Massachusetts Institute of Technology
6.892: Theory of Parallel Systems
Professor Charles E. Leiserson

Lecture 5
February 20, 1997
Scribe: Tito Pena

Lecture 5: Data-Parallel Linked Data Structures

This lecture is about representing linked data structures in the data-parallel model, and about algorithms to operate in parallel on lists and trees.

1 Data Structure Representation

The representation that is typically employed in parallel programming languages is to have one datum correspond to one virtual processor. Take the conceptual representation of the list given in Figure 1. To store the given list, we use an array of records where each record is of the form (x,p). x corresponds to the datum and p is the datum's location in the array; here it corresponds the the processor index.

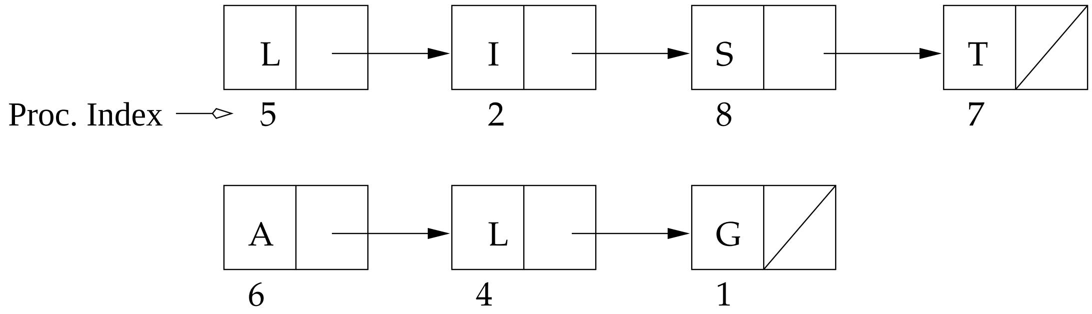

Figure 1: Example of singly linked list

Looking inside the memory of our parallel machine, we would see somehing similar to Figure 2. In virtual processor 5, we have the record $(L, 2)$ indicating that the piece of datum is L and the next pointer indexes into processor 2. Looking in virtual processor 2, we find the record $(I, 8)$, and so on. Notice that the number of virtual processors may be larger than the actual physical processors present. What is typically done in these cases is to group the virtual processors into "blocks" of physical processors. Figure 2 shows the "block layout" of virtual processors to physical processors.

2 List Ranking

Suppose that we are given a singly linked list (or collection of lists) with n objects and we wish to compute the distance to the end of the list for each object. For the linked list shown in Figure 1, Figure 3 below shows the list after the list ranking computation.
A naive approach to this problem is to simply propagate distances back from the end of the list. This method has a *critical path* of $\Theta(n)$ and *work* of order $\Theta(n)$ in the worst case. It

