1. Consider a 3D stack of blocks; each block is a cube given as eight [x, y, z] coordinates. Assume that all cubes are the same size. A configuration is valid if:

- **There is at least one block on the "floor" (i.e., the z axis is zero for one of its faces)**
- **Every other block is lying exactly on top of another block, such that there is a path to a floor block (i.e., the blocks are stacked, not floating in mid-air).**

Let isValid(A) be a function that takes in an n x 8 x 3 array and outputs whether or not it consists of a valid configuration.
(a) Write pseudocode for a sequential version of isValid
(b) Write pseudocode for a parallel version of this algorithm. Assume that you have as many processors available as needed.
(c) Analyze (i.e., prove) the running time of both algorithms and the work of the parallel one
(d) Extra credit: Assume that the cubes can have different sizes. Write pseudocode for a parallel algorithm that can solve this version of the problem. Here, a cube is considered supported if it lies completely on top of another one (i.e., a big cube cannot lie on top of a smaller one).

Solution:

a. Sequential Implementation

Sequential Implementation

```
Algorithm isValid
begin
    cubeList := cubes
    for cube in cubes do
        zeroZFace := 0
        for vertex in vertices do
            zeroZCoordinate = checkVertex(vertex)
            if zeroZCoordinate then
                zeroZFace <- zeroZFace + 1
        if zeroZFace = 4 then
            floorCubeCount <- floorCubeCount + 1
            floorCubeList <-: cube
            cubeList.remove(cube)
    if floorCube > 0 then
        result = checkForPathToFloor(cubeList, floorCubeList)
    else then
        result = false
    return result
end

Algorithm checkForPathToFloor
begin
    for floorCube in floorCubeList do
        for cube in remainingCubeList do
            result = exactlyOnTop(floorCube, cube)
            if !result then
                return false
            else then
                continue
    return true
end

Algorithm exactlyOnTop
begin
    vertexMatch := 0
    for vertex in cubeVertices do
        if x_coordinate_floorCube = x_coordinate_cube AND
            y_coordinate_floorCube = y_coordinate_cube then
            vertexMatch <-: vertexMatch + 1
    if vertexMatch = 4 then
        return true
    else then
        return false
end

Algorithm checkVertex
begin
    if z_coordinate = 0 then
        return true
    else then
        return false
end
```

(b) Parallel Implementation

```
Parallel Implementation

Algorithm isValid_parallel
begin
    cubeList := cubes
    /* Parallel Step */
    for cube in cubes pardo
    /* Parallel Step */
        zeroZFace := 0
        for vertex in vertices do
            zeroZCoordinate = checkVertex(vertex)
            if zeroZCoordinate then
                zeroZFace <- zeroZFace + 1
        if zeroZFace = 4 then
            floorCubeCount <- floorCubeCount + 1
            floorCubeList <-: cube
            cubeList.remove(cube)
    if floorCube > 0 then
        result = checkForPathToFloor(cubeList, floorCubeList)
    else then
        result = false
    return result
end

Algorithm checkForPathToFloor_parallel
begin
    /* Parallel Step */
    for floorCube in floorCubeList pardo
    /* Parallel Step */
        for cube in remainingCubeList pardo
            result = exactlyOnTop(floorCube, cube)
            if !result then
                return false
            else then
                continue
    return true
end

Algorithm exactlyOnTop_parallel
begin
    vertexMatch := 0
    /* Parallel Step */
    for vertex in cubeVertices pardo
        if x_coordinate_floorCube = x_coordinate_cube AND
            y_coordinate_floorCube = y_coordinate_cube then
            vertexMatch <-: vertexMatch + 1
    if vertexMatch = 4 then
        return true
    else then
        return false
end

Algorithm checkVertex_parallel
begin
    if z_coordinate = 0 then
        return true
    else then
        return false
end
```

(c) Analysis of running time of both and work of parallel one

1. Running time of sequential algorithm

   $O(n^3) – O(n^2)$ {for isValid} + $O(n^3)$ {for checkForPath} + $O(n)$ {exactly on top} + $O(1)$ {checkVertex}

2. Running time of Parallel algorithm

   $O(n^2) – O(n)$ {for isValid} + $O(n^2)$ {for checkForPath} + $O(1)$ {exactly on top} + $O(1)$ {checkVertex}

3. Work done by the parallel algorithm

   The work done by a parallel algorithm would be around $2n + 2$ operations. This is because, first assignment of cube to processor takes 1 operation, calculating the zero Z vertex takes n operations. Assignment of floor cubes and remaining cubes to processors take 1 operation again (Since number of floor cubes + number of remaining cubes = total number of cubes = total number of processors). Now to check if the cubes are exactly on top, it'll approximately be ~n.

