T U M

I N S T I T U T F Ü R I N F O R M A T I K

Optimal Tree Contraction and Term Matching on the Hypercube and Related Networks

Ernst W. Mayr Ralph Werchner

TUM-I9532
November 1995

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-1995-I9532-300/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

Optimal Tree Contraction and Term Matching on the Hypercube and Related Networks

Ernst W. Mayr*
Institut für Informatik
Technische Universität
München, Germany

Ralph Werchner†
Fachbereich Informatik
J.W. Goethe-Universität
Frankfurt am Main, Germany

November 8, 1995

Abstract

An optimal tree contraction algorithm for the boolean hypercube and the constant degree hypercubic networks, such as the shuffle exchange or the butterfly network, is presented. The algorithm is based on novel routing techniques and, for certain small subtrees, simulates optimal PRAM algorithms. For trees of size n, stored on a p processor hypercube in in-order, the running time of the algorithm is $O(\left\lceil \frac{n}{p} \right\rceil \log p)$. The resulting speed-up of $O(p/\log p)$ is optimal due to logarithmic communication overhead, as shown by a corresponding lower bound.
The same algorithmic ingredients can also be used to solve the term matching problem, one of the fundamental problems in logic programming.

*mayr@informatik.tu-muenchen.de
†werchner@informatik.uni-frankfurt.de

1 Introduction

Tree contraction is a fundamental technique for solving problems on trees. A given tree is reduced to a single node by repeatedly contracting edges, *resp.,* merging adjacent nodes. This operation can be used for problems like *top-down* or *bottom-up algebraic tree computations* [ADKP89], which themselves can be applied to solve the membership problem for certain subclasses of languages in DCFL [GR86] or to evaluate expressions consisting of rational operands and the operators $+, -, \cdot, /$.

In the context of parallel processing the objective is to contract the tree using a small number of stages. In a stage, a set of disjoint pairs of adjacent nodes in the current tree is selected, and the edges connecting these pairs are contracted, *i.e.,* each pair of nodes is merged into a single node. The contractions are not allowed to produce a node with more than two children. Brent [Bre74] was the first to show that a logarithmic number of such stages is sufficient, and he applied this result to the restructuring of algebraic expression trees, producing trees of logarithmic depth. Subsequent work [MR85, GR86, GMT88, KD88, ADKP89] concentrated on the efficient parallel computation of contraction sequences, and eventually resulted in work optimal logarithmic time tree contraction algorithms on the EREW PRAM.

We consider the tree contraction problem on the boolean hypercube and on similar networks. In this model, computing a suitable contraction sequence is more difficult, and the additional problem of routing pairs of nodes to be contracted to common processors arises. For the contraction of (suitably represented) paths, both problems become trivial and can be solved in logarithmic time by a parallel prefix operation [Sch80]. For arbitrary trees, we combine two known (PRAM) contraction techniques with a recursive approach. To achieve a logarithmic running time we separate the local communication operations from the few long distance communication steps and perform them in appropriately sized subcubes. The necessary routing steps are performed by new logarithmic time algorithms designed for special classes of routings. Our algorithm contracts a tree of size n on a p processor boolean hypercube in $O(\left\lceil\frac{n}{p}\right\rceil \log p)$ steps, which we also show is asymptotically optimal by a matching lower bound.

Term matching is an important special case of the unification problem. It is one of the most time consuming tasks in logic programming and term rewriting systems. A term is an expression composed of function symbols (with arbitrary arity), constants, and variables. In the term matching problem, we are given two terms A and B, with B containing no variable symbols. The task is to find a substitution of terms for the variables (in A) making A syntactically identical to B. Consider, for example, $A = \mathrm{F}(\mathrm{G}(x, y, \mathrm{a}), \mathrm{F}(x, \mathrm{b}))$ and $B = \mathrm{F}(\mathrm{G}(\mathrm{b}, \mathrm{F}(\mathrm{a}, \mathrm{b}), \mathrm{a}), \mathrm{F}(\mathrm{b}, \mathrm{b}))$; the desired substitution is $x \mapsto \mathrm{b}$; $y \mapsto \mathrm{F}(\mathrm{a}, \mathrm{b})$.

In [PW78] Paterson and Wegman solve the general unification problem sequentially in linear time. Later, this problem was shown $\mathcal{P}$-complete in [DKM84]. The first parallel algorithms for the term matching problem achieving optimal speed-up and logarithmic running time are due to Kedem and Palem [KP92]. Their algorithms use the CRCW PRAM or the randomized CREW PRAM model. Employing a recursive approach conceptually very similar to the one presented in this paper, Kosaraju and Delcher [KD90] developed a work-optimal logarithmic time term matching algorithm for the CREW PRAM model.

Term matching naturally decomposes into two phases. First, all pairs of *corresponding* symbols in A and B are determined, where this correspondence is defined by considering the ordered trees T_A and T_B corresponding to the terms A and B respectively. For a node v on level i of a k-ary tree T let $\mathrm{path}_T(v) \in \{1, \ldots, k\}^i$ denote the *path description* of v in T. It is $\mathrm{path}_T(v) = (j_1, \ldots, j_{i-1}, j_i)$ iff v is the j_i-th child of its parent w and $\mathrm{path}_T(w) =$

$(j_1, \ldots, j_{i-1})$. Two nodes u and v correspond to each other iff $\mathrm{path}_{T_A}(u) = \mathrm{path}_{T_B}(v)$.

Since the symbols of A appear in the same order as the corresponding symbols in B, a single monotone routing (respectively shift) of the symbols in both terms suffices to align corresponding symbols with each other. In the following, this first part will be called the *term alignment problem*. In the second phase of term matching, all pairs of corresponding symbols and all subtrees of T_B corresponding to the same variable are tested for equality.

One contribution of this paper is an optimal hypercube algorithm for the term alignment problem. A p processor hypercube can compute and perform the desired routing for terms of size n in $O(\lceil \frac{n}{p} \rceil \log p)$ steps. Applying our general approach, we first align the terms A and B roughly, considering only their global structure, and postpone the exact alignment to a recursive call in lower dimensional subcubes. The task of comparing subtrees mapped to the same variable generally has to be performed by a standard sorting algorithm. This can be done by a randomized sorting algorithm [VB81, ALMN90] within the same time bound $O(\lceil \frac{n}{p} \rceil \log p)$. The best known deterministic sorting algorithm [CP90] exceeds this time bound by a factor of $O((\log \log p)^2)$.

2 Fundamental Concepts and Notation

We first give a short description of our model of computation. A *network of processors* is a set of processors, interconnected by bidirectional communication links. Each processor has the capabilities of a RAM, a unique processor-id and additional instructions to send or receive one machine word to respectively from a direct neighbor. We assume the word length of the processors to be $\Theta(\log p)$ bits where p is the number of processors. The topological structure of the communication links can be described by an undirected graph (V, E).

