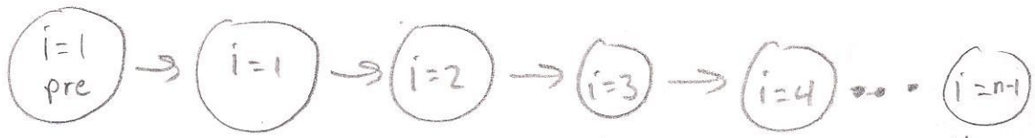
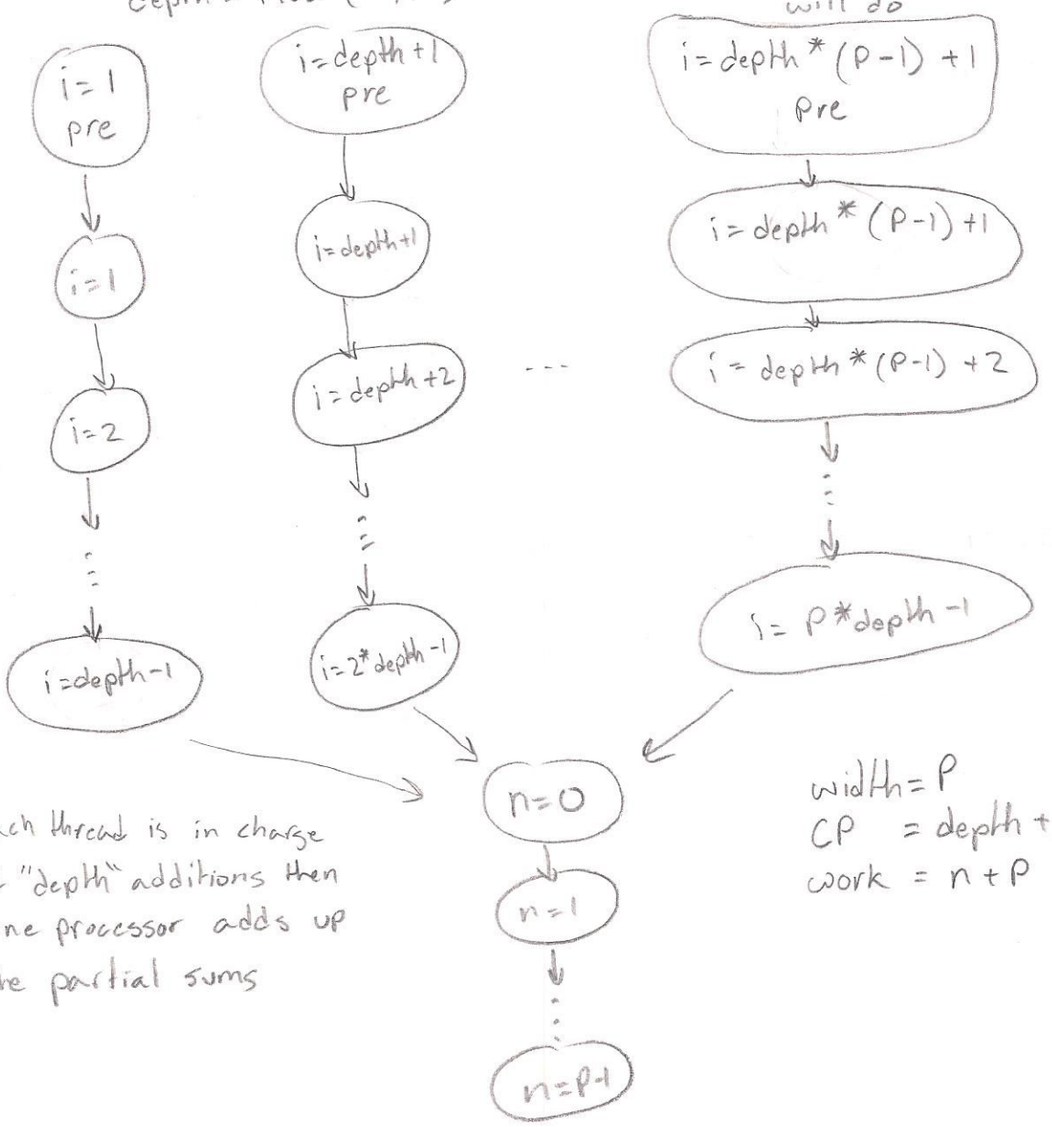
3.1.1)



