ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute of Theoretical Computer Science
Bernd Gärtner, Rasmus Kyng, Angelika Steger, David Steurer, Emo Welzl

Solution 1

Each leaf node u in the tree $T = (V, E)$ is given a number $b(u)$. We want to devise a parallel algorithm with $O(\log n)$ depth and $O(n)$ total computation that computes for each node v the value $sl(v)$, the summations of the $b(u)$ values over all the leaves u that are descendants of v. We use the Eulerian tour technique again. After having identified the parents, we now define new weights for the arcs. We set$w(< parent(v), v >) = w(< v, parent(v) >) = 0$ if v is not a leaf and $w(< v, parent(v) >) = b(v)$ if v is a leaf. In other words, all arcs have weight zero, except the backward edges from the leaves. Now, we compute all the prefix sums of these weights, on the linked list provided by our Eulerian path. For each node v, the difference between the prefix sum on the arc $< v, parent(v) >$ and $< parent(v), v >$ is equivalent to the sum of all arcs in the subtree rooted in v, plus $w(< v, parent(v) >)$. This is clearly the same as the summations of the $b(u)$ values over all the leaves u that are descendants of v (including itself). Thus, we simply set $sl(v)$ to be the prefix sum on the arc $< v, parent(v) >$ minus the prefix sum on the arc $< parent(v), v >$. Again, the depth and total computations of the algorithm is asymptotically equivalent to the algorithm for computing the pre-order numbering, i.e., $O(\log n)$ depth and $O(n)$ work. The only difference is that at the end we subtract two prefix sum to compute $sl(v)$ for each node v in parallel, which does not change the depth and total work value asymptotically.

Solution 2

We are given an array of length n and each element in it is tagged with a number in $\{1, 2, \ldots, \sqrt{n}\}$, which indicates the index of its subproblem.Our goal is to devise an algorithm that in $O(\log n)$ depth and using $O(n \log n)$ work, creates one array A_i for each of the subproblems $i \in \{1, \ldots, \sqrt{n}\}$, holding all the elements tagged with number i.

For the sake of simplicity, we discuss in the terms of processors and show that we can handle this task by using n processors and $\mathcal{O}(\log n)$ time-steps. This immediately provides us with the depth $\mathcal{O}(\log n)$ and work $\mathcal{O}(n \log n)$.

As the hint suggests, imagine that we set a balanced binary tree on the n elements, where each leaf corresponds to an element in the array. Assume that for each leaf there is one processor which keeps a linked list of size one. This linked list includes the value of the corresponding element in the array. Assume that the processor associates a tag

and an id to the linked list, where the tag is simply the tag of the element and the id is the index of the element in the array.

Let v be a node in the binary tree, who has two children u and w. Furthermore, assume that node u (similarly w) is keeping a linked list for each group of elements in its subtree with the same tag (the elements corresponding to the leaves of the subtree rooted at u). Furthermore, assume each linked list has a processor that is responsible for it and knows the id of the linked list, its tag, and start/end of the linked list. We claim that it is possible that each of these processors does $\mathcal{O}(1)$ work, and afterwards node v will have the merged version of these linked lists. More precisely, node v will have a linked list for each group of elements which are in its subtree and have the same tag and each linked list has an assigned processor that is responsible for it and knows the id of the linked list (if the linked list is not merged we keep the id, otherwise take the minimum one), its tag, and start/end of the linked list. If our claim is true, by starting from the above initial configuration where we only have set the leaves, after $\mathcal{O}(\log n)$ time-steps we will end up with a group of linked lists in the root of the tree such that each linked list includes exactly all elements with some tag from $\{1, \cdots, \sqrt{n}\}$.