A *d-dimensional (boolean) hypercube* is a network of processors represented by the graph $G = (V, E)$ with

$$\begin{aligned} V &= \{0,1\}^d \ , \\ E &= \{(u,v) \mid u \text{ and } v \text{ differ in exactly one bit}\} \ . \end{aligned}$$

A network with bounded degree and a structure very similar to the hypercube is the *d-dimensional shuffle-exchange*. The processors are numbered as they are in the hypercube. Two processors are connected by a link if their ids differ only in the last bit (*exchange* edges) or if one id is a cyclic shift by one position of the other id (*shuffle* edges). A compendium of results concerning hypercubes and shuffle-exchange networks can be found in [Lei92].

For the tree contraction problem on networks of processors we assume that the data comprising each node can be stored in a constant number of processor words and that two adjacent nodes stored by the same processor can be merged in constant time. Note that this implies in particular that the number of nodes must be polynomial in the number of processors.

On networks the complexity of a problem may depend heavily on the distribution of the input data over the processors of the network. For the tree contraction problem we assume the in-order sequence of the nodes to be evenly distributed over the sequence of processors ordered by processor-id. To uniquely describe the structure of the tree in this way, each nontrivial subtree is assumed to be enclosed by a pair of parentheses. For the term matching problem we assume that both terms – the pre-order representations of the corresponding trees – are evenly distributed over the sequence of processors. Permitting more general tree representations, like lists of edges, seems to imply that sorting and/or list ranking subproblems have to be solved. The best known hypercube algorithms for these problems would yield inferior complexity bounds for tree contraction.

On a hypercube or hypercubic network with p processors the representation of a tree of size n can be transformed between pre-, in-, and post-order in time $O(\lceil \frac{n}{p} \rceil \log p)$ [MW92]. Thus the asymptotic complexity of both the tree contraction and the term matching problem does not depend on whether the input data is given in pre-, in-, or post-order. If the tree is given by an arbitrary, but balanced distribution of the nodes linked by pointers, the nodes have to be rearranged into pre- or in-order before applying the algorithms presented in this paper. This task can be performed by a list ranking on the Euler tour of the tree and a corresponding routing operation. On a p processor hypercube, this transformation can be carried out in $O(\lceil \frac{n}{p} \rceil \log^2 p \log\log\log p \log^* p)$ steps for trees of size n [HM93]. The dominating part is the time required for the list-ranking. As already mentioned, this running time is much higher than the running times of our tree contraction and term matching algorithms designed for the in- or pre-order representation.

The algorithms given in the next sections use the following basic operations known to be executable in logarithmic time on a hypercube or shuffle-exchange network:

- parallel-prefix-operation and segmented parallel-prefix-operation [Sch80];
- monotone routing (the relative ordering of the data items remains unchanged) [NS81];
- sparse enumeration sort (sorting p^α data items on a p processor hypercube, for some fixed $\alpha < 1$) [NS82];
- parentheses-structured routing (routing between matching pairs in a well-formed string of parentheses) [MW92].

3 Tree Contraction on the Hypercube

In this section we show how binary trees (*i.e.,* each internal node has exactly two children) of size n can be contracted on a p processor hypercube or hypercubic network in $O(\lceil \frac{n}{p} \rceil \log p)$ steps. It can easily be seen that, using standard techniques, more general expressions trees can be transformed into binary trees without asymptotically increasing the complexity. We concentrate on the contraction of trees with at most p nodes in logarithmic time on the hypercube. For inputs of a size $n > p$, a hypercube of size $\Theta(n)$ can be simulated with a slowdown of $\Theta(\frac{n}{p})$. Because of the simple communication structure, the hypercube algorithm can be simulated on a shuffle exchange network of the same size with just constant slowdown. Furthermore, due to a general simulation result [Sch90], any algorithm on the shuffle exchange network can be simulated with constant slowdown on each of the other so called hypercubic networks, such as the de Bruijn network, the cube connected cycles network, or the butterfly network.

We give an outline of the contraction algorithm. It proceeds in three phases:

1. We identify and contract small subtrees in a recursive call of the algorithm.
2. The remaining tree may still be quite large, but, if so, has a structure that allows a significant further reduction by eliminating the leaves and contracting chains of nodes with a single child each to one edge, an operation similar to the "compact" operation in [MR85].

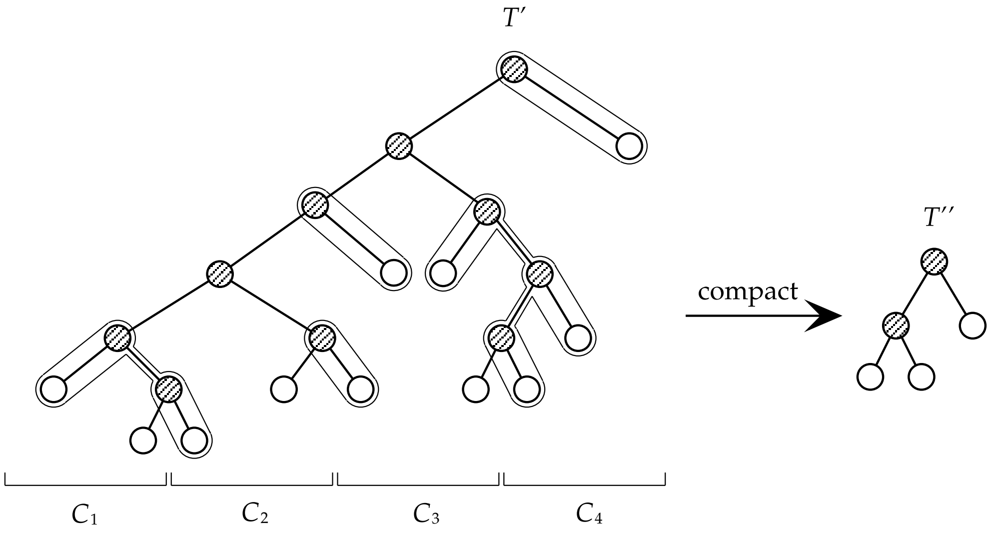

Figure 1: The compact operation applied to T'

3. To contract the remaining tree, we emulate the PRAM algorithm of [KD88]. We first compute the communication structure of this algorithm when executed on the remaining tree. After rearranging the nodes of the tree accordingly in the hypercube, each step of the PRAM tree contraction algorithm can then be simulated in constant time.

3.1 Contraction of Subtrees

In the first phase, we divide the processors of the d-dimensional hypercube into contiguous blocks of size $2^{\lfloor \frac{3}{4}d \rfloor}$. These blocks are $\left\lfloor \frac{3}{4}d \right\rfloor$-dimensional subcubes, say $C_1, C_2, \ldots, C_{2^{\lceil d/4 \rceil}}$. We determine those subtrees contained entirely within a single C_i. For this purpose, we first compute the *height*, *i.e.,* the level of nesting, of the parentheses enclosing the subtrees. An opening parenthesis with height h is part of a local subtree iff there is a parenthesis to its right, but within its block, with a height $\leq h$. Performing a parallel-prefix-computation within each block C_i, we can for each parenthesis compute the minimum height of the parentheses to its right within the block. Comparing this value with its own height, each opening parenthesis can determine whether it is matched within its block. An analogous computation is performed for the closing parentheses.

