Vrije Universiteit Amsterdam

Parallel Programming Practical
IPL: Rubiks Cube Solver

Christopher Esterhuyse
2553295 – cee600
September 12, 2019

1 Introduction & Background

Since its invention in 1974, the *Rubik's Cube* puzzle has left a lasting impression on the world. A cube in *solved* state presents a grid of $N \times N$ stickers of uniform color per face[1], with each face a different color. The cube affords *twisting* along axes of rotation in-between the grid tiles. This scrambles the colored stickers. Assuming the original twisting pattern is obfuscated (realistically, this happens very quickly) the solver attempts to apply twists until the cube is again *solved*. This astoundingly simple combinatory puzzle yields devilish implications for human solvers to wrap their heads around its complexity. The systemic solution of arbitrary Rubik's cubes has become idiomatic to many fields of computer science as a result of these properties.

Solving a cube can be approached systematically using several methods. The A* search algorithm is a best-first search. Starting from the initial state, possible transitions are considered recursively to determine further reachable states, where each transition represents some atomic *step* toward solving the problem, for which *fewer* steps are associated with higher *quality* solutions. Indeed this notion of 'quality' is necessary for the algorithm, defining the *order* in which traversal options are prioritized in a hope to minimize search time. In our case, we do not consider any interesting *bounding* techniques, and we wish to find all best solutions of *equivalent* quality, where a solution's quality is given by the number of axial rotations necessary. To this end, we employ IDA* in particular ('iteratively-deepening A*'). Concretely, the search tree is constructed depth-first up to depth 1, exploring transitions in arbitrary order. If no solutions are found, the tree is discarded and search begins again up to depth 2. This continues until a

[1]The original Rubik's Cube had $n = 3$ but variations exist for other sizes. In this work, we generalize N

tree contains at least one solution. The nature of the algorithm precludes the discovery of sub-optimal solutions, and guarantees that no solution of equivalent quality is overlooked. At first glace, iterative-deepening appears wasteful, with each iteration re-doing work of previous steps. In practice, this cost is nominal, as even modest-sized Rubik's Cubes have considerable *branching factor*; The work at depth D (written '$work(D)$') vastly outweighs $\sum_{i=0}^{D-1} work(i)$ for arbitrary D.

The Ibis Portability Layer ('IPL') library provides a distributed message-passing service for otherwise-isolated Java processes. Each Java process imports the IPL library and executes methods that result in message-passing at runtime via some IPL Server instance.

This report details the design, development and evaluation of a IPL-Java implementation of an IDA* Rubik's Cube solver. The Vrije Universiteir's in-house computing cluster 'DAS-4' provides the infrastructure necessary to support a parallel and distributed program. We focus on maximizing the effective utilization of these resources to achieve maximal speedup over sequential benchmarks of the same solver.

2 Design & Implementation

This section describes the design for the distributed IPL-Java Rubik's Cube solver. Sections incrimentally elaborate on the design starting from the sequential IDA* reference implementation. The final version is then used for experimentation in §3 and beyond.

2.1 Sequential Reference Implementation

As outlined in §1, the solution of an arbitrary Rubiks Cube can be fundamentally considered a search problem of the tree rooted in the initial cube, with branches representing individual *twists* yielding new cube states. The canonical IDA* algorithm explores the branches depth-first to a set bound. If no solution is found, the bound is incremented and the search begins again in a new *round*. The solution is returned as the *count* of paths to solution leaves for the first round for which the count is positive.

The precise encoding of the state of a cube and the definition of a *state transition* is relevant to the definition of the *quality* of the solutions found. Different models may represent the same 'logical' solution as reachable in a different minimal number of steps. In our case, a cube with size of $N \times N$ has $N - 1$ 'splits' between stickered segments may *twist* in any of its 3 dimensions, in any split, in either clockwise or anti-clockwise direction, yielding a branching factor of $2 \times 3(N - 1)$. This representation models the natural intuition of gripping the Rubik's Cube in both hands and rotating them antagonistically, articulating a *twist* of ¼ rotation at precisely one split.

