Network Implementations of the DTEP Algorithm*

Ernst W. Mayr C. Greg Plaxton†

May 15, 1987

Abstract

The dynamic tree expression problem (DTEP) was defined in [Ma87]. In this paper, efficient implementations of the DTEP algorithm are developed for the hypercube, butterfly, perfect shuffle and multi-dimensional mesh of trees families of networks.

*This work was supported in part by a grant from the AT&T Foundation, ONR contract N00014-85-C-0731, and NSF grant DCR-8351757.

†Primarily supported by a 1967 Science and Engineering Scholarship from the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

The dynamic tree expression problem (DTEP) was introduced by Mayr [Ma87] and is based upon previous work by Ruzzo [Ru80], Miller & Reif [MR85] and Ullman & Van Gelder [UV85]. This paper develops efficient implementations of the DTEP algorithm for the hypercube, butterfly, perfect shuffle and multi-dimensional mesh of trees families of networks.

In Section 2 we give the formal definition of DTEP and an algorithm for solving it, which will be referred to as the DTEP algorithm. Section 3 provides the details of the computational model of the networks we are considering, along with implementations for two useful primitive operations. In Section 4 these primitives are used to produce implementations of the DTEP algorithm and an analysis of their time and communication requirements is performed. We will be primarily concerned with implementations for single instruction stream, multiple data stream (SIMD) parallel computers, a well-known class first defined by Flynn [Fl66].

There is a list of symbols in the appendix which should serve to clarify the programming notation.

2 The DTEP Algorithm

A DTEP instance is a triple (P, I, Z) where

1. P is a set of n Boolean variables $p_0, \ldots, p_{n-1}$,
2. I is a set of inference rules of the form $p_i \leftarrow p_j$ or $p_i \leftarrow p_j p_k$, and
3. $Z \subseteq P$.

The task is to compute the minimal model for (P, I, Z), ie. the minimal $\mathcal{M} \subseteq P$ satisfying

1. $Z \subseteq \mathcal{M}$,
2. $(p_j \in \mathcal{M}) \wedge (p_i \leftarrow p_j \in I) \Rightarrow p_i \in \mathcal{M}$, and
3. $(p_j, p_k \in \mathcal{M}) \wedge (p_i \leftarrow p_j p_k \in I) \Rightarrow p_i \in \mathcal{M}$.

The Boolean variables belonging to Z may be thought of as axioms. The inference rules are applied to these axioms to prove as many of the remaining variables **true** as possible. A *derivation tree* for $p \in P$ is a labelled binary tree with node labels taken from P such that: (i) labels of the leaves belong to Z; (ii) if an internal node has label p_i and a single child labelled p_j then $p_i \leftarrow p_j \in I$; (iii) if an internal node has label p_i and children labelled p_j, p_k then $p_i \leftarrow p_j p_k \in I$; (iv) the root is labelled p. The *size* of a derivation tree T is the number of nodes that it contains and is written $|T|$.

Clearly, $p \in \mathcal{M}$ if and only if there is a derivation tree for p. The following algorithm makes use of this fact to construct the minimal model $\mathcal{M}$ for a given DTEP instance (P, I, Z). We will consider a parallel implementation using n^3 processors, each identified by a unique triple (i, j, k), $0 \le i, j, k < n$. There are $n + n^2 + n^3$ variables, which are initialized as follows

1. $P_i \equiv (p_i \in Z)$, $0 \le i < n$,
2. $P_{ij} \equiv (i = j) \vee (p_i \leftarrow p_j \in I)$, $0 \le i, j < n$, and
3. $P_{ijk} \equiv (p_i \leftarrow p_j p_k \in I) \vee (p_i \leftarrow p_k p_j \in I)$, $0 \le i, j, k < n$.

procedure DTEP

(1) **loop**

(2) $P_i \xleftarrow{\vee} (P_j \wedge P_k \wedge P_{ijk}) \vee (P_j \wedge P_{ij})$

(3) $P_{ij} \xleftarrow{\vee} P_k \wedge P_{ijk}$

(4) **exit when** no new P_i has been derived

(5) $[P_{ij}] \longleftarrow [P_{ij}]^2$

(6) **end loop**

end DTEP

Line 5 of DTEP computes the square of the Boolean matrix $[P_{ij}]$. The problem of implementing general matrix multiplication on the hypercube and perfect shuffle was studied extensively by Dekel, Nassimi and Sahni [DNS81]. For the special case of Boolean matrix multiplication, Agerwala and Lint have given a parallel implementation of the four Russians' algorithm which runs in $O(\log n)$ time using $n^3/(\log n \log\log n)$ processors [AL78].

Since the diagonal entries of the matrix $[P_{ij}]$ are always **true**, any variable which becomes **true** at any time during the course of the computation will remain so. Using the method of Miller & Reif it can be proven that if $p_i \in \mathcal{M}$ has a derivation tree T in (P, I, Z) then P_i becomes **true** within at most $(\log_{4/3} 2) \log |T|$ iterations of the loop [MR85]. The correctness of the terminating condition used above is easy to establish using a proof by contradiction.

The parallel running time for a single iteration depends upon the model of computation. On a CRCW PRAM, each iteration can be performed in $O(1)$ time even when concurrent writes must agree; the trick is to write only **true** values. On a CREW PRAM, each iteration can be implemented to run in $O(\log n)$ time by using a tree computation whenever it is necessary to OR together many Boolean values. The EREW PRAM remains powerful enough to achieve $O(\log n)$ but the constant of proportionality is higher than for CREW since there are instances where a tree computation must be used to make copies of a value needed by many processors.

Thus, DTEP is guaranteed to run in $\mathcal{NC}$ whenever each $p_i \in \mathcal{M}$ has a derivation tree of "expolylog" size, that is, bounded by $2^{\log^c n}$ for some constant $c \ge 0$. An important consequence of this is that any problem which can be transformed, within $\mathcal{NC}$, to a derivation system with expolylog bounded derivation trees is itself in $\mathcal{NC}$. The planar monotone circuit value problem [Go80], known to be in $\mathcal{NC}$, is an example of a problem which admits such a transformation.

3 Network Primitives: Replicate and Collect

Our first goal is to develop, in a systematic manner, "efficient" implementations of the DTEP algorithm for several well-known networks. Assume without loss of generality that the given DTEP instance (P, I, Z) has $|P|$ equal to a power of two and define n, m, N and M by the equations

$$n = |P|, \ m = \log n, \ N = n^3, \ M = \log N.$$

We adopt the usual abstract view of a network as a graph in which nodes represent processors and edges represent bidirectional communication links. Each processor has a local memory with words of length $O(\log n)$ and we make the uniform cost model assumption that the standard set of ALU operations can be performed in constant time on operands of this size. Every processor is assigned a unique $O(\log n)$ bit number which will be referred to as its *id*.

