Parallel Dominosa

Graham W. Smith and Elana M. Stroud

May 14, 2015

Abstract

We implemented the puzzle Dominosa of the Simon Tatham puzzle collection in a language we believe it has not been implemented in before, Java. A puzzle creator generates a puzzle of a given size and a puzzle solver finds the solution to a given puzzle. Furthermore, we created parallel implementations of both the creator and solver, which we also believe to be a first.

1 The Puzzle

Dominosa is a puzzle presented as a 2D grid of numbers 0 through n, for some specified n. For each unordered pair x,y for x and y in [0,n], there is a domino of those numbers that needs to be found in the grid. The puzzle is played by choosing orthogonally adjacent numbers in the grid to be paired in dominoes such that each number in the grid is part of exactly one domino.

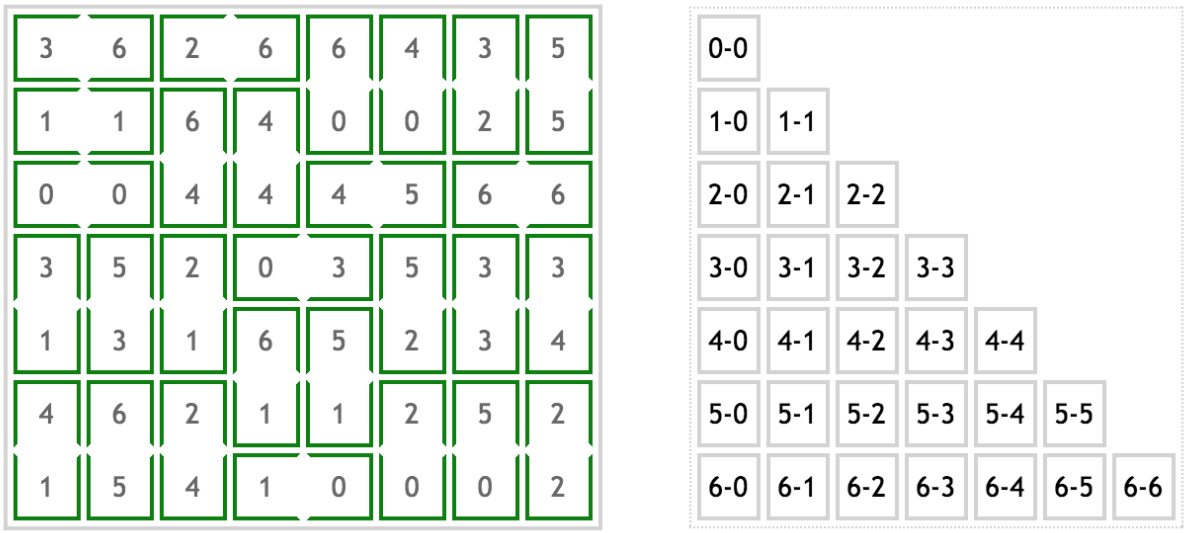

Figure 1: A solved instance of Dominosa for n=6 on the left. All the dominoes for n=6 are shown on the right. [1]

The number of dominoes in the puzzle are triangle numbers, as can be seen on the right side of Figure 1, such that the number of dominoes in a puzzle of size n is $\frac{(n+1)(n+2)}{2}$. This means that a grid for a puzzle of size n has dimensions width=$(n+2)$, height=$(n+1)$. Since there are $n+1$ distinct numbers 0 through n, this also means that each number appears $n+2$ times in the puzzle.

Mathematically speaking, Dominosa is related to the Exact Cover problem. "Given a collection S of subsets of a set X, an exact cover is a subcollection S^* of S such that each element in X is contained in exactly one subset in S^*. One says that each element in X is covered by exactly one subset in S^*" [3]. In our case, the set X is the set of all points in the grid, S is a collection of all pairs of points in X that are orthognally adjacent in the grid (i.e. all possible domino placements), and S^* is a valid way of covering the grid with dominoes before taking the numbers on the dominoes into consideration. Solving Dominosa is finding an exact cover S^* with the added restriction that S^* has exactly one domino representing each ordered pair of numbers 0 through n.

