Vrije University of Amsterdam

An Iterative Depending Depth First Search (IDS) Approach to Parallelize Rubik's Cube Problem using Ibis Library

Java Assignment Report

Ehsan Sharifi Esfahani
Student Number: 2603181

January 2019

Introduction

Rubik's Cube is a 3-D combination puzzle cube invented in 1974 by Erno Rubik, a Hungarian professor. The standard version is a 3-D combination puzzle with six faces covered by nine stickers, each of the faces have one of six colors; white, red, blue, orange, green and yellow. The puzzle is scrambled by making a number of random moves, where any cube can be twisted 90,180 or 270 degrees. The task is to restore the cube to its goal state, where all the squares on each side of the cube are of the same color. To solve a scrambled Rubik's cube, one needs an algorithm, which is a sequence of moves in order to come closer to a solution of the puzzle [1].

In this assignment, a parallel algorithm for solving the Rubik`s cube was implemented using Ibis libraries. Then, a performance analysis is done to benchmark the given solution.

Parallel algorithm

The given sequential code, based on Iterative Depending Depth First Search (IDS) approach, was used to implement and validate the parallel version using Ibis capabilities. IDA search method is a traversal graph strategy in which a depth-first search algorithm is run with a restriction on the traversal depth. Basically, we increase the allowed traversal depth until the solution/solutions is/are found in this method. [3].

The Rubik cube is solved by searching all the sates in a depth-limited version of traversal tree search, IDA method, recursively since there is no need to save a lot of states in recursive method. Each cube state represents a node in the tree, with edges representing possible moves in the cube, twists variable in the code. So, bound variable represents the smallest number of moves to reach to the goal from start.

A master-slave methodology is used in parallel version. Briefly, all the instances in the Ibis pool start to traverse the search tree and save the leaves. They stop to do so when a specific depth is reached in the search tree, in our implementation this depth is determined when twists value equals two. That is, we saved the leaves in the second level of the traversal tree, when twists value is 2, as our sub jobs. Because we experimentally reached to too coarse-grained jobs in the first level and too fine-grained jobs in the third level.

It should be that we named our method as a master-slave method, however, the master node, in the beginning, does not distribute the jobs among the clients. Generating and distributing the sub-jobs by server instance can decrease achievable performance, while the client nodes do nothing during this period. The main rule of master node here is receiving the results of solving the sub-tree by the other nodes and send them a fatal signal to kill themselves if someone can find at least a solution.

In the next phase, all the ibis instances get almost an equal number of sub jobs, leaves, for solving. That is, we applied static method to distribute the available jobs among the nodes. Because, every job needs the same amount of running instructions to traverse with a specific bound and DAS-4 is homogeneous environment. Then, each node sends their searching results to the server node. After receiving all the results from the workers by server node, it sends them a new bound, bound++, to all the workers if the amount of aggregated results is zero. Otherwise, server node sends a bound with a negative value, -1 in our program, to workers. Workers after receiving this negative value start to kill themselves and finish their running. At the end of algorithm, the server prints the aggregated final results. The communication between client and server is done by a defined class named "Communication" in the code.

In our bonus implementation to improve performance, each Ibis instance creates a new thread for each job to solve it. So, there are different threads in every node which they are running concurrently and traverse a different part of the search tree. This is done by a new class called "ClientThread". Therefore, we take advantage of multi-threading technique in a multi-node environment. Moreover, sever thread is a light weight thread which only manage the communications among the Ibis nodes and aggregate the found answers. Therefore, it is mostly under loaded. So, we also use the processing power of the server node and give it an equal number of jobs to solve them. Furthermore, to mitigate memory overhead and improve performance, we assign a new cache, using CubeCache class, to every thread.

Implicit and explicit multi-threading in Ibis Libraries

To improve performance, we tried to take advantage of multi-threading technique in the parallel code. To do this, we used two kinds of multi-threading methods which we called them **implicit multi-threading** and **explicit multi-threading.** In the former, one thread will be born for each received message by the Upcall port automatically. However, I should explicitly inherit from the "Thread" class to be able to create a thread using the latter method. So, in master/server node, there are two kinds of threads. One for sending new bound and receiving results which runs in the background using the Upcall method and the second thread type works in the foreground to traverse and solve the cube with new bound.

Performance analysis

Kernel part of each algorithm is a major part of execution time. In this algorithm, the major part is calculated by only server Ibis instance. Because it should aggregate the outcomes to be able to solve the cube. Therefore, server node is a proper option to find the longest execution time.

According to Amdahl's law, also known as strong scaling, the theoretical speedup in latency of the execution of a task *at fixed workload* that can be expected of a system can be formulated as below [2]:

$$S_{latency} = \frac{1}{f + \frac{(1-f)}{p}}$$

Where:

- $S_{latency}$ is the theoretical speedup of the execution of the whole task;
- f is serial fraction of code;
- p is the number of processors.

