Homework 27

Brian Knotten, Brett Schreiber, Brian Falkenstein

October 30, 2018

1

1.1

A parallel algorithm for AND with $p = n$ is given:

```
P_And(x_1 ... x_n, p):
   if p == 1:
       return and(x_1...x_n)
   else:
       return and(P_And(x_1...x_(n/2), p/2), P_And(x_(n/2+1) ... x_n, p/2))
```

In the case where $p = n$, the algorithm will cascade down until each processor has a single x value (n values, n processors, each gets one). After this step, the ands will cascade upwards, with one processor anding the result of 2 processors. Thus, at the first time stamp, all n processors are utilized (although, they do not really do anything, they just return the and of the single value they have), at the second time stamp, $n/2$ processors are utilized, at the third time $n/4$ processors, etc.

NEED TO SPECIFY WHY ITS EREW

The efficiency is clearly bad. Given the equation for efficiency:

$$E(n, p) = \frac{S(n)}{pT(n, p)}$$

Clearly, $S(n)$ will be n, as it takes n time to and an input of size n (just and the first two, then the result of that with the third, etc.). We can further define $T(n, p)$, the recurrence relation, as:

$$T(n, p) = T(n/2, p/2) + 1 = log(n)$$

This recurrence relation is true because at each step, we are simply doing the and of 2 elements, which takes constant time. Further, these ands happen simultaneously. Thus, we can compute the efficiency as:

$$E(n, p) = \frac{S(n)}{pT(n, p)} = \frac{n}{nlog(n)} = \frac{1}{log(n)}$$

For large n values, this efficiency is rather bad.

Using the folding principle, which states:

$$T(n, p) <= kT(n, kp)$$

We can see that if we have $p = n^{1/3}$ processors instead of n, we'd get an upper bound on the running time of:

$$T(n, p) <= n^{2/3}T(n, n^{1/3})$$

$$THISISNOTRIGHT!!!!$$

IE, reducing the number of processors from n to $n^{1/3}$, at most increases the run time by $n^{2/3}$.

1.2

The parallel algorithm for AND with $p = n/log(n)$ is the same as the algorithm when $p = n$. The difference here is that at the base level, where in the last algorithm each processor had 1 value to pass up, each processor will have $log(n)$ values that it must sequentially and. This is because the problem is split into equal size sub problems, in this case n/p, IE each processor initially gets a subproblem of size n/p. In the case where $p = n$, $n/p = 1$. However, when $p = n/log(n)$, plugging in, we get $n/p = log(n)$. That means, instead of the first step of the algorithm taking constant time, it now takes $log(n)$, as all the processors sequentially and their $log(n)$ values. Beyond this, the algorithms function almost identically, however this case will spend less time cascading answers up than the one with $p = n$. This leads us to a recurrence relation of:

$$T(n, p) = T(n/2, p/2) + log(n) = log^2(n)$$

And an efficiency of:

$$E(n, p) = \frac{S(n)}{pT(n, p)} = \frac{n}{(n/log(n))(log^2(n))} = \frac{1}{log(n)}$$

Again using the folding principle, we can set an upper bound on the run time for this algorithm if we reduce the number of processors from $p = n/log(n)$ to $p = n^{1/3}$. We note that the difference in number of processors here is:

$$\frac{n}{log(n)} - n^{1/3}$$

NEED TO FIND THIS DIFFERENCE AND PLUG INTO FOLDING PRINCIPLE

1.2

Without the restriction of an EREW machine, and with n processors, we can find an algorithm for AND that takes constant time. In the first time step, the algorithm copies 1 of the input data into each processor. Then, each processor executes the following code:

```
processorAnd(x1):
    if x1 == 0:
        ans = 0
```

