# Report on Rubik's Cube Solver in Haskell

Hongcheng Tian, Roberto Brera, Matthew Rosenberg

December 19, 2024

## 1 Introduction

This report discusses the implementation of an IDA* search-based Rubik's Cube solver in Haskell, the use of pattern databases (PDB) for heuristic estimation, and subsequent attempts at parallelizing the solution process. We will start with the cube and move representations in Haskell, then detail the IDA* algorithm, our experiences applying it to different cube sizes (2x2 and 3x3), and finally discuss parallelization strategies, both for a single cube and for multiple cubes.

## 2 Cube Representation in Haskell

We represent the Rubik's Cube in a flexible yet straightforward way. Each face is a 2D array of colors, and the cube is a record of six faces: up, down, left, right, front, and back.

```haskell
type Color = Char -- 'R', 'G', 'B', 'Y', 'O', 'W'
type Face = [[Color]]

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

This structure allows easy indexing and manipulation of each cube face.

## 3 Move Representation in Haskell

We define a dedicated data type for moves. Each move corresponds to a face rotation (clockwise or counterclockwise).

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

```
applyMove :: Move -> Cube -> Cube
applyMove F  = moveF
-- similarly for other moves
```

The function `moveF` shows how a single move updates the cube's faces, demonstrating the indexing and rearrangements needed. Each move rotates one face and updates the adjacent faces accordingly.

```
-- Perform a move that rotates the front face clockwise
-- 1. front: rotate the front face
-- 2. up: update the last row of the up face with reversed values
   from the last column of the left face.
-- 3. left: update the last column of the left face with the first
   row of the down face.
-- 4. down: update the first row of the down face with reversed
   values from the first column of the right face.
-- 5. right: update the first column of the right face with the last
    row of the up face.
-- 6. back: no update required

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)
```

## 4 IDA* Algorithm Using Pattern Database

IDA* (Iterative Deepening A*) is an algorithm that combines the breadth of iterative deepening with the informed nature of A*. It repeatedly performs a depth-limited DFS, increasing the cutoff based on the next promising threshold. This allows it to find an optimal solution while still having the low memory footprint associated with DFS (as opposed to BFS).

### 4.1 IDA*

IDA* works using a heuristic $h$ that never overestimates the cost to achieve the goal. We start with a threshold equal to $h(\mathit{start})$ and do a depth-first search cutting off paths where $g + h > \mathit{threshold}$. If no solution is found, we increase the threshold.

## 4.2 Algorithm Pseudocode

```
function IDA*(start, goal, h)
    threshold := h(start)
    loop
        temp := search(start, 0, threshold)
        if temp = FOUND then
            return FOUND
        if temp = inf then
            return NOT_FOUND
        threshold := temp

function search(node, g, threshold)
    f := g + h(node)
    if f > threshold then
        return f
    if node = goal then
        return FOUND
    min := inf
    for each successor of node
        temp := search(successor, g + cost(node, successor), threshold)
        if temp = FOUND then
            return FOUND
        if temp < min then
            min := temp
    return min
```

## 4.3 Pattern Database and our implementation

A PDB stores precomputed distances for a subset of states. For smaller puzzles (like 2x2 cube), we can generate a pattern database that gives accurate or nearly accurate estimates of the distance to the solved state. This helps IDA* prune the search space drastically, since states that appear in the PDB give a good heuristic.

The `PDB.hs` module is responsible for generating and managing the pattern database (PDB) used by our Rubik's Cube solver. The PDB serves as a lookup table that maps specific cube states to their minimal number of moves to reach the solved state. By consulting this database during search, we can derive a heuristic value to guide the IDA* algorithm.

### 4.3.1 State Representation and Key Conversion

In order to store and retrieve cube states efficiently, we represent each state as a `Word8Vector`. A `Word8Vector` is essentially a vector of `Word8` values, where each `Word8` encodes a single sticker's color. This encoding step transforms a complex 3D Rubik's Cube configuration into a compact, hashable key suitable as a map key:

```haskell
import qualified Data.Vector.Unboxed as V

newtype Word8Vector = Word8Vector (V.Vector Word8)
    deriving (Eq, Show, Ord)
```

```
cubeToKey :: Cube -> Word8Vector
cubeToKey cube =
    Word8Vector $ V.fromList $ concatMap (map colorToWord8) [
        concat (up cube),
        concat (down cube),
        concat (left cube),
        concat (right cube),
        concat (front cube),
        concat (back cube)
    ]
```

This linearization of the cube's faces into a vector ensures a consistent and unique representation of any cube state.

### 4.3.2 Serialization and Persistence

We use the `Data.Binary` library for serialization. Defining a `Binary` instance for `Word8Vector` allows us to easily save and load the entire PDB to and from a file.

```
-- Define the Binary instance for the newtype
instance Bin.Binary Word8Vector where
    put (Word8Vector vec) = Bin.put (V.toList vec)
    get = Word8Vector . V.fromList <$> Bin.get
```

We rely on:

- **`Bin.encodeFile`**: Writes the entire PDB map (which is a `Map Word8Vector Int`) to a file, serializing both keys and values.
- **`Bin.decodeFile`**: Reads the PDB back into memory, reconstructing the map.

This mechanism makes it possible to build the PDB once and then quickly load it in future runs, saving computation time.

### 4.3.3 Building the PDB Using BFS

The PDB is constructed by performing a breadth-first search (BFS) starting from the solved cube state. BFS explores states level by level, ensuring that when we first reach a particular cube state, we've found the minimal number of moves required to get there. This property directly ensures that the distances recorded in the PDB are minimal and accurate.

The BFS loop:

- Starts from the solved state at depth 0.
- Expands successors by applying all moves and recording their depths.
- Uses a visited set to avoid revisiting states.
- Stops when a specified depth limit is reached or when no more states are available.

Because BFS discovers states in order of their increasing distance, the first time we encounter a state, that depth is its minimal depth.

### 4.4 Statistical Overview of Our PDB

| 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 |

Table 1: Statistical Overview of Our PDB

## 5 Linear IDA* + PDB on Different Cube Sizes

### 5.1 2x2 Cube

On a 2x2 Rubik's Cube, the state space is much smaller. With a decent PDB, IDA* can solve fully scrambled states extremely fast (e.g., around 0.05s on average). The PDB covers a significant portion of reachable states, making the heuristic effective early in the search.

### 5.2 3x3 Cube

For the 3x3 cube, the approach is much less successful. The state space is enormous, and our partial PDB rarely helps during the initial layers of the search. Consequently, without frequent heuristic guidance, the algorithm degenerates into a near-brute-force search.

While 2x2 can be quickly solved, the 3x3 scenario demonstrates the limitation of IDA* + PDB: it's simply not scalable to the complexity of a standard Rubik's Cube. More advanced algorithms (like the two-phase algorithm) are known to be more efficient but were not implemented due to complexity and time constraints. There is more information on this later in the report.

## 6 Parallelization with a Single Cube

### 6.1 Our attempt

Parallelizing the search on a single cube by splitting the search tree at the root and assigning each subtree to a thread seemed promising initially but turned out to offer little performance improvement for the following reasons:

1. Different threads may repeat work on overlapping states. For example, Thread 1 goes L $\rightarrow$ R, and Thread 2 goes R $\rightarrow$ L. They both reach the same state and continue performing redundant searches from that point.

2. A shared visited structure is challenging to implement and may cause contention. Ensuring thread-safe access to a common data structure for visited states can introduce significant overhead, negating the benefits of parallelization.

3. Without effective pruning or shared information, parallelization doesn't yield significant speedups. Efficient parallel algorithms often rely on mechanisms to share partial results and prune the search space, but implementing these mechanisms correctly and efficiently is non-trivial.

4. Load balancing is another issue. Some subtrees may be significantly larger and more complex than others, leading to uneven distribution of work among threads. This imbalance can result in some threads being idle while others are overwhelmed, thus reducing overall efficiency.

In our attempts to parallelize IDA* for a Rubik's Cube, we used a structure similar to the following to manage visited states:

