CS1510 Parallel Problems 6, 7, & 8

Rebecca Negley, Sean Myers

November 7, 2011

6. We consider the problem of multiplying two n by n matrices.

   - Design a parallel algorithm that runs in time n on a CREW PRAM with n^2 processors. What is the efficiency of this algorithm?

     Solution: Assign each of the n^2 processors to a unique location in the $n \times n$ solution matrix. Any location will be written to only by its assigned processor, so we have no concurrent writing. Each processor goes through n steps to reach its final value. At step i, it reads the ith value in the corresponding row of the first matrix and the ith value of the corresponding column of the secon matrix, multiplies the two values together, and adds their product to the overall sum. Since each of the n^2 processors can do the n steps in parallel, the entire algorithm runs in $O(n)$ time. Its efficiency (using $S(n) = O(n^3)$ from the standard method) is $E(n, n^2) = \frac{n^3}{n^2 * n} = 1$.

   - Design a parallel algorithm that runs in $O(log\ n)$ time on a CREW PRAM with n^3 processors. What is the efficiency of this algorithm?

     Solution: For each of the n^2 unique locations in the $n \times n$ solution matrix, we have n processors to compute the value. In the first step, the ith processor for a particular location will multiply the ith value in the corresponding row of the first matrix with the ith value in the corresponding column of the second matrix. This takes 1 timestep. We then want the sum of the n values computed by the n processors as our final result. Since addition is associative, we can use a divide and conquer approach to add the values together by using half of the n processor to compute the sum of the first half of the numbers and the other half of the processors to compute the sum of the second half of the processors and adding together their two results. This forms a binary recursive call tree of depth $log\ n$ where each level of the recursive call tree takes 1 timestep, so this takes $log\ n$ time. Since each value of the solution matrix can be computed in parallel, the whole algorithm then takes $O(log\ n)$ time. Its efficiency is $E(n, n^3) = \frac{n^3}{n^3 * log\ n} = \frac{1}{log\ n}$.

   - Design a parallel algorithm that runs in $O(log\ n)$ time on a CREW PRAM with $n^3/log\ n$ processors. What is the efficiency of this algorithm?

     Solution: This algorithm will be very similar to the one above. For each of the n^2 locations in the solution matrix, we assign $n/log\ n$ processors. For the first $log\ n$ steps, each processor computes, for some i, the product of the ith value of the corresponding row of the first matrix with the ith value in the corresponding column of the second matrix. Since $n/log\ n$ of these values can be computed in parallel in one timestep by $n/log\ n$ processors, computing all of the products takes $n/(n/log\ n) = log\ n$ time. Then, we must sum these n values together. In the next step, each processor is assigned to one of the product values. We then add the value at a processor to some product that has not yet been added yet. We do this for $log\ n$ steps until all of the products have been added to one processor. We then sum all of the processors values together as in the algorithm above by having one processor compute the sum of its value and the value of one other (using half as many processors as the step before) until we reach our final sum. This takes $log(n/log\ n) = O(log\ n)$ time. Hence, we reach our final value in $O(log\ n)$ total time. The efficiency of this algorithm is $E(n, n^3/log\ n) = \frac{n^3}{n^3/log\ n*log\ n} = 1$.

- Design a parallel algorithm that runs in $O(log\ n)$ time on a EREW PRAM with $n^3/log\ n$ processors. What is the efficiency of this algorithm?

  Solution: Observe that, in the algorithm above, each of the values in the first matrix was read by one processor for every location in the corresponding row of the solution matrix, so it was read n times. Similarly, the values in the second matrix were read by one processor for every location in the corresponding column of the solution matrix, again n times. If we can first copy each value into an n-sized array in time $O(log\ n)$, we can then run our algorithm above and the whole process will take $O(log\ n)$ time. This is precisely what we gave in the second part of problem 3. We will first copy the values in the first matrix. Since there are n^2 values, we have $n/log\ n$ processors to copy each value. First, one processor reads the value. Then, at each step, each processor that knows the value passes it on to one other processor until all $n/logn$ processors know the value, taking $log(n/log\ n) = O(log\ n)$ time. We then write the values to each location of the array using another $log\ n$ steps. We do the same for the second matrix and then follow the algorithm above. Instead of reading from the original matrix locations, we now read from the array location corresponding to the row or column value (for the first or second matrix, respectively) of the solution matrix. This way, no memory location is accessed concurrently. Hence, we have a $O(log\ n)$ algorithm for an EREW PRAM. Its efficiency is $E(n, n^3/log\ n) = \frac{n^3}{n^3/log\ n} = 1$.

7. Design a parallel algorithm that given a polynomial $p(x)$ of degree n and an integer k computes the value of $p(k)$. Your algorithm should run in time $O(log\ n)$ on a EREW PRAM with $O(n/log\ n)$ processors. Assume that the polynomial is represented by its coefficients.

