Written Assignments (45 points)

1.1. Parallelizing the *IsKColorable* Algorithm for Undirected Graphs (25 points)

1.1.1.

Algorithm 1: IsKColorableParallel

Input: Graph $g = (V, E)$ and $k \in \mathbb{N}$
Output: $\exists f : V \to [k], \forall \{u, v\} \in E, f(u) \neq f(v)$

```
1  colorable ← True;
2  finish {
3      foreach Assignment f of a value in [k] to each node in V do {
4          async {
5              foreach {u, v} ∈ E do {
6                  async {
7                      if f(u) = f(v) then
8                          colorable ← False;
9                  }
10             }
11         }
12     }
13 }
14 return colorable;
```

1.1.2. WORK(G) = $O(n^2/2)$.

1.1.3. In the worst case scenario, there will only be one processor available to execute my algorithm. If this were the case, then the outer **foreach** loop would execute one node at a time and the inner **foreach** loop would execute one edge at a time.

In other words, my algorithm would operate the same as the original isKColorable algorithm, sequentially. Therefore, in the worst case, my algorithm will perform the same amount of WORK as the sequential version, and it is not possible for my algorithm to have a larger value of WORK than the sequential version.

1.1.4. Because WORK(G) is defined as the sum of time TIME(N) for all N nodes in a computational graph, parallelizing isKColorable would allow computations to occur at the same time, decreasing the total sum of time TIME(N), and therefore decreasing the amount of WORK that is being performed.

For example, if I were to implement my algorithm using n processors, where n is the number of nodes in the graph, then the outer **foreach** loop would take up only 1 unit of time, as all n nodes in V are being visited at the same time. We would then need to take into account the inner **foreach** loop that will loop $n/2$ times (because there are $n/2$ edges) for each node being visited. Therefore, the total time this would take would be measured by $1 \times n/2 = n/2$, so the value of WORK would be $O(n/2)$, which is less than the sequential version, which has a WORK value of $O(n^2/2)$.

1.2. Parallel Fibonacci using Futures (20 points)

1.2.1. WORK(n) = $O(\varphi^n)$.

1.2.2. CPL(n) = $O(n+1) \approx O(n)$.
Explanation: Once get(n) is called on a future for some value n, all subsequent calls of get() for that value will complete immediately because the computation has already been done. And because fib() is called on all values between 0 and n, inclusively, at most once with the use of futures, and one unit of WORK is done for each fib() call, the sum of execution times of all nodes, CPL(G), would be n+1.

1.2.3. CPL(n) = $O(n+1) \approx O(n)$.
Explanation: Bob's implementation creates a stream of futures to compute fib() from 0 to MaxMemo-1, however, these futures are not computed unless asked to. The value of the call to fib() would then be stored in cache. Therefore, similar to the parallel Fibonacci function, because fib() is called on all values between 0 and n, inclusively, at most once with the use of futures, and one unit of WORK is done for each fib() call, the sum of execution times of all nodes, CPL(G), would be n+1. Bob's implementation would be faster than the sequential implementation of the Fibonacci sequence because with the use of futures, you do not have to wait for the previous call of fib() to finish before you compute the next. They run asynchronously. Additionally, with memoization, all calls of fib() are stored, so that they are able to be instantly retrieved with any following calls with the same input.