```
type VisitedStates = H.BasicHashTable Cube Bool

initVisitedStates :: IO (MVar VisitedStates)
initVisitedStates = do
    visitedStates <- H.new :: IO VisitedStates
    newMVar visitedStates
```

This structure, utilizing a hashtable from 'Data.HashTable.IO', aimed to coordinate access to the set of visited states across multiple threads. However, the need for thread-safe operations introduced significant contention, further complicating the parallelization effort.

In short, naive parallelization of a single cube's search is not straightforward. Effective parallelization requires careful management of state sharing, load balancing, and memory usage, along with strategies to minimize redundant work and contention.

In short, naive parallelization of a single cube's search is not straightforward.

## 6.2 Other Parallelization Strategies Not Implemented

We identified advanced parallelization strategies from similar projects, but did not implement them in this section due to complexity and time constraints. These include:

- **Work Distribution**: Dividing the search space at upper levels into tasks and distributing them among worker threads, potentially using a work-stealing queue and parameters like `maxParDepth` to control granularity. This ensures that each worker thread has a manageable portion of the search space, potentially improving load balancing and reducing idle times.

- **Worker Pool**: Maintaining a fixed pool of worker threads:

  1. Each worker pulls tasks from a shared queue.
  2. Explores its assigned subtree.
  3. Reports results back through a result channel.
  4. Can steal work from other workers when idle.

  By using a worker pool, the system can dynamically balance the load, ensuring that all threads are utilized efficiently.

- **Bound Management**: Sharing a global bound among workers and propagating improved bounds to prune unpromising branches early. By sharing information about the bounds, workers can avoid exploring paths that are unlikely to lead to a solution, thus improving the efficiency of the parallel search.

These strategies aim to address the shortcomings of naive parallelization by focusing on efficient task distribution, dynamic load balancing, and effective pruning mechanisms. While more sophisticated and scalable, they require careful management of granularity control, memory usage, synchronization, and advanced heuristics, making their implementation non-trivial.

These more advanced parallelization techniques represent more sophisticated and scalable solutions than our initial attempts, and are implemented in an attempt to parallelize IDA* in the following sections.

## 7 Parallelization with Multiple Cubes

A simpler scenario is solving multiple cubes in parallel. Each cube is independent, and we can just run multiple cube-solving tasks simultaneously. This approach doesn't require modifying linear IDA*, since each thread just solves a cube from start to finish.

```haskell
import Control.Concurrent.Async (forConcurrently_)

main = do
    -- ...
    -- linesOfMoves is a list of scrambles, one per line
    forConcurrently_ (zip [1..] linesOfMoves) $ \(idx, line) -> do
        -- Solve each cube independently in a separate thread
        let scrambleMoves = parseLineOfMoves line
        let scrambledCube = applyMoves scrambleMoves solvedCube
        maybeSolution <- idaStar scrambledCube (heuristic pdb)
           allMoves
        -- handle result
```

By using `forConcurrently_`, each cube is processed by its own thread. Running with `+RTS -N4` enables multiple CPU cores, allowing true parallelism. This approach typically shows parallel speedup if the tasks are sufficiently CPU-bound.

`forConcurrently_` operates similarly to `parMap_`, but is safer with IO, which simplifies debugging. Using this method, we observed substantial speedup as we increased the number of threads, capping at around a 6-fold improvement with 8 threads on a 10-core machine.

See Figure 1 for the speedup graph and Figures 2-5 for core usage per run. All the mentioned graphs are based on a set of 10,000 cubes. As expected, the initial phase of the usage runs primarily on one core while the program reads the serialized PDB and processes it into a map. From then on, parallelization works effectively but eventually sees diminishing returns.

We were surprised at the speed with which the added threads stopped improving efficiency. Undoubtedly, there is a loss due to thread overhead and context switching, but capping our improvement at 6x was unexpected. Scaling up the number of cubes that are solved is likely to show further improvement, since the PDB portion of the process is a constant-time operation. However, we tested only up to 10,000 cubes. Up to that point, our hypothesis was true. You can see figure 6 for the extreme example of the impact of loading in the PDB while solving a single cube.

## 8 Parallelization of IDA* on a 3x3 cube

### 8.1 Rubik's Cube States

The solvable states of the Rubik's cube are determined by several factors related to the arrangement and orientation of its components:

- **Corner Arrangement**
  - **Corner Permutations**: This refers to the spatial arrangement of the eight corner pieces within their eight available slots.
  - **Corner Orientations**: This indicates whether a corner is twisted correctly, clockwise (CW), or counterclockwise (CCW).
- **Edge Arrangement**
  - **Edge Permutations**: This pertains to the spatial arrangement of the twelve edge pieces within their twelve available slots.
  - **Edge Orientations**: This denotes whether an edge piece is flipped or not.

In Haskell, this can be represented as follows:

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

Using this framework, it is easy to calculate all the possible states of a cube, and thus develop more accurate heuristics.

In a $2 \times 2$ cube, we can only permute and orient the corners, yielding:

$$\frac{8! \times 3^7}{24} \approx 3{,}674{,}160 \quad \text{states}$$

However, in a $3 \times 3$ cube, we can also permute and orient all the edges, yielding:

$$\frac{8! \times 3^7 \times 12! \times 2^{11}}{2} \approx 4.3252 \times 10^{19} \quad \text{states}$$

Ironically, a $4 \times 4$ cube – "Rubik's Revenge" – would have approximately $10^{45}$ solvable states.

### 8.2 Kociemba's Two-Phase-Algorithm

The complexity of the Rubik's Cube has thus necessitated the development of efficient algorithms to navigate this vast state space. One such approach is the two-phase algorithm, which leverages two iterations of the Iterative Deepening A* (IDA*) search strategy to systematically solve the cube. This method decomposes the solution process into two distinct phases, each targeting specific aspects of the cube's state.

**Phase 1: Solving the Orientations** In the first phase, the algorithm focuses on solving the orientations of the cube's pieces. This involves reaching a **G1 state**, defined as any state that can be generated from the solved state through orientation-preserving moves, specifically the set $\{U, D, R^2, L^2, F^2, B^2\}$. The objective is to orient all edge and corner pieces correctly, effectively reducing their orientation degrees of freedom to zero.

Mathematically, the number of possible G1 states is approximately $8! \times 12!$, accounting for the permutations of the 8 corners and 12 edges while maintaining their correct orientations. Achieving this state simplifies the subsequent phase by ensuring that all pieces are correctly oriented, thereby allowing the algorithm to concentrate solely on their permutations without altering their orientations.

**Phase 2: Solving the Permutations** Once the cube is in a G1 state, the second phase commences with the goal of solving the permutations of the pieces to reach the fully solved state. This phase involves applying only the G1 moves, which preserve the orientations established in Phase 1. By restricting the move set to orientation-preserving operations, the algorithm ensures that the orientations remain locked, thereby focusing exclusively on the permutation of pieces.

## 8.3 Implementation of a parallelized Two-Phase solver

Whilst in the previous section we have focused on parallelizing different batches of cubes, thus keeping the actual solving algorithm at an atomic level, in this section we propose a parallelization of the IDA* algorithm which lies at the core of the solving procedure. In order to do so, we use an existing implementation of a cube solver (see "twentyseven" in the references), but modify the central IDA.hs module. Here we emphasize some key design choices. The full implementation of the algorithm is given in the Appendix.

### 8.3.1 Shared Structures Amongst the Threads

Shared data structures are essential for coordination and communication between multiple threads. The most important shared structures include the `SearchCoordinator` and `SearchState`. These manage the state of the search, track active tasks, and handle synchronization to ensure thread-safe operations.

```
data SearchCoordinator a l node = SearchCoordinator {
    taskCount :: IORef Int,
    solutionFound :: IORef Bool,
    currentBound :: IORef a,
    activeSearches :: IORef Int
}

data SearchState a l node = SearchState {
    coordinator :: SearchCoordinator a l node,
    resultChan :: !(Chan (SearchResult a l)),
    taskQueue :: !(Chan (Maybe (SearchTask a l node))),
    activeWorkers :: !(MVar Int),
    stateId :: !Int,
    config :: !ParConfig
}
```

### 8.3.2 Worker Threads

Each worker continuously retrieves tasks, or nodes, from the shared `taskQueue`, processes them by exploring possible successors, and communicates results back through the resultChan. The workers monitor the solutionFound flag to terminate gracefully once a solution is discovered or when there are no more tasks to process.

```
worker :: (NFData node, NFData a, NFData l, Ord a, Num a, Show a)
       => SearchState a l node
       -> Search [] a l node
       -> IO ()
worker state search = do
    modifyMVar_ (activeWorkers state) (\n -> return (n + 1))
    let loop = do
            foundSolution <- readIORef (solutionFound $ coordinator state)
            tasksPending <- readIORef (taskCount $ coordinator state)
            if foundSolution || (tasksPending == 0)
                then shutdown
                else do
                    mtask <- readChan (taskQueue state)
                    case mtask of
                        Nothing -> shutdown
                        Just task -> do
                            processTask state search task
                            loop

        shutdown = do
            modifyMVar_ (activeWorkers state) $ \n -> do
                let newCount = n - 1
                when (newCount == 0) $
                    writeChan (resultChan state) Stop
                return newCount
    loop
```

### 8.3.3 The Main Loop

The main loop handles the overall search process by processing incoming results from worker threads and managing the iterative deepening bounds. Upon receiving a Found result, it signals all workers to terminate and returns the solution path. If a NextBound is received, the loop updates the search bounds and reinvokes the workers to continue the search within the new limits. This loop ensures that the algorithm incrementally explores deeper levels of the search tree, dynamically adjusting based on the progress and findings of the worker threads.

