(CS838: Topics in parallel computing, CS1221, Tue, May 4, 1999, 8:00-9:15 a.m., Pavel Tvrdik)

Section #26: Parallel tree contraction

26.1 Rake and Compress

In this lecture, we describe basic parallel techniques for tree contraction. Given a rooted tree, with labeled nodes and edges, the task is to reduce it to one single node. This problem appears in many areas, but the most important case is ***expression evaluation***, where the tree represents an expression, every node is an operation or function, and subtrees are the argument expressions. One possible parallel approach to tree contraction is ***D&C***, but it requires complicated algorithms for splitting arbitrary trees into subtrees of equal size and it is a difficult problem.

Much easier is to contract a tree ***bottom-up***. Processors are assigned leaves of the tree and perform local modifications of the tree by removing leaves. This operation is called **Rake** (see Figure 26.1(a)). But **Rake** itself is not sufficient for parallel tree contraction, for two reasons. 1) If the tree is thin and tall, therefore the height of the tree is linear rather than logarithmic (the worst case is just a linked list), then **Rake** cannot be applied in parallel. 2) **Rake** itself tends to linearize the original tree.

The complementary operation to **Rake** is **Compress** and it is based on ***pointer jumping*** (see Figure 26.1(b)). Ideally, **Compress** and **Rake** can be applied in parallel to disjoint parts of a tree. **Compress** produces leaves for **Rake** and **Rake** produces linear lists for **Compress**.

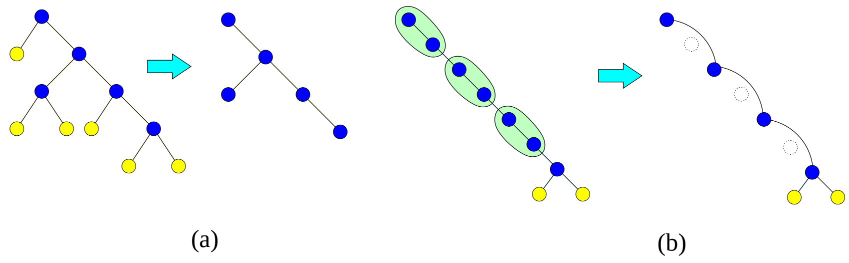

Figure 26.1. Two basic operations for contracting trees. (a) **Rake**. (b) **Compress**.

26.2 Basic tree contraction CREW PRAM algorithm

Let us describe a ***generic*** parallel contraction algorithm, called **Basic Contract**, on an arbitrary tree $T = (V, E)$ with $|V| = n$. Assume the following data structures:

Input:	$P[1, \ldots, n]$; /* $P[x]$ is a pointer to the parent of x */
	$children[1, \ldots, n]$; /* $children[v] = \{v_1, \ldots, v_k\}$ – pointers to all children */
	$index[1, \ldots, n]$; /* $index[v_i] = i$ – each child v knows its index in $children[P[v]]$ */
Auxil:	$label[1, \ldots, n]$; /* $label[v] = \{f_1, \ldots, f_k\}$, where $f_i \in \{U, M\}$ */
	/* $f_i = M$ = marked iff a child supplied its value to its parent */
	$UnMarkChil(x)$ **returns** `int`; /* function returning the # of unmarked children */
Output:	the value accumulated in the root

```
/* initialize the data structures */
for all nodes $v \in T$ do_in_parallel $initialize(v)$;
while $UnMarkChil(root) > 0$ do
    { for all nodes $v \in T$ do_in_parallel
        if $P[v] \neq$ nil then
            { case $UnMarkChil[v]$ of
                0: { Rake($v$); $label[P[v]][index[v]] := M$; $P[v] :=$ nil; }
                1: if $UnMarkChil[P[v]] = 1$ then { Compress($v$); $P[v] := P[P[v]]$; }
            endcase }};
Rake($root$);
```

