Parallel Computing (CS F422)
II Semester 2019-2020
Assignment 2
Design Document (P1)

Execution Details:

Step 1: Specify the initial position in the text file "input.txt".

Input Format: It is written as a grid of 4x4 grid of integers from 0 to 15(Black space is specified as **0**). For example,

1 2 4 8
5 0 6 3
9 10 7 12
13 14 11 15

Step 2: Compile and run the .c files.

2.1. For puzzle_a:

- **2.1.1.** gcc –g puzzle_a.c –o puzzle_a –lpthread
- **2.1.2.** ./puzzle_a <maximum_number_of_threads>

2.1. For puzzle_b:

- **2.1.1.** gcc –g puzzle_b.c –o puzzle_b –lpthread
- **2.1.2.** ./puzzle_b <maximum_number_of_threads>

(NOTE: <maximum_number_of_threads> is a command line argument passed. It should be kept below *MAX_THREADS* defined as macro in the respective .c files)

Step 3: Interpretation of the output.

The output contains the following:

3.1. Solution: It specifies the sequence of moves required to reach the desired orientation. It is a string consisting of characters, 'L', 'R', 'U', 'D' signifying that the blank element should be exchanged with the block present on its left, right, up, down respectively.

3.2. Time taken: It signifies the time required by the program in finding the solution.

(NOTE: Solution may be different on different runs of the programs, as the path taken by the algorithm is dependent on spawning of the threads)

Design of the algorithms:

The 15-puzzle problem is based on a puzzle whose aim is to orient the blocks into the specified order by moving the blank block, exchanging its position with adjacent blocks.

This problem is based on exploratory technique of decomposition. Imagining any given state of the puzzle as a node in a tree, and is connected to the nodes corresponding to the states that are reachable from the current states using possible moves on the grid. The puzzle solver can be

interpreted as a tree search problem, search space consisting of all possible paths towards reaching the final orientation. Using exploratory decomposition, the search space is partitioned into smaller parts executing independently.

Size of state space = 16! ≈ 2.10e13

Parallelism

Performing unparalleled search heuristics on such a large state space may consume a lot of memory and time resources. Consider a tree with branching factor *f* and of depth *d*, for a single thread,

$$T_1 = 1 + f + f^2 + f^3 + f^4 + \cdots + f^d.$$

Since for searching the backtracking is performed on the whole space by the single thread.

Now if we consider *p* threads, the backtracking can be performed independently by each of the threads on separate subtrees of the tree. Suppose for some level **k = pr**.

$$T_p = 1 + f + f^2 + \cdots + (f^k + f^{k+1} + \cdots + f^d)/p \approx f^d/p$$

Therefore,

$$T_p \approx \frac{T_1}{p}$$

Best First Search

Using traditional tree traversal may not help reduce our memory and time utilization over a factor of *p*. so instead of just doing the breadth first traversal on out state tree. So we use a priority based traversal based on the Dijkstra's algorithm.

Priority metric used is **Manhattan distance,** for a state s

$$dist(s) = \sum_{i=0}^{15} |x_{xurr} - x_{expected}| + |y_{xurr} - y_{expected}|$$

At any time, every thread extracts the state with least Manhattan distance from the work queue and pushes the next possible states that had not been explored before into the work queue.

This search heuristic will help us reduce our search space by a large extent.

Same state can be reached by taking different paths, so this may lead to exploring the same state again and again, which is redundant traversal, so to avoid this situation, as hash of explored states is maintained so that a state is explored only once.

Implementation Aspects:

The above stated heuristics are executed using POSIX threads by UNIX environment.

The following implementation details will form the base for puzzle_a.c and its modification will be stated after that.

The following data structures are maintained globally,

1. Work Queue (Heap)
2. Explored States (Hash Table, collisions handled by chaining)
3. Solution Found (Flag variable)

The corresponding global structures are accessed and altered by all the threads, so we have corresponding mutex for these.

- Check whether the given initial position of the 15-puzzle is a solvable instance or not, it is checked by a function by the main thread itself in O (1) time. If it is not solvable, the user is notified regarding the same and program exits.
- The specified number of threads are created using pthread_create and threads begin executing the tree_search function. Each thread along with the work queue, maintains a local work queue from which it fetches and inserts the states it explores. This will optimize our repeated lock on the global work queue, so we may use a try lock on the global work queue, instead of waiting in the lock queue.
- The specified local work queue has a maximum size, also specified as a macro, if the local work queue has reached that size, it is necessary for the corresponding thread to get a lock on work queue and populate the local states into the global work queue and hash table.
- If the local work queue is either full or empty, a lock is to be acquired on the global work queue and hash table, such that we extract the new state corresponding to minimum Manhattan distance. Otherwise a try lock is executed on the work queue, if lock is not acquired the node extracted is from the local work queue (which is also implemented as a heap data structure).
- Then the thread finds the next possible states from the given state and stores in the local work queue and goes to the next iteration.
- Termination Condition: After extracting the node corresponding to minimum distance, it is checked whether It has distance = 0. If termination condition is met, the other threads need to be cancelled.
- Cancellation: Initially, every thread sets its cancellation state as ENABLED using pthread_setcancelstate (), and the thread which received to the solution node, calls cancel for every other node and appropriate cancellation points are placed using pthread_testcancel().

Multiple working queues:

Instead of using a single work queue, we declare multiple work queues and the threads may insert into a randomly selected work queue and extract from another randomly selected queue.

Motivation:

1. When we consider only a single working queue, and define a single mutex for this queue, this may lead to queueing delays due to locks.
2. There may be some cases where Manhattan distance may lead to multiple paths and some of them may not lead to the shortest solution, if we use multiple queues, we may be running multiple paths to the desired state. Therefore, obtaining better(shorter) solution.

Implementation (only the modifications):

- Instead of using a single global heap, we declare an array of structures of size *NUM_OPENLISTS,* declared as a macro in puzzle_b.c, and therefore declared an array of mutex corresponding to each of the mutex.
- Whenever we extract the minimum distance node or insert anything into the heap, we randomly generate an index and acquire lock on the corresponding mutex.