Rubik's ParaCube: a Collection of Parallel Implementations for Optimal Rubik's Cube Solver

Tiane Zhu (tianez@), Chengzhi Huang (chengzhh@)

1 Introduction

A Rubik's Cube is consisted of 6 faces. For a 3x3x3 Rubik's Cube, each face is divided into 3 rows and 3 column, i.e. 9 squares. The "solved" condition of a 3x3x3 Rubik's Cube requires that each square on the same face be consisted of same color, and no two faces share the same color. Conventionally, the 6 colors are chosen to be White (W), Blue (B), Red (R), Yellow (Y), Green (G) and Orange (O).

Here is an alternative way to view the formation of a Rubik's Cube: it is consisted of 27 sub-cubes (a.k.a cubies), where 26 of them are visible. 8 out of 26 are corner cubies; 6 are center cubies; and the rest, 12, are edge cubies. A Rubik's Cube can be scrambled by rotating a row or column for 90, 180, or 270 degrees. By rotating a row (or column), one is essentially rotating the entire 3x3x1 slice.

To describe the problem, we need to describe the condition of the Rubik's Cube. To provide a solution, we need to record the scramble steps to solve it. Note that the condition of a Rubik's Cube can be described as the "solved" condition followed by a series of steps to scramble it. Hence, we need a way to formally denote each possible scrable option given a Rubik's cube.

1.1 Singmaster Notation

Singmaster notation was devised by David Singmaster. The notation uses six letters to denote the clockwise turns: F for front, B for back, U for up, D for down, L for left and R for right. More specifically, since a scramble can be turned 90, 180, or 270 degrees, the following notation is used:

1. F, B, U, D, L, R: 90 degrees clockwise
2. F2, B2, U2, D2, L2, R2: 180 degrees clockwise
3. F', B', U', D', L', R': 270 degrees clockwise (equivalent to 90 degrees anticlockwise)

There are more details to the Singmaster notation; but, for our purpose of parallelization, we do not want to dive deeper into this topic. For simplicity, hereafter, and in the documentations of solver code, we use these slightly modified definitions:

1. F1, B1, U1, D1, L1, R1: 90 degrees clockwise
2. F2, B2, U2, D2, L2, R2: 180 degrees clockwise
3. F3, B3, U3, D3, L3, R3: 270 degrees clockwise (equivalent to 90 degrees anticlockwise)

2 Summary

In this project, we implemented an optimal Rubik's Cube solver based on the Iterative Deepening A* (IDA) algorithm; similar to Richard Korf's algorithm, the algorithm heuristics is based on a corner pattern database.

This solver is equipped with multiple different parallelization strategies, including ones implemented with OpenMP and the others implemented with MPI.

We are proud to present the speedup factors we achieved with our implementations. The OpenMP implementations were measured on latedays machines with Xeon Phi's; the MPI implementaions were measured on Bridges Machines. Together with the results, we would love to share our analysis of parallelization challenges and the ideas behind improvements.

3 Background

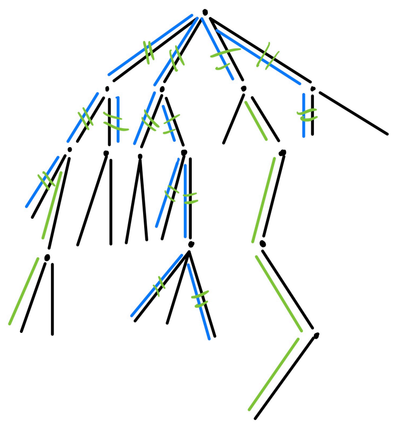
Given a scrambled Rubik's Cube, we want the solver to return the smallest number of moves to transition the input Cube to the "solved" condition. Note that our inputs can be an ordered list of scrambles to generate a Cube case from the "solved" state, or simply colors on each of the Cube's face.

Solving a Rubik's cube may not be hard. Different strategies have been proposed, e.g. Thistlethwaite's 52-move algorithm. These algorithms mostly employ the idea of grouping cubies into favorable states, which are easier to solve than pre-grouping Cube; this process is repeated until the Cube is solved. It is not hard to show that such idea cannot generate optimal solution; hence, these algorithms are not the target for us.

3.1 IDA: Choice for Optimality and Memory

To find the optimal (shortest) solution, one can easily come up with the idea of using a search. The search will be performed on a tree, where the tree nodes are states of a Rubik's Cube and the edges are transitions from a Cube state to another. Since one can always twist a Rubik's cube (possibly with repeating states), using Depth First Search (DFS) does not seem viable. Breadth First Search (BFS), on the other hand, will find the shortest solution. Therefore, our first implementation was a BFS solver. We intend for this solver to be a reference solution solver at simple problems; unsurprisingly, it would be killed when searching a solution from the tree with depth only 7. Note that without pruning, the branching factor for the tree is 18. At depth 7, the tree has already branched approximately $18^6 = 34012224$ times.

Since the optimality achieved through BFS could not scale, we turned towards using the IDA algorithm. IDA employs the general idea of a bounded guess DFS. During each iteration, a bound is selected. The tree is then traversed with DFS, where leaf nodes are the ones with no child nodes with estimated distance towards goal state remaining within bound. The bound is relaxed every time the DFS for an iteration finished with no solution found.



For example, in the above graph, iteration i may be traversing the tree over blue edges with no solutions found. Then, iteration $i+1$ may traverse with blue plus green edges.

Note that the primary concern for IDA is the constraint on memory, hence it does not store which nodes has been visited in the previous iteration (because otherwise, e.g. if leaf nodes are stored for continuation, memory will soon be exhausted given the high branching factor of the search tree). Although there are clearly repeated work in different iterations, it is not our concern here for two reasons:

1. the high branching factor makes the repeated work less noticable; i.e. the repeated work for each iteration is roughly 1/18 of the current iteration work.

2. we believe the challenges of parallelization and such repeated work is completely independent.