Theorem 1 *After $O(\log_{4/3} n)$ applications of* **Basic Contract** *to n-node tree T, it is reduced to a root. If* **Rake** *and* **Compress** *take $O(1)$ time, then the parallel time with $p = \Theta(n)$ is $O(\log n)$.*

26.3 Binary expression tree evaluation

This generic approach can be applied to arbitrary tree. We will show a specific implementation of this algorithm for parallel evaluation of binary expressions. If T represents an arithmetic expression, then internal nodes represent operators and leaves contain constant input values. For simplicity, we will consider only expressions with $+$ and $\cdot$ and integer constant values in leaves (see Figure 26.2(a)).

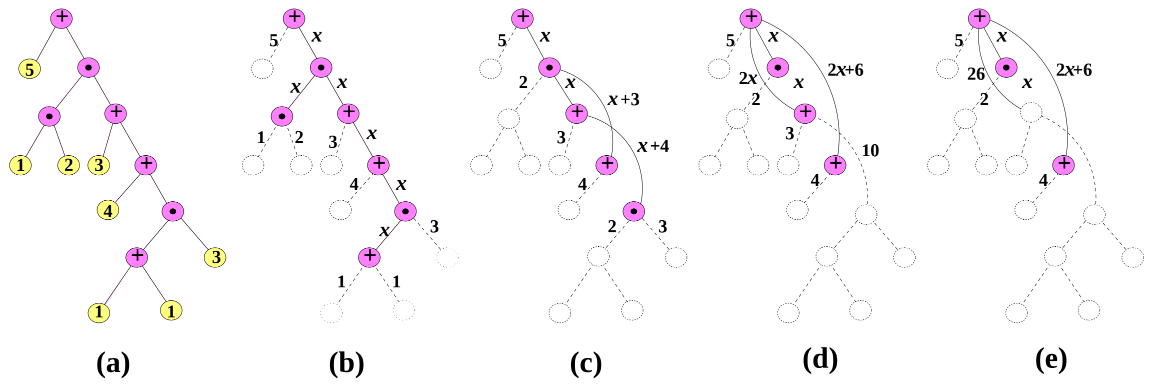

Figure 26.2. An example of the **Basic Contract** evaluation of an expression tree. (a) The expression tree. (b) Initialization and the effect of the first parallel **Rake**. (c)-(e) Three parallel steps ending up with one essential and two non-essential chains.

The implementation of expression tree evaluation needs the following data structures: Every edge $\langle v, P[v]\rangle$ is labeled with a linear function $f_v(x) = a_v x + b_v$, where a_v, b_v are integer constants. Initially $f_v(x) = x$ for all $v \in T$, $v \neq root$. Every internal node v represents an operator $op[v]$. Every leaf v has value $val[v]$. The **Rake** of a leaf v means placing $f_v(val[v])$ on the edge $\langle v, P[v]\rangle$. If an internal node v with operator $\odot$ knows values of both left son l and right son r, then its subtree can be replaced by a leaf with constant value $val[v] = f_l(val[l]) \odot f_r(val[r])$ computed by the **Rake** operation, where f_l and f_r are function labels of edges $\langle l, v\rangle$ and $\langle r, v\rangle$, respectively.

An internal node that is a part of a chain of nodes with one evaluated argument can be compressed. Figure 26.3 shows a general case. Consider an internal node v with $op[v]$, whose right son r supplied its constant value $val[r]$ by placing $c_r = f_r(val[r]) = a_r val[r] + b_r$ on edge $\langle r, v\rangle$. Assume that the left child l does not know its value (i.e., its subtree has not been evaluated yet). Then node v can be compressed, i.e., jumped over, if its contribution is properly incorporated into the modified tree. Hence if l sets $P[l] := P[v]$, then the new edge must be labeled by a linear function that combines functions attached to the original edges $\langle l, v\rangle$ and $\langle v, P[v]\rangle$ plus the value c_r. If this new linear function is denoted $f'_l(x) = a'_l x + b'_l$, then $f'_l(x) = a_v((a_l x + b_l) op[v] c_r) + b_v$, which after simplification gives coefficients a'_l and b'_l.