```
let mainLoop prevBound = do
    result <- readChan (resultChan state)
    case result of
        Found path -> do
            -- Solution found; stop everything
            atomicWriteIORef (solutionFound coord) True
            replicateM_ numWorkers $ writeChan (taskQueue state) Nothing
            return (Just path)

        -- process incoming candidate new bound
        Next newBC@(NextBound nb) -> do
            when (nb > prevBound) $ do
                atomicModifyIORef' boundState $ \curr -> (curr <> newBC, ())
            active <- readMVar (activeWorkers state)
            tasksPending <- readIORef (taskCount coord)
            -- Workers can't reach any tasks beyond bound
            if tasksPending == 0
                then do
                    NextBound candidate <- readIORef boundState
                    if candidate /= maxBound && candidate > prevBound
                        then do
                            let nextBound = candidate

                            -- Update global state for new iteration
                            atomicWriteIORef (currentBound coord) nextBound
                            atomicWriteIORef boundState (NextBound maxBound)
                            atomicWriteIORef (solutionFound coord) False
                            -- 1) clean up old workers
                            activeCount <- readMVar (activeWorkers state)
                            replicateM_ activeCount $ writeChan (taskQueue state) Nothing

                            -- 2) Initialize tasks again
                            atomicModifyIORef' (taskCount coord) (\n -> (n + 1, ()))
                            writeChan (taskQueue state) $ Just SearchTask {
                                taskNode = root,
                                taskDepth = 0,
                                taskBound = nextBound,
                                taskPath = [],
                                taskG = 0
                            }

                            -- 3) Relaunch new workers
                            forM_ [1..numWorkers] $ \i -> do
                                forkIO $ worker state s

                            mainLoop nextBound
                        else do
                            return Nothing
                else do
                    -- There are still tasks pending, accumulate candidate and continue
                    mainLoop prevBound

        Stop -> do
            mainLoop prevBound
```

## 9 Conclusion

In this report, we have detailed the implementation and optimization of an IDA* search-based Rubik's Cube solver in Haskell, utilizing pattern databases for heuristic estimation and examining various parallelization strategies. Our approach to representing the cube and its moves in Haskell provides a flexible and efficient framework for state manipulation. Through the development of a pattern database, we were able to make a reliable heuristic, which guaranteed us an optimal solution when working on smaller cubes such as the 2x2. However, our attempts to directly apply the same methodology to the 3x3 cube highlighted the limitations of this approach, as the state space increased exponentially.

Parallelization efforts yielded mixed results. While solving multiple cubes in parallel proved effective, achieving near-linear speedup up to a point, parallelizing the search process for a single cube presented considerable challenges. Our initial naive parallelization attempts demonstrated limited success due to redundant work and synchronization overhead. The core difficulty in parallelizing IDA* lies in managing redundant state explorations across multiple threads. As each thread independently traverses different branches of the search tree, the probability of overlapping efforts—where multiple threads explore identical or similar states—increases dramatically. This redundancy essentially undermines the potential performance gains from parallel execution. Our attempts to employ a shared visited set using thread-safe data structures introduced substantial synchronization overhead, which further diminished the benefits of parallelization.

Looking forward, one promising direction to mitigate redundancy is the implementation of a shared visited set with a strategic update interval. By allowing threads to periodically synchronize their visited states rather than maintaining constant access, it may be possible to strike a balance between reducing redundant searches and minimizing synchronization costs. This approach could enable more efficient sharing of explored states, thereby enhancing the overall scalability and performance of the parallelized IDA* algorithm.

## 10 References

- https://github.com/hkociemba/RubiksCube-TwophaseSolver
- https://github.com/Lysxia/twentyseven/tree/master/src/Rubik
- https://github.com/Lysxia/twentyseven/blob/bfcd3ddcf938bc6db933a364d331877b0fd1257e/src/Rubik/Solver.hs#L6
- https://github.com/Lysxia/twentyseven/blob/bfcd3ddcf938bc6db933a364d331877b0fd1257e/src/Rubik/IDA.hs#L2
- https://hackage.haskell.org/package/twentyseven-0.0.0/docs/Rubik-Solver.html
- https://kociemba.org/cube.htm
- https://www.cubelelo.com/blogs/cubing/understanding-the-rubiks-cube-permutations?srsltid=AfmBOooDB7hySQi8CvaMu5EVUuTNfZRp1tsUiIqx7t_eZ64edpWKUVoR

## 11 Figures

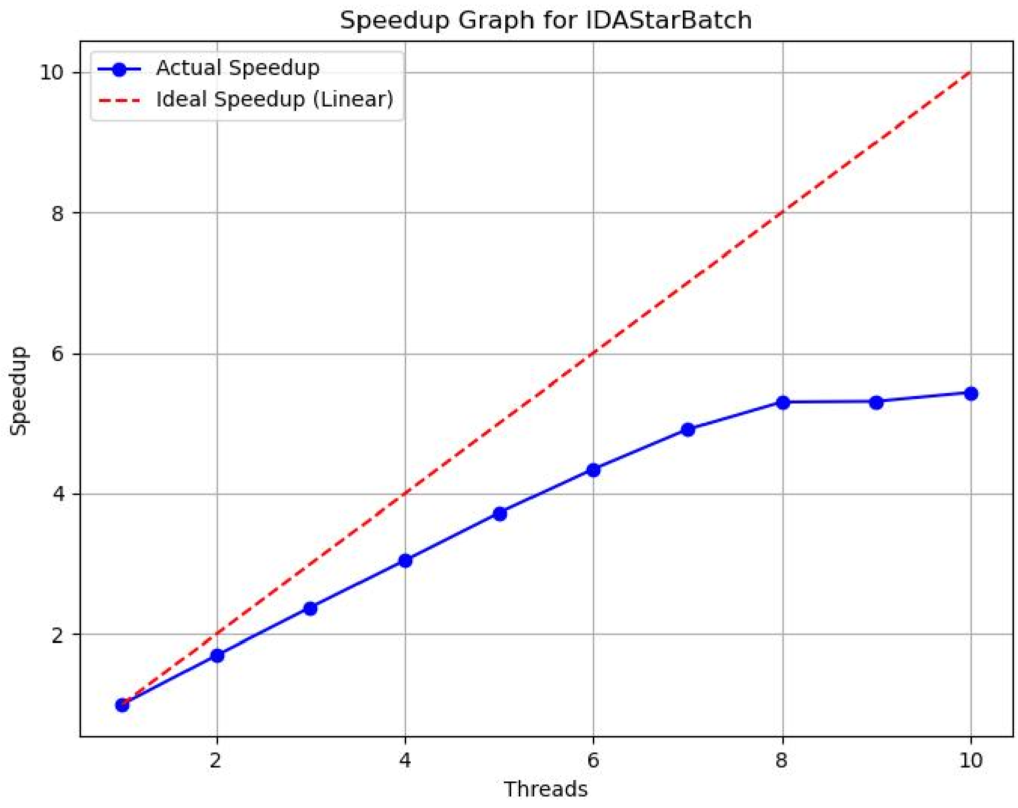
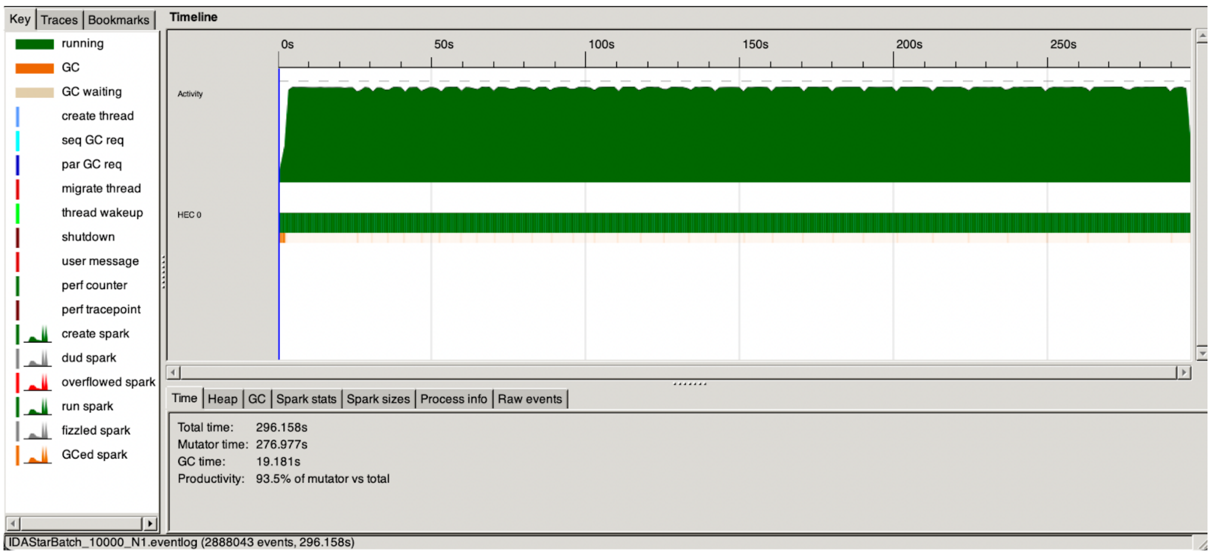


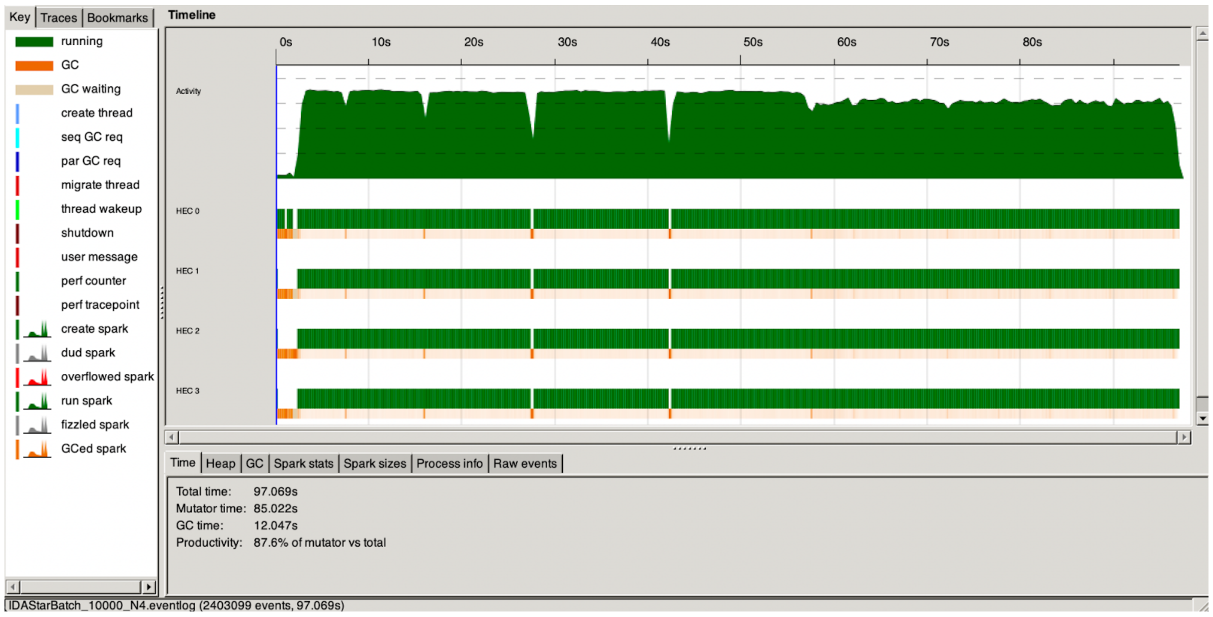
Figure 1: Speed up of algorithm when adding cores