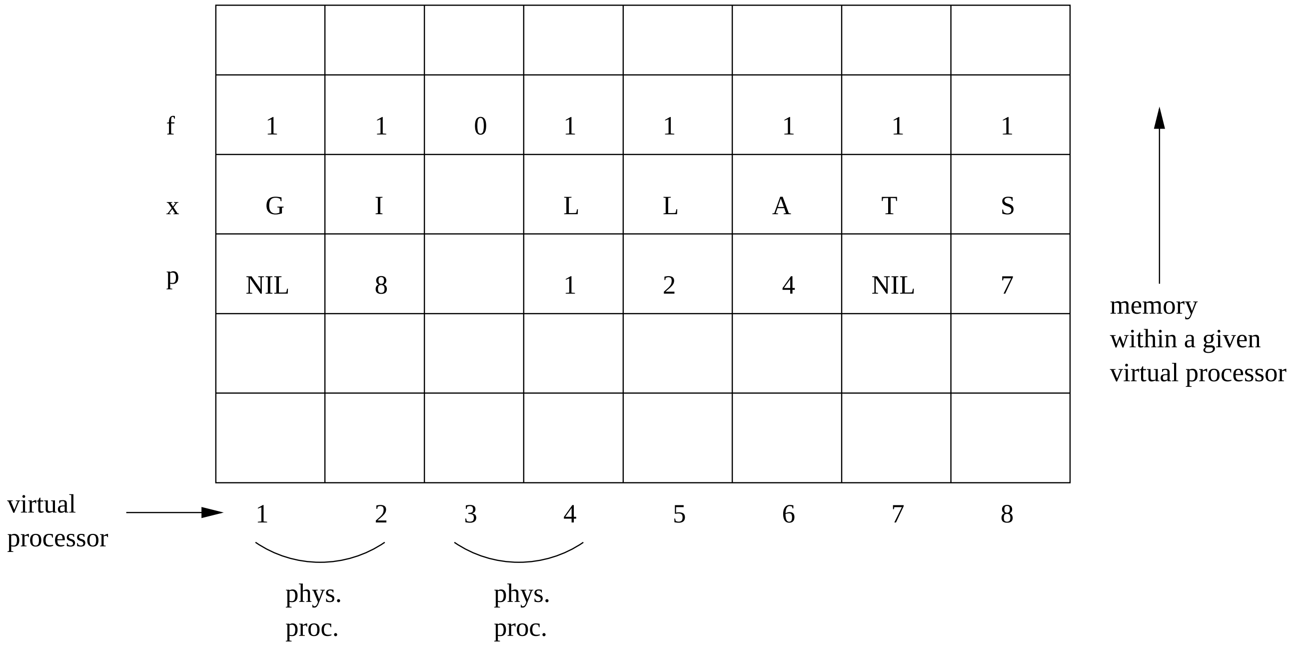

Figure 2: Memory Map

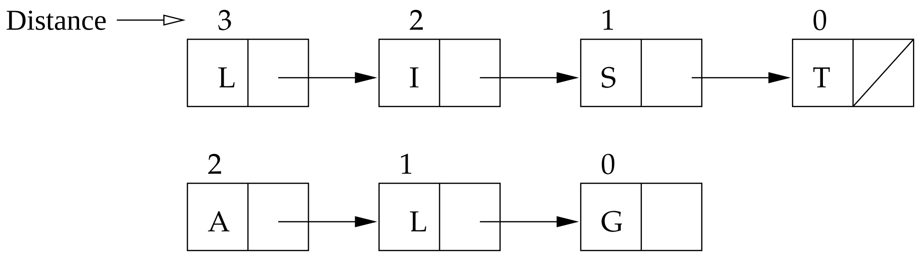

Figure 3: Idea of list ranking

is essentially a serial algorithm. For the linked list shown in Figure 1, Figure 3 below shows the list after the list ranking computation.

An efficient parallel solution using a technique known as ***pointer jumping*** requires only $\Theta(\lg n)$ time in the worst case. For our solution, we shall assume that there are as many processors as list objects and that the ith processor is responsible for the ith object. The algorithm is given below in parallel pseudocode.

LIST-RANKING(L)

for each processor i **do**, in parallel
    if $p[i] =$ NIL **then** $d[i] \leftarrow 0$
    else $d[i] \leftarrow 1$

while $\exists i$ with $p[i] \neq$ NIL **do**
  for each proc i **do**, in parallel
    if $p[i] \neq$ NIL **then**
      $d[i] \leftarrow d[i] + d[p[i]]$
      $p[i] \leftarrow p[p[i]]$

Figure 4 shows how the algorithm computes the distances. Each part of the figure shows the state of the list before an iteration of the **while** loop. Part (a) shows the list right after initialization. Parts (b) and (c) shows the list after the second and third iterations of the **while** loop. Part(d) shows the list in its final state.

