Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, A. R. Molla, O. Saukh February 17, 2016

Algorithm Theory, Winter Term 2015/16
Problem Set 14 - Sample Solution

Exercise 1: All-prefix-sums on Multi-dimensional Matrices (6 points)

In this exercise, we consider a generalization of the the all-prefix-sums problem discussed in the lecture. We study the following all-prefix-sums problem defined on a d-dimensional array.
We are given a $n \times n \times \cdots \times n$ array $\mathcal{A}$ with entries $a_{i_1,i_2,\ldots,i_d}$ (for $i_j \in \{1,\ldots,n\}$). The goal is to calculate the all-prefix-sums array $\mathcal{S}$ with entries $s_{i_1,i_2,\ldots,i_d}$ which are defined as follows

$$s_{i_1,i_2,\ldots,i_d} := \sum_{j_1=1}^{i_1}\sum_{j_2=1}^{i_2}\cdots\sum_{j_d=1}^{i_d} a_{j_1,j_2,\ldots,j_d}$$

Note that for $d = 1$, the problem is the usual all-prefix-sums problem from the lecture.

(a) (3 points) To warm up, we first consider the case $d = 2$. Give an efficient algorithm to solve the 2-dimensional all-prefix-sums problem. What are the work T_1 and span T_∞ of your solution.

Hint: *The problem can be solved by performing $2n$ standard all-prefix-sums computations.*

(b) (2 points) Generalize the above algorithm to $d \geq 2$ dimensions. What are work T_1 and span T_∞ of the resulting parallel algorithm?

(c) (1 points) What is the minimum number of processors needed such that asymptotically, the maximum possible speed-up can be achieved?

Solution

(a) For $d = 2$, we can think of $\mathcal{A}$ as a 2-dimensional matrix of order $n \times n$. The $(i,j)^{th}$ entry of $\mathcal{A}$ is $a_{i,j}$. By the definition, the $(i,j)^{th}$ entry of all-prefix-sums matrix $\mathcal{S}$ is $s_{i,j} := \sum_{k=1}^{i}\sum_{l=1}^{j} a_{k,l}$. That is, $s_{i,j}$ is the sum of all entries of $\mathcal{A}$ with indices in $[1,i]\times[1,j]$. We can write $s_{i,j}$ as

$$\begin{aligned} s_{i,j} &= (a_{1,1} + a_{2,1} + \cdots + a_{i,1}) + (a_{1,2} + a_{2,2} + \cdots + a_{i,2}) + \cdots + (a_{1,j} + a_{2,j} + \cdots + a_{i,j}) \\ &= \bar{s}_{i,1} + \bar{s}_{i,2} + \cdots + \bar{s}_{i,j}, \end{aligned}$$

where $\bar{s}_{i,k} = (a_{1,k} + a_{2,k} + \cdots + a_{i,k})$ is the sum of the first i entries of the k-th column of $\mathcal{A}$. Therefore, $s_{i,j}$ is the sum of the first j entries of the i^{th} row of the matrix $\bar{\mathcal{S}} = (\bar{s}_{i,j})$. Notice that any column k of the matrix $\bar{\mathcal{S}}$ is the all-prefix-sums of the entries in the k^{th} column of $\mathcal{A}$. Hence, we first construct $\bar{\mathcal{S}}$ by computing the all-prefix-sums of all the columns of $\mathcal{A}$ separately. We can do this in parallel for each column of $\mathcal{A}$. Since $\mathcal{A}$ has n columns, we have n instances of all-prefix-sums problem. Using the parallel algorithm of the lecture, we can compute all the n all-prefix-sums in parallel. We thus compute the n columns of $\bar{\mathcal{S}}$ in parallel.

Now the final matrix $\mathcal{S}$ is the all-prefix-sums of each rows of the matrix $\bar{\mathcal{S}}$. Again we can use the parallel all-prefix-sums algorithm of the lecture for each rows of $\bar{\mathcal{S}}$ in parallel and compute $\mathcal{S}$. We know that (from the lecture), for one all-prefix-sums instance, total work is $T_1 = n$ and span is $T_\infty = \log n$. Here we compute $2n$ such instances: n for computing $\bar{\mathcal{S}}$ and n for computing the final $\mathcal{S}$. Hence, the total work will be $T_1 = 2n^2$ and the span will be $T_\infty = 2\log n$, since the final $\mathcal{S}$ computation depends on the computation of the intermediate matrix $\bar{\mathcal{S}}$ (each dependant tree height is $\log n$).

(b) The generalization is straightforward. Let us first look at the case for $d = 3$. We can think of it as a combination of two phases: first compute n instances of 2-dimensional all-prefix-sums problems and then compute the final all-prefix-sums array corresponding to the third dimension. We first solve these n 2-dimensional all-prefix-sums in the similar way as done in question (a). In fact, we can compute all the n instances in parallel. The work for this phase would be $n \cdot 2n^2 = 2n^3$. Then we have to compute the all-prefix-sums corresponding to the third dimension. There are again n^2 instances of 1-dimensional all-prefix-sums problems. This will give us the final all-prefix-sums array for $d = 3$. The total work will be $T_1 = 2n^3 + n^2 \cdot n = 3n^3$. The span will be $T_\infty = 2\log n + \log n = 3\log n$, since the computation of the all-prefix-sums corresponding to the third dimension depends on the previous n instances of 2-dimensional all-prefix-sums computations. In the similar way, we can extend the solution for general $d \geq 2$. We can do step by step as above: first compute n instances of $(d-1)$-dimensional all-prefix-sums problems and then compute the final all-prefix-sums array corresponding to the d^{th} dimension. The total work will be $T_1 = dn^d$ and the span will be $T_\infty = d\log n$.

