and PBIDIRBFS were found to require a large amount of memory when compared to the amount required by algorithm PGENSET4. This was due to the explicit generation and storage of cover sets by these two algorithms. These algorithms also had large time requirements when compared to algorithm PGENSET4, which may be attributed to the expensive set operations like intersection, membership, etc., which do not occur very much in PGENSET4. However, the algorithm PGENSET4 was found to have a large sequential bottleneck due to the presence of several recursively defined generator-set operators, and thus resulted in very poor speedup figures. The algorithms PTOPDOWN and PBIDIRBFS, on the other
hand, showed good speedup figures when the number of processors were less than or equal to the maximum number of processors that could be utilized by the problem. Since the problem size in our experiments were small (due to system limitations and bugs), the increase in speedup could be seen only for small number of processors.

<table>
<thead>
<tr>
<th># proc.</th>
<th>Time(in secs)</th>
<th>Speedup</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.51</td>
<td>1.0</td>
<td>1.0000</td>
</tr>
<tr>
<td>2</td>
<td>1.59</td>
<td>1.6</td>
<td>0.7094</td>
</tr>
<tr>
<td>4</td>
<td>1.37</td>
<td>1.8</td>
<td>0.4580</td>
</tr>
<tr>
<td>8</td>
<td>1.02</td>
<td>2.4</td>
<td>0.3075</td>
</tr>
<tr>
<td>16</td>
<td>1.04</td>
<td>2.4</td>
<td>0.1508</td>
</tr>
<tr>
<td>32</td>
<td>1.06</td>
<td>2.3</td>
<td>0.0739</td>
</tr>
<tr>
<td>64</td>
<td>1.21</td>
<td>2.0</td>
<td>0.0323</td>
</tr>
</tbody>
</table>

Figure 1. Sample Test Case 1.

<table>
<thead>
<tr>
<th># proc.</th>
<th>Time(in secs)</th>
<th>Speedup</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.79</td>
<td>1.0</td>
<td>1.0000</td>
</tr>
<tr>
<td>2</td>
<td>1.93</td>
<td>1.4</td>
<td>0.7228</td>
</tr>
<tr>
<td>4</td>
<td>1.45</td>
<td>1.9</td>
<td>0.4810</td>
</tr>
<tr>
<td>8</td>
<td>1.27</td>
<td>2.2</td>
<td>0.2746</td>
</tr>
<tr>
<td>16</td>
<td>1.26</td>
<td>2.2</td>
<td>0.1384</td>
</tr>
<tr>
<td>32</td>
<td>1.44</td>
<td>1.9</td>
<td>0.0605</td>
</tr>
<tr>
<td>64</td>
<td>1.52</td>
<td>1.8</td>
<td>0.0287</td>
</tr>
</tbody>
</table>

Figure 2. Sample Test Case 2.

6. Conclusion

Only hypotheses could be formed regarding the workings of the algorithms, since tests on large data sets could not be conducted due to system problems. Even with the limited testing, many of the merits and limitations of the algorithms could be identified. These findings could be used to improve upon the algorithms to get better performance.

Acknowledgements

Supported by grants to the University of Maryland from Control Data Corporation and AT&T as matching funds for NSF Award DCR-8451430, and to the Center for Automation Research from DARPA Award DACA76-84-C-0004.

REFERENCES