ASSIGNMENT - 2

1) a) There is no particular dependency here, since b[i] is only reading a[i], and there is no write before, which conflicts. All of the access is individual.

b) The program can be parallel, and hence it is O(1), since f, also has O(1) dependence.

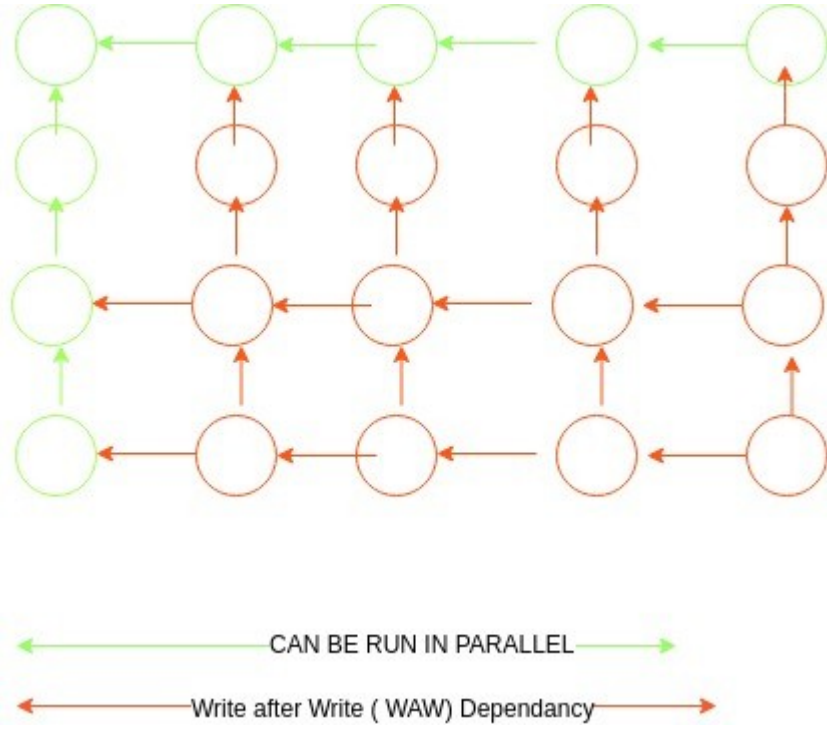
c) GANT CHART BELOW:

ASSUMING WE HAVE P = (1/2)*N and time per process is t	
	Task P+1
Task 1	Task P+2
Task 2	.
Task 3	.
Task 4	.
.	.
.	.
.	.
Task P	Task N
----------> Time (t)	**----------> Time (2t)**

2) a) Complexity is O(m*n).

b) All the dependencies are Write after Write(waw), and the current f[x][y] depends upon, f[x-1][y] and f[x][y-1], since they are written in previous iterations.

c) Work: O(m*n), Width: O(2), since two threads can initially run to initialize the values of first row and column, Critical Path: O (m+n).



3.1 . REDUCE

3.1 A) array[0] → result → result op(result,array[i]) (N times) ….. N times

Let us assume that for now N is 5

array[0] → result →result = op(result,array[i]) →result = op(result,array[i]) →result = op(result,array[i]) →result = op(result,array[i])

Since the code depends on the previous result, it complicates things

There is a dependancy on result . This is the main dependancy.

It has Write after Write (WAW) dependency in that the previous result value is necessary to compute and write into now's result value.

Work = O(1 + n - 1) = O(n)
Critical Path = O(1 + n -1) = O(n)
Width : 1 thread only → O(1)
If this program is not rewritten, then the Function cannot be parallelized, since there are too many dependancies

3.1 B) If we have P processors, we have an option of changing the code like this so as to split the addition process through different processors available. But ideally, we select the number of processors.

