JAVA Assignment: Rubik's Cube

Stefano Sandonà

Vrije Universiteit Amsterdam, Holland

1 The Rubik's Cube

A Rubik's Cube is a puzzle designed by Mr Rubik. In a Rubik's Cube, each of the faces is covered by colored stickers. In the initial setting, all the stickers of a face are of the same color and different colors are used for different faces. After performing some random twists, the aim of the game is to twist the cube until every side returns to be of a single color.

2 The Sequential Algorithm

The idea behind the sequential algorithm is simple. From a starting cube, we can twist it in each axis, row and direction (Figure 1). For each cube of size S, doing a twist, there are *6*(S-1)* new possible cubes. In Figure 2, the exolution of the search is represented as a tree. It's easy to understand from the figure that there is a rapid growth of the tree, in fact at the tree level i there are $(6*(S-1))^i$ nodes (cubes). The used state space search strategy, is the **deepening depth-first search (IDDFS)**. A depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches d, the depth of the shallowest goal state. When the bound is increased, the solution search starts again from the begin (the root), and continues until the bound level is reached. On each iteration, the algorithm visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first. The first 2 iterations of the algorithm, with the order of the cube's evaluation is shown in Figure 3.

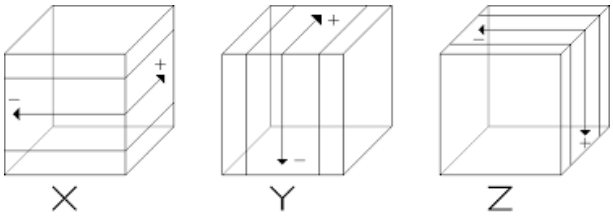


Figure 1: Possible directions and axis for twists in a 3d cube

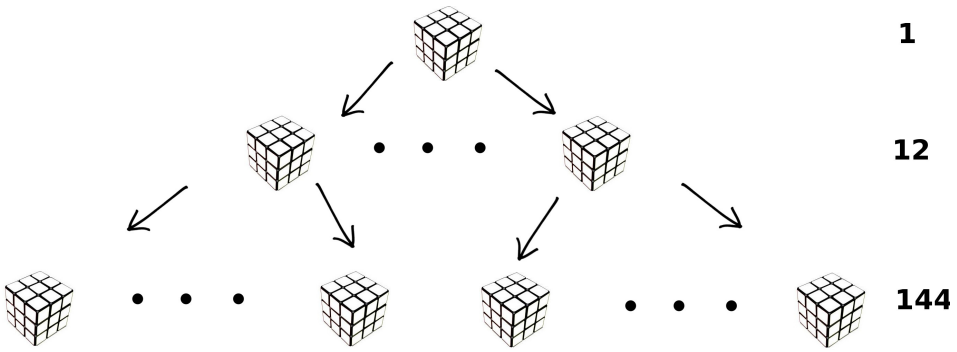
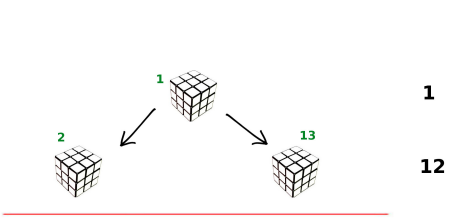
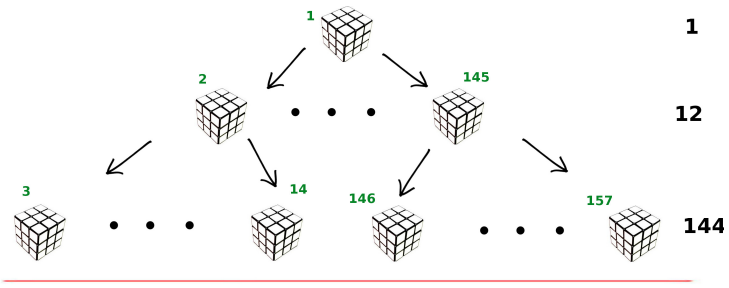