We will initially restrict our attention to SIMD parallel computers. One way to understand the SIMD model is to imagine many processors synchronously executing duplicate copies of a program with no conditional branch instructions and in which every statement S_i is accompanied by a Boolean condition C_i. The statements of the program operate on local variables and data received through messages from adjacent processors. Each processor has the same set of local variables as any other, but they may have different initial values. There is no global memory. We will not be concerned with the question of how the network communicates with the outside world; the input (output) is simply given by the initial (final) values of some subset of the variables.

Program execution takes place in the following manner. When all of the processors arrive simultaneously (as must be the case) at some statement S_i, they first evaluate C_i. Those for which C_i is **true** are said to be *enabled* and proceed to execute S_i. The remaining processors are *disabled* for the period of time that it takes to execute S_i. This process is then repeated at the statement following S_i. In our programs, the condition C_i will be a function of the processor id which can be computed in constant time. For example, if the processor id is z and C_i is given by the expression $z_5 = 1 \wedge z_{[0,3)} = z_{[6,9)}$ it could be evaluated in constant time by the "machine" expression

$$(z \text{ AND } m_1) \neq 0 \wedge (z \text{ AND } m_2) = ((z \text{ DIV } m_3) \text{ AND } m_2)$$

where $m_1 = 100000_2$, $m_2 = 111_2$ and $m_3 = 1000000_2$ are masks obtained in constant time by indexing into tables which can be precomputed in $O(\log n)$ time.[1]

Algorithmic complexity will be measured in terms of communication overhead as well as time. We will consider the execution of a program to consist of a sequence of *steps*. Each step is allowed only $O(1)$ time and is made up of a *computation phase* and a *communication phase*. During the computation phase no messages are passed between processors. During the communication phase each processor can send (and/or receive) an $O(\log n)$ length

[1]Table lookup is not necessary if we are given an instruction capable of shifting a register $O(\log n)$ bit positions in constant time (eg. MUL); under this assumption the masks can be computed on the fly.

message to (from) each of its neighbors. Define the *communication cost* of an algorithm to be the total number of messages which it uses. We will sometimes refer to the total number of steps used by an algorithm as its *step count*. In this paper, "exact" step counts should be interpreted as being accurate to within an additive constant, eg. $5\log n$ means $5\log n + O(1)$. Note that a step count of $f(n)$ implies a running time which is $O(f(n))$.

For each network family we will describe two implementations of the DTEP inner loop which attempt to minimize: (i) step count; (ii) communication cost under the constraint that the step count lie within a constant factor of optimal, ie. it must be $O(\log n)$. For a synchronous, fixed interconnection network there is little motivation for minimizing communication cost since a communication phase will use the same amount of time regardless of how many links are actually used to send messages. However, the amount of message traffic may be of critical importance in a time-shared asynchronous environment or when the network for which the algorithm has been designed is being simulated on another type of network. We will also indicate how much improvement in the running time can be obtained by modifying our implementations slightly to take advantage of a multiple instruction stream, multiple data stream (MIMD) environment.

The motivation for counting steps is to allow the constant multiplicative factors on the leading term of the running time of two programs to be compared with reasonable accuracy *without* resorting to the tedious approach of counting up CPU cycles. If it is true that the running times of individual steps tend to be clustered around a single value then this approximation will be a useful one. Unfortunately, our definition of a step allows k independent calculations to be "interleaved" in such a way that the step count goes down by a factor of k while the actual running time remains more or less unchanged. This is accomplished by passing *all* local data which is relevant to any of the k calculations to all neighbors which require any data and merging the computation phases. In order to preserve the desired correlation between running time and step count, we do not allow interleaving in our minimum step count implementations.

The observations made in Section 2 regarding EREW PRAMs indicate that any network which admits an efficient implementation of the DTEP algorithm must be able to rapidly: (i) distribute copies of a particular value to many processors, and (ii) OR together many values stored at different processors. This motivates the definition of two primitive operations which we refer to as Replicate and Collect. The Replicate primitive takes four arguments: a pointer p, non-negative integers *start* and *width* satisfying $start + width \leq M$ and an integer *select* which should be thought of as a *width*-bit mask. The effect of the operation may be written

$$*p \textbf{ at } z \Longleftarrow *p \textbf{ at } z_{[start+width,M)} \circ select_{[0,width)} \circ z_{[0,start)}$$

where z denotes the processor id. For example, if p points to some variable x, $M = 6$, $start = 3$, $width = 2$ and $select = 01_2$ then x at processor $(z_5{*}{*}z_2z_1z_0)_2$ would be assigned the value of x at processor $(z_5 01 z_2z_1z_0)_2$. An important observation to make at this point is that by passing a field of the processor id to *select* rather than a constant, it is possible to perform transposition. There are several examples of this usage of Replicate in Section 4. The Collect primitive requires only the first three of the above parameters and performs

Network	*Processors*	*Degree*	*Diameter*	*High Flux*	*Layout Area*
hypercube	$N = n^k = 2^{km}$	$\log N$	$\log N$	yes	$\Theta(N^2)$
butterfly	$N \log N$	4	$\log N$	yes	$\Theta(N^2)$
perfect shuffle	N	3	$2 \log N$	yes	$\Theta(N^2/\log^2 n)$
kD mesh of trees	$(k+1)N - kn^{k-1}$	$2, 3, k$	$2 \log N$	no	$\Theta(N^{2-2/k}), k > 2$

Table 1: Some important network properties.

the operation

$$*p \textbf{ at } z \Longleftarrow \begin{cases} \bigvee_{0 \leq i < 2^{width}} *p \textbf{ at } z_{[start+width,M)} \circ i_{[0,width)} \circ z_{[0,start)} & \text{if } z_{[start,start+width)} = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Usually a call of the form $\mathsf{Collect}(p,\ start,\ width)$ will be followed by $\mathsf{Replicate}(p,\ start,\ width,\ 0)$ to obtain the combined effect

$$*p \textbf{ at } z \Longleftarrow \bigvee_{0 \leq i < 2^{width}} *p \textbf{ at } z_{[start+width,M)} \circ i_{[0,width)} \circ z_{[0,start)}. \tag{1}$$

We now consider the problem of implementing these two primitive operations on the hypercube, butterfly, perfect shuffle and multi-dimensional mesh of trees networks.

3.1 Hypercube

A degree d hypercube has 2^d processors with ids ranging from 0 to $2^d - 1$. Processor i is adjacent to processor j if and only if the binary representations of i and j differ in a single bit position. Some important properties of the hypercube as well as the other networks which we will study are given in Table 1. The hypercube has high flux[2] but unbounded degree. Our programs contain **if** statements but could easily be cast into the format described in the previous section.

