ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute of Theoretical Computer Science
Bernd Gärtner, Mohsen Ghaffari, Angelika Steger, David Steurer

Algorithms, Probability, and Computing **Solutions KW50** **HS19**

Solution 1

We provide a strategy which decreases the number of remaining items in each phase until we only have the item with the maximum value left. Let k_t for $t \geq 1$ denote the number of remaining items at the end of the t-th phase. In the first phase, we do a simple step of comparing $n/2$ consecutive pairs to reduce the number of remaining items to at most $n/2$; thus, $k_1 = n/2$. In the t-th phase for $t \geq 2$ we partition the remaining k_{t-1} items into ${k_{t-1}}^2/n$ many groups of size n/k_{t-1}. Then, we apply the algorithm from the lecture on each group separately. Note that we require $(n/k_{t-1})^2$ processors for each group, which implies that

$$\frac{k_{t-1}^2}{n} \cdot \frac{n^2}{k_{t-1}^2} = n$$

processors are needed. Furthermore, recall that each of these phases can be done in $\mathcal{O}(1)$ time-steps. Therefore, by using only n processors in the t-th phase we drop the number of remaining items from k_{t-1} to k_t in $\mathcal{O}(1)$ time-steps. We have $k_t = \frac{k_{t-1}^2}{n}$ for $t \geq 2$, where $k_1 = n/2$. Now, we prove that $k_t = \frac{n}{2^{2^{t-1}}}$ for $t \geq 1$ by induction. The base case holds because $k_1 = n/2 = n/2^{2^0}$. As the induction hypothesis, assume that $k_t = \frac{n}{2^{2^{t-1}}}$ for some $t \geq 1$. For the inductive step, we have

$$k_{t+1} = \frac{k_t^2}{n} \overset{\text{I.H.}}{=} \frac{(\frac{n}{2^{2^{t-1}}})^2}{n} = \frac{n}{2^{2^t}}.$$

Furthermore, $\frac{n}{2^{2^{t-1}}} \leq 1$ for $t \geq \log\log n + 1$. Thus, after at most $\log\log n + 1$ phases, we are left with only one item, namely the maximum one. Since each phase requires $\mathcal{O}(1)$ time-steps, the run time of this algorithm is in $\mathcal{O}(\log\log n)$.

Solution 2

Based on Brent's principle, if an algorithm does x total work and has depth t (i.e., critical path of length t), then using p processors, this algorithm can be run in $x/p + t$ time. In this setting, we have $x = \mathcal{O}(n)$ and $t = \mathcal{O}(\log n)$. Thus, to get an algorithm which runs in $\mathcal{O}(\log n)$, we need $\Omega(n/\log n)$ processors.

In the Parallel Prefix Problem, as the input we are given an array A of length n and we want to compute an array B of length n that contains all the prefix sums, i.e., $B[j] = \sum_{i'=1}^{j} A[i']$ for all $j \in \{1, \cdots, n\}$. We provide an algorithm which solves this problem

with $n/\log n$ processors and runs in $\mathcal{O}(\log n)$ time-steps. Assume that we have $n/\log n$ processors $p_1, \cdots, p_{n/\log n}$. We divide the array into $n/\log n$ sub-arrays $A_1, \cdots, A_{n/\log n}$ of $\log n$ consecutive elements in the array. The processor p_i is supposed to be responsible for the sub-array A_i. First, each of the processors will compute the prefix sum for the last element in its sub-array (with respect to the sub-array) which is doable in $\mathcal{O}(\log n)$ time-steps; that is, at the end of this phase $B[i \log n] = \sum_{i'=(i-1)\log n+1}^{i \log n} A[i']$ for $i \in \{1, \cdots, n/\log n\}$. Now, we consider only values of B[j] which correspond to the last element of the sub-arrays, i.e., $B[i \log n]$. This reduces the size of the problem to $n/\log n$. We know how to solve the problem on these $n/\log n$ elements by applying the method from the lecture, in $\mathcal{O}(\log n)$ time-steps. Note that this is doable since the number of processors and elements are equal. After this phase, $B[i \log n]$ for $i \in \{1, \cdots, \frac{n}{\log n}\}$ has our desired prefix sum, meaning $B[i \log n] = \sum_{i'=1}^{i \log n} A[i']$. Now, each processor computes its related values in $\mathcal{O}(\log n)$. More precisely, processor p_i for $i \geq 1$, set $B[(i-1)\log n + j'] = B[(i-1)\log n] + \sum_{i'=(i-1)\log n+1}^{(i-1)\log n+j'} A[i']$ for $j' \in \{1, \cdots, \log n - 1\}$, where we assume $B[0] = 0$. Note this can be done in $\mathcal{O}(\log n)$ time-steps. Thus, we have an algorithm which needs $n/\log n$ processors and solves the Parallel Prefix Problem in $\mathcal{O}(\log n)$ time-steps.

