CSE 4001 : PDC | Fall 2021 | DA-1

by Sharadindu Adhikari, 19BCE2105

Q1.

sl. No.	class ID	class Name	class category	credits	Instructor ID	classroom
1.	900001	Advanced calculus	Math	5	22087	2201
2.	900002	Advanced Music Theory	Music	3	220039	7012
3.	900003	American History	History	5	220148	3305
4.	900004	Computer in Business	Computer Science	2	220387	5115
5.	900005	Computers in Society	Computer Science	2	220387	5117
6.	900006	Introduction to Biology	Biology	5	220498	3112
7.	900007	Introduction to Database Design	Computer Science	5	220516	5105
8.	900008	Introduction to physics	Physics	4	220087	2205
9.	900009	Introduction to political Science	Political Science	5	220337	3308

class ID >= "900003" OR class category = "Computer Science" OR (credits >= "2" AND Instructor ID = "220387")

Let Task a ⇒ class ID >= "900003"

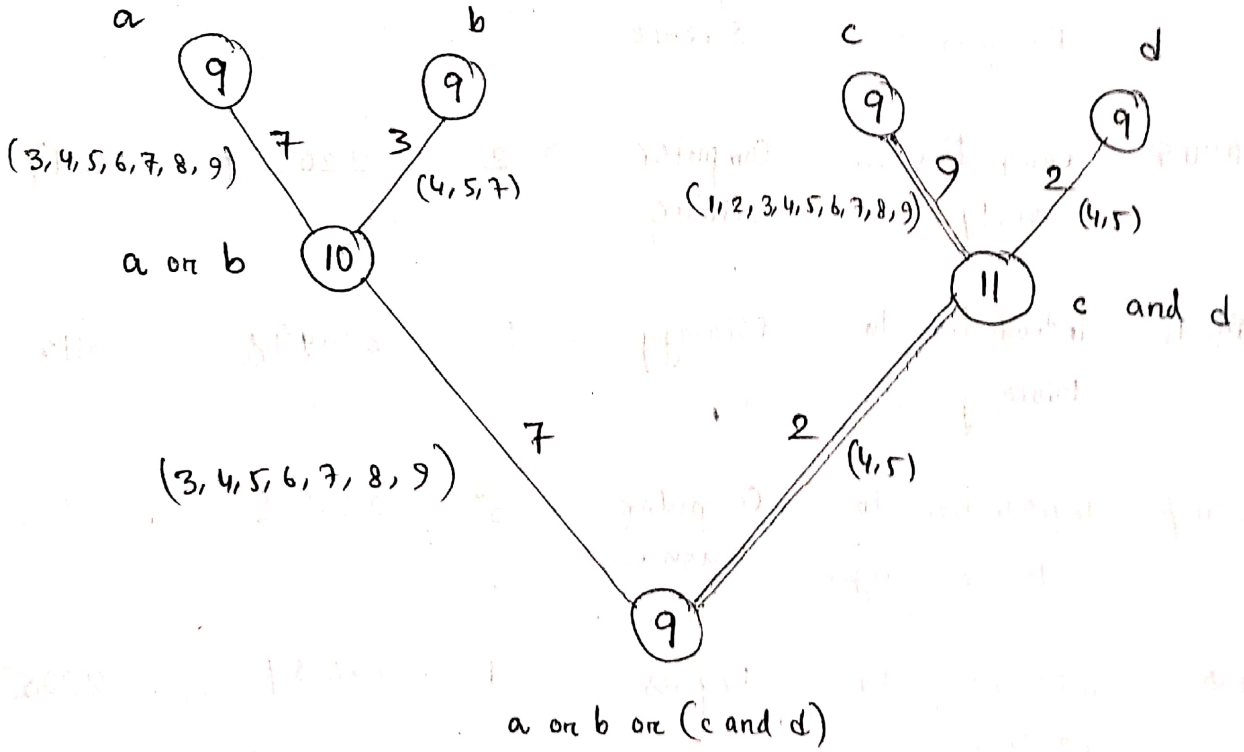
Task b ⇒ class category = "Computer Science"

Task c ⇒ Credits >= "2"

Task d ⇒ Instructor ID = "220387".

Query ⇒ a or b or (c and d)

Task Dependency Graph - 1 :