All tasks are dependent on variable "result"

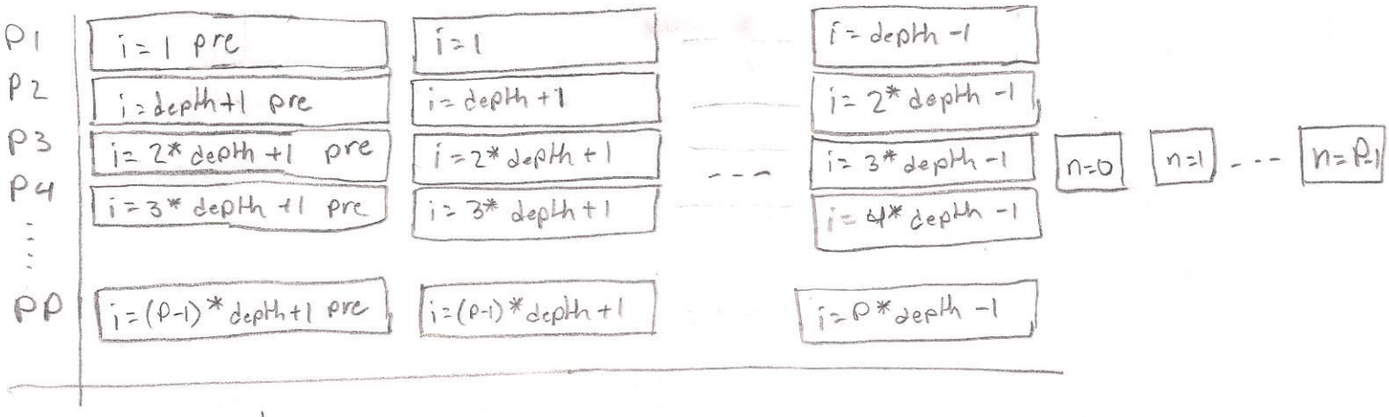
width = 1
CP = n
work = n

depth = floor(n/p) ← this is the number of additions each processor will do



each thread is in charge of "depth" additions then one processor adds up the partial sums

width = p
CP = depth + p
work = n + p



Each thread is incharge of "depth" number of sums. Then one thread (P3) is in charge of adding up the partial sums. the pre task foreach thread is the initial set of the result variable

---

3.2) reduce<int, max> - Yes. It would find the max element in each section of the array then compare the max for each section against each other to find the whole max

reduce<std::string, concact> - yes, the algo would concact each section then concat each section together, it would maintain proper order

reduce<float, sum> - yes, code is templeted, and int is similar to float & would work the same way

reduce<float, max> - yes, similar to reduce<int, max>

4)

all tasks are dependent on the i-1 iteration. This is because to calculate pr[i], you need pr[i-1]

width = 1
cp = n+1
work = n+1

---

parallel version

Similar to the reduce function, the array will be split into smaller chunks and prefix sum will be computed then the last sum of each chunck is added into the next chunk

Pass 1

ar	pr
0	0
1	0
2	1
3	3
4	6
5	10
	15

thread 1
offset = 15

ar	pr
6	0
7	6
8	13
9	21
10	30
11	40
	51

thread 2
offset = 51

ar	pr
12	0
13	12
14	25
15	39
16	54
17	70
	87

thread 3
offset = 87

do partial prefix sum for each chunck of the data in this case 3 chunks. The offset is the largest partial sum of that chunk. This is step 1

Pass 2

error = 0

pr	fin
0	0
0	0
1	1
3	3
6	6
10	10
15	15

error = 15

pr	fin
0	15
6	21
13	28
21	36
30	45
40	55
51	66

error = 51 + 15 = 66

pr	fin
0	66
12	78
25	91
39	105
54	120
70	136
87	153

Now add all the offsets from the previous chunks to each member of the partial sum the sum of the offsets is the error. This sumation is independent and can be threaded similar to step 1. This is step 2 and final step

