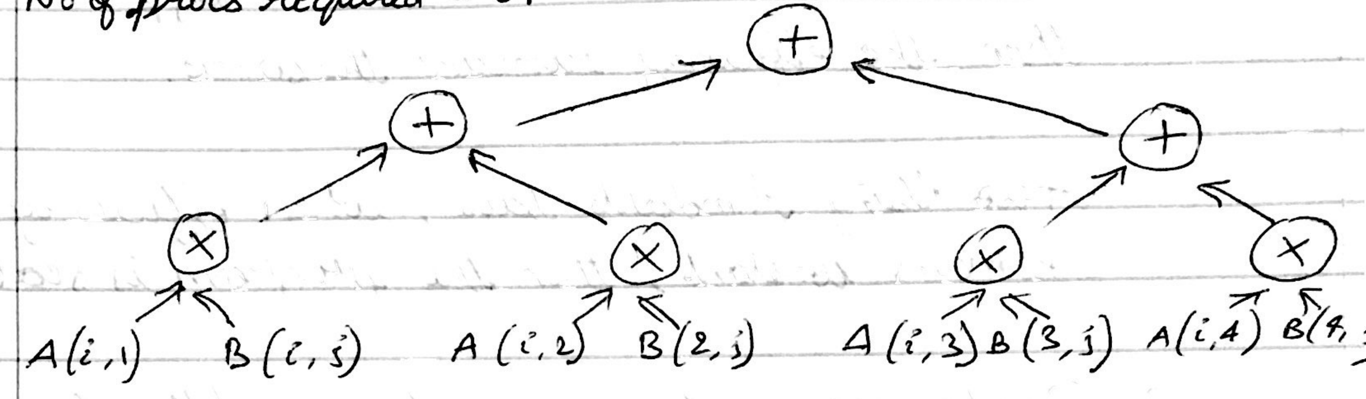
1) i) DAG for $n = 4 = 2^2$.

No of procs required = 64



Algorithm -

Input - Two $n*n$ matrices A & B, $n = 2^k$.

Processor $P_{i,j,l}$ computes $A(i,l)\,B(l,j)$ in a single step. Processors are denoted by $P_{i,j,l}$.

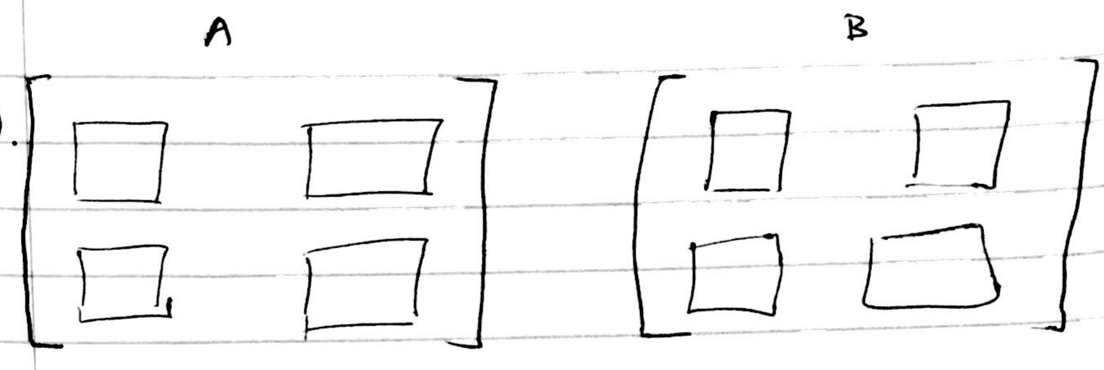
```
function matrixMultiply (Matrix A, Matrix B)
{
    Compute C'(i,j,l) = A(i,l) B(l,j).
    for h = 1 to log n
        if ( l ≤ n/2^h )
            C'(i,j,l) = C'(i,j,2l-1) + C'(i,j,2

    if (l = 1)
        C(i,j) = C'(i,j,1).
}
```