The code of Basic Contract is modified now as follows:

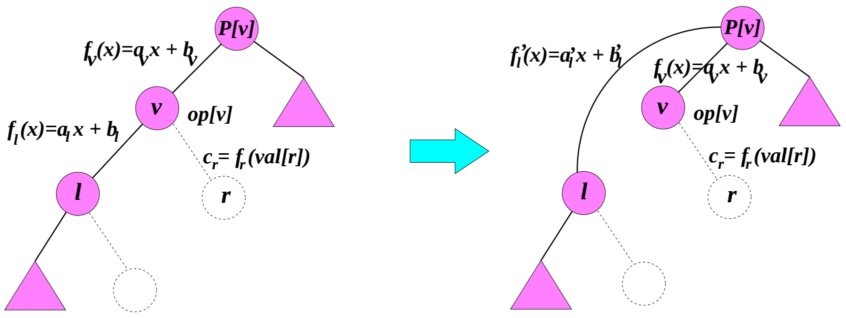

Figure 26.3. The effect of operation **Compress**(l) in a binary expression tree.

Input:	$P[1,\dots,n]$; /* $P[x]$ is a pointer to the parent of x */
	$val[1,\dots,n]$; /* $val[v]$ – value in v after its subtree is evaluated */
	$op[1,\dots,n]$; /* $op[v]$ is the operator of an internal node v */
	$side[1,\dots,n]$; /* $side[v] \in \{L,R\}$ */
Auxil:	$(a,b)[1,\dots,n]$; /* $a[v]$ and $b[v]$ are labels of edge $\langle v, P[v]\rangle$ */
	$contr[1,\dots,n][L,R]$ /* auxiliary array to store contributions from children */
	$UnMarkChil(x)$ **returns** `int`; /* function returning the # of unmarked children */
Output:	the value of the expression tree stored in the root

```
/* initialize the data structures */
for all nodes $v \in T$ do_in_parallel /* $initialize(v)$ */
        if $UnMarkChil(v) = 0$ /* leaves */
            then {$contr[P[v]][side[v]] := val[v]$; $P[v] :=$ nil; }
            else $(a,b)[v] := (1,0)$; /* internal nodes */
while $UnMarkChil(root) > 0$ do
    { for all nodes $v \in T$ do_in_parallel
            if $P[v] \neq$ nil then
                { case $UnMarkChil[v]$ of
                    0: { $val[v] := eval(op[v], contr[v][L], contr[v][R])$; /* Rake($v$) */
                          $contr[P[v]][side[v]] := a[v]val[v] + b[v]$; $P[v] :=$ nil; } /* Mark */
                    1: if $UnMarkChil[P[v]] = 1$ then /* Compress($v$) */
                        { $(a,b)[v] := simplify((a,b)[v], (a,b)[P[v]], op[P[v]], contr[P[v]][side\_sibl[v]])$;
                          $P[v] := P[P[v]]$; }
                endcase }};
$val[root] := eval(op[root], contr[root][L], contr[root][R])$;
```

26.4 Work-optimal EREW PRAM tree contraction

26.4.1 Discussion of the Basic Contract approach

The previous algorithm has two drawbacks:

▷ ***It is not work-optimal***. The number of operations it requires is $n \log n$, in contrast to $O(n)$ operations required in sequential evaluation. The reason behind this inefficiency is that whenever a **Compress** is performed, we get two ***chains*** but at most of them is ***essential***, needed for evaluation of the accumulated value in the root. The other chain is nonessential for the total value, but even though, processors keep contracting it and spending operations on that. Hence if T is a linear list, we get the same problem as with list ranking algorithm based on pointer jumping.

