Joshua Rowley

Abstract

A refactoring and multithreading of python simple sudoku solver

CAB401 parallelism Project

By Joshua Rowley n10233121

Contents

[Initial application structure 2](#_Toc54100246)

[Potential parallelism 3](#_Toc54100247)

[Process of parallelism 4](#_Toc54100248)

[Software, tools and techniques 5](#_Toc54100249)

[Modified code 6](#_Toc54100250)

[Results 7](#_Toc54100251)

[Validity of outputs 8](#_Toc54100252)

[Performance problems / barriers 9](#_Toc54100253)

[Reflection 10](#_Toc54100254)

[References 11](#_Toc54100255)

# Initial application structure

The original application’s purpose is to solve large simple sudoku puzzles and if it cannot solve them it returns the iteration count and the replaced values. Simple sudoku puzzles are puzzles that can be solved through only the use of the 3 basic sudoku rules. These rules are 1. No duplicate values in a single row, 2. No duplicate values in a column and 3. No duplicate values in each segment. A segment’s length and width are equal to the square root of the total grid’s sizes. The third rule is the most inconsistent with larger sizes of puzzles not always being powers of 2. However, in the case of this application we will assume the size of the puzzle is 15 to the power of 2. Figure 1 Shows each segment of a 9x9 sudoku puzzle.

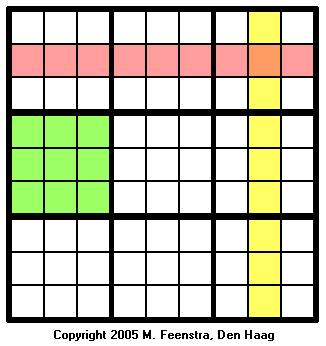


Figure 1 Sudoku sections

The first version of the linear application looped through every column from top to bottom. Then checks if a value is present. If no value is found a call is made to 3 other functions. These 3 functions check each rule. The functions checked the required cells and returned a list of values it could not be. These 3 lists are saved separately into 3 Lists that are the same size as the input sudoku puzzle. After checking all cells and adding to these lists a fourth function is called, this function then combines the three lists that are created during the checking step. They are combined by removing from the possible values upon combining the three lists an output is produced if the result of this elimination is a single value. Therefore, by elimination this value is the only one possible in the square.

The output from this process is then checked for the total null values, which is then used as the conditional for if the while loop must continue. However as stated initially this sudoku solver cannot solve puzzles that require inferencing from other cells and due to this it must be checked if any progress was made during the loop this is done by comparing the previous null check with the current null check thereby stopping the loop if no additions are made during an iteration. This is to ensure the loop will end if it cannot solve any puzzles.

# Potential parallelism

In the original code there are 3 primary loops that have potential for parallelization. These loops are a while loop for total execution of the rules, a for loop through each cell that applies the rules and another loop that combines the rules outputs. These loops contain potential to be multithreaded as they have no dependencies on each other however the large amount of memory that will be needed makes this solution unfit to be effectively scaled as, a sudoku puzzle of 9x9 will require 9x9x4 datapoints of memory.

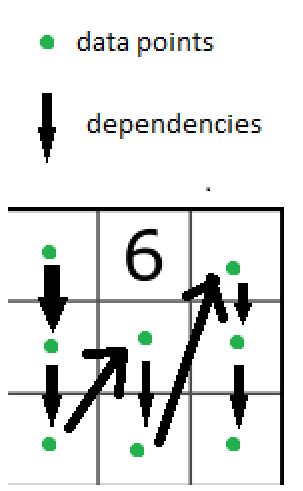


Figure 2 data dependencies

Another option would be to optimize these lists to remove the excess memory usage this could be done in several ways. Firstly, combining the process of the three rules functions into a single function that can return a value will. This can be done easily as the code for each rule can be stacked together directly. Through this it removes the need for the 3 lists and combining them and instead directly input values into the sudoku puzzle for use in the next operations. These changes result in data dependencies from the start of the loop to the end. The data dependencies of this model can be seen in figure 2. As the data that needs to be compared to is changing every single cycle. This does result in significant improvements to the linear equation as it reduces the required iterations to solve the puzzle. As using previously assigned values can reduce possible values for other cells.

