Design and Analysis of Parallel Algorithms
Exercise 1

s0840449

1 Question 1

1.1 Part (a)

The main differences between the original CRCW algorithm and the EREW equivalent that we were asked to create for the coursework are the reduced number of processors, and the inability to do either concurrent reads or writes.

Since we only have n processors, we must remove one of the parallel loops in the rank calculation. Making the inner loop sequential is simple; the only extra work needed is to increment $wins[i]$ instead of assigning to it, since we no longer have parallel associative writes.

However, the resulting algorithm is only CREW, not EREW, as the reads from $A[i]$ and $A[j]$ in each timestep will cause processors to overlap. Copying the values of A to another array, C, removes the problem of an $A[j]$ read overlapping an $A[i]$ read. However, there are still overlapping $C[j]$ reads - since the inner loop is sequential, each of the n processors reads from $C[j]$ on the jth iteration. Therefore the final step required is to make each processor read from a different j on each iteration. This can be achieved by offsetting j with each processors' i value. Doing so gives us Algorithm 1.

This algorithm computes the rank in $\theta(n)$ time: lines 2-3 are repeated once in parallel, making them $\theta(1)$, while lines 5-7 are repeated n times in parallel, making them $\theta(n)$. The overall rank-calculation run- time is therefore $\theta(1 + n) = \theta(n)$. As before, lines 8 and 9 take $\theta(1)$, so the overall run-time is also $\theta(n)$. The memory use increases by a constant amount, from $4n$ (n i variables, n j variables, the A array and the $wins$ array) to $6n$ (n i variables, n j variables, n j_i variables, the C array, the A array, and the $wins$ array.)

Algorithm 1 The n processor EREW algorithm for Q1.

1: **for** $i = 0$ to $n - 1$ **do in parallel**
2: $wins[i] \leftarrow 0$
3: $C[i] \leftarrow A[i]$
4: **for** $j = 0$ to $n - 1$ **do**
5: $j_i \leftarrow (i + j) \bmod n$
6: **if** $(A[i] > C[j_i])$ or $(A[i] = C[j_i]$ and $i > j_i)$ **then**
7: $wins[i] \leftarrow wins[i] + 1$
8: **for** $i = 0$ to $n - 1$ **do in parallel**
9: $A[wins[i]] \leftarrow C[i]$

1.2 Part (b)

I do not believe that the algorithm presented here can be made cost optimal by scaling down in this manner. The crux of the issue is the calculation of the rank of an element. The most efficient way to calculate the rank of elements of an array is actually to sort them, which takes $\theta(nlg(n))$ time. If we restrict ourselves to not moving the elements (after all, we're trying to sort them **using** the rank), then the most efficient way is brute force search, which is $\theta(n^2)$. Reducing the run-time of this brute force method by applying parallelism simply requires us to swap time for processors, meaning we end up with an n processor, $\theta(n)$ time algorithm - which is still cost $\theta(n^2)$. As sequential sorting has a best-known cost of $\theta(nlg(n))$, this is not cost optimal.

2 Question 2

To order items without doing traditional sorting (i.e. without using direct comparisons), we must determine a way to calculate each element's rank in the final array, and then move all elements to their rank in one go.

I began by adapting a CREW version of the algorithm used in question 1. The main difference is that an element, x, has a higher rank than another element, y, if: both elements are less than the average, and the position of x is higher than that of y; both elements are greater than or equal to the average, and the position of x is higher than that of y; or element x is greater than or equal to the average value, and element y is less than the average.

This gives Algorithm 2. As the only asymptotic change from the algorithm shown in question 1 is the addition of a $\theta(log(n))$ step in line 1, the overall running time is still $\theta(n)$.

Algorithm 2 An n processor, $\theta(n)$ time, CREW algorithm for Q2.

```
1: sum ← reduction(+, A)
2: avg ← sum/n
3: for i = 0 to n − 1 do in parallel
4:     wins[i] ← 0
5:     for j = 0 to n − 1 do
6:         if A[i] < avg and A[j] < avg and i > j then
7:             wins[i] ← wins[i] + 1
8:         else if A[i] ≥ avg and (i > j or A[j] < avg) then
9:             wins[i] ← wins[i] + 1
10: for i = 0 to n − 1 do in parallel
11:     A[wins[i]] ← A[i]
```

To beat the $\theta(n)$ barrier we need to find a way to calculate the ranks without comparing to every other element. To do this, we can ask ourselves exactly what an element's rank should be, if we were to pick a random element from the array. There are two cases to consider - either the element we picked is less than the average, or it is greater than (or equal to) the average. In the first case, the rank should be the number of less-than elements that are to the left of our chosen element. For the other case, a similar statement is true - the rank should be the number of greater-than-or-equal-to elements that are to the left of our chosen element, **plus** the total number of elements that are less than the average (since they all must come before our element).

These deductions lead us quickly to an algorithm. We can track a running count of less-than and greater-than-or-equal-to elements through the array using two prefix scans. With that information, calculating the rank then becomes a constant amount of effort, as it is a parallelized loop which does a fixed step value calculation. This gives us Algorithm 3.

