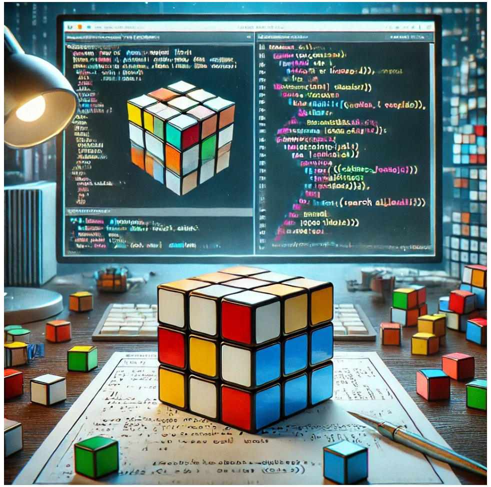
COMS 4995 PARALLEL FUNCTIONAL PROGRAMMING
Stephen Edwards

Rubik’s Cube Solver using IDA*



Hongcheng Tian, Roberto Brera, and Matthew Rosenberg

Cube Representation in Haskell

```haskell
type Color = Char

type Face = [[Color]]
```

```haskell
data Cube = Cube {
    up    :: Face,
    down  :: Face,
    left  :: Face,
    right :: Face,
    front :: Face,
    back  :: Face
} deriving (Eq, Show)
```

```haskell
initCube :: Int -> Cube
initCube n = Cube {
    up = replicate n (replicate n 'W'),
    down = replicate n (replicate n 'Y'),
    left = replicate n (replicate n 'O'),
    right = replicate n (replicate n 'R'),
    front = replicate n (replicate n 'G'),
    back = replicate n (replicate n 'B')
}
```

In our Haskell implementation, a Rubik's Cube is represented by a custom `Cube` data type with six labeled faces: up, down, left, right, front, and back. Each face is a 2D list of colors represented by characters ('R', 'G', 'B', 'Y', 'O', 'W'). The `initCube` function initializes each face with a uniform color. This structure allows easy manipulation, display, and transformation of the cube's state.

Move Representation in Haskell

```haskell
-- Define possible moves
data Move = F | Fi | R | Ri | U | Ui | B | Bi | L | Li | D | Di deriving (Eq, Show)

-- Function to map a Move to its corresponding Cube -> Cube function
applyMove :: Move -> Cube -> Cube

-- Function to apply a list of Moves sequentially to a Cube
applyMoves :: [Move] -> Cube -> Cube
applyMoves moves cube = foldl (\c m -> applyMove m c) cube moves

-- Perform a move that rotates the front face clockwise
moveF :: Cube -> Cube
moveF cube = cube {
    front = rotateFaceClockwise (front cube),
    up = replaceRow (up cube) (n-1) (reverse (getCol (left cube) (n-1))),
    left = replaceCol (left cube) (n-1) downFirstRow,
    down = replaceRow (down cube) 0 (reverse (getCol (right cube) 0)),
    right = replaceCol (right cube) 0 upLastRow
}
  where
    n = length (front cube)
    upLastRow = up cube !! (n-1)
    downFirstRow = down cube !! 0
```

The move logic for manipulating a virtual Rubik's Cube in Haskell involves defining a `Move` data type for possible moves (e.g., `F`, `Fi`, `R`, `Ri`, etc.) and implementing functions to rotate faces 90 degrees clockwise or counterclockwise. Each move function, such as `moveF`, updates the cube's state by manipulating two-dimensional arrays that represent each face of the cube.

Helper functions like `replaceRow`, `replaceCol`, and `getCol` facilitate these updates, ensuring that when a face is rotated, the adjacent faces are also correctly adjusted, maintaining the integrity of the cube's state.

IDA* Algorithm Using Pattern Database

What is IDA*?

- **Iterative Deepening A*** combines the space efficiency of depth-first search with the optimality of A*.
- Searches to a specific threshold *f* = *depth*+*h*, then increases *f* iteratively if no solution is found.
- Uses a heuristic function *h(n)* to guide the search.