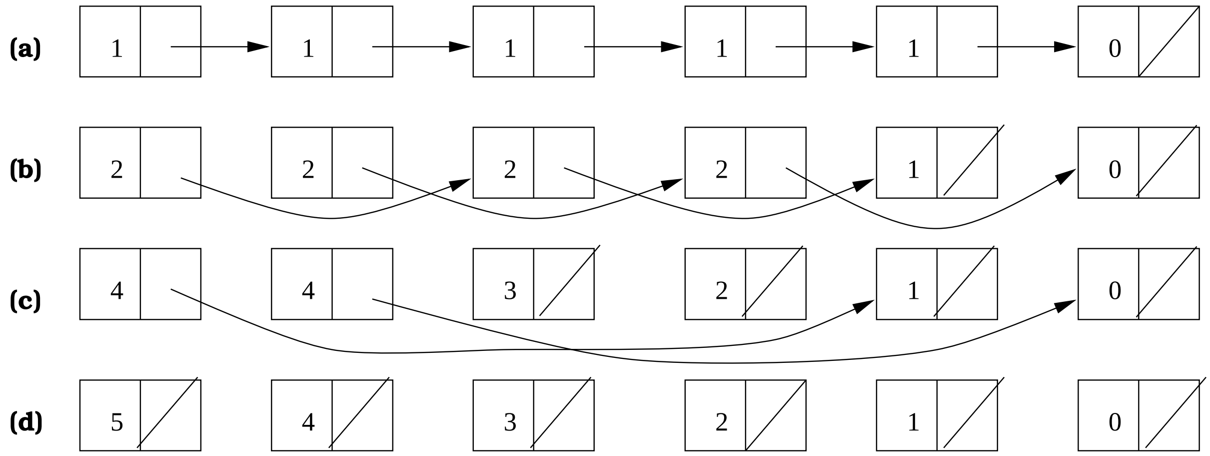

Figure 4: Using pointer-jumping for list ranking

2.1 Analysis

LIST-RANKING maintains the invariant that, for every node, the sum of the values on the chain that starts at that node is the correct distance from the end of the list. The invariant is initially true. Every time we splice an object out from the list, we add his value to our own and then splice him out. Thus we are always maintaining the correct distance to the end of the list.

The *critical path* of the LIST-RANKING algorithm is $\Theta(\lg n)$ in the worst case. The key observation is that each step of pointer jumping transforms each list into two interleaved lists: one consisting of the objects in the even positions and the other consisting of objects in the odd positions. Thus, each pointer jumping step doubles the number of lists but halves their size. In $\lceil \lg n \rceil$ steps, there will be n lists each containing one element each. The *work* involved is $O(n \lg n)$ in the worst case, since the algorithm requires n processors and it takes $\Theta(\lg n)$ time to execute. This algorithm is therefore not work-efficient, since the same computation can be done sequentially in $\Theta(n)$ work.

3 Parallel Prefix on Lists

A ***prefix computation*** is defined in terms of a binary, associative operator $\oplus$. The computation takes an input sequence $\langle x_1, x_2, \cdots, x_n \rangle$ and produces as output a sequence $\langle y_1, y_2, \cdots, y_n \rangle$ such that

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_1 \oplus x_2 \\ &\vdots \\ y_{k-1} &= y_k \oplus x_k \end{aligned}$$

for $k = 2,3,\ldots,n$. For convenience, we define the notation

$$[i,j] = x_i \oplus \cdots \oplus x_j$$

for integers i and j in the range $1 \leq i \leq j \leq n$. Then, by associativity,

$$[i,j] \oplus [j+1,k] = [i,k] = x_i \oplus \cdots \oplus x_k$$

for $0 \leq i \leq j < k \leq n$. In terms of this notation, our input becomes $\forall i, [i,i]$ and our output is $\forall i, [1,i]$.

The pseudocode for List-Prefix is very similiar for List-Ranking with the only difference being the initialization and the updating of d or y values. Below is the pseudocode for the EREW algorithm.

List-Ranking(L)

for each processor i **do**, in parallel
    $y[i] \leftarrow x[i]$

while $\exists i$ with $p[i] \neq$ nil **do**
    for each proc i **do**, in parallel
        if $p[i] \neq$ nil **then**
            $y[p[i]] \leftarrow y[i] \oplus y[p[i]]$
            $p[i] \leftarrow p[p[i]] \longleftarrow$ ***pointer jumping***

Note that every element updates its successor's value, not its own. Figure 5 demonstrates how the algorithm computes the prefix of the list. The steps are identical to those shown in Figure 4 for the list ranking algorithm.

