Assignment 2 Extracting Parallelism

1. Transform:
   a. The Dependency in given Code is:

      RAW-> Read after write.

      This program has two operations in 'for' loop.

      First is calculation of f(a) and second is putting the outcome in variable b. The first operation can be executed in parallel on p processors. Second operation can also be executed in parallel on p processors. But the two operation have dependency. Second operation cannot be completed before first operation is done. Hence that gives us dependency of second operation on first.

   b. Width, Critical Path, Work:

      Consider we have p processors and 'n' array elements.

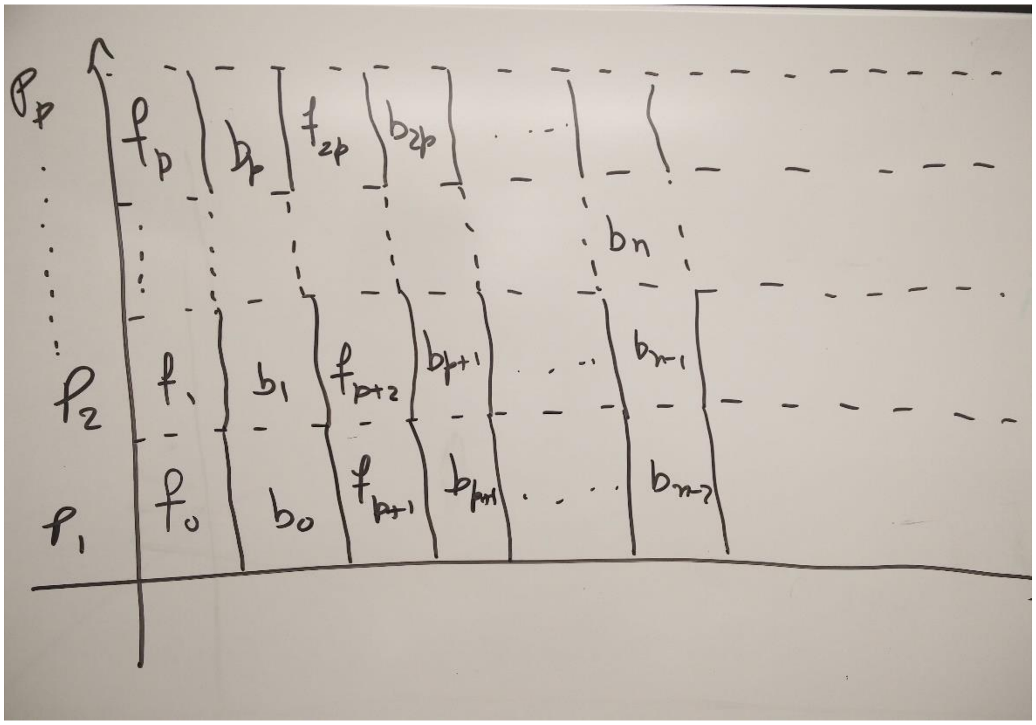
      Width: p

      Critical Path: 2n/p

      Work: 2n

   c. Schedule for processors:

      Consider we have p processors.



2. Reduce:
    a. Int, Sum
        i. Dependency, Width, Critical Path and work:
        RAW-> Read after write.
        In this program the Sum() function calculates sum between array elements of array 'a' and variable 'result'. Each iteration is independent and they can be executed in any order. But no two iteration can be processed in parallel as each one would be trying to access same variable 'result'. Hence unless previous iteration finishes writing new iteration cannot read and eventually write. So the RAW dependency.

ii. Introducing Mutual exclusion clause:
Every iteration is independent of each other
And can be executed in any sequence. But
Introducing mutual exclusion does not help out.
For example we assign every iteration a separate
Resource or processor. They still cannot execute
in parallel.

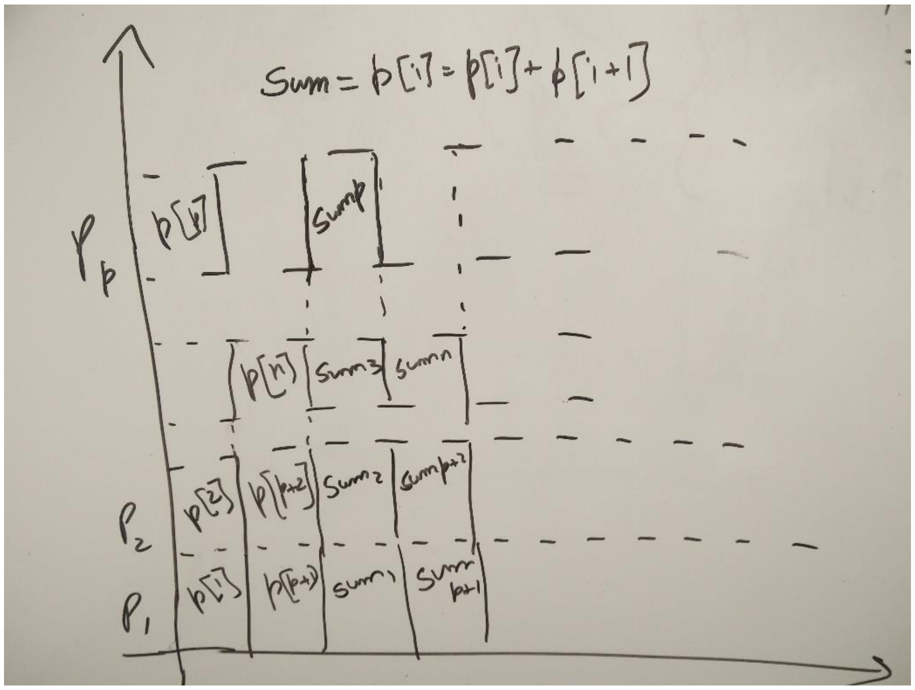
iii. Code: consider we have p processors

```
For (int i =0; i<n; i++)
{
        p[j] = sum( p[j], array[j]);
        j = j%p;
        j++;

}
While(k<p+1)
{
        For (int i = 0; i<p; i = i+k)
        {
                p[i] = sum(p[i], p[i+k-1]);
        }
        k++;
}
Work = n-1;
Width = p;
```