Pattern Databases (PDBs)

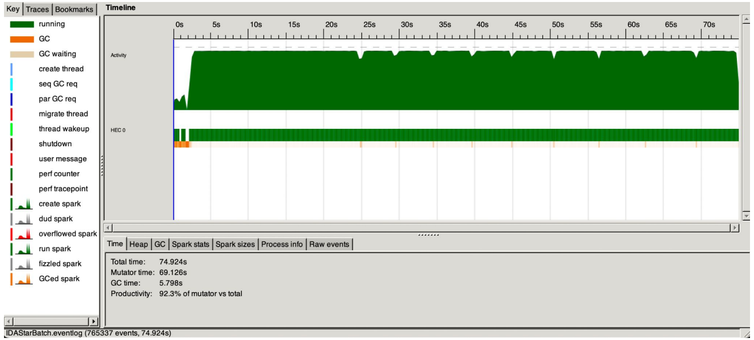
- **Precomputed databases** of optimal distances (minimum number of moves) to the goal for subsets of the cube's pieces.
- Built by performing BFS from the solved state.

Why Use a PDB?

- **Provides admissible and consistent heuristics**: The heuristic never overestimates the cost to the goal. For unseen states, a fallback value is used (e.g. 8 for states not in a PDB with a 7-moves distance limit)
- Admissibility guarantees that solutions found by A*/IDA* are optimal

Applying IDA* with PDBs to the Rubik's Cube

1. **Precompute PDB** for subsets of the cube.
2. During search, estimate cost using the PDB heuristic.
3. Perform an IDA* search using this heuristic.
4. Iteratively deepen the search until the optimal solution is found.
    a. This is guaranteed as the heuristic is admissible



Details of PDB Caching

- Color Conversion* Functions convert Rubik's Cube face colors (chars) to Word8 values and vice versa for efficient storage.
- Cube State Representation: cubeToKey converts a Rubik's Cube state into a Word8Vector, a compact vector of Word8 values, for efficient manipulation and comparison.
- PDB Generation: generatePDB creates a pattern database mapping cube states to their distances from the solved state using BFS.
- PDB Storage: The pattern database (PDB) is stored as a Map from Word8Vector to Int, allowing for efficient lookups and insertions of cube states and their distances.
- File Operations: savePDB and loadPDB handle saving and loading the PDB to/from files using binary serialization for efficient storage and retrieval.

Size	Max BFS Depth	Storage Size	Number of Rows	Generation Time
2x2	7	42.1MB	1,053,180	79.1 seconds
2x2	8	152.6MB	3,814,920	286.9 seconds
3x3	6	68.9MB	983,926	129.7 seconds
3x3	7	644.4MB	9,205,558	1209.6 seconds

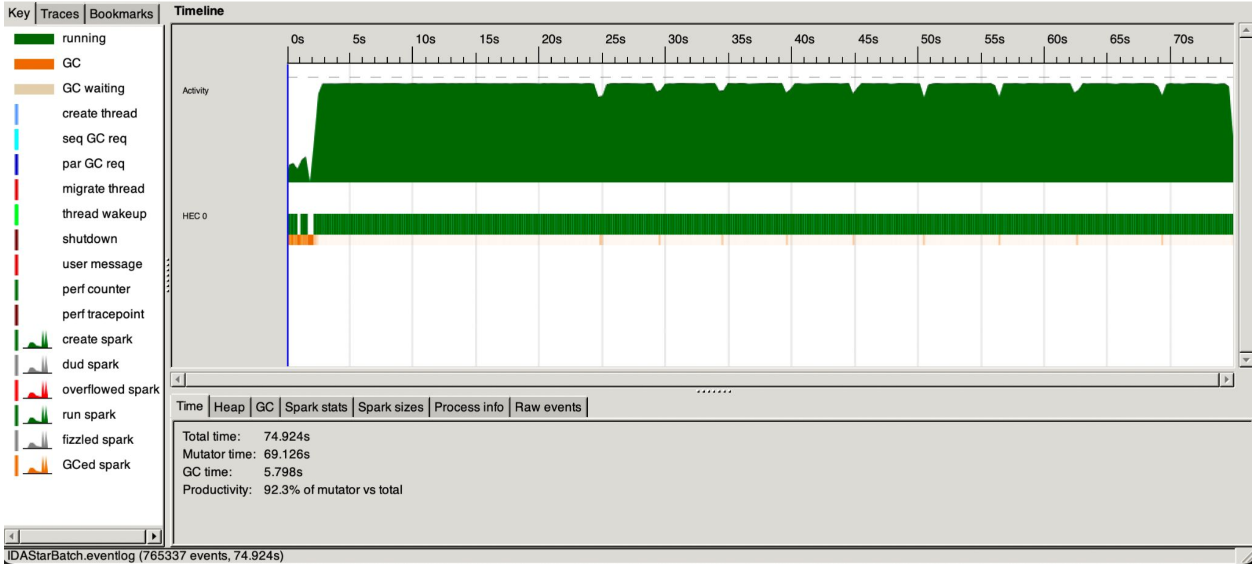
```
colorToWord8 :: Color -> Word8
word8ToColor :: Word8 -> Color
cubeToKey :: Cube -> Word8Vector

-- Bin.Binary instance: Defines how to serialize and
deserialize Word8Vector.
instance Bin.Binary Word8Vector where
    put (Word8Vector vec) = Bin.put (V.toList vec)
    get = Word8Vector . V.fromList <$> Bin.get

-- Int representing the distance to the solved state.
type PDB = Map.Map Word8Vector Int

savePDB :: FilePath -> PDB -> IO ()
savePDB file pdb = Bin.encodeFile file pdb

loadPDB :: FilePath -> IO PDB
loadPDB file = Bin.decodeFile file
```

