Diego Fernandez (df2855)

Normal Magic Squares

Introduction

A Magic Square of order n is defined as an n by n grid of positive integers such that the sum of each column, row, and diagonal all add up to the same number. A Normal Magic Square is defined as a Magic Square that uses each integer from 1 to n^2 exactly once to fill the square.

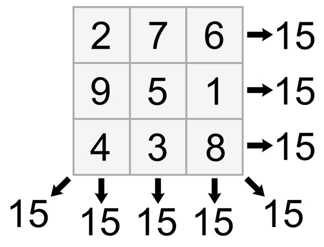


Image 1: A 3 by 3 normal magic square.

The sum of all columns, rows, and diagonals is known as the magic constant. For example, in the case of the image above, the magic constant is 15. This value is known for all normal magic squares. The formula is as follows:

$$M = n \cdot \frac{n^2+1}{2}$$

Project

The goal of this project is to create and parallelize an algorithm that counts the total number of normal magic squares for a value n. These values are known for orders up to six, which was just recently discovered. The total number of squares scales incredibly rapidly (the permutations of possible squares would be the factorial of n squared), so this project only focused on orders four and below. All known values are shown below. Note that due to symmetry, there are always eight times as many total squares as unique squares.

Order	Total (including rotations and reflections)	Unique (excluding rotations and reflections)
1	1	1
2	0	0
3	8	1

4	7040	880
5	2,202,441,792	275,305,224
6	17,753,889,197,660,635,632	2,219,236,149,707,579,454

Platforms

Two machines were used to test this program, both laptops. The first is an HP Spectre with 4 cores and 8 threads while the second is an Alienware with 24 cores and 16 threads. The HP Spectre is about 3.5 years old while the Alienware is about 2.5 years old. The Alienware, being a gaming laptop, is much faster than the HP Spectre, but has recently felt slower over the past few months.

	HP Spectre	**Alienware m16 R1**
Processor	11th Gen Intel(R) Core(TM) i7-1195G7 (2.90Ghz)	13th Gen Intel(R) Core(TM) i7-13700HX (2.10 GHz)
Threads	8	24
Cores	4	16
RAM	16 GB	16 GB

Sequential Implementation

The first step for this project was to create a sequential implementation in Haskell. I decided to use an existing algorithm as a base for my own, but was surprised to find out that these are hard to find. Especially since I wanted an algorithm that worked for a generic value of *n*. In the end I found an algorithm in StackOverflow which I could modify to take any *n* (See References).

The algorithm was a backtracking algorithm which filled in values in the square and would prune them early if any broke a rule for Magic Squares. It had a few other time saving measures as well, such as alternating rows and columns and automatically filling the last element of each row / column. The algorithm worked like this:

Each iteration the algorithm would alternate between filling in a row and filling in a column. This meant each iteration the size of the row / column would decrease. The order in which we fill elements for a 4 by 4 square is shown below. Since the magic constant for any *n* is known, we can figure out what the value for the last element in each row / column is by summing the rest of the row / column and subtracting that from the magic constant.

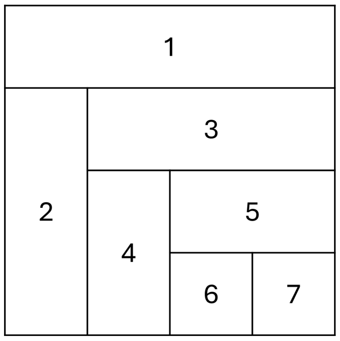


Image 2: The order of iteration for a 4 by 4 square

Because of this in each iteration, we obtain all possible permutations of values from the set of remaining elements of lengths equal to the length of the current row / column minus one. Then we calculate what the last remaining value should be. If that value is in the set of remaining elements we fill in that row / column then we recursively continue to the next, otherwise we prune.

This is repeated until the last element if filled in, afterwards we make sure that the square meets the requirements to be considered a magic square. That is, we check that each row, column, and diagonal add up to the magic constant. If so, we increment the count by one.

This algorithm was able to be reproduced in Haskel without too much issue. Below are the average time values obtained from running it on both machines.

Base Sequential Algorithm: `stack run “seq1” n — +RTS -s`

n	HP Spectre	Alienware m16 R1
3	0.002	0.003
4	9.170	6.241

As can be seen the order 3 square can be solved so fast that we won’t be able to correctly compare efficiency gains so for the rest of the report **we will only focus on 4 by 4 magic squares**.