There are many valid numberless exact covers S^*, let's call the set of them T. From what we've already said above we know $|X| = (n+1)(n+2)$, $|S^*| = (n+1)(n+2)/2$. In S there are $n(n+2)$ possible vertical domino placements and $(n+1)(n+1)$ valid horizontal ones for a total of $|S| = 2n^2 + 4n + 1$. Determining/estimating the cardinality of T is much more involved and is not done here. But algorithms exist to find the set T of all exact covers, notably Knuth's Algorithm X and Dancing Links.

2 The Simon Tatham Implementation

Dominosa is a member of a collection of puzzles and their implementations collected by Simon Tatham[2], the creator of PuTTY. The Simon Tatham code is interested primarily in creating and solving puzzles that are interesting to humans, ones that could be solved deterministically without brute force guessing. This means the following: There is a Simon Tatham algorithm that deterministically finds a solution in the way a human might when playing the puzzle. The Simon Tatham puzzle creating algorithm, then, creates a puzzle with a solution that may not necessarily be reachable by the solver. The code to produce a puzzle runs the creator and runs the solver on the result, repeating this until a solution is produced that the deterministic solver can find. This ensures the puzzle presented to the user will both be solvable and reasonably playable by a user. The pseudocode is as follows:

Create a puzzle:
    Repeat until success:
        Create the set S of all possible domino locations
        Shuffle S
        For each possible domino placement in S
            Choose that domino for the puzzle's solution unless it conflicts
        We've now chosen dominoes, but some singleton points may be uncovered
        Repeat until no singletons remain:

BFS for a path from one singleton to another along complete dominoes
Remove the dominoes on that path from the chosen solution
Pair adjacent points on the path as dominoes
If running solver finds the solution, success

Solve a puzzle:
Create the set S as well as helper objects for the dominoes and grid
Repeat until dead end or complete:
Search for unordered pairs only found in one available location, pick those locations
Search for squares only covered by one available location, pick those locations

3 Parallelization

Since our implementation is in Java, our paralleliztion is accomplished using Java Threads and implementing Runnable. We first parallelized the solver using the fork-join technique. Inside the repeating loop we fork on the part that searches for distinct determinable pairs, performing the search in parallel via data decomposition splitting the 1D array into ranges for each Thread to perform on. We join and immediately fork again to perform the search by squares in parallel, decomposing the grid up into ranges of rows to be performed on by each thread.

We had intended to analyze the performance of the our parallel solver, comparing different Thread counts among other things, using a puzzle that took a non-trivial time to solve. What we discovered, however, was that for the biggest puzzle we had, 20 by 20, the time taken was too trivial to amortize startup costs or notice much difference between the parallel and serial implementations. It turned out that it was prohibitively expensive to produce puzzles of even of size 40 by 40 using our implementation, which would likely still be solved in a similarly trivial time. So while we had initially only written the creator so that we would have puzzles to feed into our solver, it became necessary to parallelize the creator in hopes of producing larger puzzles. For the creator we used a naïve approach to parallelize: Since the puzzle creator keeps trying solutions until one can be deterministically solved, we just make THREADNUMBER puzzle creators and have them each look for a solution at the same time.

4 Analysis

We ran metrics on several aspects of our Simon Tatham implementation, both exploring parallel principles and otherwise. After our initial discovery that the solver code performs too quickly on the puzzle sizes we had to perform any deeper analysis, we turned instead to gaining insight to the problem from as many different angles as we could manage. We first wanted a better sense of the time taken to create a puzzle of size n, especially since we noticed quite a bit of variance in runtimes between times where the creator got lucky and found the solution quickly versus the opposite scenario. So we first serially gathered

data demonstrating the trend of time taken to produce puzzles, as seen in Figure 2, which is related to the number of tries to reach a working solution, which we hypothesize is related to the cardinality of T for a given n. We also performed the same analysis using a 4 Thread version of the same thing. We found that the shape stayed the same but overall times decreased by a factor consistent with our subsequent speedup analysis.

