15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Tree II **Date:** February 19, 2009
Scribe: Hyun Kyu Shim

12.1 Operation On Trees

We begin our discussion with using trees to represent recursive operations on a structured data set.

12.1.1 LEAFFIX:

An operation which writes ”sum” of all children node values recursively to every node in the tree.
(”sum” is any binary associative operation)

Figure 12.1.1: Example of LEAFFIX for addition

12.1.2 ROOTFIX:

Likewise, an operation which writes ”sum” of parent node values recursively to every node in the tree.

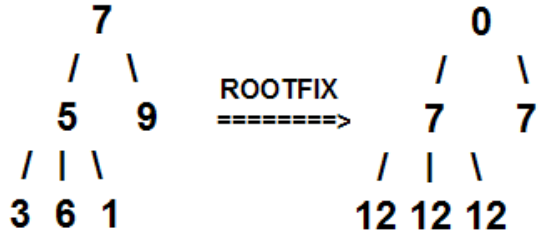

Figure 12.1.2: Example of ROOTFIX for addition

So far the example decribed above are tree operations of a single alphabet type with a singular operation. (In above example alphabet was numbers and operation was addition) However, we can extend the idea to any alphabet type with associative binary opeartion set.

12.1.3 EXPRESSION EVALUATION:

An operation to evaluation an expression tree.
Expression tree has nodes that takes on either a numeric value or a binary operator.

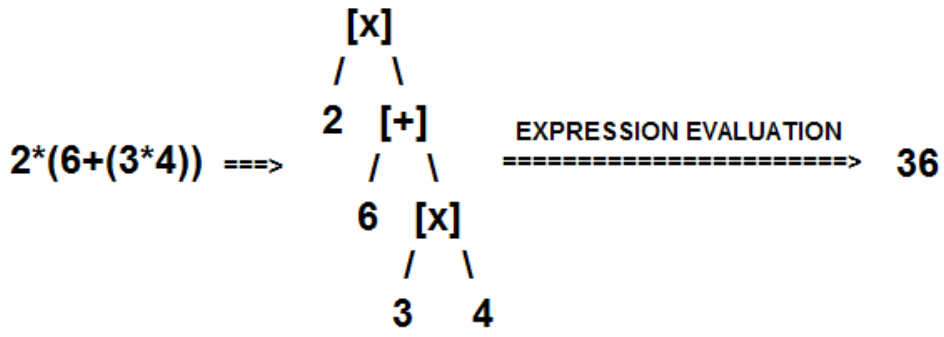

Figure 12.1.3: Example of an expression tree and evaluation

12.1.4 TREE OPERATION RUN-TIME

First we look at two extreme tree structure situations.

BALANCED TREE:
If tree is balanced, we can perform above operations in O(n) work and O(log(n)) depth. This follows from virtue of indepedent and parallel calls to children it can make.

CHAINED TREE (linked-list):
Using random mate algorithm from serveral lectures ago, we can perform above operations in O(n) work and O(log(n)). Note that operations having to be associative is important for this technique.

We are able to perform above operations efficiently for both extreme situations. However, method for balanced tree does not work for a chained tree and vice versa. So, is there a general algorithm that works for all possible trees?

Fortunately, YES!

12.2 Tree Contraction

12.2.1 Tree Structure:

For the purpose of understanding the concept, we limit our tree to be a binary tree constructed from the following 3 node types.

1)unary nodes: has one child
2)binary nodes: has two child
3)leaf nodes: has no child

We can represent trees with branching factor greater than 3 with this structure by linking unary node to binary node and attaching binary node to that unary node. See for yourself how to do this (think Red-Black Tree). Note that associativity is important.

Note that unary node is not perfectly analogous to classic binary tree structure with one node having either left or right child. This will become apparent later when we discuss functions of parameters.

To make things clear, here is an example of such tree with indicators U = unary, B = binary, and L = leaf

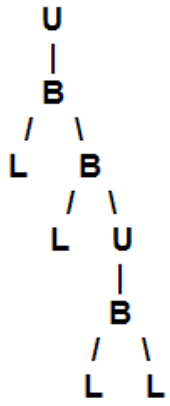

Figure 12.2.4: Example binary tree structure in discussion

Now that we understand how our tree is structured, lets consider two main steps that define the tree contraction algorithm.

12.2.2 RAKE:

Rake step joins every left leaf of binary nodes to the parent. By join, we mean that it undergoes a functional process which achieves the operation we want to make. This will be discussed formally later.
Here is an example of the rake step.