5) mergesort super highlevel pseudocode

```
mergesort (arr)
    if len(arr) ≤ 1:
        return arr

    left = left half of arr
    right = right half of arr

    left = mergesort (left)
    right = mergesort (right)

    return merge(left, right)
```

an array of size 1 is sorted

split into 2 parts

sort both left + right parts recursively

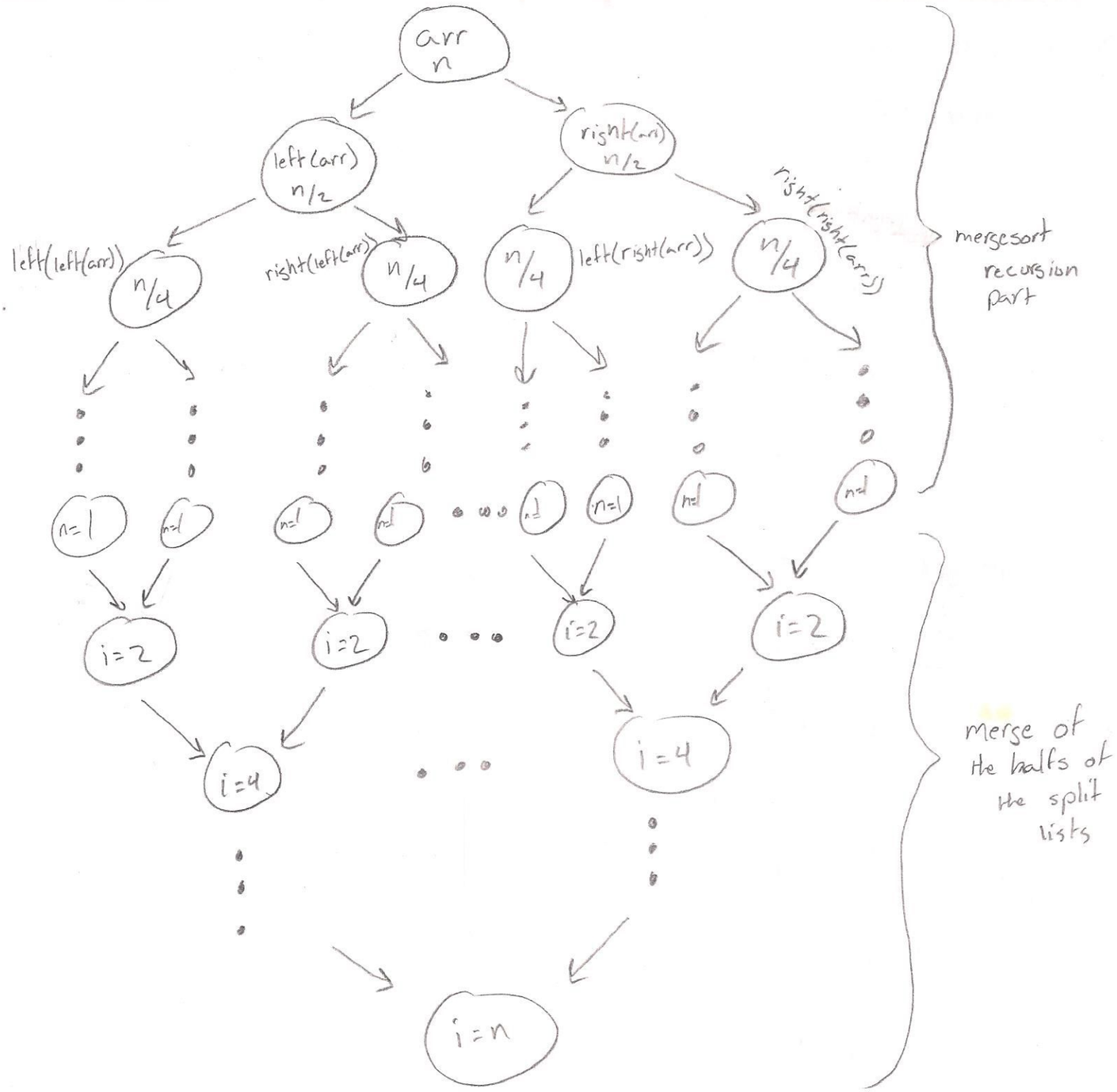
merge the two parts in order

```
merge(left, right)
    result = empty array

    while left is not empty
            and right is not empty:
        if left[0] ≤ right[0]:
            result += left[0]
            rem left[0]

        else:
            result += right[0]
            rem right[0]

    if left is not empty
        result += left

    if right is not empty
        result += right

    return result
```