There are two differences between the above description and the pseudocode, made for algorithmic simplicity:

- A parallel pre-processing loop is introduced to make the prefix operations simpler. A prefix operator could be defined which does a conversion to $\{0, 1\}$ before it sums, but asymptotically there is no difference.
- At each element, the number of items counted (in either *lesser_than* or *greater_than*) includes the element itself, so 1 must be subtracted to correct for this.

Algorithm 3 An n processor, $\theta(log(n))$ time, CREW algorithm for Q2.

```
1:  sum ← reduction(+, A)
2:  avg ← sum/n
3:  for i = 0 to n − 1 do in parallel
4:      if A[i] < avg then
5:          lt[i] ← 1
6:      else
7:          gte[i] ← 1
8:  less_than ← prefix(+, lt)
9:  greater_than ← prefix(+, gte)
10: for i = 0 to n − 1 do in parallel
11:     if A[i] < avg then
12:         rank[i] ← less_than[i] − 1
13:     else
14:         rank[i] ← greater_than[i] + less_than[n − 1] − 1
15:     A[rank[i]] ← A[i]
```

Analysing the algorithm asymptotically, we see that line 1 can be done in $\theta(log(n))$, and line 2 is $\theta(1)$. Lines 4 to 7 are done in parallel, so are $\theta(1)$. Lines 8 and 9 can both be done in $\theta(log(n))$. Finally, lines 11 to 15 are done in parallel, so are $\theta(1)$. Altogether we have:

$$\begin{aligned} time &= \theta(log(n)) + \theta(1) + \theta(1) + \theta(log(n)) + \theta(1) \\ &= \theta(log(n) + 1 + 1 + log(n) + 1) \\ &= \theta(log(n)) \end{aligned}$$

The final question is whether the algorithm is cost optimal. First, we calculate the cost of the algorithm presented above:

$$\begin{aligned} cost &= number\ processors \times time \\ &= \theta(n) \times \theta(log(n)) \\ &= \theta(nlog(n)) \end{aligned}$$

An algorithm is cost optimal if it's cost is at most the run time of the best known sequential algorithm. By converting Algorithm 3 to run sequentially, we find that it has a $\theta(n)$ run-time (see Appendix A for the analysis), which gives us an upper bound on the problem. Since the cost of the parallel algorithm is asymptotically worse than $\theta(n)$, it is **not** cost optimal.

I believe the algorithm can be made cost optimal. By applying Brent's theorem, we can see that a cost of $\theta(n)$ is possible by using $\theta(\frac{n}{log(n)})$ processors:

$$\begin{aligned} t' &= t + \frac{m-t}{p} \\ &= \theta(log(n)) + \frac{\theta(n) - \theta(log(n))}{p} \\ &= \theta(log(n)) + \frac{\theta(n)}{p} \\ cost &= p \times t' \\ &= \theta(plog(n) + n) \end{aligned}$$

Now we must see if we can apply this algorithm with only $\theta(\frac{n}{log(n)})$ (for example, it might be that to get a cost of $\theta(n)$ we must use a sorting approach, which is not allowed). Firstly, both prefix and reduction can be implemented in a cost-optimal manner using $\theta(\frac{n}{log(n)})$ processors[1, 2][1] The remainder of the algorithm is actually made up of two sections which are cost optimal: isolating each of the parallel loops shows that they each have time $\theta(1)$ and cost $\theta(n)$. Since the equivalent sequential loops (the 'best known algorithms') takes $\theta(n)$ time, the parallel loops are each cost optimal. Therefore, the entire algorithm can be made cost optimal by using $\theta(\frac{n}{log(n)})$ processors, using the cost optimal versions of *prefix* and *reduction*, and using round-robin scheduling for the parallel loops.

[1] I could not find the exact paper which showed that reduction can be done cost optimally on $\theta(\frac{n}{log(n)})$ processors, but it can be extrapolated from the fact that we can do summation cost optimally with $\theta(\frac{n}{log(n)})$ processors.

A Sequential Analysis of Algorithm 3

Line 1 is $\theta(n)$. Line 2 is $\theta(1)$. Lines 4 to 7 are each constant time, and are repeated n times, so are $\theta(n)$ altogether. Lines 8 and 9 are each $\theta(n)$. Finally, lines 11-15 are $\theta(1)$ each and are repeated n times, so are $\theta(n)$. Note also that the assign to $A[rank[i]]$ would have to be done to a **copy** of the A array to avoid overwriting memory that we still need. This then requires a third for- loop, to copy the memory back to A at the end. The final for-loop requires an extra $\theta(n)$ time. Therefore, altogether we have:

$$\begin{aligned} time &= \theta(n) + \theta(1) + \theta(n) + \theta(n) + \theta(n) + \theta(n) \\ &= \theta(n + 1 + n + n + n + n) \\ &= \theta(n) \end{aligned}$$

References

[1] Richard E. Ladner, Michael J. Fischer. *Parallel Prefix Computation*, Journal of the ACM, Oct 1980.

[2] *Design and Analysis of Parallel Algorithms* course notes. http://www.inf.ed.ac.uk/teaching/courses/dapa/overheads.pdf