   So total number of operations are: $1 + n + 1 + n = 2n + 2$

(d) Cubes can have different sizes

```
Parallel Implementation

Algorithm isValidDifferentSizes_parallel
begin
    cubeList := cubes
    /* Parallel Step */
    for cube in cubes pardo
    /* Parallel Step */
        zeroZFace := 0
        for vertex in vertices do
            zeroZCoordinate = checkVertex(vertex)
            if zeroZCoordinate then
                zeroZFace <- zeroZFace + 1
        if zeroZFace = 4 then
            floorCubeCount <- floorCubeCount + 1
            floorCubeList <-: cube
            cubeList.remove(cube)
    if floorCube > 0 then
        result = checkForPathToFloor(cubeList, floorCubeList)
    else then
        result = false
    return result
end

Algorithm checkForPathToFloor_parallel
begin
    sortedFloorCubeCoordinates := sortCubeCoordinates(floorCubeList)
    sortedRemainingCubeCoordinates := sortedCubeCoordinates(cubeList)
    /* Parallel Step */
    for floorCube in floorCubeList pardo
    /* Parallel Step */
        for cube in remainingCubeList pardo
            result = exactlyOnTop(sortedFloorCubeCoordinates, sortedRemainingCubeCoordinates)
            if !result then
                return false
            else then
                continue
    return true
end

Algorithm exactlyOnTop_parallel
begin
    vertexMatch := 0
    /* Parallel Step */
    for vertex in cubeVertices pardo
        if x_coordinate_floorCube > x_coordinate_cube AND
            y_coordinate_floorCube > y_coordinate_cube then
            vertexMatch <-: vertexMatch + 1
    if vertexMatch = 4 then
        return true
    else then
        return false
end

Algorithm checkVertex_parallel
begin
    if z_coordinate = 0 then
        return true
    else then
        return false
end

Algorithm sortCubeCoordinates
begin
    /* Sort based on x,y,z coordinates */
    return sortedCoordinates;
end
```

2. Suppose that two n x n matrices A and B are stored on a mesh of n^2 processors such that Pi;j holds A[i; j] and B[j; i]. Write pseudocode for an asynchronous algorithm that computes the product of A and B in O(n).

Solution:

Systolic array is a way of realizing the matrix multiplication algorithm with n^2 processors and O(n) time complexity, by (i) placing the n^2 processors in square (n × n), and (ii) assigning the computation of I(i, j), A(i, j), and O(i, j) to the (i, j)-th processor.

Algorithm: Systolic Array

```
procedure SYSTOLIC (W, X)
        I0 = [(i, j) -> 0]
        A0 = [(i, j) -> 0]
        O0 = [(i, j) -> 0]
        SYSTOLICINNER (W, X, 0, I0, A0, O0)
End procedure

procedure SYSTOLICINNER (W, X, k, I, A, O)
        if k = 3n – 1 then
                return 0
        end if
        I’ = [(i, j) -> (i = 0) ? X (k - j, j) : I(i – 1, j)]
        A’ = [(i, j) -> ((j = 0) ? 0 : A(i, j – 1)) + (W(i, j) x I’(i, j)))]
        O’ = O[∀i. (i, k – n - i) -> A(i, n - 1)]
        SYSTOLICINNER(W, X, k + 1, I’, A’, O’)
end procedure
```

Explanation:

1. First initialize the I_0, A_0, O_0 to an n x n zeros matrix. These matrices represent the intersections, multiplications and additions involved in the computations.
2. k represents the number of iterations the algorithm take for completion. It is always equal to 3n – 1. Suppose that we have a 4 x 4 matrix, there are 11 intersect, multiplicate and accumulate operations.
3. $[(i,j)\mapsto 0]$ this means an assignment of value zero to the element (i, j) of the matrix
4. $I' = [(i, j) \mapsto (I = 0) ? X(k - j, j) : I(i - 1, j)]$ Means that For any element (i, j) of matrix I', if the row index i = 0, then you assign the value of X(k − j, j) to I'(i, j). If not, you assign I(i − 1, j) to I′(i, j).
5. I′ is a temporary matrix that collects the elements of X that are at the intersection of the two matrices (elements ready for performing operations on them). Thus, the relevant input of X is denoted by I′.
6. A′ is an accumulator. Each element W(i, j) is being multiplied by each element of I′(i, j). Recall that I′(i, j) is obtained from X. Thus, we are multiplying X and W but in a different manner. O′ is the output matrix. It is moving the accumulated values of A′ to their correct positions (as if we were multiplying matrices in the standard manner).