Here is a piece of recursive pseudo-code for IDA from Wikipedia:

```
path              current search path (acts like a stack)
node              current node (last node in current path)
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)
cost(node, succ)  step cost function
is_goal(node)     goal test
successors(node)  node expanding function, expand nodes ordered by g + h(node)
ida_star(root)    return either NOT_FOUND or a pair with the best path and its cost

procedure ida_star(root)
    bound := h(root)
    path := [root]
    loop
        t := search(path, 0, bound)
        if t = FOUND then return (path, bound)
        if t =  then return NOT_FOUND
        bound := t
    end loop
end procedure

function search(path, g, bound)
    node := path.last
    f := g + h(node)
    if f > bound then return f
    if is_goal(node) then return FOUND
    min :=
    for succ in successors(node) do
        if succ not in path then
            path.push(succ)
            t := search(path, g + cost(node, succ), bound)
            if t = FOUND then return FOUND
            if t < min then min := t
            path.pop()
        end if
    end for
    return min
end function
```

3.2 Richard Korf's Algorithm

IDA requires that the heuristic function h (which is used to estimate remaining distance from a node to solution) to be admissible, i.e. the estimation is no greater than the true remaining distance. The core idea of Richard Korf's proposal on providing a better heuristic for the Rubik's Cube search tree is to make use of a series of pattern databases. For example, one database is the Corner Pattern Database (CornerDB); this database stores all possible mappings between an encoded corner state and an h value, i.e. the highest possible number of steps to the "solved" condition from any Rubik's Cube conditions with matching corner cubies state. Similar ideas can be employed on edge states (edge databases). For our implementation, only the CornerDB is used.

3.3 Computational Structure and Solution Depth

Let K be the average branching factor hereafter. For now, without pruning, $K = 18$.

First, since IDA is an iterative algorithm, we define the search graph for an iteration i to be G_i, and the bound to be B_i. B_i increases as the number of iterations increases; more specifically, B_{i+1} is always the smallest estimation larger than B_i towards the goal state. Relaxing bounds allows G_i to keep growing bigger. Note that although B_i can sometimes increase by more than 1, the current estimation is always increased by 1 every time a child node is visited from a parent node (in the search tree for Rubik's Cube, as the path cost is always 1). Therefore, G_{i+1} will always be at least one level bigger than G_i, with roughly K times more nodes.

An important note, since IDA is an algorithm that stops once a solution is found, if solution is found at depth $i = I$, the algorithm will likely not traverse the entire G_I. Even for the blue and green edges demonstration above, some green edges will be traversed before the blue edges in a serial execution, as the sequential algorithm is inherently Depth first.

For a Rubik's Cube case, we define solution depth as the number of scramble steps needed to solve it, which is luckily the same as the tree traversal depth. Note that if a solution requires two F1 moves, its depth is 1, since it can be completed with one F2 move.

3.4 Cube Representation and Key Data Structures

Our solver used the two pieces of information to store a Cube and perform transitions on it:

1. 6 three-by-three 2d arrays of bytes to represent colors of each face.

2. 1 eight-byte array to denote the orientation of each corner Cubie. Each Rubik's Cube has these following 8 corner cubies (represented by the color combinations) with their corresponding starting position (represented by the three faces they are on):

   ```
   Cubie  = [RBY, RGY, RGW, RBW, OBW, OBY, OGY, OGW]
   Corner = [ULB, URB, URF, ULF, DLF, DLB, DRB, DRF]
   ```

   With scrambles, each Cubie may move to different corner locations. Byte i in the 8-byte array corresponds to the orientation of the Cubie currently on `Corner[i]`. Note that orientations can have 3 values, enumerated as 0, 1, and 2. Both orientation and Cubie coloring are encoded into a database lookup key. According to numerous different references, there are $8! \times 3^7 = 88179840$ possible corner conditions (considering different placements, and orientations of each Cubie). (This number is also the possible number of conditions for an 2x2x2 Cube).

Next, for the search tree representation, node representation in the search tree has evolved multiple times. For IDA implementations, a node data structure is used to record the current problem. It evolved from partial problem state into complete problem state (as in the earlier recursive IDA implementation, some parameters are carried by the function calls). The complete problem state for one iteration of IDA includes an array of nodes, each with the following fields:

1. A current state cube

2. op, the next transition to explore (Any of F1, F2 ... D2, D3)

3. g, the current cost (which is also the depth of the search tree)

4. min, next iteration bound (minimum of current iteration estimates exceeding current iteration bound)

Finally, the database is adapted from another Rubik's Cube Solver repository: `https://github.com/benbotto/rubiks-cube-cracker`. This database is pretty well optimized in terms of size and used arrays of values of size 4 bits to conserve space; it is possible to store heuristic values this way, since the key is simply the index and the heuristics value for any corner condition towards goal state is no greater than 15. (Recap: heuristic value for a corner condition is the smallest possible number of steps to the goal state from any Cube state with matching corner condition).

Note, this section is meant to introduce the main data structures shared by all implementations of the algorithm. OpenMP and MPI implementation-specific data structures will be covered below in their separate sections, along with implementation details and analysis of progression.

4 Approach

Here is a list of Implementations progression for the OpenMP Implementation:

1. BFS Implementation (`BFS`)
2. IDA Recursive Sequential Implementation without Korf Database
3. IDA Recursive Sequential Implementation, (`IDA_REC_SEQ`)
4. IDA Recursive OpenMP Implementation with only first level parallelized
5. IDA Recursive OpenMP Implementation parallelizing at a level to spawn enough tasks (`IDA_REC_OMP`)
6. IDA Iterative Sequential Implementation, (`IDA_ITER_SEQ`)
7. IDA Iterative OpenMP Implementation with all tasks created all at once (`IDA_ITER_OMP_MAIN_WORKER`)
8. IDA Iterative OpenMP Implementation with each worker progress top few levels of tree and grab tasks as needed (`IDA_ITER_OMP`)
9. IDA Iterative OpenMP Implementation with worker grab task and task stealing (`IDA_ITER_OMP_UNEVEN`)
10. IDA Recursive OpenMPI Implementation with only first level parallelized (`IDA_MPI`)
11. IDA Recursive OpenMPI Implementation with as many nodes as the processors (`IDA_MPI2`)
12. Incomplete IDA Iterative OpenMPI Implementation with job stealing (`IDA_MPI3`)
13. IDA Recursive OpenMPI Implementation with more frontier nodes than the processors and best-first expansion (`IDA_MPI4`)

The names in the parentheses are method (implementation) names that can be used to parametrize the `paracube` binary. For example,

```
./paracube -f input/t11 -t 4 IDA_ITER_OMP
```

will start the solver, and solve the given input/t11 case using the `IDA_ITER_OMP` method. For MPI,

```
mpirun -np 17 ./paracube IDA_MPI4 -p 17 -f input/t11
```

will start the solver, and solve the give input/t11 case with 1 master node and 16 worker nodes using the `IDA_MPI4` method.