2.2 Master-Slave Concurrency Model

The distributed implementation makes use of a concurrency model where a singular *master* creates work tasks and distributes them to *slaves* that listen for tasks, compute them as they come in and report findings to the master. This section explores details of this approach and how they manifest in this implementation.

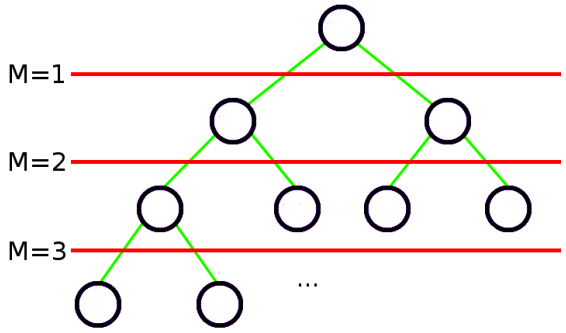

Figure 2.1: An illustration of how the choice of parameter M determines the division between work done by masters and slaves for some problem with branching factor 2. For each choice for M, all nodes above the line are visited by the master, with edges crossing the threshold mapping to created sub-problem instances for slaves.

2.2.1 Work Partitioning

Work for this problem manifests as traversal of a search tree. As such, work partitioning coincides with nodes delegating sub-trees to one another. Our approach has the master initiating the search and exploring to some M (for '(M)aster search depth') and delegating all nodes at this depth as sub-problems to slaves. The precise value of M thus manifests as a parameter, the choice of which determines the granularity of slave tasks. The trivial depth of $M = 0$ causes a degeneration to a singleton sub-problem to one slave, given the root node itself. At the other extreme, $M = \infty$ results in the master never delegating any sub-tasks, ultimately finding the solutions itself (assuming they exist to be found). Figure 2.1 illustrates the division of work in response to the choice of M.

2.2.2 Message Protocol

The implementation follows the model described in §2.2.1. The network topology is conceptually centralized at the master[2]. Communication occurs as either $Master \rightarrow Slave$ or $Slave \rightarrow Master$. Below all necessary messages in the communication protocol of the IPL-Java implementation are enumerated. Note that m and s are abbreviations for 'Master' and 'Slave' respectively:

Message	From	To	Meaning	Representation
$\langle hello \rangle$	s	m	s is ready for work.	*null* reference
$\langle done \rangle$	m	s	s should terminate.	*null* reference
$\langle solve, C \rangle$	m	s	s should solve sub-problem cube C and reply the result.	*non-null* serialized Cube object.
$\langle result, R \rangle$	s	m	s has solved a sub-problem and found R solutions.	*non-null* serialized Integer object.

[2]Owing to the use of IPL, the network is physically centralized at the IPL Server rather than the Master.