The hypercube implementations of Replicate and Collect are given below. Both use *width* steps. Note that Replicate would perform exactly the same function with the condition in line 2 simplified to $z_{start+i} = select_i$, but this would increase the communication cost from $O(N)$ to $O(N \log N)$. The exact communication cost of both Replicate as well as Collect is $N - 2^{M-width}$ messages, which is approximately N for any non-trivial value of *width*. With regard to Collect, it is possible to achieve the effect of equation 1 directly by removing lines 7 and 9. This saves *width* time by eliminating the need for a call to Replicate, but increases the communication cost to $O(N \log N)$. We will take advantage of this trade-off in Section 4.1 in order to minimize the step count of our DTEP implementation.

It is interesting to note that, unlike the other networks listed in Table 1, the hypercube could handle replicating over a non-contiguous set of address bits just as easily; however, this feature is not needed for implementing DTEP.

[2] Roughly speaking, a network has high flux if it can sort rapidly. For a formal definition, see [GMU87].

procedure Replicate(p, $start$, $width$, $select$)
  for $i \longleftarrow 0$ **to** $width - 1$
    if $z_{[start+i, start+width)} = select_{[i,width)}$ **then**
      $*p^{(start+i)} \Longleftarrow *p$
    end if
  end for
end Replicate

procedure Collect(p, $start$, $width$)
  for $i \longleftarrow 0$ **to** $width - 1$
    if $z_{[start, start+i]} = 0$ **then**
      $*p \overset{\vee}{\Longleftarrow} *p^{(start+i)}$
    end if
  end for
end Collect

3.2 Butterfly

The "standard" butterfly network of degree d has $(d+1)2^d$ processors arranged in $d+1$ rows of 2^d. Each of these rows is called a *rank*, and the ranks are numbered consecutively from 0 to d. The processor adjacencies are defined as follows: processor z of rank r is connected to processors z and $z \oplus 2^r$ of rank $r+1$ for all r, z such that $r \in [0, d)$ and $z \in [0, 2^d)$.[3] This means that processors in ranks 0 and d have degree two while the remaining processors have degree four. There is an obvious variation of the standard butterfly network in which the dth rank is eliminated by mapping its processors onto those of rank 0. We will adopt this variation as our definition of a butterfly network, which explains the butterfly entries in Table 1 for number of processors and node degree.

It should be apparent that the hypercube is nothing more than a butterfly in which all of the ranks have been identified; alternatively, the butterfly may be viewed as an expanded version of the hypercube. As such, the butterfly can perform replication and collection just as fast as the hypercube as long as: (i) the address bits in question form a contiguous interval, and (ii) the data is initially located in a rank corresponding to one of the two endpoints of this interval. The first condition is always satisfied for us since Replicate and Collect have been defined to operate over the interval $[start, start + width)$. If the second condition is not satisfied then the butterfly loses ground to the hypercube because it must perform an Adjust to copy the data to one of the two appropriate ranks. The rank chosen will depend upon which rank(s) currently hold the data and where the results will be needed by subsequent calculations.

The Replicate and Collect procedures written below assume that the data resides in rank 0 and also put the result in rank 0. This implementation is sound but obviously wasteful; in Section 4.2 we will see that it is possible to do without most of the calls to Adjust which are implied by a naive translation of the hypercube implementation of the

[3]Note that our convention for numbering the ranks is the opposite of that chosen in [Ul84].

DTEP algorithm. The complexity of Adjust is *shift* steps and N*shift* messages, where we refer to the value of *shift* after line 13 has been executed. The communication cost could be decreased in those cases where not all values need to be preserved (eg. preceding a call to Replicate). This potential optimization has been omitted since we were unable to take advantage of it in our DTEP implementation. The routines Replicate′, Replicate″, Collect′ and Collect″ all execute in *width* steps using $O(N)$ messages.

procedure Replicate(p, *start*, *width*, *select*)
(1) Adjust(p, 0, *start*)
(2) Replicate′(p, *start*, *width*, *select*)
(3) Adjust(p, $start + width$, $M - start - width$)
or
(4) Adjust(p, 0, $start + width$)
(5) Replicate″(p, $start + width$, *width*, *select*)
(6) Adjust(p, *start*, $M - start$)
end Replicate

procedure Collect(p, *start*, *width*)
(7) Adjust(p, 0, *start*)
(8) Collect′(p, *start*, *width*)
(9) Adjust(p, $start + width$, $M - start - width$)
or
(10) Adjust(p, 0, $start + width$)
(11) Collect″(p, $start + width$, *width*)
(12) Adjust(p, *start*, $M - start$)
end Collect

procedure Adjust(p, r, *shift*)
(13) $\sigma, shift \longleftarrow (M > 2shift)\ ?\ +1, shift : -1, M - shift$
(14) **for** $i \longleftarrow 0$ **to** $shift - 1$
(15) $r + \sigma i$: $*p_\sigma \Longleftarrow *p$
(16) **end for**
end Adjust

procedure Replicate′(p, *start*, *width*, *select*)
(17) **for** $r \longleftarrow start$ **to** $start + width - 1$
(18) **if** $z_{[r,start+width)} = select_{[r-start,width)}$ **then**
(19) r: $*p_{+1}, *p'_{+1} \Longleftarrow *p$
(20) **end if**
(21) **end for**
end Replicate′

procedure Replicate″(p, *start*, *width*, *select*)
(22) **for** $r \longleftarrow start + width - 1$ **downto** *start*

(23) **if** $z_{[start,r]} = select_{[0,r-start]}$ **then**
(24) r: $*p, *p' \Longleftarrow *p_{+1}$
(25) **end if**
(26) **end for**
end Replicate″

procedure Collect′(p, $start$, $width$)
(27) **for** $r \longleftarrow start$ **to** $start + width - 1$
(28) **if** $z_{[start,r]} = 0$ **then**
(29) r: $*p_{+1} \Longleftarrow *p \vee *p'$
(30) **end if**
(31) **end for**
end Collect′

procedure Collect″(p, $start$, $width$)
(32) **for** $r \longleftarrow start + width - 1$ **downto** $start$
(33) **if** $z_{[r,start+width)} = 0$ **then**
(34) r: $*p \Longleftarrow *p_{+1} \vee *p'_{+1}$
(35) **end if**
(36) **end for**
end Collect″

3.3 Perfect Shuffle

Like the butterfly, the perfect shuffle is a high flux network with bounded degree. It was first introduced by Stone [St71]. A base b, degree d perfect shuffle has b^d processors with ids $[0, b^d)$. Each processor is linked to three others via the exchange, shuffle and unshuffle connections which allow processor i to communicate with processors $(i \bmod b = b - 1)$? $i-b+1 : i+1$, $\hookleftarrow i$ and $\hookrightarrow i$, respectively. From this point on we will be concerned only with the case $b = 2$, so processor i communicates via the exchange connection with processor $i \oplus 1$. One may view the perfect shuffle as a stripped-down version of the hypercube with only those edges corresponding to bit 0 adjacencies remaining (the exchange connection) and augmented by some connections (shuffle, unshuffle) which have the effect of cycling the mapping of variables to processors in such a way that a bit i dependency can be transformed into a bit 0 dependency.