Critical path length = 9 + 11 + 9 = 29

Total amount of work reqd. = 9 + 9 + 9 + 9 + 10 + 11 + 9

= 66

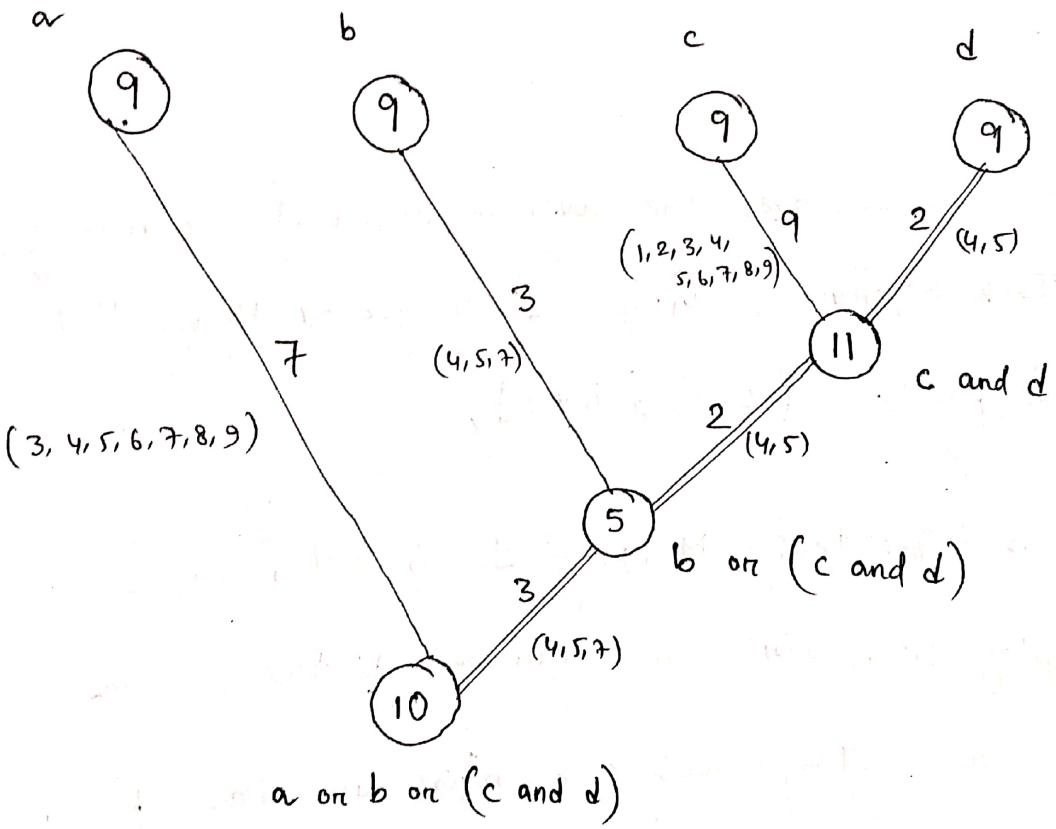
$$\therefore \text{Average degree of concurrency} = \frac{\text{total amount of work}}{\text{critical path length}}$$

$$= \frac{66}{29}$$

$$= 2.275862069$$

$$\approx 2.28$$

Task dependency Graph - 2 :



Critical path length = $9 + 11 + 5 + 10 = 35$

Total amount of work reqd. = $9 + 9 + 9 + 9 + 11 + 5 = 10$ # ~~66~~

$= 62$

∴ Average degree of concurrency = $\frac{\text{Total amount of work}}{\text{Critical path length}}$

$= \frac{62}{35}$

$= 1.771428571$

≈ 1.77

Comments:

- As we can observe, the average degree of concurrency for Task dependency Graph 1 is greater than that of Graph 2. [2.28 > 1.77],

  We can infer that Mapping 1 is better than Mapping 2, since more number of tasks can run concurrently over the entire duration of execution of the program.

Q2.

(i) In this decomposition, we divide the input array 'A' consisting of n elements.

The n elements are divided in a fashion such that 'p' processors receive approximately equal chunks of the array, i.e. $\frac{n}{p}$ elements.

Coming to the part of how the parallel algorithm would work, The output of r buckets is common to all processors. So, each processor will iterate through the elements and counts the occurrences of numbers 1..... r in its own chunk. And thereby add the index to the corresponding bucket in a mutually exclusive way so that the memory location doesn't get overridden.

After this calculation is complete, the vectors are summed through communication / reduction onto a single processor, which is the answer vector B. In this way the parallel algorithm would work, by appending the index of the element to the corresponding bucket where 'r' buckets are shared by 'p' processors.

(ii)

In this ~~de~~ type of decomposition, we divide the output array B into 'p' chunks as evenly as possible; each processor holds approximately

$\frac{r}{p}$ elements of B.

i.e., $\frac{r}{p}$ buckets of the output data.

This is how the parallel algorithm works.

Each processor must hold a complete copy of the array A which is used to count all the occurences of its elements of B.

Each processor would iterate through the array A.

Once done, each processor communicates its portion of the array B to a lead processor which concatenates the results to produce the full version of B.

If there is an element which fits into the buckets held by that processor, the processor would add the index to the corresponding bucket.

Since the input array is read only, there is no need for synchronization.

(iii)

If A is large (large n) compared to the output of 'r' elements of B, it may be infeasible for all 'p' processors to store A in duplicate. This situation lends itself well to the input decomposition which splits up A into smaller chunks. However, this version involves communication overhead at the end to reduce the counts down to the final answer.

There are not too many circumstances which would favour the output decomposition in this formulation of the problems as even if B and 'r' are very large compared to A and 'n', the output is still required to eventually be stored on a single processor. It is likely to be just as efficient to avoid the communication costs associated with moving A around ~~wi~~ and concatenating chunks of B by having a single processor do the whole computation.

Having said all that,

~~In a sing~~ In case of input data decomposition, there is synchronization issue, as the output 'r' buckets is common to all processors.

In case of output data decomposition, for a large value of 'n' compared to 'r', storing copies of

the array A is infeasible (as already described ~~abo~~ earlier). Also, each processor iterates through the same array A, wasting the clock cycles as we know that a given element A will only be going into one bucket at the end.

The relative values of 'n' and 'r' thus make it interesting to select the decomposition techniques.

1. If 'r' is small compared to 'n' decomposition, output data won't produce sufficient parallelization. In this case, decomposing input data would be favourable.

2. If 'n' is small compared to 'r', storing copies of A on each processor wouldn't be an issue. Thus, output data decomposition is favourable.

3. If both 'n' and 'r' are large, the best option would be to go with both input and output data decomposition. If only one has to be chosen, however, input data decomposition is more favourable as synchronization is only required for that particular bucket memory.