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 Ernő Rubik. In a Rubik's Cube, each of the faces is covered by colored stickers and 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 goal of the game is to twist the cube until every face returns to be of a single color.

2 The Sequential Algorithm

From a starting Rubik's cube, we can twist it in each row, axis and direction (Figure 1) and from each cube of size S, after a twist, there are *6*(S-1)* new possible configurations. The evolution of the search can be represented as as a tree. The idea behind the given sequential algorithm to solve the Rubik's Cube is simple. 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 m, 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 2. At the tree level i there are $(6*(S-1))^i$ nodes (cubes), so there is a rapid growth of the search tree. The time complexity of IDDFS is $O(b^m)$ and its space complexity is O(m), where b is the branching factor (*6*(S-1)*) and m is the depth of the shallowest goal.

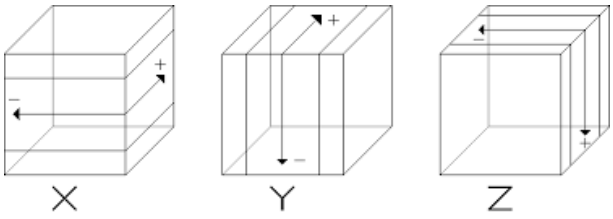
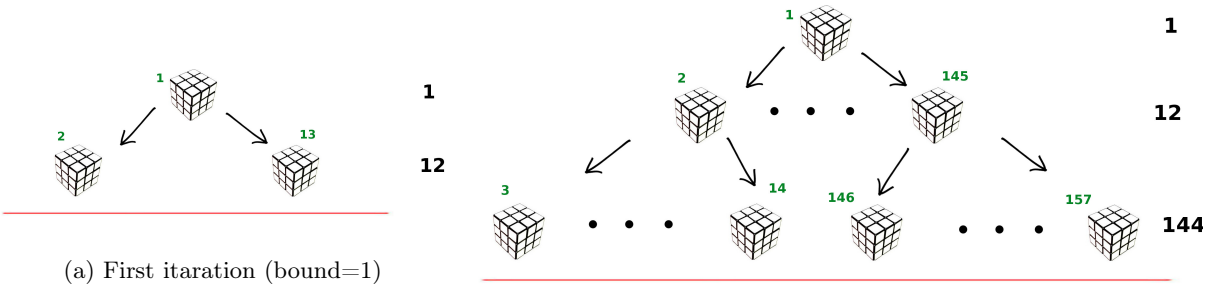


Figure 1: Possible axis and directions for the twists in a 3d cube



(a) First itaration (bound=1)

(b) Second itaration (bound=2)

Figure 2: The IDDFS Algorithm (3d cube)

3 The Parallel Algorithm

An efficient parallelization of this algorithm is not trivial because of the work distribution. At each tree level i, there are $(6*(cubeSize-1))^i$ cubes, and this number is most likely not a multiple of the available machines involved in the computation. Another aspect, consists on the organization of the entire work, due to the fact that after a tree bound level is reached, the termination or the progress of the work has to be checked. The next paragraph was introduced to clarify terms and concepts that will be used in the rest 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 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 applying a certain configuration to a port, 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 is generated, and this is often not perfectly divisible by the number of machines involved. A naive approach could generate the first levels of the tree until there are enough nodes (more than the number of Ibis instances involved) and distribute them as fairly as possible. However, this could lead to a big load imbalance and, as a consequence, to have not good performance. For example, with a 3d cube and 5 machines, the tree will be unrolled until level 1. The 12 resulting cubes would be divided into 3 3 2 2 2 and assigned to the 5 machines. This means that the first 2 machines would work 1.5 times the others. To outline the gravity of this scenario, if m (the tree level in which lives the solution) is big enough, it could be possible that the last 3 machines worked for an entire day (24 hours), while the first two machines worked for one day and a half (36 hours). The total computational time would be 36 hours and for 12 hours the last 3 machines would be idle. The basic idea of amplifying the tree until there are enough nodes is correct and logical, but the load imbalance factor must be considered.

