Finding Test-and-Treatment Procedures Using Parallel Computation [*]

Louis D. Duval, Robert A. Wagner, Yijie Han
and Donald W. Loveland

Department of Computer Science, Duke University, Durham, NC 27706

Abstract

A parallel algorithm to generate optimum test-and-treatment decision trees is presented. Constructing such trees is NP-hard. The algorithm is designed for a machine whose number of connections is $3p/2$, where p is the number of processing elements(PEs), and where the PEs are simple enough such that a machine with 2^{20} PEs is currently implementable and a 2^{30} PE machine is feasible. A speedup of $O(p/(s \log p))$ over a sequential dynamic programming algorithm for this task is achieved, by paying careful attention to the communication problem, where s bits are used to represent costs. This algorithm is realized on the Boolean Vector Machine, a cube-connected-cycle system with 2^{20} PEs which is currently running in prototype form, with 512 PEs. The algorithm is concrete, in that all processor allocation and other control issues have been solved. The particular NP-hard problem is of independent interest; it generalizes the binary testing problem by introducing treatments on an equal basis with tests. Applications of this test-and-treatment problem are found in medical diagnosis, systematic biology, machine fault location, laboratory analysis and many other fields.

1 Introduction

1.1 The Test-and-Treatment Problem

The test-and-treatment(TT) problem originally defined by D. W. Loveland is a generalization of the binary testing problem studied by many researchers(see [1, 2, 7, 8, 10]). This problem is of independent interest since it finds applications in numerous real-world applications.

The test-and-treatment problem requests the selection of a "best" test and treatment procedure under a minimum expected cost criteria. The problem specification consists of a universe $U = \{0, 1, ..., k-1\}$ of k objects, each with an associated weight P_i, and a set of tests and treatments $\{T_i, 1 \leq i \leq N\}$, each with an associated cost. The $T_i, 1 \leq i \leq m$, denote tests, and the $T_i, m < i \leq N$, denote treatments. We assume that only one object is actually faulty, its identity is unknown, and each object i has a priori likelihood P_i of being the faulty object. Each test and treatment is specified by a subset of the universe; if the unknown object is in the test or treatment set then the test responds positively, or

[*]Work reported herein is partially supported by the Air Force under grant number AFOSR 81-0221 and AFOSR 83-0205.

"is successful", or the treatment is successful. If the test is successful, the objects not in the test set are eliminated from consideration (and if negative, the test set of objects is eliminated), while a successful treatment ends the procedure. A failed treatment means the processing must continue. A successful TT procedure must provide for each object to be treated; a TT problem specification is *adequate* if there exists a successful TT procedure. With each test and treatment T_i a cost t_i of executing that test or treatment is given with the problem specification.

From the above description we see that a TT procedure is a binary decision tree, with both test and treatment nodes. A typical TT procedure is given in Fig. 1, where a single-line arc is used for both test outcomes (the positive outcome to the left by convention) and a treatment failure, and a double-line arc denotes a treatment set. (The double-line arc is for emphasis only, since every branch of a successful TT procedure must terminate in a treatment set.)

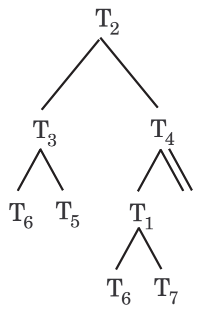

Fig. 1. $U = \{0, 1, 2, 3, 4, 5\}$, $P_i = 1/6$ for all i, $T_1 = \{2\}$, $T_2 = \{0, 4\}$, $T_3 = \{0, 1\}$, $T_4 = \{0, 1, 3\}$, $T_5 = \{4\}$, $T_6 = \{0, 2\}$, $T_7 = \{5\}$, $m = 3$ $t_i = 1$ for all i.

The TT procedure tree has an expected cost defined as: $Cost(Tree) = \sum_{i \in U}$ (cost of all tests and treatments encountered if i is the faulty object) $\cdot P_i$. The desired solution is the procedure which minimizes this cost. Thus $Cost = \min_{all\ trees} Cost(Tree)$.

It has been shown [4, 7] that finding an optimal solution to the binary testing problem is in general NP-hard. Since the test-and-treatment problem generalizes the binary testing problem, the test-and-treatment problem is also NP-hard. A parallel algorithm for this problem is presented which is implemented on the Boolean Vector Machine(BVM). We are able to achieve time complexity $O(ks(k + \log N))$ using $p = O(N2^k)$ processors, where k is the size of the universe of objects containing the malfunctioned one, s is the precision required, and N is the total number of tests and treatments available. This result represents a speedup of $O(p/\log p)$, with regard to the known sequential algorithm which could be obtained by modifying the backward induction algorithm given by Garey [1].