If there is a linked list in u (similarly w) and there is no linked list with the same tag in node w (respectively u) simply keep the linked list in v with the same tag, id, and assigned processor. Now, assume there are two linked lists L and L' respectively in u and w and assigned processors p and p'. Without loss of generality, assume that the id of L is smaller than the id of L'; then, we simply concatenate the two linked lists and assign the id of L to the new linked list and let processor p be responsible for it (we do not need processor p' anymore). There is one subtle point here: How do the processors know about each other's linked lists? To address this, we assign a fixed part of the shared memory to every possible choice of a node and a tag to keep the necessary information, like id and start/end, if a linked list with that tag appears in that node during the process. Thus, for instance in the above argument processor p (similarly p') knows which part of the memory it should check to find out whether there is a linked list with similar tag in w (respectively u) or not. If yes, then it compares the ids and if the id of its linked list is smaller, it will concatenate its linked list to the other one and will write the updated information (id and start/end) in the part of the memory which is assigned to this newly created linked list, which is determined by the tag and node v.

We start with n processors and during the process might even get rid of some of them and during each of the $\mathcal{O}(\log n)$ time-steps every processor does $\mathcal{O}(1)$ work. Therefore, so far we need $\mathcal{O}(n \log n)$ work and $\mathcal{O}(\log n)$ depth.

We need to turn each linked list into an array at the end. To do that, we assign to each element in the linked list weight one and apply a prefix sum on all the weights. This assigns each element in the linked list with a unique integer from 1 to ℓ, where ℓ is the length of the linked list. Now, create an array of size ℓ and write each element in the index corresponding to its prefix sum. This does not change the asymptotic value of the depth and work needed, since we can do parallel prefix with $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n)$ work for an array of size n. After all, each element is in exactly one array and the summation of the lengths of the arrays is equal to n.

Finally, it is worth to stress that the value $\sqrt{n}$ is irrelevant and the aforementioned argument applies to any set of tags.

Solution 3

We want to show that it is possible to simultaneously identify the minimum neighbors for all the fragments, using $O(\log n)$ depth and $O(m + n \log n)$ work.

(A) We use a similar idea to the one from Exercise 1. Again consider a balanced binary tree with n leaves and height $\mathcal{O}(\log n)$. We assign to each of the leaves the adjacency list of one of the nodes in the graph. Assume the tag of each linked list is the root of the fragment that the corresponding node belongs to and its id is simply the label of the node (suppose the nodes are labeled from 1 to n). Furthermore, there are n processors each responsible for one of the linked lists. Now, we can apply the same argument as in Exercise 1 to show that after logarithmically many rounds and by doing $\mathcal{O}(n \log n)$ work, we have f linked lists, where f is the number of fragments, such that each linked list contains exactly all elements from one of the fragments (the elements here are the nodes which share at least one edge with the fragment). Analogous to Exercise 1 we can turn each of these linked lists to an array by applying the parallel prefix. However, here we would need $\mathcal{O}(m)$ work and $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ depth since each linked list could have up to m elements. Thus, we can do the required sub-task in depth $\mathcal{O}(\log n)$ and work $\mathcal{O}(m + n \log n)$. Note that the sum of the lengths of these arrays is at most $2m$ since for each edge both of its end-points appear as the elements of the initial linked lists. Furthermore, if a node is connected to more than one node in some fragment f_i, then several copies of it might exist in the array A_i created for f_i but that is not an issue at all.

(B) Each array A_i generated in part (A) for a fragment f_i includes all nodes which share at least one edge with nodes in f_i (as mentioned above, there might be several copies of node). Now, in parallel we check the pointer of each node in the array, which is pointing to the root of the fragment that it belongs to. If the pointer is pointing to the root of fragment f_i, say r_i, we replace it with $+\infty$, otherwise we replace it with its successor, which is the root of some other fragment. It is easy to see that this is done in depth $\mathcal{O}(1)$ and work $\mathcal{O}(m)$ since as discussed in part (A) the sum of the lengths of these arrays is $2m$ and we handle each entry separately in constant time.