```
Total time:     296.158s
Mutator time:   276.977s
GC time:        19.181s
Productivity:   93.5% of mutator vs total
```

IDAStarBatch_10000_N1.eventlog (2888043 events, 296.158s)

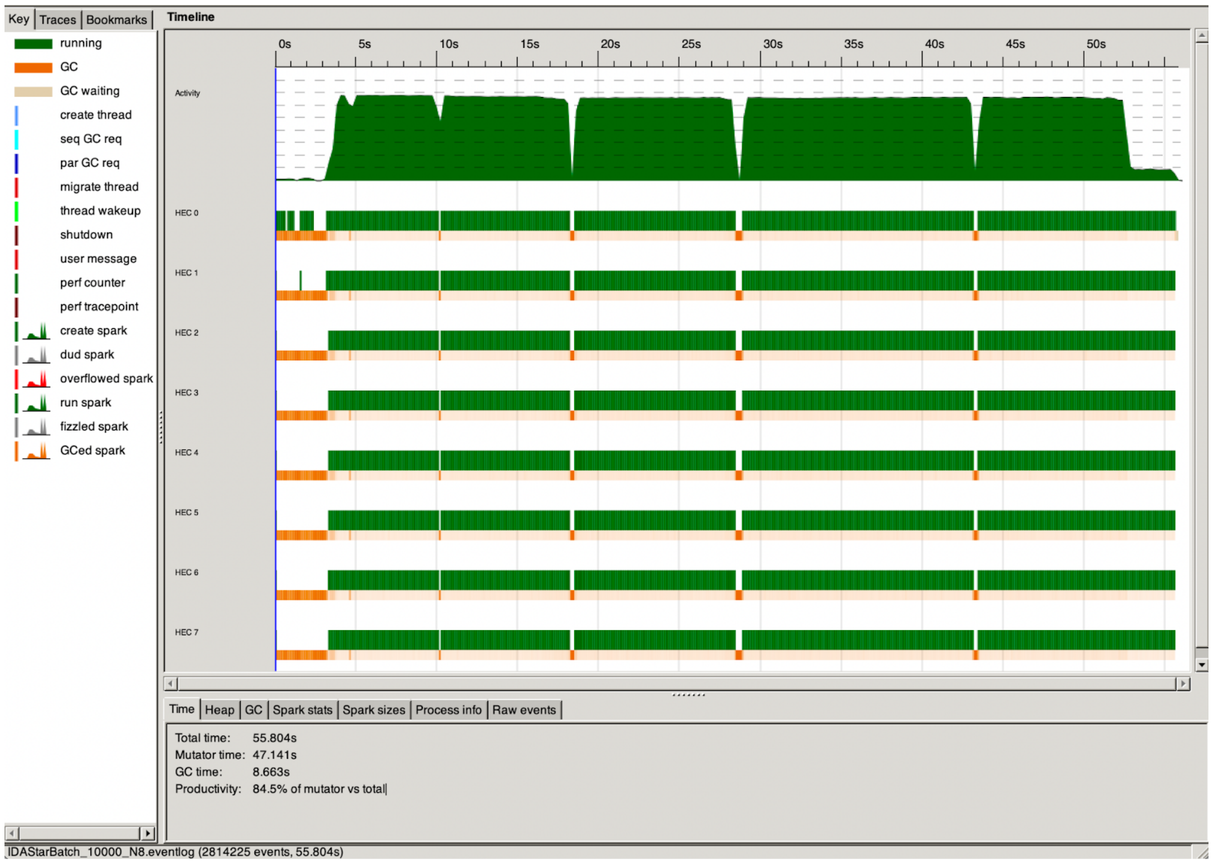
Figure 2: Core usage with 1 thread