In a recursive call to the algorithm, for all subcubes C_i in parallel, the subtrees within blocks are contracted. Clearly, the resulting tree is again binary. In general, a subcube C_i may contain a sequence of several maximal local subtrees. This causes the need to make the algorithm capable of contracting a sequence of trees. We only describe here how to contract a single tree; the modifications required for dealing with a sequence of trees, however, are quite straightforward, and are left to the reader.

3.2 Compaction

Denote by T, T', and T'' the trees at the beginning of the algorithm, after the first phase, and after the second phase, respectively. Let L be the set of pairs (u, v), with v a leaf of T', u its parent, and v not the left sibling of another leaf. We reduce T' to T'' by eliminating all pairs of nodes in L (see Figure 1).

This reduction is equivalent to one "rake" operation (eliminating the leaves) and a repeated "compress" operation (contracting the maximal chains of nodes with a single child). Together, they were called a "compact" operation in [MR85]. The result, T'', is again a binary tree.

We first prove that the resulting tree is small. Then we show how the compact operation can be carried out on the hypercube in logarithmic time.

Lemma 1 *Starting with an expression tree of size at most 2^d, applying the first and second phase of our algorithm, the resulting tree T'' is a binary tree with at most $2^{\lceil d/4 \rceil} - 1$ leaves.*

Proof: Each leaf of T' corresponds to a maximal subtree of T initially stored totally within one of the blocks $C_1, C_2, \ldots, C_{2^{\lceil d/4 \rceil}}$. Assign to each leaf the index of its subcube. Then the sequence of numbers assigned to the leaves in T' is sorted and two leaves that are siblings are assigned different numbers. As the only leaves of T' surviving in T'' are those whose right sibling is also a leaf, they are all assigned distinct numbers. Since these numbers are in the range from 1 to $2^{\lceil d/4 \rceil} - 1$, T'' has at most $2^{\lceil d/4 \rceil} - 1$ leaves.

■

To perform the compact operation efficiently on the hypercube we first swap some siblings in the tree such that for all $(u, v) \in L$, the leaf v is a right child of u. For our in-order representation of the tree this means that we modify the tree according to the following pattern:

$$\ldots (v\, u(\ldots)) \ldots \quad \longrightarrow \quad \ldots ((\ldots) u'\, v) \ldots\ , \tag{1}$$

where u is modified to u' to denote that its children have been interchanged.

The routing required for the modification belongs to a special class of partial permutations called *parentheses structured* routings [MW92]. To define this class consider the following situation: Some of the processors are storing each an opening or closing parenthesis. Together with some of the opening parentheses data items are stored. The sequence of parentheses is well-formed, and each data item has to be routed from its opening parenthesis to the processor storing the matching closing parenthesis. A partial permutation that can be described in this way is called a parentheses structured routing. The algorithm proposed in [MW92] performs the routing in logarithmic time on the hypercube, the shuffle exchange network or any other hypercubic network. In (1), the routing of the data item (u, v) can be guided by the pair of parentheses enclosing the subtree rooted at u.

Let $(u_1, v_1), (u_2, v_2), \ldots, (u_r, v_r)$ be a maximal chain of tupels from L in T', *i.e.*, u_i is a child of u_{i+1} for $1 \leq i < r$, and u_r is either the root or the sibling of an internal node, and v_1's sibling w is either a leaf or the parent of two internal nodes. After the above routing step, this chain is, in in-order notation, a sequence

$$\ldots u_1 v_1) u_2 v_2) \ldots) u_r v_r \ldots\ .$$

In each such chain, the edges (u_i, v_i) are contracted by routing the leaves v_i one position to the left. The remaining path $u_1, \ldots, u_r$ is contracted to a single node by a segmented parallel-prefix-operation. Finally, the resulting node is routed to w by a sparse-enumeration-sort, and the two nodes are merged.

3.3 Simulation of PRAM Contraction Algorithm

In the third and last phase, we simulate the logarithmic time PRAM tree contraction algorithm proposed in [KD88] or [ADKP89]. Although Lemma 1 guarantees that there

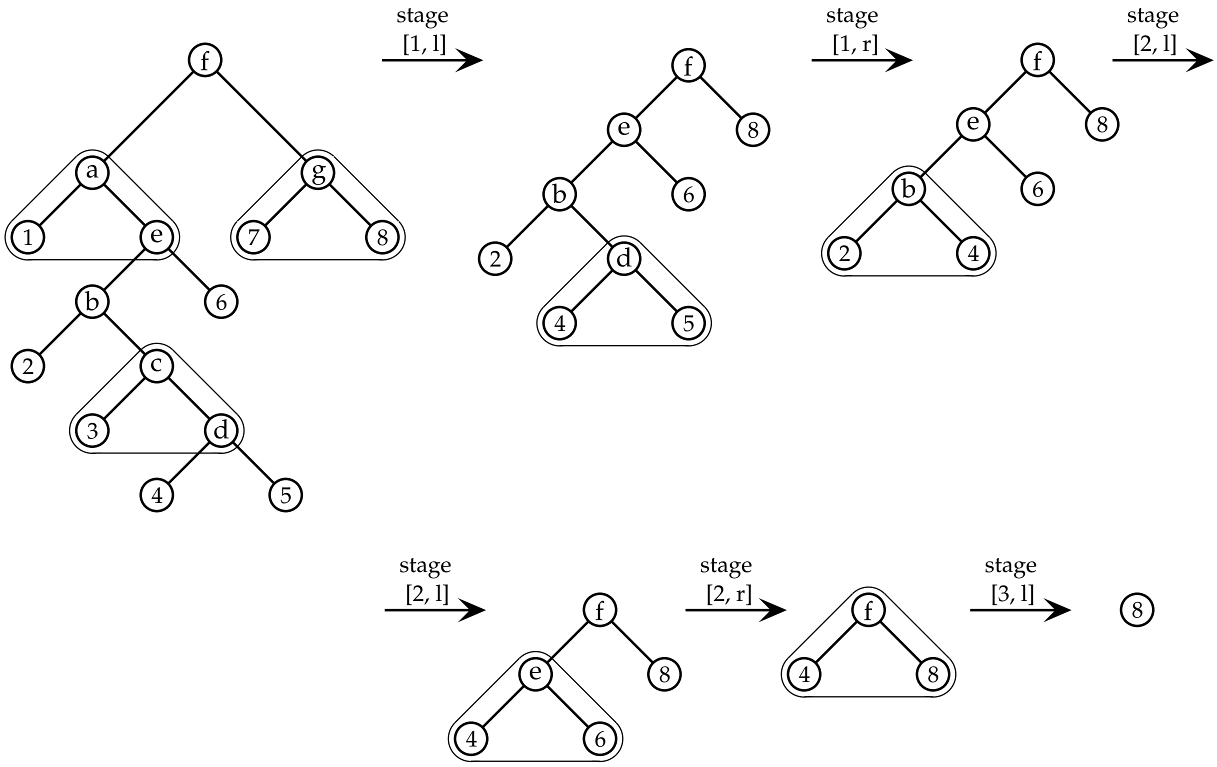

Figure 2: Contraction of T''

are only very few nodes left, a step-by-step simulation of the PRAM algorithm, performing a routing phase for each parallel random access step of the PRAM, would still result in a suboptimal algorithm.