The threshold shows how we are splitting the task we have, whether it is 2 processors and a 50% split each , or 4 processors, and 25% split each.

```
//CODE HERE
template < typename T , typename op >
//We include the processor number here to be able to divide the tasks accordingly, by the number of processors
T reduce ( T * array , size_t n,int p_num ) {
int answers[p_num];
T result = answers;
//Here thresh is how many we are summing up along with
int i=0;

while(i<p_num)
{
answers[i] = array[(n/p_num)*i];
//Spawn new thread here, and use new processor to achieve parallelism
        for ( int j =((n/p_num)*i)+1; j < j + n/p_num ; ++ j )
        {
                answers[i[ = op(answers[i-1],array[j]);
        }
        i++;


}
int total_sum = 0;
```

```
for(i=0;i<p_num;i++)
{
        total_sum += answers[i]; //Finally summing all the values in the different processors
}

}
```

Work : O(n)
Width: O(P)
Critical Path: O(N/P)
The only dependancy is the overall sum from all processors has to wait for them to finish to compute.

3.1 C)

We assume that the number of processors is p_num	Let's call this time as a constant C
J= 0 to J = 0 + n/p_num.	Time for one loop (t)
J=n/p_num to 2n/p_num	
J = 2n/p_num to 3n/p_num	
,	
.	
.	
.	
J = (p_num-1)*n/p_num to ((pnum-1)*n/p_num)+ n/p_num	
--------> Time (t) = time taken for 0 to n/p_num (one iteration)	----------> time taken is constant to sum data from all processors

3
.2 VARIANTS

3.2 A) Yes the parallel version would be correct for int,max since this is also a process which can be computed in parallel. We can divide the entire list into groups and calculate the maximum for each sublist , and then merge the maximums together, to find a new maximum.

3.2 B) This question depends on whether the order of the strings being inputted can be stored in a separate processor. If so, then yes it applies. Else it cannot be since the concatenation will be ruined.

3.3 C) Yes it would be since, the sum is a process which can be divided and merged in the future, whether int or float

3.4 D) Yes it would be since finding maximum value task can be split into multiple tasks, and this would assist in splitting the workload and later merging the same.

4) PREFIX SUM

4 A) Dependancies of the Problem:

arr[0] → READ → pr[0]
pr[i] → pr[i-1] + arr[i]

pr[i] depends on the previous iteration of the loop (pr[i-1]), so this process cannot be completed parallelly.

Each step of the process entirely depends on the previous step, and hence needs an O(n) computation done.

Work Done: O(n)
Critical Path: O(n)
Width : O(1)

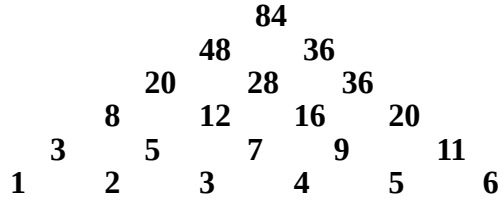
Width: 1 Thread can only function at a given.

Program can be made parallel , and thereby reducing work per server, and thereby the critical path to O(logn). The entire code has been written in "parallel_prefix_sum.cpp".

- This is done, by using a tree mechanism.
- A summation tree is constructed using all the values given. This summation tree's root contains the total sum.
- The construction process' critical path is O(logn), this is because we are recursively accessing the tree, by splitting into servers as we construct it .
- The program has two parts. Part 1, and Part 2 have log(n) each
- The processes can be recursively threaded.

Final Critical Path: O(logn)
Final Work Done: O(n)

We do it by building a tree for the task:



We do this entire tree build up, by using pointers as indicated in the program "parallel_prefix_sum.cpp".

The logic is that we maintain 2 pointers in each node of the tree:

The tree drop down is as follows:

sum and fromleft.

Sum is the respective values of sums at the tree , as shown above. Fromleft is initialized at 0 ONLY for the root.
Fromleft of the left node, is the same as the fromleft of the top node.
Fromleft of the right node, is the sum of the fromleft of my node + “sum” of the left side node.

In the end after we complete the fromleft process, we will find the working prefix sums, at the fromleft of each of the leaf nodes which are stored in memory for this reference.