```
Total time:     97.069s
Mutator time:   85.022s
GC time:        12.047s
Productivity:   87.6% of mutator vs total
```

IDAStarBatch_10000_N4.eventlog (2403099 events, 97.069s)

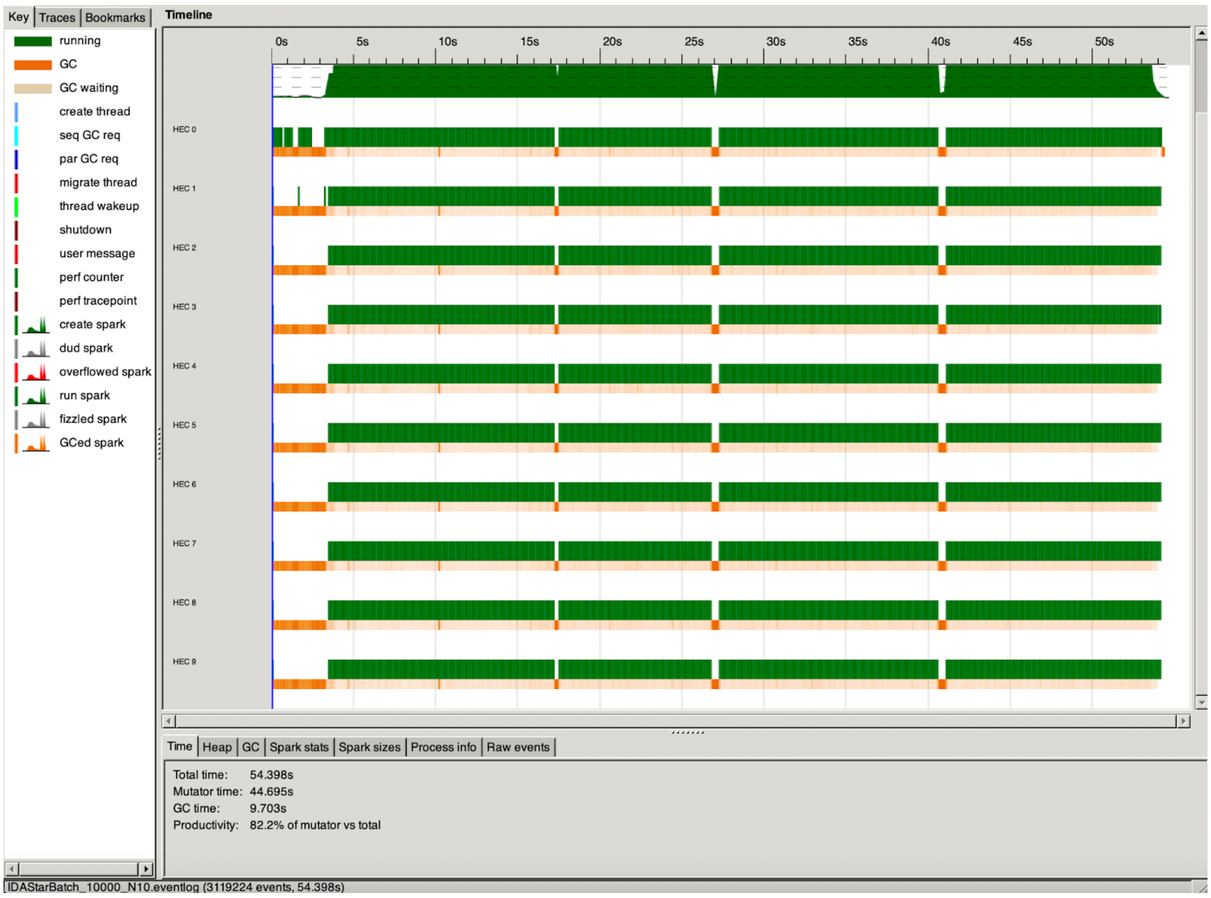
Figure 3: Core usage with 4 threads



```
Total time:     55.804s
Mutator time:   47.141s
GC time:        8.663s
Productivity:   84.5% of mutator vs total
```

IDAStarBatch_10000_N8.eventlog (2814225 events, 55.804s)

Figure 4: Core usage with 8 threads



```
Total time:     54.398s
Mutator time:   44.695s
GC time:        9.703s
Productivity:   82.2% of mutator vs total
```