In our approach, we first compute the communication structure of the complete execution of the PRAM algorithm (without a complete simulation). Then, we route the nodes that will have to communicate to adjacent hypercube processors. After rearranging the data, we are able to simulate the PRAM tree contraction algorithm without slowdown.

In [KD88] the tree is reduced using the operation "rake". This operation, applied to a leaf l of the tree, involves three nodes: the leaf l, its parent p, and its sibling s. The two edges connecting these nodes are contracted and the nodes l and p are eliminated while node s survives.

If the given tree has m leaves, the contraction algorithm proceeds in $2 \cdot \lfloor \log m \rfloor$ stages called [1,$\mathtt{l}$], [1,$\mathtt{r}$], [2,$\mathtt{l}$], [2,$\mathtt{r}$], $\ldots$, [$\lfloor \log m \rfloor$,$\mathtt{l}$], [$\lfloor \log m \rfloor$,$\mathtt{r}$]. The leaves are numbered left to right from 1 to m. In stage [i,$\mathtt{l}$] (resp., [i,$\mathtt{r}$]) the rake operation is applied to all leaves numbered by odd multiples of 2^{i-1} that are *left* (resp. *right*) children. In this fashion, no node is involved in more than one rake operation in each stage (see Figure 2 for an example).

We apply this contraction algorithm to the tree T''. For a stage x, let T_x be the reduced tree just before stage x. The contraction of T'' can be represented by a tree $\tilde{T}$ (continuing the example, see Figure 3). For each stage x of the contraction procedure there is one level of $\tilde{T}$ containing exactly the nodes of T_x. In $\tilde{T}$, a node v with children v_1, v_2, v_3 corresponds to the contraction of v_1, v_2, v_3 to v. The depth of $\tilde{T}$ is bounded by $\lfloor d/2 \rfloor$ since T'' has $m < 2^{\lceil d/4 \rceil}$ leaves. Thus $\tilde{T}$ can easily be embedded in a d-dimensional cube with dilation 2. To contract T'', we compute this embedding of $\tilde{T}$, route the nodes of T'' to the corresponding leaves of $\tilde{T}$, and perform the contraction along the (embedded) edges of $\tilde{T}$. Note that the nodes of T'' can be routed in logarithmic time by a sparse

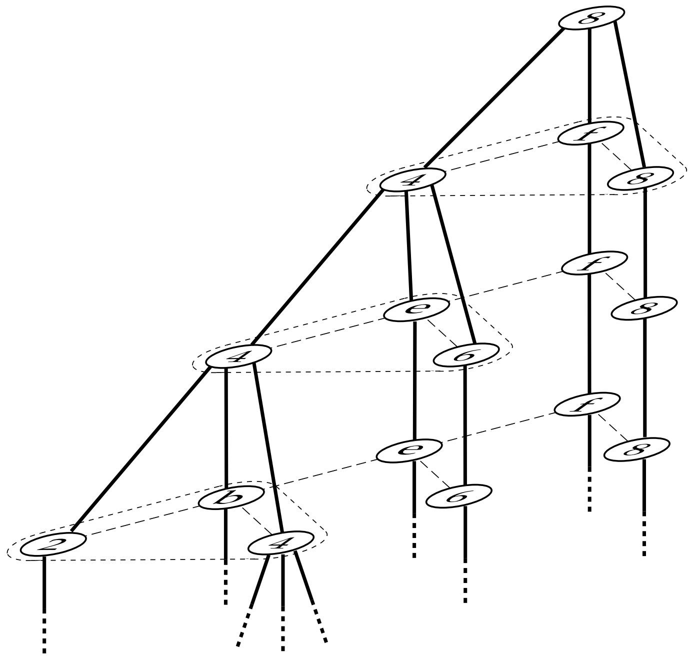

Figure 3: The top 4 levels of the corresponding $\tilde{T}$

enumeration sort followed by a monotone routing. The major problem is to compute their destinations according to the embedding of $\tilde{T}$.

For each node v of $\tilde{T}$, we define $p(v) \in \{0,1,2,3\}^{\lfloor d/2 \rfloor}$ as the description of the path from the root of $\tilde{T}$ to v. The i-th element of $p(v)$ is $j \in \{1,2,3\}$ if the i-th edge on this path points to a j-th child, and an appropriate number of 0's is appended to pad $p(v)$ to length $\lfloor d/2 \rfloor$. Substituting each number of $p(v)$ by its two-bit-representation, the hypercube address of v in the embedding of $\tilde{T}$ can easily be obtained. It remains to show how to compute $p(v)$ efficiently for the leaves of $\tilde{T}$, *i.e.* the nodes of T''.

Investigating the tree contraction algorithm, the following facts can easily be derived:

(a) The nodes of $T_{[i,\mathrm{l}]}$ are the leaves of T'' numbered by multiples of 2^{i-1} and their lowest common ancestors in T''.

(b) Let v be a node of $\tilde{T}$ on a level corresponding to some stage x of the contraction. The descendants of v in $\tilde{T}$ are those nodes of T'' that have been contracted into v. These are v itself and those nodes u connected in T'' (considered as an undirected tree) to the parent of v by a path (including u and v) containing no node of T_x.

We show how to compute the strings $p(v)$ for all nodes v of T''. The elements of $p(v)$ corresponding to the stages $[i,\mathrm{l}]$ and $[i,\mathrm{r}]$ are computed independently for all $i = 1,2,\ldots,\lfloor \log m \rfloor$. For each i, this computation is performed by one of $\lfloor \log m \rfloor$ disjoint $2\lceil d/4 \rceil$-dimensional subcubes. The computation can be visualized by a two-dimensional array of processors, with each row and column placed in a $\lceil d/4 \rceil$-dimensional subcube. Every column corresponds to a path from the root of T'' to a leaf and every row corresponds to an internal node of T''. A processor is called *active* if its row corresponds to a node on the path corresponding to its column. The essential steps in the computation of the strings $p(v)$ are performed by the active processors. The other processors are only used for supporting prefix computations and broadcast operations along the rows and the columns.

