Assignment 3 : Parallel Algorithm Design
CSE560: GPU Programming

Anshuman Suri
2014021

1. The two cases can be analyzed differently as follows:
    a. Let the task dependency graph root have *D* branches. The first claim is that *D* is the maximum degree of concurrency. This can be shown by contradiction, since it were not, the node responsible for the maximum degree of concurrency would have been in some subtree, which could have been joined with the root node to get a higher degree of concurrency (since we have control over the task dependency graph). Thus, to consider the maximum degree of concurrency, the degree of the root node is considered. Now, let the different subtrees rooted at the node each have *d_i* nodes in them (excluding the root). Then,

    $$\sum_{i=1}^{D} d_i = t - 1$$

    (the *-1* accounts for the root node, which is not counted in the *d_i* s)

    Each of *d_i* is in the range *[1,l-1] (*except the one containing the critical path length, which would have at least *l-1* nodes), as we are given that the critical-path length of this graph is *l*.
    When all of them are 1 (except the subtree containing the critical path length, which contains *C* nodes, *C* >= *l-1*), the equation reduces to:

    $$\sum_{i=1}^{D} d_i = (D-1) + C = (D + C - 1) = t - 1$$, *ie*

    $$D = t - C$$

    Since we are maximizing *D* in this case, we will set *C* as l-1, that is

    $$D = t - l + 1$$

    Thus, $D <= t - l + 1$

    b. Since the critical path length is given to be *l*, at least one path needs to be of that length. Since the maximum degree of concurrency is determined by the the maximum degree over all nodes, we aim to distribute all nodes over every node. In the beginning, we have *t-l* nodes to distribute (as *l* of them are already used over the critical path). At this point, the maximum degree of concurrency is 1, as it is essentially a linear chain. After distributing these nodes over remaining task dependency graph, the maximum degree of concurrency increases to 2 (even if

all are not distributed, since at least one node gets a connection).If after this step we are left with nodes, we continue the above algorithm, increasing the maximum degree of concurrency by one, after which we will have t-2l *nodes.* Following this pattern, the degree of concurrency is *x*, when we have t-x*l remaining nodes. Thus, the maximum degree of concurrency is (the ceiling is because even if one of the nodes get an extra edge, the maximum degree of concurrency will increase):

$D = ceil(t/l)$
Thus, $D >= ceil(t/l)$

Combining these two inequalities, we get:

$ceil(t/l) <= D <= t - l + 1$

2. The time complexity while sorting *n* integers using bucket sort (when the input data is in the range *[1,r]*) is *O(n+k)*

    a. For input data decomposition, we can decompose the input array into chunks such that all processes get roughly the same amount of data. Thus, each node gets roughly *n/p* elements of the array. After that, each node will sort the elements it has individually, and then finally we combine the bins we get from each node to get the final output array.

    The algorithm basically first makes splits on the input data and passes it to *p* processors. These processors individually run normal bucket sort on the data they get. Ultimately, all processors combine their outputs (adding the frequency histograms they individually have) to get the ultimate output.

    b. For output data decomposition, we can pass the input array into different processors (via sharing) such that each process only looks at a specific range as buckets. Thus, each process will look at only roughly *r/p* bins. Ultimately, all of the bins from all processors will be combined to get the final output.

    The algorithm basically sends copies of the input data to all available processors. Any given processor looks at only roughly *r/p* of the given values that can be possible; any other value is ignored. Ultimately, the frequencies from the processors are concatenated (as any process and the values it handles is known at the time of decomposition, there is no need to sort the individual semi-bucket lists) to get the desired output.

3. For the input data decomposition method,the worst case time can be calculated as:

    a. Each of the *p* processes will run normal bucket-sort on partitions of data which are each of size *n/p*. First, we need *p O(n/p)* to copy data and distribute it into the processors. Since this is done in parallel it is equivalent to *O(n/p*) Each of the processes runs in parallel, taking *O(r + n/p)* time. After they are all done, we need to combine the histograms from the *p* processors, which would take *O(p*r)* time. Thus, the worst case running time for bucket sort via this decomposition will be:

        *O(n/p) + O(r + n/p) + O(p*r) , ie, O(n/p + p*r)*

    b. Each of the *p* processes will run a modified bucket sort, looking at only a chunk of the keys concerned. First, we need *O(n)* to read the data into the processes (it will not be copied into each process; it can be shared, as it is read-only). After that, each of the processes (running in parallel) takes *O(n + r/p)* time. After they are all done, we need to concatenate the results from the *p* processors, which takes *O(p*r/p), ie, O(r)* time in total. Thus, the worst case running time for bucket sort via this decomposition will be:

        O(n) + O(n+r/p) + O(r) , ie, O(n + r)

    Thus, the performance of the two decompositions is dependent on the values of *n,p* and *r*.

    *n/p + p*r v/s n + r*
    r(p-1) v/s n*(1-1/p)*
    r(p-1) v/s n*(p-1)/p*
    r v/s n/p

    Thus, when the value of *r* is less than the ratio *n/p*, we would prefer the first approach of data decomposition (input data decomposition). On the other hand, if the ratio of *r* is greater than *n/p*, we would prefer to go with the second approach (output data decomposition). Thus the relative values of *n* and *p* do have a bearing on which type of data decomposition we select.