Parallel algorithms for NP-complete or NP-hard problems exist in the literature [5, 9]. There are also efforts in designing parallel approximation algorithms [12]. Our contribution is to show that the important TT problem can be solved on the BVM in a fairly efficient way. The efficiency is achieved through the resolution of processor allocation and processor communication problems. The results shown here also demonstrate that BVM is a powerful and cost-effective parallel machine [14].

1.2 The Boolean Vector Machine

The BVM is a CCC [11] parallel machine. Logically the BVM can be viewed as a bit array. Each row of bits forms a register. Each column forms a PE. The registers are denoted R[0], R[1], R[2],..., and are termed the PE's "memory". Each PE also contains 2 nonmemory registers A and B which act like 1-bit accumulators. Let r be a positive integer and $Q = 2^r$, there are a total of 2^{r+Q} PE's, as required by a complete CCC network. The address of a PE can be represented by (i, j) with the first component being the cycle number and the second the address within the cycle. Within cycle i, PE (i, j) is connected only to its predecessor $(i, (j + Q - 1)\%Q)$[1] and its successor $(i, (j + 1)\%Q)$. In addition each PE (i, j) is connected to its lateral neighbor $(i \wedge 2^j, j)$, thus connecting the cycles together.

The BVM is a bit-oriented machine. Only Boolean function operations are allowed. Each of its instructions involves possibly register A and B and at most one other register. Its instruction has the form:

$\{A \text{ or } R[j]\},\ B = f, g(F, D, B)\{\text{IF or NF}\} < \text{set} >;$

Two assignment operations will be performed simultaneously by executing this instruction. The first assigns $f(F, D, B)$ to either A or $R[j]$; the second assigns $g(F, D, B)$ to B. f and g are any Boolean functions of three arguments. F may be A or $R[j]$. D may be $A.N$ or $R[j].N$. N denotes a neighbor PE of PE (c, p). It can be:

S: successor PE $(c, (p + 1)\%Q)$;

P: predecessor PE $(c, (p + Q - 1)\%Q)$;

L: lateral PE $(c \wedge 2^p, p)$;

XS: even successor exchange PE $(c, p \wedge 2^0)$;

XP: even predecessor exchange XP=P if p is even; XP=S if p is odd;

I: input one bit to PE (0, 0), PE $(2^Q - 1, Q - 1)$ outputs one bit at the same time. All other PEs get bits from their predecessors except PEs (., 0), which get bits from PEs $(. - 1, Q - 1)$.

The {IF or NF} < set > denotes the activate/deactivate set. < set > is a subset of $\{0, 1, ..., 2^{r-1}\}$. IF < set > means all the PE's (i, j), $0 \leq i < 2^Q$ and $j \in$ < set >, will be activated while the remaining PE's will be deactivated. The meaning of NF < set > is just the opposite. If the part {IF or NF} < set > is not present in the instruction, then all the PE's are activated. There is also a special memory register, E, which is used as an enable/disable register. PE i will be enabled or disabled according to whether its bit of the E register is 1 or 0. The E register itself is always enabled. The value of a PE's memory(except for register E) will not be changed while that PE is disabled. The entire state of a PE will remain unchanged by an instruction which deactivates that PE.

For further details of the BVM, the reader is referred to [13, 14].

[1]As in the C language [6], %, /, &, |, $\wedge$ are the modulo, integer division, and, or, and exclusive-or operations, respectively.

1.3 ASCEND/DESCEND Algorithms

An algorithm is in the ASCEND(DESCEND) [11] form if it consists of a sequence of basic operations on pairs of data, where the addresses of the pairs differ successively in bit 0, bit 1, ..., bit $p-1$ (bit $p-1$, bit $p-2$, ..., bit 0), here and henceforth bits are counted from the least significant bit.

Preparata and Vuillemin showed in [11] that the ASCEND/DESCEND algorithms can be simulated on a CCC at a slowdown of a factor of 4 to 6, regardless of the network sizes. Thus designing an ASCEND/DESCEND algorithm for a hypercube, and transforming it into a CCC algorithm seems to be a reasonable way of designing an efficient CCC algorithm. The algorithm presented in this paper is in the ASCEND/DESCEND form.