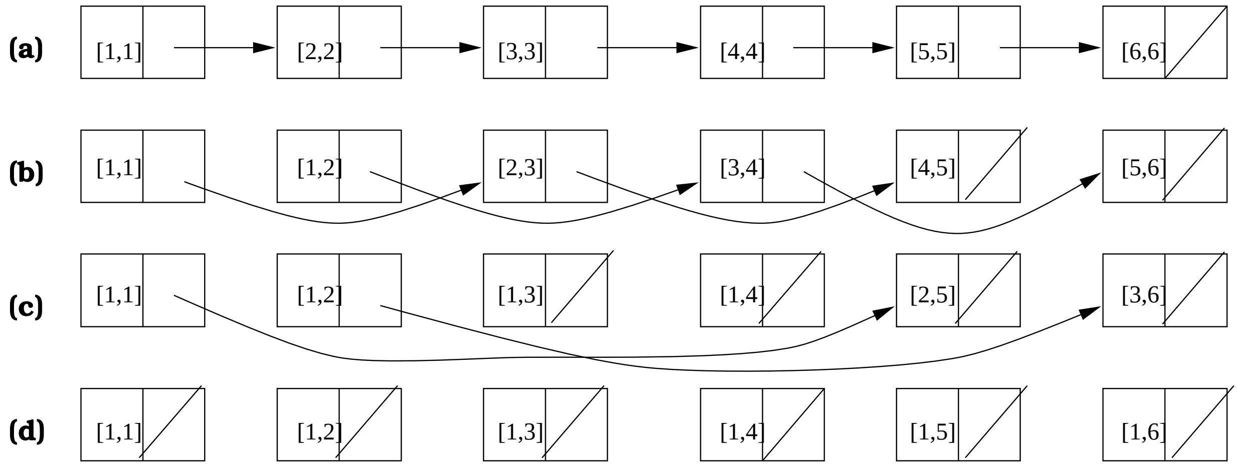

Figure 5: The parallel prefix algorithm for LIST-RANK on a linked list

3.1 Analysis

The reason why the algorithm works is similar to the one of LIST-RANKING: the algorithm maintains the invariant that, for every node, the sum of the values on the chain that ends at that node is the correct prefix of the original list.

Since the two algorithms, LIST-RANKING and LIST-PREFIX use the same pointer-jumping mechanism, the critical path and work is identical for both, namely $\Theta(\lg n)$ and $\Theta(n \lg n)$ respectively, in the worst case.

4 Euler Tour Technique

In this section, we show how to compute the depth of every node in an n-node binary tree. Performing this operation in a naive parallel manner yields us a running time of $O(n)$ worst case time. We could compute the depths by propogating a "wave" downward from the root of the tree. The wave reaches all nodes at the same depth simulataneously, and thus by incrementing a counter cariied along the wave, we can compute the depth at each node. Unfortunately, because this has a worst case time of $O(n)$ it is no better than a serial algorithm. Using the **Euler-tour** technique, however, we can compute the depths of the node in the binary tree in $O(\lg n)$, whatever the height of the tree is.

In the **Euler-tour** technique, we imagine each node to be composed of 3 virtual processors, call them A, B, and C, as shown in Figure 6.

We then form the Euler tour of the tree by connecting each processor in the following manner:

- A node's A processor points to the A processor of its left child, if it exists, and otherwise to its own B processor.

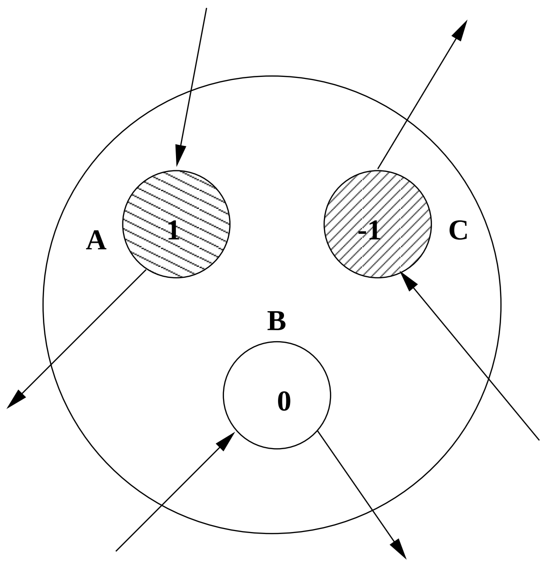

Figure 6: Node composed of 3 virtual processors

- A node's B processor points to the A processor of its right child, if it exists, and otherwise to its own C processor.
- A node's C processor points to the B processor of its parent if it is a left child and to the C processor of its parent if it is a right child. The root's C processor points to NIL.

Thus, the head of the linked list formed by the Euler tour is the root's A processor and the tail is the root's C processor as given in Figure 7.
We then perform a parallel prefix computation using ordinary addition as the associatuve operation. When we are done, the depth of each node resides in the node's C processor. Figure 8 shows the result of the parallel prefix on the linked list formed by the Euler tour.

4.1 Analysis

The claim was made that the depth of each node resides in its C processor. The reason is that the values are placed in A, B, and C in such a way that the net effect of visiting a subtree is to add 0 to the running sum. The A processor of each node i contributes 1 to the running sum in i's left subtree. The B processor contributes 0 because the depth of node i's left child equals the depth of node i's right child. Finally, the C processor contributes -1 so that from the perspective of node i's parent, the entire visit to the subtree rooted at node i has no effect on the running sum.