Critical path = $\log_2(p)$

iv. Schedule on processor:



b. Variants

i. No, Mutual exclusion is not correct as the iteration can be executed in any order but there is RAW dependency on 'result' variable. Local variable is correct as it can store the partial max between two elements and later can be used to compare between partial max of other processors.

ii. Yes, Mutual exclusion clause is useful as in this the iterations need to be executed sequentially. Local variable is useful. But once we have partial concat we need to execute remaining steps sequentially.

iii. This is similar to above sum example. Mutual exclusion is not useful but local variable is.
iv. Here also mutual exclusion is not usefull but local variable is.

3. Find first:
   a. For(int I =0;I<n;I++)
```
{
    If(arr[I] ==val)
    {
        Return I;
    }
}
```
   b. Complexity: O(pos)
   c. Parallel Algorithm with O(n)
   d. Parallel Algorithm with O(pos) work
      i. Consider we have p processors.
      ii. Repeat below until we reach end of array or we get 1 as output from any 1 processor.
      iii. Input p-1 variable and search val in p-1 processor.
      iv. Compute the comparison and output either 0 for failure or 1 for success.
      v. Pass the output to processor p.

vi. If processor p detects 1output from anyone processor output the index else pass the next p+1 to 2p varibles.

vii. Width: p

viii. Critical Path: pos/p

e. Linked List: Above algorithm cannot be implemented on linked list as there is no way to directly access any index location in linked list as in array. For that we have to start from head node and traverse. But during building of linked list if we make an index of each pointer location it is possible to traverse like array.

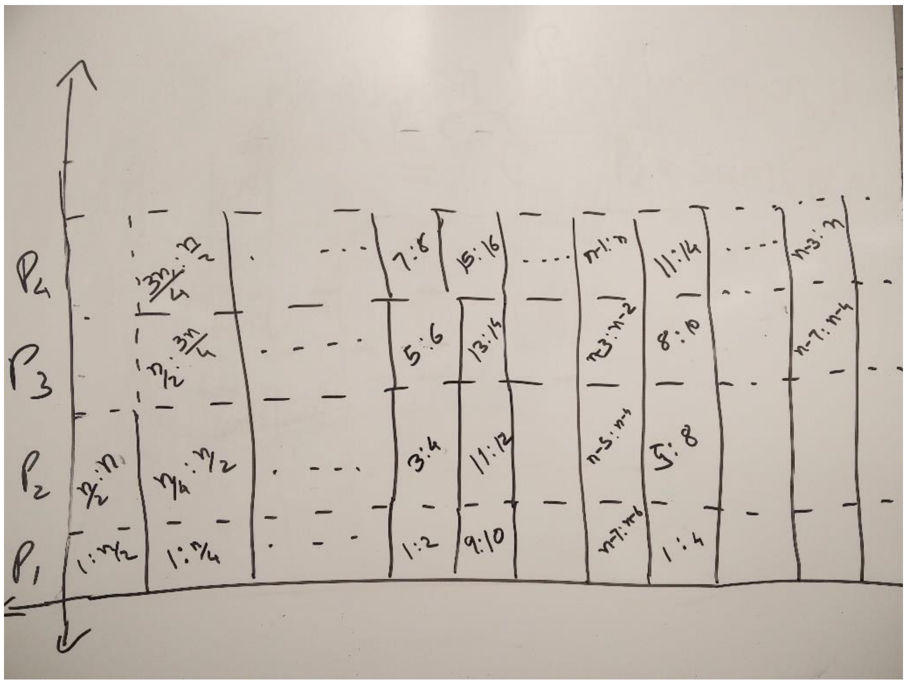
4. Prefix Sum:
   a. Dependency Structure: In this code we have RAW dependency. As the sum of current element depends on sum of previous element which also depends on its previous element and so on. Hence we have to execute this in sequence to maintain the prefix sum property.
   b. Parallel code:

```
For(I =0; I<n;I++)
{
        pr[i] = pr[i] + arr[i];
}
```

```
For(int I =0;I<n;I++)
{
        pr[i] = pr[i] + pr[i-1]
}
```

5. Merge Sort:
    a. Merge Sort Algorithm:
        i. Divide the list into half and then divide those two list into half and so on until only single elements are remaining.
        ii. Start merging the list with below rules.
        iii. While merging between the top two elements in list put the smaller element first.
        iv. Keep merging until we are left with single list.
    b. Dependency on Merger Sort:
        i. In merge sort specifically in merging process we need to make sure to merge all list of length l before proceeding to merge list of length 2l.
        ii. We can merge all list of length l in parallel but need to wait until all list of length l are merged before proceeding to length 2l.
        iii. Critical Path: '2*$\log_2(n)$' for below parallel algorithm and 'n' for above sequential algorithm
        iv. Work: n-1
        v. Width: if n<p then 'n/2' else if n>p then 'p' for below parallel algorithm and '1'for sequential algorithm.

c. Schedule on P = 4 processor:



d. More Parallelism:

i. Take input the list which needs to be sorted.
ii. Create thread merge sort and pass the list to the thread.
iii. In the thread if elements are greater than one then divide the list and create two threads. Pass each half to each of the two threads and start waiting.
iv. If element size is 1 then return the element.
v. Else out of the elements returned by child threads write the smallest element first and larger element following that.
vi. Return the newly created list.