2 A Parallel BVM Algorithm for the Test-and-Treatment Problem

Rather than enumerate all the possible TT trees and take the minimum cost directly as in section 1.1, we use the approach of dynamic programming and note that the optimal tree must apply the minimum cost action (test or treatment) to already optimal subtrees. The optimal subtrees are obtained by beginning with the empty tree and combining trees as just described. Thus we start with $C(\phi) = 0$ and $C(S) = \infty$ for $S \neq \phi$. For an arbitrary nonempty set S of objects we compute the cost $C(S)$ as

$$\begin{aligned} C(S) = \min[&\min_{1\le i\le m}(t_i * p(S) + C(S\cap T_i) + C(S - T_i)),\\ &\min_{m<i\le N}(t_i * p(S) + C(S - T_i))]. \end{aligned}$$

where $p(S) = \sum_{j\in S} p_j$. This definition is from first principles: the value t_i is charged to each object subject to that action and the total weight of those objects to be charged is $p(S)$. For tests, one adds in the cost $C(S\cap T_i)$ of the set $S\cap T_i$ to which the test responds positively (the test set) plus the cost $C(S-T_i)$ of the set $S-T_i$ of objects not responding to the test. Treatments terminate action on the objects of $T_i, m < i \le n$, (i.e., treat them) so the only objects needing further action are the objects in $S - T_i$; we add in the cost $C(S-T_i)$. The essence of an argument by induction that $C(S)$ is correctly computed uses the assumption that $C(S\cap T_i)$ and $C(S-T_i)$ are the correct costs for the subtrees and then we note from the above description that the correct minimum is taken to compute $C(S)$. We see that $C(U) = Cost(tree)$ as desired.

In actual computation we will assign an array $M[S,k]$ to calculate $C(S)$: $M[S,i] = t_i p(S) + C(S\cap T_i) + C(S-T_i)$, $0 \le i \le m$, and $M[S,i] = t_i p(S) + C(S-T_i)$, $m < i < N$, therefore

$$C(S) = \min\{M[S,i] | 0 \le i < N\}.$$

Note that in a sequential algorithm, the subscript i would be represented by a time sequence of values held in a scalar variable. The sequential dynamic programming algorithm implied by this equation thus uses $O(2^k)$ space, and runs in time $O(kN2^k)$. The factor of k appearing in the time complexity is incurred because the algorithm requires operations on sets of size as large as k during the calculation of some $M[S,i]$ entries.

The above observation is expressed in the following algorithm.

```
TT( )
{
    foreach $i$: $0 \leq i < N$ do {
        $TP[S, i] = t_i * p(S)$, if $|S| > 0$,
        $M[S, i] = \left[ \begin{array}{ll} 0, & \text{if } |S| = 0, \\ \infty, & otherwise. \end{array} \right.$
    }
    for ($j = 1$; $j \leq |U|$; $j++$) {
        foreach $(S, i)$: $U \supseteq S$ and $|S| = j$ and $0 \leq i < N$ do {
            $M[S, i] = M[S - T_i, i]$;
            $M[S, i] += TP[S, i]$;
            if ($i \leq m$) $M[S, i] += M[S \cap T_i, i]$;
        }
        foreach $(S, i)$: $U \supseteq S$ and $|S| = j$ and $0 \leq i < N$ do {
            $M[S, i] = \min(M[S, x] | 0 \leq x < N)$;
        }
    }
}
```

$|S|$ in the algorithm denotes the size of set S.

The algorithm above is not in ASCEND or DESCEND form. Indeed, it may not even be an EREW (Exclusive-Read Exclusive-Write Parallel RAM) algorithm. It therefore cannot be simulated directly by the BVM, which uses a CCC network. However the algorithm can be placed into one of these standard forms by a series of transformations, each of which produces a computationally equivalent algorithm. The transformations first ensure that variables whose value is required in several instances of one parallel expression ("broadcast" variables) are sent to the nodes of separate binary trees, with each node of the tree for variable v representing one of the destination expressions for v. Computations in which several variables are "gathered" into a single expression are treated in symmetric fashion. After these transformations are performed, the data movements are reorganized, to conform to the restriction that the neighbor links of the hypercube be used sequentially.

In the notation used above, set characteristic numbers are used as array subscripts. This notation is compact, but it conceals certain broadcasts, and may conceal some parallel assignment operations in which some variable is assigned a value more than once. These situations must be detected, and rewritten, to eliminate multiple assignments to any variable, and to make explicit the set of expressions which are the destinations for each variable referenced by the assignment statement. Multiple parallel assignments to one variable are resolved here by using additional subscripts, in effect splitting each such target variable into enough subtargets that every instance of the parallel assignment statement has a unique subtarget. The values assigned to the subtargets are then combined in an explicit loop, to complete the assignment.