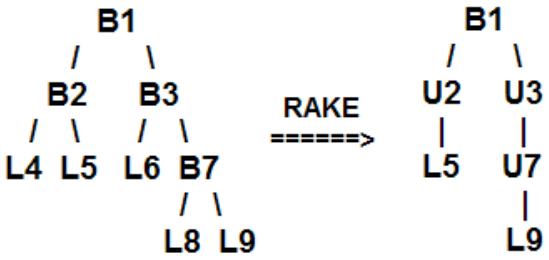

Figure 12.2.5: Example of RAKE with node type indicated by unique numbering

12.2.3 COMPRESS:

Compress step is actually a sequence of several events
1)Find an independent set of unary nodes. (Independence here is defined such that no two are neighbors meaning no parent to child relation)
2)Join each node in indepedent set with it's child
(Note that independent set is not unique)
Here is an example of compress step continuing from the above rake step.

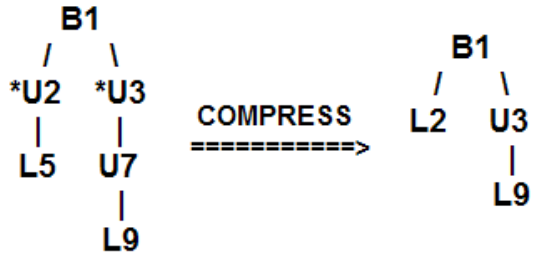

Figure 12.2.6: Example of COMPRESS with * indicating the independent set chosen

Now that we know the two main steps used in

12.2.4 Tree Contraction Algorithm

To solve any of the operations described above, we run graph contraction as follows.

Repeat until tree becomes a unary node
{
    RAKE;
    COMPRESS;
}

Depth of Graph Contraction Algorithm has depth of O(log(n)) and work of O(n) with high probability. These bounds rely on each step being carried out efficiently. Let's examine them in closer detail.

12.2.5 Finding Indepedent Set

From the instructions of the steps in Tree Contraction, one event remains to be known if it can be done efficiently. This is finding the indepedent set of unary nodes.
Claim: If we pick Indepedent Set using Random-Mate technique, then we will remove constant fraction of nodes each iteration with high probability. Here is the summary of the argument. (Recall that in Random-Mate, all candidates flip a coin and any node who flips heads and pointing to a node who flipped tails is chosen)

1) $E[|\text{INDEPEDENT SET}|] = \frac{1}{4} * |\text{ SET }|$
2) Consider each iteration to be indepedent, and use Chernoff bounds to show that it is probabilistically

unlikely that a sequence of iterations will result in some desired proportion not being chosen.
Refer to lecture on Random-Mate for details

Now that we know that each iteration contracts tree by a constant fraction we need to guarantee that there will be unary nodes available to contract on. This is why the rake step is important.

12.2.6 Rake in more detail

Previously, we assumed Rake to contract on left leaf node of binary nodes. Now, consider the following tree.

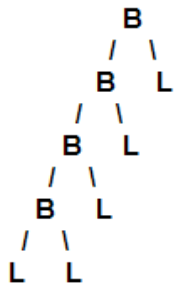

Figure 12.2.7: Example case of joining left child not optimal

It can be observed that each iteration would contract tree by one node which would lead to linear depth. However, it is obvious that choosing to join right child would be efficient.
So we need some way to alternate between joining left or right child so that we achieve O(log(n)) depth.

It turns out that alternating between left and right would be good enough. But the argument is very subtle.

1) First note that tree with N leaves $\implies$ N-1 binary nodes
2) After one rake to left and right is done, a constant fraction of leaves must have been removed. The reason is because of the linear correlation between number of leaves to binary nodes.

Formally,
$E[$ #leaves removed on one iteration $]$ = $\frac{1}{2}$ * # of leaves
use same argument as Random-mate. However, one can be clever with structual induction and get strict bound.

12.2.7 Tree Contraction Algorithm Run-Time:

From the above, we know that a contant fraction of leaf node will be contracted by Rake which turn to unary nodes. Of those unary nodes and others that preexisted, a contant fraction of them will be contracted by compress.
So Depth of this algorithm is O(log(n))
We contract graph and therefore never do redundant calculations.
So Work of this algorithm is O(n)
We obtained the bounds we desired.

12.3 Functional Definition of Join

In Tree Contraction Algorithm, we said join undergoes a specific functional process. We define that formally here and discuss them.

Each node in the tree is now a function.

12.3.1 Leaffix Additon

Lets look at slightly modified LEAFFIX addition to understand this concept more clearly. LEAFFIX describe above sums values from all children. Let's modify it so that it adds value of itself as well. Generally, the functions will be of the following form.