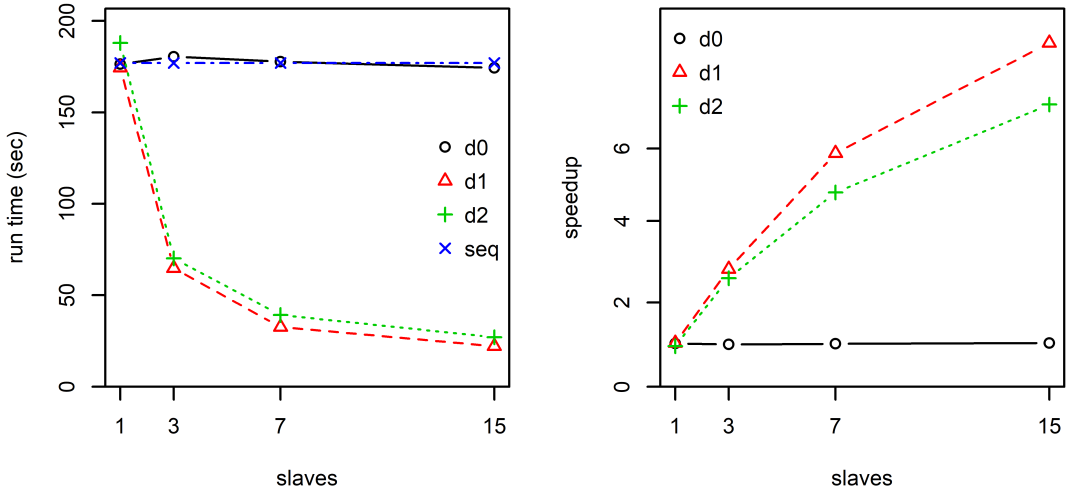

Figure 2.2: Run time and speedup of a preliminary version of the IPL-Java implementation using transient connections for a cube of size 3. Each line indicates results for a different choice for the parameter M. The speedup plot shows a horizontal bar at 1 to indicate the sequential reference.

At the implementation-level, only two kinds of IPL *PortType* are required, one for each direction of communication. As can be seen in the table, $\langle hello \rangle$ and $\langle result, R \rangle$ may be transmitted over the same links, as they share a type and their contents are distinguishable. The same holds for $\langle done \rangle$ and $\langle solve, C \rangle$ in the opposite direction.

2.2.3 Connection Persistence

IPL's 'reliable' connection capability was selected to greatly simplify the communication needed in our implementation. Communication between the master and slaves are sparse and sporadic. Both the upkeep of an existing connection and the setup of a new one requires work. As such, it was initially uncertain which form of connection would prove more suitable. Early experiments were performed on an implementation with *transient* connections, necessitating a connection and disconnection event for each message transmission, but requiring no upkeep. Runs yielded results in figures 2.2 and 2.3 for a cube of size 3 and 4 respectively[3]. In either case, we see that the optimal choice of parameter M was 1, a very low value. Larger values incurred too much overhead for larger M, as more individual connections were created and destroyed per session. Thus, *persistent* connections were used for the final implementation instead, where each slave is connected to the master at initialization and the same connections are maintained for the duration of the program. The rest of the report uses this version of the implementation.

[3]The experimental setup for these runs is identical to that of those in §3.

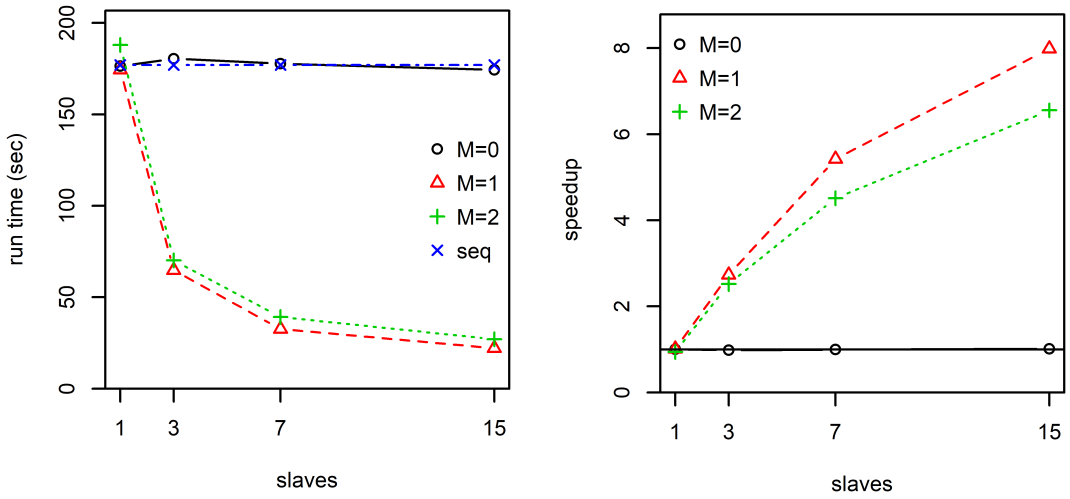

Figure 2.3: Run time and speedup of a preliminary version of the IPL-Java implementation using transient connections for a cube of size 4. Each line indicates results for a different choice for the parameter M. The speedup plot shows a horizontal bar at 1 to indicate the sequential reference.