3.2.1 Cube Weight

To avoid the load imbalance, a way to measure the total amount of work assigned to a node is needed. To do this, to each cube could be assigned a "weight", 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)* (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 3. In this scenario, that is not related to the Rubik's Cube, but is presented just to explain the general idea, *n=2*, *z1=1*, *z2=2*, *m=4*. Applying the previous formula, the weight of the black node at *z1* level is $2^0+2^1+2^2+2^3=15$, because from it 14 other nodes will be generated and 15 (counting also the starting node) will be evaluated in total. The weight of the black node at *z2* level, is $2^0+2^1+2^2=7$.

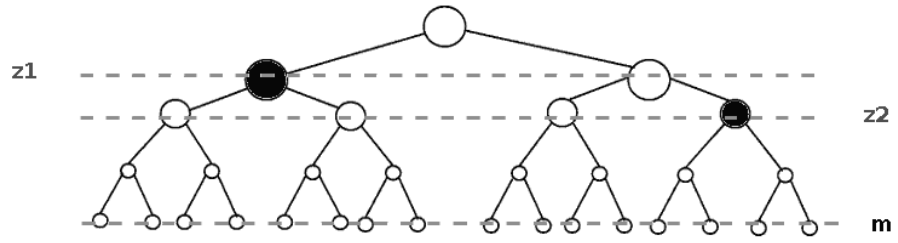


Figure 3: Rubik's Cube Solution Tree

3.2.2 Load balancing

To have a fair distribution of the work, it is correct to assign to each Ibis instance more or less the same cube weight. Given the fact that the number of cubes on a tree level often is not a multiple of the nodes involved in 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 additional cube, for a difference of maximum one cube (e.g. with 9 cubes and 4 machines 3 2 2 2, with 17 cubes and 3 machines 6 6 5). Accepting this, the best 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 Equation1, at the begin of the computation n is known because it is the size of the cube, 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 per node is one. This is possible using a simple *for* construct (Listing 1).

```
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];
}
```

Listing 1: fair distribution of cubes

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 (Listing 1), 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 (1% of load imbalance permitted). Solving the inequality shown in Equation 4, the solution is the one shown in Equation 5.

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

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

At the beginning n (the size of the cube) is known and also k (the number of Ibises that joined the pool), so the value to find is z. In addition, to reduce the load imbalance, it's useful to reduce the initial tree unroll at the minimum. For this reason, the final value to find is the one shown in Equation 6.

$$\min_{z>0} \lfloor \frac{n^z}{k} \rfloor > 100 \tag{6}$$

3.2.3 Work split

Once found the correct value of z, this is the tree level in which to have an imbalance of 1 cube, isn't such a big risk for the performance. To reduce the initial unroll, the tree is extended (A) until the level *z-1*, if at that level there are enough nodes (more than the Ibis instances), otherwise (B) until the level z. If we are in case A, the $n^{(z\text{-}1)}$ resulting nodes are divided into equal parts to the machines ($n^{(z\text{-}1)}/k$ cubes per machine). If some nodes have left out (k is not a divisor of the number of nodes of this level), these are expanded to the next tree level (Figure 4), otherwise the splitting phase is terminated (Figure 5). If we have expanded some nodes, we try to split their children among the Ibis instances. Until they are less than the number of ibis instances we generate another level of the tree from them (Figure 6). When they are enough, we split them as fairly as possible (using the approach shown in Listing 1). If we are in case B we just split the $n^{(z)}$ nodes as fairly as possible (Listing 1). In Figure 4, Table 1 there is an example of work riparitition among 11 Ibis instances, in Figure 5, Table 2 among 16 Ibis instances and in Figure 6, Table 3 among 13 Ibis instances.

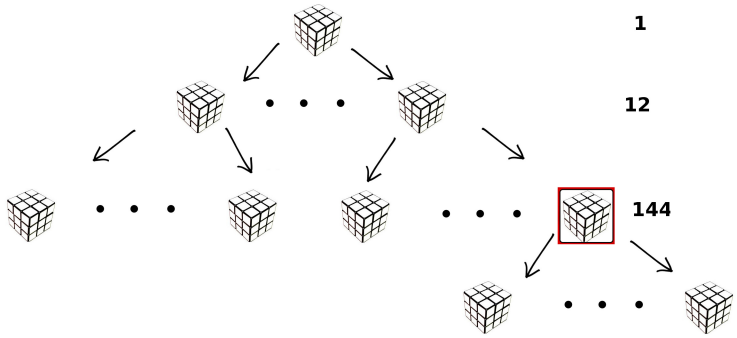


Figure 4: Cubes distribution - Ibis instances=11, Size=3, Twists=11, Bound=7

Ibis Instance	0	1	2	3	4	5	6	7	8	9	10	Total
Cubes of level 2	13	13	13	13	13	13	13	13	13	13	13	143
Cubes of level 3	2	1	1	1	1	1	1	1	1	1	1	12