Pruning Rotations and Reflections

As was explained, there are eight times as many magic squares as there are unique magic squares. That means if we can somehow efficiently obtain the number of unique squares, we will implicitly know the total number of normal magic squares. With an ideal efficiency of gain of 8x, this was the first sequential improvement I sought to add.

I went through various versions of algorithms to do this, each with varying levels of efficiency but this is the one I landed on:

We can fully describe the orientation of a square through its four corners. That is, all eight different rotations and reflections of a unique square will have unique corner values. Therefore an algorithm that can always prune exactly seven of them will be up to eight times as efficient. The conditions I came up with are:

1. The top left corner must be the smallest of all corners (prunes six rotations / reflections)
2. The top right corner must be smaller than the bottom left corner (prunes the last one)

Meeting these conditions is simple, in the first row we just make sure the last value is bigger than the first value. Then in the first column we make sure the last value is bigger than the top right corner. Then finally when we reach the bottom right corner we make sure that its value is bigger than the top left corner.

The results of implementing the algorithm and running `stack run “seq2” 4 – +RTS -s` is shown below.

Rotation/Reflection Prunner Sequential Algorithm: `stack run “seq2” 4 – +RTS -s`

	HP Spectre	**Alienware m16 R1**
Average Time	1.590	1.039
Speedup (from seq1)	5.76	6.01

Parallel 1

My first approach at parallelizing the algorithm was a simple one. Simply assign a maximum depth for rows / columns to parallelize and create a spark for each permutation of that row / column. To do this I used `par` and `pseq`. Because there are already so many possible permutations of the first row, I expected the most efficient depth to be 1. Tests seem to collaborate with the expectations. Results can be seen below for both machines.

HP Spectre: `stack run “par1” 4 – +RTS -N4 -s -l`

	Depth 1	**Depth 2**
Average Time	0.646	0.739
Speedup	2.46	2.15
Sparks	3,360	139,716
Converted	2,238	19,653
GCd	666	59,344

Fizzled	785	60,719

Alienware: `stack run “par1” 4 – +RTS -N8 -s -l`

	Depth 1	**Depth 2**
Average Time	0.206	0.220
Speedup	5.03	4.71
Sparks	3,360	139,584
Converted	2,160	17,772
GCd	666	61,031
Fizzled	535	61,034

The maximum speedup obtained from the alienware was much higher than the one obtained from. But for both, depth one was the clear winner.

Parallel 2: Granular

I believe one of the biggest cons of the first parallel approach is that the depths are too impactful. That is, there is nothing smaller than depth 1, nor is there anything between depth 1 and 2. This is especially problematic for the theoretical order 5 square where even a depth of 1 would create exponentially more sparks. As such I decided to significantly change the original code.

Rather create all possible permutations of each row and column at the same time, the new approach would simply recurse through each element individually. The order would still be the same. We would still go row then column then row. In addition we would still group the last element of each row / column with the previous element since we can automatically know the value without needing to recurse. This would keep all of the efficiency from the first algorithm while allowing us to control the depth more. The order in which we fill in elements of a 4 by 4 square in the new algorithm is shown below.

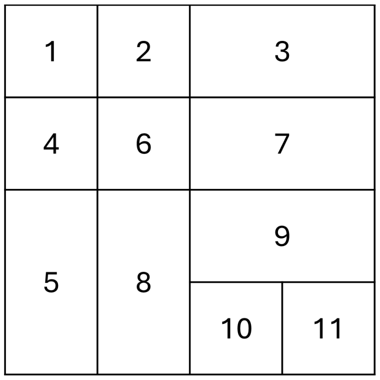


Image 3: The order of iteration for a 4 by 4 square

What I did not expect was for this algorithm to also be more efficient sequentially. The results that show this are below. Notice how when only using one core it is significantly faster than par1.

HP Spectre: `stack run <mode> 4 – +RTS -N1 -s -l`

	seq1	**seq2/par1**	**par2**
Average Time	9.179	1.590	0.685
Sequential Speedup (vs seq1)	1	5.76	13.4

Alienware: `stack run <mode> 4 – +RTS -N1 -s -l`

	seq1	**seq2/par1**	**par2**
Average Time	6.241	1.039	0.497
Sequential Speedup (vs seq1)	1	6.01	12.55

