Parallel Programming – MPI and OpenCL Sudoku Solver

PROJECT REPORT

Ninad S Shetty (Sec 'A' - 140905103) Prajwal P (Sec 'A' - 140905123)

6th Semester, CSE, Manipal Institute of Technology

4th APRIL, 2017

INTRODUCTION

In this report, we present our implementation of a highly parallelized Sudoku Solver using the many core architecure. We begin with a discussion of the abstract andmotivation for our work and also present our contributions. Then we present a detailed account of the work completed. Later, we discuss our evaluation methodology and then we present our results. We also shoe related work in existing Sudoku and other logic game solvers. Last section provides a discussion of the future direction this project could take.

ABSTRACT

Sudoku is a logic-based number-placement puzzle game where the players goal is to complete a nxn table such that each row, column and box contains every number in the set {1, . . . , n} exactly once.Total number of valid Sudoku puzzles is approximately 6.671×1021 (9*9 matrix). Trying to populate all these grids is itself a difficult problem because of the huge number of possibilities and combinations of puzzles, Assuming each solution takes 1 micro second to be found, then with a simple calculation we can determine that it takes **211,532,970,320.3 years** to find all possible solutions. If that number was small, say 1000, it would be very easy to write a Sudoku solver application that can solve a given puzzle in short amount of time. The program would simply enumerate all the 1000 puzzles and compares the given puzzle with the enumerated ones and we don't have to design a parallel algorithm for finding Sudoku solutions. Unfortunately, this is not the case since the actual number of possible valid grids is extremely large so it's not possible to enumerate all the possible solutions. This large number also directly eliminates the

possibility of solving the puzzle with brute force technique in a reasonable amount of time.

MOTIVATION

This project is motivated by the increase in popularity of Sudoku and similar logic puzzles over the past decade . Numerous serial Sudoku solver implementations exist and are readily available. However, due to its recency, there is relatively little work on parallel implementations. Parallelizing a serial Sudoku solver can improve its speed and increase the viability of solving larger size Sudokus. The Sudoku solver algorithm also conveys some of the basic tradeoffs of parallel software development such as dependencies and work-sharing which are interesting to inspect and study

OBJECTIVES

We propose that a method for solving the puzzle quickly will be derived that takes advantage of many core architecture on modern and future computer systems which will bring down the time cost of solving an *NxN* Sudoku to a reasonable amount.

Our goal is to allow for the algorithm to take advantage of the many-core architecture to further reduce its running time. This is very important and is at the heart of our work because for large *N*, the number of cells per row and column in the Sudoku grid, finding a solution becomes an extremely difficult and computationally intensive problem. Hence, the parallel algorithm must also scale as the grid size increases.

LITERATURE SURVEY

Parallelization of Sudoku Alton Chiu (996194871), Ehsan Nasiri (995935065), Rafat Rashid (996096111) {alton.chiu, ehsan.nasiri, rafat.rashid}@utoronto.ca University of Toronto December 20, 2012

Parallelized Sudoku Solver using the pthreads library on the Linux kernel.

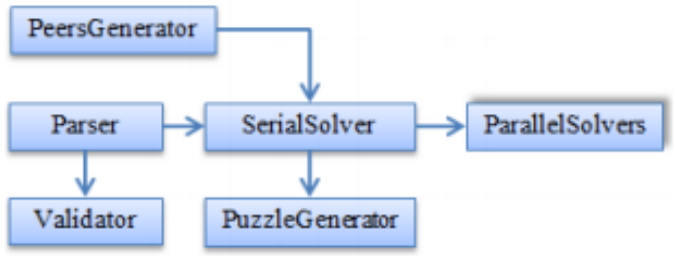


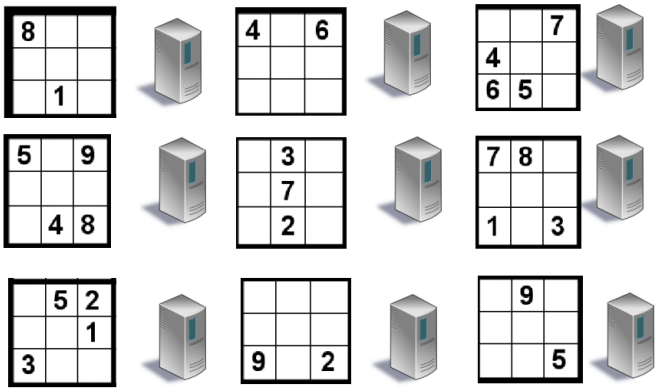
Figure 4: Implementation breakdown

The arrows illustrate the dependencies between the different components. The Parser is used to read and write Sudoku puzzles from a file. As an example, a 16x16 puzzle would contain numbers from 1 to 16, with 0s denoting the unsolved cells. The Validator is used to verify both solved and unsolved Sudokus produced by our solvers and our Sudoku Generator. The Generator produces puzzles with unique solutions. The Peers Generator produces the peers list given the size of the puzzle as its input, as the peers list is different for puzzles with different sizes. By generating the list of peers for each cell, it removes the need to calculate the indexes of each peer during the execution of the algorithm. Their parallel algorithms uses the pthreads library.