   Solution: First, we must compute $\{k^0, k^1, \ldots, k^n\}$. We will begin by computing k^0, k^1, k^2, $\ldots$,$k^{nlog\ n}$. We assign k^0 to 1 and read k and assign the value to k^1 in the first step. The second step computes k^2. In the third step, we compute k^3 and k^4 by multiplying $k^1 * k^2$ and $k^2 * k^2$. We continue this pattern where, at the $i+1$st step, we compute $k^{2^{i-1}+1}, \ldots, k^{2^i}$. Each value needs to be read by at most three processors, so we can sequentialize those reads and still keep constant time at each step. To compute k^j, if j is even, we only need to read $k^{j/2}$. If j is odd, we must read $k^{(j-1)/2}$ and $k^{(j+1)/2}$. Then, if $p \leq 2^{i-1}$ and we are computing $k^{2^{i-1}+1}, \ldots, k^{2^i}$, we must read k^p to compute values k^{2p-1}, k^{2p}, and k^{2p+1} (if $2p-1 > 2^{i-1}$, $2p > 2^{i-1}$, and $2p+1 > 2^{i-1}$ respectively). Hence, we can compute $k^{2^{i-1}+1}, \ldots, k^{2^i}$ in constant time as long as we have one processor to compute each value. If we stop when we reach $k^{n/log\ n}$, we will be running each step in constant time. (Since there are only $n/log\ n$ values to compute, there must be a sufficient number of processors to compute each step in constant time.) Since the information doubles at each step, we will run $O(log(n/log\ n)) = O(log\ n)$ steps. Hence, we compute k^0, k^1, k^2, $\ldots$,$k^{n/log\ n}$ in $O(log\ n)$ time. It will also be useful to fill an n-sized array with the value $k^{n/log\ n}$ using the method from 3(b) (described again briefly in 6(b)). We only need an $n/log\ n$ sized array, but the n-sized array will work and has already been described to run in $O(log\ n)$ time.

   Now, given $k^{(i-1)*n/log\ n+1}$, $k^{(i-1)*n/log\ n+2}$, $\ldots$,$k^{i*n/log\ n}$ and an array with at least $n/log\ n$ copies of $k^{n/log\ n}$, we can compute $k^{i*n/log\ n+1}$, $k^{i*n/log\ n+2}$, $\ldots$,$k^{(i+1)*n/log\ n}$ in constant time. Each processor multiplies one of the values from the first set by one of the $k^{n/log\ n}$ copies to give one of the values in the new set. This must be done $log\ n$ times to reach k^n. Hence, in total $O(log\ n)$ time we have k^j for all j values.

Now, we must compute each $a_i * k^i$ value. Since each value only needs to be accessed for one computation, we do not have to worry about concurrent reads. We have each processor do one of these computations at each step, so after $log\ n$ steps, we have completed all $n/log\ n * log\ n = n$ computations. We then must sum our n values together. We have already seen how to do this in $O(log\ n)$ time. I will not repeat the algorithm, but it is described briefly at the end of 6(c). Hence, in $O(log\ n)$ steps, we computed all k^j values, multiplied them by the appropriate coefficients, and added all products together. Hence, we have computed $p(k)$ in $O(log\ n)$ steps. Since we could evaluate the polynomial sequentially in n time using Horner's method, this algorithm's efficiency is $E(n, n/log\ n) = \frac{n}{n/log\ n * log\ n} = 1$.

8. We consider the problem of computing F_n, the nth Fibonacci number, given an integer n as input. Show how to solve this problem in time $O(logn)$ on a EREW PRAM with n processors. Make the unrealistic assumption that F_n will fit within one word of memory for all n that is assume that all arithmetic operations take constant time. Recall that F_n is defined by the following recurrence: $F_0 = F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$.

   Solution: The first thing we need to sort out before we continue is how we go about "dividing and conquer"-ing in this algorithm. To start, let's define A as a matrix: $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ and f as: $\begin{bmatrix} f_j \\ f_{j-1} \end{bmatrix}$ where $A * f_j = f_{j+1}$ where $f_{j+1} = \begin{bmatrix} f_{j+1} \\ f_j \end{bmatrix}$. Let's say we wanted to find f_{j+2} from f_j. Well, we can see that $A * f_j = f_{j+1}$ and $A * f_{j+1} = f_{j+2}$. Another way to see that is: $A * (A * f_j) = f_{j+2}$. This pattern continues for any $f_{j+n} = A^n * f_j$.

   If we want to find f_n, we can relate it to the discovery before, except f_j is f_1, which we know is $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and to find f_n this way, we will need to matrix multiply A^n matrices (let's assume that multiplication is constant). This sounds like it would take 4n operations(to multiply n matrices, but we can also note that if we are calculating f_1000, we will have A^1000. A^1000 is equivalent to: $A^500 * A^500$ Also, $A^500 = A^250 * A^250$. This pattern continues onward where if it is even, the multiplication will be $A^n = A^{n/2} * A^{n/2}$ or if A is odd, then it is $A^n = A^{(n-1)/2} * A^{(n-1)/2} * A$. Since n is being halved each time, there will be a depth of log(n) recursive calls to this, if it was a function. Going back up through the recursive call tree, there is roughly 2-13 multiplications per node(13 if it is odd), we will just say it is 13 multiplications. So the recursive definition for finding A^n is: $rec(n) = rec(n/2) + 13$ where the +13 is the multiplication of the recursive function, and the possible multiplication of another A matrix. This will break down to a time of: $log(n) + 13n$.

   If our goal is to make an algorithm that will run in log(n) time, all we have to do is parallelize the 13n. We can write the code likes this:

```
fib(n,p):
    if(n/2 is odd):
        res = parcall(fib(n/2, p/2))
        results = res*res*A (use p processors to make it 1 operation)
    if(n/2 is even):
        res = parcall(fib(n/2, p/2))
        results = res*res* (use p processors to make it 1 operation)
```

   By using