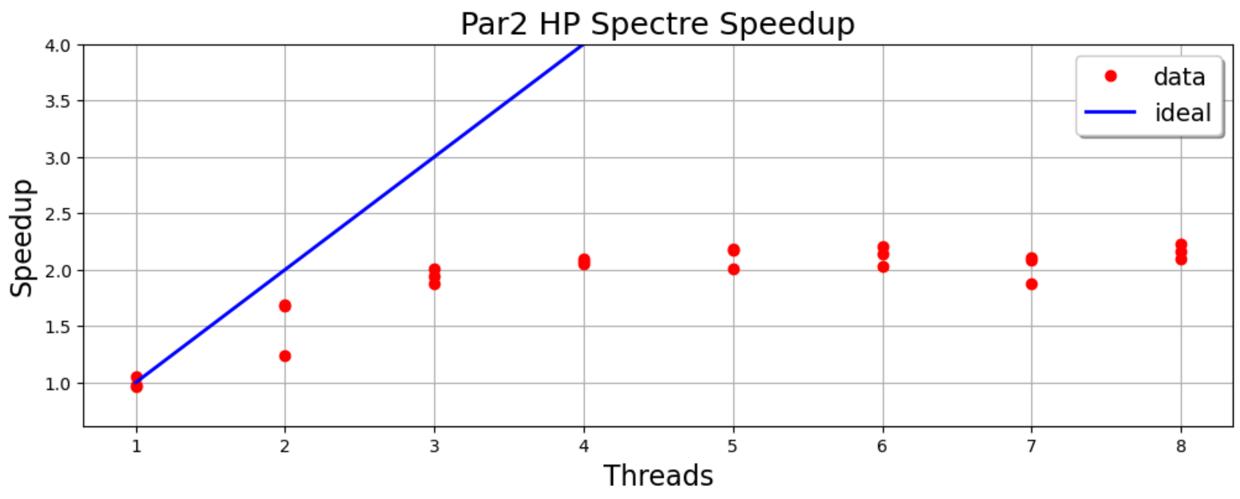
As can be seen even with just one core, and no parallelization this method sequentially doubled the speedup of the previous method, which was not even the intended part! After this discovery I went on to experiment to see which depth I should use.

Alienware: `stack run “par2” 4 – +RTS -N8 -s -l`

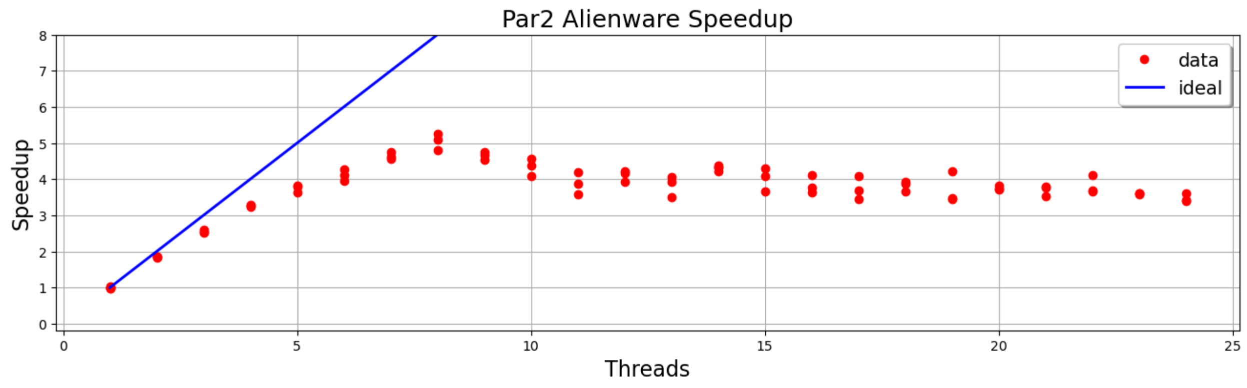
	Depth 1	**Depth 2**	**Depth 3**
Average Time	0.127	0.105	0.110
Parallel Speedup	3.91	4.73	4.52
Sparks	13	208	2938
Converted	7	47	297

GCd	1	37	1290
Fizzled	5	125	1351

While par1 could reach parallel speedups higher than five, this current method seems to only just reach it occasionally. Despite this, the significant sequential speedup still makes this the superior method. The speedup graph can be seen below for both laptops.
[
HP Spectre: `stack run “par2” 4 – +RTS -N<threads> -s -l` (depth 2)



