6.338 Applied Parallel Computing
Final Report

Parallelizing SAT Solver

With specific application on solving Sudoku Puzzles

Hank Huang
May 13, 2009

This project was focused on parallelizing a SAT solver program with specific application on solving Sudoku puzzles. The project starts with an implementation of a SAT-based Sudoku solver program in Java. The program was re-written in Matlab with reasonable use of the Star-P library for parallelization. Unfortunately, performance of the parallelized program was inferior to the serial version due to excessive communication cost among nodes.

1. Introduction

A SAT problem is not a problem from the Scholastic Aptitude Test famous for college admissions, but rather it is a Boolean Satisfiability problem, a problem of central importance in Complexity Theory in the field of Computer Science. It has a wide range of applications from solving complicated network configurations to proving mathematical theories. It is a decision problem of assigning Boolean values to variables in a Boolean formula in order to make the formula true. However, due to the complexity of the problem, a general SAT problem is difficult to solve by brute force in a short amount of time even for a fast computer.

Fortunately, most applications of SAT problems have relatively small problem space. In addition, advances in algorithms has allowed the development of a few efficient SAT solver algorithms, most notably the DPLL algorithm. The algorithm allows a common SAT problem of reasonable size to be solved within a reasonable time. Moreover, the algorithm allows room for improvements by applying parallelization at various stages. This project will attempt to parallelize the DPLL algorithm and use the improved framework to solve Sudoku puzzles.

2. SAT Problem and DPLL

Generally speaking, a SAT problem is a Boolean formula comprised of a set of Boolean variables connected by basic Boolean operations, AND, OR, and NOT (Sipser, 2006). In the realm of SAT solving, however, a SAT problem usually

exists in conjunctive normal form (CNF) for the sake of allowing finding a solution efficiently (Sipser, 2006). CNF divides a Boolean formula into clauses of literals. Literals can be in positive form or negative form, i.e. X or ~X (not X). Literals are joined by OR operations to form a clause. Finally, clauses are joined together by AND operations to form a formula.

In order for an entire CNF formula to be true, each clause has to be true, and therefore, one or more literals in each clause have to be assigned as true. Because of this property, it was possible for the development of the DPLL algorithm. The DPLL algorithm is a highly efficient, complete, backtracking based algorithm. (Sinz, 2007)

The basic backtracking algorithm runs by choosing a literal, assigning a true value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable. If the simplified formula is satisfiable, then the original formula is satisfiable; otherwise, the algorithm backtracks and assigns the opposite value to the literal to perform the same recursive check. The DPLL algorithm enhances the basic algorithm by performing two additional procedures at each step, unit propagation and pure literal elimination.

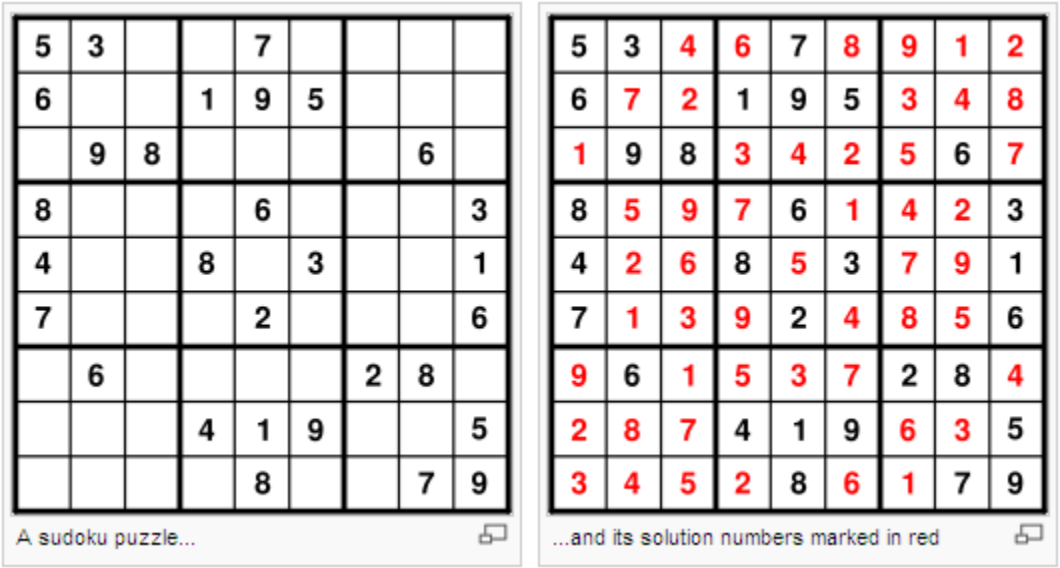
Unit propagation is performed by finding a clause containing a single literal, assigning that literal to be true, and simplifying all clauses accordingly. This often leads to significant reduction of the problem space because any other clauses containing that same literal become true automatically and can be deleted from the problem. Pure literal elimination is performed by keeping track of literals that only exist in positive form or negative form only, but not both. These literals can be set to true and thus forcing the clauses containing them to be true. These clauses can then be deleted from the problem, as they no longer constrain the search space. Finally, a solution is given as a map of literals and their assignments.