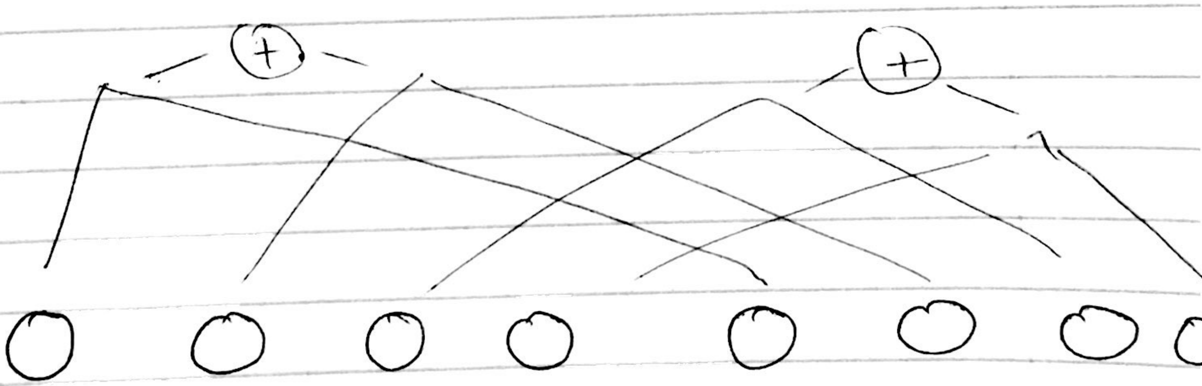
since n^3 operations.

Depth = $O(\log n)$ Parallel work = $O(n^3)$.

1(b).



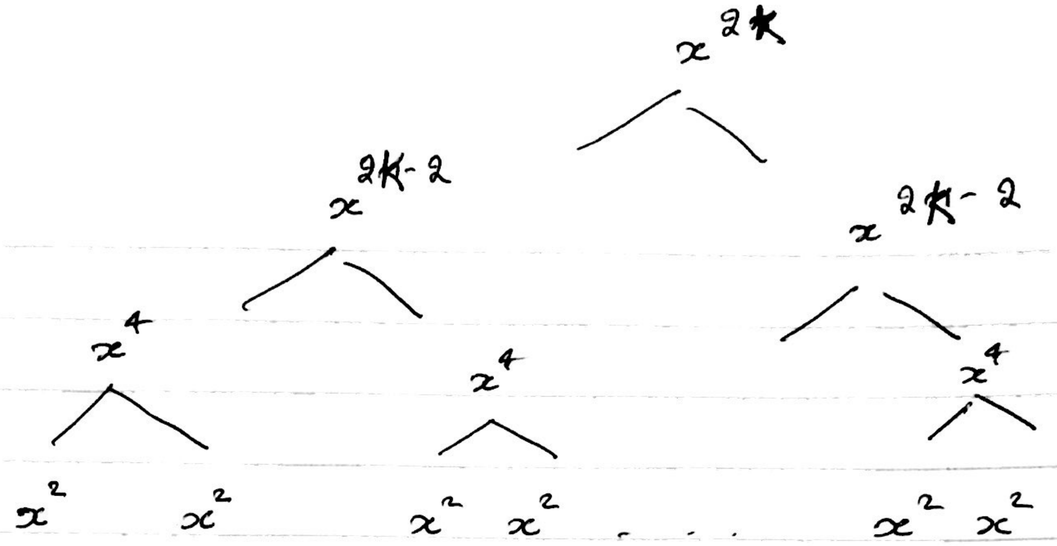
Partition matrices A & B in square blocks of P.
P is the num of processors available.
Create matrix of processes of size $p^{1/2} * p^{1/2}$ so that each process can maintain a block of A matrix & a block of B matrix.
Each block is sent to each process, & the copied sub blocks are multiplied together and the results added to the partial results in C sub blocks



$W = O(n^3)$. Depth $= O\left(\frac{n}{\sqrt{p}}\right)$.

Since n is divided in $\sqrt{p}$ blocks of matrices.

2).



Consider $n = 2^k$.

Depth =

$n = 2^k$.

The above DAG computes $x^2.(x^2)$ at each level.

By ~~doing~~ repeatedly multiplying x^2 log k times.
we can get ~~log~~ n.

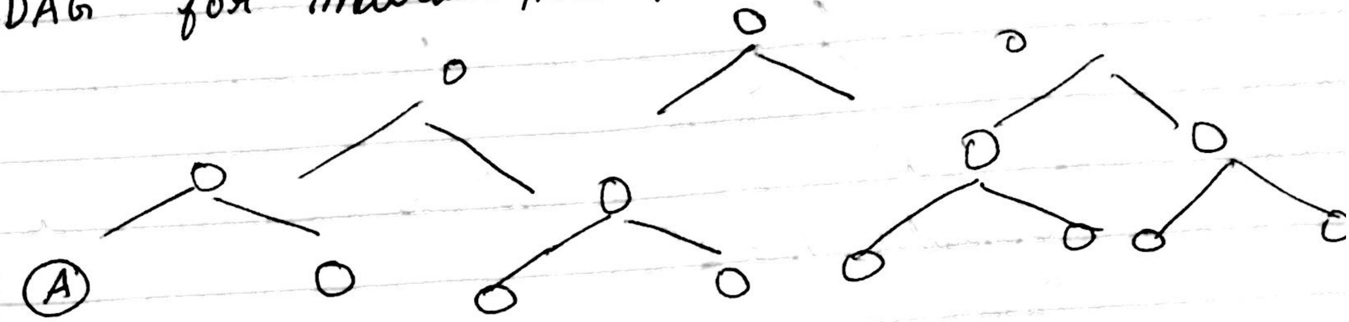
This looks like a balanced binary tree.

