27 Feb 1998

Parallel Genetic Algorithm to Solve the Satisfiability Problem

Nicole Nemer-Preece

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

Recommended Citation

N. Nemer-Preece and R. W. Wilkerson, "Parallel Genetic Algorithm to Solve the Satisfiability Problem," *Proceedings of the ACM Symposium on Applied Computing*, pp. 23 - 28, Association for Computing Machinery, Feb 1998.
The definitive version is available at https://doi.org/10.1145/330560.330565

PARALLEL GENETIC ALGORITHM TO SOLVE THE SATISFIABILITY PROBLEM

Nicole Nemer-Preece and Ralph Wilkerson
Computer Science Department
University of Missouri-Rolla
18 Miner Circle
Rolla, Missouri, 65409
nnemer@umr.edu and ralphw@umr.edu

Keywords: parallel processing, genetic algorithms, satisfiability problem, David-Putnam method, model generation

ABSTRACT

This paper offers a parallel genetic algorithm solution to the satisfiability problem. It combines components of the Davis-Putnam method and genetic algorithms for the solution. This solution is useful in the areas of theorem proving, constraint satisfaction programming, and VLSI design. The algorithm is implemented and run on a Paragon. The results show performance improvement by increasing the number of nodes. Two parallel methods are compared: one that implements interprocessor communication and one that does not. The results show performance improvement with the method that uses interprocessor communication.

INTRODUCTION

The satisfiability problem (SAT) is determining whether or not a logic formula can be evaluated *true* for some assignment of the logic variables in that formula. This problem finds much use in the areas of theorem proving, constraint satisfaction programming, and VLSI design.

In propositional logic, an atom A is a logical variable that has a value of either *true* or *false*, 1 or 0. An atom or its negation is called a literal. A clause C is a disjunction of literals. For example: $C1 = A \vee B \vee \neg C$. C1 is *true* if and only if at least one of A, B, or -C is *true*. Otherwise C1 is *false*. A formula F is a conjunction of clauses. For example: $F1 = C1 \wedge C2 \wedge C3$. F1 is *true* if and only if C1 is *true* and C2 is *true* and C3 is *true*. An assignment of *true* or *false* to the variables in a formula is called an interpretation.

A formula is satisfiable if and only if there exists an interpretation for the formula that will make the formula *true*. Otherwise the formula is unsatisfiable. If the formula is satisfiable, then the interpretation that satisfies the formula is called a model. Model generation is one method of solving the SAT problem. It entails generating an interpretation for the formula that makes the formula *true*.

The SAT problem is NP hard, which means that the time required for the solution is exponential based on the size of the input. The size of the input for a SAT problem is the number of variables in the clauses of the formula. For a formula with n variables, there are 2^n interpretations for that formula. One method of finding a model for a formula is with a truth table.

Determining whether F has a model is trivial with the truth table. Examining the final column of the table can determine the satisfiability of the formula. If at least one row has the value of *true*, then the formula is satisfiable. If every row in the table evaluates to *false* in the final column of the truth table, then the formula is unsatisfiable. This solution is easy to understand. However, it would take more than a lifetime to compute on the fastest computer if the number of atoms is large.

Completeness and soundness are characteristics of a search method. The truth table solution is an example of a solution that is complete. Completeness means that given an unsatisfiable formula, the search for the solution can determine that the formula is unsatisfiable [WOLB92]. The truth table solution is also sound. Soundness means that no unsatisfiable formula will be found satisfiable with this method [WOLB92]. Another sound and complete solution for the SAT problem is the Davis-Putnam Method [DP60].

DAVIS-PUTNAM METHOD

Using several rules, the Davis-Putnam Method reduces the search space until it is empty (the formula is satisfiable) or a contradiction occurs (the formula is unsatisfiable). If a clause has only 1 variable in it, the Rule for Elimination of One-Literal Clauses can be used. It will assign the variable in the clause a value to make the clause true and then that clause can be removed from the formula [D60] If a variable occurs only in the positive/negative in all of the clauses, then The Affirmative-Negative Rule can be used. This rule assigns that variable *true* / *false*. Then all clauses that contain that variable can be removed from the formula [DP60]. The Rule for Eliminating Atomic Formulas (also called the Split Rule) is a process of choosing a single variable and then assigning that variable the value of *true* and assigning that variable the value of *false* and determining the satisfiability of the formula in each case [DP60].