Figure 2: Rubik's Cube Solution Tree



(a) First itaration (bound=1)



(b) Second itaration (bound=2)

Figure 3: The IDDFS Algorithm

3 The Parallel Algorithm

An efficient parallelization of this algorithm is not trivial because of the work distribution. At each tree level, there are *(6*(cubeSize-1))*^{i} cubes, and this number is most likely not a multiple of the available nodes involved in the computation. Another aspect, consist on the organization of the entire work, due to the fact that after each tree bound level is reached, we have to decide if the work is finished or not. The next paragraph was introduced to clarify terms and concepts that will be used in the rest part of the report.

3.1 The Ibis Portability Layer (IPL)

For this project was used an efficient platform for distributed computing: Ibis. This is a Java-centric communication system designed for grid computing, developed at VU University, Amsterdam. A fundamental concept in Ibis is the Pool, that is a set of one or more Ibis instances, often distributed in different machines. Each Ibis instance, that joined a specific pool, has an unique id and can cooperate with the others, running a single distributed application. With the election mechanism of Ibis, one machine can be elected as "master" and so be the "coordinator" or the central point of the entire work. Thanks to the registry mechanism, each Ibis instance can also know all the other Ibises that joined the same Pool and communicate with them. The core of the Ibis system is the Ibis Portability layer (IPL), that is a communication library specifically designed for usage in a grid environment. It is based on setting up connections. The programmer can create send/receive ports (with a name) and connect them in a flexible way: one-to-one, one-to-many, many-to-one. In addition, the programmer can define properties of connections and ports like reliability, FIFO ordering, ... Proceeding in this way, after a port configuration, the only thing to do is to send a message through this port, the library will take care of the rest.

3.2 Cubes distribution

The work distribution, or better the cubes distribution, is the central part of the parallel implementation. The more balanced is the work among the involved nodes and the more will be the achieved performance. The major problem with a static partitioning, is that at each tree level, a number multiple of 6 of cubes are generated, and this is often not perfect divisible by the number of involved machines. A naive approach could be to generate the first levels of the tree and when there are enough nodes (more than the number of Ibis instances involved), distribute them as fairly as possible. However, this could lead to a big load imbalance and, as conseguece, not good performance. The basic idea of amplify the tree until there are enough nodes is correct, but the laod imbalance factor must be considered. After an accurate evaluation, a "weight" can be assigned to each cube, based on the amount of the next cubes that will be generated from it. This weight is represented in Equation1, where m is the tree level in which there will be the solution (unknown at the begin), n is *6*(cubeSize-1)* so the number of possible children per cube and z is the actual tree level of the cube.

$$Weight = \sum_{i=0}^{m-z} n^i \tag{1}$$

As clarification, a practical example is reported in Figure 4. In this example, that is not related to the Rubik's Cube, but just to explain the general idea, *n=2*, *z1=1*, *z2=2*, *m=4*. Applying the previous formula, the weight of the first cube (at z1 level) is $2^0+2^1+2^2+2^3=15$, because from it 14 cubes will be generated, so 15 (counting also the root node) will be evaluated in total. The weight of the second cube (at z2 level), is $2^0+2^1+2^2=7$.