(C) We can find the minimum value in each of the modified arrays from (B) in $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ depth and $\sum_{i=1}^{f} \mathcal{O}(m_i)$ work, where f is the number of fragments and m_i is the size of array A_i. Since $\sum_{i=1}^{f} m_i = 2m$, the work needed is $\mathcal{O}(m)$.

Solution 4

Consider a component which contains nodes with identifiers $1, \cdots, n'$ for some $n' \leq n$. We want to show that at the end of the algorithm, the component will have its identifier equal to the minimum identifier among the nodes, which is 1. Recall that initially we

have fragments $f_1, \cdots, f_{n'}$, where fragment f_i includes node i with a self-loop. Then, in each time-step for each root node r, we remove its self-loop in its fragment and instead set $D(r) = p(r)$, i.e., the root of the proposed fragment with which the fragment of r wants to merge. Notice each fragment proposes to the neighboring fragment with minimum identifier. By applying an inductive argument, we show that node 1 is the root of the fragment that it belongs to at the end of each time-step. Initially this is trivially true. Assume that the statement is true by the end of the i-th time-step. Now, all the proposals are applied. If there is only one fragment left in this component, then no merging happens and 1 remains as the root of the component. Otherwise node 1 proposes to another root r and r also trivially proposes to node 1. Thus we will have a pseudo-tree with the cycle of length two between nodes 1 and r. Then, we do $\log n$ repetitions of *pointer-jumping*. More precisely, for $\log n$ repetitions, for each node v, we set $D(v) = D(D(v))$. Afterwards, the pointer $D(v)$ of each node is to 1 or r. As a final step, we set $D(v) = \min D(v), D(D(v))$, so that all nodes will point to 1. Thus, our claim is correct.

Solution 5

Let us first sketch the main idea of the solution and discuss intuitively why it should work. We pick $n^{0.9}$ pivots at random. Then, we sort these pivots, which is doable in our desired work and depth. If we consider the sorted sequence of the elements, the pivots partition it into parts of size roughly $n^{0.1}$. If we label the sorted pivots from e_1 up to $e_{n^{0.9}}$, one would expect to find the k-th element somewhere close to $e_{\lfloor k/n^{0.1} \rfloor}$ in the sorted sequence. The idea is to show that if we actually pick a close neighborhood of pivot $e_{\lfloor k/n^{0.1} \rfloor}$ in the sorted sequence, namely $n^{1-\epsilon}$ elements before and $n^{1-\epsilon}$ elements after for some small constant $\epsilon > 0$, then the k-th element will be among them with our required probability. Thus, we will be left with $\mathcal{O}(n^{1-\epsilon})$ elements. One can simply sort all these elements in $\mathcal{O}(n^{1-\epsilon} \log n^{1-\epsilon}) = \mathcal{O}(n)$ work and $\mathcal{O}(\log n^{1-\epsilon})$ depth. In the rest of the solution, we provide a formal argument for this claim.

Let us mention that we will apply the following variant of the Chernoff bound several times. Furthermore, we say an event happens with high probability (or shortly w.h.p.) if it occurs with probability $1 - \mathcal{O}(n^{-c})$ for some constant $c \geq 1$, let's say $c = 5$ for this exercise.

Theorem 1. *(Additive Chernoff bound) Suppose $X_1, \cdots, X_n$ are independent random variables taking values in $\{0, 1\}$ and let X denote their sum, then for any $t > 0$*

$$\Pr[|X - \mathbb{E}[X]| \geq t] \leq 2e^{-2t^2/n}.$$

We set each element as a pivot with probability $1/n^{0.1}$ independently. (Note that the independence is a key property here since it allows us to apply the Chernoff bound.) In this way, in expectation we will have $n^{0.9}$ pivots, but actually we might get more or less than $n^{0.9}$ pivots. By applying a Chernoff Bound, one can show that with high probability the number of selected pivots is upper bounded by $2n^{0.9}$. In that case, we can sort the

pivots with $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ depth. Let N denote the number of sampled pivots and $e_1, e_2, \ldots, e_N$ the pivots in sorted order. As we said, one would expect the k-th element to appear somewhere close to $e_{k/n^{0.1}}$ in the sorted sequence. In Proposition 2, we phrase this formally.