In order to perform a Replicate or Collect the appropriate range of bits has to be cycled through the low order position so that exchange operations can be used. The complexity of Cycle is $shift$ steps and $N shift$ messages, where we refer to the value of $shift$ after line 13 has been executed. Like the butterfly Adjust procedure, the communication cost of Cycle could be decreased in certain cases. The routines Replicate′, Replicate″, Collect′ and Collect″ all execute in $2width$ steps using $O(N)$ messages.

procedure Replicate(p, $start$, $width$, $select$)
(1) Cycle(p, $M - start - width$)

```
    Replicate′(p, width, select)
    Cycle(p, start)
        or
    Cycle(p, M − start)
    Replicate″(p, width, select)
    Cycle(p, start + width)
end Replicate

procedure Collect(p, start, width)
    Cycle(p, M − start − width)
    Collect′(p, width)
    Cycle(p, start)
        or
    Cycle(p, M − start)
    Collect″(p, width)
    Cycle(p, start + width)
end Collect

procedure Cycle(p, shift)
    σ, shift ⟵ (M > 2shift) ? +1, shift : −1, M − shift
    for i ⟵ 0 to shift − 1
        *p^{shuffle^σ} ⟸ *p
    end for
end Cycle

procedure Replicate′(p, width, select)
    for i ⟵ width − 1 downto 0
        if z_0 ∘ z_[M−i,M) = select_[0,i] then
            *p ⟸ *p^{shuffle^−1}
            *p^{exchange} ⟸ *p
        end if
    end for
end Replicate′

procedure Replicate″(p, width, select)
    for i ⟵ 0 to width − 1
        if z_(0,width−i) = select_(i,width) then
            if z_0 = select_i then
                *p^{exchange} ⟸ *p
            end if
            *p^{shuffle^−1} ⟸ *p
        end if
    end for
end Replicate″
```

procedure Collect′(p, $width$)
  for $i \longleftarrow 0$ **to** $width - 1$
    if $z_{[0,i)} = 0$ **then**
      $*p \Longleftarrow *p^{shuffle^{-1}}$
      if $z_0 = 0$ **then**
        $*p \overset{\vee}{\Longleftarrow} *p^{exchange}$
      end if
    end if
  end for
end Collect′

procedure Collect″(p, $width$)
  for $i \longleftarrow 0$ **to** $width - 1$
    if $z_0 \circ z_{[M-i,M)} = 0$ **then**
      $*p \overset{\vee}{\Longleftarrow} *p^{exchange}$
      $*p^{shuffle^{-1}} \Longleftarrow *p$
    end if
  end for
end Collect″

3.4 Multi-Dimensional Mesh of Trees

The k-dimensional mesh of trees of side n, where n is a power of two, may be constructed in the following manner:

1. First assign a unique k-tuple of integers from $[0, n)$ to each of n^k processors. We think of these as being arranged at the corresponding points in k-space. These processors will be referred to as *leaf* processors.

2. For each dimension d, $d \in [0, k)$,

   (a) Partition the leaf processors into n^{k-1} sets of n such that the k-tuples of the processors within a set differ only in the dth component.

   (b) For each such set of n processors

      i. Arrange the set in increasing order of the dth component.

      ii. Connect the set together by forming a binary tree of height $\log n$ using $n-1$ new processors to form the internal nodes of the tree.

Thus, the k-dimensional mesh of trees contains kn^{k-1} trees and a total of

$$n^k + kn^{k-1}(n-1) = (k+1)n^k - kn^{k-1}$$

processors. An interesting aside is that a k-dimensional mesh of trees of side two is the same as a degree k hypercube with every edge replaced by a path of length two.

As indicated in Table 1, the mesh of trees is *not* a high flux network. However, it is powerful enough to achieve an $O(\log n)$ time implementation of the DTEP inner loop because it is (not surprisingly) good at performing tree computations. Our DTEP implementation uses a three-dimensional mesh of trees, but the routines given below are valid for the general case. Note that *width* and *start* must be multiples of m. The step complexity of both Replicate and Collect is $2width$ since information needs to be passed up and down the trees. The communication cost of Replicate is dominated by the last iteration and is $N + O(n^2 \log n)$. The message complexity of Collect is dominated by the first iteration and yields the same result.

When a call to Collect spans more than one dimension, it is possible to achieve the result of equation 1 more rapidly by employing a larger number of messages in the obvious fashion. There is an example of this in Section 4.4.

procedure Replicate(p, $start$, $width$, $select$)
  assert $(start \bmod m = 0) \wedge (width \bmod m = 0)$
  for $i \longleftarrow width$ **downto** m **step** m
    PassUp(p, $(start + i)/m$, $start$, $select$)
    Replicate′(p, $(start + i)/m$, $start$, $select$)
  end for
end Replicate

procedure Collect(p, $start$, $width$)
  assert $(start \bmod m = 0) \wedge (width \bmod m = 0)$
  for $i \longleftarrow 0$ **to** $width - m$ **step** m
    Collect′(p, $(start + i)/m$, $start$)
    PassDown(p, $(start + i)/m$, $start$)
  end for
end Collect

procedure Replicate′(p, d, $start$, $select$)
  for $h \longleftarrow 0$ **to** $m - 1$
    if $z_{[start,md)} = select_{[0,md-start)}$ **then**
      $d, h\colon *p^{leftchild}, *p^{rightchild} \Longleftarrow *p$
    end if
  end for
end Replicate′

procedure Collect′(p, d, $start$)
  for $h \longleftarrow m$ **downto** 1
    if $z_{[start,md)} = 0$ **then**
      $d, h\colon *p^{parent} \Longleftarrow *p \vee *p^{sibling}$
    end if
  end for
end Collect′

procedure PassUp(p, d, $start$, $select$)
  for $h \longleftarrow m$ **downto** 1
    if $z_{[start, md+h)} = select_{[0, md+h-start)}$ **then**
      d, h: $*p^{parent} \Longleftarrow *p$
    end if
  end for
end PassUp

procedure PassDown(p, d, $start$)
  for $h \longleftarrow 1$ **to** m
    if $z_{[start, md+h)} = 0$ **then**
      d, h: $*p \Longleftarrow *p^{parent}$
    end if
  end for
end PassDown