The entire working code can be found in “parallel_prefix_sum.cpp”

A sample code of how the parallel prefix works is this:

```
float prefix_sums(TreeNode* top,vector<float> &prefixsums)
{
	if(top->right == NULL && top->left == NULL)
	{
		prefixsums.push_back(top->fromleft);
		return top->fromleft;
	}

	if(top->left!=NULL)
	{

		top->left->fromleft = top->fromleft;
		prefix_sums(top->left,prefixsums); //Possibly run on different processor
	}
	if(top->right!=NULL)
	{

		top->right->fromleft = top->fromleft + top->left->sum;
		prefix_sums(top->right,prefixsums);//Possibly run on different processor
	}

	return top->sum;

}
```

5. MERGE SORT ALGORITHM

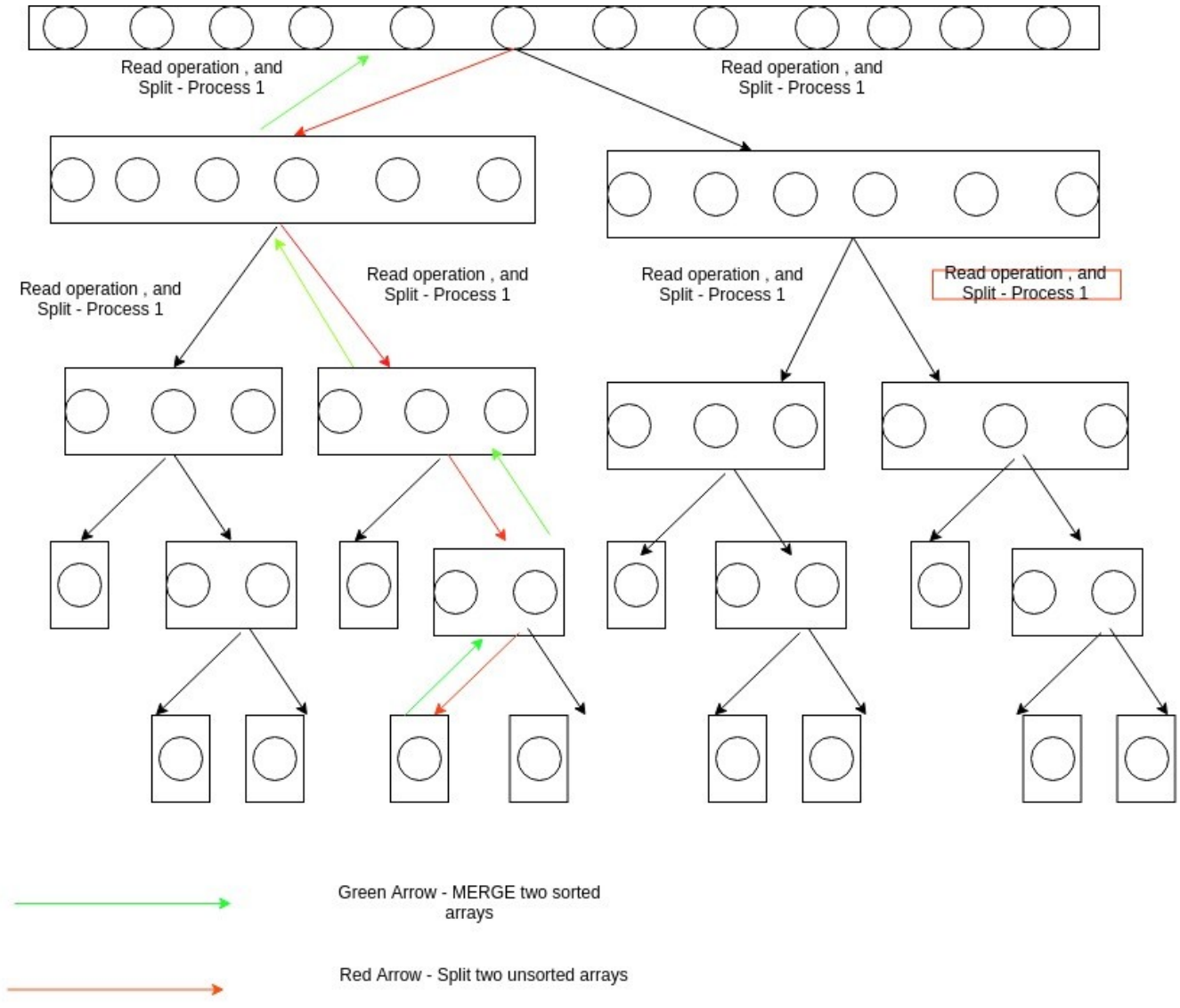
A) Merge sort runs with nlogn speed. The basic premise of merge sort is , the higher levels of sorting wait for the lower levels (smaller number of elements to be sorted)