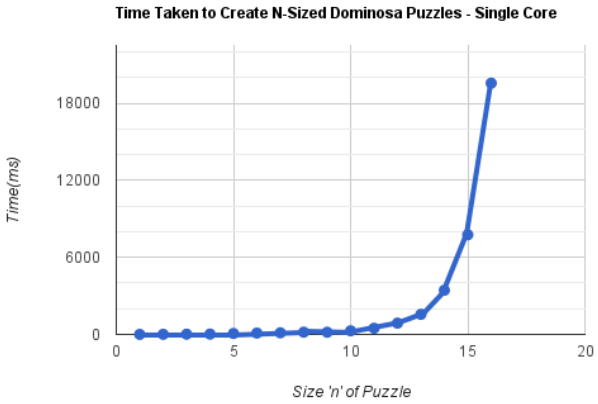

Figure 2: The time taken to create a puzzle of size n on 1 Thread. Closely related to the number of tries taken to reach a working solution, which is highly variable. Each data point is the average of 25 trials. Shape seems to be consistent with the a very rough estimate for the cardinality of T made using Wolfram Alpha, and tries taken should be closely related to the cardinality of T.

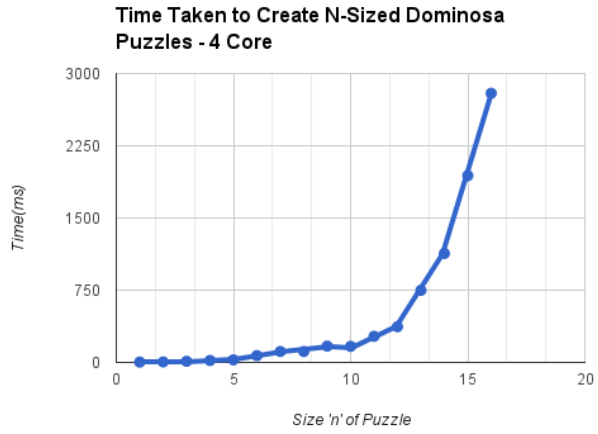

Figure 3: The 4 Thread version of Figure 2. The shape is the same with the times shifted down by a ratio consistent with the speedup at 4 Threads, shown in Figure 4.

We also ran our program on size n=20 for 1,2,4,8,16 Threads to obtain speedup data. Our results here were very interesting. Firstly, we in general noticed a wide variation between runtimes for a given n and thread count. We took a very large sample in an effort to

counteract this variation (about 80 samples) and found that the average runtime for n=20 actually increased past 4 threads. We wanted to know whether despite the large sample size this trend could have been due to particularly lucky or unlucky puzzle creators, so we took the sample standard deviation of our data for each thread count to give more insight to our sample mean. We interestingly found that our sample standard deviation in runtime increased when the sample mean of the runtimes did, but the standard deviations were not large enough to explain the drop after 4 threads as luck, especially for 80 trials.