IDA* Algorithm Using Pattern Database Results

./IDAStarBatch pdb_2x2_7.dat 2 scrambles_3000.txt +RTS -N1 -ls -RTS

`pdb_2x2_7.dat`: The pattern database for the 2x2 cube contains all the states that are within 7 moves of the solved state

`scrambles_3000.txt` An input file containing 3000 scrambled cubes, each is obtained from performing 30 random moves from the solved state.

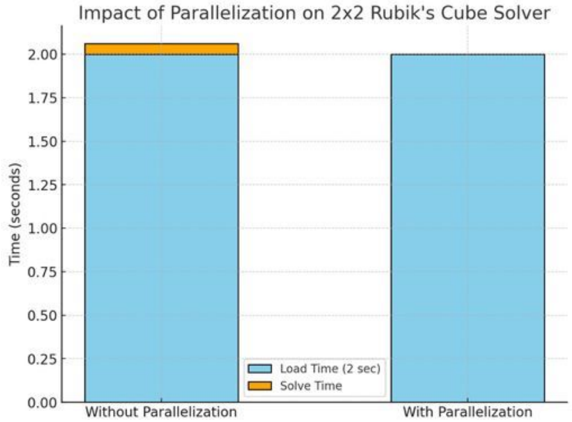


Parallelization of IDA*

We attempted to parallelize the solving process of the 2x2 Rubik's Cube using IDA*, but this did not improve efficiency. The main challenges included:

- The speed at which the 2 x 2 is solved with a linear algorithm did not leave much room for improvement
    - A huge percentage of the solution time was loading in the PDB which has to be done sequentially so as per Amdahl's law we didn't see much benefit
- Solving a 3x3 cube is challenging due to the vast state space; our PDB doesn't cover enough states. We will elaborate on this more later in the presentation.
- Maintaining a shared visited set is difficult and leads to contention from frequent read/write operations. This was necessary to avoid threads were repeatedly searching the same nodes/states.

```
-- Shared Visited States

import qualified Data.HashTable.IO as H
type VisitedStates = H.BasicHashTable Cube
Bool
initVisitedStates :: IO (MVar VisitedStates)
initVisitedStates = do
    visitedStates <- H.new :: IO
VisitedStates
    newMVar visitedStates
```