```
IDAStarBatch_10000_N10.eventlog (3119224 events, 54.398s)
```

Figure 5: Core usage with 10 threads

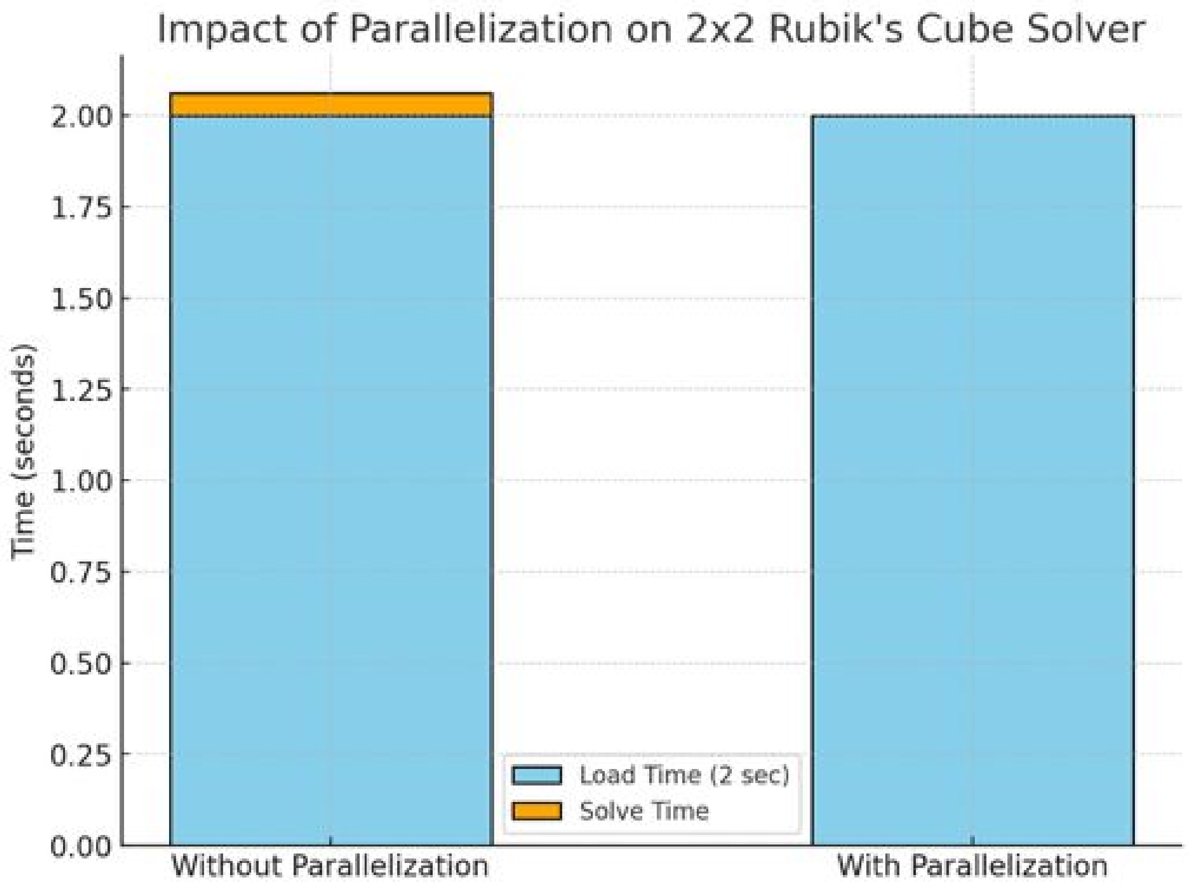


Figure 6: Impact of loading in PDB

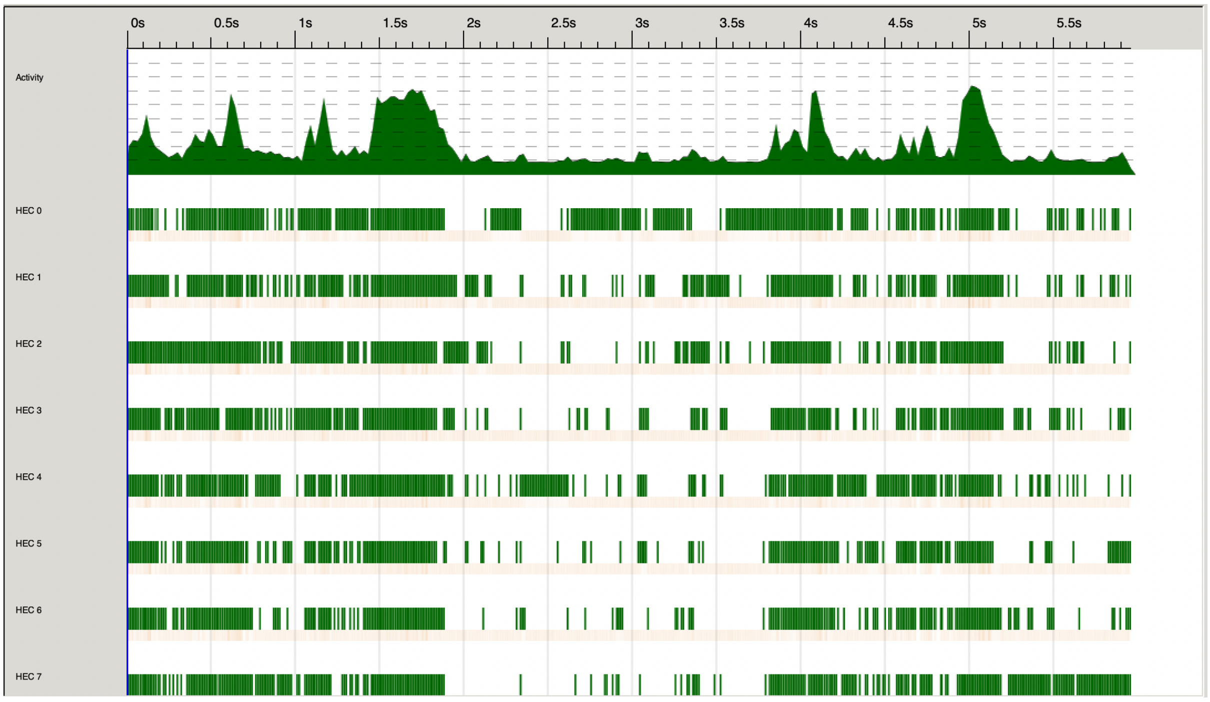


Figure 7: IDA* parallel: Core usage with 8 cores

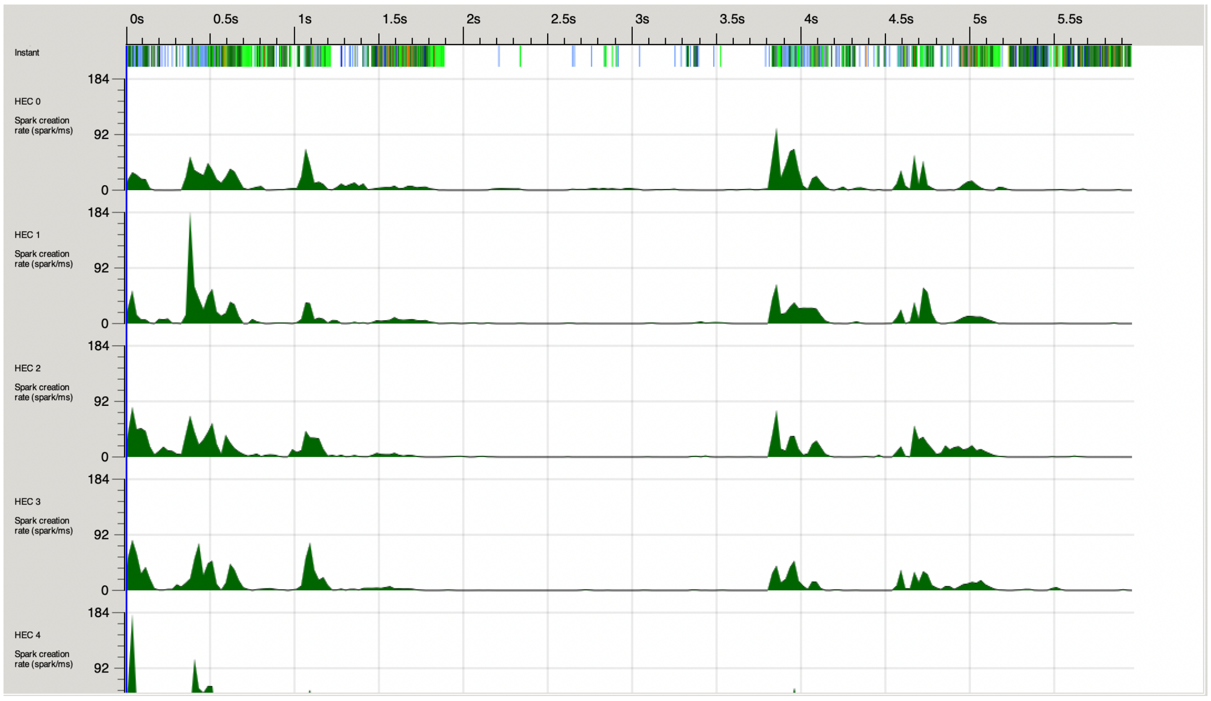


Figure 8: IDA* parallel: Spark creation with 8 cores

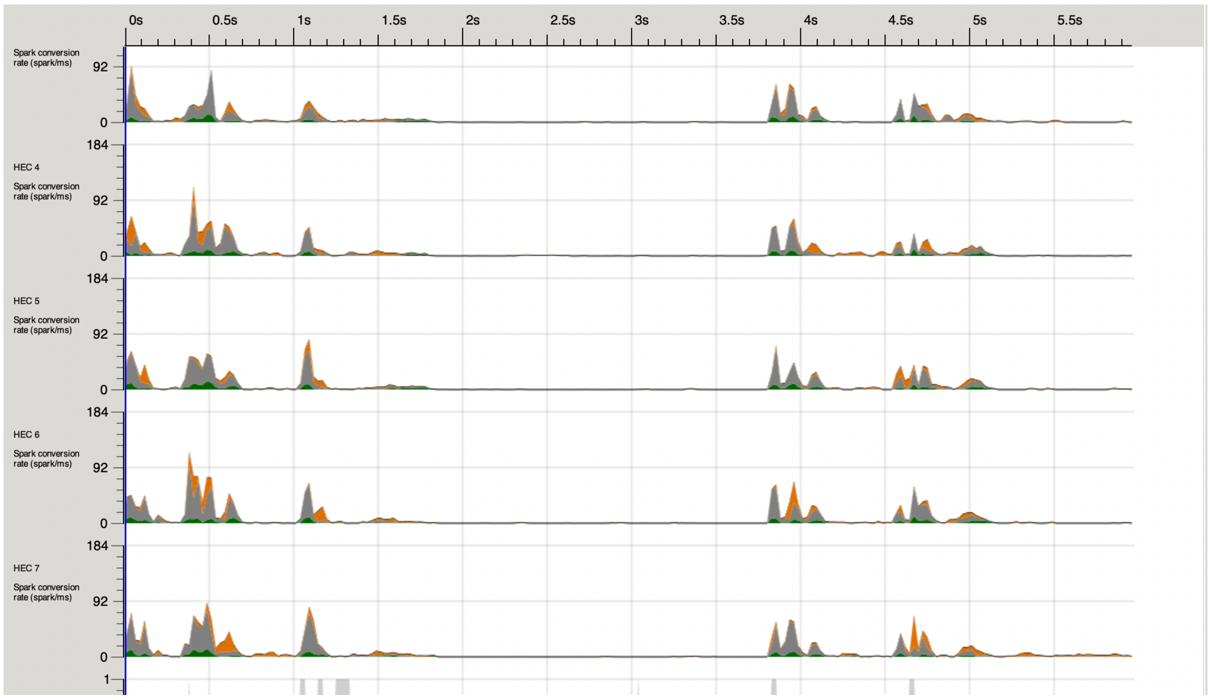


Figure 9: IDA* parallel: Spark conversion with 8 cores

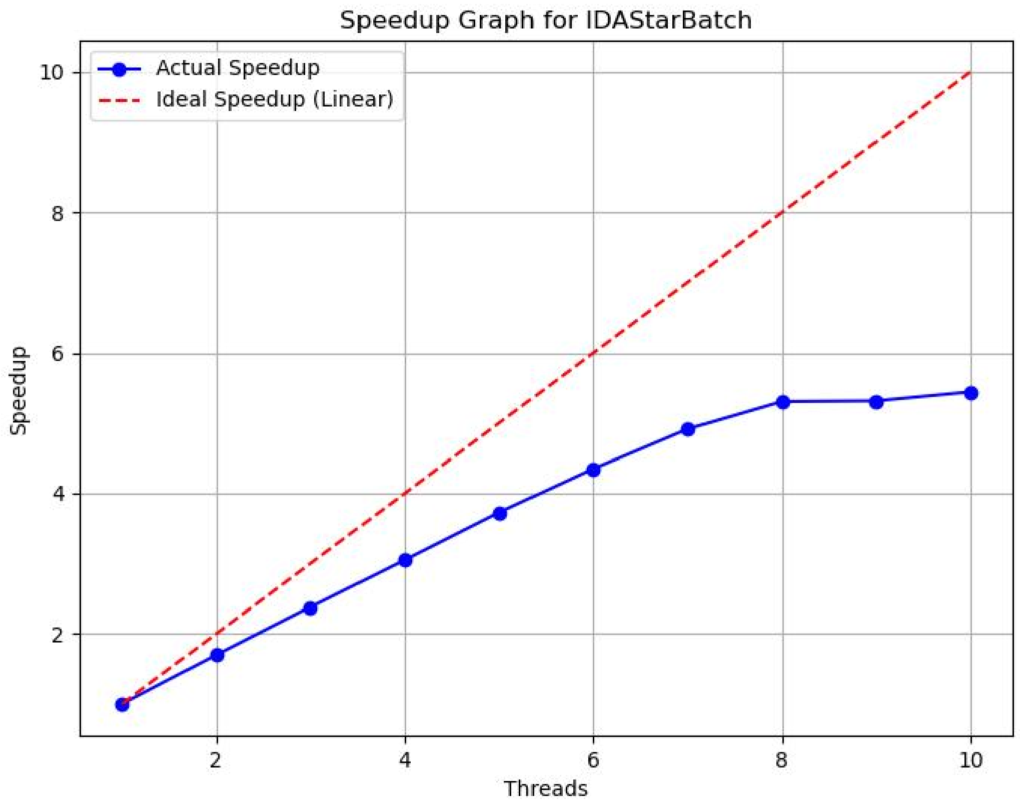


Figure 10: IDA* parallel: Speed-up graph for 1-8 cores

# 12 Appendix

## 12.1 IDA.hs

Note: This code needs to run within an updated version of the ”twentyseven” repository