The list formed by doing an Euler-tour of the binary tree is a $3n$ element list and thus the parallel prefix computation takes only $\Theta(\lg n)$ time in the worst case. Because there are no concurrent memory accesses involved, the algorithm is an EREW algorithm.

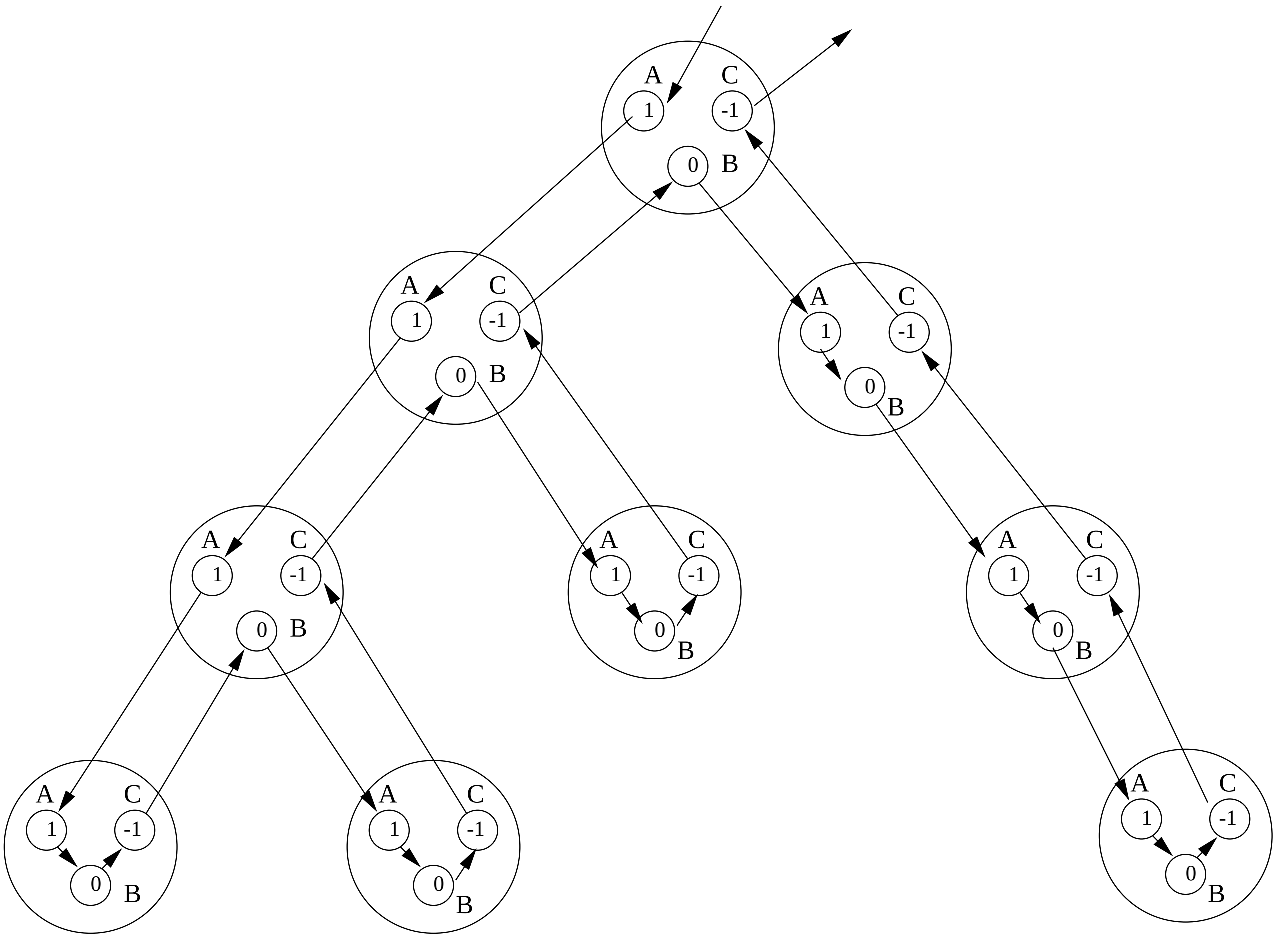

Figure 7: Using the Euler-tour technique to compute the depth of a node in a binary tree