The smaller sorting creates lists which are to be sorted in the higher levels, each of which take nlogn (n being the size of each of the two lists to be sorted).

That being said, merge sort can be parallelized, even using it's own existing algorithm , by running each of the individual sorting process in threads, Since they are all running in parallel. For this we modify the "merge" function at the end of the recursion process.

B) Critical Path: O(n) . This is because , sequential merge has no speedup.
Work: O(nlogn), this is because this is the time complexity, and it takes this long to finish and merge
Width: O(n), n threads, finally one for each element.

Drawing done in **draw.io.**

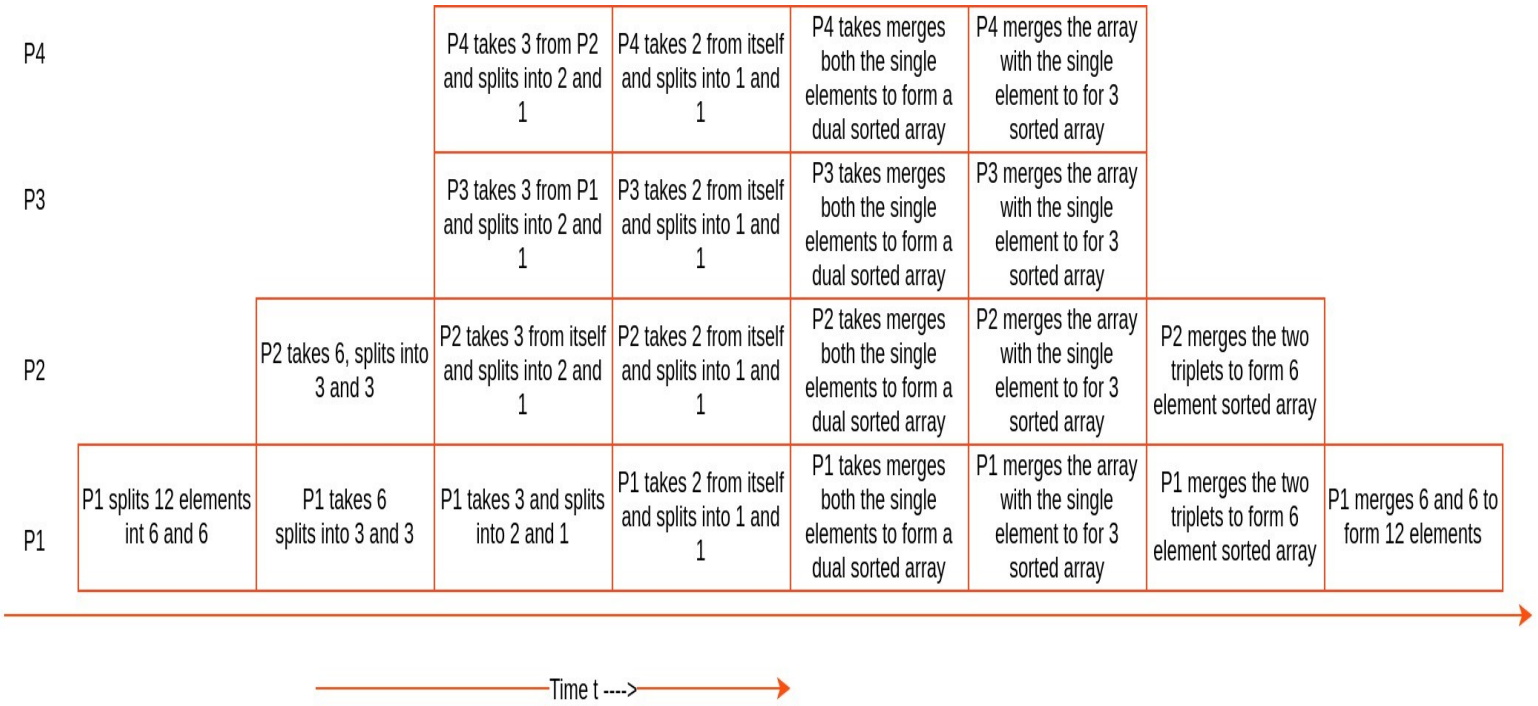


C) SCHEDULE:

We have taken the assumption that P=4, Now let us take the case that the size of the list is 12. This way, we can visualize it accurately.

As it is visible from the image above, the **GREEN** path shows how the merging happens after splitting. This is the sorting phase. Each green phase, uses the “merge” function. Each Path, like the Red one, uses One thread, and is run indivudually. The server, which delegated the split to another server, is responsible for merging the two of it’s branches.

The critical Path is O(nlogn). The depth of this tree is log(n), number of elements in the tree. But this unfortunately is NOT the critical path. The actual critical path is O(n), since the sequential merge demands this.



GANT chart:

drawn with draw.io

D) Yes this program can be made more parallel. The method is as follows.

1) Identify a threshold after which the suitable parallel merging can be applied.

This has the following reasons: When we run individual threads for each of the individual 2 element sorts, it takes too much unnecessary memory. These can be done by a single processor, even until we reach a certain human defined threshold. Since this program is a demo, the program written in cpp, “modified_merge_sort.cpp”, contains a threshold of 2, so that the actual use of the modified merge can be indicated.

