Report

Assignment 3

Q- Explanation of the added lock primitives to ensure that the linked list is thread-safe.

A. : We will first describe the requirements of a thread-safe linked list, and then describe how our additions apply to each operation.

A thread-safe code is applicable to multi-threaded programs because it only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfil their design specifications without unintended interaction. When a code is thread-safe, implementation is guaranteed to be free of race conditions when accessed by multiple threads simultaneously. Then, in a case of a linked list, making it thread-safe guarantees that two threads can't access the same ressources at the same time, which would cause, if it was the case, some errors or undefined behaviour into the linked list.

A current way to make a linked list thread-safe is to use locks. In this assignment, we were restricted to OpenMP locks and we were not allowed to use barriers, critical sections, or atomics to make the singly linked list thread-safe.
In a linked list, appart from the list initialization function (init_list), we have three types of operations : inserting an element into the list (append ; add_first ; insert), deleting an element from the list (remove_by_value ; remove_by_index ; delete_list ; pop) and iterating through the list to perform a specific computation (search_by_value ; print_list ; count). The main requirement is to make the threads able to hold a lock on different specific nodes of the list at each iteration (we are talking about iterations because to reach a node of the list, we usually have to start from the head). Then we decided to add a lock of type « omp_lock_t » as an additional attribute of a node, in the node structure, to make the threads able to hold a lock on each node of the linked list.

We created a parallel region where 4 different threads sharing the same « test_list » pointer operate. Every functions that operate on the list are executed into this parallel region. Since we want to avoid memory leaks, we decided to find a way to execute « init_list » only once despite the fact that there are several threads. One way to do this is to declare and initialize a lock, which is not associated to any node, outside the parallel region (same for its destruction) and then let the first thread get the lock on it without blocking the other threads. To ensure no other late threads get also the lock, we unset the lock outside the parallel region, so it can't be unset before every threads finishes operating on the list. Note that to be unset, because the unset call is outside the parallel region, and that it's the same thread which set the lock, that is able to unset it, the only way that this lock can be unset is that the master thread set it. But because it doesn't matter if a lock is unset at the end of an execution, and that it will be destroyed anyway, we can proceed that way.

For inserting and iterating operations such as « append », « add_first », « insert », « search_by_value », « print_list » and « count », starting from the head, we only lock the current node the thread operates on, since that this is the only node that may be modified (it's pointer to the next node will point to a different one when inserting a new node after him), or analyzed (printing its value, returning it's index or just counting it). Note that we lock the head in the function « add_first » because the added node is going to be the new head.
The deleting operations such as « remove_by_value », « remove_by_index », « delete_list » and « pop » is a little bit different : we have to lock the current node we are on (because we are operating on it, and we may be need to free it, if this is the desired node to be removed), but also its previous node. Indeed, because we stop the iteration on the node we want to remove, its previous node will be updated (its pointer to the next node, which is the node being removed, will point to a different node, which will be the next node of the removed one). Note that we must lock every nodes in the function « delete_list » because we will modify each of them (they are all going to be freed, and then we don't want an other thread comes modify them during the operation).

No line of code that was already in the C file has been removed or modified. We only add some, and each of them has its own comment starting with « // » on the same line. The lines of code that were initially in the file are still executed in the same order.

We are still encountering an issue when running the code. Sometimes we get double free errors, or infinite loop. The double free error certainly occurs after the call to the function « delete_list » : the output clearly shows that the function doesn't always free and set to NULL the removed nodes correctly (instead of having an empty list where no value of nodes appears, we have really high or really low values appearing in the list that is supposed to be deleted), such that, after, when an other thread executes this function, the lists doesn't act as it's empty, and the unexpected existing nodes into this list are not seen as NULL. Then, this thread tries to free them, and we obviously get a double free error which states that the same pointer is about to be freed two times without beeing reallocated after the first free. This issue can happen when a thread is waiting to set a lock on a node that is being modified by an other thread. When the lock is unset, the waiting thread can now get the lock, but the locked node could have been, for example, deleted, and then the thread that just got the lock, can be doing undefined operations causing inifite loop, or resulting in a double free error.

A solution to solve this problem would have been to use « omp_test_lock » function in a « while » loop, instead of « omp_set_lock », and update the value of the node that the thread tries to lock at each time the test lock fails. But since we had troubles to get the updated value of each node of the list, for each function, we couldn't find a way to fix this. The code can sometimes still execute and finish completely without error but with strange nodes values printed, due to « delete_list » function.