Parallel Assignment

Sravanthi Malepati
002-53-8438

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 iff:

-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 × 8 × 3 array and outputs whether or not it consists of a valid configuration.

(a) Pseudocode for a sequential version of **isValid**

Function isValid(A):

```
base_cube=[]

cube_counter=0  \\to get the count of cubes which are lying on the floor

for i=1 to n do

    for j= 0 to 7 do

        set list[ ]=A[ i ][ j ] \\list[ ] will contain individual set of co-ordinates (x,y,z) of an array n*8

        if(list[ 2 ] ≠ 0 )

         there is no base cube in the whole array

        break

       else

         cube_counter++ \\z co-ordinate of base cube is zero

        if(cube_counter = = 4) \\As count of co-ordinates of base cube should be 4

         base_cube.append(i)\\ base cube is found

        cube_counter=0

        break

        endif

endif
```

As there are the base cubes, now we have to search for the stack cubes lying in the plane. If there are any stacked cube then that cube's top face co-ordinates will be same as z co-ordinates of base cube. And also z co-ordinate will be in increment order w.r.t number of cubes. So to find the stacked cubes that are existing in the plane we go for the below steps.

```
for l = 1 to n

    for m =  l+1 to n

            cnt = 0

            cube_one = base_cube(l)

            cube_two = base_cube(m)

    for i = 1 to 8 \\as we have 8 co-ordinate points

            ord_cube = cube_one[i]

            ord1_cube = cube_two[i]

            if( ord_cube = = ord1_cube )

                cnt++

            endif

    if cnt equals 4

        then stack_cube.append

    endif

end
```

(b) Pseudocode for the **parallel** version of the algorithm:

Let us assume we have 'n' processors so each cube will have one processor. This pseudo code involves two steps. The first one being identification of cubes that have Z co-ordinate is zero. That is cubes are lying in the XY plane. Secondly we check for Stacking of the cubes.

```
Func isValid(A):

    cube_base = []

    cube_stack = []

    for i = 1 to n pardo

        cube_base_count = 0
```

```
        cube_stack_count = 0
    for j = 1 to n pardo
        if( A_{i,j}[2] = 0 )
            cube_base_count++
        if(cube_base_count = = 4)
            cube_base.append(i)
        endif
\\to check and get the count of stacked cubes
    if(cube_stack(i,j) [2] ==cube_stack(i+1,j)[2])
        cube_stack_count ++
    endif
    if(cube_stack_count == 4)
        cube_stack.append(i)
    endif
```

(c) Running Time of above algorithms and the **Work** of the parallel algorithm :

In sequential algorithm, for the first loop the running time is O(8n); to find the base cube and for the second loop the running time is O(8n^2) to find the stack cubes.

Therefore., Running time of sequential algorithm is O(8n + n^2) = O(n^2).

Running time of parallel algorithm is O(1) if we assume n^2 processors.

Work of parallel algorithm is O(n^2) which defines total number of executable steps.

(d)

Consider the cubes of different size and we need to check whether they are stacked or not. That can be checked by identifying whether cubes will overlap or not. Here I have specified conditions in which cubes will overlap or not.

```
for i = 1 to n pardo
    for j = 1 to 8 pardo
\\ initially we check for the base cube
```

set list[ ]=A[i][j] \\list[ ] will contain individual set of co-ordinates (x,y,z) of an array n*8

if(list[2] ≠ 0)

there is no base cube in the whole array

break

else

cube_counter++ \\z co-ordinate of base cube is zero

if(cube_counter = = 4) \\As count of co-ordinates of base cube should be 4

base_cube.append(i)\\ base cube is found

cube_counter=0

break

endif

endif

set cube_one = p(i,j)

set cube_two = p(i,j+1)

if cube_one's left face is to the right of cube_two's right face, then cube_one is totally to the right of cube_two \\ to compare X co-ordinates

if cube_one's right face is to the left of cube_two's right face, then cube_one is totally to the left of cube_two \\ to compare X co-ordinates

if cube_one's top face is to the bottom of cube_two's bottom face, then cube_one is totally below cube_two && volume of cube_one is greater than cube_two then return true

if cube_one's bottom's face is top of cube_two's top face, then cube_one is totally above cube_two && volume of cube_one is less than cube_two then return true

if cube_one's front face is behind cube_two's back face, then cube_one is totally behind cube_two \\ to compare Y co-ordinates

if cube_one's left face is to the left of cube_two's left face, then cube_one is totally to the right of cube_two \\ to compare Y co-ordinates

end

2. Suppose that two n X n matrices A and B are stored on a mesh of n^2 pro-cessors 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).