Hence Depth = $\log_2 k$

Work = $O(k)$.

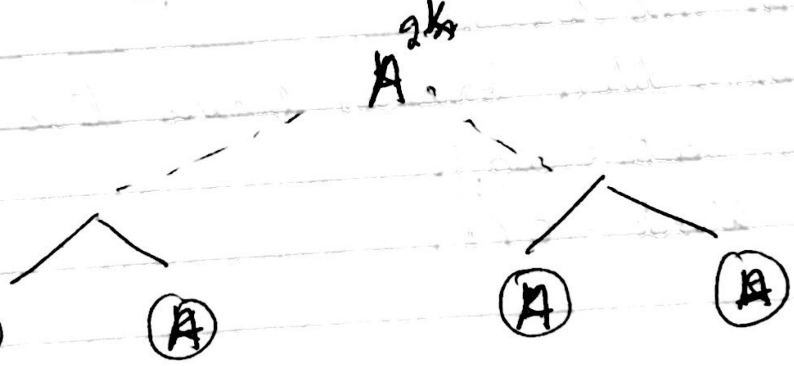
2(b). DAG for matrix multiplication $(m * m)$.

1



Depth = $\log m$. Work = m^3.

Consider 1 as A and repeatedly multiply.



Depth = $\log k$. Work = $O(k)$.

Hence total work = $O(k) \cdot O(m^3)$.

Total depth = $\log m \cdot \log k$.

3. Divide the input array into p different equal blocks. Add the elements in each block with each processor.

Here work = $O(n)$ time = $O(n/p)$.

The top most reduction results are [illegible] in another array and same splitting & addit[illegible] of blocks is done.

$$Time_p = \cancel{O(\log_p N)}\, O(n/p + \log_p N)$$

$$Speedup = \frac{T_{serial}}{T_{parallel}} = \frac{N}{\frac{N}{p} + \log_p N}$$

$$\frac{n - \log_p n}{p} + \log_p n$$

$$\left[T_p(n) \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + D(n) \right]$$

Brent's theorem

4). $T_{serial} = n$. $T_{parallel} = n/p + \log(p)$

$$\text{Efficiency} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

1 $$E = \frac{n}{p\left[\frac{n}{p} + \log p\right]} = \frac{n}{n + p\log p}$$

Suppose p increases by factor k and E is constant. Then. Let n be n'.

$$E = \frac{n'}{n' + kp\log kp} \quad (2).$$

Substitute (1) in (2).

$$\frac{n}{n+p\log p} = \frac{n'}{n'+kp\log kp}$$

$$\cancel{nn'} + n.kp\log kp = \cancel{n'n} + n'p\log p$$

$$n' = n.\frac{kp.\log kp}{p\log p}$$

$$= n.\left(\frac{k\log kp}{\log p}\right)$$

Scanned by CamScanner

If n increases by a factor of $\frac{K \log Kp}{\log p}$.

then the efficiency remains the same.

~~Acc~~ Using Amdahl's law, when efficiency remains constant, then the program is scalable.

3). Algorithm - Take the sums of adjacent pairs of A
The size of A' is exactly half size of A. If size of A is not power of 2, pad it with 0's.
Input: Array A with $n = 2^k$ elements, Output: prefix sums $1 \leq i \leq n$.
function prefix sum (Array A).

if (n == 1)
    return A[0].
~~for $1 \leq i \leq n/2$.~~ //A' be the sum of adjacent pair.
~~$R' =$ prefix sum (A').~~ //recursive call
for $1 \leq i \leq n/2$.
    Set $y_i = x_{2i-1} + x_{2i}$.
prefix sum(Y).

for $1 \leq i \leq n$.
{
    i even : set $S_i = z_{i/2}$
    $i = 1$: set $S_i = x_i$.
    i odd > 1 : set $S_i = z_{(i-1)/2} + x_i$ }