4 Network Implementations of DTEP

In this section we will present several implementations of the DTEP algorithm. In every case each processor maintains a set of nine local variables: P_i, P_j, P_k, P_{ij}, P_{ik}, P_{kj}, P_{ijk}, *previous*, *change*. The subscripts which appear on the first seven variables do not denote indexing in the usual sense; they are intended to indicate what value the variable is expected to contain at a particular processor. Every processor has an M bit z field in its id which can be split into three m bit fields corresponding to i, j and k. Formally, we have

$$z_{[2m,M)} = i, \; z_{[m,2m)} = j, \; z_{[0,m)} = k$$

or equivalently, $z = i_{[0,m)} \circ j_{[0,m)} \circ k_{[0,m)}$.[4] It will be convenient to refer to a processor with $z = i \circ j \circ k$ as processor (i, j, k). This notation is unambiguous for the hypercube, a single rank of the butterfly, the perfect shuffle and the leaf processors of the three-dimensional mesh of trees since there is exactly one processor corresponding to each possible triple. For example, at processor $(*, j, k)$ the variable P_{kj} will "normally" contain the value of the element a_{kj} in the k row and jth column of the $n \times n$ direct implication matrix maintained by the DTEP algorithm. Although not explicitly subscripted, *change* and *previous* depend on i alone.

In order to assist the reader in following our programs, every line which affects the values of one or more local variables is labelled with a corresponding number of triples in the right margin. The triple indicates how the values of a particular variable are distributed amongst the processors. For instance, line 5 of Section 4.1 assigns a value to P_j and is labelled with $(*, j, *)$. This means that all processors with the same j field, $z_{[m,2m)}$, also have the same value for P_j, ie. P_j does not currently depend on the i or k fields. As

[4] Sometimes we will write such an equation as simply $z = i \circ j \circ k$ when the intended "width" of the integers on the right hand side of the equation is clear; leading zeros should be preserved accordingly.

another example, consider line 18 in the same program. It assigns a value to P_{kj} and the corresponding triple is $(k, j, *)$. This says that the value of a_{kj} (as defined above) is currently stored in local variable P_{kj} at those processors with $z_{[2m,M)} = k$ and $z_{[m,2m)} = j$, regardless of the value of the k field.

The input to DTEP consists of n P_i values, n^2 P_{ij} values and n^3 P_{ijk} values. Unless otherwise specified, these will be assumed to reside in processors $(i, 0, 0)$, $(i, j, 0)$ and (i, j, k) respectively, at the start of execution. The output is given by the final values of P_i in processors $(i, 0, 0)$. We have assumed that any processor can terminate the execution of the entire machine, which eliminates the need to broadcast a termination flag in every iteration of the loop. Even if all processors must halt independently, the cost of this broadcast can be hidden from the inner loop analysis by employing a termination bit in every message. The idea is that every time a processor receives a message it will check the termination bit. If it is set, that processor broadcasts termination to its neighbors and then halts.

4.1 Hypercube

The program below implements the DTEP algorithm and performs inter-processor communication solely through calls to Replicate and Collect. By simply plugging in the routines developed in the previous section, one obtains $O(\log n)$ time implementations of the DTEP inner loop for all four of the networks we are studying. The program works as follows. Lines 1 and 2 copy the input P_i and P_{ij} values to processors $(i, *, *)$ and $(i, j, *)$ respectively. Line 4 initializes P_j, P_k and saves the current set of P_i values in *previous*. Lines 5 and 6 redistribute P_j and P_k so that they depend upon the appropriate fields of bits in the processor id. Lines 7 and 8 attempt to derive more P_i, P_{ij} values. Lines 9 and 10 collect and distribute the updated set of P_i values. Lines 11 to 15 check to see whether any new P_i has been derived. Lines 16 and 17 collect and distribute the new set of P_{ij} values. Lines 18 to 20 produce appropriately transposed versions of $[P_{ij}]$ in the P_{ik} and P_{kj} variables. Lines 21 to 23 complete the matrix multiplication; line 21 performs the "multiplications" while lines 22 and 23 perform the "additions".

Running on a hypercube the complexity of this program is given by the entries in the last two columns of Table 2. We can reduce the step count to $9 \log n$ by using the version of Collect with $O(N \log N)$ communication cost described in Section 3.1, which allows lines 10, 17 and 23 to be eliminated.

In a MIMD environment and with a larger hypercube, there is another level of parallelism which can be exploited: independent computations can be performed at the same time on separate subcubes of size N. The loop can be restructured so that it runs in $3 \log n$ steps on a hypercube with $4N$ processors, ie. four subcubes of size N. Assuming that lines 7 and 8 are moved to the top, the first $\log n$ steps make use of two subcubes to perform the first half of the computation of line 9 and the entire computation of line 16 simultaneously. The other two subcubes are idle during this period of time. During the second $\log n$ steps, three subcubes are used to complete the computation of line 9 while performing lines 19 and 20. All four subcubes are used during the third and final set of $\log n$ steps in order to

Network	*Processors*	*Minimum*	*Steps*	*Communication*
hypercube	$N = n^3 = 2^{3m}$	$9 \log n$	$13 \log n$	$12N + O(n^2)$
butterfly	$N \log N$	$12 \log n$	$16 \log n$	$2N \log N + O(N)$
perfect shuffle	N	$13 \log n$	$23 \log n$	$N \log N + O(N)$
3D mesh of trees	$4N - 3n^2$	$17 \log n$	$19 \log n$	$10N + O(n^2 \log n)$

Table 2: Analysis of DTEP inner loop implementations.

execute lines 5, 6, 13 and 22 simultaneously. Notice that this MIMD algorithm would be easy to implement since each of the four subcubes operates in a SIMD manner.