To initialize the computation of the arrays we have to assign the internal nodes and the leaves of T'' to the rows and columns and we have to identify the active processors.

The leaves are assigned to the columns in in-order, and the internal nodes are assigned to the rows in breadth-first-search-order which can be obtained by a stable sorting of the nodes of T'' according to their depth. To identify the active processors, each internal node v computes the set of its descendants in T''. In the in-order representation of T'', these are the nodes in the maximal intervals to the left and to the right of v with a depth greater than v. Some prefix operations are sufficient to find the intervals.

The i-th array computes the nodes of $T_{[i,\mathtt{l}]}$ using Fact (a) above and simulates two stages of the contraction. For $x = [i, \mathtt{l}]$ and $x = [i, \mathtt{r}]$ the behavior of each node v of T_x determines $j \in \{1, 2, 3\}$ so that v is a j-th child in $\tilde{T}$ on the level corresponding to stage x. This j is broadcast to those nodes u of T'' which are descendants of v in $\tilde{T}$ since it is part of their string $p(u)$. Fact (b) tells how to find these descendants. First, v's parent broadcasts j along the path towards the root of T'' until reaching a node of T_x. Then, the nodes reached on this segment broadcast j towards the leaves of T'', again stopping the broadcast operation at nodes of T_x.

From $p(u)$ the destination of node u in the embedding of $\tilde{T}$ is easily computed. After routing the nodes of T'' to their destinations each stage of the contraction is simulated in constant time. Thus, the three phases of our hypercube contraction algorithm for trees of size p consist of a logarithmic number of parallel steps and a recursive call in $\left\lfloor \frac{3}{4}d \right\rfloor$-dimensional subcubes. Considering the remark made at the beginning of this section we obtain our main result:

Theorem 1 *A tree of size n can be contracted on a hypercube with p processors in $O(\left\lceil \frac{n}{p} \right\rceil \log p)$ steps.*

The running time of this algorithm can be improved by a constant factor using the fact that no s-node of some stage $[i, \mathtt{l}]$ can be an l-node in the following stage $[i, \mathtt{r}]$. Hence, the four address bits corresponding to two consecutive stages $[i, \mathtt{l}]$ and $[i, \mathtt{r}]$ can never be 0111 or 1111. The remaining 7 possibilities can be encoded into 3 bits, and the algorithm can be modified making the recursive call for $\lceil d/3 \rceil$-dimensional subcubes.

The primitive operations used in the algorithm and the final simulation of the PRAM tree contraction algorithm can all be performed on the shuffle exchange network in logarithmic time. Combined with a general simulation result in [Sch90], we have:

Corollary 1 *A tree of size n can be contracted on any p processor hypercubic network in $O(\left\lceil \frac{n}{p} \right\rceil \log p)$ steps.*

The results of Theorem 1 and Corollary 1 can be shown to be asymptotically optimal for any input size if we assume that the only operations allowed on the nodes of the tree is copying, routing and contracting two nodes. In this case, the number of messages required to contract a tree T with each node u of T stored in processor $p(u)$ of a network G has to be at least $\frac{1}{2}\sum_{1 \le i < k} \mathrm{dist}_G(p(u_i), p(u_{i+1}))$, where $u_1, \ldots, u_k$ is a simple path in T and $\mathrm{dist}_G(p(u_i), p(u_{i+1}))$ is the distance between the processors $p(u_i)$ and $p(u_{i+1})$ in G.

It is easy to construct a tree with n nodes where this lower bound is $\Omega(n \log p)$ assuming a balanced distribution of the in-order sequence of nodes in a p processor hypercube.

We would like to mention that the technique of precomputing the communication structure of the PRAM tree contraction algorithm, rearranging the nodes of the given tree and embedding the corresponding tree $\tilde{T}$ can also be used to construct $\mathcal{NC}^1$ circuits for the evaluation of expressions over finite domains. Although this result has already been obtained in [MP92] we briefly sketch our construction as an alternative to the rather involved proof given there.

Similar to the circuit given in [MP92] our construction consists of three stages. To contract a tree T of size m the third stage of the circuit is a ternary tree $\tilde{C}$ of depth $2\lfloor\log m\rfloor$ capable of performing the contraction of T according to an embedding of the corresponding tree $\tilde{T}$ in $\tilde{C}$. The first stage computes for every node v its path description $p(v) \in \{1,2,3\}^{2\lfloor\log m\rfloor}$ in $\tilde{T}$. For this computation, the above algorithm can be implemented by $\mathcal{NC}^1$ circuits for prefix and broadcast operations and sorting. In the second stage, every node v of T is routed to that input of $\tilde{C}$ corresponding to $p(v)$. This routing can be done by sorting the nodes according to $p(v)$, and by a monotone routing through a butterfly network guided by the destination addresses $p(v)$. The total size of the circuit is dominated by the size of the second stage. Thus we have (also see [MP92]):

Theorem 2 *Algebraic expressions of size m over a finite domain can be evaluated by an $\mathcal{NC}^1$ circuit of size $O(\log^2 m \cdot m^{2\log 3})$.*

Applying the transformation given in [KD90] the size of our circuit for the evaluation of algebraic expressions can be decreased to $O(\log^k m \cdot m)$ (for some constant k) maintaining logarithmic depth.

4 Dynamic Expression Evaluation

As an application, the contraction algorithm given in the previous section can be used to evaluate algebraic expressions of size n on a p processor hypercube in $O(\lceil \frac{n}{p} \rceil \log p)$ steps, provided that the operators in the expression satisfy certain closure properties [ADKP89]. Furthermore, the value of each subexpression can be computed within the same time bound: after contracting the tree, we reverse the contraction steps, starting with a single node, ending up with the original tree, and maintaining an expression tree with the values of all subexpressions already computed.

Tree contraction can be applied to the evaluation of algebraic expressions with rational numbers and the operators $+, -, \cdot, /$, but the model of computation has to be adjusted appropriately. As the numbers computed in the evaluation may grow rapidly we have to supply the network with the ability to communicate rational numbers and to perform the basic arithmetic operations on rational numbers in constant time. We call this model of computation a *rational number network*.

It can be shown that our expression evaluation algorithm on the rational number hypercube is asymptotically optimal for each input size. Thus, the maximum speed-up achievable for the expression evaluation problem on the hypercube is $O(p/\log p)$. Note that, on the EREW-PRAM, the optimal speed-up of $\Theta(p)$ can be achieved for input sizes in $\Omega(p \log p)$.

Intuitively, the lower bound for the rational number hypercube derives from the fact that the initial distribution of the operands in the hypercube can be chosen in such a way that nodes adjacent in the expression tree are stored in distant processors of the hypercube. Consider, for example, expressions of the type

$$x_1 \cdot ((x_2 \cdot \ldots ((x_{p-1} \cdot ((x_p \cdot y_p) + y_{p-1})) \ldots + y_2)) + y_1)$$

where the operands x_i and y_i are stored at maximal distance from each other. In our proof, Abelson's argument [Abe80] bounding the communication complexity from below by the rank of a certain matrix is applied simultaneously to the d partitions dividing the d-dimensional hypercube into two $d-1$-dimensional subcubes. For the constant degree hypercubic networks, the lower bound can be obtained considering a small bisector of the network.