There have been many implementations of the Davis-Putnam Method [CF90, McC94, SM95, ZS94]. One of these was developed at Argonne National Labs. It is called

ANL-DP[McC94]. MACE is a front end to ANL-DP that allows the input to be in first-order form. MACE uses clauses produced from OTTER, a high performance theorem prover, to generate propositional clauses to port into ANL-DP. MACE converts the first-order form logic into CNF (conjunctive normal form is a formula that is a conjunction of clauses where each clause is a disjunction of literals) for ANL-DP. ANL-DP then uses the Davis-Putnam Method to search for a model of the formula. This method has been used to solve various mathematical applications from combinatorial logic, group theory, ring theory, and ternary boolean algebra [McC94].

GENETIC ALGORITHMS

Random search is an example of a search technique that is sound but not complete. One example of a guided random search is a genetic algorithm. A genetic algorithm(GA) is a search method that may reduce the time required to solve complex problems that are computationally intractable. It is based on Darwin's ``survival of the fittest" idea. This method searches for a solution by allowing candidates for the solution to evolve from the best candidates seen so far. There are several characteristics of a genetic algorithm that are defined by parameters. There are no set values for these parameters. Design of the GA includes manipulating the parameters to find a better solution. Because the problem being solved is the satisfiability problem the strings consists of 0's and 1's.

Parent selection is the process of determining which strings will be used to create the new offspring. Reproduction will create new offspring with 2 parent strings. This paper uses a uniform crossover and random bits crossover. For random bits crossover, a random number of bits are chosen to be flip bits. The first offspring is a duplicate of the first parent and the second offspring is a duplicate of the second parent. Then in the flip bit positions, the two offspring swap the values of the bits in those positions. Survival selection is the process of determining which of the current generation and the newly created offspring will live on the next generation. This selection must ensure that at least some of the new offspring will live to ensure evolution.

The process of parent selection, reproduction, and survival selection continues until a solution is found or until the number of generations exceeds a maximum number of generations. Because this search is random, it in not complete. If the process continues without finding a solution, that does not mean that there is no solution. The very next offspring pool may generate a solution. Additionally, if there is no solution, this method does not determine that either. It will continue generating populations, searching for a solution, with no way of determining that a solution does not exist.

Algorithm Description

A hybrid of Davis-Putnam rules and genetic algorithms is used to determine satisfiability in this work. Two components of the Davis-Putnam procedure will be used and combined with the genetic search. A typical formula has 45 to 512 atoms. Generating a truth table for problems of these sizes would take 10 hours to 1.2e129 years assuming it takes 1 nanosecond to generate one row in the truth table.

Before starting the genetic algorithm search for a model, some of these atoms are classified as ``unchangeable". If an atom occurs only in the positive form, or only in the negative form, then that atom is assigned a value of *true* or *false* respectively, and marked as unchangeable. This action implements the Affirmative-Negative Rule from Davis-Putnam [DP60]. If a clause has only a single literal in it, then that atom is marked as unchangeable and assigned a value to make that clause *true*. This action implements the Rule for Elimination of One-Literal Clauses from David-Putnam [DP60]. The values for the atoms that are not marked as unchangeable are then determined by the GA search. The search space is reduced right from the start.

For the satisfiability problem, the string with the best fitness will be the string that makes the most number of clauses *true*. So if the number of clauses in a formula is M, then a solution string will have a fitness of M. The initial population will be randomly generated and the number of strings in the initial population is 20. The selection method will be the roulette style selection. Fifteen (15) unique parents are selected with this method. Each parent is paired with the best string do far. Each pair reproduces two offspring. The crossover method is the uniform crossover. A new mask is created for each pair-wise reproduction. If the fitness of the selected parent is above a parameterized cutoff point then the random bits crossover is used. This method is used to attempt to slightly change the string to avoid changing the candidate too much and moving away from a solution. With fifteen pairs of parents, 30 offspring are generated. Upon deriving each string, the corresponding fitness is calculated and stored. If one of these newly generated offspring has a fitness of M then the process can terminate with a solution.

Survival selection will ensure that at least some of the new offspring live on to the next generation. The size of the survival population is 30. The top 10 best fit offspring strings survive to the next population. The remaining 20 to survive to the next population are the 20 most fit among the current population and the offspring combined (not including the offspring that have already been chosen for survival). In this process, there are no duplicates to survive to the next generation. After the entire survival population has been determined, it becomes the current population for the next cycle.