Proposition 2. *The k-th element is between pivots $e_{\lceil \mu - n^{0.6} \rceil}$ and $e_{\lfloor \mu + n^{0.6} \rfloor}$ in the sorted sequence w.h.p. where $\mu = \frac{k}{n^{0.1}}$.*

In the above statement, the indices might take values smaller than 1 or larger than N. To fix that, one can simply consider two auxiliary pivots e_0 and e_{N+1} which are respectively the smallest and largest element and assign the non-defined pivots to these two pivots.

Proof. Let random variable X_i be 1 if the i-th element in the sorted sequence is a pivot and 0 otherwise. Define random variable $X := \sum_{i=1}^{k} X_i$ which is the number of pivots smaller than or equal to the k-th element. $\mathbb{E}[X_i] = \frac{1}{n^{0.1}}$ for each $1 \leq i \leq k$, which implies that $\mathbb{E}[X] = \frac{k}{n^{0.1}} = \mu$ by linearity of expectation. Now, by applying the Chernoff bound, we get

$$\Pr[|X - \mu| \geq n^{0.6}] \leq 2e^{-2\left(n^{0.6}\right)^2/n} = 1 - O(n^{-5}).$$

Thus, w.h.p. the k-th element is larger than $e_{\lceil \mu - n^{0.6} \rceil}$ and smaller than $e_{\lfloor \mu + n^{0.6} \rfloor}$. □

Proposition 3. *Among every $n^{0.9}$ consecutive elements in the sorted sequence, there are at least $n^{0.7}$ pivots w.h.p..*

Proof. Consider a sequence of length $n^{0.9}$. Let random variable Y denote the number of pivots in this sequence. We have $\mathbb{E}[Y] = \frac{n^{0.9}}{n^{0.1}} = n^{0.8}$ and Y is the summation of $n^{0.9}$ independent random variables. Thus, by applying the Chernoff bound we get

$$\Pr[Y \leq n^{0.7}] \leq \Pr[|Y - \mathbb{E}[Y]| \geq n^{0.8} - n^{0.7}] \leq 1 - O(n^{-6}).$$

There are at most n such sequences; thus, by applying the union bound with probability $1 - \mathcal{O}(\frac{1}{n^5})$, among every $n^{0.9}$ consecutive elements in the sorted sequence, there are at least $n^{0.7}$ pivots. □

We put Proposition 2 and Proposition 3 in parallel to prove our claim. We can find out which elements are between $e_{\lceil \mu - n^{0.6} \rceil}$ and $e_{\lfloor \mu + n^{0.6} \rfloor}$ by $\mathcal{O}(n)$ work and $\mathcal{O}(1)$ depth. Based on Proposition 2, the k-th element must be among them w.h.p.. Furthermore, by Proposition 3, the number of these elements is less than $n^{0.9}$. This is true since from $e_{\lceil \mu - n^{0.6} \rceil}$ to $e_{\lfloor \mu + n^{0.6} \rfloor}$ in the sorted sequence, there are $O(n^{0.6})$ pivots. Therefore, now we are left with at most $n^{0.9}$ elements, that we can sort in $\mathcal{O}(n^{0.9} \log n^{0.9}) = \mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ depth. The k-th element that we are looking for is the $(k - k')$-th element in this sorted sequence, where k' is the number of elements smaller than $e_{\lceil \mu - n^{0.6} \rceil}$. It is left to compute k'. Create an array B of size n and compare all elements with $e_{\lceil \mu - n^{0.6} \rceil}$ in parallel and set $B[i] = 1$ if $A[i] < e_{\lceil \mu - n^{0.6} \rceil}$ and $B[i] = 0$ otherwise. Now, a parallel prefix on B gives us k', which can be done in $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ depth.