Observe that if we assign a PE to each (S, i) pair with $M[S, i]$ and $TP[S, i]$ placed in different portion of that PE's memory, the instruction $M[S, i] += TP[S, i]$ can be executed in parallel by all PEs at once. (The necessary bit-serial addition algorithm is a subroutine, executed in parallel on all PEs.) The minimization part of the algorithm can be transformed into the following ASCEND form:

```
**for** $(t = 0;\ t < \log N;\ t++)$
    **foreach** $(S, i)$: $U \supseteq S$ and $|S| = j$ and $0 \leq i < N$ **do**
        $M[S, i] = \min(M[S, i], M[S, i\#t])$;
```

where $i\#t$ is the binary number obtained by complementing the t-th bit (from the right) of i.

Now consider the instruction:

foreach (S, i): $M[S, i] = M[S - T_i, i]$.

This operation represents an implicit broadcast of information, in which the source of each datum, and the set of array elements each must be sent to, are both obscure. We begin by expanding the operation into its components as follows:

```
**foreach** $(S, i)$ **do** {
    $R[S, i] = M[S - T_i, i]$;
    $M[S, i] = R[S, i]$;
}
```

Consider now the operation

$R[S, i] = M[S - T_i, i]$.

For each S and i, this operation is well defined, i.e., each PE which receives information during this activity receives information from only one PE. However, one PE must send information to several PEs. In general, $M[S - T_i, i]$ must be broadcast to $R[(S - T_i) \cup V, i]$, for each V such that $S \cap T_i \supseteq V$. The following loop accomplishes the required broadcast, for all i, $0 \leq i < N$:

```
$R[S, i] = M[S, i]$;
**for** $(e = 0;\ e < k;\ e++)$
    **foreach** $(S, i)$: $U \supseteq S$ and $0 \leq i < N$ and $e \in S \cap T_i$ **do**
        $R[S, i] = R[S - \{e\}, i]$;
$M[S, i] = R[S, i]$;
```

Let $I_t = \{j \in U | j \leq t\}$. Using induction one can show that just before e takes on value t, $R[(S - T_i) \cup (S \cap T_i \cap I_{t-1}), i]$ holds $M[S - T_i, i]$. After all iterations of the loop on e, $I_t = U$, and $S \cap T_i \cap I_t = S \cap T_i$. Thus, for all S and i, $R[S, i] = M[S - T_i, i]$.

Similarly, the operation

if $(i \leq m)$ $M[S, i] += M[S \cap T_i, i]$

can be transformed into:

```
$Q[S, i] = M[S, i]$;
**for** $(e = 0;\ e < k;\ e++)$
```

```
    **foreach** $(S, i)$: $U \supseteq S$ and $0 \leq i < N$ and $e \in S - T_i$ **do**
        $Q[S, i] = Q[S - \{e\}, i]$;
**if** $(i \leq m)$ $M[S, i] += Q[S, i]$;
```

The complete algorithm now appears as

```
TT()
{
    **foreach** $i$: $0 \leq i < N$ **do** {
        **if** $(|S| > 0)$ $TP[S, i] = t_i * p(S)$;
        $M[\phi, i] = 0$;
        **if** $(|S| > 0)$ $M[S, i] = \infty$;
    }
    **for** $(j = 1;\ j \leq |U|;\ j++)$ {
        **foreach** $(S, i)$: $P_1(S, i)$ {
            $Q[S, i] = R[S, i] = M[S, i]$;
        }
        **for** $(e = 0;\ e < |U|;\ e++)$ {
            **foreach** $(S, i)$: $P_1(S, i)$ and $e \in S \cap T_i$ {
                $R[S, i] = R[S - \{e\}, i]$;
            }
            **foreach** $(S, i)$: $P_1(S, i)$ and $e \in S - T_i$ {
                $Q[S, i] = Q[S - \{e\}, i]$;
            }
        }
        **foreach** $(S, i)$: $P(S, i, j)$ {
            $M[S, i] = R[S, i]$;
            $M[S, i] += TP[S, i]$;
            **if** $(i \leq m)$ $M[S, i] += Q[S, i]$;
        }
        **for** $(t = 0;\ t < \log N;\ t++)$
            **foreach** $(S, i)$: $P(S, i, j)$ {
                $M[S, i] = \min(M[S, i], M[S, i\#t])$;
        }
    }
}
```

Where $P_1(S, i) \equiv U \supseteq S$ and $0 \leq i < N$, and $P(S, i.j) \equiv P_1(S, i)$ and $|S| = j$.

