Arbitrary Size Parallel Sudoku Creation

William Dudziak

1 May 2006

Outline

1. What is Sudoku?

       Sudoku puzzles are logic puzzles in which a set of provided symbols are arranged within a grid in such a manner such that the player can deduce all the remaining symbols and completely fill in the grid. The most common form of Sudoku grid is what is referred to as the *standard* sudoku grid. This standard grid is nine cells wide and nine cells high; in addition, it is subdivided into nine interior *blocks* sized 3x3. The rules of Sudoku state that in each column, each row, and each interior block, nine unique symbols must be represented (once per cell).

       The Sudoku puzzle itself is an incomplete grid presented to the player where it is known beforehand that given certain initial clues, a unique solution is logically deducible. The player is then tasked to fill in those remaining cells based on the provided initial clues. The popularity of the game lies in the simplicity of the logic involved in solving Sudoku; however the logic for creating these puzzles can sometimes be more complex.

2. Large Sudoku

Sudoku itself is a subset of a scalable game referred to as Latin Squares. The standard Sudoku grid is 9x9, however without changing the nature of the rules, we can extend the definition of the

Sudoku game to any n^2 x n^2 grid, n=3 being the standard game. However, because larger Sudoku have exponentially larger search spaces (possible configurations of the board), it is not surprising that the nature of the Sudoku creation and solution must be adapted to meet the demands of the larger search space.

Figure 1.

Grid Size	Search Space	Unique Sudoku Boards	Valid Latin Squares	Search Space / Sudoku
4x4	4.29E+09	2.88E+02	5.76E+02	1.49E+07
9x9	1.97E+77	6.70E+21	5.52E+27	2.93E+55
16x16	1.79E+308	5.96E+98	2.736E+129	3.00E+209
25x25	5.15E+873	4.36E+308	-	1.18E+565
36x36	~1.00E+2017	~1.00E+600	-	~1.00E+1417
49x49	~1.00E+4057	-	-	-

Figure 1 illustrates the primary dilemma we face when generating sudoku of increasing size: we must explore an exponentially larger area of the search space to find individual valid Sudoku boards.

3. Creating Sudoku

The most common method to create Sudoku puzzles, and that adopted by this text is to first create a valid Sudoku grid, and then slowly remove cell values. After each removal testing is performed to determine if the grid has a unique solution. When an item is removed and the solution is no longer unique, the item is replaced, and the puzzle is then considered complete.

Thus, given a robust solver, the creation of the puzzle itself is often the element of least concern when creating large Sudoku puzzles. The element removal to generate the puzzle is order O(N^4), and the search space required to find a single valid puzzle is order O(N^N^2). The magnitude difference between these calculations can be easily obscured by overhead when N is small, however when N is large (N>4), the difference is quite apparent.

3.1 Branching Search

Branching search is a method for finding complete Sudoku grids. The generic form of the search is described below:

Step 1.   Place a value in an empty spot in the grid.

Step 2.   Determine how many solutions the grid has.

Step 3.   If possible solutions = 1, the algorithm finishes;

               If possible solutions > 1 we loop back to step 1;

               If possible solutions = 0, we remove the value just placed, and loop back to step 1.

Most forms of this search can be randomized by randomizing the placement value and/or the placement position in Step 1.

3.1.1 Logical Reductions

There are considerable logical reductions we can use to aid our search; however they all rely on the ability to store 'candidate' values at given positions within the grid. For this project, keeping track of this list of candidate values was accomplished using bit masking in C/C++. When we loop back to Step 1 in the search algorithm (Section 3.1), we only choose values which are listed as current candidates for the given position.

At present, there are only four logical rules that can be applied to reduce the candidates in the grid. I have not seen these succinctly listed anywhere; the reductions listed below represent an agglomeration and abstraction of several dozen reduction rules specified by various sources.

Reduction 1 (Subset Reduction):

In an N^2 x N^2 grid, for each Row/Column/Block, if there are K cells which together contain exactly K candidate values, those candidates may be removed from all other cells in that Row/Column/Block. Complexity: $O(N^4 * 2^{N^2})$.

Reduction 2 (Xor Reduction):

In an N^2 x N^2 grid, take any Row or Column denoted RC and a Block denoted B where RC $\cap$ B != $\varnothing$, Let C = RC $\cap$ B. Any candidate values not represented within the set RC-C can be removed from all items in the set B-C. Likewise, any values not represented within the set B-C can be removed from all items in the set RC-C. Complexity: $O(N^5)$.