Parallel Algorithms

Systolic matrix multiplication:

$A = n \times n$ $B = n \times n$ matrix

n^2 processors ; A_{ij} & B_{ji} are stored in P_{ij}

$$A = \begin{pmatrix} 1 & 0 & 3 & 4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

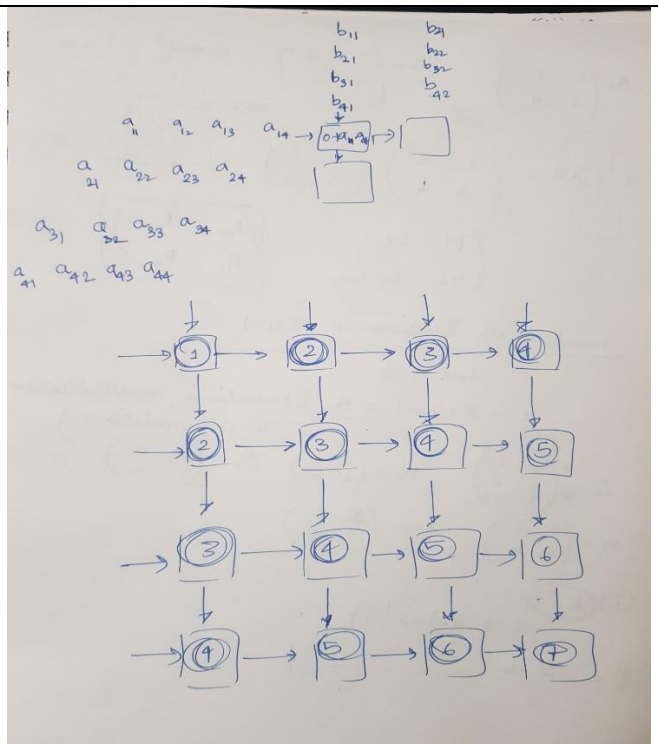
$$A \times B = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

Systolic implementation:

P_{ij}

i) b_{41}

a_{14} | 0 |



$A_2 \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ $B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$ $n = 2$

2×2

$$C = AB = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \quad \begin{matrix} 1 \\ 15 \\ 28 \\ 43 \end{matrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} \qquad \begin{pmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{pmatrix}$$

number of iterations $K = 3n - 1$

Here $n = 2$

$K = 3(2) - 1 = 5$ (intersections, multiplications & Accumulations)

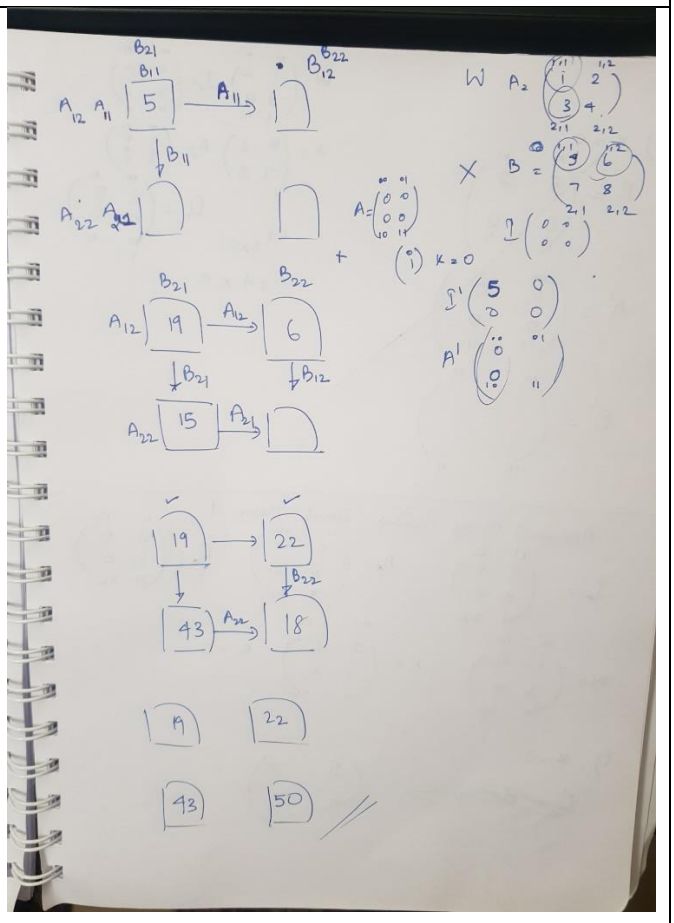
$I_0 \Rightarrow \begin{pmatrix} (0,0) & (0,1) \\ (1,0) & (1,1) \end{pmatrix}$ $A_0 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$ $O_0 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$

$W = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ $X = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$

(i) $K = 0$

$0 \leq 3(2) - 1 \{5\}$;

$I' = 5 \; 6$



3. We discussed the WT scheduling principle in the context of PRAM algorithms. Prove that this principle will always work in the dag model.