Solution 3

As our input we have a graph on the node set $\{1, \cdots, n\}$ whose edge set is given in the form of an $n \times n$ binary adjacency matrix, where the entry at location (i, j) is 1 if the i-th and j-th nodes are adjacent, and 0 otherwise. We want to devise a parallel algorithm with $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n^2)$ work that transforms this adjacency matrix to linked lists. In the linked lists, for each node $v \in V$, the nodes adjacent to v are given in a linked list $L[v] =< u_0, u_2, \cdots, u_{d(v)-1} >$, where $d(v)$ is the degree of the node v.

Since we can handle each row independently, it suffices to show that for a row we can construct the adjacency list for the corresponding node in $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n)$ work. If we have two linked lists L and L' and their starting and ending, we can concatenate L and L' in c time-steps for some constant $c > 0$. Simply, set the successor of the last element of L to be the first element of L'. Now, we have a linked list L'', where the starting is the same as the starting of L and the ending is identical to the ending of L'. To each entry a_{ij} in row a_i for $1 \leq j \leq n$, we initially assign a linked list of one element which has the value j if $a_{ij} = 1$ and an empty linked list if $a_{ij} = 0$. Now, we divide these linked lists into $n/2$ consecutive disjoint pairs and concatenate each pair. In the next round, we divide the $n/2$ newly created linked lists into $n/4$ consecutive disjoint pairs and concatenate each pair. After $\log n$ steps we will be left with a linked list L[i] which includes all nodes adjacent to node i. Thus, the depth is $\mathcal{O}(\log n)$. In the t-th time-step, the work done is $c \cdot \frac{n}{2^t}$ since we have to concatenate $n/2^t$ pairs and for each we need c time-steps. Therefore, the overall work for a row is

$$\sum_{t=1}^{\log n} c\frac{n}{2^t} = \mathcal{O}(n).$$

Solution 4

Consider a Depth First Search traversal of the nodes (according to the adjacency lists, which is the same as how the Eulerian path is defined in the lecture notes). Our objective is to compute a post-oder numbering $post : V \to \{0, \ldots, n-1\}$ of the nodes. That is, in this numbering, for each node v, first a post-order numbering of the subtree rooted in the first child of v appears, then a post-order numbering of the subtree rooted the second child of v, an so on, and finally v appears.

Using the Eulerian tour technique, we can solve the problem, as follows: After having identified the parents, we now define a new weight for the arcs. We set $w(< parent(v), v >) = 0$ and $w(< v, parent(v) >) = 1$. Notice that the former are forward arcs in the DFS and the latter are backward arcs. Then, we compute all the prefix sums of these weights, on the linked list provided by our Eulerian path (i.e., maintained by the successor pointers). Hence, each arc knows the number of backward arcs before it (and including itself), in the Eulerian path. Set $post(r) = n - 1$ for the root node r. For each node $v \neq r$, set $post(v)$ to be the prefix sum on the arc $< v, parent(v) >$ minus one, which is equivalent to the total number of backward arcs before this arc. This gives exactly our desired post-order numbering because each backward edge corresponds to a node which appears before node v in the post-order numbering.

Based on the lecture notes (similar to the computation of pre-order numbering), the aforementioned algorithm has $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n)$ work. The only difference is that at the end we subtract the prefix sum on the arc $< v, parent(v) >$ by one for each node v in parallel to obtain $post(v)$, which is doable in $\mathcal{O}(1)$ depth and $\mathcal{O}(n)$ work.

Solution 5

We want the number of descendants $des(v)$ for each node v, which is the total number of nodes in the subtree rooted at node v.

We use the Eulerian tour technique again. After having identified the parents, we now define the weights for the arcs similar to pre-order numbering. We set $w(< parent(v), v >) = 1$ and $w(< v, parent(v) >) = 0$. In other words, we set the weight of all forward arcs in the DFS to 1 and the weight of all backward edges to 0. Then, we compute all the prefix sums of these weights, on the linked list provided by our Eulerian path. Hence, each arc knows the number of forward arcs before it (and including itself), in the Eulerian path. Set $des(r) = n$ for the root node r. For each node $v \neq r$, set $des(v)$ to be the prefix sum on the arc $< v, parent(v) >$ minus the prefix sum on the arc $< parent(v), v >$, plus one. The prefix sum on an arc is equivalent to the total number of forward arcs before this arc (including itself). Furthermore, each forward arc corresponds to visiting an unvisited node in the DFS. Therefore, the prefix sum on the arc $< v, parent(v) >$ minus the prefix sum on the arc $< parent(v), v >$ is equal to the total number of forward arcs in the subtree rooted at v. We add one to this value since it does not include the forward arc corresponding to node v itself.

The depth and the work required by this algorithm is similar to the one for computing the pre-order numbering, except at the end we compute $des(v)$ for each node v in parallel, which can be done in depth $\mathcal{O}(1)$ and work $\mathcal{O}(n)$. Therefore, this algorithm performs in depth $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.