Homework 28

Brian Knotten, Brett Schreiber, Brian Falkenstein

November 1, 2018

7

Calculating the result of plugging k into a polynomial of degree n with coefficients $c_1...c_n$ can be done in $\log n$ using $\frac{n}{\log n}$ processors and the following algorithm:

First, make $\frac{n}{\log n}$ copies of k. k is only stored in one memory location, but later in the algorithm, many processors will need to access its value. So first the algorithm will spend $\log n$ time to make $\frac{n}{\log n}$ copies. Let k be originally stored in memory location l_1. Use one processor to read k from l_1 and copy k into l_2. Now use two processors: one to read k from l_1 and copy into l_3, and one to read k from l_2 and copy into l_4. Every constant time interval, the number of memory locations that hold k can double. At the $\log \frac{n}{\log n}$th time interval, $\frac{n}{2 \log n}$ processors double $\frac{n}{2 \log n}$ copies of k to get $\frac{n}{\log n}$ total copies. So copying k for each processor takes $\log \frac{n}{\log n}$ steps.
Next, divide the coefficient list $c_1...c_n$ into $\frac{n}{\log n}$ equal slice. Since there are n coefficients, there are $\log n$ coefficients in each equal slice. Assign every processor to a slice of the coefficient list. There are as many processors as there are partitions of the coefficient list.
Each processor can perform the following task independently and at the same time: Given a unique copy of k and the coefficient list slice $c_i, c_{i+1}, c_{i+2}, ...c_{i+\frac{n}{\log n}}$, compute the number $c_i k^i + c_{i+1} k^{i+1} + c_{i+2} k^{i+2} + ...c_{i+\frac{n}{\log n}} k^{i+\frac{n}{\log n}}$. Each processor will take $\log n$ steps to compute this since adding and multiplying takes constant time, and there are $\log n$ coefficients in each slice of the coefficient list.
At this point there are $\frac{n}{\log n}$ numbers which were computed above. To form the full polynomial result these numbers have to be summed up. $\frac{n}{2 \log n}$ processors can each independently add the pairs of numbers from each adjacent coefficient slice. This will give a result of $\frac{n}{2 \log n}$ numbers which $\frac{n}{2 \log n}$ processors can then add. Every constant time interval, the number of summed numbers halves. At the $\log \frac{n}{\log n}$th time interval, there will be exactly one number left, which is the result of plugging k into the polynomial with coefficients $c_1...c_n$.

This algorithm takes $O(\log n)$ steps, since making $\frac{n}{\log n}$ copies of k takes $\log \frac{n}{\log n}$ steps (less than $\log n$), the coefficient slice computation takes $\log n$ steps, and the summation of the $\frac{n}{\log n}$ slice results takes $\log \frac{n}{\log n}$ steps. This is an EREW algorithm, since k is copied to ensure that no two processors are reading from the same k, and each processor's results are stored independently and summed into one result.
The efficiency of this algorithm, assuming $S(n) = n$, is $E(n, p = \frac{n}{\log n}) = \frac{n}{p * \log n} = \frac{n}{\frac{n}{\log n} \log n} = \frac{n}{n} = 1$.

8

From the hint, we can generalize that:

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Define the matrix:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

The first step of the algorithm is to make n copies of A in memory. This allows all n processors to perform operations on the matrices without violating exclusive read. This copying will take $log(n)$ time, as at the first step, a processor makes 1 copy, then 2 processors use the existing 2 copies to make 2 more, etc. Note that the multiplication of 2x2 matrices is associative, so we can use $n/2$ processors to square $n/2$ copies of A, and store the results somewhere separately from each other in memory. The next step uses $n/4$ processors to square the $n/4$ results from the previous

step. This continues until we are left with the resulting $2x2$ matrix that is A^n. This matrix will have the form:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

The algorithm then writes F_n to the location in memory associated with the answer.
Because each step of the algorithm cuts the number of matrices left to square in half, and we have the same number of matrices to square as n, the number of steps of the algorithm is $log(n)$. Each step takes constant time, as each processor squares their $2x2$ matrix, which simply consists of calculating a constant number of additions and multiplications, which are assumed to take constant time (as per the question). At each step, all the matrix squaring occurs in parallel. Thus, the runtime for this algorithm is $O(log(n) + log(n) + 1) = O(log(n)$.
This algorithm is exclusive read. Because we make n copies of the A matrix at the start, and each processor writes its result to a different location, we ensure exclusive read. Further, the algorithm is exclusive write, since again each processor writes the result of their operation to a separate memory location, and the actual return value isn't set until the end, when a single processor sets it.

25

In a directed acyclic graph, there must exist at least one vertex h which does not have any incoming edges. If h didn't exist, then there would be a cycle, and therefore it wouldn't be an acyclic graph.
Let the algorithm be as follows: Find all vertices in G which do not have any incoming edges and collect them into a set H. (These vertices are operations which do not require any prior operations to be completed before they can start.) Divide H into p equal parts, and let each processor perform $|H|/p$ of the operations. When all the processors complete, remove H from G to produce a smaller DAG. Then, repeat the process above until an empty graph is reached.
Since the longest path in G is size L, the algorithm must iterate exactly L times, because there exists a dependency chain within G of size L, that is, there are a maximum L operations which have to be done in sequence, not parallel. There may be paths in G shorter than L, but since L is larger, it will appear in the big O of this algorithm, and no other paths. On the other hand, in the worst case, even though the processors are all working in parallel, n/p operations still must be done. So the total running time is the time of the longest chain plus $O(L + n/p)$.