4.1 Pre-IDA_REC_OMP

Here is a quick recap on the algorithms we implemented for the checkpoint report.

The naive BFS implementation can be used to resolve cubes with Solution depth of no more than 6. It faces significant memory constraints as the branches expand exponentially with a high branching factor.

Next, we tested the basic IDA Recursive Sequential Implementation without Korf Database to resolve cubes of depth 9 (rather quickly) on GHC machines.

With the help of the CornerDB, the IDAKDB (a.k.a `IDA_REC_SEQ`) implementation can resolve cubes up to 11 rather quickly.

The last implementation for the checkpoint report was the ParaIDAKDB implementation; this implementation showed speedups for cube solving for cases t8 through t11. The speedup graph and analysis can be found in the checkpoint report.

4.2 Speedup through pruning

Before further progressing onto OpenMP implementation, we introduced a function named `can_prune`, it takes a previous transition and a current transition as arguments. The logic is as follows:

First, If the previous transition is any of F1, F2 or F3, we do not explore current transitions of F1, F2 or F3. The reason is that any combinations of such two moves, will have effects on the cube equivalent to one of the moves, e.g. F1 F2 == F3, F3 F3 == F2. Similar reasons apply for all transitions in classes B, L, R, U, and D.

Next, If the previous transition is any of

1. F1, F2 or F3, we do not explore B1, B2 or B3
2. L1, L2 or L3, we do not explore R1, R2 or R3
3. U1, U2 or U3, we do not explore D1, D2 or D3

This is because exploring any transition in the F class then B class is equivalent to exploring any transitions in B class then F class. Therefore, for the two commutative classes, we kept only one pair. E.g. U2 D1 == D1 U2.

After pruning, we estimated the branching factor K with a count of transitions from each iteration. The resulting count showed that K is approximately 12.

5 OpenMP Implementation

5.1 IDA_REC_OMP and IDA_ITER_SEQ

The `IDA_REC_OMP` implementation was written to distribute enough tasks to different workers. The number of tasks created is dependent on the number of workers; we created a simple function mapping the number of workers to tree depth (`task_creation_depth_limit`) for task creation. For example, 16 workers may map to depth 3 and create 12^3 tasks, allowing each worker an average of 100 tasks. Load balancing is the main concern of the implementation at this phase. We believed that with finer grained tasks, load balancing will be better and the algorithm will scale better. Here is a piece of pseudo-code for the algorithm:

```
procedure ida_star(root)
    bound := h(root)
    path := [root]
    loop
        #pragma omp parallel {
            #pragma  omp single
                t := search_shared(path, 0, bound)
        }
        if t = FOUND then return (path, bound)
        if t =  then return NOT_FOUND
        bound := t
    end loop
end procedure

function search_shared(path, g, bound)
    node := path.last
    f := g + h(node)
    if f > bound then return f
    if is_goal(node) then return FOUND
    min :=
    for succ in successors(node) do
        if succ not in path then
            if node.depth > task_creation_depth_limit {
                #pragma omp task firstprivate(...) {
                    t := search(path, g + cost(node, succ), bound) // the same as the previous search
                    if t = FOUND then #pragma omp critical { update path }
                    if t < min then min := t
                }
            } else {
                path.push(succ)
            }
        end if
    end for
    return min
end function
```

Although speedups were observed for core counts up to 16, we were not satisfied with the results. For performance, we analyzed that the fidelity between this version of parallel algorithm and the sequential algorithm is too low. The net effect was that speedups over different runs were not stable (since the tasks were picked up by the threads in different orders during different runs); this randomness caused evaluation hard. We then tried to change the original recursive implementation into an iterative implementation to enable more freedom with task management when compared to the recursive calls.

Also note that, normally, we would expect the the worker who has found a solution to notify all other workers to terminate. This feature is also missing in this version of code. The amount of unnecessary work has greatly harmed the speedup from having parallel worker threads.

This missing notification is added in the next version of code, the iterative implementation of sequential IDA. With `IDA_ITER_SEQ`, to keep track of the nodes and current traversal, we used an array of tree nodes from the current traversal node to root of tree. This array is used as a stack for iterative IDA and an array when outputting results. The *op* field in the nodes definition (problem state definition above) is used to record which transition to explore next. Current node *op* is compared with previous node *op* to prune unnecessary branches, as mentioned in section 4.2.

5.2 IDA_ITER_OMP_MAIN_WORKER

From the sequential iterative IDA, we re-implemented the same parallelization idea of having a main worker spawn tasks, and letting other worker threads grab the tasks. The main worker will join the other workers once task creation is complete. The purpose of this algorithm is to serve as a basis for improvement in the IDA algorithm. Additionally, without random task grabbing order, we were able to eliminate the randomness mentioned above.

Furthermore, here are the three improvements for this algorithm when compared to the previous `IDA_REC_OMP`:

1. The worker that found a solution now notifies the other works by writing a global flag. The frequency

of notification is purely dependent on how often each worker checks the global flag. For simplicity, we let each worker check the flag at each node in this implementation. Later on, we reduced the frequency to whenever a node is fully traversed (i.e. all child nodes that should be visited in current iteration has been visited).

2. A new data structure was introduced: a list of tree nodes shared by many workers. This is similar to the data structure in the recursive implementation; the main difference is that in the previous recursive implementation, this list was not shared. We hope that eventually many workers can operate on this list and perform some sort of task grabbing in order they'd like. Some potential problems including contention will be explored below.

3. The function mapping inputs to number of tasks to create has been changed. A very important aspect of the algorithm is that for any sub-tree, each branch from the sub-tree root may be of vastly different heights due to heuristics and pruning. This means that simply dividing the task up for each thread will not suffice when the tree depth is high, i.e. a case with high solution depth at high iteration count. We introduced a configurable maximum task size, hoping to make sure that tasks are not too different. The same function now generates a depth between minimum task creation depth (based on number of workers) and maximum task size.

5.3 IDA_ITER_OMP

We experimented with different maximum task size, in terms of depth of subtree, D, to search. Through various experiments and measuresments, we explore that D = 9 seems to be the sweet spot for cases up to t14 (solution depth 14). With smaller D, too many tasks were created (note that tasks were created sequentially by the main worker previously) and the task management cost became too high. With larger D, we would be gradually reverting back to the state of the implementation before introducing D.