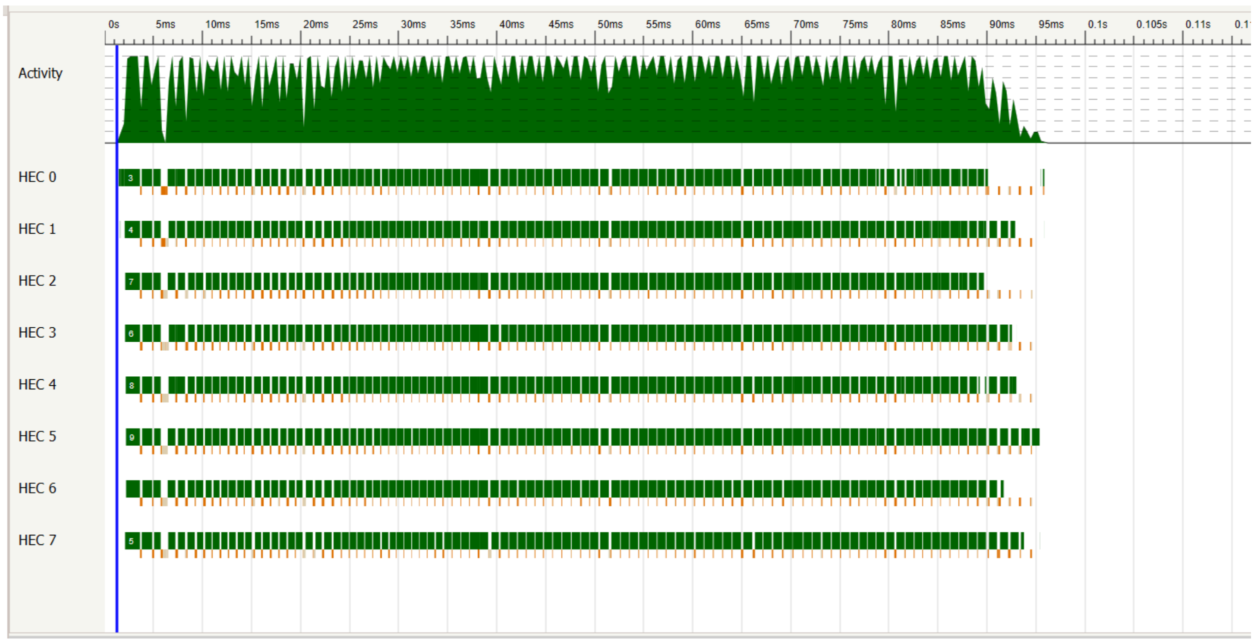
Alienware: `stack run “par2” 4 – +RTS -N<threads> -s -l` (depth 2)



We can see that 4 threads seems to be the best value for the slower laptop while 8 threads seem to be the most efficient for the Alienware, sometimes passing a speedup of 5, before it tapers off at around that value. The HP Spectre laptop only has 4 cores, so it could be that having multiple threads per core does not cause better parallelization. For the alienware the change seems more drastic, so it could be that something about the algorithm causes it to slow down with more than eight threads. Looking at the threadscope and spark detail can give us more information as for the cause.

Alienware: `stack run “par2” 4 – +RTS -N8 -s -l`

```
None
  SPARKS: 208 (37 converted, 0 overflowed, 0 dud, 36 GC'd, 135 fizzled)
  INIT    time    0.000s  (  0.001s elapsed)
  MUT     time    0.109s  (  0.086s elapsed)
  GC      time    0.281s  (  0.009s elapsed)
  EXIT    time    0.000s  (  0.001s elapsed)
  Total   time    0.391s  (  0.096s elapsed)
```



The biggest problem seems to be the size of our sparks and garbage collection. While garbage collection makes up a small amount of elapsed time, it is a huge amount of CPU time. A possible reason for this is the very inconsistent sparks. Magic Squares are almost inherently very inconsistent. The values are not evenly distributed about the square and many sparks just die off immediately, while others go on for a very long time. Getting around this fact may be difficult, but would be a good start in improving speed up from parallelization.

Results and Future Work

Overall I am very happy with the results obtained from this project. I was able to obtain significantly more speedup than I had originally expected. With **12.55x** Sequential speedup and **4.73x** Parallel speedup this project was able to speedup the original algorithm that I obtained from stack overflow by **59.4x.** And that is without even counting the language speedup since Haskell is much faster than Python.

Despite this there is still more work that can be done. The way I parallelized the algorithm was not as efficient as it could have been. In addition, the large amount of garbage being collected makes me think I could be smarter about which data structures I am using. The algorithm itself can also have some improvements. In particular, I want to test out iterating in a ‘swirl’ pattern filling in the entire outside before going inside, which would allow me to prune the final reflection / rotation check much sooner.

References

Original Python Algorithm:
https://stackoverflow.com/questions/55435518/how-to-optimize-recursive-permutation-algorithm-for-all-4x4-magic-squares