However, with this method multithreading the process becomes more difficult as the puzzles data is constantly changing during the execution of the loop. A solution to this would impact the required iterations which may negatively impact the time improvements, as an extra iteration is likely to take anywhere from 2-10 seconds.

# Process of parallelism

The three mapping methods that can be used are through row sets, column sets and quadrant sets. The row set and column set data mapping methods are similar in the required data to send to each thread. While the quadrant method has some benefits but will require additional resources to break the data down for the threading process. Each method will be expanded on below

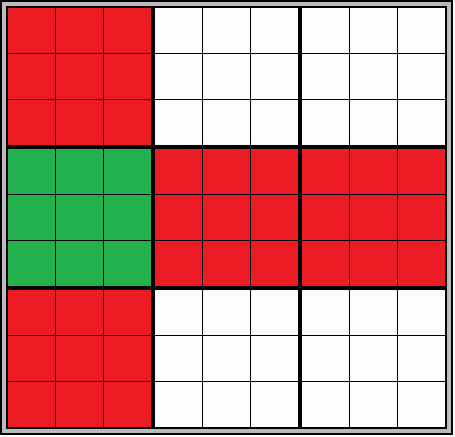


Figure 3 quadrant data

The quadrant splitting method as the name implies would divide the data into a number of equal sized chunks based on the sections they are in. in a 9x9 matrix these sections are 3x3 in the case of a 225x225 matrix these sections are 15x15. The benefits of this method are that the required data for each thread is reduced as they would only need the rows and columns the quadrant is a part of. An example of the data that would be required is shown in figure 3. The column data is simple to extract as the data is stored as columns while the row data will require an additional looping proccess to correctly extract the information.

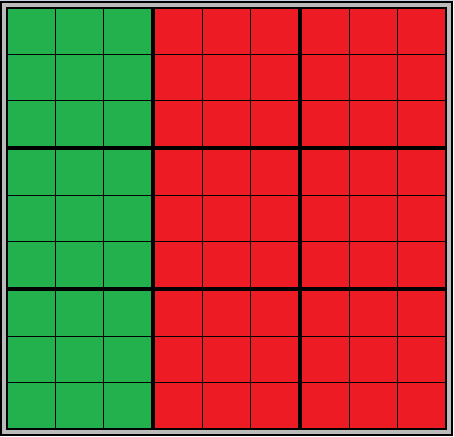
The columns and row method splits the data into groups by either rows or columns. These groups must be divisible by the length of the sudoku for a 225\*225 sudoku puzzle groups of 225, 75, 45, 25 and 1. With this method of mapping the data to the threads the whole sudoku puzzle must be sent to each thread as the whole table is needed for the processing of the data. A visualisation of the data for a column can be seen in figure 4. While this method stores a larger amount of information it simplifies the data proccess and allows for generalisation of the function. An issue that must be taken into account is that as the size of the groups decrease the iteration count to solve the puzzle will increase due to the lack of cross comunication between the proccesses. This should not become apparent until groupsizes reach a certain size.

Figure 4 column data

Due to seperating the data dependencies there is no need to synchronize data between threads. There is a potential to synchronize between the threads however the benefits of this do not out way the issues of waiting for synchronization to occur. As synchronization between threads would have to occur inbetween cell operations. This would overall increase the execution time rather than decrease.

# Software, tools and techniques

Python is used for this project as it is the language, I am most comfortable in using, as I have the most experience in creating python projects. The compiler that was used for the duration of the project was sublime text as it is a versatile application that can handle a wide variety of language. The library that was used in the final version of the parallelized project was the multiprocessing library that is included as a standard library. Another method that was explored and implemented was the threads library however the threads library does not use multiple cores of the CPU. While Threads implementation was a dead end in development it greatly helped in the implementation of the process function as they have very similar initialization methods however, they differ greatly in the returning of values. The threading method could replace values in a global variable to return data while this is much more simplistic and could result in conflicts should data try to write to the same cell. The process function can return data to an implemented data type called Que which can store the data from the processes separately from each other ensuring no corruption due to access errors.