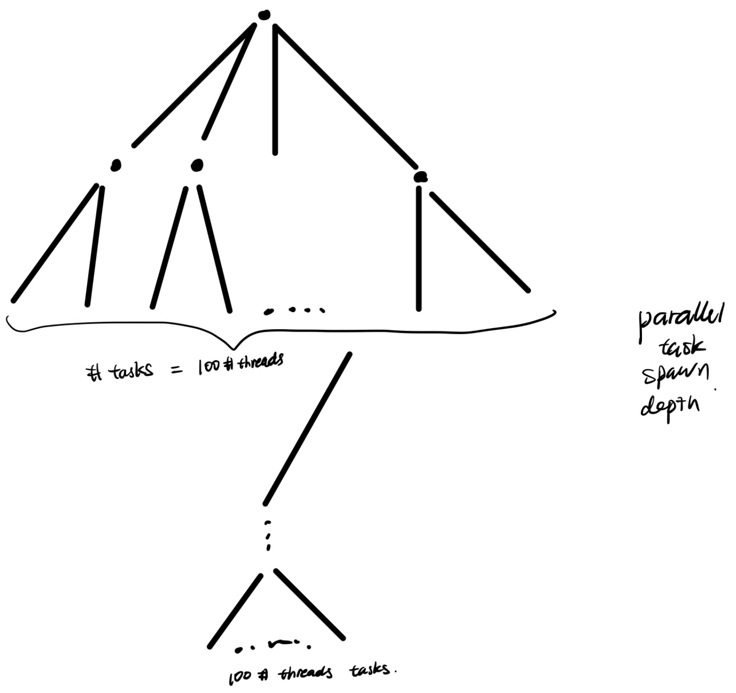
Next, an implementation where tasks were created by worker threads as needed was introduced. We hope to avoid task management cost and amortize out task creation cost to different workers. Recall that we started sharing the data structure for top level task creation path. Each worker process in turn gains exclusive access on this path. Tree traversal will be continued from where the previous task left off. Once current node depth reaches D, the worker would stop, copy problem into its private problem storage, and release exclusive access before starting to work on the sub-problem.

A huge concern here is the synchronization wait time for the sequential access to the shared data structure. Using a wallclock timer, we measured that for the test cases we had, often enough, although workers are waiting on each other at the beginning, the tasks were different enough so that the workers were barely waiting for exclusive access when trying to grab a next task. The wait time introduced by this synchronization with D = 9 is for case t13 is less than 1%.

The speedups we observed for this implementation is very good, even very similar to the `IDA_ITER_OMP_UNEVEN` task stealing implementation below. We were hoping to further improve the algorithm along the ideas of that "each branch from the sub-tree root may be of vastly different heights due to heuristics and pruning". Hence, task stealing idea was introduced next.

5.4 IDA_ITER_OMP_UNEVEN

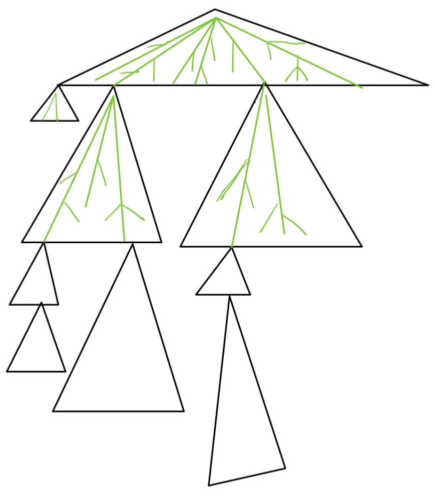
Firstly, to better illustrate the purpose of task stealing, consider the following search graph for an iteration



With one level of task spawn, in the most extreme case, only one out of the many tasks is non-trivial. This task may be given to a thread and the other threads are simply waiting for it to perform sequential algorithm. We can reasonably expect the speedup from this scenario to be abysmal. Note, this imbalanced subtree property of IDA is eventually limiting to the overall speedup. (We will discuss in further details in the Results and Analysis sections).

To enable task stealing, after allocating their local copy of problem storage, each worker exposes a pointer to the storage into a global pointer array. For simplicity, we also defined a task stealing depth limit S (with a best value 3 from experiments and measurements). After repeatedly dividing tasks through stealing, ultimately, the task spawning is stopped when the value of (IDA iteration bound - current node depth) reaches threshold value S.

To some extent, this is less about task stealing than enabling smart multi level task spawn. In the figure below, the triangular areas are subtrees that are not pruned, the green edges are example edges of the entire tree. We want to load balance and spawn tasks at different levels, while some tasks has already been handed over to individual worker threads:



Note that the order of worker thread to steal from is quite important. This is one of the ways through which we can reduce possible contention. For example, we have experimented with the strategy below (see pseudocode). We believe this order will, to the largest degree, reduce the amount of workers trying to steal from the same worker unless there are only few to steal from. Consider 8 threads, instead of a simple for loop of i from 0 through 7, thread 5 would steal from workers in the following order:

```
order (i):              0 1 2 3 4 5 6 7
thread_id to steal from: 5 4 7 6 1 0 3 2
```

which can be generated iwth the following logic

```
thread_id[0] = omp_get_thread_num()
for i = (1..7)
    j = next_power_2(i) / 2
    i_alt = i - j
    thread_id[i] = thread_id[i_alt] xor j
```

Although we have observed some speedup with this order than the simple for loop order (which will be the same for each worker thread). The speedup was not significant enough given our test cases, and hence it was not adopted eventually.

Note this algorithm is still having very different execution order when compared to the sequential IDA, since the parallel portion of it is inherently breadth first, i.e. workers work on independent tasks first. As mentioned in the final Results and Analysis section, we eventually determined that it is not reasonable for all workers to work on the same subtree at once just to create fidelity; reasons include contention and, more importantly, the goal state can very well be hidden in the last node that the sequential algorithm is going to traverse in an iteration, in which case, there will be no problem of fidelity.

6 OpenMPI-IDA*

As timed allowed, we decided to further progress out implementation using OpenMPI. We were hoping that with this new implementation platform and programming model, more load balancing techniques can be explored.