3 Experiments

This section details the experimentation on the implementation discussed in §2 in an attempt to determine whether the implementation effectively leverages IPL to achieve speedup over the sequential reference implementation.

3.1 Experimental Design

Evaluation of the implementations was conducted using the Distributed Asci Supercomputer (mark 4) cluster at the Vrije Universiteit Amsterdam. It offers dual quad-core worker nodes, each with 2.4 GHz processors, 24 GB of memory, networked using 1 Gbit/s bandwidth links. Processes were run *solo* on DAS-4 worker nodes. Run times were measured from the averages of recorded times per node, repeated thrice although no case exemplified significant variance.

In all cases, the program is fed an initial Rubiks Cube, scrambled deterministically using a zero-seeded pseudo-random number generator with a solution of 7 moves. The time taken to initialize and terminate slaves and the IPL service are *not* included in the runtime measurement presented in figures.

3.2 Results

Figure 3.1 shows the performance of the implementation in the face of a mild problem, with a cube of size 3. As expected, there was no result that benefited from using a nonzero M parameter when only one slave was available as the solitary slave would in

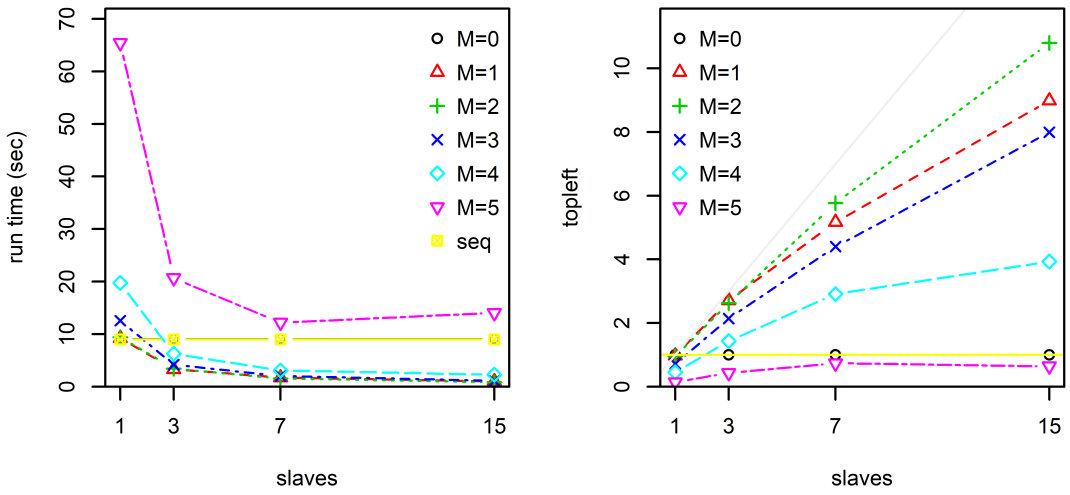

Figure 3.1: Run time and speedup for IPL-Java implementation for a cube of size 3. Each line indicates results for a different choice for the parameter M. The speedup plot shows a horizontal bar at 1 to indicate the sequential reference and a faint diagonal to indicate the linear speedup target.

all cases replicate the work of the master in the event of $M = \infty$, but with additional network overhead growing exponentially with M. This plot shows that the 'sweet spot' for M was 2, corresponding with 144 tasks per search depth[4]. Small values for M resulted in more significant load imbalance between slaves as tasks divided less evenly between them (leaving some partially idle). Large values for M generated highly-granular tasks which incurred excessive overhead as more messages had to be transmitted transmitted. Speedup increased with more workers, but sub-linearly; this stunted growth demonstrates that a significant portion of work was not divisible among slaves, and is likely a result of the problem size.

