Parallel Programming Languages and Systems, Assignment 1

s1140740

1 Question 1

At the start of the program `P1` one would execute the loop continuously while `x` is not equal to `y` and `P2` would busy wait until `x` is equal to `y`. When `x` becomes 0, that is when `x = x - 1` is executed and `x = 0` is stored in memory from `P1`, there are four possible combinations of values process `P1` reads from memory before it checks whether `x` is equal to `y` as part of the while loop.

1. **`P1` reads its own value of `x` and `y`.** Thus, `x = 0` and `y = 0` and `P1` will break out of the loop. There are two possibilities now:

   (a) **`P1` executes `y = y + 1` before `P2` executes `<await (x==y);>`.** In this case, `x = 0` and `y = 1`. Since `P1` is done and `x != y`, `P2` will never execute and the program will not terminate.

   (b) **`P2` executes `<await (x==y);>` before `P1` executes `y = y + 1`.** After the loop is exited what remains in `P1` is to read the value of `y`, increment it, and store it in memory. Moreover, there is no dependency on `x` in the rest of `P1`, thus its value will be the same in all three cases. Since only the read and the store are memory operations there are only three possible times when value of `y` from `P2` can be stored:

      i. **Before `P1` reads `y`.** `P1` will read value of `y` from `P2`, increment it by one and store it. Thus the program will terminate with `x = 8` and `y = 3`.

      ii. **After `P1` reads `y` but before it stores `y`.** `P1` will read its own value of `y`, that is 0, increment it to 1 while `P2` stores `y = 2`, and then store `y = 1`, overriding `P2`'s value. Thus the program will terminate with `x = 8` and `y = 1`.

      iii. **After `P1` stores `y`.** `P1` will read its own value of `y`, that is 0, increment it to 1, and store `y = 1`. After this, `P2` stores `y = 2`. Thus the program will terminate with `x = 8` and `y = 2`.

2. **`P1` reads values of `x` and `y` from `P2`.** Thus, `x = 8` and `y = 2`. Since all instructions of `P2` have executed, only `P1` remains active. Further, since `8 != 2`, `P1` will decrement `x` until `x == y`, that is until both `x` and `y` are equal to 2 (`P2` is done and there is no one to modify value of `y`). At that point it will exit the loop. Last instruction of `P1` is `y = y + 1` and hence the program will terminate with `x = 2` and `y = 3`.

3. **`P1` reads its own value of `x` but value of `y` from `P2`.** Thus, `x = 0` and `y = 2`. Since all instructions of `P2` have executed, only `P1` remains active. Further, since `0 != 2`, `P1` will enter the body of the loop and read value of `x` stored by `P2`, that is 8. It will therefore start decrementing `x` until it becomes 2 at which point it will exit the loop and the program will terminate with `x = 2` and `y = 3`.

4. **`P1` reads its own value of `y` but value of `x` from `P2`.** Thus, `x = 8` and `y = 0`. Since `8 != 2`, `P1` will enter the loop and start decrementing `x`. Depending on when the value of `y` from `P2` is stored in memory this program will behave differently:

   (a) **`P2` stores `y = 2` before `x` becomes 2.** `P1` will decrement `x` until it becomes 2 when it will exit the loop, increment the value of `y` to be 3 and the program will terminate with `x = 2` and `y = 3`.

   (b) **`P2` stores `y = 2` after `x` becomes 2 but before `P1` breaks out of the while loop.** Hence, `x < y` and thus `P1` will decrement `x` to negative infinity and the program will never terminate.

   (c) **`P2` stores `y = 2` after `P1` breaks out of the while loop.** This way `P1` is in the same situation as in case 1.b (except `x` is 0 now) and depending on when `P2` stores `y` we get three possible outcomes, each with different value of y:

      i. **Before `P1` reads `y`.** Program terminates with `x = 0`, `y = 3`.

      ii. **After `P1` reads `y` but before it stores `y`.** Program terminates with `x = 0`, `y = 1`.

      iii. **After `P1` stores `y`.** Program terminates with `x = 0`, `y = 2`.

2 Question 2

- use barriers to synchronise rounds
- degree is immutable, no sync needed
- rndvalue and first legal colour are same within the same round
- after activity compares its colour, degree and rndvalue with others it will either stop or hit a barrier
- when all activities hit the barrier, next round can progress (`usedcolor` and *first legal colour* will be different now)

The DLF algorithm is synchronised in rounds. To achieve this synchronisation among different parallel activities, barriers could be used. Thus all threads that finished executing the current round would have to wait for other threads to finish the round and only then be able to continue.

There are three variables that each activity shares with other activities: the degree of the vertex it represents $d(v)$, the randomly generated value $rndvalue(v)$ and the first legal colour. The degree and randomly generated value are immutable variables that are not written to after they are generated. Thus there does not need to be any synchronisation wrapping them and other activities could just read them from shared memory. The first legal colour, however, needs to be synchronised to ensure that

Within each round every uncoloured vertex v executes the following five steps:

1. Choose parameter $rndvalue(v)$ uniformly distributed on [0..1]
2. Determine what the first legal colour is.
3. Barrier here?
4. Read $deg(v)$, $rndvalue(v)$, and first legal colour of its neighbours.
5. Compare its own parameters with those received from its neighbours and check which vertex has the highest priority.
6. If vertex v's proposed colour does not clash with proposals from its neighbours or if v has the highest priority amongst its neighbours, keep the proposed colour, inform neighbours of the choice by adding it to their `usedcolor`, and stop.

7. Otherwise, reach a barrier and wait for other activities in the current thread to reach the barrier as well. When they do, stop.