Parallel Programming Practical

Java Assignment - Distributed Rubik's cube solver

Laurens Verspeek
VUnetID: 2562508

February 3, 2016

1 Introduction

This report discusses an implementation of a distributed Rubik's cube solver that it is capable of running on multiple (DAS-4) machines in parallel and compares it with the sequential version of the solver. To communicate between the different machines, the Ibis Portability Layer (IPL) is used. A work queue located on the master node is used to handle passing work to the worker machines. The setup and design overview of the implementation is discussed in section *Implementation*. The speedup performance compared to the sequential algorithm is discussed in section *Results*.

2 Implementation

The distributed Rubik's cube solver is a parallel implementation of a sequentional IDA* Puzzle solver algorithm for the Rubik's cube. The sequential algorithm will not be explained in detail here as that is not the focus of this report. This report describes the parallelisation of the sequential algorithm with IPL. However to understand the parallel version, the sequential algorithm is discussed shortly. After that the parallel implementation will be discussed.

2.1 Sequential algorithm

The sequential algorithm to solve the cube was already provided in the assignment. A Rubiks cube was created randomly or from a file, which is useful for testing as you can test on the same starting cube for every run. The size and the number of random twists of the cube can be changed. This generated cube is then passed to the sequential algorithm, which start by setting the bound to 1 (which means that only 1 twist is allowed) and creates all children from the starting cube by twisting the cube 1 time in each direction. When none of the children cubes are solved, the bound gets limited to two and the algorithm starts all over, but now also recursively creating the grandchildren of the original cube. The bound gets increased until a solution is found.

2.2 Parallel algortihm

The sequential algorithm could easily be improved, as it is not a very smart algorithm. However, we were not allowed to change the sequential algorithm other than making it parallel. Luckily, this 'no so smart' sequential algorithm is embarrassingly parallel. We start by electing one master from all the available nodes. The other nodes are worker nodes. The master starts by creating all children from the first cube and put them in a job queue.

However, this only puts 12 jobs in the queue, which is already insufficient if you for example use 16 nodes. To increase the amount of jobs in the queue, the grandchildren of the original cube are also stored in the queue if the bound is larger than 1 (otherwise there are no grandchildren). This will give all the nodes sufficient jobs, so that there are

no idle nodes. The workers ask the master for jobs with an upcall and once the queue is ready the master sends a job (cube) to the worker. If it is the first time a worker makes an upcall to the master, the master creates a connection back to the worker and stores this connection in a hashmap. Once the worker is done with his first job and asks the master for another job, the master can send the job through the previously stored connection in the hashmap. This way the master does not need to setup a new connection for each upcall, only for each worker.

As multiple workers can request a job at the same time from the master, the jobQueue and the removal of jobs from that queue are synchronized. A lock on an object is used to make the workers wait for the jobQueue to be ready. The workers then recursively solve their cubes and the master also helps with solving cubes. Once the workers are done, they ask for a new job from the master and in the same request they report the results of their previous job. The master regenerates the queue for each bound. To make sure workers don't increase the solutions variable at the same time and to keep the variable that keeps track of the number of active workers correct, Atomic Integers are used as type for these variables, so they can only be accessed by one thread at the time (as each upcall creates a new thread in the master). Once the queue is empty and all the workers reported back to the master, the master checks if the Atomic Integer variable solutions is greater than 0. If it is, it reports this back to the user, otherwise it rebuilds the queue and increments the bound by one.

3 Results

The sequential and the parallel algorithms were tested on the DAS-4 system. All the cubes are generated with the same seed, so they have the same degree of difficulty to solve. The parallel algorithm was tested with 1, 2, 4, 8 and 16 nodes. The timers provided by Java (System.currentTimeMillis) were used to measure the execution time. The benchmarks are run on three different types of cubes: one with size 3 and 11 random twists, one with size 3x3x3 and 12 random twists and one cube with size 4x4x4 and 11 twists. All the results can be found in table 1. All the results in that table are the average of 3 measurements.

To make these results more visual, the speedups are displayed in the following 3 graphs. Here we can see that the IPL is almost just as good as the optimal linear speedup that can be achieved. However, it is a bit slower as it has to build a queue with jobs and the master needs to setup connections with the workers. The more nodes there are, the more connections the master needs to setup and the more communication it requires between the master and the workers to request jobs and send the results back. This can clearly be seen from the three graphs, as the speedup varies more from the optimal speedup when the number of nodes increase.