```haskell
{-# LANGUAGE ScopedTypeVariables, MultiParamTypeClasses,
             FunctionalDependencies, FlexibleInstances,
             BangPatterns, RecordWildCards,
             RankNTypes, UndecidableInstances,
             TypeApplications, DeriveGeneric,
             StandaloneDeriving #-}

module Rubik.IDA (
    Search(..),
    Succ(..),
    SearchResult(..),
    Result,
    search,
    selfAvoid,
    selfAvoidRoot,
    ParConfig(..),
    DebugLevel(..),
    defaultParConfig,
    withBasicLogging,
    withVerboseLogging,
    withFullDebug
) where


import qualified Data.Set as S
import Control.Parallel.Strategies
import Control.DeepSeq
import Control.Concurrent
import Control.Monad (forM_, when, replicateM_)
import System.IO.Unsafe (unsafePerformIO)
import System.Random (randomRIO)
import Data.IORef
-- import Debug.Trace (trace)
import GHC.Generics
import System.IO (hFlush, stdout)


------------------------------------------------------------
-- Core Data Structures
------------------------------------------------------------

data Succ label length node = Succ {
    eLabel :: label,
    eCost :: length,
    eSucc :: node
} deriving (Show, Eq)

data Search f a l node = Search {
    goal :: node -> Bool,
    estm :: node -> a,
    edges :: node -> f (Succ l a node)
}

type Result a l = Maybe [l]

-- Represents the next bound candidate discovered during search
data BoundCandidate a = NextBound !a
    deriving (Show, Eq)

instance NFData a => NFData (BoundCandidate a) where
    rnf (NextBound x) = rnf x


instance (Ord a) => Semigroup (BoundCandidate a) where
    (NextBound a) <> (NextBound b) = NextBound (min a b)
```

```
data SearchResult a l =
    Next !(BoundCandidate a)
  | Found [l]
  | Stop

deriving instance Generic (SearchResult a l)

instance (NFData a, NFData l) => NFData (SearchResult a l) where
    rnf (Next a) = rnf a
    rnf (Found ls) = rnf ls
    rnf Stop = ()

instance Ord a => Semigroup (SearchResult a l) where
    f@(Found _) <> _ = f
    _ <> f@(Found _) = f
    Next a <> Next b = Next (a <> b)
    Stop <> x = x
    x <> Stop = x

instance Ord a => Monoid (SearchResult a l) where
    mempty = Stop


------------------------------------------------------------
-- Parallel Configuration and Logging
------------------------------------------------------------

data DebugLevel =
    Silent
  | Basic
  | Verbose
  | VeryVerbose
    deriving (Show, Eq, Ord)

data ParConfig = ParConfig {
    maxParDepth :: !Int,
    numWorkers :: !Int,
    chunkSize :: !Int,
    boundUpdateThreshold :: !Int,
    debugLevel :: !DebugLevel
} deriving Show

defaultParConfig :: ParConfig
defaultParConfig = ParConfig {
    maxParDepth = 5,
    numWorkers = 4,
    chunkSize = 100,
    boundUpdateThreshold = 1000,
    debugLevel = Silent
}

withBasicLogging :: ParConfig -> ParConfig
withBasicLogging config = config { debugLevel = Basic }

withVerboseLogging :: ParConfig -> ParConfig
withVerboseLogging config = config { debugLevel = Verbose }

withFullDebug :: ParConfig -> ParConfig
withFullDebug config = config { debugLevel = VeryVerbose }

logDebug :: DebugLevel -> DebugLevel -> String -> IO ()
logDebug configLevel messageLevel msg = do
    when (configLevel >= messageLevel) $ do
        putStrLn $ "[ParIDA*] " ++ msg
        hFlush stdout


------------------------------------------------------------
-- Search Coordinator
------------------------------------------------------------

data SearchCoordinator a l node = SearchCoordinator {
    taskCount :: IORef Int,
    solutionFound :: IORef Bool,
    currentBound :: IORef a,
    activeSearches :: IORef Int
}

initSearchCoordinator :: (Num a) => a -> IO (SearchCoordinator a l node)
initSearchCoordinator initBound = do
    tasks <- newIORef 0
    solution <- newIORef False
    bound <- newIORef initBound
    searches <- newIORef 0
    return SearchCoordinator {
        taskCount = tasks,
        solutionFound = solution,
        currentBound = bound,
        activeSearches = searches
    }


------------------------------------------------------------
-- Search State and Task
------------------------------------------------------------

data SearchTask a l node = SearchTask {
    taskNode :: !node,
    taskDepth :: !Int,
    taskBound :: !a,
    taskPath :: ![l],
    taskG :: !a
}

data SearchState a l node = SearchState {
    coordinator :: SearchCoordinator a l node,
    resultChan :: !(Chan (SearchResult a l)),
    taskQueue :: !(Chan (Maybe (SearchTask a l node))),
    activeWorkers :: !(MVar Int),
    stateId :: !Int,
    config :: !ParConfig
}

initSearchState :: (Show a, Num a) =>
                  SearchCoordinator a l node ->
                  ParConfig ->
                  IO (SearchState a l node)
initSearchState coord config = do
    results <- newChan
    tasks <- newChan
    workers <- newMVar 0
    sid <- randomRIO (1, 1000 :: Int)
    return SearchState {
        coordinator = coord,
        resultChan = results,
        taskQueue = tasks,
        activeWorkers = workers,
        stateId = sid,
        config = config
    }


------------------------------------------------------------
-- Core IDA* Exploration Logic
------------------------------------------------------------

exploreNode :: forall node a l.
            (NFData node, NFData a, NFData l, Ord a, Num a, Show a)
            => Search [] a l node
            -> ParConfig
            -> Int
            -> node
            -> a
            -> [l]
            -> a
            -> [SearchResult a l]
exploreNode Search{..} config@ParConfig{..} depth n g ls bound
    | g == bound && g == f && goal n = [Found (reverse ls)]
    | f > bound = [Next (NextBound f)]
    | depth >= maxParDepth = [mconcat $ map processSucc (edges n)]
    | otherwise =
        let chunks = splitIntoChunks chunkSize (edges n)
        in map mconcat $ withStrategy (parTraversable rdeepseq) $
           map (map processSucc) chunks
  where
    !f = g + estm n
    processSucc (Succ l c s) =
        mconcat $ exploreNode (Search goal estm edges) config (depth + 1) s (g + c) (l:ls)
            bound

splitIntoChunks :: Int -> [a] -> [[a]]
splitIntoChunks n = go
  where
    go [] = []
    go xs = let (chunk, rest) = splitAt n xs
            in chunk : go rest


------------------------------------------------------------
-- Processing Tasks
------------------------------------------------------------

processTask :: forall node a l.
            (NFData node, NFData a, NFData l, Ord a, Num a, Show a)
            => SearchState a l node
            -> Search [] a l node
            -> SearchTask a l node
            -> IO ()
processTask state search task@SearchTask{..} = do
    didWork <- processTaskWork
    when didWork $
        atomicModifyIORef' (taskCount $ coordinator state) (\n -> (n - 1, ()))
  where
    processTaskWork = do
        foundSolution <- readIORef (solutionFound $ coordinator state)
        if foundSolution
            then return False
            else do
                bound <- readIORef (currentBound $ coordinator state)
                let currentEstimate = taskG + estm search taskNode
                if taskBound > bound || currentEstimate > taskBound
                    then return False
                    else do
                        let results = exploreNode search (config state) taskDepth taskNode
                            taskG taskPath taskBound
                        forM_ results $ \result -> case result of
                            Found path -> do
                                shouldReport <- atomicModifyIORef' (solutionFound $
                                    coordinator state) $ \curr ->
                                    if curr then (curr, False) else (True, True)
                                when shouldReport $
                                    writeChan (resultChan state) (Found path)