1)binary node = $F(X, Y)$ (X,Y are the children paramenters)
2)Unary node = $G(X)$ (X is the child parameter)
3)leaf node = I_0 (Identity function)

More specifically,

$F_a^+(X, Y)$ = a + X + Y where a is the value of the node itself and X and Y are function result of the children
$G_a^+(X)$ = a + X likewise
I_a^+ = a
(subscript a signifies value of the node, and superscript + signifies the operation of that node)

Leaffix Join in Rake
Now, lets look at how Join in Rake would work.
W.L.O.G., say we are contracting on the left leaf.
$F_a^+(I_b, Y) \implies G_{a+b}^+(Y)$
Notice that Join transforms the function in the node.

Leaffix Join in Compress
Now lets look at how Join in Compress would work.
There are three cases to consider.
i) $G_a^+(I_b^+) \implies I_{a+b}^+$
ii) $G_a^+(G_b^+(X)) \implies G_{a+b}^+(X)$
iii) $G_a^+(F_b^+(X, Y)) \implies F_{a+b}^+(X, Y)$

Because we define tree contraction formally with function composition, associativity of operations was emphasized. Now, lets consider something a bit more complicated.

12.3.2 Expression Tree Evaluation

Now, we look at how we can represent expression tree nodes in terms of functions. Initially it is not trivial since a node can either have a numeric value or binary associative operator.

Let's limit ourselves to binary operands just two operands {*,+} in the expression tree for ease of understanding.

Initially unary nodes do not exist in the expression tree. Binary nodes are the operators and it needs two

operands. All leaf nodes are all number valued nodes. However, graph contraction creates unary nodes, so we need a special function layout.

It turns out that using linear combination works. First, consider the function for binary nodes.

$F^{+}_{a,b}(X,Y) = a + b(X+Y)$
$F^{*}_{a,b}(X,Y) = a + b(X*Y)$
(Subscript a,b are scale factors, and superscript is the operation in mind. Initially, a would be 0 and b would be 1 which gives us the operation we have in mind)

Now consinder the function for unary and leaf nodes

$G_{a,b}(X) = a + b*X$
$I_a = a$ (Notice that unary function is like the binary node with operator and operand calculated. Also superscript of operator is not stated since it is not an invariant for the above operations)

Expression Tree Join in Rake
W.L.O.G.,Let's consider the contraction on the left leaf node.
$F^{+}_{a,b}(I_c, X) = a + b(c+X) \implies G_{(a+bc),b}(X)$
$F^{*}_{a,b}(I_c, X) = a + b(c*X) \implies G_{a,bc}(X)$

Expression Tree Join in Compression
There are four cases to consider.
i) $G_{a,b}(I_c) \implies I_{(a+bc)}$
ii) $G_{a,b}(G_{c,d}(X)) \equiv a + b(c+dX) \implies G_{(a+bc),bd}(X)$
iii) $G_{a,b}(F^{+}_{c,d}(X,Y)) = a + b(c+d(X+Y)) \implies F^{+}_{(a+bc),bd}(X,Y)$
iv) $G_{a,b}(F^{*}_{c,d}(X,Y)) = a + b(c+d(X*Y)) \implies F^{*}_{(a+bc),bd}(X,Y)$

Now we have defined function composition rules to be used by join in our tree contraction. All transitions can be done in constant time. So, our tree contraction algorithm bounds still hold.

12.3.3 Usage of Functions in the Algorithm

Note that in LEAFFIX, structure of the tree does not change during the operation. Each node from the operation will contain the identity function indicating its value. The differing types of functions indicated above will signify to the tree contraction algorithm of the evaluation process of the tree. One way to do this by adding extra pointers indicating its children through tree contraction, so the algorithm knows the contractions but all nodes retain its original ordering.

Note that in TREE EVALUATION, structure of the tree may change during the operation, resulting in single identity function signifying the solution at the end. We defined functions for {*,+}. Try to create functions for other binary associative operators.

12.4 Line Breaking

We'll end the lecture with a food for thought.
Consider the LINE BREAKING problem which is standard I/O problem

LINE BREAKING:
Given a long stream of input, we need to break them in such way that no lines are beyond certain number of characters, and that no words in the stream is broken. We assume that no word is longer than maximum line capacity and that words are delimited by white space.

Here is a sequential algorithm for it

```
COUNT = 0;
NUM = # of words
FOR i=1 to NUM
{
        COUNT += length(word_i) + 1; *add one for space
        IF(COUNT ≥ MaxLineLength) *out of space
        {
                StoreLineBreak(i); *indicate that a line break occurs after i th word
                COUNT = length(word_i) + 1; *rest count
        }
}
```

This is currently O(n) work and depth. Can you parallelize this in O(log(n)) depth or lower??? Answer is yes, but how would you do it?