Figure 3.2 shows the results for another problem that differs only in the cube size, here set to 4. The runtimes reflect the vast increase in total work necessary, with this cube size inducing the larger branching factor of $2 \times 3(3) = 18$. While the solution is still found at depth 7 in the search tree, the increased branching factor increases the size of each worker tasks *much faster* than it increases the *number* of these tasks. Appropriately, the implementation approaches linear speedup more closely, seeing good results for both $M = 2$ and $M = 3$. Nevertheless, the cost of message passing is non-trivial even at this scale preventing the speedup from attaining truly linear speedup. By extrapolating, we predict that the results would continue to approach linear speedup for problems of increasing size.

[4]Here, N (cube size) is given as 3, resulting in a branching factor of $2 \times 3(\mathrm{N}-1) = 2 \times 3 \times 2 = 12$ as explained in §2. This cube generates 12^M tasks per search depth, thus 144 for $M = 2$.

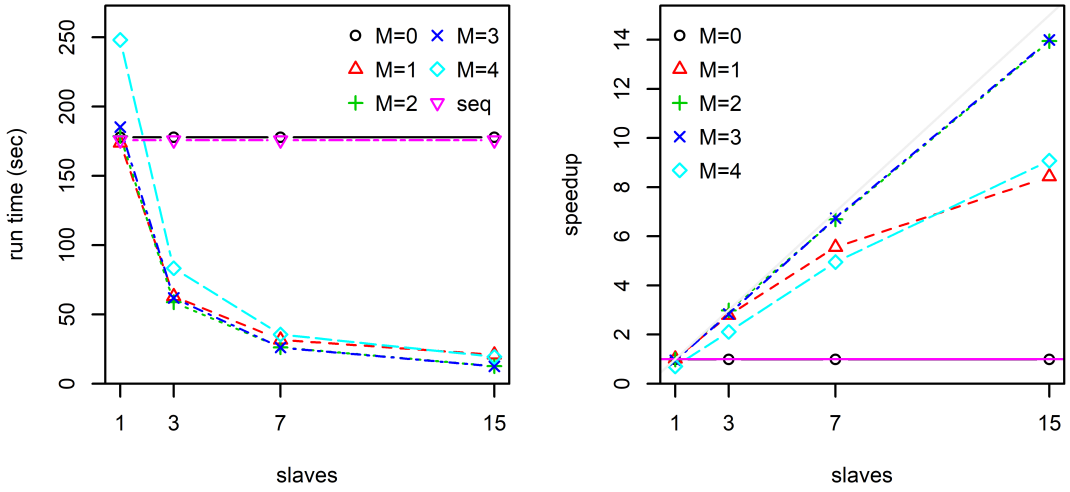

Figure 3.2: Run time and speedup for IPL-Java implementation for a cube of size 4. Each line indicates results for a different choice for the parameter M. The speedup plot shows a horizontal bar at 1 to indicate the sequential reference and a faint diagonal to indicate the linear speedup target.

4 Conclusion

The IPL-Java implementation of an IDA* solver for arbitrary Rubik's Cubes was developed successfully, shown to effectively exploit the master-slave model of concurrency to *approach* linear speedup.

Significant communication overhead prevented truly linear speedup from being observed in any experiments conducted for this work. This overhead was minimized with the use of *persistent* master-slave communication links, deemed superior to the alternative *transient* communication links (created for only the duration of each message transmission in an effort to eliminate upkeep cost for network links). Results in §3 suggest that linear speedup would be asymptotically approached with the application of the solver to ever-increasingly large problem instances, suggesting that this solver's utility grows in the most desirable cases: where the sequential implementation is left woefully inadequate.