3. Sudoku puzzles and SAT

A common Sudoku puzzle is a 9-by-9 square grid consists of 9 3-by-3 sub-grids (figure 1). It is partially filled with numbers ranging from 1 to 9 in no less than 16 cells. To complete a Sudoku puzzle, the following rules must be followed:

- No two same digit can appear in a single column
- No two same digit can appear in a single row
- No two same digit can appear in a single sub-grid
- Exactly one number must occupy each cell

It is essentially a problem of finding a solution under fixed constraints, and therefore, it can be solved by a SAT solver after the problem is translated into a SAT problem.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A sudoku puzzle...

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

...and its solution numbers marked in red

Figure 1: A Sudoku puzzle and its solution.

There are two stages to translating a Sudoku puzzle into a SAT problem. During the first stage, the SAT problem equivalent of an empty Sudoku grid is generated. At first, we must identify all Boolean variables, and then we will form the clauses according to each rule. During the second stage, the partially filled Sudoku cells will be incorporated into the SAT problem as clauses that contain a single variable each.

3.1 Variables and Literals

There will be a total of 9x9x9 = 729 variables. Variables exist as a 9x9x9 integer array, Var, where Var(i,j,k) indicates whether the number k occupies cell(i,j). To make things simple, Var(i,j,k) will be identified as an integer or value i*100 + j*10 + k. A variable appearing in a clause as a positive integer represents a positive literal associated to that variable, and vice versa. For example, “129” indicates the number 9 appears in row 1 column 2 of the Sudoku grid, while “-129” indicates the number 9 does NOT appear in row 1 column 2 of the grid.

3.2 Clauses

Clauses are also represented by integer arrays. For each column, row, and sub-grid, each number from 1 to 9 can appear exactly once. Therefore, two

type of clauses need to be created for each column, row, and sub-grid: an At-Least clause and a series of At-Most clauses. An At-Least clause is comprised of 9 variables as positive literals, while an At-Most clause is comprised of only 2 variables both as negative literals. The combination of both type of clauses will force the "exactly once" property to be true.

Example:

An At-Least clause: [119 129 139 149 159 169 179 189 199].
An associated series of At-Most clause:
[-119 -129], [-119 -139], [-119 -149]…[-119 -199]
[-129 -139],[-129 -139]…[-129 -199]
…
[-189 -199]
These 37 clauses together ensure that "9" appears exactly once in row 1.

Moreover, each cell must be occupied by exactly one number from 1 to 9. Similarly, an At-Least clause and a series of At-Most clauses must also be created for each cell. As a result, at least a total of 3996 (9*37 + 9*37 + 9*37 + 81*37) clauses must be created for an empty Sudoku puzzle. Finally, a unit clause containing exactly one variable is created for each pre-filled cell in the grid.

3.3 SAT Problem

The SAT problem is an array of all the of clauses mentioned in section 3.2

4. Parallelizing SAT Solver

Since the DPLL algorithm is backtracking based, at the beginning of each step, a clone of the SAT problem is created to allow for backtracking. Within each recursive step of the DPLL algorithm, at least one of three potential operations happens. The first two operations, unit propagation and pure literal elimination, are performed when conditions permit; otherwise, the algorithm chooses a literal arbitrarily or intelligently, sets this literal to true, simplifies the clauses in serial, and solves the reduced problem recursively.