Reduction 3 (Subset Reduction):

In an N^2 x N^2 grid, for each possible value V, if there are K rows which together contain exactly K possible columns for the value V, then any other rows with candidates of value V in those columns can have the candidates removed. This argument can be applied to Columns as well. Complexity: $O(2^{N+1}*(N^2$ Choose N)) (Note: This reduction is only applicable after Subset Reduction).

Reduction 4 (Heuristic Search):

Individually test branches of the tree using rudimentary tree search methods and determine if any candidates for cells are invalid due to certain chain reactions within the grid when values are placed. Complexity: O((total remaining candidates)^2).

3.1.2 Hierarchical Reduction Strategy

The strategy presented in this project is based on two premises; using these two premises, a robust hierarchical search strategy was developed.

Premise 1. It's difficult to invalidate the grid by placing a value at low levels in the tree search. Conversely, it's easy to invalidate the grid by placing values at high levels in the tree search.

Premise 2. The value of advanced reduction techniques scale as the number of candidates within the whole grid. (As the grid is filled in, the value of advanced techniques diminishes).

Based strictly on premise 1, it was discovered that simply taking one step backward in the tree branching (3.1 Step 3) was not always the most efficient means of exploring the tree. It was realized that if a branch of the search tree contained a 'dead-end', it gave credence that its neighbors too may yield dead-end trees; and rather than exploring all the possible dead ends,

when one is reached it is best to take the largest step backwards up the search tree as possible. However, this need to jump backwards several steps must also be balanced by how close we are to the end of the search tree. The last section of the tree is the hardest to make progress with, and taking unnecessarily large steps backwards at this point in the search would be disadvantageous. To balance the two constraints, the number of steps back is calculated by: $\lfloor$(total candidates remaining in grid) $/ N^4 \rfloor + 1$, note: total candidates on a blank N^2 x N^2 grid are N^6. Therefore, in the early phases of tree exploration, the steps taken back are near N^2, however during the last stages of tree exploration the total candidates remaining are significantly smaller than N^4, and only one step at a time backwards is completed allowing for complete tree exploration when we are nearing a valid Sudoku leaf node.

Based strictly on premise 2, an approach was developed so that when the number of candidates within the grid diminishes as the grid is filled in with values; the more advanced reduction techniques are dismissed, and replaced with simple heuristics. Specifically, reduction techniques: Subset Reduction, Xor Reduction, and Individual Candidate Reduction are used exclusively until the total candidates within an N^2 x N^2 grid are reduced to $N^4 / 2$. When the number of candidates fall below this threshold, the Heuristic Search is applied exclusively.

3.1.3 Termination Conditions

Despite the best efforts of our logical reductions, and the hierarchical reduction strategy, it is often that the sheer size of the tree structures (~1.00E+2017 nodes in a tree describing a 36x36 grid) can overwhelm the algorithm, and it can find itself caught in a local minimum unable to make progress. In our implementation this is denoted by reaching the same tree depth $N^4 / 2$ times. At such a stage, all calculations to that point are dismissed, and the tree search begins again from scratch.

3.2 Permutation

Permutation is a method for 'mixing up' the values of a predetermined complete Sudoku grid. Permutation can be completed several orders of magnitude faster than tree exploration,

however is limited in that the grids it creates are not random, and share many inescapable properties of the grids they were permuted from. Permutation is not seen as a viable method to 'create' Sudoku, rather as a method of reusing/recycling Sudoku to create what may seem to the player like new puzzles.

4. MPI Parallelization

Ideally, we would like all processes to search through the tree at once, and at each iteration have them collude, all of them changing to the position within the search tree of the process which closest to the targeted leaf nodes. However, since our search sometimes requires us to step backwards in the tree, setting the position of a process requires setting not just the current position, but all positions leading to current one, thus allowing a path for backtracking if necessary. However since the data storage for the backtracking path is large, and the implementation we are using is not for a shared-memory system, this ideal scenario will have to be modified.

Since our algorithm has a natural break within it when we switch from advanced reduction techniques to basic techniques (See 3.1.2), we can exploit this in parallel. As the progression of the algorithm crosses this breakpoint, the backtrack path is essentially discarded and reset. This simple property allows us to update the progress of some of the slower processes in parallel.

