Parallel Programming Languages and Systems
Exercise 1

s0840449

1 Question 1

```
1 int lock = 1;
2 co [i = 1 to n] {
3     while (lock < 1 || DEC(lock)) { }
4     /* Start critical section. */
5     ...
6     lock = 1;
7     /* End critical section. */
8 }
```

This answer is essentially a reimplementation of ''Test and Test and Set' using the `DEC` operation. The core difference is that *true* and *false* for the lock are redefined as 1 and < 1 respectively. This approach satisfies all of the required properties:

Mutual Exclusion

Assume that two threads are in the critical section simultaneously. For this to happen, `lock` must have been `>= 1` for both threads, and `DEC(lock)` must have returned false. `DEC(lock)` returns false if and only if $lock - 1 >= 0$, or equivalently $lock >= 1$. Since `DEC` also decrements *lock* at the same time, and *does so atomically*, `lock` must have been `>= 2` when the first of the two threads called `DEC(lock)`. However, `lock` starts by being initialized to 1, and can then only be `<= 1`: any call to `DEC` will decrement its value, and a thread exiting the critical section will set it back to 1. Therefore it is impossible for `lock` to be 2, and so it is impossible for the two threads to both be in the critical section.

Absence of Deadlock

Assume that two threads wish to enter the critical section (either newly or from within the `while` loop), but cannot - i.e. they are deadlocked. For this to happen, each thread must have either $lock < 1$ or $DEC(lock) == true$. For $lock < 1$, `DEC` must have been called by some other thread. Whichever thread called it must also still be in the critical section, as if it had exited then `lock` would be set to 1. In that case, there is no deadlock: another thread is in the critical section. So $lock < 1$ must be false for both threads. Therefore, `DEC` must be returning true for both threads. `DEC` only returns true if the result of $lock - 1$ is less than 0; i.e. if $lock < 1$. We have just seen that this cannot be the case for both threads, so `DEC` would return true for one of them, and therefore there is no deadlock.

Absence of Unnecessary Delay

There are two situations to be considered: the first time a thread reaches the critical section, and when a thread tries to enter the critical section sometime after a thread has exited the critical section. In either case, we must have `(lock < 1 || DEC(lock)) == false`.

When the very first thread tries to enter the critical section, `lock` is set to 1. As such, both the left and right sides of the condition evaluate to false: $1 \not< 1$, and $1 - 1 = 0 \geq 0$. Therefore, the thread will enter the critical section, and will not be unduly delayed.

If another thread has already been through the critical section, it must have set $lock = 1$. Therefore, for the second case the same logic holds, and the thread will not be unduly delayed.

Efficiency

The solution minimizes cache coherence traffic by avoiding many calls to the `DEC` function (i.e. avoiding writing to the shared variable.) `lock` can either be 1 or `< 1`, and a thread may only enter if it is 1. Therefore, checking `lock` < 1 is sufficient to determine whether there is no point trying to obtain the lock, or if the thread has a chance to obtain the lock. It should be noted that this only works in a practical setting. Theoretically the code could execute such that the check of `lock` < 1 always evaluates to false, i.e. if waiting threads happened to only ever check the condition when a thread left the critical section, and if they all then checked `lock` < 1 before any called `DEC`. However, practically speaking this is *very* unlikely, and actually means that you have been lucky enough to avoid spin-locking, so is probably efficient enough that you would not care!

2 Question 2

The main difference in a shared variable version of the 'Distributed Largest- First' (DLF) algorithm is how proactive processes can be. Due to the use of shared memory processes are able to 'snoop' details on other nodes (such as their chosen colour or random value.) This removes the need to pass this information using messages but means that processes must be synchronized to avoid data for a round being accessed before it has been written or overwritten before it has been read. The DLF algorithm could be mapped to either the *interacting peers* paradigm or, at a slight stretch, the *bag of tasks* paradigm.

Interacting peers (IP) is a natural fit for the DLF algorithm, as each node is assigned to a peer (process), and data from neighbouring peers (nodes) is used to perform the computation for that node. The 'rounds' described in the DLF algorithm are equivalent to loop iterations in IP. Each round can be split into the three IP stages - setup, computation, and termination - which are separated by barriers to enforce data synchronization. The *setup* stage would choose the random value and colour

proposal for that round. The *computation* stage would check neighbouring nodes to find overlapping choices of colour. Finally, the *termination* stage would check if all nodes had a colour chosen. Note that in IP all processes normally terminate at the same time. This is not true for the DLF algorithm; once a process has a set colour it should terminate immediately rather than unnecessarily execute further rounds. To support this in IP the barriers would have to be able to cope with a changing number of processes, e.g. by allowing a process to tell the barrier it is finished.

The DLF algorithm could also be implemented using a bag of tasks (BoT) approach. To do so, each round for each node would be considered a single task that a worker process would execute (returning either the colour that the node should take, or that the node failed and needs another round.) There are two complications which reduce the clarity of the mapping. Firstly, the setup for each round (chosing a colour and random value) cannot be part of the task, as *all* nodes must choose a colour before *any* comparisons can be done. Also, while any worker is executing a 'round N' task no 'round N + 1' tasks can be issued by the farmer or both round N and N + 1 tasks would have incorrect data. The latter requirement is not actually a problem - the original DLF algorithm involves waiting until all nodes have finished each round anyway, so no time is lost - but fixing the setup problem is messy in BoT. Two possible solutions are to have the farmer perform setup for each node in-between rounds (which wastes worker time and may be slow for many nodes) or to consider the setups to be independent task sets executed between rounds (which requires more communication with workers).

The DLF algorithm does not map neatly to the other paradigms covered in PPLS. The *pipeline* paradigm has a similar concept of rounds, however their purpose is different. In pipeline parallelism, data item i passes through round N while data item i + 1 is passing through round N - 1 (and so on). In contrast, in the DLF algorithm all data items (nodes) pass through round N -1 at the same time, then all through round N, and so on. The *producer-consumer* paradigm also does not fit, as there are no 'producer' or 'consumer' processes - each process both creates and consumes data for and from other processes.

The difficulties encountered in extending the DLF algorithm to cope with more nodes than processors depends on the paradigm used.[1] Coping with multiple nodes per process in IP would require the nodes to be split among the peers. Each peer would then perform setup for all of its nodes before the first barrier, then computation for all of its nodes, and so on. The difficulty here in the case of a large graph would be balancing the execution. It is possible for one peer to find colours for its nodes very quickly, while another peer has nodes that take a long time each. Some form of 'node-stealing' between rounds might be appropriate to alleviate that possibility. In the case of the BoT paradigm there is already no connection between the worker processes and nodes (any worker may execute a round for any node) so no changes are required!

[1]It is assumed that the number of processes is less than the number of nodes (i.e. that a processor executes only one thread). If this were not the case then a thread could be launched for each node, resulting in the default case explored above (except with *logical concurrency* rather than physical.)