Exercise 3

3.1

The **insert** operation can result in concurrent write if it is not carefully designed, given the case where $\textbf{index}(A[i]) == \textbf{index}(A[j])$.

The other parts are guaranteed to be thread-safe.

3.2

For a parallel version of **copy**(A, B), we will need the help of another array B.

```
procedure PARALLEL_COPY(A, B)
    ▷ C stores the size of each bucket in B
    C ← Array[1..nb]
    pfill(C, 0)
    for i=1..nb in parallel do
        C[i] ← len(B[i])
    end
    P ← pPrefAdd(C)
    for i=1..nb-1 in parallel do
        ▷ we set P[0] as 0
        beg ← P[i − 1] + 1
        end ← P[i]
        for j=beg..end in parallel do
            A[j] ← B[i][j − beg]
        end
    end
end
```

3.2.1 Comments

- For the beg and end in the above algorithm, we note that $(P[i] - (P[i-1]+1)+1)$ is equal to $C[i]$, namely $\textbf{len}(B[i])$.
- I don't see the necessity of using **pfill**$()$. It's likely used for avoiding data corruption.

3.2.2 Complexity Analysis

Operation	*Time Complexity*	*Processors Required*
Get the lengths	$O(1)$	nb
Parallel Prefix Sum	$O(\log \text{nb})$	nb/2
Insert	$O(1)$	n
Overall	$O(\log \text{nb})$	n

For fast Parallel Bucket Sort, we need enough number of buckets. However, when it comes to the parallel copy operation, a lower nb is preferred. Herein lies a tradeoff.

Exercise 5

(a)

Given an index i, to find the final position j of a_i in the merged list, we can use Binary Search to find the biggest element b_k in B that is smaller than a_i (or the smallest element in B that is bigger than a_i), which requires $O(\log n)$ comparisons. (See section **Parallel Merge sort** in the lecture slide for example) Then, $j = i + k$, as there are $i + k$ elements in total that are smaller than a_i.

(b)

Denote by $\mathbf{computeInsertPos}(A, B, i)$ the algorithm described in (a), which returns the final position j to insert our $a_i = A[i]$.

```
procedure PARALLEL_MERGE(A,B)
    C ← Array[1..2n]
    for i = 1..n in parallel do
        ▷ j1 ≠ j2 as the elements are unique
        j1 ← computeInsertPos(A, B, i)
        j2 ← computeInsertPos(B, A, i)
        C[j1] ← A[i]
        C[j2] ← B[i]
    end
end
```

(b.1) Comments

- If the elements are not unique and $j_1 == j_2$ by accident, we can set j_2 as $j_1 + 1$ to avoid data race.
- The total work T_1 is apparently $O(n \log n)$ and the span T_∞ is $O(\log n)$.

(c)

This task seems impossible from the software perspective. With $O(n)$ processors, we can compare a_i with $b_{1..n}$ in $O(1)$. (but don't we need to broadcast a_i to all the processors?)

That said, it's still not possible to reduce the results in $O(1)$. Consider the following 2 cases:

1. $\max\limits_{\{k : a_i \geq b_k\}} k$;
2. $\sum\limits_{\kappa \in \{k : a_i \geq b_k\}} 1.$

In both cases, the reduction takes $O(\log \#\{k : a_i \geq b_k\})$. $O(1)$ is only made possible if we have some hardware "hacks" or some magic.

If such a hack exists, we can use $n \cdot O(n) = O(n^2)$ processors for merging two lists in $O(1)$.