	procedure DTEP	
(1)	Replicate($\&P_i$, 0, $2m$, 0)	$(i, *, *)$
(2)	Replicate($\&P_{ij}$, 0, m, 0)	$(i, j, *)$
(3)	**loop**	
(4)	$previous, P_j, P_k \longleftarrow P_i$	$(i, *, *), (j, *, *), (k, *, *)$
(5)	Replicate($\&P_j$, $2m$, m, $z_{[m,2m)}$)	$(*, j, *)$
(6)	Replicate($\&P_k$, $2m$, m, $z_{[0,m)}$)	$(*, *, k)$
(7)	$P_i \overset{\vee}{\longleftarrow} (P_j \wedge P_k \wedge P_{ijk}) \vee (P_j \wedge P_{ij})$	(i, j, k)
(8)	$P_{ij} \overset{\vee}{\longleftarrow} P_k \wedge P_{ijk}$	(i, j, k)
(9)	Collect($\&P_i$, 0, $2m$)	$(i, 0, 0)$
(10)	Replicate($\&P_i$, 0, $2m$, 0)	$(i, *, *)$
(11)	**if** $z_{[0,2m)} = 0$ **then**	
(12)	$change \longleftarrow P_i \not\equiv previous$	$(i, 0, 0)$
(13)	Collect($\&change$, $2m$, m)	$(0, 0, 0)$
(14)	**exit when** $\neg change$ **at** $(0, 0, 0)$	
(15)	**end if**	
(16)	Collect($\&P_{ij}$, 0, m)	$(i, j, 0)$
(17)	Replicate($\&P_{ij}$, 0, m, 0)	$(i, j, *)$
(18)	$P_{ik}, P_{kj} \longleftarrow P_{ij}$	$(i, k, *), (k, j, *)$
(19)	Replicate($\&P_{ik}$, m, m, $z_{[0,m)}$)	$(i, *, k)$
(20)	Replicate($\&P_{kj}$, $2m$, m, $z_{[0,m)}$)	$(*, j, k)$
(21)	$P_{ij} \longleftarrow P_{ik} \wedge P_{kj}$	(i, j, k)
(22)	Collect($\&P_{ij}$, 0, m)	$(i, j, 0)$
(23)	Replicate($\&P_{ij}$, 0, m, 0)	$(i, j, *)$
(24)	**end loop**	
	end DTEP	

4.2 Butterfly

As shown in Table 2, the butterfly implementation uses $N \log N$ processors. For convenience, we have assumed that the input P_i values are to be found in rank $2m$ and the input

P_{ij} values are in rank m. The output P_i values are in rank 0. In order to minimize communication complexity it is necessary to eliminate as many calls to Adjust as possible since it uses $N \log N$ messages. We were able to get rid of all but two, so the communication cost is as shown in Table 2. As it stands the algorithm has step complexity $17 \log n$. This can be reduced to $16 \log n$ by concatenating P_j and P_k in order to perform lines 7 and 8 with a single call to Replicate′.

For the minimum step count version, the idea that we used for the hypercube applies once again. In this case lines 12, 19 and 27 can be eliminated at the expense of a constant factor increase in communication cost. However, this cannot be done without further restructuring since the rank in which the sets of values in question are left is affected. It is not difficult to perform this restructuring in order to obtain a step count of $12 \log n$. This is $3 \log n$ higher than for the hypercube because three adjustments are performed.

Under a limited MIMD model in which individual ranks still operate in a SIMD fashion, the butterfly with $N \log N$ processors can achieve a step count of $5 \log n$, as stated in Table 3. Calls to Replicate and Collect which make use of disjoint rank intervals may be performed simultaneously, while those for which the intervals overlap can be pipelined.

	procedure DTEP	
(1)	Replicate″($\&P_i$, 0, $2m$, 0)	0: $(i,*,*)$
(2)	Replicate″($\&P_{ij}$, 0, m, 0)	0: $(i,j,*)$
(3)	**loop**	
(4)	0: $previous, P_j, P_k \longleftarrow P_i$	0: $(i,*,*),(j,*,*),(k,*,*)$
(5)	Replicate″($\&P_j$, $2m$, m, $z_{[m,2m)}$)	$2m$: $(*,j,*)$
(6)	Replicate″($\&P_k$, $2m$, m, $z_{[0,m)}$)	$2m$: $(*,*,k)$
(7)	Replicate′($\&P_j$, $2m$, m, 0)	0: $(*,j,*)$
(8)	Replicate′($\&P_k$, $2m$, m, 0)	0: $(*,*,k)$
(9)	0: $P_i \overset{\vee}{\longleftarrow} (P_j \wedge P_k \wedge P_{ijk}) \vee (P_j \wedge P_{ij})$	0: (i,j,k)
(10)	0: $P_{ij} \overset{\vee}{\longleftarrow} P_k \wedge P_{ijk}$	0: (i,j,k)
(11)	Collect′($\&P_i$, 0, $2m$)	$2m$: $(i,0,0)$
(12)	Replicate″($\&P_i$, 0, $2m$, 0)	0: $(i,*,*)$
(13)	**if** $z_{[0,2m)} = 0$ **then**	
(14)	0: $change \longleftarrow P_i \not\equiv previous$	0: $(i,0,0)$
(15)	Collect″($\&change$, $2m$, m)	$2m$: $(0,0,0)$
(16)	**exit when** $\neg change$ **at** $2m$: $(0,0,0)$	
(17)	**end if**	
(18)	Collect′($\&P_{ij}$, 0, m)	m: $(i,j,0)$
(19)	Replicate″($\&P_{ij}$, 0, m, 0)	0: $(i,j,*)$
(20)	Adjust($\&P_{ij}$, 0, $2m$)	$2m$: $(i,j,*)$
(21)	$2m$: $P_{ik}, P_{kj} \longleftarrow P_{ij}$	$2m$: $(i,k,*),(k,j,*)$
(22)	Replicate″($\&P_{ik}$, m, m, $z_{[0,m)}$)	m: $(i,*,k)$
(23)	Adjust($\&P_{ik}$, m, $2m$)	0: $(i,*,k)$
(24)	Replicate′($\&P_{kj}$, $2m$, m, $z_{[0,m)}$)	0: $(*,j,k)$
(25)	0: $P_{ij} \longleftarrow P_{ik} \wedge P_{kj}$	0: (i,j,k)
(26)	Collect′($\&P_{ij}$, 0, m)	m: $(i,j,0)$

Network	*Processors*	*Minimum*
hypercube	$4N$	$3\log n$
butterfly	$N\log N$	$5\log n$
3D mesh of trees	$4N-3n^2$	$8\log n$

Table 3: Minimum step counts for MIMD implementations.

(27) Replicate″($\&P_{ij}$, 0, m, 0) — 0: $(i,j,*)$
(28) **end loop**
end DTEP

4.3 Perfect Shuffle

For the perfect shuffle implementation it is convenient to assume that P_i is given in $(0,0,i)$ and P_{ij} in $(0,i,j)$. The output value of P_i is still to be found in processor $(i,0,0)$, however. We were able to eliminate all but one of the calls to Cycle so the communication cost is as shown in Table 2. There is an interesting trick which can be used to decrease the number of steps per iteration by $2\log n$. As observed by Dekel et al., the perfect shuffle can compute the transpose of the product of two matrices more rapidly than the actual product [DNS81]. This fact may be used to essentially get rid of the calls to Replicate on lines 20 and 21. In order to make use of the transpose of $[P_{ij}]^2$ it is necessary to unroll the loop body by a factor of two and maintain some extra local variables. Unfortunately, there is now a data alignment problem between consecutive iterations. This could be solved with a call to Cycle or by unrolling the loop body by a further factor of three. Of course, the results in Table 2 reflect the latter choice.