The algorithm simplifies clauses by first deleting the clause containing the chosen literal and then searching through the rest of the clauses to remove any clause that contains the same literal while deleting the negation of the chosen literal from clauses:

(X Y Z)(~X ~Y)(X W V ~Y)(Y ~Z) → choose X → removing clauses containing X → (~X ~Y)(Y ~Z) → removing ~X from clauses → (~Y)(Y ~Z)

As illustrated, the operation is performed serially. Moreover, both unit propagation and pure literal elimination are performed in serial as well by iterating over the collection of clauses. Unit propagation is performed when a unit clause is found in the set of clauses. Pure literal elimination is performed when a pure literal prevails in the set of clauses. Since these operations are done at each recursion step of the algorithm, parallelizing these operations can theoretically speed up the performance of DPLL.

Example:

Unit Propagation:

(X)(X v Y v Z)(~X v Y)(Y Z ~W) (~Z W) →(Y)(Y Z ~W)(~Z W)

Here, the variable X appears in a unit clause. Setting X to true will force the unit clause to be true altogether as well as the adjacent clause. Thus, both clauses can be deleted from the collection of clauses. Moreover, ~X is now known to be false and thus no longer constrains the third clause; ~X is removed from that clause.

(Y)(Y Z ~W)(~Z W) → (~Z W)

Similarly, the unit clause of variable Y *propagates* outward to the other clauses. Unit propagation is performed until no more unit clause exists in the collection of clauses. In this case, assigning W to true as a final step will satisfy the original problem. The solution is X, Y, W = true.

Pure Literal Elimination is performed in a similar way. A pure literal is selected and set to true, and all clauses are simplified in the same way as they are in unit propagation.

As an effort to parallelize these three intermediate operations in DPLL, these operations were re-written to accept two arguments, a literal and a set of clauses. Moreover, the collection of clauses were initially created and stored across different nodes in a cluster of machines. As a result, at each step of the recursion, each method is performed in parallel on different machines simplifying their own subset of clauses according to the literal argument.

5. Performance

Performance is measured by running 3 programs on 6 different but most difficult Sudoku puzzles found online. The first program is the original implementation in Java. The second program was the translated implementation in Matlab. Both were run on my personal machine, which has a 2.67GHz Intel Core 2 Duo E6750 processor and 2GB of RAM. The third program is the parallelized version in Matlab, and it was run on the Beowulf cluster (beowulf.csail.mit.edu) with 16 compute nodes. Each compute node has two 2.40GHz Intel Xeons and 2GB of RAM, while the front end has four 2.40GHz Intel Xeon processors and 3.5GB of RAM.

Unfortunately, the benefit of parallelization did not realize in this case (see Table 1). Both serial programs in Java and Matlab performed comparably. However, the parallel version of the program ran significantly slower than its serial counterpart. A disadvantage of parallelization that caused this slow-down was the expensive cost of communication.

At each recursive step of unit propagation, the algorithm first searches the collection of clauses spread across the nodes to find the unit clause. It then arbitrarily picks one and asks each node to perform unit propagation with the literal in that clause, and repeats until no unit clause is found. The back and forth communication was more costly then computation itself. For pure literal elimination, it was even more costly for the algorithm to keep track of which literals became pure.

Puzzle Number	Number of Recursion Steps	Performance in milliseconds		
		Java	Matlab Serial	Matlab Star-P
1	145	76	89	4,105
2	219	88	101	3,809
3	732	148	168	5,672
4	2,644	465	481	12,691
5	4,052	858	912	19,780
6	8,234	1,485	1,674	24,536

Table 1: Performance comparison

6. Conclusion

On average, a Sudoku puzzle, with extra clauses added to constrain the search space for better performance, was only comprised of about 12,000 clauses. The size of the problem did not justify for the cost of communication among nodes, as computation required was rather light. Had the SAT solver been implemented to solve a different, larger problem with significantly more clauses than a 9x9 Sudoku puzzle, parallelization could theoretically save more time.

References

Sinz, C. (2007). Visualizaing SAT Instances and Runs of the DPLL Algorithm. *Journal of Automated Reasoning , 39* (2), 219-243.
Sipser, M. (2006). *Introduction to the Theory of Computation.* Boston: Thomson Course Technology.