# Modified code

The code initial was optimized as it had a few glaring issues in terms of optimization, this newly optimized code will be used as the basis for further optimization through parallelization. After Identifying the loop that can be parallelized, firstly the data must be assigned to any number of processes. The start of the divisions can be found by multiplying the process id by a scaling factor. The scaling factor can be found by multiplying the quadrant sizes by the total length divided by the total proccess count times 15. For proccess total of 3 this returns increments of 75. Finding the end of division is simply adding the scaling factor to the start of the division. Rets is a list that will contain the returned values from the proccesses. The data is returned through the Queue which is an in built class which can be called to wait for proccesses to finish and acts as a pipe for the data. This data is then fed into the process function as arguments for the Threadingcalc function. This process is then added into a list for later operations. The list of processes is then looped through to get the return values from each and then append them to the return list. Then the processes are closed in the next loop with the .join() function. Finally the rets list is looped through and replaces the coresponding global data with the processed data. This is the method of creating additional processes and getting return data. The modifications that needed to be made to the calculation function was relativly simple after optimizing the original code and addapting it to 225x225 msudoku puzzles.

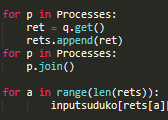
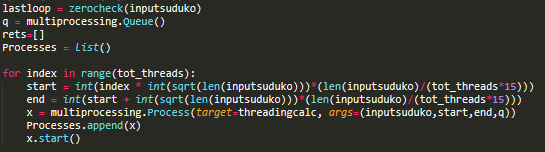
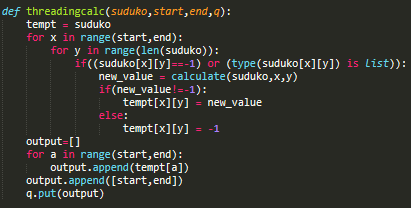


Figure Parallelized method

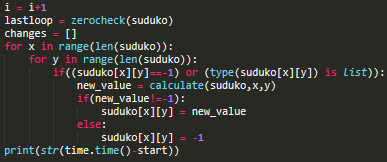


Figure Linear calculation method

# Results

d. Timing and profiling results, both before and after parallelization and a speedup graph.

As the method requires the process count to be a divisor of 225 the process counts that will be tested are linear, 1, 3, 5, 9 and 15 processes. This should give clear improvements to the execution time and allow for an in-depth view of the impacts of breaking down the processes.

As was predicted from the outset the iteration count is impacted negatively with parallelism, as there is no communication between the processes as it would decrease execution time greatly. Despite increasing the iteration count by 42-57% there is a clear improvement in the execution time for low process counts. 3 and 5 processes result in a significant improvement to the overall execution time.

