CMPS 6610 Problem Set 03

Answers

Name: Pavan Kumar Sanjay

Place all written answers from `problemset-03.md` here for easier grading.

- **1b.** The iterative solution moves from one element to another checking if the key exists within the array. As there is no capability of parallelization within this solution the work and span both will be equal to O(n)
- **1d.** This solution involves dividing the arrays into 2 pieces and solving each piece individually before merging the result. So the work can be given by the following reccurence:

$W(n) = 2 * W(n/2) + 1$

Using brick method:

$W(n) = \sum_{i=1}^{\log_2 n} 2^i$

Therefore W(n) = O(n)

Meanwhile the span is given by:

$S(n) = S(n/2) + 1

Here similarly using brick method:

$S(n) = \sum_{i=1}^{\log_2 n} 1 = \log_2 n $

Therefore S(n) = O(log n)

- **1e.** On replacing reduce with ureduce it is confirmed that there is no change in output. However our recurrences for work and span change into:

  $W(n) = W(n/3) + W(2n/3) + 1 = \sum_{i=0}^{\log_(3/2) (n)} 2^i = O(n^{\log_(3/2) 2}) $ ----(reccurence for work done)

  $S(n) = S(2n/3) + 1 = \sum_{i=0}^{\log_3/2 n} 1 = O(\log n) $ ----(reccurence for span)

Compared to reduce the total work done increases while the span decreases.

- **2a.**

  The sparc specification for the algorithm is designed below: dedup A =

  let

  // Here x is an empty sequence and y is an element from the original sequence

  appending x,y = {

```
x U {y} if $y \notin x$

x        otherwise

}
```

// specification for iterate

iterate f x a = {

```
x

iterate f f(x,a[0]) a[1 .....|a|-1]

}
```

in

```
iterate(appending,<>,A)
```

end

Since we will be running iterate twice to check the precense of elements.
Work and span = O(n^2)

- **2b.**

The sparc specification for the algorithm is designed below:

multidedup A = let

```
// Here x is an empty sequence and y is an element from the original sequence

appending x,y = {

x U {y} if $y \notin x$

x        otherwise
}
```

// specification for iterate

iterate f x a = {

```
x

iterate f f(x,a[0]) a[1 .....|a|-1]


}

// merging deduplicated local lists
merge a0,.... an = a0 U a1 U .....an
```

in iterate(appending,<>,merge(iterate(appending,<>,for [A0, A1.....Am] in A))) end

Based on the algorithm the amount of work done to get a dedupted list is O(n^2). For m such lists the work will be = $O(mn^2)$ [as there are m lists]

Span = $O(mn^2)$ as there is no parallelization

- **2c.** In my opinion the sequences are very useful in terms of implementing certain features without the use of many for loops however for this use case it does nopt effect the work and span too much as deduplication is a process which cannot be parallelized very easily. In the above cases iterate was the only sequence operation that could be used to implement the functions.
- **3b.**

As based on the above implementation the iterate method is used to iterate through each element of the array increasing and decreasing the counter based on the expressions. The work done for each element is constant time.

Therefore:

Work Done = O(n)

As there is no parralelization and each element is iterated through one by one:

Span = O(n)

- **3d.**

Here assuming that maps function is parallelized the only significant computation would be the scan function. The most efficient method of performing scan is the one which involves partial outputs and further addition its work(W(n)) and span(S(n)) is given by:

$W(n) = W(n/2) + n = O(n)$ ---[Here W(n) is root dominated this can be proven as the cost at level-1 = n/2 which signifies geometric decrease] $S(n) = S(n/2) + 1 = 1 + 1 + 1..... + 1(Totalnumberoflevelsis \log_2 n) = O(\log n)$

- **3f.**

For the divide and conquer implementation we are splitting our list into two and then merging them based on the values returned. The merging process would take constant time as they consists of finding the minimum and some additions and subtractions. Therefore

Work Done = $ W(n) = 2W(n/2) + 1 = \sum_{i=1}^{\log_2 n} 2^i = O(n) $ (Calculated using brick method)

Span = $ S(n) = S(n/2) + 1 = \sum_{i=1}^{\log_2 n} 1 = O(log n)$ (Calculated using brick method)