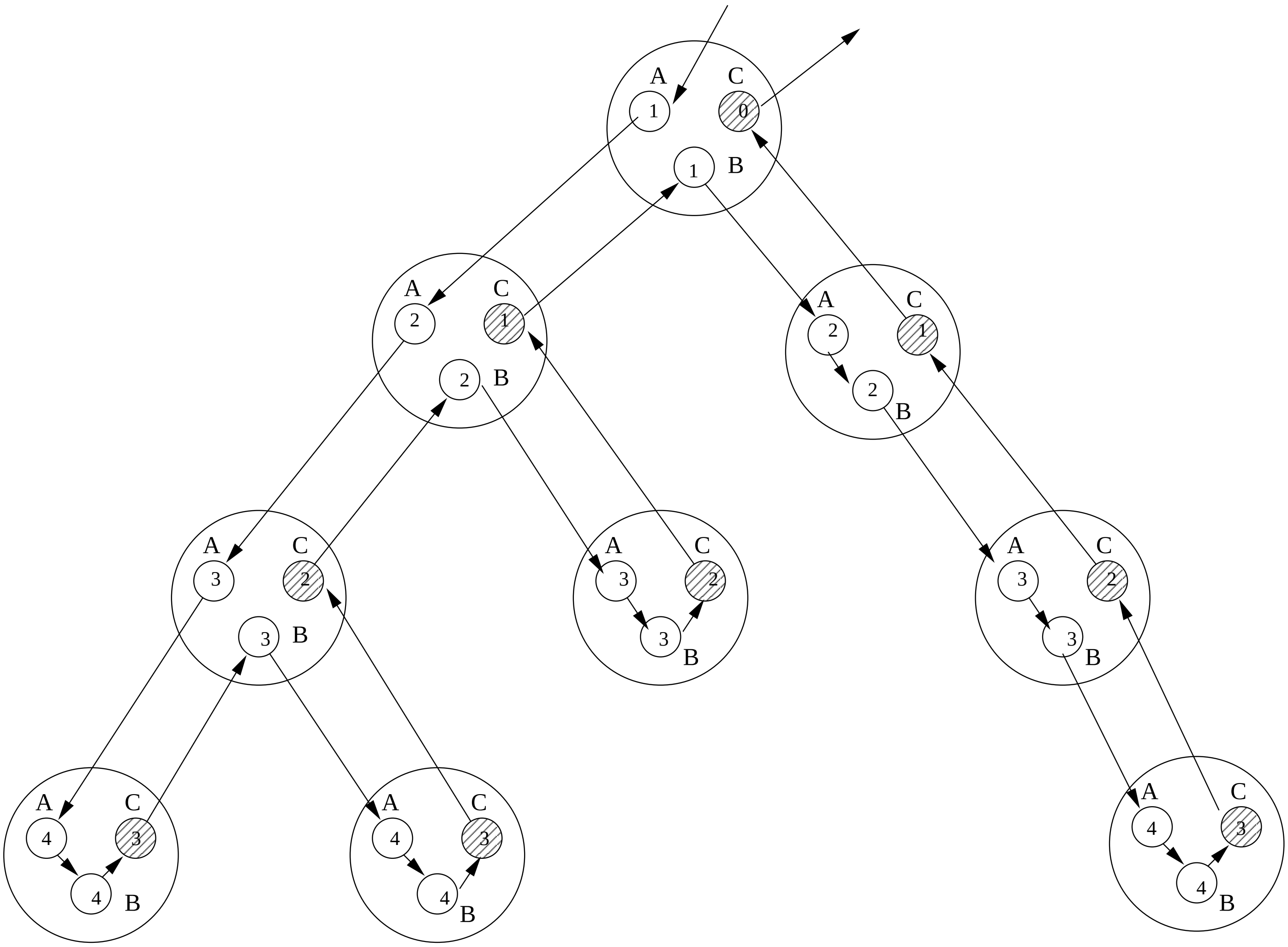

Figure 8: Result of parallel prefix on linked list

Another application of the Euler-tour technique is the computation of the size of subtrees. For each node, we are interested in the number of nodes in the subtree. This can be done as follows: store 1 in the A processor, and 0 in the B and C processors. Then compute the parallel prefix. The answer is the difference $C - A$, after the prefix.