METHODOLOGY

We initially divide the entire Sudoku Input matrix into Rows. We then find the permutation of all possibilities for a row and match it with input. For example, a row with 9 elements has 9! Possibilities. We need to understand that each row has 9! Possibilities. This process of finding all the possibilities is computed in parallel by threads in the kernel. We then gather these rows into a larger 2D matrix.

In the next step we compute all combinations of all possibilities of all rows. Hence we can have 9! X 9! Combinations. These combinations have to be checked if they abide by the Sudoku Rules i.e NO repetitions in Rows or Columns or the Smaller blocks. This process of checking is once again done in parallel by threads in the kernel.

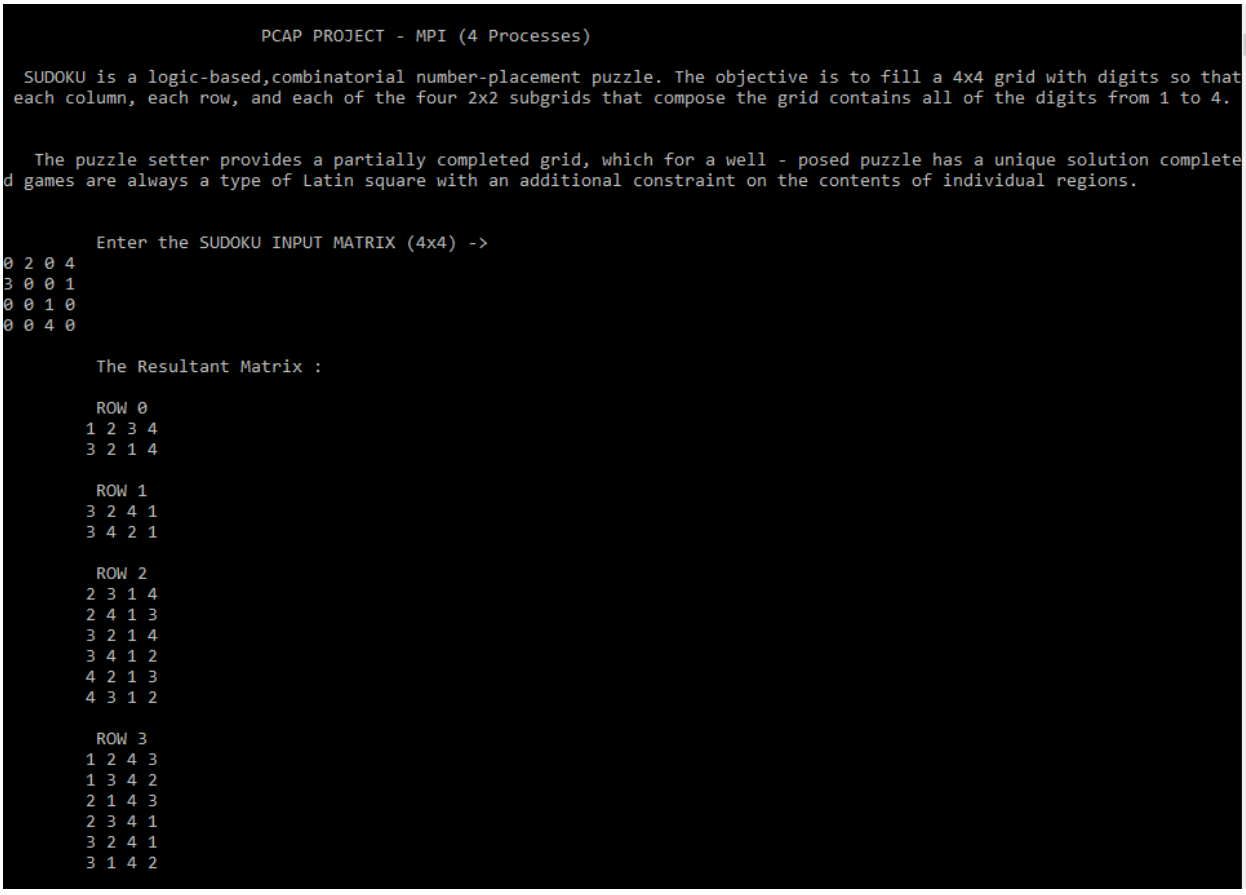


RESULTS

Sudoku has given rise to other logic games and brainteasers in the past decade [1]. Like solving Sudoku, the logic and strategy in each of these games can be translated into rules and converted into automated solving programs.

An immediate speedup is observed for parallel algorithms as we add more threads to perform work. This speedup is achieved due to the fact that processes independent of each other can run concurrently.