This evolution continues until a solution is found (a string that has a fitness M) or until a maximum number of generation has been created. The pseudocode for the process is given below:

```
Algorithm GADP
n := N - Find_Unchangeables
solved := Create_Initial_Population
gencount := 0
WHILE ( ! solved AND gencount < MAXGENS)
        Select_Parents
        solved := Reproduction
        if (solved)
                break;
        Select_Survival
        increment gencount
ENDWHILE
if (solved)
        Output_solution
```

The procedure Find_Unchangeables marks the variables that are identified with the first 2 rules of the Davis-Putnam method as unchangeable. The number of these unchangeable variables is returned from this function. Therefore, instead of a candidate string being of length N, it is reduced to length n, which is N minus the number of variables that are marked as unchangeable. Create_Initial_Population will randomly generate 20 strings. For each, the fitness value is calculated. If one of these strings is a solution, Create_Initial_Population returns TRUE; otherwise it returns FALSE. This flag value is stored in the variable solved. Select_Parents generates a list of fifteen (15) unique parents that will be used in the reproduction process. Reproduction will perform the crossovers with each parent that was selected paired with the best string so far. Two offspring are created for each crossover, 30 offspring in total. Like Create_Initial_Population, Reproduction returns a TRUE / FALSE for the flag solved.

```
Algorithm Reproduction
solved := FALSE
FOR each selected parents AND NOT solved
        solved := One_Crossover

Algorithm One_Crossover
Create_Mask
Generate_Both_Offspring
Perform_Mutation_Function
Calculate_Fitness(offspring1)
Calculate_Fitness(offspring2)
IF (either offspring1 or offspring2 is a solution)
        solved := TRUE
```

Reproduction will call One_Crossover for each parent that has been selected for reproduction. If One_Crossover returns TRUE (meaning that a solution has been found) then the remaining pairs of parents do not perform the reproduction and Reproduction will return TRUE.

One_Crossover will perform reproduction for one parent pair. It will use uniform crossover unless the selected parent has a fitness above cutoff; then it will use random bits crossover. Two offspring are generated. Each bit of each string may be mutated with a mutation rate of 0.001. The fitness of each of the offspring is calculated. If either of the offspring is a solution, then One_Crossover will return TRUE; otherwise it returns FALSE.

Table 1 gives a description of the input files used in these tests. They are the first order input files that are packaged with MACE. Table 2 shows the results of running GADP on these input files. The Atoms column shows the number of variables in the problem that are not marked as unchangeable by Find_Unchangeables and the total number of variables. The Gens column is the number of generations required to find a sequential solution. For the tough-nut.in input problem, there is no solution found for the formula is unsatisfiable. In this situation, GADP stops because of the MAXGENS requirement. MAXGENS is 5000 generations for these sequential test.

Input Problem	Description
cl_sw	combinitorial logic
ordered	group theory
ring	ring theory
ring_unit	ring theory
tba	ternary boolean algebra

Table 1. Input Problem Descriptions

Input Problem	Atoms C / T	Clauses	Gens
cl_sw	30 / 45	9728	1224
ordered	64 / 96	7728	1686
ring	74 / 164	14966	4847
ring_unit	52 / 168	5671	4692
tba	44 / 105	8688	3584

Table 2. Summary of Sequential Results

PARALLELIZATION

In parallelizing this algorithm, the third component of the Davis-Putnam method is implemented, the Split Rule. Given 2^p processors, the top p highest ranking variables that have not been marked as unchangeable are determined. The ranking used on the variables is based on the Jeroslow-Wang Rule [HV95]. Each variable x_i is assigned a value JW where $JW = \sum_{j=1}^{NC} 2^{-n_j}$ where x_i occurs in clause j, NC is the number of clauses in the formula, and n_j is the number of variables in clause j. The ranking is based on this JW value. To enumerate the possible interpretations for these p variables, there are 2^p interpretations. Each processor will be assigned one of the interpretations to these p variables with the highest JW values. These assigned variables are then marked as unchangeable. The search space is logarithmically reduced.

Broadcasting and Nonbroadcasting Versions

This work implements two versions of this parallel algorithm. One version, Broadcasting, allows each processor to communicate with all of the other processors after every increment generations. For these tests, increment equals 20. The other version, Nonbroadcasting, does not allow the interprocessor communication. In the Broadcasting version, each processor will broadcast its best fit candidate so far to all of the other processors. Therefore each processor will receive p - 1 new strings to consider. The pseudocode for GADP with Broadcasting is given below:

```
Algorithm GADP-B(N is number of logic variables in the
CNF formula; and there are 2^p processors in the
environment)
n := N - Find_Unchangeables
p := Mark_Jeroslow-Wang_Variables
n := n - p
Assign_Jeroslow-Wang_Variables(p)
solved := Create_Initial_Population
gencount := 0
WHILE ( ! solved AND gencount < MAXGENS)
        Select_Parents
        solved := Reproduction
        if (solved)
                break;
        Select_Survival
        IF mod(gencount,increment) == 0
                Do_Broadcast
                solved:=Add_New_Strings
        if (solved)
                break;
        increment gencount
 ENDWHILE
 if (solved)
        Do_Broadcast
        Output_solution
```

