COMS3008: Parallel Computing Assignment

Tau Merand 908096 Vincent Varkevisser 705668

October 24, 2017

Introduction

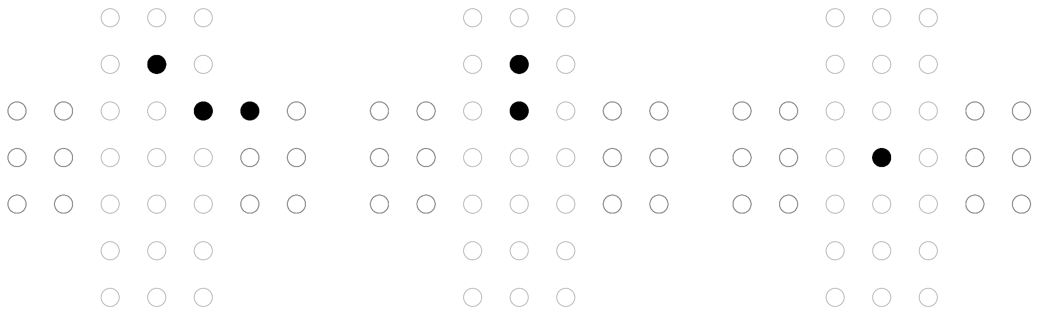
The game of peg solitaire is a one player game played on a 33 holed cross shaped board that involves jumping pegs over other pegs, in a manner similar to checkers. The rules are as follows:

1. A move consists of jumping a peg over an orthogonal neighbor into an empty space. The peg that was jumped over is then removed from the board.

2. Pegs can only jump onto an empty space.

3. The game is won if the final peg is in the centre space.

4. If no pegs can legally move or the final peg is not in the centre the game is lost.



(a) A start state

(b) The right most peg jumps over its left neighbour

(c) The winning state

Figure 1: A winning set of valid moves

Backtracking Search

Recursive backtracking using depth first search was chosen as the method for state space exploration. The standard backtracking depth first algorithm is as follows:

```
Function backtrack
    input  : An initial, possibly un-winnable, board state.
    output: A sequence of moves to get from the initial state to the
            winning state, if a winning state cannot be reached the
            sequence should be empty.

    initialState ← The initial state
    legalMoves ← A list of all the legal moves for initialState
    result ← Empty list to store moves from the initial state to the
      winning state
    foreach move in legalMoves do
        state ← The state after playing move on initialState
        if state is a winning state then
            result ← move
            break
        end
        childResult ← backtrack(state)
        if childResult is not empty then
            Prepend move to childResult
            result ← childResult
            break
        end
    end
    return result
```

Algorithm 1: A standard recursive backtracking using DFS

But because pegs are indistinguishable, game states where pegs are in the same position are identical, regardless of the moves taken to arrive at that state. Thus exploring the state space looking for a sequence of states leading to the winning state will probably involve evaluating the same states many times. Thus a significant speed up can be achieved by saving states that are known to not lead to the winning state as in the following algorithm:

```
Function dynamicBacktrack
    input  : An initial, possibly un-winnable, board state.
    input  : A set of all states that have been searched and are
             known to be un-winnable.
    output: A sequence of moves to get from the initial state to the
             winning state, if a winning state cannot be reached the
             sequence should be empty.
    initialState ← The initial state
    infeasibleSet ← The set of un-winnable states
    legalMoves ← A list of all the legal moves for initialState
    result ← Empty list to store moves from the initial state to the
      winning state
    foreach move in legalMoves do
        state ← The state after playing move on initialState
        if state is in infeasibleSet then
            continue
        end
        if state is a winning state then
            result ← move
            break
        end
        childResult ← dynamicBacktrack(state, infeasibleSet)
        if childResult is not empty then
            Prepend move to childResult
            result ← childResult
            break
        else
            Add state to infeasibleSet
        end
    end
    return result
```

Algorithm 2: Recursive backtracking using DFS and dynamic programming methods

Parallel Implementation

To parallelise the above algorithms it was decided to make use of the OpenMP parallel for loop to allow the concurrent depth first exploration of multiple moves from the initial state. The OpenMP parallel for loop partitions the iteration space of the loop at line 6 of Algorithm 2.

Static, dynamic and guided OpenMP load balancing constructs were all tried, but dynamic gave by far the best performance. This is because dynamic (with default chunk size) gives one element of the iteration space to a thread at a time, allocating new tasks to threads as they finish their allocated task in a round robin approach. This gives good load balancing in this particular problem since some elements of the iteration space may require exploring the game graph very deeply while others may quickly lead to previously seen infeasible states.

The infeasible set is an instance variable of the object and as such each thread can concurrently reference it. Simultaneously, concurrent threads writing to the infeasible set need to avoid data races, thus the function to add a state to the infeasible set must be an OpenMP critical region.

Nested parallelism was also tried and was found to be highly inefficient. Further various upper limits on the depth with which nested parallelism could take place wee tried but the added overhead of parallelism within the recursive algorithm remained slower than no nested parallelism at all.

Results

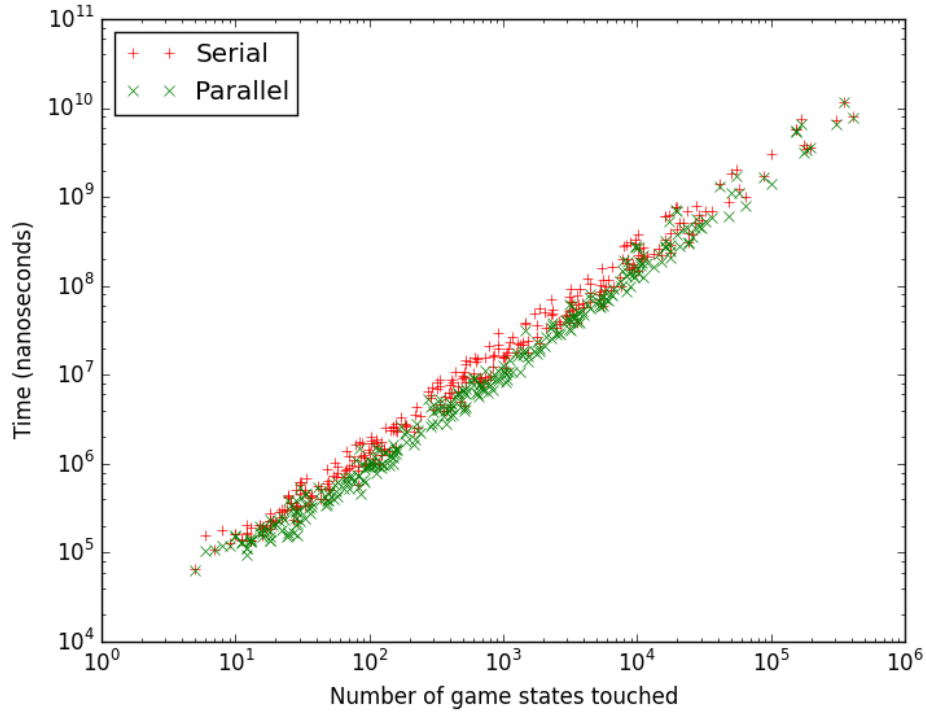


Figure 2: Graph comparing running times of serial and parallel implementations of Algorithm 2

As can be seen in Figure 2 the parallel algorithm is generally faster than the serial, but we did not see the $\frac{n}{p}$ kind of performance gains one naively expects from parallelism. This is possibly due to the recursive nature of our implementation and our dynamic programming approach.