On the BVM each PE will stand for a pair (i, j), where i and j are binary numbers and ij, the concatenation of i and j, is the address of the PE. $|i|$, the number of bits in i, is k. The component i denotes a subset S of U, $a \in S$ if and only if the ath bit of i is 1. j is the index of a test or a treatment.

The predicates $e \in S \cap T_i$ and $e \in S - T_i$ can be implemented by using the processor-ID vector. The Processor-ID vector is a bit pattern in which the memory of processor (i, j) contains ij. This vector can be computed in $O(\log p)$ time [13]. The processor-ID bits will let each PE know which set S it represents. The sets T_i are inputs, and are assumed here to be placed into the memory of each PE which needs them initially. The most interesting part of the algorithm is the loop indexed by the variable e. Note that by imposing the conditions $e \in S \cap T_i$ (respectively, $e \in S - T_i$) the result becomes $R[S, i] = R[S - T_i, i]$ (respectively, $Q[S, i] = Q[S \cap T_i, i]$).

The TT algorithm presented here is an ASCEND/DESCEND hypercube algorithm. Using Preparata and Vuillemin's technique [11], we can pipeline the computation through the cycles of BVM resulting in an efficient BVM algorithm. The further details of this BVM algorithm are shown in [3].

Using $N2^k$ processors, one for each pair (S, i), the time complexity of algorithm TT() is $O(k(k + \log N))$. The main part of the algorithm is a two-level nested loop. The outside loop is indexed by j and has k iterations. Inside this loop are two sequential loops, indexed by e and t, respectively, which have k and $\log N$ iterations, respectively. Thus the time complexity is $O(k(k + \log N))$. When this hypercube algorithm is transformed into a BVM algorithm it will be slowed down by at most a constant factor, incurred in pipelining the computation through the cycles of the BVM. When the time complexity is measured based on the number of bit operations performed, the time complexity becomes $O(sk(k + \log N))$, where s is the number of bits used to represent a number in arrays M and R.

3 Conclusion

Many NP-complete problems can be solved on the BVM fairly efficiently, as we illustrate using the test-and-treatment problem. Indeed, the test-and-treatment problem itself is of real interest as it has many important applications. A parallel algorithm for this problem is presented and implemented on the Boolean Vector Machine. The communication and PE allocation problems have been solved so that a speedup of $O(p/(s \log p))$ is achieved. The factor of s would vanish if the sequential algorithm were also implemented on a bit-serial processor.

References

1. Garey, M. R. Optimal binary identification procedures. *SIAM J. Appl. Math.* **23**, 2, (Sept. 1972), 173-186.

2. Garey, M. R. and Graham, R. L. Performance bounds on the splitting algorithm for binary testing. *Acta Inform.* **3** (1974), 347-355.

3. Han, Yijie. Algorithms on an n-PE Bit-Serial CCC Network. Master's Thesis. Dept. Computer Science, Duke Univ., May 1984.

4. Hyafil, L. and Rivest, R. L. Constructing optimal binary decision trees is NP-complete. *Inf. Process. Lett.* **5** (1976), 15-17.

5. Karnin, E. D. A parallel algorithm for the knapsack problem. *IEEE Tran. Comput.* **C-33**, 5 (May 1984), 404-408.

6. Kernighan, B. W. and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

7. Loveland, D. W. Selecting optimal test procedures from incomplete test sets. *Proc. First Int. Symp. Policy Anal. Inf. Sci.*, Duke University, Durham, NC, 1979, pp. 228-235.

8. Loveland, D. W. Performance bounds for binary testing with arbitrary weights. *Acta Inform.* **22** (1985), 101-114.

9. Mead, C. and Conway L. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980, pp. 305-313.

10. Payne, R. W. and Preece, D. A. Identification keys and diagnostic tables: A review. *J. Roy. Statist. Soc. Ser. A* **143**, 3 (1980), 253-292.

11. Preparata, F. P. and Vuillemin, J. The cube-connected cycles: A versatile network for parallel computation. *Comm. ACM* **24**, 5 (May 1981), 300-309.

12. Wah, B. W., Li, G., Yu, C. F. Multiprocessing of combinatorial search problems. *Computer* **18**, 6 (June 1985), 93-108.

13. Wagner, R. A. A programmer's view of the Boolean Vector Machine, Model-2. CS-1981-8, Dept. of Computer Science, Duke Univ., Durham, NC, Oct. 1981.

14. Wagner, R. A. The Boolean Vector Machine [BVM]. *IEEE 1983 Conf. Proc. of 10th Ann. International Symposium on Computer Architecture*, pp. 59-66.