Mark_Jeroslow-Wang_Variables will determine the p highest ranking variables based on the Jeroslow-Wang rule. Assign_Jeroslow- Wang_Variables will assign a value to those variables based on the current processor number. Because there are 2^p processors, each will have a unique assignment to give to the p variables. Do_Broadcast will perform the broadcast that will allow each processor to send its best fit string to all of the other processors. Add_New_Strings determines the fitness of each of these newly received strings and for each string it determines if that string is better than the worst string in the current population. If so, that new string replaces the string in the current population. Duplication of strings is avoided. If a new string is a solution, then Add_New_Strings will return TRUE; otherwise it returns FALSE.

The Nonbroadcasting version does not allow each processor to communicate with the other processors. Each processor works independently towards a solution without sharing information with other processors. If a processor finds a solution string, it will inform all of the other processors that a solution has been found. Every increment generations, each processor will check to see if such a transmission has occurred. The pseudocode for GADP with No Broadcasting is given below:

```
Algorithm GADP-NB(N is number of logic variables in the
CNF formula; and there are 2^p processors in the
environment)
n := N - Find_Unchangeables
Mark_Jeroslow-Wang_Variables(p)
n := n - p
Assign_Jeroslow-Wang_Variables(p)
solved := Create_Initial_Population
gencount := 0
WHILE ( ! solved AND gencount < MAXGENS)
        Select_Parents
        solved := Reproduction
        if (solved)
                break;
        Select_Survival
        IF mod(gencount,increment) == 0
                found:=Check_Solution
        if (found)
                break;
        increment gencount
ENDWHILE
if (solved)
        Inform_Others
        Output_solution
```

At Check_Solution each processor will probe the receive buffer to see if any other processor has found a solution. This function will return TRUE or FALSE. When a processor does find a solution, it will Inform_Others so that the other processors can stop searching.

RESULTS

The results for the parallel versions are given in Tables 2, 3 and 4. Each test is run 15 times. The PEs columns tells the number of processors used for that test. The Best Gens tells the number of generations required to find a solution for the best run of the 15 tests. The Worst Gens column tells the number of generations required to find a solution for the worst of the 15 test and Avg Gens is the average number of generations required for the tests that found a solution. The No Sols column states the number of tests that did not generate a solution within MAXGENS generations. MAXGENS is 1000 for these parallel tests.

Input Problem	PEs	Best	Worst	Avg	No Sols
cl_sw	32	10	46	23.9	0
	16	14	80	36.3	0
	8	20	136	45.9	0
	4	4	723	197.5	0
ordered	32	5	100	33.9	0
	16	40	80	54.7	0
	8	52	118	79.5	0
	4	42	177	109.3	0
ring	32	80	760	316.4	5
	16	226	760	482.9	5
	8	150	760	530.9	7
	4	77	978	602.4	8
ring_unit	32	60	193	99.8	3
	16	73	380	188.1	3
	8	69	800	254.8	6
	4	180	536	326.7	6
tba	32	40	860	313.2	2
	16	28	500	231.4	4
	8	32	500	242.2	6
	4	20	660	421.3	9

Table 3. Parallel Results with Broadcasting

Table 3 summarizes the results for the implementation of GADP-B, the version h interprocessor communication every increment generations. The results for this parallel broadcasting implementation has better performance over the sequential implementation. For each of the input files (not including the tough-nut problem), the parallel version finds a solution in fewer generations than the sequential version. The factors of improvement for the different input files range from 6 to 15 with 4 processors. The parallel version with 4 processors for cl_sw is 6 times better than the sequential version and the parallel version with 4 processors for ordered_semi is 15 times better than the sequential version. This is better than linear speedup with this sequential version.

Within these parallel broadcasting tests, there is improvement with an increase in processors. In all cases except for one, the average number of generations required to find a solution decreases with an increase in the number of processors. With the tba input file when the number of nodes increases from 16 to 32, the average generations also increases. This is unexpected. In one case, the improvement is considerable. With the cl_sw input file, when the number of nodes increases from 4 to 8 (a factor of 2) the average generations decreases by more than 50%.