Table 1: Cubes distribution - Ibis instances=11, Size=3, Twists=11, Bound=7

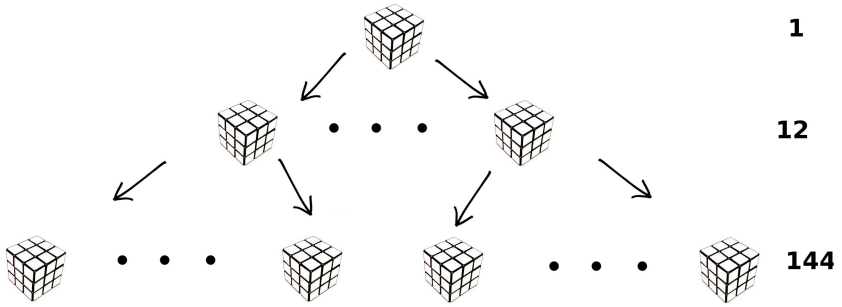


Figure 5: Cubes distribution - Ibis instances=16, Size=3, Twists=11, Bound=7

Ibis Instance	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
Cubes of level 2	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	144

Table 2: Cubes distribution - Ibis instances=16, Size=3, Twists=11, Bound=7

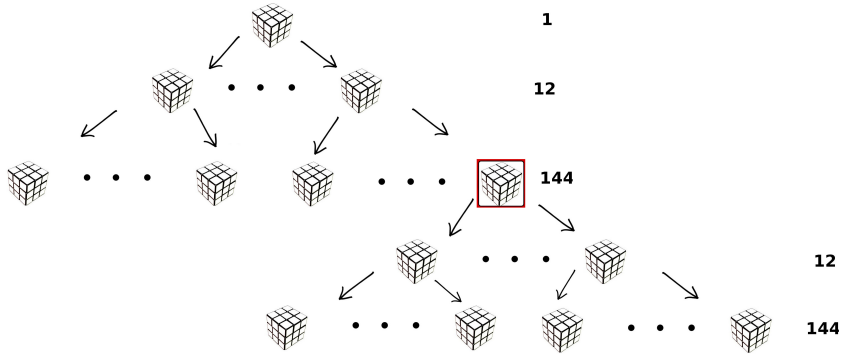


Figure 6: Cubes distribution - Ibis instances=13, Size=3, Twists=11, Bound=7

Ibis Instance	0	1	2	3	4	5	6	7	8	9	10	11	12	Total
Cubes of level 2	11	11	11	11	11	11	11	11	11	11	11	11	11	143
Cubes of level 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cubes of level 4	12	11	11	11	11	11	11	11	11	11	11	11	11	144

Table 3: Cubes distribution - Ibis instances=13, Size=3, Twists=11, Bound=7

After the cubes of the first levels are correclty splitted, each Ibis instance will have a work queue and for each cube inside it, the algorithm will proceed like in the sequential version. In Table 4, are reported the numbers of cubes evaluated by each Ibis instance, when the algorithm's bound is 7, with a Cube of size 3. These numbers were measured running the implementation with different number of Ibises. The Expanded Cubes, are the ones from which are generated other cubes during the initial splitting work phase while the Assigned Cubes are the ones effectively evaluated. As shown, for every launching configuration, the load imbalance is always under the 1%.

