Solving a 15 puzzle

Rahul Ganesh Prabhu

2018A7PS0193P

The Problem

A 15 puzzle is a problem whose objective is to transform an arrangement of 15 tiles from an initial position into the position given below:

We do this using a parallel A* Search. An A* search involves an open list (implemented as a min-heap) and a closed list (implemented as a hash table). Every thread extracts work from the work queue, expands it to all possible successors, and inserts the successors into the open list if it hasn't been encountered (which we check from the closed list).

We implement this with one open list, and then k open lists. In the second case, the k open lists are accessed randomly.

Partitioning

We partition the problem into tasks by defining a task as generating new successor boards from a given board configuration (along with checking if they are solved). This division is done by means of exploratory decomposition. Every thread generates new successor board configurations, hence broadening the search tree and creating more tasks via exploration. Other threads then recursively do the same to the new tasks by taking them from an open list. This approach will find a solution, but may not always find the *optimal* solution.

Communication

Since we use the POSIX threads (`pthreads`) framework, we do not need any explicit communication scheme. All threads have access to the same open and closed lists via global pointers. The data is kept consistent across threads and saved from race conditions using mutex locks, forcing serial behavior in critical areas.

Agglomeration

We do not need to agglomerate the tasks in this problem, as they are of the perfect size.

Mapping

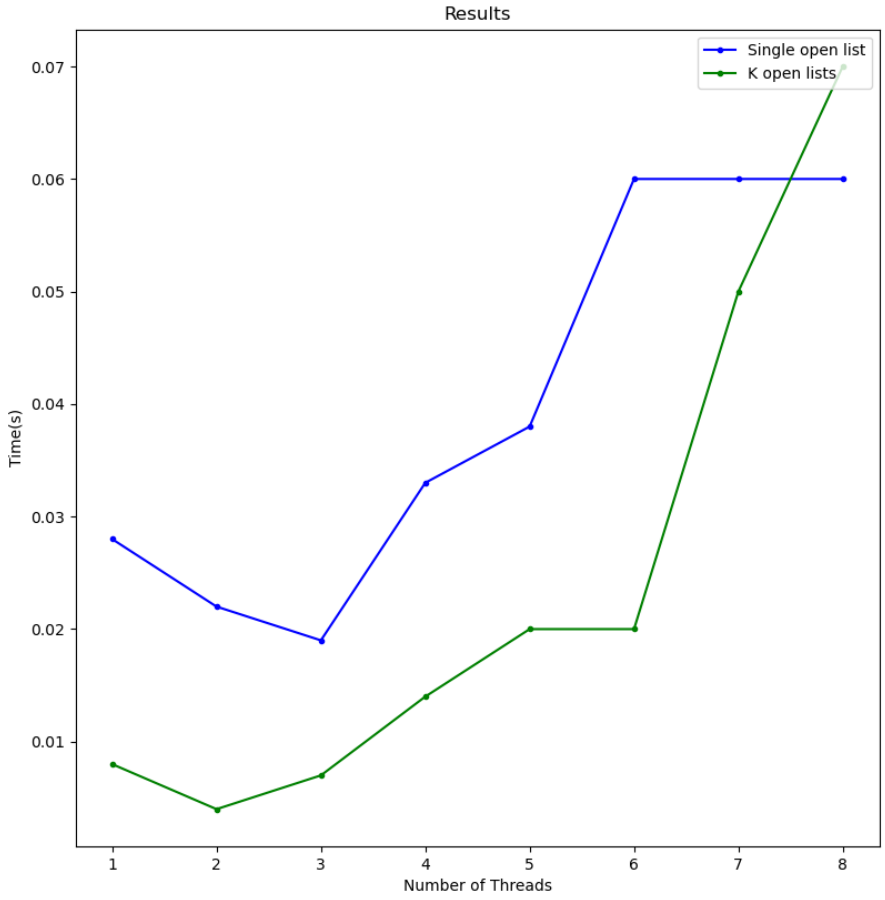
There is no explicit mapping scheme here. Every thread is allotted a task whenever the open list contains board configurations to explore. If there is no task to allot, the threads wait for new tasks to be added to the list. One disadvantage of this approach is that some threads waste time waiting for a new task and lay idle.

Results

The implementation here is difficult to test, since optimal solutions to any 15-puzzle can be reached in 80 moves or less. This means that it usually takes less than a second to solve the problem, and the possible benefits from

using more threads is not visible. Instead, they become a *detriment*, blocking other threads and often remaining idle.

Nevertheless, the results are given below. Case (b) was tested with $k = 16$ open lists.



As we can see, as we increase the number of threads, after a point we have a degradation in performance. However, it is difficult to make any inferences from this, due to the order of magnitude of time being so low.

An inference we can make is that there is a noticeable speedup on using k open lists over a single one. This is because when we used a single open list, there was a lot of blocking due to contention and hence a slowdown. This is

mitigated by the use of multiple open lists.