# 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:

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

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:

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