Solution:
In graph theory, a graph is a series of vertices connected by edges. In a directed graph, the edges are connected so that each edge only goes one way. A directed acyclic graph means that the graph is not cyclic, or that it is impossible to start at one point in the graph and traverse the entire graph. Each edge is directed from an earlier edge to a later edge.

Scheduling the task includes dividing the tasks into smaller subtasks. Dependencies among these subtasks are represented using Directed acyclic graph (DAG). DAGs helps in finding an optimal solution for the multiprocessor task scheduling problem. Directed Acyclic Graph (DAG), In multiprocessor system, tasks are divided into several subtasks for parallel execution. To represent data dependencies and communication time between tasks, directed acyclic graphs are used.

Work-time Model:
The complexity is measured in terms of two costs: work and time. Roughly speaking the work corresponds to the total number of operations we perform, and span to the longest chain of dependencies.

Steps for the work time in Dags:
1. Given an input, dynamically create a DAG
2. Nodes represent the sequential computation – weighted by the amount of work
3. Edges represent the dependencies

For an expression e, W(e) to indicate the work needed to evaluate that expression and T(e) to indicate the time.

Example:
In the expression e1 + e2 where e1 and e2 are themselves other expressions (e.g. Function calls) we could run the two expressions in parallel giving the rule

$$T(e1 + e2) = 1 + \max(T(e1), T(e2))$$

This rule says the two subexpressions run in parallel so that we take the max of the span of each subexpression. But the addition operation must wait for both subexpressions to be done. It therefore must be done sequentially after the two parallel subexpressions and hence the reason why there is a plus 1 in the expression $1 + \max(T(e1), T(e2))$.

Representing parallel computations as DAG's:
We can represent a parallel computation with a DAG (directed acylic graph) by recursively composing the DAG's of subcomputations either in sequential or in parallel form.

Fig: Sequential (left) and parallel (right) composition of two evaluated expressions. Both the DAGs correspond to the DAGs of the subexpression

Example:
Problem Statement: Given a list of integers, we want to find the sum of all prefixes of the list, i.e. the running sum. We are given an input array A of size n elements long. Our output is of size n + 1 elements long, and its first entry is always zero.

Algorithm for sum of n values using n processors (i)
Each processor I, 1 <= i <= n, executes:
Input: A[1, … , n], $n = 2^k$
Output: sum $S = \sum_{j=1..n} A[j]$
begin
    for 1 <= i <= n **pardo**
        B[i] := A[i]
    for h = 1 **to** log n **do**
        for 1 <= i <= $n/2^h$ **pardo**
            B[i] := B[2i – 1] + B[2i]
    if i = 1 **then**
        S := B[1]
end

Argument:
1. How much time? – O(logn)
2. How many operations? – O(n)

WT Scheduling principle – total time <= O(n/p + logn)

Proof – WT Scheduling principle always works in DAG:
1. A schedule of a task system G = (V, E, w) is a function $s : V \rightarrow N *$ such that s(u) + w(u) <= s(v) whenever e = (u, v) ∈ E.
2. Let G = (V, E, w) be a task system. There exists a schedule if and only if G contains no cycle.
3. In other words, a schedule must preserve the dependence constraints induced by the precedence relation ≺ and embodied by the edges of the dependence graph; if u ≺ v, then

the execution of u begins at time s(u) and requires w(u) units of time, and the execution of v at time s(v) must start after the end of the execution of u.

4. Clearly, if there is a cycle $v_1 \rightarrow v_2 \ldots \rightarrow \ldots \rightarrow v_k \rightarrow v_1$, then $v_1 \prec v_1$ (v_1 depends on itself), and a schedule s would satisfy $s(v_1) + w(v_1) <= s(v_1)$, which is impossible because we assumed $w(v_1) > 0$ (w(v1) represents work done at the vertex v1)
5. Conversely, if G has no cycle, then there exist vertices with no predecessor (V is finite) and we can thus topologically sort the vertices and schedule them one after the other (i.e., assuming a single processor) according to the topological order.
6. Formally, if $v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}$ is the ordered list of vertices obtained by the topological sort, let $s(v_{\pi(1)}) = 0$ and $s(v_{\pi(i)}) = s(v_{\pi(i-1)}) + w(v_{\pi(i-1)})$ for $2 <= i <= n$.
7. Dependence constraints are respected, because if $v_i \prec v_j$ then the topological sort ensures that $\pi(i) < \pi(j)$.

Performance Measures:

Given a graph G, a Scheduler S, and P processors

1. Tp(S): time on P processors using scheduler S
2. Tp: Time on P Processors for the best scheduler
3. T1: Time on a single processor (sequential cost)
4. Tinf: Time assuming infinite resources

T1 = work (The total number of operations executed by a computation)

Tinf = Depth (The longest chain of sequential dependencies in the parallel DAG)