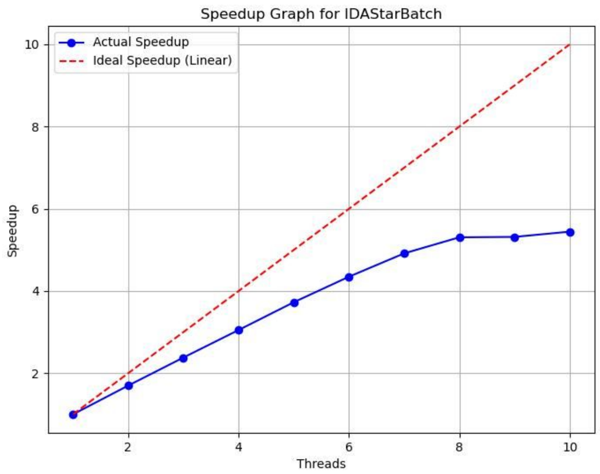


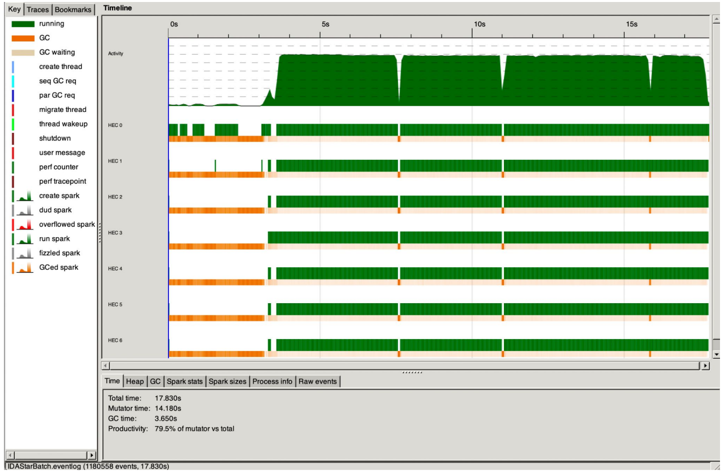
Parallelizing the solve step on a single cube offers little benefit because the 2-second PDB load time dominates the process. Even with perfect parallelization, the load time remains a sequential bottleneck.

Parallelization by Cube

- When running the IDA* algorithm to solve the 2x2 Rubik's Cube multiple times, we did observe significant speedup by assigning each cube to a different core using `Control.Concurrent.Async (forConcurrently_)`
  - This function is very similar to ParMap but has support for IO operations which made debugging much simpler
- Utilized multiple cores effectively, with each solving a separate instance.
- Diminishing returns caused by thread overhead, memory contention, and limited parallelism.
- Speedup peaked at 8 threads; performance leveled off or declined beyond that.

* Test computer has 10 logical cores





* The orange at the beginning is loading the PDB

The 3x3 case: Possible States of the cube

The (solvable) states of the Rubik's cube are determined by:

1) <u>Corner arrangement</u>
   a) ***Corner permutations*** = Spatial arrangement of the 8 corners, within their 8 available slots
   b) ***Corner orientations*** = Whether a corner is twisted correctly, CW, or CCW
2) <u>Edge arrangement</u>
   a) ***Edge permutations*** = Spatial arrangement of the 12 edges, within their 12 available slots
   b) ***Edge orientations*** = Whether an edge is flipped or not

Haskell representation:

```
data CubeState = CubeState {
    edgesPermutation   :: [Int],     -- Edge indices (0 to 11) representing their current positions
    edgesOrientation   :: [Bool],    -- Edge flips (False = correct, True = flipped)
    cornersPermutation :: [Int],     -- Corner indices (0 to 7) representing their current positions
    cornersOrientation :: [Int]      —- Corner twists (0 = correct, 1 = 120° CW, 2 = 120° CCW)
} deriving (Eq, Show)
```

The number of (solvable) states explodes with dimensions

In a 2x2 cube we can only permute and orient the corners, yielding

$8! \times 3^7 / 24 \sim 3{,}674{,}160$ (solvable) states

However, in a 3x3 cube we can also permute and orient all the edges, yielding

$8! \times 3^7 \times 12! \times 2^{11} / 2 \sim 43{,}252{,}003{,}274{,}489{,}856{,}000$

A 4x4 cube – "Rubik's Revenge" would have **$\sim 10^{45}$** solvable states…

Size of Rubik’s Cube

State Space:

- 2x2 Cube: **3.6 million states**
- 3x3 Cube: **43 quintillion states**

Even with linear IDA* and state pruning techniques, traversing such an enormous number of states is impractical.

Hence, whilst the 2x2 Cube can be “brute forced,” it is clear that the 3x3 Cube requires a much more sophisticated algorithm.