Where ans is some variable that all processors have write access to, and is initialized to be 1. Thus, in the second time stamp, each processor checks their value, and either does nothing if they have a 1, or they write a 0 if they have a 0. The solution will then be stored in ans, which will be 1 if all the inputs are 1 (no processor overwrites it with a 0), or 0 if even one if the inputs is 0. Because we simply have a constant number of operations that each take constant time, this algorithm is O(1).
NOT SURE HOW TO DEFINE RECURRENCE RELATION IN THIS INSTANCE

3

3.1

The EREW algorithm with $p = n$ is defined:

```
createArray(n, x, p):
    if p == 1:
        A = new Array[n]
        for i in range(0, len(A)):
            A[i] = x
        return A;
    else:
        return concat(createArray(n/2, x, p/2), createArray(n/2, x, p/2))
```

That is, with $p = n$, at the first time stamp, each processor creates an 'array' of size 1 and sets its value at index 0 to be x. Then, cascading up the call tree, each processor concatenates the arrays from 2 processors, creating a larger array. So at time 2, some processor will combine 2 size 1 arrays to create a size 2 array. This continues until

the final array of size n is constructed. The first step of the algorithm takes constant time, as each processor simply creates a new array object and stores a value in it (and they all do this concurrently). It then takes $log(n)$ time to cascade upwards, as at each time step, the processors cut the number of array concatenations left to do in half. This leads us to a recurrence relation of:

$$T(n,p) = T(n/2, p/2) + 1 = log(n)$$

As at each step, the algorithm cuts the number of arrays left to concatenate in half, and each step takes constant time. Thus, with $p = n$, we get efficiency:

$$E(n,p) = \frac{S(n)}{pT(n,p)} = \frac{n}{nlog(n)} = \frac{1}{log(n)}$$

Similarly to problem 1, if we decrease the number of processors from n to $n^{1/3}$, by the folding principle, we get an upper bound of:

6

6.1

In a matrix multiplication $A \times B = C$, every entry of C is dependent on a row of A and a column of B, but is independent from every other entry in C. Therefore all of the entries of C can be computed independently at the same time.

With n^2 processors (assuming A and B are both $n \times n$ square matrices) C can be computed in n time with the following CREW algorithm:

MatrixMultiplicationAlgorithm(A, B):
    for each processor $p_{i,j} \in (n \times n)$, concurrently:
        let $C_{i,j} = 0$
        for $k = 1$ to n do:
            update $C_{i,j} += A_{i,k} * B_{k,j}$
    Output C

$T(n, p = n)$This algorithm takes n time because it makes n multiplication and addition steps at the same time. It is an Exclusive Write algorithm because each processor only writes to one, distinct cell of C. It is a Concurrent Read algorithm because processors assigned cells in the same row in C will both read from the same row in A, and processors assigned cells in the same column in C will both read from the same column in B.

The efficiency of this algorithm is: $E(n, p = n^2) = \frac{S(n)}{p*T(n,p=n^2)} = \frac{n^3}{n^2*n} = 1$.

By the folding principle, the upper bound of the running time for this algorithm on $n^{1/4}$ processors is (TIME) i.e. $T(n, p = n^2) \leq \frac{1}{n^{7/4}} \cdot T(n, p = n^{1/4})$

6.2

With n^3 processors (assuming A and B are both $n \times n$ square matrices) C can be computed similarly to the algorithm in 6.1: each of the n^2 entries of C will use n processors to compute the n products $A_{i,k} * B_{k,j}$ required for the entry $C_{i,j}$ concurrently in constant time. Then, for each entry, the n products are summed in $log(n)$ time, as at each time step the number of sums being calculated is halved.

The efficiency of this algorithm is: $E(n, p = n^3) = \frac{S(n)}{p*T(n,p=n^3)} = \frac{n^3}{n^3*log(n)} = log(n)$

By the folding principle, the upper bond of the running time for this algorithm on $n^{1/4}$ processors is (TIME) i.e. $T(n, p = n^3) \leq \frac{1}{n^{7/4}} \cdot T(n, p = n^{1/4})$

6.3