Given two matrices A and B of size n*n that are stored in a mesh of n^2 processors such that Pi,j holds the values of the A[i,j] and B[j,i].

Example: Processor Pi,j holds values Ai,j and Bj,i. Let us illustrate this with the above matrices.

$P_{1,1}$ holds $a_{1,1}$ and $b_{1,1}$ $P_{1,2}$ holds $a_{1,2}$ and $b_{2,1.}$,for 1st row and 1st column multiplication.

Processor $P_{1,1}$ holds [a_{11}*b_{11}] $P_{1,2}$ holds [$a_{1,2}$*$b_{2,1}$] and ……. So on.

But this constraint fails when we try to multiply 2nd row with 1st column i.e

$P_{2,1}$ should holds $a_{2,1}$ and $b_{1,2}$ but it is holding $a_{2,1}$ and $b_{1,1}$ from the above matrices example.

This has to altered so that the constraint $P_{i,j}$ holds values $A_{i,j}$ and $B_{j,i}$. Hence we will be using shifts in the Matrices. These shifts are of two types

Up Shift – the entire rows in the matrix are shifted to one index up. First row takes last row and 2nd row takes 1st row place.

Left Shift – Which means all the columns are shifted to left a unit place.

Pseudocode for an asynchronous algorithm that computes the product of A and B in **O(n)**.

```
for temp = 0 to n-1 step 1 do
begin
        for all Pi;j where i and j ranges from 1 to n do // Pi;j is mesh processor
                if i is greater than temp then
                        rotate A in the left direction //Ai,j is shifted left
                        endif;
                if j is greater than temp then
                        rotate B in the upward direction //Bi,j is shifted up
                        endif;
        end
 //computes the product for each rotated row and column. Then performs addition at the end
which is a final output after matrix multiplication

for all Pi;j where i and j lies between 1 and n do
       compute the product of a and b and store it in temp
        for temp = 1 to n-1 step 1 do for all Pi;j where i and j ranges from 1 to n do
             rotate A in the left direction
             rotate B in the upward direction
               C = C + A * B
        end
```

3. The **Work-Time** scheduling principle schedules P processors to execute an algorithm.

WT Scheduling Principle – It states that a task can be computed in T parallel steps with Wi instructions being performed in step i where I ($1 \le i \le T_\infty$).

Hence $W=\sum_{i=1}^{T\infty} W_i$.This computation can be carried out by N processors in $\lfloor W/P \rfloor + T_\infty$

Algorithm has T(n) time steps. These time-steps can be done parallely i.e., pardo .

Consider a DAG (Directed Acylic Graph), G =(u,v) with total number of operation is represented by work W(n), time T(n) and P(n) number of processors. Where DAG is a graph that is directed and without cycles connecting the other edges. This means that it is impossible to traverse the entire graph starting at one edge. The edges of the directed graph only go one way. The graph is a topological sorting, where each node is in a certain order.

Lower Level follows a general scheduling algorithm, where in a DAG has each vertex independent of each other and every processor can access the data computed by any other processor without additional cost. Let $W_i(n)$ be the number of operations in time i<T(n), each schedule can execute P instructions at a time . Since, each execution has to cover all vertices, total length is W/P.

Upper Level describes the algorithm in terms of a sequence of time units, a processor cannot execute a vertex before its ancestor nodes. Hence, length of the schedule is the path of DAG, which is T(n).

Scheduling :
Let Wi denote the work of the instruction at level i. These instructions can be executed in W_i/P steps. Thus the total time is

$$\sum_{i=1}^{S} \left|\frac{W_i}{P}\right| \le \sum_{i=1}^{S} \left|\frac{W_i}{P}\right| + 1 \le \left|\frac{W}{P}\right| + S$$

Where W_i is Work done in ith instruction

P is total number of processors

S is Total running time

W is total work done.