(c) We use the Brent's Theorem (see the lecture). Recall that if p is the number of processors, then $p = O(\frac{T_1}{T_\infty})$. Therefore by putting the above values of T_1 and T_∞, we get $p = O(\frac{n^d}{\log n})$.

Exercise 2: Merging Two Sorted Arrays (6 points)

You are given two sorted arrays $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_n]$, each of size n. The goal is to merge them into one sorted array $C = [c_1, \ldots, c_{2n}]$ of length $2n$.

(a) (1.5 points) We first consider the following subproblem. Given an index $i \in \{1, \ldots, n\}$, we want to find the final position $j \in \{1, \ldots, 2n\}$ of the value a_i in the array C. Give a fast sequential algorithm to compute j. What is the (sequential) running time of your algorithm?

(b) (1.5 points) Use the above algorithm to construct a parallel merging algorithm. The work T_1 of your algorithm should be at most $O(n \log n)$ and the span T_∞ should be (asymptotically) as small as possible. What is the span T_∞ of your algorithm?

(c) (3 points) We now want to solve the merging problem in constant time (in parallel). Show that by using $O(n)$ processes, the subproblem considered in (a) can be solved in $O(1)$ time. Use this to get a constant-time parallel algorithm to merge the two sorted arrays. How many processors do you need to achieve a constant-time algorithm?

Solution

Assume that all arrays are sorted in ascending order.

(a) Note that the first $(i-1)$ values in A are before a_i in C, because the array A is sorted. Now we have to find out how many values in B are before a_i. That is to find the largest index k such that $b_k \leq a_i$. One easy way for this is to compare a_i with the elements of B one by one starting from b_1 and find the index k. This will take $O(n)$ time in general, since the size of the array B is n. However, we can do it faster using the divide and conquer approach (this is exactly the binary search). We recursively break the array B into two parts of equal size and check in which side a_i falls (and ignore the other side). Using divide and conquer approach we can find the index k in $O(\log n)$ time. Once we find the k, then the final position of a_i would be $(i-1) + k + 1$ (assuming the array indices starting from 1).

(b) In the above algorithm, we see that one processor can find the final position of a value a_i in $O(\log n)$ time. Now we consider n processors corresponding to each value a_i in A and compute their positions in the output array C in parallel. All the processors can find the final position of every values of A in $O(\log n)$ time. Then in the same way, we compute the position of all the values of B in C using n processors and $O(\log n)$ time. Hence, we can merge the two sorted arrays into one sorted array in $O(\log n)$ time, using n processors. The total work is $T_1 = O(n \log n)$ and the span is $T_\infty = O(\log n)$.

(c) Consider a particular value a_i of A and we want to find the final position of a_i in C. Let us take n processors $p_k : k = 1, 2, \dots, n$. Each processor p_k compares the value a_i with two consecutive values b_{k-1} and b_k in B. All the processors do it in parallel. (Note that the array indices starting from 1, so we assume $b_0 = -\infty$ for consistency). Since the values b_k are in ascending order (sorted), there will be only one processor p_t which see that $b_{t-1} \leq a_i$ and $b_t > a_i$. That is there are exactly $t-1$ values in the array B which are smaller than a_i. Hence, the processor p_t can decide the final position of a_i which would be $(i-1)+(t-1)+1$ (since there are $i-1$ values smaller than a_i in A). The processor p_t can write the value a_i safely in the final array C. Note that the processor p_n may observe that $b_n \leq a_i$, then the final position of a_i would be $(i-1)+n+1$. Since all the processors computing this in parallel, it takes constant time. Also we used n processors for this. We can extend this algorithm for all the values in A using n^2 processors in $O(1)$ time: for each a_i in A, run the algorithm in parallel. For this, we need a total n^2 processors and they can write all the values a_i in the correct place in C. Notice that there would not be any conflicts when writing in C, since a processor p_t only writes the value in one cell of the array C. Thus we can put all the values of A in the output array C in constant time.

Now we want to put all the values of B in C. Again we can use the same approach as above i.e., we find the right index of a particular value b_j in B by comparing with values in A. We have to be a bit careful in this case. During the comparison of a value b_j with two consecutive values a_{k-1} and a_k, each processor p_k checks if $a_{k-1} < b_j$ and $a_k \geq b_j$, i.e., processors find index the t, for which $a_{t-1} < b_j$ and $a_t \geq b_j$ holds. This "strict" less inequality is necessary to avoid any concurrent writing or conflicts in C. The processor p_t which found the index t, can decide the final position of a_i as $(j-1)+(t-1)+1$.

Therefore, we can merge the two sorted array of size n into one sorted array in $O(1)$ time using n^2 processors.