Table 4: Cubes distribution - Size=3, Twists=11, Bound=7

Ibis Instance	Assigned Cubes	Expanded Cubes	Total Cubes	Load Imbalance
2	19544622	1	39089245	0%
4	9772311	1	39089245	0%
5	7804273(2), 7826894(3)	17	39089245	0,28%
8	4886154	13	39089245	0%
11	3551510(10), 3574131(1)	14	39089245	0.63%
13	3006718(12), 3008603(1)	26	39089245	0.06%

3.2.4 Special cases

The weakest part of this implementation, is the tree unrolling phase at the beginning of each bound increment, in order to better distribute the work. During this step, it could be possible to find the problem solution if the cube was previously twisted for few times. However, that would mean a small searching space, and so a non convenience on solving the cube in parallel. In the case a tree level is completely generated during the initial unrolling phase and a solution is found, the current algorithm will stop, and the master Ibis will show the problem solution, without performing any extra communication with the slaves. Another case that has to be considered is the one in which performing the second tree extension with the cubes left out by the distribution (in the Figure 4 the cube surrounded by the red square), these are solved cubes. The algorithm, correctly treat also this scenario, storing the partial results found at each level of the unrolling and adding these to the tree level results found during the next solving phase.

3.3 Solution search and Ibises coordination

To avoid the sending of the cubes at the beginning, each Ibis instance generates the same initial cube, performs the splitting phase and assigns certain cubes to itself. To reach the final solution, the machines involved in the computation have to coordinate and cooperate. When a tree level bound is reached, the distributed results have to be summed, in order to decide if continue with the next bound or not. To do that, a "master slave" approach is adopted. The "master" Ibis instance collects and sums the partial results and decides if the work is finished or is necessary to continue with the next bound. For what concerns the communication setting, 4 ports are used. On the master side 2 ports are needed, one to receive the results from the slaves and one to communicate them if the work is finisched. Specularly, on the slaves side, 2 ports are needed, one to communicate the solution found and one to know if the next bound has to be evaluated or the work is finished. A graphic representation of the ports organization is presented in Figure 7.

(a) Results communication - Many (Slaves) to 1 (Master)

(b) Termination communication - 1 (Master) to Many (Slaves)

Figure 7: Send Ports (circles) and Receive Ports (squares) Organization

3.4 Results

The realized parallel implementation obtained very good results. As shown in Figure 8 and Tables 5, 6, the speedups are almost linear for each configuration. The worst measured case is the configuration with 13 machines and cube size 4. In this situation, the tree is extended until level 2, the 324 resulting nodes are split into equal parts between the 13 Ibises, than the 12 cubes left out (324mod13=12) are further extended, obtaining 216 additional nodes, that are distributed as fairly as possible. Another interesting aspect is that in some cases, the speedup is superlinear (Figure 8, 12 twists, 3 machines), due to the more

cache and memory resources available on parallel systems. In a single machine, in fact, if the memory is not enough there will be a lot of swapping and trashing, actions that are reduced having bigger available resources.