procedure DTEP
(1) Replicate′($\&P_i$, $2m$, 0) — $(i,*,*)$
(2) Replicate′($\&P_{ij}$, m, 0) — $(i,j,*)$
(3) **loop**
(4) $previous, P_j \longleftarrow P_i$ — $(i,*,*),(j,*,*)$
(5) Replicate″($\&P_j$, m, $z_{[0,m)}$) — $(*,j,*)$
(6) $P_k \longleftarrow P_j$ — $(*,k,*)$
(7) Replicate″($\&P_k$, m, 0) — $(*,*,k)$
(8) $P_i \overset{\vee}{\longleftarrow} (P_j \wedge P_k \wedge P_{ijk}) \vee (P_j \wedge P_{ij})$ — (i,j,k)
(9) $P_{ij} \overset{\vee}{\longleftarrow} P_k \wedge P_{ijk}$ — (i,j,k)
(10) Collect″($\&P_i$, $2m$) — $(0,0,i)$
(11) Replicate′($\&P_i$, $2m$, 0) — $(i,*,*)$
(12) **if** $z_{[0,2m)} = 0$ **then**
(13) $change \longleftarrow P_i \not\equiv previous$ — $(i,0,0)$
(14) Collect′($\&change$, m) — $(0,0,0)$
(15) **exit when** $\neg change$ **at** $(0,0,0)$
(16) **end if**

(17)	Collect″(&P_{ij}, m)	$(0,i,j)$
(18)	Replicate′(&P_{ij}, m, 0)	$(i,j,*)$
(19)	$P_{kj} \longleftarrow P_{ij}$	$(k,j,*)$
(20)	Replicate′(&P_{kj}, m, $z_{[0,m)}$)	$(j,k,*)$
(21)	Replicate″(&P_{kj}, m, 0)	$(*,j,k)$
(22)	$P_{ik} \longleftarrow P_{kj}$	$(*,k,i)$
(23)	Cycle(&P_{ik}, $2m$)	$(i,*,k)$
(24)	$P_{ij} \longleftarrow P_{ik} \wedge P_{kj}$	(i,j,k)
(25)	Collect″(&P_{ij}, m)	$(0,i,j)$
(26)	Replicate′(&P_{ij}, m, 0)	$(i,j,*)$
(27)	**end loop**	
	end DTEP	

Since there are quite a few minor differences between it and the preceding program, the minimum step count version is presented in its entirety. The input/output variables are the same except that P_i begins in processor $(i,0,0)$. As above, it is possible to save $2\log n$ steps by loop unrolling and computing the transpose of the square; in this case it is the work performed by lines 44 and 49 which becomes unnecessary.

At this point, one might hope to obtain a MIMD version with an even lower step count, as we did for the hypercube. Unfortunately, the perfect shuffle organization does not lend itself well to partitioning schemes; a significant amount of overhead seems to be necessary to maintain the partition. In the present case, it appears that the extra steps required to handle this overhead would entirely offset any potential decrease, so this strategy is not worthwhile.

	procedure DTEP	
(28)	Replicate″(&P_i, $2m$, 0)	$(*,*,i)$
(29)	Replicate′(&P_{ij}, m, 0)	$(i,j,*)$
(30)	**loop**	
(31)	$previous, P_j, P_k \longleftarrow P_i$	$(*,*,i), (*,*,j), (*,*,k)$
(32)	Cycle(&P_i, $2m$)	$(i,*,*)$
(33)	Cycle(&P_j, m)	$(*,j,*)$
(34)	$P_i \overset{\vee}{\longleftarrow} (P_j \wedge P_k \wedge P_{ijk}) \vee (P_j \wedge P_{ij})$	(i,j,k)
(35)	$P_{ij} \overset{\vee}{\longleftarrow} P_k \wedge P_{ijk}$	(i,j,k)
(36)	Collect″(&P_i, $2m$)	$(*,*,i)$
(37)	**if** $z_{[m,M)} = 0$ **then**	
(38)	$change \longleftarrow P_i \not\equiv previous$	$(0,0,i)$
(39)	Collect″(&$change$, m)	$(0,0,0)$
(40)	**exit when** $\neg change$ **at** $(0,0,0)$	
(41)	**end if**	
(42)	Collect″(&P_{ij}, m)	$(*,i,j)$
(43)	$P_{kj} \longleftarrow P_{ij}$	$(*,k,j)$
(44)	Replicate″(&P_{kj}, m, $z_{[2m,M)}$)	$(*,j,k)$
(45)	$P_{ik} \longleftarrow P_{kj}$	$(*,k,i)$

Cycle($\&P_{ik}$, $2m$) $(i,*,k)$
$P_{ij} \longleftarrow P_{ik} \wedge P_{kj}$ (i,j,k)
Collect″($\&P_{ij}$, m) $(*,i,j)$
Cycle($\&P_{ij}$, m) $(i,j,*)$
end loop
end DTEP

4.4 Multi-Dimensional Mesh of Trees

Our multi-dimensional mesh of trees implementation is only a slightly modified version of the program given in Section 4.1. By eliminating three redundant PassUp, PassDown pairs we obtain the step count and communication cost stated in Table 2. For example, lines 9 and 10 from Section 4.1 get translated into the block of code given below.

⋮
(1) Collect′($\&P_i$, 0, 0)
(2) PassDown($\&P_i$, 0, 0)
(3) Collect′($\&P_i$, 1, 0)
(4) Replicate′($\&P_i$, 1, 0, 0)
(5) PassUp($\&P_i$, 0, 0, 0)
(6) Replicate′($\&P_i$, 0, 0, 0)
⋮

The minimum step count version can be achieved by using more messages in lines 2 to 4 so that 5 and 6 can be eliminated. This does not result in an asymptotic increase in message complexity; it just increases the coefficient on the leading term from 10 to 12.

Using the techniques we have discussed for the other networks, it is easy to derive a MIMD implementation of the DTEP inner loop which runs in $8 \log n$ steps without increasing the number of processors.

5 Conclusions

Tables 2 and 3 summarize the results of our analysis. The communication cost of our implementations could be further reduced by only attempting to derive P_i at those processors where it is **false**. Note that this requires data dependent conditions for enabling/disabling processors.

It is possible to use bit compression techniques to reduce the processor requirements of every one of our implementations by a factor of $\log n$ [Pl87]. For all of the networks we have considered except the perfect shuffle, this can be done without increasing the coefficient on the leading term of the running time. For the same set of networks, an extension of an idea due to Dekel & Sahni [DS83] allows the processor requirements to be lowered by an additional factor of $\log n$. However, this reduction increases the running time by a constant factor and requires a MIMD model for the butterfly and multi-dimensional mesh of trees [Pl87].