▷ ***It requires CREW PRAM***. A ***head*** of a chain, i.e., a node with 2 unevaluated children among which one is the last node of a linked list of nodes that have one child evaluated, becomes, thanks to compressing, the parent of all the nodes from the linked list (see Figure 26.4). The head cannot be jumped over until its second child submits its value (i.e., is marked). Until then, all these waiting nodes will attempt to read the counter $UnMarkChil[P[v]]$ and once this counter is set to 1, they all will read $P[P[v]]$.

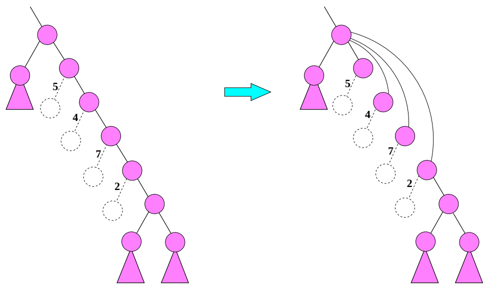

Figure 26.4. Basic Contract requires Concurrent-Read PRAM.

26.4.2 Shunt operation

Both difficulties can be overcome if the contraction will not produce linear chains at all. A chain is produced when the tree contains a binary subtree, where each internal vertex has one child that is a leaf and one that is not. After a parallel **Rake** operation on all leaves, such a subtree becomes a linear list. If **Rake** and **Compress** is always applied simultaneously, i.e., if every individual **Rake** operation is followed immediately by a **Compress** of its sibling, chains cannot be formed. This combined operation is called **Shunt**. Figure 26.5 illustrates its effect. Note that **Shunt** is applied to a leaf node and as a side effect, its parent is compressed, disconnected, and therefore disallowed to form non-essential chains in the future. Due to this side effect, **Shunt** must be applied carefully. There are several situations in which parallel execution of **Shunt** operations could cause problems.

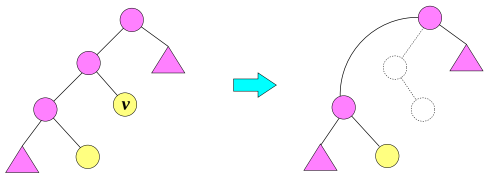

Figure 26.5. **Shunt**(v) combines the effect of **Rake**(v) and **Compress**$(sibling[v])$.

- ▷ ***Shunt** is not defined for children of the root*. That's because **Compress** cannot be applied to the root.
- ▷ ***Shunt** cannot be performed on two siblings simultaneously*. Not only would it require Concurrent-Read PRAM, but it could lead to a nondeterministic result due to parallel racing. Both leaves should be raked (i.e., disconnected) and compressed (i.e., jump over the parent) at the same time, and so the final status is not well defined. Figure 26.6 shows the situation.

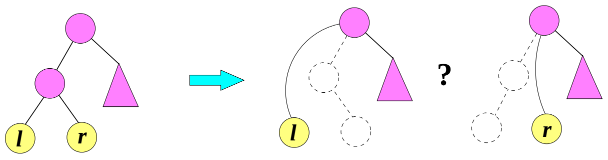

Figure 26.6. **Shunt** applied to two siblings in parallel is nondeterministic.

- ▷ ***Shunt** cannot be performed on two adjacent leaves in left-to-right ordering.* This time, we would disconnect the tree, see Figure 26.7. **Shunt**$(i+1)$ will set up $P[d] = b$, but at the same time, **Shunt**(i) will disconnect b due to compressing c. An obvious way to prevent both situation is to apply **Shunt** to ***odd-numbered leaves*** first and to ***even-numbered leaves*** then. But even this measure is not sufficient.
- ▷ ***Shunt** cannot be applied to consecutive left and right odd-numbered leaves.* See Figure 26.8. This situation not only disconnects the tree (**Shunt**(o) disconnects b due to compressing

