

# 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  $\langle \text{await } (x==y); \rangle$ .** In this case,  $x = 0$  and  $y = 1$ . Since P1 is done and  $x \neq y$ , P2 will never execute and the program will not terminate.
  - (b) **P2 executes  $\langle \text{await } (x==y); \rangle$  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  $8 \neq 2$ , 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  $0 \neq 2$ , 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  $8 \neq 2$ , 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

The DLF algorithm is synchronised in rounds. To achieve this synchronisation among threads, 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 on to the next round. Moreover, we need to ensure that threads do not read other threads' values from a previous round. Hence, we add a set-up phase during which variables for this round are set and that ends with a barrier as well. There are four parameters that each thread shares with other threads:

- **The degree of the vertex it represents  $d(v)$ .** This is an immutable variable that does not change throughout the execution of the algorithm, hence no synchronisation is required for it.
- **The randomly generated value  $rndvalue(v)$  and the first legal colour.** The values of these parameters stay the same within a round but we need to ensure no thread is reading their values from the previous round. Thus, we only allow modification of these parameters in the set-up phase which ends with a barrier. This way, the parameters will be set for the current round and once the round is over the end-of-round barrier will be hit and the parameters will be re-set again only after the end-of-round barrier is passed and set-up phase of the next round is in progress.
- **The palette of colours used by neighbours  $usedcolor(v)$ .** This list is used in the set-up phase to determine the first legal colour of a vertex. Furthermore, it is accessed during the body of the round by threads representing the neighbours of the vertex to inform the vertex about the neighbours' final colour choices. Hence, it could potentially be written to by multiple threads and we need to ensure that only one thread is accessing the list at a time to avoid leaving it in inconsistent state. This could be achieved by a mutual exclusion variable with the use of a lock, semaphore or even a monitor. This way, the list is read only during the set-up phase and other threads can modify it only during the body of the round. Hence after the round is over and the set-up phase of the next round begins the list contains all the colours claimed by neighbours up to and including the current round.

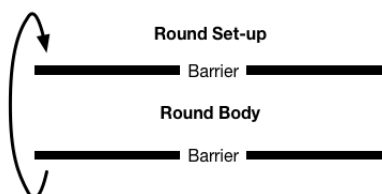


Figure 1: A round of the algorithm

Using one thread for multiple vertices could lead to efficiency problems. For example, let's say that thread  $A$  which is responsible for vertex  $v$  wants to update  $usedcolor(u)$  where  $u$  is represented by thread  $B$ . Let's also say, that the lock to  $usedcolor(u)$  is currently being held by thread  $C$ . Now, thread  $A$  has to wait until thread  $C$  releases the lock to continue executing code for vertex  $v$  and any other vertices it represents. If we had a 1 to 1 mapping of vertices to threads the code for vertices currently represented by thread  $A$  could execute concurrently.