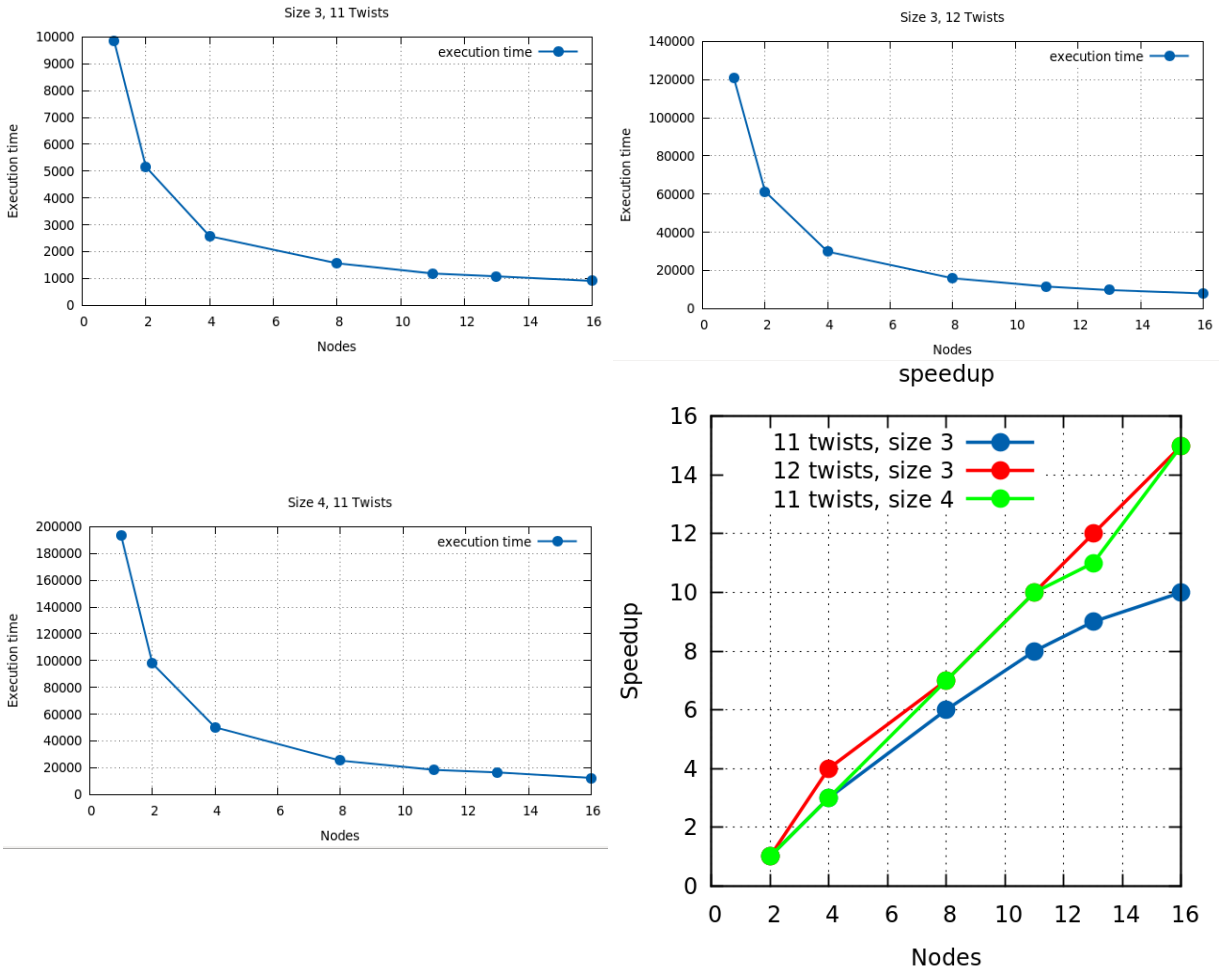


Figure 8: Execution Time and Speedups

nodes\configuration	11 Twists (size 3)	12 Twists (size 3)	11 Twists (size 4)
1	9863	121067	193453
2	5155	61350	97769
4	2576	29841	50208
8	1565	15899	25313
11	1185	11547	18391
13	1081	9710	16512
16	909	7931	12375

Table 5: Execution Times

nodes\configuration	11 Twists (size 3)	12 Twists (size 3)	11 Twists (size 4)
2	1,91	1,97	1,97
4	3,82	4,05	3,85
8	6,30	7,61	7,64
11	8,32	10,48	10,51
13	9,12	12,46	11,71
16	10,85	15,26	15,63

Table 6: Speedups

3.5 Final considerations

The adopted solution is applicable to each number of machines paying the cost of the initial work distribution phase. The more is high the number of the Ibises that will join the pool and the more tree levels will be generated in order to balance the work. The powerful and the flexibility of the Ibis system were really helpful during the solution developement, hiding all the "problems" deriving from a distributed application like the communiation.

3.6 Thread optimization

The program was optimized using threads to make use of the multicores in each machine. Each machine owns a dual-cpu quad core, so there are 8 cores in total. The algorithm is the same as the previous but instead of find the best level until expand the tree, it directly expands it until the 2nd level, so it is sure that there are enough cubes to split. The cubes of this level that are assigned to a machine, then are divided among 24 threads (3 times the number of cores). This number was found after some experiments. In fact, due to the fact that there are 8 cores, and that the algorithm is compute bound, launching too many threads has no sense. Table 7 shows the obtained execution times. Table 8 shows the obtained speedups.

nodes\configuration	**11 Twists (size 3)**	**12 Twists (size 3)**	**11 Twists (size 4)**
2	1024	7018	15591
4	892	4110	6517
8	882	2384	4089
16	518	1784	2680

Table 7: Execution times

nodes\configuration	**11 Twists (size 3)**	**12 Twists (size 3)**	**11 Twists (size 4)**
2	9,63	17,25	12,40
4	11,05	29,45	29,68
8	11,18	50,78	47,31
16	19,04	67,86	72,18

Table 8: Speedups