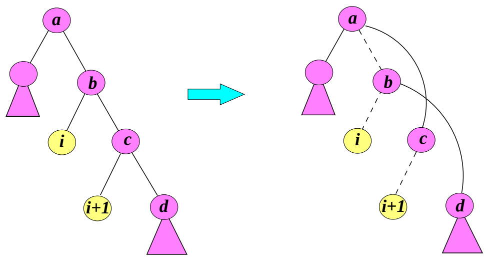

Figure 26.7. **Shunt** applied to two consecutive leaves disconnects the tree.

c, while **Shunt**$(o+2)$ sticks e to b), but it is nondeterministic again. **Shunt**$(o+2)$ performs $P[c] := \texttt{nil}$, but **Shunt**(o) performs $P[c] := P[b]$ before disconnecting b.

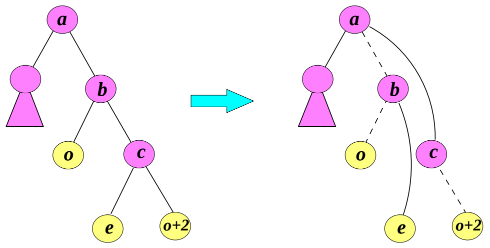

Figure 26.8. **Shunt** applied to two consecutive odd leaves disconnects the tree and requires Concurrent Write.

It is easy to see that to implement this kind of contraction, we need to order leaves of tree. This is equivalent to numbering the leaves in depth-first order. Hence, it is a similar problem as ***pre-order numbering*** in the previous section, except that the internal nodes do not count. The leaf numbering starts from 0. Let us call this procedure **LR_numbering**.

```
Input: $EA'[1,\ldots,m]$;
Auxiliary: $IsLeaf[1,\ldots,n]$; /* flags identifying leaves */
Output: $LR\_Numbering[1,\ldots,n]$;
---
for all nodes $v \in T$ do_in_parallel
        $IsLeaf[v] := 0$;
for all arcs $xy \in EA'$ do_in_parallel
        if $rank[xy] = rank[yx] + 1$
                then $\{Weight[xy] := 1; IsLeaf[y] := 1\}$ else $Weight[xy] := 0$;
apply Parallel Scan on $Weight[1,\ldots,m]$;
for all arcs $xy \in EA'$ do_in_parallel
        if $rank[xy] = rank[yx] + 1$
                then $LR\_Numbering[y] := Weight[xy] - 1$ else $LR\_Numbering[y] := 0$
```

Even though generalizable to arbitrary tree, we describe here the generic **Shunt Contract** method for binary trees.

```
Input:    $EA'[1,\ldots,m]$; /* Euler array */
          $P[1,\ldots,n]$; /* $P[x]$ is a pointer to the parent of $x$ */
          $side[1,\ldots,n]$; /* $side[v] \in \{L,R\}$ */
          $sibling[1,\ldots,n]$; /* $sibling[v]$ points to the other child of $P[v]$ */
Auxil:    $IsLeaf[1,\ldots,n]$; /* flags identifying leaves */
          $active[1,\ldots,n]$; /* flags keeping track of nodes still in game */
          $LR\_numbering[1,\ldots,n]$; /* Left-to-right numbering of leaves +/
Output:   the value of the reduced tree stored in the root
```

```
Procedure Shunt($v : node$);
   { Rake($v$); $active[v] := 0$; $active[P[v]] := 0$;
     Compress($sibling[v]$); $P[sibling[v]] := P[P[v]]$; }
```

```
/* initialize the data structures */
for all nodes $v \in T$ do_in_parallel /* $initialize(v)$; */
call LR_numbering($T$);
for all nodes $v \in T$ do_in_parallel
    if $IsLeaf[v]$ then $active[v] := 1$ else $active[v] := 0$;
repeat $\log n$ times
    for all nodes $v \in T$ do_in_parallel
        if ($v \neq root$ and $active[v]$)
            if ($Is\_Odd(LR\_Numbering[v])$ and $P[v] \neq root$)
                then { if ($side[v] = L$) then Shunt($v$);
                           if ($side[v] = R$) then Shunt($v$) };
                else $LR\_Numbering[v] := LR\_Numbering[v]/2$;
Rake($root$);
```