Our IDA* algorithm with the message passing model will generally be divided into two parts: the master node will be responsible for generating enough frontier nodes and then assign each frontier node to worker nodes for parallel searching. After obtaining frontier nodes, worker nodes will start sequential IDA* and then report the searching bound back to the master node.

6.1 Basic MPI-IDA* algorithm

6.1.1 Generating frontier nodes

Initially there is one root, it will be expanded into as many frontier nodes as processors. Out of the many algorithms to perform this task, we arbitrarily chose breadth-first search for this initial distribution. For the cases where the expansion of a root node exceeds the number of processors, we only generate a part of its children and contract the other parts to its parent as well as an indication (record) on which of the successor nodes has been generated. The initial cost bound is set to be the minimum of estimated distance to goal state (f) value of all generated frontier nodes. Note f is the sum of g, which is the cost of current node from root, and h, the corresponding heuristic value read from database given the corner conditions.

6.1.2 Assigning frontier nodes

After generating enough frontier nodes, the master node will start sending frontier nodes to worker nodes. Notice that instead of sending frontier nodes themselves, we compress the information by sending the path from the tree root to frontier nodes. After receiving messages from the master node, the worker nodes will start performing sequential IDA* independently and report back the minimum f value that exceeds the current bound. If a worker node finds an optimal solution, it will report back to the master node, and master node will attempt to abort the MPI session immediately. If the worker nodes fail to find the optimal solution in an iteration, the master node will update the cost bound to the minimum of all f values that the worker nodes reported back, and broadcast this updated cost bound to every worker node for starting their next iterations.

6.2 Improving MPI-IDA*

In the basic MPI-IDA* algorithm shown above, since we only generate as many frontier nodes as the number of processors, in every iteration, worker nodes that finish their jobs earlier will become idle and wait for the other worker nodes, which will result in poor load balancing. In order to mitigate this issue, we propose a better way to generate frontier nodes.

6.2.1 Better Initial Distribution

First observe that tree nodes with the same f value usually tend to generate sub-trees of comparable size. Bearing this in mind, during the generation of frontier nodes, we implement best first expansion, i.e. we only expand those having lower f value until number of frontier nodes equal to the number of processors. This will give us better load balancing by improving initial distribution. The speed-up of this best-first expansion will be discussed in section 7.2. The data structured used here is a priority queue. The task generation based on f is key point here.

6.2.2 Dynamic Workload Assignments

Within each iteration, a processor which completes searching its sub-tree will wait for remaining processors to finish. The ideal solution would be to let idle nodes steal work from busy nodes. Unfortunately, we were not given time to implement this in OpenMPI. Instead, we create more frontier nodes (4000 in our case) than the number of processors so that whenever a worker node finishes processing its sub-tree, it can immediately inform the master node and get the other part of the work. This will ensure a better form of load balancing except that idle nodes still need to wait for busy nodes when the remaining number of frontier nodes are less than the number of processors.

Our final version MPI-IDA* will be the following:

```
num_expand        number of frontier nodes to expand
cube              scrambled cube
init_node         node containing initial scrambled-cube state, op path and depth
is_contract       is partial contraction
pq                priority queue containing un-expanded nodes, nodes with smaller f-values
                  will be popped first
n_current         top node in the priority queue
frontier_nodes    array containing all frontier nodes
num_termination   number of worker nodes that finish their jobs in an iteration
bound             threshold in current iteration
next_bound        cost threshold in next iteration
contract(node)    function to contract unvisited children nodes to their
                  parents with indication of visited nodes
successors(node)  node expanding function
search(path, g, bound) sequential ida* search algorithm
init_node_from_cube(cube)   function to initialize node from cube

function best_first_expand(init_node, num_expand)
    pq.push(init_node)
    while (pq.size() < num_expand)
        n_current = pq.top()
        pq.pop()
        for (child : successors(n_current))
```

```
            if (child is not the last successor of the current node)
                pq.push(child)
                break
            else
                pq.push(contract(child))

    while (pq is not empty)
        frontier_nodes.push_back(pq.top())
        pq.pop()
    return frontier_nodes

function run_master(cube)
    init_node := init_node_from_cube(cube)
    frontier_nodes := best_first_expand(init_node, num_expand)

    while (1)
        MPI_Recv(&buf, ANY_TAG, mpi_sta)

        // Receive ALLOCATE_TAG from worker node
        if (mpi_sta.TAG == ALLOCATE_TAG)
            if (i == frontier_nodes.size())
                num_termination++
            else
                // Send START_TAG and frontier node to worker node i
                MPI_Send(frontier_nodes[i], START_TAG, mpi_sta.MPI_SOURCE)
                i++

        // Receive UPDATE_TAG from worker node
        if (mpi_sta.TAG == UPDATE_TAG)
            if (buf.found == FOUND)
                MPI_Abort()
            else
                next_bound := min(buf.bound, next_bound)

        if (num_termination == num_worke_nodes)
            bound := next_bound
            i = 0
            num_termination = 0
            for (n : worker_nodes)
                MPI_Send(frontier_nodes[i], START_TAG, n)

function run_worker()
    while (1)
        // Request jobs from master node
        MPI_Send(NULL, ALLOCATE_TAG, 0)
        // Waiting for START_TAG from master node
        MPI_Recv(&node, START_TAG, 0)
        ans = search(path, node.depth, node.bound)
        // Report result back to master node
        MPI_Send(&ans, UPDATE_TAG, 0)

procedure ida_solve_mpi(cube)
    if (rank == 0)
        run_master(cube)
    else
        run_worker()
```

7 Results And Analysis

To our very best understanding, IDA with Rubik's Cube heuristics is not simple to parallelize. We are proud of the results we obtained, since there are many reasons for which, achieving linear speedup is hard at high degree of parallelism.

7.1 IDA_ITER_OMP_UNEVEN

We present the results from a few aspects.