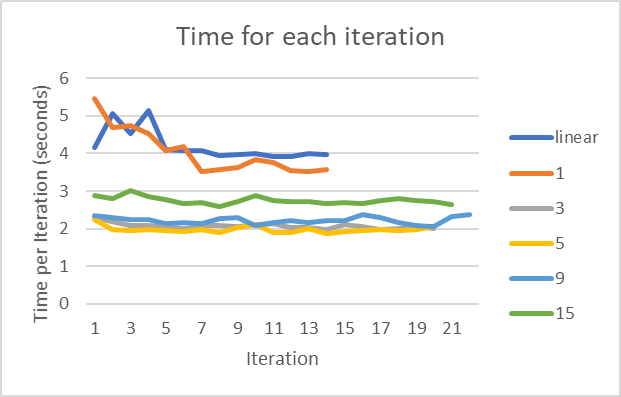


Figure iteration execution time

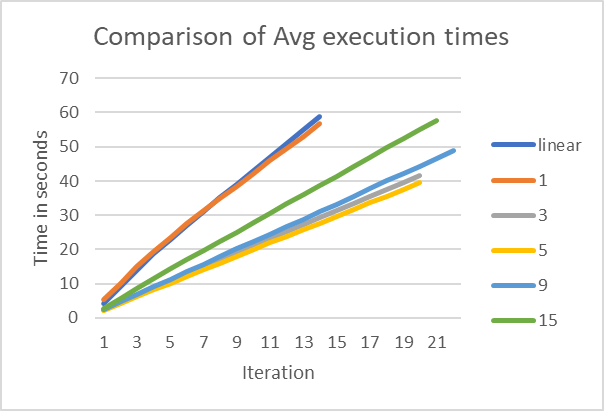


Figure Execution time graph

From the changes in Iteration execution time it is clear that time doesn’t change greatly between steps when not using the linear version. The linear version decreases greatly after the first 5 iterations. This is due to the reducing amount of values that need to be calculated. For 3, 5 and 9 processes the execution time does not vary greatly. This would be due to the reduction of values that must be calculated is not significant for each process.

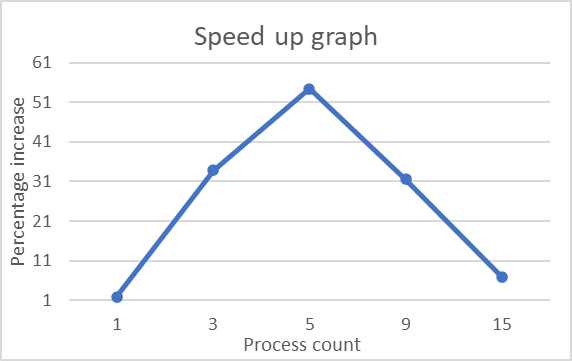


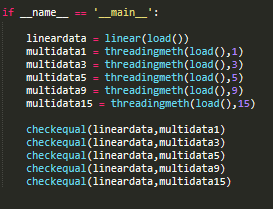
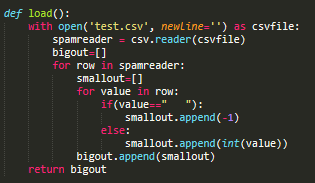
Figure Speed up in Percentage

The speed improvements for each process count can be seen in figure 9. From the graph it is clear that the greatest improvements can be found when the data is not divided into smaller divisions. The best results occurring when the data is divided into 5 subdivisions of 45. This optimal value will change based on the puzzle as the different solution will change the load imbalance of the data.

# Validity of outputs

Validation of the results is simple as the output for the program is a sudoku puzzle with values filled in. To ensure the parallel version does not break the results the linear output is looped through and directly compared to the output of the parallel version. Then the total mismatching values are printed to the console. This method of comparison while slow and wasteful is effective in understanding if the changes that have been made have broken any step in the chain. Another method would have been to save the original data output into a CSV file to be loaded at a later date however waiting a short time removes the need for an additional file to be present.

Figure Validity testing



# Performance problems / barriers

As the program does not solve the puzzles completely but solves for simplistic results, load imbalances should be kept to a minimum. Despite this however load imbalances could occur if a sudoku puzzle’s supplied values are mostly on one side. This issue could have been somewhat mitigated if instead of dividing columns by equal indexes they were divided by missing values. However to determine this it would require looping over the whole sudoku puzzle an additional time which I do not believe would be worth the execution time as the chances of having imbalanced initial data is much lower unless intentionally done so. And in the case of data being heavily sided to one group of columns the data groups could be broken down by rows instead which would remove the significant imbalance by changing the divisions.

# Reflection

During this project I have learnt how to create and manage Parallelized functions in python using the multiprocessing library. I have also learned to effectively breakdown functions into data independent methods through mapping the data dependencies visually. My attempt at parallelization of a program was mostly successful as the best speedup achieved was 54% at 5 cores. This speed up may be able to be improved upon by better data storage techniques and better mapping to individual processes the overall result was satisfactory. Despite the relative success of the project as a whole if I was to redo this project I would most likely choose a more complex program as I did not for see the sudoku puzzle program being as simple as it was. However, by the time I realized its relative simplicity I was to far into development to turn back and change my overall project.