Cube properties	**Sequential**	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**
size=3; 11 twists	12.505s	12.503s	6.668s	3.112s	1.593s	0.909s
size=3; 12 twists	137.298s	147.489s	62.983s	33119s	17.924s	9.375s
size=4; 11 twists	196.192s	209.180s	109.416s	56.472s	27.884s	14.130s

Table 1: Run times of different cubes with different number of nodes

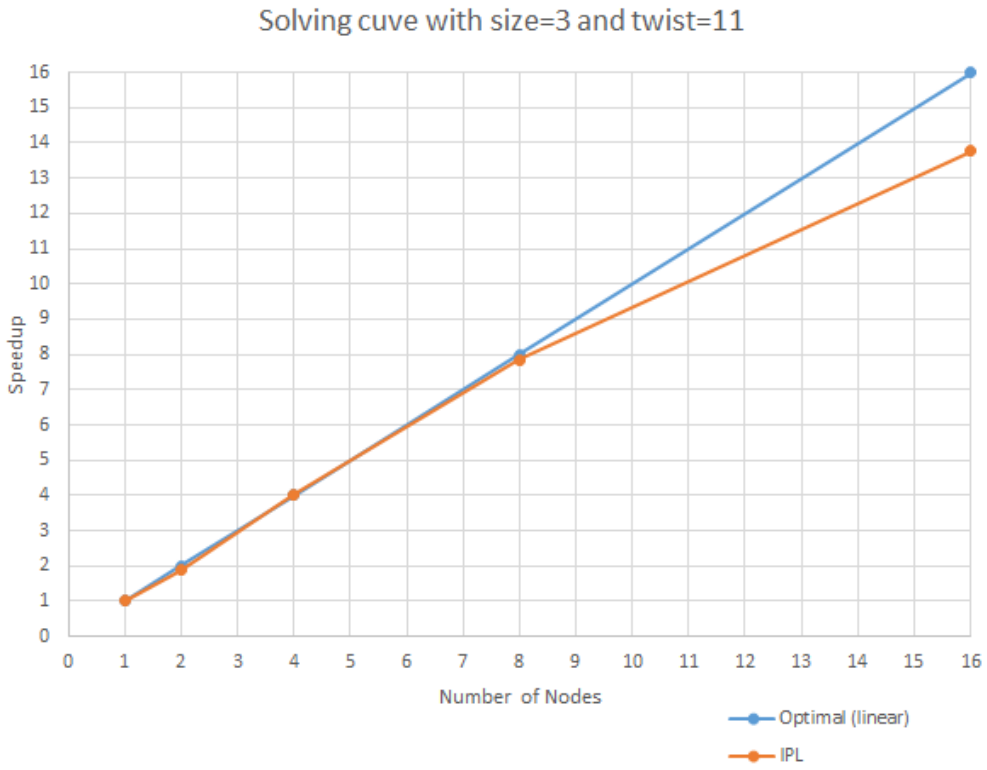

Figure 1: Speedup of the parallel implementation with different number of nodes compared to the sequential algorithm