7.1.1 Absolute Runtime

For the checkpoint version of OpenMP parallelization, with 4 threads, input case t11 requires approximately 40000ms to complete. For the task stealing OpenMP implementation, this same input case t11 requires only 4880 ms to complete with 4 threads. This is more than 8× faster. On average, the speedups for other cases are also very noticable. Note that since the new algorithm scales better, the results at higher core counts will only be better than 8×. Below are the table to compare runtimes for cases t11 and t10 between the two implementations:

	1	2	4	8	16
t10	15970.5066	10290.5766	6796.0824	4425.493	3753.4566
t11	93589.3556	60826.4756	39414.3522	28903.532	24988.423

Table 1: Checkpoint OMP absolute runtime in ms

	1	2	4	8	16
t10	3335.111	2031.965	1199.159	753.198	482.133
t11	16582.16	7937.705	4887.533	2244.976	1942.877

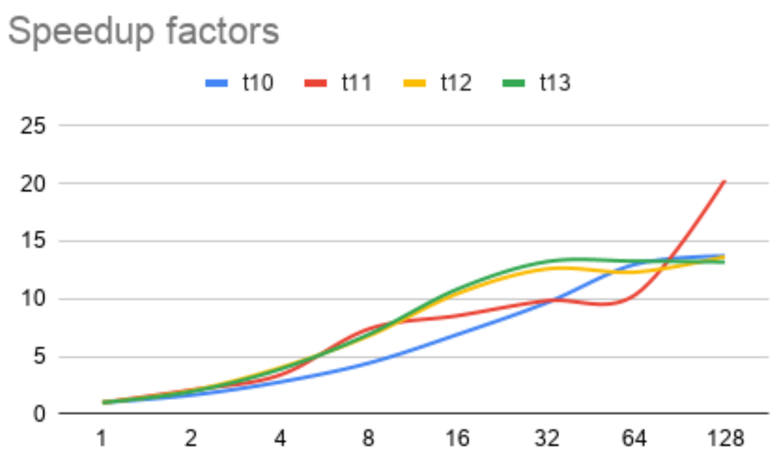
Table 2: Task Stealing OMP absolute runtime in ms

7.1.2 Scaling Factor

First, we present the scaling factor results:

	1	2	4	8	16	32	64	128
t10	1	1.641323054	2.78120833	4.427933956	6.917408682	9.657470898	12.99842933	13.70223789
t11	1	2.089037071	3.392746402	7.386341769	8.534848063	9.825481181	10.37752888	20.28068828
t12	1	2.023726278	4.019036572	6.785506016	10.46016241	12.58392021	12.30555961	13.6365893
t13	1	1.947263426	3.894505955	6.924669819	10.84914962	13.21445822	13.25753313	13.18187134

Table 3: Speedup over num threads = 1



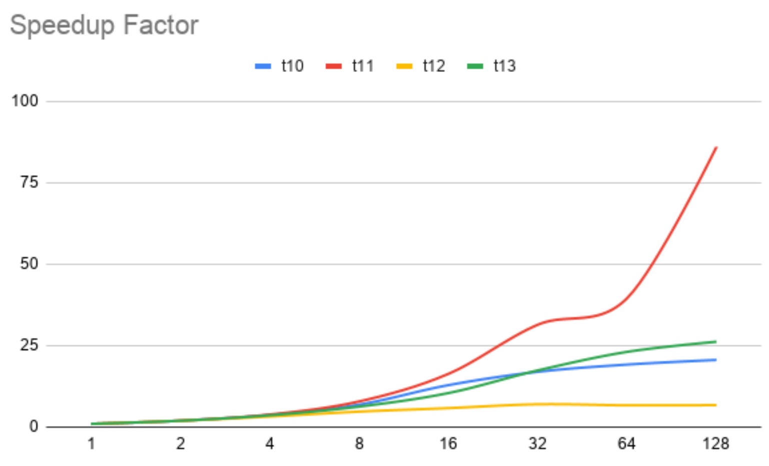
Note that it may appear that scaling has slowed down, but notice for t11, the scaling at num_threads = 128 is great. This measurement is persistent, and we provide a detailed explanation below in the analysis portion, as for why the scaling factor is not always linear, or may even appear to be inconsistent (i.e. in the t11 case) or superlinear.

7.2 MPI_IDA*

The following is the scaling factor results with `IDA_MPI4`

	1	2	4	8	16	32	64	128
t10	1	1.95584	3.71722	6.89632	12.88925	16.98076	19.20603	20.63036
t11	1	1.99426	3.86253	7.87419	16.34622	31.43206	39.57494	86.04127
t12	1	1.94990	3.25169	4.75886	5.83740	7.01641	6.70384	6.75849
t13	1	1.96897	3.64491	6.29554	10.45602	17.41019	23.11048	26.21127

Table 4: Speedup over num worker nodes = 1



Note for test case t12, the scaling factor is pretty low, while the other scaling factors are fairly decent. Test case t11 is an interesting case. We believe that the scale up is achieved by doing no unnecessary work (see section 8.2). We speculate that test case t12 may not be scaling so well for similar reasons.

8 Analysis

Given the complexity of tasks for IDA*, which we will explain below, we believe MPI and OpenMP are far superior platform choices than GPUs.

8.1 Property of IDA with Korf Database as heuristics

Firstly, we introduce the test cases used to perform this analysis. Given a series of steps to resolve a Rubik's Cube case, these steps can be any of the steps in these three lists:

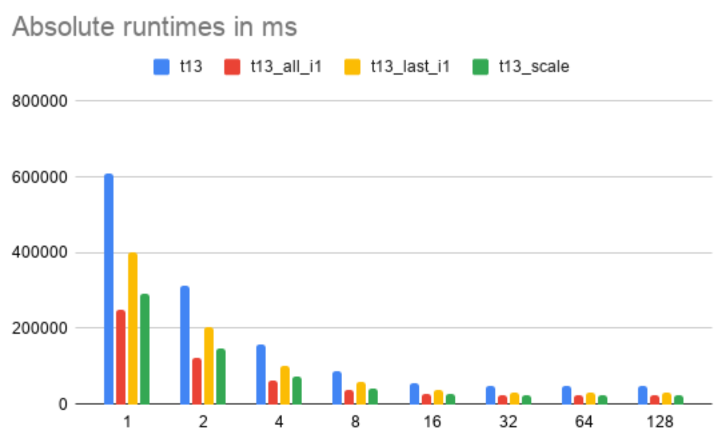
1. $S_1 = [F_1, B_1, L_1, R_1, U_1, D_1]$
2. $S_2 = [F_2, B_2, L_2, R_2, U_2, D_2]$
3. $S_3 = [F_3, B_3, L_3, R_3, U_3, D_3]$

Since we are not aware of the solutions steps needed ahead of time, there is no reason to enforce one order of traversal over another. Hence, for simplicity, the order of our algorithm's node traversal is always in S_1, S_2, S_3. This means that to find a solution, `U = [F1 B1 U1]`, will take strictly less time than the solution `V = [F3 B3 U3]`.