Therefore, Amdahl predicted that speedup will get to a maximum amount ($\frac{1}{f}$) and it gets fixed when the number of processors get infinite. Amdahl`s law is pessimistic because it is assumed that the parallelization is perfect. However, in practice, the situation is worse than predicted by Amdahl's Law due to: load unbalancing, scheduling, cost of communications, I/O, too fine-grained job, too coarse-grained jobs and so on. Moreover, a key assumption in Amdahl's law is that the fraction of sequential work is fixed, regardless of the size of the problem. But, in many practical problems, that assumption is not true. So, this law cannot give us a good prediction in most cases, however, it can give us a reasonable insight of parallelism.

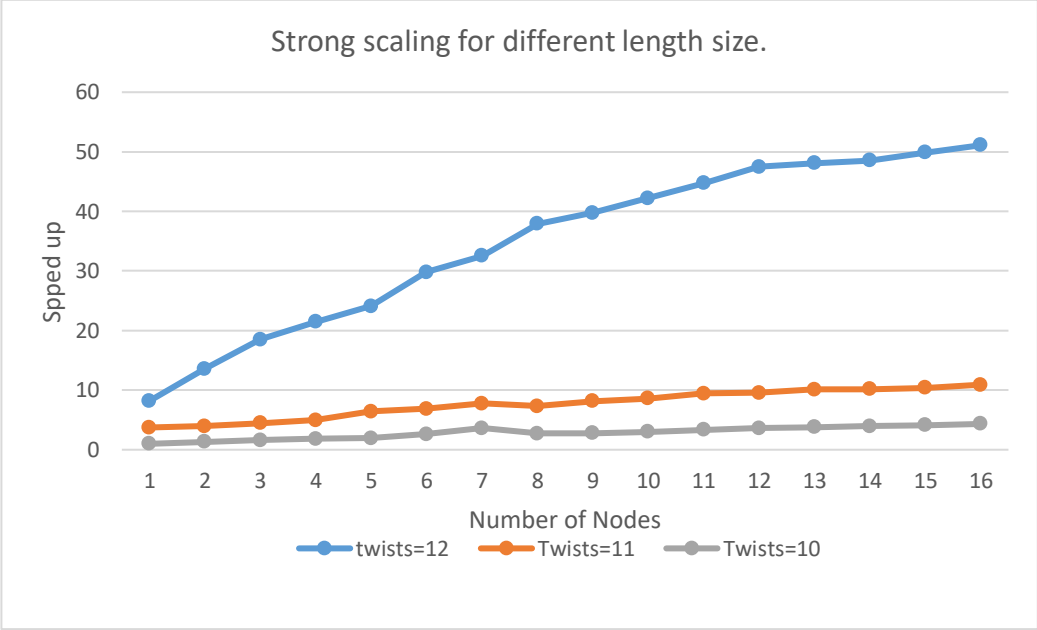

Fig 1. Speedup chart for different problem sizes with multi-threaded version

As it can be seen in figure 2, speed up is increased when the number of nodes raise and then it gradually gets constant. For instance, the achievable performance, when twists is 11, with more than 10 nodes is almost about 8. Interestingly, the achievable speedup with one node for the problem with twists 12 is about 8. This is due to that we are using multi-threading.

To make a comparison between IPL (single-thread) and multi-threaded versions, we depict the speedup chart in figure 2. In single-thread version, each client solves the sub-jobs by only one thread sequentially. It is clear that the achievable performance in multi-threaded version in more than the other one. For instance, the speedup, when the twist is 12, in multi-thread version is about 50, however, the corresponding figure in single-thread is about 14.