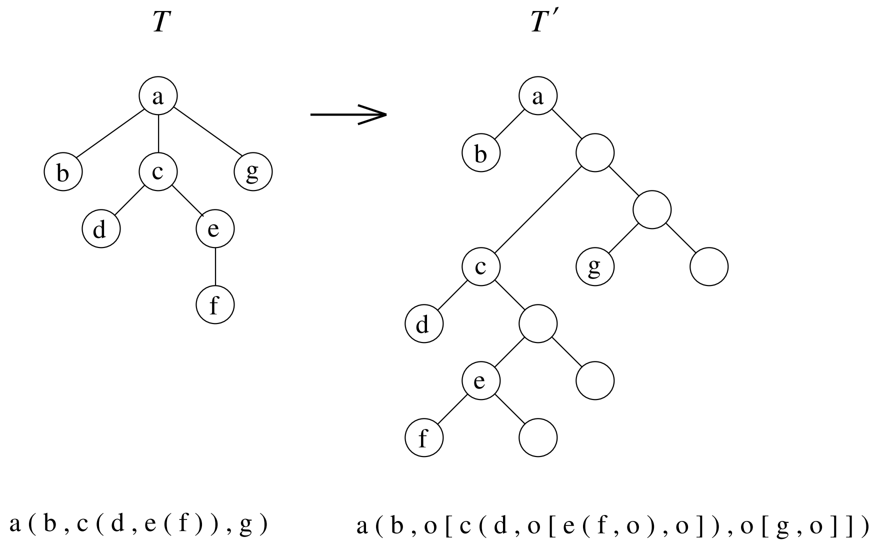

Figure 4: Transformation into a binary tree

Theorem 3 *Expressions containing n rational numbers and the operators $+, -, \cdot, /$ can be evaluated, including all subexpressions, on a rational number hypercube or hypercubic network with p processors in $O(\lceil \frac{n}{p} \rceil \log p)$ steps, and this bound is optimal.*

5 Term-Matching

Our term alignment algorithm is conceptually very similar to the tree contraction algorithm presented above. Since both algorithms use the primitive operations listed in section 2 in a similar fashion, we will keep our presentation on a high level. We freely switch between tree and linear (*i.e.,* strings with parentheses) representations of the terms, depending on which is more convenient for the discussion. In our term alignment algorithm for the hypercube, we follow the general approach of Kosaraju and Delcher [KD90] for the CREW PRAM model.

We only show how to solve the term alignment problem on a p processor hypercube in $O(\log p)$ steps for terms A and B with a total length of $n \leq p$, stored in the first n processors. For $n \geq p$ the problem is solved by simulating a hypercube of size at least n with a slowdown factor of $\Theta(n/p)$.

We identify the terms A and B with their corresponding trees. Each matching pair of parentheses in A and B corresponds to a node of T_A or T_B. In a preprocessing step, we convert T_A and T_B into binary trees T_0 and T_1 maintaining the node correspondence between the original nodes of T_A and T_B as shown in figure 4. This can easily be done by introducing k new dummy nodes $d_1, \ldots, d_k$ for each original node v with k children $v_1, \ldots, v_k$. We make d_i the right child of d_{i-1} if $i > 1$, and we make d_1 the right child of v. The node v_i becomes the left child of d_{i-1} if $i > 1$, and v_1 continues to be the left child of v. Each original path description $p_{T_A}(v) = (j_1, j_2, \ldots, j_i)$ is transformed into $p_{T_0}(v) = (2^{j_1-1}, 1, 2^{j_2-1}, 1, \ldots, 1, 2^{j_i-1}, 1)$. Note that T_0 and T_1 are binary trees, *i.e.,* each internal node has exactly two children.

To further simplify the problem we temporarily ignore the function symbols, constants and variables in A and B. To indicate whether a node is a left or right child it is represented by a pair of parentheses of type ( ) or [ ] respectively (by default, the root is represented by a pair ( )). Using the algorithmic techniques described in [MW92] to transform algebraic expressions between pre-, in-, and post-order the transformation of A and B into the representations of T_0 and T_1 by parentheses of two different types can be perfomed in logarithmic time.

To compute all pairs of corresponding nodes for two trees T_0 and T_1 (which *is* the *term alignment problem*) we use a divide-and-conquer approach. First we eliminate some of the nodes of T_0 and T_1 which are known to have no corresponding node in the other tree. The remaining trees $\hat{T}_0$ and $\hat{T}_1$ are partitioned into binary trees $T_0^{(1)}, T_0^{(2)}, \ldots, T_0^{(r)}$ and $T_1^{(1)}, T_1^{(2)}, \ldots, T_1^{(r)}$ so that the root of each $T_0^{(i)}$ corresponds to the root of $T_1^{(i)}$. The algorithm is called recursively for all pairs $\left(T_0^{(1)}, T_1^{(1)}\right), \ldots, \left(T_0^{(r)}, T_1^{(r)}\right)$. After the recursive call each node of T_0 knows the address of its corresponding node in T_1 provided that such a node exists. In the conquer-step we finally reverse the routing of the divide-step and reconstruct T_0 and T_1.

In the divide-step we are aiming at a bound of $O(p^{2/3})$ on the size of each $T_0^{(i)}$ and $T_1^{(i)}$. The running time of the divide-step and the corresponding conquer-step will be $O(\log p)$. By the result of [MW93] appropriately sized subcubes can be allocated to the generated subproblems so that the total runing time of the divide-and-conquer algorithm is $O(\log p)$.

The divide-step is performed in 3 phases:

1. After numbering the leaves of T_0 and T_1 from left to right determine the sets V_0 and V_1 of those leaves numbered by multiples of $\lfloor p^{2/3} \rfloor$. Determine W_0, the set of lowest common ancestors of V_0 in T_0 and W_1, the set of lowest common ancestors of V_1 in T_1.

2. Compute M_0, the set of nodes from W_0 corresponding to an inner node of T_1 and N_1, the set of nodes in T_1 corresponding to the nodes of M_0. Analogously compute M_1 and N_0. Let C_0 be the nodes of $M_0 \cup N_0$ and their lowest common ancestors in T_0. Analogously define C_1.

3. In T_0 delete all descendants of nodes in $W_0 \setminus M_0$. Let $\hat{T}_0$ be the remaining tree. Divide $\hat{T}_0$ into the subtrees $T_0^{(1)}, T_0^{(2)}, \ldots, T_0^{(r)}$ ($r = 2|C_0| + 1$) by cutting for each $v \in C_0$ the edges between v and its children.