Our normal t1-t13 test cases all contain solutions of pattern similar to `V`, i.e. all steps come from S_3. Hence, we created a few other test cases:

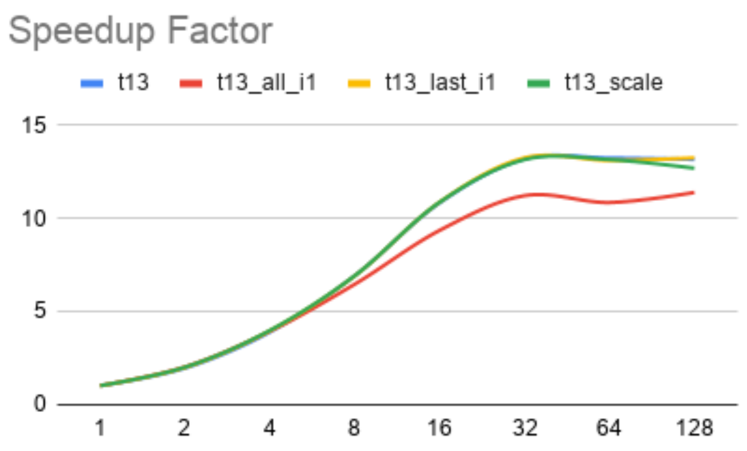
1. t13_all_i1, where the solution depth is still 13, but all solution steps come from S_1.
2. t13_last_i1, where the solution depth is also 13, but only the last solution step comes from S_1.
3. t13_scale, where the solution depth is also 13, but one of the 13 solution step (higher up in the search tree, earlier in the traversal steps) comes from S_1.

Had we created test case t13_first_i1, (where the solution depth is also 13, but the first solution step comes from S_1), we would expect to see similar results when compared to t13_scale. As we will demonstrate, the order of traversal in the top levels of the search tree matters a lot for algorithm's absolute speed. See below graph for `IDA_ITER_OMP_UNEVEN` on the new cases:



We can observe that, as expected, the absolute time in terms of execution is the lowest for t13_all_i1, next lowest for t13_scale. Next that t13_scale is still much faster than t13_last_i1, since the solution step from S_1 is much higher up in the search tree.

Here are the speedup measurements when running `IDA_ITER_OMP_UNEVEN` on these new test cases:

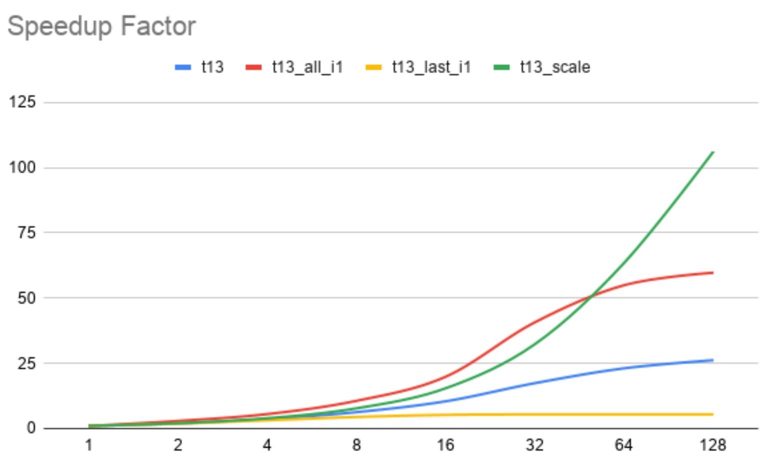


	1	2	4	8	16	32	64	128
t13	1	1.9472	3.89450	6.92466	10.8491	13.2144	13.2575	13.1818
t13_all_i1	1	2.0091	3.9305	6.4686	9.3658	11.2325	10.8604	11.4029
t13_last_i1	1	1.9795	3.9605	6.9173	10.8892	13.2720	13.1271	13.2671
t13_scale	1	1.9976	3.9789	6.9260	10.8838	13.1754	13.1725	12.7074

Table 5: Speedup over num threads = 1

We can observe that scaling is much weaker for the t13_all_i1 case with `IDA_ITER_OMP_UNEVEN`. This is expected, and alludes to the next section analyzing the part of work that the parallel implementations will carry out, while the sequential implementations will not perform. Note this is also the ultimate limiting factor for which parallel implementations do not always scale as well in the Rubik's Cube Solver.

Here are the speedup measurements when running `IDA_MPI4` on these new test cases:



	1	2	4	8	16	32	64	128
t13	1	1.96897	3.64491	6.29554	10.45602	17.41019	23.11048	26.21127
t13_all_i1	1	2.91462	5.54191	10.61928	19.85256	40.74670	54.97080	59.74378
t13_last_i1	1	1.89606	3.16883	4.44172	5.24991	5.44232	5.45020	5.46048
t13_scale	1	2.06226	3.95001	7.75040	15.43716	32.35369	63.52471	106.24747

Table 6: Speedup over num worker nodes = 1

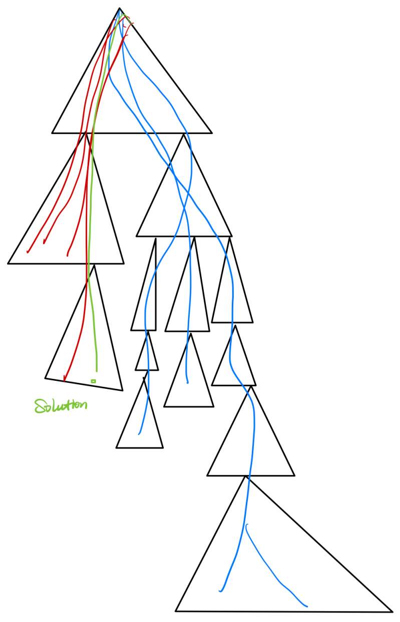
The `MPI_IDA4` numbers are more interesting to observe. We failed to find a detailed explanation on why the scaling factor for case t13_last_i1 is much lower. We speculate that this is caused by the same reason as we will explain in the next section.

8.2 Efficiency factor of the algorithm in Parallel

In this section, we intend to provide an analysis on the main reasons why we believe parallelizing IDA* algorithm with the Korf Database as heuristics becomes increasingly harder as the degree of parallelism increases.