Theorem 2 **Shunt Contract** *runs correctly on EREW PRAM.*

Proof. Let v_1 and v_2 be two nonconsecutive ***left*** leaves of T. Then $P[v_1] \neq P[v_2]$ and $P[v_1] \neq P[P[v_2]]$. It follows from the definition of **Shunt** that no collision can appear. ■

Theorem 3 *If* $p = \Theta(n/\log n)$, *then* $T(n, p) = O(\tau_{\text{Shunt}} \log n)$.

Proof. Assign $\log n/2$ leaves to each processor. One application of **Shunt Contract** eliminates one half of current leaves, so that the total number of parallel **Shunt** operations is at most

$$\log n/2 + \log n/4 + \cdots + \log n/(2^{\log\log n}) + 1 + \cdots + 1 \leq 2\log n.$$

In case of expression trees, the effect of **Shunt** is depicted on Figure 26.9.

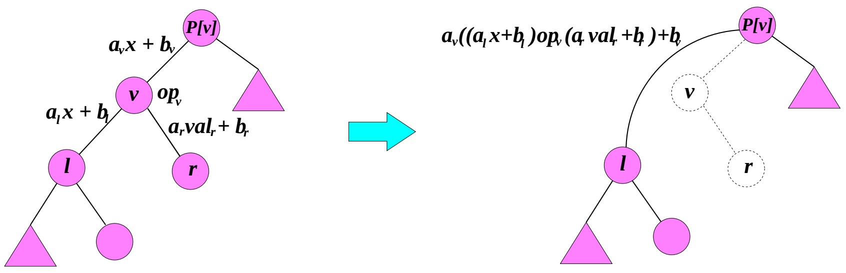

Figure 26.9. **Shunt** in an expression tree.

Figure 26.10 shows an example of applying contraction based on shunting to binary expression evaluation

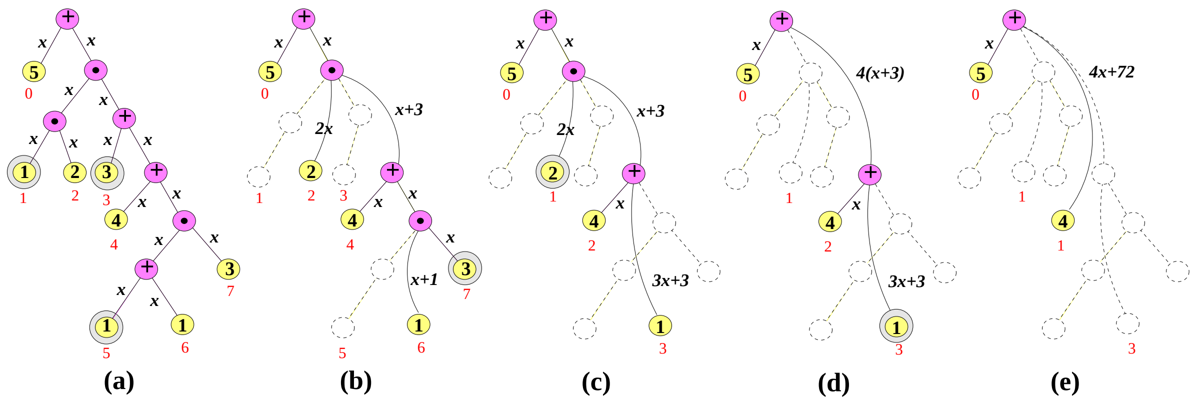

Figure 26.10. Example of contracting an expression tree by parallel **Shunt** operation.