A List of Symbols

$\&$	address operator
$*$	indirection operator; also used as wildcard
$\vee$	logical OR operator
$\wedge$	logical AND operator
$\neg$	logical negation operator
$\equiv$	logical equivalence operator
$=$	equality operator
$\longleftarrow$	local assignment operator
$x \stackrel{op}{\longleftarrow} y$	$x \longleftarrow x\ op\ y$
$\Longleftarrow$	inter-processor assignment operator
$x \stackrel{op}{\Longleftarrow} y$	$x \Longleftarrow x\ op\ y$
$(c)\ ?\ x\ :\ y$	conditional expression: **if** c **then** x **else** y
$[a, b]$	$(a \leq b)\ ?\ \{a, a+1, \ldots, b\}\ :\ \{\}$
$[a, b)$	$(a < b)\ ?\ \{a, a+1, \ldots, b-1\}\ :\ \{\}$
$(a, b]$	$(a < b)\ ?\ \{a+1, a+2, \ldots, b\}\ :\ \{\}$
(a, b)	$(a+1 < b)\ ?\ \{a+1, a+2, \ldots, b-1\}\ :\ \{\}$
x_i	ith bit of x (low order bit is x_0)
$x_{\{a,a+1,\ldots,b\}}$	$(a \leq b)\ ?\ (x_b x_{b-1} \cdots x_a)_2\ :\ 0$
$\circ$	bit string concatenation, eg. $1_{[0,2)} \circ 12_{[1,4)} = 01_2 \circ 110_2 = 01110_2$
$\ll$	shift left operator, eg. $101_2 \ll 3 = 101000_2$
$\gg$	shift right operator, eg. $101_2 \gg 1 = 10_2$
$\oplus$	bitwise XOR operator
x **at** z	x at processor z
$[a_{ij}]$	the matrix of a_{ij}'s
$\log x$	$\log_2 x$
$\Theta(f(n))$	$O(f(n))$ and $\Omega(f(n))$

For each network family we require some additional notation for specifying processor ids and adjacencies. For the hypercube we have

z	M bit processor id
$x^{(l)}$	x **at** $z \oplus 2^l$

Each processor in a butterfly network has a processor id consisting of two components: *rank* and z. The following notation is used

rank	$\lceil \log M \rceil$ high bits of id; specifies rank
z	M low bits of id; specifies position within rank
x_{+1}	x **at** $(rank+1, z)$
x_{-1}	x **at** $(rank-1, z)$
x'_{+1}	x **at** $(rank+1, z \oplus 2^{rank})$

x'_{-1}	x **at** $(rank-1, z \oplus 2^{rank-1})$

Also, if a statement is labelled with a number r followed by a colon then it is executed only at those processors with $rank = r$. A butterfly with N processors per rank has $\log N$ ranks; we identify the top and bottom ranks. All arithmetic involving ranks should be assumed to be performed modulo $\log N$ (eg. x_{-1} **at** $(0, z)$ is the same as x **at** $(\log N - 1, z)$).

For the perfect shuffle we specify ids and adjacencies in the following manner

z	M bit processor id
$x^{exchange}$	x **at** $z \oplus 1$
$\hookleftarrow$	unary rotate left operator, eg. $\hookleftarrow 10111_2 = 01111_2$
$\hookrightarrow$	unary rotate right operator, eg. $\hookrightarrow 01011_2 = 10101_2$
$x^{shuffle}$	x **at** $\hookleftarrow z$
$x^{shuffle^{-1}}$	x **at** $\hookrightarrow z$

For the multi-dimensional mesh of trees family, we assign each processor an id which is most conveniently thought of as a triple $(dim, height, z)$. The *height* of a processor is its distance from the nearest root. Assume N, n, k are as defined in Table 1 and $M = \log N$, $m = \log n$. The *dim* field is irrelevant for the n^k leaf processors (those with $height = m$) since they each belong to every dimension. We use the following notation

dim	$\lceil \log k \rceil$ high bits of id; belongs to $[0, k)$
$height$	$\lceil \log(m+1) \rceil$ middle bits of id; belongs to $[0, m]$
z	M low bits of processor id
z^d	$z_{[md, md+m)}$, $d \in [0, k)$
$subst(z, x, d)$	$z_{[md+m, M)} \circ x_{[0,m)} \circ z_{[0,md)}$, $d \in [0, k)$
x^{parent}	x **at** $(dim, height-1, subst(z, z^{dim} \gg 1, dim))$
$x^{leftchild}$	x **at** $(dim, height+1, subst(z, z^{dim} \ll 1, dim))$
$x^{rightchild}$	x **at** $(dim, height+1, subst(z, (z^{dim} \ll 1) + 1, dim))$
$x^{sibling}$	x **at** $(dim, height, subst(z, z^{dim} \oplus 1, dim))$

Note that a reference to the parent of a leaf processor is not well defined unless it is accompanied by a dimension. In our programs the intended dimension of the parent of a leaf will not be given explicitly but should be obvious. If a statement is labelled with a pair d, h followed by a colon then it is only executed at those processors with $dim = d$ and $height = h$.

References

[AL78] T. Agerwala and B. Lint. Communication in parallel algorithms for Boolean matrix multiplication. In *Proc. 1978 IEEE International Conference on Parallel Processing*, pages 146-153, 1978.

[DNS81] E. Dekel, D. Nassimi and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comp.*, 10:657-675, 1981.

[DS83] E. Dekel, S. Sahni. Binary trees and parallel scheduling algorithms. *IEEE Transactions on Computers*, 32(3):307-315, 1983.

[Fl66] M. Flynn. Very high speed computing systems. *Proc. IEEE*, 54:1901-1909, 1966.

[GMU87] L. Goldschlager, E. Mayr and J. Ullman. *Theory of Parallel Computation*. To appear.

[Go80] L. Goldschlager. A space efficient algorithm for the monotone planar circuit value problem. *Information Processing Letters*, 10:25-27, 1980.

[Ma87] E. Mayr. The dynamic tree expression problem. Stanford University Department of Computer Science Technical Report No. STAN-CS-87-1156.

[MR85] G. Miller and J. Reif. Parallel tree contraction and its applications. In *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 478-489, 1985.

[Pl87] G. Plaxton. Research notes.

[Ru80] W. Ruzzo. Tree-size bounded alternation. In *Proc. 11th Annual ACM Symposium on Theory of Computing*, pages 352-359, 1979.

[St71] H. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computing*, C-20:153-161, 1971.

[Ul84] J. Ullman. *Computational Aspects of VLSI*, Computer Science Press, Rockville, 1984.

[UV85] J. Ullman and A. Van Gelder. Parallel complexity of logical query programs. Stanford University Department of Computer Science Technical Report No. STAN-CS-85-1089.