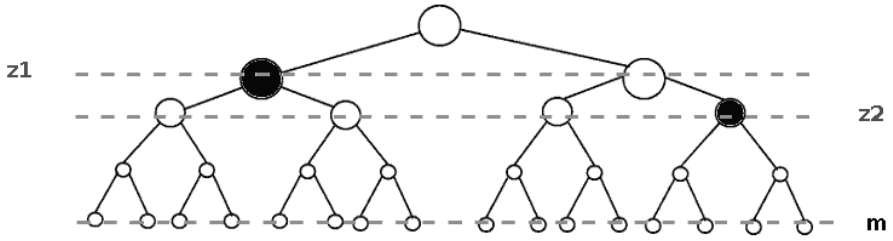


Figure 4: Rubik's Cube Solution Tree

Give a weight to a cube is useful, because it should be correct to assign to each Ibis instance more or less the same work to do. Given the fact that the number of cubes on a tree level often is not a multiple of the nodes involved on the computation, there will be load imbalance at a certain point. Also distributing the nodes at a specific tree level as fairly as possible, there will be one or more Ibis instances with one more cube than the others, for a gap of maximum one cube. The only thing to do, is to select the correct tree level in which have this load imbalance, in order to limit the deriving damages. Looking at the previous formula, at the begin of the computation n is known because the size of the cube is known, m is unknown, is part of the solution, and z is the value that we want to find. As just said, given enough cubes (more than the number of Ibis instances), the maximum difference of assigned cubes is one cube. In fact, this is possible using a simple *for* construct, shown here:

```
int sum = 0;
int rem = numCubes % numProcs;
for (i = 0; i < numprocs; i++) {
    cubes_per_proc[i] = numCubes / numProcs;
    if (rem > 0) {
        cubes_per_proc[i]++;
        rem--;
    }
    displs[i] = sum;
    sum += cubes_per_proc[i];
}
```

The aim of this work is to reduce the tree nodes unroll and the load imbalance at the same time. At a certain tree level z, there are n^z cubes and the relative weight per cube is the one shown in Equation1. With the previous code, calling k the number of machines involved in the computation, the lowest amount of work assigned per slave, is shown in Equation 2, while the highest is shown in Equation 3.

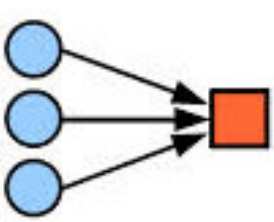
$$LW = \sum_{i=0}^{m-z} n^i * \lfloor \frac{n^z}{k} \rfloor \tag{2}$$

$$HW = \sum_{i=0}^{m-z} n^i * \lfloor \frac{n^z}{k} \rfloor + \sum_{i=0}^{m-z} n^i \tag{3}$$

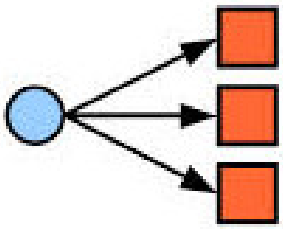
To reduce the load imbalance as much as possible, the two amounts of work should differ as little as possible. That means that *HW*100/LW* has to be as near to 100 as possible, for example lower than 101. Solving the inequality shown in Equation 4, the solution is the one shown in Equation 5.

$$\frac{(\sum_{i=0}^{m-z} n^i * \lfloor \frac{n^z}{k} \rfloor + \sum_{i=0}^{m-z} n^i) * 100}{(\sum_{i=0}^{m-z} n^i * \lfloor \frac{n^z}{k} \rfloor)} < 101 \tag{4}$$

$$\lfloor \frac{n^z}{k} \rfloor > 100 \tag{5}$$



(a) Results communication - Many to 1



(b) Work status communication - 1 to Many

Figure 5: Send (circle) and Receive (square) Ports organization

3.3 Ibis setting

Each node involved in the computation is an Ibis instance with an unique identifier. When a tree level bound is reached, each Ibis instance has to communicate its results (if a solution is founded and, in the case, how many) and has to decide if continue with the next bound or not. The most obvious solution, is to adopt a "master slave" approach and use the "master" instance to collect the partial results and decide if the work is finished or continue with the next bound. To do this 4 ports are needed. On the master side 2 ports are needed, one to receive the results from the slaves and one to send if the the work is finisched. Specularly, on the slaves side, 2 ports are needed, one to communicate the solutions found and one to receive if the if the next bound has to be evaluated or the work is finished. A graphic representation is presented in Figure 5.