Heuristic Issues

1. **Pattern Database Limitation**: We couldn't store a full pattern database because the state space is enormous, making it impractical to generate and store all possible configurations.
2. **Ineffective Heuristics**: Simpler heuristics (e.g., number of misplaced pieces or Manhattan distance) are ineffective because they **underestimate** the number of moves required and fail to capture the complexity of the cube’s state transitions.

Avg solve time of 2 x 2 or linear solution threadscope here

(Kociemba) Two-Phase-Algorithm

1) **Solve the orientations**

In this first run of IDA*, reach **G1** state – any state that can be generated from the solved state by the orientation-preserving moves **<U,D,R2,L2,F2,B2>**. There are ~ 8! x 12! such G1 states.

Essentially, we want to reach a cube state where all the edge/corner orientations are zero (solved).

2) **Solve the permutations**

In this second run of IDA*, apply purely **G1** moves – keeping the orientations locked to the solution – to reach the overall solved state.

<u>Complex</u>: requires two separate heuristic functions, stored in several MB's of tables, for each IDA* application, and efficient manipulation of the ever-changing cube state

IDA* : A parallelization attempt

```
Initialize global search coordinator and search state
Initialize current bound with h(root)
Initialize worker threads
Enqueue initial task (starting node) into the task queue

For each worker thread: - parallel threads
    Fetch a task from the task queue
    Terminate worker thread if no tasks are left
    Process the current task:
        If solution found, notify search coordinator
        If bound exceeded, update candidate bound
        Enqueue successor tasks (nodes) to the task queue
```

IDA* : The “main loop”

```
Read results from worker threads:
    If a solution is found in search coordinator
        Terminate workers and return path
    If a new minimum bound candidate is found:
        Update new bound if exceeds current bound
    If all tasks are processed: - workers cannot proceed within bound
        Update current bound
        Kill worker threads - we need to start a DFS search anew
        Reinitialize root task for new bound
        Relaunch new worker threads
```

See the big performance inefficiency here?

IDA* : Sequential vs Parallel

```
=== Using parallel IDA* implementation ===
0.17335s
F2 L B2 R U' B' L2 D' F L2 U' R2 U L2 B2 L2 B2 D L2 U F2 U' F2
Parallel IDA* is solving the cube...

=== Using parallel IDA* implementation ===

=== Using parallel IDA* implementation ===
0.05048s
U2 L2 B2 D B' U' R F' B R' D2 F2 U L2 B2 U' B2 R2 U' F2 L2 D'
Parallel IDA* is solving the cube...

=== Using parallel IDA* implementation ===

=== Using parallel IDA* implementation ===
0.05477s
L U F' L R2 D B' U2 D L U B2 U' B2 R2 U' F2 D2 L2 U F2 R2 D' L2
Parallel IDA* is solving the cube...

=== Using parallel IDA* implementation ===

=== Using parallel IDA* implementation ===
1.54758s
L U' B R2 U D2 R D' R' F U' L2 D L2 R2 F2 D2 B2 U' R2 F2 U L2 F2 B2
Parallel IDA* is solving the cube...
```

```
=== Using sequential IDA* implementation ===
0.03704s
F2 L B2 R U' B' L2 D' F L2 U' R2 U L2 B2 L2 B2 D L2 U F2 U' F2
Parallel IDA* is solving the cube...

=== Using sequential IDA* implementation ===

=== Using sequential IDA* implementation ===
0.01452s
U2 L2 B2 D B' U' R F' B R' D2 F2 U L2 B2 U' B2 R2 U' F2 L2 D'
Parallel IDA* is solving the cube...

=== Using sequential IDA* implementation ===

=== Using sequential IDA* implementation ===
0.00682s
L U F' L R2 D B' U2 D L U B2 U' B2 R2 U' F2 D2 L2 U F2 R2 D' L2
Parallel IDA* is solving the cube...

=== Using sequential IDA* implementation ===

=== Using sequential IDA* implementation ===
0.25002s
L U' B R2 U D2 R D' R' F U' L2 D L2 R2 F2 D2 B2 U' R2 F2 U L2 F2 B2
Parallel IDA* is solving the cube...
```