The way the algorithm is implemented, every 10 seconds each of the child processes communicate with the parent process keeping the parent process up-to-speed on their current activities. If a node passes through the breakpoint between advanced reduction techniques, and heuristic reduction, the process communicates its current position in the search tree to the parent where it is archived for future use. As child processes check in with the parent, if they have not yet advanced to the heuristic reduction, the parent updates them with the search tree position it has on file. The MPI program terminates after any of the child processes check in and report that they have discovered a complete Sudoku grid and have transferred it to the parent.

5. Experimental Results

We attempted to generate four of the large forms of sudoku, 16x16, 25x25, 36x36, and 49x49. Of these types, 16x16 and 25x25 executed too quickly to draw relevant statistics from, and 49x49 grids took an intractable amount of time to generate. Thus leaving us with the 36x36 grid creation to draw statistics from.

Figure 2.

Average Time Required to Create a Unique 36x36 Sudoku Grid Given Various Numbers of Child Processes
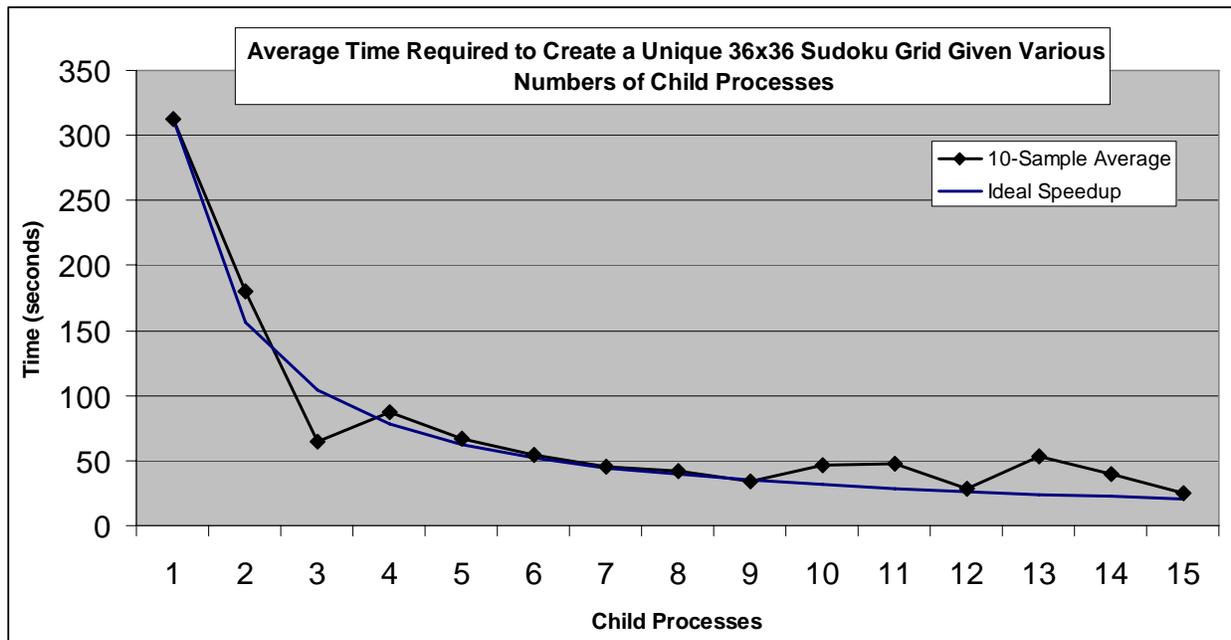
The results presented in Figure 2 suggest that the method used for creating the large Sudoku is fairly close to ideal, however it also suggests that there is a resistance, perhaps an inherent minimum speed is preventing the 10-Sample averages from keeping with the ideal curve. It is possible that the fluctuations toward the right of the graph are due to network overhead, however given the approach we used, it is regarded as highly unlikely.

Both the oscillatory nature of the values on the right side of the curve, and the 'dip' in the left side are attributed to statistical variance. The fact that all the oscillations on the right are to the positive is addressed in the previous paragraph.

6. Conclusion

      Parallel large Sudoku creation turns out to be a domain very similar to many other parallel tree-traversal problems, in this way the exploration of the problem within this paper sheds light on the general implementation of parallel tree searches. The key to the search is that sibling processes must communicate with each other and update those processes which are lagging behind the pack with values from some of the leading processes. If this update of the lagging processes is performed correctly and often, it can add extensibility to the parallel implementation of many tree searches.

References

Bertram Felgenhauer, Frazer Jarvis. *Mathematics of Sudoku I*. 25 January 2006.

Donald E. Knuth. *Dancing Links*. 15 November 2000.

Acknowledgements