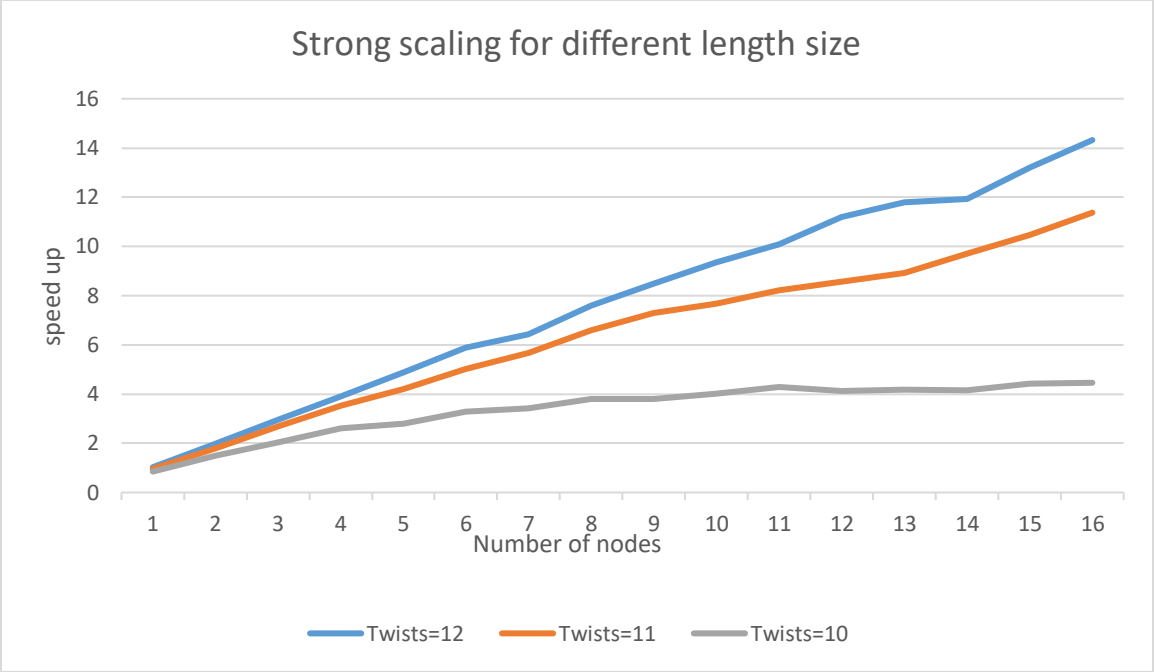

Fig 2. Speedup chart for different problem sizes with IPL (single-thread) version

Efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized. As the chart below is illustrated, the efficiency is gradually reduced when the number of processors is increased. However, it is possible to keep the efficiency in many parallel systems fixed by increasing both the size of the problem and the number of processing elements simultaneously, a system with this feature usually called scalable parallel system. Figure 3 shows the efficiency for multi-threaded implementation.

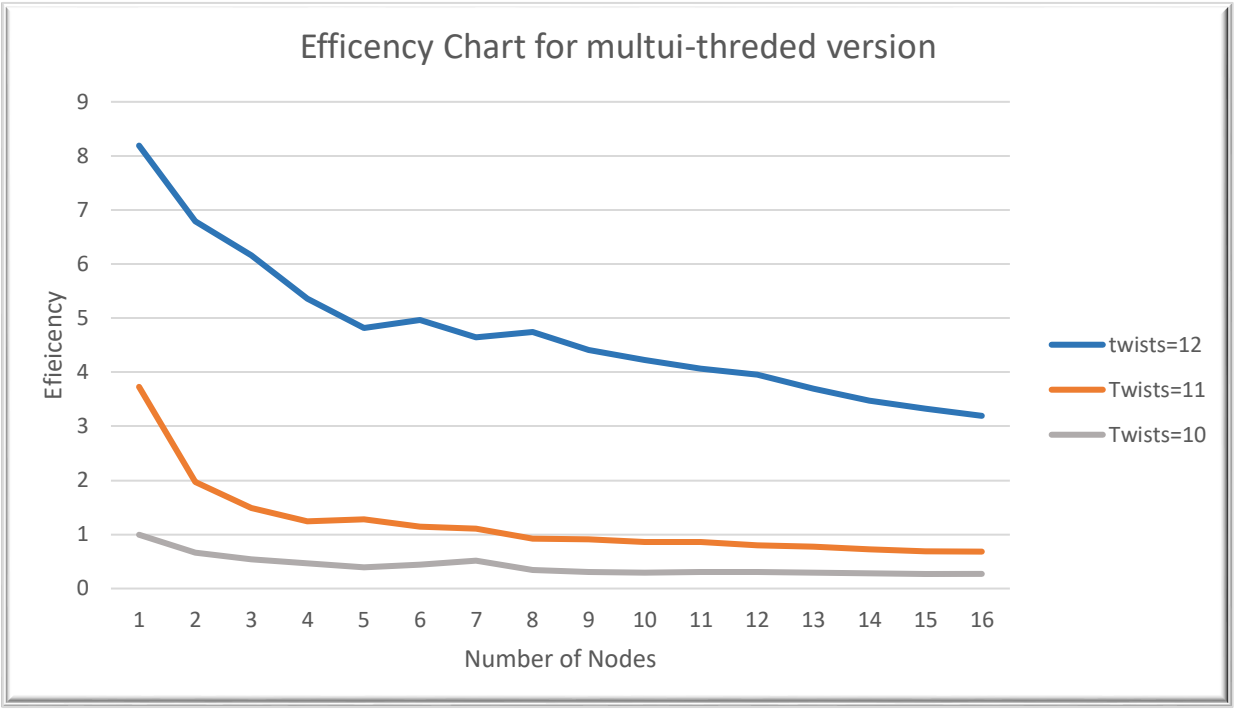

Fig 2. Efficiency of the parallel code with different twist sizes with multi-threaded version

References

1- *Wikipedia Encyclopedia*, https://simple.wikipedia.org/wiki/Rubik%27s_Cube.

2- *Wikipedia Encyclopedia*, https://en.wikipedia.org/wiki/Amdahl%27s_law, *visited at Nov 2018.*

3- *3. Wikipedia Encyclopedia https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search, visited at Jan 2019.*