                            Next newBound -> do
                                -- This is a candidate bound higher than current iteration
                                writeChan (resultChan state) (Next newBound)

                            Stop -> writeChan (resultChan state) Stop

                        when (taskDepth < maxParDepth (config state)) $ do
                            let successors = edges search taskNode
                                validSuccessors = filter (\(Succ _ c s) ->
                                    let newG = taskG + c
                                    in newG + estm search s <= taskBound) successors

                            let numNewTasks = length validSuccessors
                            when (numNewTasks > 0) $ do
                                atomicModifyIORef' (taskCount $ coordinator state) (\n -> (n +
                                        numNewTasks, ()))
                                forM_ validSuccessors $ \(Succ l c s) -> do
                                    let newTask = SearchTask {
                                        taskNode = s,
                                        taskDepth = taskDepth + 1,
                                        taskBound = taskBound,
                                        taskPath = l:taskPath,
                                        taskG = taskG + c
                                    }
                                    writeChan (taskQueue state) (Just newTask)
                        return True


------------------------------------------------------------
-- Worker Threads
------------------------------------------------------------

worker :: (NFData node, NFData a, NFData l, Ord a, Num a, Show a)
       => SearchState a l node
       -> Search [] a l node
       -> IO ()
worker state search = do
    modifyMVar_ (activeWorkers state) (\n -> return (n + 1))
    let loop = do
            foundSolution <- readIORef (solutionFound $ coordinator state)
            tasksPending <- readIORef (taskCount $ coordinator state)
            if foundSolution || (tasksPending == 0)
                then shutdown
                else do
                    mtask <- readChan (taskQueue state)
                    case mtask of
                        Nothing -> shutdown
                        Just task -> do
                            processTask state search task
                            loop

        shutdown = do
            modifyMVar_ (activeWorkers state) $ \n -> do
                let newCount = n - 1
                when (newCount == 0) $
                    writeChan (resultChan state) Stop
                return newCount
    loop


------------------------------------------------------------
-- Parallel IDA* Search
------------------------------------------------------------


searchParallel :: (NFData node, NFData a, NFData l, Ord a, Num a, Show a, Bounded a)
               => ParConfig
               -> Search [] a l node
               -> node
               -> IO (Maybe [l])
searchParallel config@ParConfig{..} s root = do
    let initialBound = estm s root
    coord <- initSearchCoordinator initialBound
    boundState <- newIORef (NextBound maxBound)  -- Track minimal next bound candidate
    atomicModifyIORef' (activeSearches coord) $ \n -> (n + 1, ())

    state <- initSearchState coord config

    -- Launch initial workers
    forM_ [1..numWorkers] $ \_ -> forkIO $ worker state s

    -- Add initial task
    atomicModifyIORef' (taskCount coord) (\n -> (n + 1, ()))
    writeChan (taskQueue state) $ Just SearchTask {
        taskNode = root,
        taskDepth = 0,
        taskBound = initialBound,
        taskPath = [],
        taskG = 0
    }

    let mainLoop prevBound = do
            result <- readChan (resultChan state)
            case result of
                Found path -> do
                    -- Solution found; stop everything
                    atomicWriteIORef (solutionFound coord) True
                    replicateM_ numWorkers $ writeChan (taskQueue state) Nothing
                    return (Just path)

                Next newBC@(NextBound nb) -> do
                    -- Only absorb the candidate if it’s greater than prevBound
                    when (nb > prevBound) $ do
                        atomicModifyIORef’ boundState $ \curr -> (curr <> newBC, ())

                    -- Received a next-bound candidate
                    active <- readMVar (activeWorkers state)
                    tasksPending <- readIORef (taskCount coord)

                    if tasksPending == 0
                        then do

                            -- Get new accumulated bound
                            NextBound candidate <- readIORef boundState

                            if candidate /= maxBound && candidate > prevBound
                                then do
                                    let nextBound = candidate

                                    -- Update global state for new iteration
                                    atomicWriteIORef (currentBound coord) nextBound
                                    atomicWriteIORef boundState (NextBound maxBound)
                                    atomicWriteIORef (solutionFound coord) False

                                    -- 1) clean up old workers
                                    activeCount <- readMVar (activeWorkers state)
                                    replicateM_ activeCount $ writeChan (taskQueue state)
                                        Nothing

                                    -- 2) Initialize tasks again
                                    atomicModifyIORef’ (taskCount coord) (\n -> (n + 1, ()))
                                    writeChan (taskQueue state) $ Just SearchTask {
                                        taskNode = root,
                                        taskDepth = 0,
                                        taskBound = nextBound,
                                        taskPath = [],
                                        taskG = 0
                                    }

                                    -- 3) Relaunch new workers
                                    forM_ [1..numWorkers] $ \i -> do
                                        forkIO $ worker state s

                                    mainLoop nextBound
                                else do
                                    return Nothing
                        else do
                            -- There are still tasks pending, accumulate candidate and
                                continue
                            mainLoop prevBound

                Stop -> do
                    -- If Stop is read, just continue.
                    mainLoop prevBound

    result <- mainLoop initialBound
    atomicModifyIORef’ (activeSearches coord) $ \n -> (n - 1, ())
    return result
```

```
------------------------------------------------------------
-- Sequential Fallback (if needed)
------------------------------------------------------------

seqSearch :: (Foldable f, Num a, Ord a)
         => Search f a l node
         -> node
         -> Maybe [l]
seqSearch s root = rootSearch (estm s root)
  where
    rootSearch d = case dfSearch s root 0 [] d of
        Stop -> Nothing
        Found ls -> Just ls
        Next (NextBound d') -> rootSearch d'

dfSearch :: (Foldable f, Num a, Ord a)
         => Search f a l node
         -> node -> a -> [l] -> a -> SearchResult a l
dfSearch (Search goal estm edges) n g ls bound = dfs n g ls bound
  where
    dfs n g ls bound
        | g == bound && g == f && goal n = Found (reverse ls)
        | f > bound = Next (NextBound f)
        | otherwise = foldMap searchSucc $ edges n
        where
            f = g + estm n
            searchSucc (Succ l c s) = dfs s (g + c) (l:ls) bound


toList :: Foldable f => f a -> [a]
toList = foldr (:) []


canParallelize :: forall f n a l.
                (NFData n, NFData a, NFData l, Show a, Foldable f) =>
                Search f a l n ->
                Bool
canParallelize _ = True


------------------------------------------------------------
-- Public Search Functions
------------------------------------------------------------

search :: forall f a l node.
       (Foldable f, NFData node, NFData a, NFData l, Ord a, Num a, Show a, Bounded a, Ord node
           )
       => Search f a l node
       -> node
       -> Maybe [l]
search s root = unsafePerformIO $ do
    if canParallelize s
        then do
            putStrLn "\n=== Using parallel IDA* implementation ==="
            let sList = Search {
                    goal = goal s,
                    estm = estm s,
                    edges = toList . edges s
                }
            searchParallel defaultParConfig (selfAvoid sList) (selfAvoidRoot root)
        else do
            putStrLn "\n=== Using sequential IDA* implementation ==="
            let sList = Search {
                    goal = goal s,
                    estm = estm s,
                    edges = toList . edges s
                }
            return $ seqSearch (selfAvoid sList) (selfAvoidRoot root)


------------------------------------------------------------
-- Self-avoiding Search Wrapper
------------------------------------------------------------
```

```
data SelfAvoid node = SelfAvoid (S.Set node) node
    deriving (Generic)

instance NFData node => NFData (SelfAvoid node)

selfAvoid :: (Ord node) => Search [] a l node -> Search [] a l (SelfAvoid node)
selfAvoid (Search goal estm edges) = Search {
    goal = goal . node,
    estm = estm . node,
    edges = edges'
  }
  where
    node (SelfAvoid _ n) = n
    edges' (SelfAvoid trace n) =
      [ Succ l c (SelfAvoid (S.insert s trace) s)
      | Succ l c s <- edges n, S.notMember s trace ]

selfAvoidRoot :: node -> SelfAvoid node
selfAvoidRoot root = SelfAvoid (S.singleton root) root
```