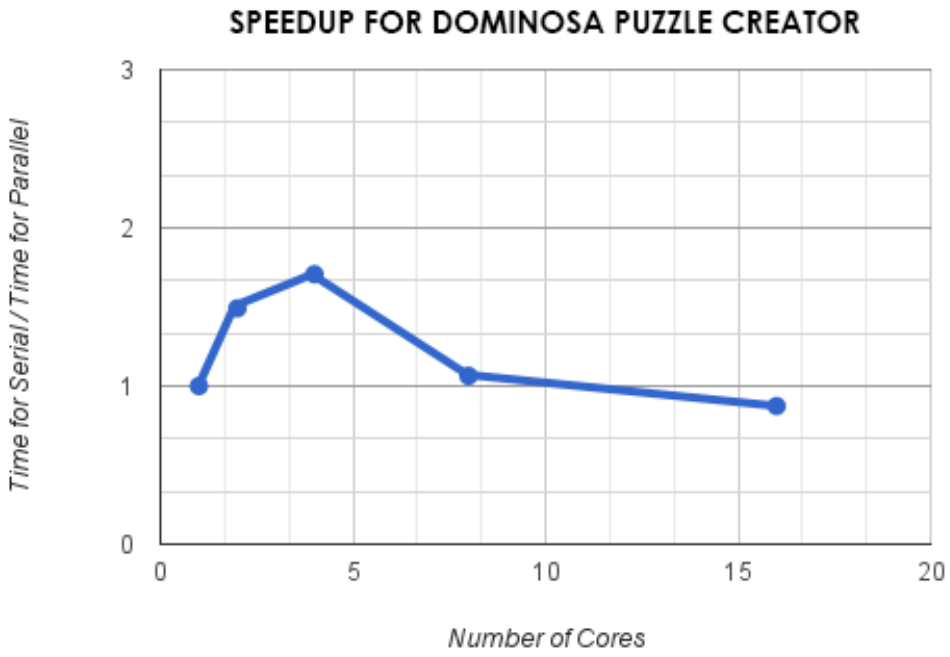

Figure 4: The speedup realized by our parallel puzzle creator for up to 16 threads for n = 20. Times reported by an Amazon c4.4xlarge. Notice the definitive dip at 8 Threads. Each data point is the average of 80 trials.

So there is quite a bit going on here that is difficult to explain. Sense can be made of the standard deviation increasing with the average runtime after some consideration: if the expectation value of the time it takes for some small probability event to occur increases, then there are more tries that occur before that time that can get lucky and find the solution very early. Likewise, since the probability is smaller it is possible we get unlucky and don't find a solution for much longer.

This could explain why standard deviation and the mean runtime are correlated, but it still does not explain why our puzzle creator loses speed when 8 or more threads are introduced. Our best explanation for this has to do with the hardware it ran on. We used a c4.4xlarge to gather the speedup data, which purports to have 8 physical cores and 8 virtual

cores[4]. Our parallel puzzle creator does not share any data across instances created in each thread, no members are static across the class and no shared data is given to them besides the puzzle size which it reads once before making a local copy. And yet since there must be some kind of interference between threads, since our naïve approach of just running THREADNUMBER insular standard puzzle creators means our threads are programatically distinct; it should be just like running the program on THREADNUMBER separate computers and using the first solution found.

We suspect that some technical detail of the c4.4xlarge hardware and its memory usage is the culprit. There are certainly supposed to be 8 physical cores, but perhaps these cores must have both individual and shared memory. We did not design our creator to be smart about space usage, since we initially only expected to use it as a way of avoiding handwriting large puzzles into files. For each try it makes, it creates new objects to use. SO it is possible that we rack up a significant amount of memory usage before the Java garbage collector frees up some of the useless memory. If this is the case, the threads/cores may vie for some amount of shared memory, causing interference and slowdown. This is our most educated guess as to the reason behind the shape of the speedup graph.

5 Further Study

There is more content in this problem and its parallelization that would be quite interesting to explore if time allowed. In Section 1 it was discussed how Dominosa is a tweaked version of an exact cover problem. At the very end of the same section we mentioned that there are algorithms that find all valid exact covers. One could imagine an alternate approach to finding a solution to a Dominosa puzzle that uses Dancing Links to find the set T of all possible ways to place non-overlapping dominoes on the puzzle and then traverses T checking whether each exact cover meets the additional requirement about the numbers on the dominoes. This algorithm has the advantage that it can find the non-deterministic solutions that the Simon Tatham algorithm rejects. However, we suspect that in serial it would perform worse than the Simon Tatham algorithm given our expectations for the cardinality of T. But we wonder what a parallel implementation would look like and how it would perform. The Dancing Links algorithm has been parallelized in a paper by Jan Magne Tjensvold [5], and it would be simple to parllelize the search through T by dividing its members among the threads.

6 The Team

Most of the technical work was completed using paired programming. Throughout the assignment we would switch who was driver and who was navigator. The problem solving was carried out by both of us together using whiteboards.

The Latex project writeup was written mainly by Graham. The data collection and manipulation, graph creation, README, and commenting of code was completed by Elana. The work was distributed evenly throughout the project.

References

[1] http://coderazzi.net/javascript/dominosa/dominosa.html

[2] http://www.chiark.greenend.org.uk/šgtatham/puzzles/

[3] http://en.wikipedia.org/wiki/Exact_cover

[4] https://piazza.com/class/i5a8yvmbufl2nu?cid=143

[5] https://janmagnet.files.wordpress.com/2008/07/decs-draft.pdf