1. IDA* does not traverse the entire graph at its final iteration. The amount of work at the previous iterations is minimal when compared to the last with K $\approx$ 12 after pruning.

2. To avoid synchronizations costs, parallel algorithms almost always avoid dependencies. Hence, the fidelity between parallel implementation and sequential implementation is low. In cases where not the entire graph is traversed, parallel algorithm is doomed to perform unnecessary work. (If we know the solution placement ahead of time we may be able to avoid this, but we do not understand enough about the heuristic values in Korf's Pattern Database; these heuristic values would affect the search tree formation greatly.)



To be more specific, when traversing the above tree and find the solution at the bottom of the green traversal path, a sequential algorithm would finish red traversal paths, then continue onto green and return as soon as solution is found. However, for parallel algorithm, multiple workers would speed up traversing the red portion greatly. One worker will then continue onto the green path, while its peers will work on the blue paths (which are unnecessary). At this point, the speedup brought by the introducing more workers into running the algorithms is reduced. This is likely the reason why the t13_all_i1 case is having a reduced speedup factor, as the additional workers are simply working on the blue tasks.

Given the above two limitations, if we want to achieve good speedup when compared against the sequential algorithm, we must find the balanced point of creating tasks and allow dependencies in algorithm; i.e. to allow some more workers to work on the green path and less workers to work on the blue path. However, this is against the spirit of parallelization and, in particular, hard since we do not know where exactly the green path is. As the balanced point for our implementations, the dependencies we are allowing here is achieved through task stealing and sub task granularity. There is certainly future work in this area, which we will mention below. But, again, for the reason that we cannot expect where the solution will hide in the search tree, there is no reason to introduce too much dependency, which may simply cause slow down.

Note that the above diagram also demonstrated how super linear speedup may be achieved (e.g in the OpenMPI implementations). If the green task is a very light weight task, while the red tasks are very heavy weight, parallelization may allow workers to find the solution before even finishing all the red tasks as in the sequential case, or finishing some of the red tasks as in cases where less workers are allocated.

8.3 Attempt for better Load Balancing

As discussed above, for the MPI implementation, during the initial distribution, we only expand on nodes having smaller f values so that all generated frontier nodes will have sub-trees of comparable sizes. Additionally, we divide the entire search into finer grained pieces (hence more) so that idle workers will be dynamically allocated more tasks. Note that smaller f values represent further distances from the goal state; hence these nodes will more likely to be a root node for larger sub-trees. Dividing up larger trees and keeping smaller trees is likely going to produce better balanced load, when compared to dividing up smaller trees and keeping larger trees.

9 Future Work

9.1 Increase Fidelity between parallel and sequential implementations

To clarify, we are listing this item as a possible future work direction, although we are not fully convinced about its value. The supporting reason would be that, for the final iteration of IDA*, performing depth first will likely find solution faster when compared to breadth first (and incur less work). Hence, fidelity to the sequential algorithm is probably going to allow measurements be closer to a near optimal speedup. However, since, again, we cannot guess where the solution will hide in the search tree, and the search order of the sequential algorithm can be any order, we did not chose to implement the parallel algorithm this way. (Not to mention that we also want to avoid synchronization costs).

9.2 More detailed analysis of the heuristics numbers in the Korf Pattern database

This is a tasks that may allow a more detailed design of the parallel algorithms (e.g. how to spawn tasks, etc.,). We believe that the heuristic values in the Pattern Database will educate us about the formation of the search tree. However, we did not perform this task due to time limitations and also that this may not be a very scalable task, e.g. what about the database for a 4x4x4 Cube? It may indicate a very different search tree formation than the one for a 3x3x3 Cube, so that our well-designed algorithm for 3x3x3 Cube will not work well for a 4x4x4 Cube Solver.

10 References

1. Nargolkar, A. (2006) Solving the Rubik's Cube with Parallel Processing. Arlington, TX: the University of Texas. Available From: UTA Libraries [accessed 14 Oct 2020].

2. Cook, D. Varnell, C. (1998) Adaptive Parallel Iterative Deepening Search. Arlington, TX: Journal of Artificial Intelligence Research. Available From: JAIR Archive [accessed 29 Oct 2020].

3. V. Nageshwara Rao, Vipin Kumar and K. Ramesh (1987) A Parallel Implementation of Iterative-Deepening A*. Austin, TX: AAAI-87 Proceedings. Available From: AAAI Archive [accessed 29 Oct 2020].

4. Z. Hafidi, E-G Talbi and G. Goncalves (1995) Load Balancing and Parallel Tree Search: the MPIDA* Algorithm. Lille, France: PARCO. Available From: Semantic Scholar Archive [accessed 29 Oct 2020].

11 Work Distribution

Nov 4 - Nov 16

- ✓ Implement software from scratch, Cube representations etc., and benchmarking tools for different implementations;
- ✓ Complete basic sequential implementation for resolving cubes in small number of steps;
- ✓ Parallelize the basic implementation to adapt to large number of steps;
- ✓ Measure all implemented code

Nov 16 - Nov 23

- ✓ Implement the more sophisticated (Iterative Deepening A based) algorithms (Korf DB) discussed in the paper.
- ✓ Construct IDA with database (Database adapted from this repo: `https://github.com/benbotto/rubiks-cube-cracker/tree/2.2.0`)
- ✓ Perform measurements for IDA with Korf (only corner) DB over BFS.

Nov 23 - Nov 30

- ✓ Adapt database complete. (We spend some time to port our code and make sure our model works with the given database).
- ✓ Parallelize IDA with Korf DB using OMP, and perform measurements.
- ✓ Incorporate different heuristics and branch pruning techniques that we came up with (work in progress, delayed as we were blocked previous on constructing our own DB)

Nov 30 - Dec 4

- ✓ (tianez) continued improvement of branch pruning techniques and OMP.
- ✓ (chengzhh) MPI implementation of the optimal cube solver. We expect the OMP and MPI algorithms to be different.

Dec 4 - Dec 9

- ✓ (team) Measurements over all implemented algorithms and understand the new performance characteristics.
- ✓ (team) deep dive into OMP and MPI implementations

Dec 9 - Dec 14

- ✓ (team) deep dive into OMP and MPI implementations
- ✓ (team) Prepare for poster session and final report.

We would like a 50/50 division.