Before the threshold is reached, we use regular O(n) merging. Which may not be too computationally intensive, in that as many processor threads are unused and may not be necessary.

After the threshold is reached, we use BINARY SEARCHED index merging, wherein we maintain individual record of the actual index, but comparing the current index with the other elements to find the perfect position of this element in the other list.

FOR EXAMPLE:

5,6,7 and 1,2,3,4

5 knows that it's own index is 0.
her ml ass
It does a binary search to find it's own location in list[1,2,3,4]. It finds that it's place is 4. Knowing that it's own index is 0 and 4 other elements are in front of it, it's position is 4 in the overall list. Performing this recursively/using a loop, through both the lists will lead us to finding the response we look for.

1,2,3,4,5,6,7

The code snippet is as follows:

```
void mergesort(int start,int end,vector<float>& array)
{
	int THRESHOLD = 2;
	if(start>=end)
	{
		return ;
	}

	int middle = (start+end)/2;
	mergesort(start,middle,array);
	mergesort(middle+1,end,array);
	if(end-start <= THRESHOLD)  // We find and merge based on threshold
		merge(start,end,array);
	else
	{
		threaded_merge(start,end,array);
	}

	printarray(array);
}
```

THREADED MERGE:

```
void threaded_merge(int start, int end, vector<float>& array) //logn Duration
{
	//Code to run individual threads and store the right information

	vector<float> array1;
	vector<float> array2;
	int middle = (start+end)/2;

	int i = start;
	int j = middle +1;
	int k = start;
	float temp[array.size()];

	for(i = start;i<=middle;i++)
		array1.push_back(array[i]);

	for(j=middle+1;j<=end;j++)
		array2.push_back(array[j]);

	k = 0;
	for(i=0;i<array1.size();i++)
	{
		int my_index = i;

		int neighbor_index = binarySearch(array2,0,array1.size(),array1[i]);
		temp[my_index + neighbor_index] = array1[i];
		k++;
	}

	for(i=0;i<array2.size();i++)
	{
		int my_index = i;
		int neighbor_index = binarySearch(array1,0,array2.size(),array2[i]);
		temp[my_index + neighbor_index] = array2[i];
		k++;
	}

	for(i=0;i<k;i++)
	{
		array[start+i] = temp[i];
	}




}
```

The entire code is available in file “modified_merge_sort.cpp”.