This parallel broadcasting version shows improvements by increasing the number of processors in the No Sols column as well. In most cases, the number of tests that do not find a solution decreases with an increase in processors. There are several cases where the increase in processors does not effect the number of tests that do not find a solution. Not a single case finds an increase in the number of tests that do not find a solution with an increase in the number of processors. This is expected. The Best Gens and Worst Gens columns are not as informative for there is no a general trend in the performance as the number of processors increases.

Table 4 summarizes the results for the implementation of GADP-NB, the version with no interprocessor communication. The results for this parallel nonbroadcasting implementation has better performance over the sequential implementation. For each of the input files (not including the tough-nut problem), the parallel version finds a solution in fewer generations that the sequential version. The factors of improvement for the different input files range from 4 to 14 with 4 processors. The parallel version with 4 processors for cl_sw is 4 times better than the sequential version and the parallel version with 4 processors for ordered_semi is 14 times better than the sequential version. This is better than linear speedup with this sequential version.

Within these parallel nonbroadcasting test, in all cases the average generations required to find a solution decreases with an increase in number of processors. Again with the cl_sw input file going from 4 to 8 processors, the improvement is considerable as the average generations decreases by more than 50%.

As with the broadcasting version, there is a decrease in the number of tests that find no solution with an increase in number of processors in most cases. There are several where the increase in number of processors does not effect the number of tests that find no solution.

Comparing table 3 with table 4 shows expected improvement for the broadcasting version over the nonbroadcasting version. In every case, the average generations required to find a solution is better for the broadcasting version than the nonbroadcasting version. In every case except one, the number of tests that find no solution is lower for the broadcasting version than the nonbroadcasting version. With the tba input file and 4 processors, the two versions have the same number of tests that fail to find a solution. These results show the enhanced performance that is given for the broadcasting version, despite the overhead of the interprocessor communication.

InputProblem	PEs	Best	Worst	Avg	No Sols
cl_sw	32	3	161	52.9	0
	16	16	292	80.2	0
	8	11	374	106.1	0
	4	19	632	260.1	1
ordered	32	38	132	69.2	0
	16	45	199	103.6	0
	8	60	227	108.3	0
	4	31	362	119.5	0
ring	32	121	710	415.8	10
	16	249	674	449.8	10
	8	309	969	620.0	12
	4	899	949	924.0	13
ring_unit	32	103	671	311.3	5
	16	99	576	315.5	7
	8	218	801	534.9	8
	4	152	851	589.1	8
tba	32	56	919	335.3	7
	16	162	887	417.7	9
	8	103	854	512.5	9
	4	128	898	550.7	9

Table 4. Parallel Results with Non Broadcasting

CONCLUSIONS

This work has shown an increase in performance with the parallel version, especially the broadcasting version. Applying the Davis-Putnam rules before the genetic algorithm reduces the search space logarithmically. The parallel versions have super linear speedup over this sequential implementation. In most cases, an increase in processors decreases the average number of generations required to find a solution and decreases the number of tests that do not find a solution. In every case, the broadcasting version outperforms the nonbroadcasting version for the average number of generations required to find a solution and (with the exception of 1 case) for the number of tests that do not find a solution.

REFERENCES

[CF90] Chen, W. and Fang, M. Theorem Proving in Propositional Logic on Vector Computers Using a Generalized Davis-Putnam Procedure. *Proceedings of Supercomputing 1990*, pp. 658 - 665, 1990.

[DP60] Davis, M. and Putnam, H. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, Vol. 7, No. 3, pp. 201 - 215, 1960.

[Gol89] Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, Massachusettts: Addison-Wesley Publishing Company, Inc., 1989.

[HV95] Hooker, J. N. and Vinay, V. Branching Rules for Satisfiability. *Journal of Automated Reasoning*, Vol. 15, No. 3, pp. 359 - 383, 1995.

[McC94] McCune, W. A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical Report, Argonne National Laboratory, 1994.

[Shu96] Shumsky, Olga. New Methods in Finite First-Order Model Search. Masters Thesis, University of Missouri-Rolla, Rolla, Missouri, 1996.

[WO92] Wos, Larry; Overbeek, Ross; Luck, Ewing; and Boyle, Jim. *Automated Reasoning: Introduction and Applications*. New York: McGraw-Hill, Inc., 1992.

[ZS94] Zhang, H. and Stickel, M. E.. Implementing the Davis-Putnam Algorithm by Tries, 1994.