MPI :



```
Total Combinations : 144

        || Rank : 0||


        ** Thread ID : 98 -> SUCCESS **BREAKING : 144 >= 144

         TOTAL POSSIBLE COMBINATIONS : 144

                 -- SUDOKU INPUT --

                0 2 0 4
                3 0 0 1
                0 0 1 0
                0 0 4 0

                 -- SUDOKU SOLUTION --

          THREAD ID : 98 | COMBINATION :  1 2 6 3 |

                1 2 3 4
                3 4 2 1
                4 3 1 2
                2 1 4 3


^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

        || Rank : 2||
BREAKING : 2600 >= 144

        || Rank : 1||
BREAKING : 1300 >= 144

        || Rank : 3||
BREAKING : 3900 >= 144
```

OPENCL :

```
                      PCAP PROJECT - OpenCL

 SUDOKU is a logic-based,combinatorial number-placement puzzle. The objective is to fill a 4x4 grid with digits so that
 each column, each row, and each of the four 2x2 subgrids that compose the grid contains all of the digits from 1 to 4.


   The puzzle setter provides a partially completed grid, which for a well - posed puzzle has a unique solution complete
d games are always a type of Latin square with an additional constraint on the contents of individual regions.


        Enter the SUDOKU INPUT MATRIX ->
0 2 0 4
3 0 0 1
0 0 1 0
0 0 4 0

        The Resultant Matrix :

        ROW 0
       1 2 3 4
       3 2 1 4

        ROW 1
       3 2 4 1
       3 4 2 1

        ROW 2
       2 3 1 4
       2 4 1 3
       3 2 1 4
       3 4 1 2
       4 2 1 3
       4 3 1 2

        ROW 3
       1 2 4 3
       1 3 4 2
       2 1 4 3
       2 3 4 1
       3 2 4 1
       3 1 4 2

 Total Combinations : 144
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

        ID : 0 | COMBINATION : 1 1 1 1

            --- SUDOKU Matrix ---
               1 2 3 4
               3 2 4 1
               2 3 1 4
               1 2 4 3

        ID : 72 | COMBINATION : 2 1 4 2

            --- SUDOKU Matrix ---
               3 2 1 4
               3 2 4 1
               3 4 1 2
               1 3 4 2

        ID : 108 | COMBINATION : 1 2 6 4

            --- SUDOKU Matrix ---
               1 2 3 4
               3 4 2 1
               4 3 1 2
               2 3 4 1

        ID : 126 | COMBINATION : 2 2 2 6

            --- SUDOKU Matrix ---
               3 2 1 4
               3 4 2 1
               2 4 1 3
               3 1 4 2


        ID : 127 | COMBINATION : 2 2 6 2

            --- SUDOKU Matrix ---
               3 2 1 4
               3 4 2 1
               4 3 1 2
               1 3 4 2
```

```
        ID : 35 | COMBINATION : 2 1 1 5

            --- SUDOKU Matrix ---
               3 2 1 4
               3 2 4 1
               2 3 1 4
               3 2 4 1

        ID : 71 | COMBINATION : 2 1 2 4

            --- SUDOKU Matrix ---
               3 2 1 4
               3 2 4 1
               2 4 1 3
               2 3 4 1

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

        TOTAL POSSIBLE COMBINATIONS : 144

                -- SUDOKU INPUT --

               0 2 0 4
               3 0 0 1
               0 0 1 0
               0 0 4 0

                -- SUDOKU SOLUTION --

         KERNEL ID : 98 | COMBINATION :  1 2 6 3 |

               1 2 3 4
               3 4 2 1
               4 3 1 2
               2 1 4 3


^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

LIMITATIONS AND POSSIBLE IMPROVEMENTS

We have impelemented our algorithm on a 4x4 matrix. We would like to scale our project to a larger 9x9 or a 16x16 puzzle by overcoming some limitations by pruning, etc.

One possibility would be to see the impact of using a larger number of threads. For this to be beneficial, we may need to test on a machine that can take better advantage of a larger number of threads. A higher number of cores would allow for more threads and possibly better parallel execution. Problems encountered with memory allocation for larger sized puzzles could possibly be fixed with a different implementation that would be more conservative with puzzle creation. For example, instead of duplicating the puzzle for each search candidate, we could save only the difference between two puzzles. Using this method, upon a contradiction on one branch of the tree, we can traverse back up the tree and re-generate the puzzle using the diffs saved.

CONCLUSION

This report introduces several parallel implementations of a Sudoku solver based on Constraint Propagation and Search methods. Strongly connected dependencies made it extremely difficult to parallelize CP. Traversing the solution space tree during a Search in parallel is the best way to reach a solution faster. We implemented our project in OpenCL and MPI methods with the major difference in how a thread handles a particular process. We took full advantage of being able to create more threads in OpenCL.

For most puzzles, we observed an immediate speedup with most methods by increasing the number of threadsOverheads caused by increasing the number of threads beyond four did not allow the runtime to decrease any further.

REFERENCES

Heterogenous Computing with OpenCL –Benedict R Gaster, Lee Howes
Parallel-Programming-in-C-with-MPI-and-OpenMP, Michale J Queen
D. Taylor, “Solving every sudoku puzzle,” Logical Genetics.