   Analogously compute the trees $T_1^{(1)}, T_1^{(2)}, \ldots, T_1^{(r)}$. Note that the nodes of C_0 correspond exactly to the nodes of C_1. Arrange the separated subtrees in corresponding pairs $\left(T_0^{(1)}, T_1^{(1)}\right), \ldots, \left(T_0^{(r)}, T_1^{(r)}\right)$ for which the problem is solved recursively.

The set of nodes of a $T_0^{(i)}$ satisfies one of the following conditions: (1) the root of $T_0^{(i)}$ is just the root of T_0 *or* (2) $T_0^{(i)}$ contains all descendants in T_0 of a child u of some $v \in C_0$ *or* (3) there is a child u of some $v \in C_0$ and a $v' \in C_0 \cup W_0$ so that $T_0^{(i)}$ contains all descendants of u in T_0 except for the descendants of v'. In either case it can be shown that $T_0^{(i)}$ is made up of at most $4\lfloor p^{2/3} \rfloor - 3$ nodes. Obviously the same condition holds for each $T_1^{(i)}$.

We outline the hypercube implementation of the three phases described above. Phase 1 is based on the fact that a node is a lowest common ancestor of V_0 iff both of its children are ancestors of nodes in V_0. Some prefix-operations and parentheses-structured routings are sufficient to identify those parentheses representing the nodes of V_0 and W_0.

For phase 3 we note that each $T_0^{(i)}$ is represented by one or two intervals of parentheses. In case there are two intervals these intervals have to be routed together. First we

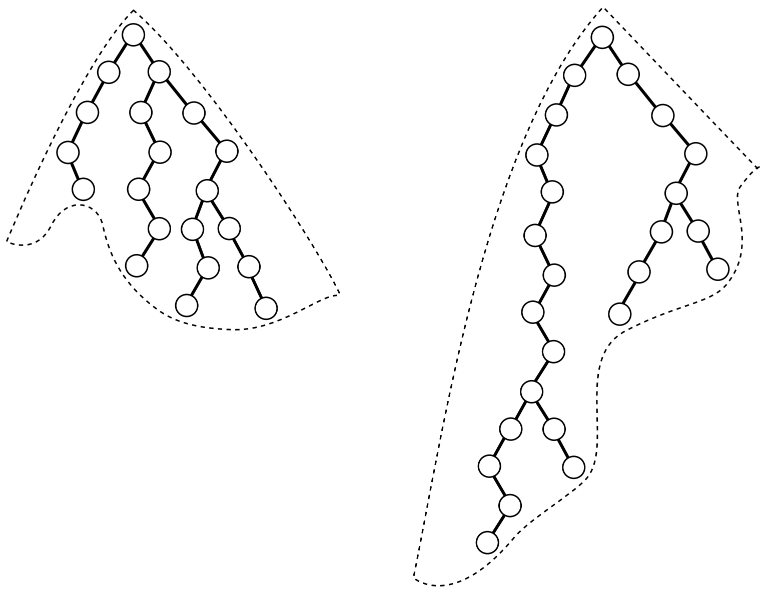

Figure 5: The trees S_0 and S_1

create empty space for the second interval right after the first interval. By a parentheses-structured routing each second interval is routed into the corresponding empty space. Since each interval arrives in reversed order, it has to be reversed again using a bit-complement-permutation routing in a suitable subcube.

In phase 2 we have to compute the correspondences for all nodes of W_0 in T_1, which is is the most complicated part of the divide-step. In fact, we compute the corresponding nodes for all ancestors of V_0. Our description of phase 2 is given in terms of trees, subtrees and nodes. The algorithm can be implemented in a straightforward way by a constant number of prefix operations, applications of sparse enumeration sort, monotone routings and parentheses-structured routings on the sequence of parentheses representing T_0 and T_1.

1. Let K_0 be the set of ancestors of V_0 including V_0, and let K_1 be the set of ancestors of V_1 including V_1. Let H be the set of heights of the nodes of $W_0 \cup V_0$ in T_0 and the nodes of $W_1 \cup V_1$ in T_1. Let K_0' and K_1' be those nodes of K_0 and K_1 with a height in H. Let S_0, S_1, S_0', and S_1' be the trees consisting of the nodes in K_0, K_1, K_0', and K_1' respectively. Each edge in S_0' or S_1' represents a path in T_0 or T_1 (see figures 5 and 6).

2. We compute all pairs of correponding nodes in S_0' and S_1': This is done by comparing for each pair $(v, w) \in V_0 \times V_1$ their path-descriptions $p_{S'_0}(v)$ and $p_{S'_1}(w)$. Every common prefix of $p_{S'_0}(v)$ and $p_{S'_1}(w)$ corresponds to a pair of correponding nodes. Note that the cardinality of V_0 and V_1 and the length of each $p_{S'_0}(v)$ and $p_{S'_1}(w)$ is bounded by $O(p^{1/3})$.

3. We compute all pairs of correponding nodes in S_0 and S_1: For an edge (v, w) of S_0' or S_1' let $P(v, w)$ denote the corresponding path in S_0 or S_1. For every edge (v, w) of S_0' corresponding to an edge (v', w') of S_1' we compare the descriptions of the paths $P(v, w)$ and $P(v', w')$. The outcomes of these comparisons can be used to repeat step 2 in a refined manner and to finally obtain the correspondences between S_0 and S_1.

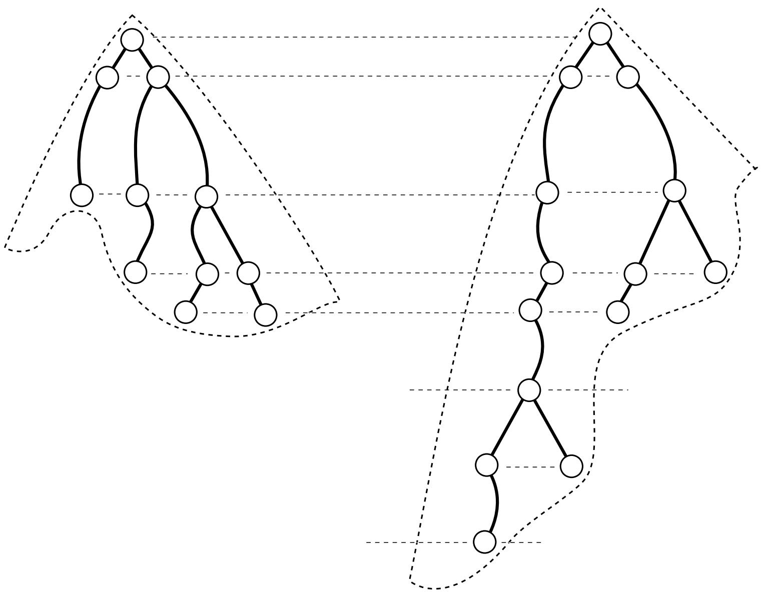

Figure 6: The trees S'_0 and S'_1

4. Now for all the nodes of K_0 the corresponding nodes in T_1 are computed provided that they are elements of K_1. Let v be a node of K_0 not corresponding to a node of K_1 whose parent w corresponds to a node $w' \in K_1$. If w' is a leaf of T_1, the node v and all its descendants don't correspond to any node of T_1 and they can be discarded from any further considerations. Otherwise v corresponds to that child v' of w' which is not in K_1. Let $X_1 \subseteq K_0$ be the set of those v considered in the second case, *i.e.,* the nodes v corresponding to a node $v' \notin K_1$ which is a sibling of a node in K_1. For $v \in X_1$ let $T_0^{(v)}$ be the subtree of T_0 rooted at v and let $T_1^{(v)}$ be the subtree of T_1 rooted at the corresponding v'.