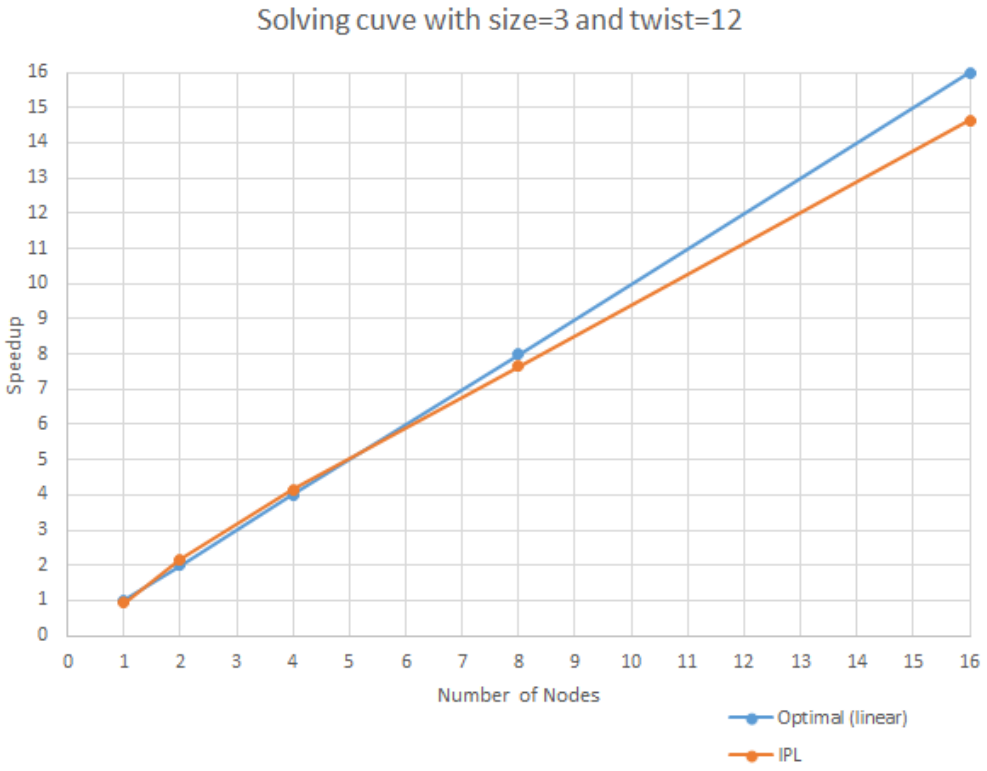

Figure 2: Speedup of the parallel implementation with different number of nodes compared to the sequential algorithm

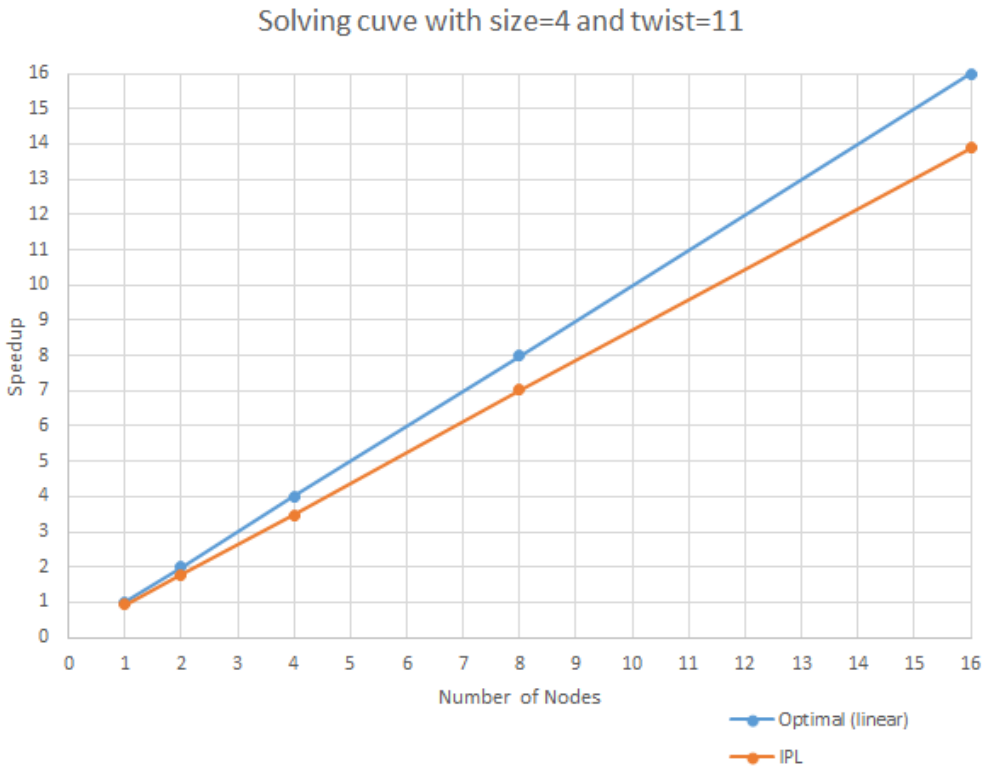

Figure 3: Speedup of the parallel implementation with different number of nodes compared to the sequential algorithm