create empty result array

go through both left + right sides and place the smaller of the first element into the result array

place the rest of the elements from either list into the result array

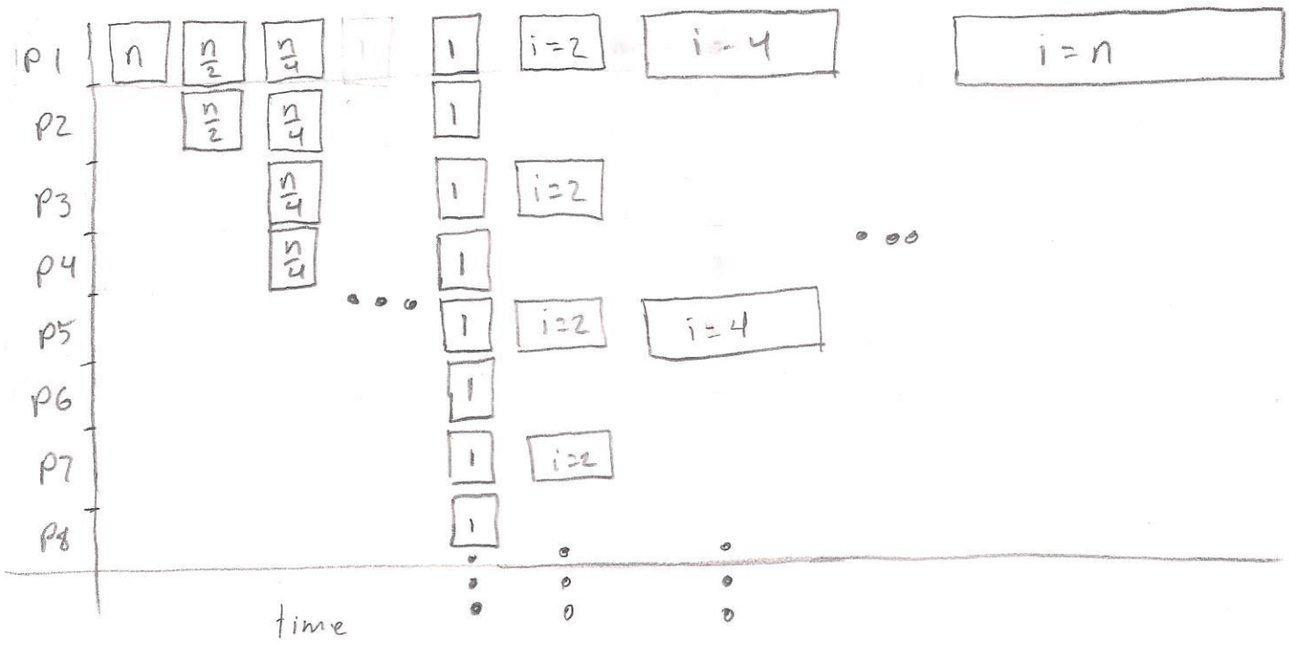


The first half of the dependency tree is the recursive part of mergesort() function it splits the array into 2 subarrays. Then these subarrays are sorted and merged in the second half of the dependency graph. The merge part of the dependency graph takes more time to compute

work = $3n-2$

CP = $2\log_2(n)+1$

width = 1



Diamond shape? If we dont have any penalty for spawning threads, then each recursive call would spawn 2 threads: one for each half of its current thread. There is very little "work" happening after than the creation of the threads. That is why the bars here are thin. when the merger of two subarrays happen, one thread will handle it but because it has to go through 2 sub arrays, the time taken to sort will double as the recursion stack collapses to the end condition 'i=n'

To make the code more parallel, I would remove some of the initial recursion so there is little dependency at the biginning of the run. For example with a P of 4, the fist 2 "layers" of the ghantt chart / dependency graph will be gone. Then the processor mergesorts the 4 halves of the array. At the end a thread sorts the 2 halves that are left.