5. It remains to compute the correspondences for those nodes of K_0 which are descendants of a node $v \in X_1$ by considering all pairs $\left(T_0^{(v)}, T_1^{(v)}\right)$ for $v \in X_1$. Note that all subtrees $T_0^{(v)}$ and $T_1^{(v)}$ for $v \in X_1$ are disjoint and that the size of each $T_1^{(v)}$ is bounded by $2\left\lfloor p^{2/3}\right\rfloor$ whereas a $T_0^{(v)}$ may still contain up to $O(p)$ nodes.

   Repeating steps 1 – 4 in an analogous manner for the pairs $\left(T_0^{(v)}, T_1^{(v)}\right)$ we obtain a set $X_2 \subset K_0$ so that the correspondences of the ancestors of X_2 are computed and for each $w \in X_2$ there is a pair of corresponding subtrees $\left(T_0^{(w)}, T_1^{(w)}\right)$ with $|T_1^{(w)}| = O(p^{1/3})$.

   Repeating steps 1 – 4 once more in an analogous manner for the pairs $\left(T_0^{(w)}, T_1^{(w)}\right)$ for all $w \in X_2$, we finally compute the correspondences for all nodes of K_0.

If n, the number of nodes of both trees, is greater than p, the term-alignment algorithm for a $\lceil \log n \rceil$-dimensional hypercube has to be simulated by the p processor hypercube with a slowdown of $O(n/p)$. If n is bounded by a polynomial in p, the resulting running time is $O(\left\lceil \frac{n}{p} \right\rceil \log n) = O(\left\lceil \frac{n}{p} \right\rceil \log p)$.

Theorem 4 *The term alignment problem for terms of size n can be solved on a p processor hypercube or hypercubic network in $O(\left\lceil \frac{n}{p} \right\rceil \log p)$ steps.*

As noted in the introduction the term matching problem can be solved by a term alignment followed by a standard sorting algorithm. Thus, we have

Corollary 2 *The term matching problem for terms of size n can be solved on a p processor hypercube with a worst case running time of $O(\left\lceil \frac{n}{p} \right\rceil \log p (\log\log p)^2)$ steps or, using a randomized sorting algorithm, with an expected running time of $O(\left\lceil \frac{n}{p} \right\rceil \log p)$ steps.*

6 Conclusion and Open Problems

Using a recursive approach, we have shown how to contract trees efficiently on hypercubes and related networks. Our technique of precomputing the communication structure of a PRAM contraction algorithm, rearranging the data and simulating the algorithm without any further overhead can be used for the contraction of small trees in networks. It could also be applied for the construction of NC^1 circuits for the expression evaluation problem which is simpler than the construction proposed in [MP92].

Due to the recursive call and the usage of the parentheses structured routing algorithm, the constant factor in the worst case running time of our contraction algorithm is quite large. But the algorithm performs much better for random trees. As noted in [PS91], with high probability most of the nodes are contained in local subtrees and will be eliminated in the recursive call or even in the reduction within a single processor.

For rational number hypercubes and hypercubic networks, the maximal speed-up achievable for the expression evaluation problem is $O(p/\log p)$. We conjecture that a similar result holds for all constant degree networks, but we have so far been unable to apply the known lower bound results for communication complexity to this case.

References

[Abe80] H. Abelson. Lower bounds on information transfer in distributed computations. *Journal of the ACM*, 27:384–392, 1980.

[ADKP89] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[ALMN90] B. Aiello, F.T. Leighton, B. Maggs, and M. Newman. Fast algorithms for bit-serial routing on a hypercube. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 55–64, 1990.

[Bre74] R. Brent. The parallel evaluation of general arithmetical expressions. *Journal of the ACM*, 21:201–206, 1974.

[CP90] R. Cypher and C.G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 193–203, 1990.

[DKM84] C. Dwork, P. Kanellakis and J.C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[GMT88] H. Gazit, G.L. Miller, and S.-H. Teng. Optimal tree contraction in the EREW model. In Stuart K. Tewksbury, Bradley W. Dickinson, and Stuart C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture, and Technology*, pages 139–156. Plenum Press: New York-London, 1988.

[GR86] A. Gibbons and W. Rytter. An optimal parallel algorithms for dynamic expression evaluation and its applications. In *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS-241*, pages 453–469, 1986.

[HM93] V. Heun and E.W. Mayr. A new efficient algorithm for embedding an arbitrary binary tree into its optimal hypercube. Technical Report I9321, Institut für Informatik, TU-München, 1993.

[KD88] S.R. Kosaraju and A.L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In *Proceedings of the 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures, LNCS-319*, pages 101–110, 1988.

[KD90] S.R. Kosaraju and A.L. Delcher. A tree-partitioning technique with applications to expression evaluation and term matching. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 163–172, 1990.

[KP92] Z.M. Kedem and K.V. Palem. Optimal parallel algorithms for forest and term matching. *Theoretical Computer Science*, 93:245–264, 1992.

[Lei92] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann Publishers, 1992.

[MP92] D.E. Muller and F.P. Preparata. Parallel restructuring and evaluation of expressions. *Journal of Computer and System Sciences*, 44:43–62, 1992.

[MR85] G. Miller and J. Reif. Parallel tree contraction and its application. In *Proceedings of the 26st Annual Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[MW92] E.W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 109–117, 1992.

[MW93] E.W. Mayr and R. Werchner. Divide-and-conquer algorithms on the hypercube. In *10th Annual Symposium on Theoretical Aspects of Computer Science,LNCS-665*, pages 153–162, 1993.

[NS81] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30:101–107, 1981.

[NS82] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *JACM*, 29:642–667, 1982.

[PS91] G. Pietsch and E. Schömer. Optimal parallel recognition of bracket languages on hypercubes. In *8th Annual Symposium on Theoretical Aspects of Computer Science, LNCS-480*, pages 434–443, 1991.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Sch80] J.T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2:484–521, 1980.

[Sch90] E.J. Schwabe. On the computational equivalence of hypercube-derived networks. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 388–397, 1990.

[VB81] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, 1981.