RIACS

Research Institute for Advanced Computer Science
NASA Ames Research Center

Satisfiability Test with Synchronous Simulated Annealing on the Fujitsu AP1000 Massively-Parallel Multiprocessor

Andrew Sohn
Rupak Biswas

RIACS Technical Report 96.07 **March 1996**

To appear in the *10th ACM International Conference on Supercomputing*, Philadelphia, Pennsylvania, May 25–28, 1996

Satisfiability Test with Synchronous Simulated Annealing on the Fujitsu AP1000 Massively-Parallel Multiprocessor

Andrew Sohn
Rupak Biswas

The Research Institute of Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 212, Columbia, MD 21044, (410) 730-2656

Work reported herein was supported by NASA via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035-1000.

Satisfiability Test with Synchronous Simulated Annealing on the Fujitsu AP1000 Massively-Parallel Multiprocessor

Andrew Sohn
CIS Dept., New Jersey Institute of Technology, Newark, NJ 07102, sohn@cis.njit.edu

Rupak Biswas
RIACS, NASA Ames Research Center, Moffett Field, CA 94035, biswas@riacs.edu

ABSTRACT

Solving the hard Satisfiability Problem is time consuming even for modest-sized problem instances. Solving the Random L-SAT Problem is especially difficult due to the ratio of clauses to variables. This report presents a parallel synchronous simulated annealing method for solving the Random L-SAT Problem on a large-scale distributed-memory multiprocessor. In particular, we use a parallel synchronous simulated annealing procedure, called Generalized Speculative Computation, which guarantees the same decision sequence as sequential simulated annealing. To demonstrate the performance of the parallel method, we have selected problem instances varying in size from 100-variables/425-clauses to 5000-variables/21,250-clauses. Experimental results on the AP1000 multiprocessor indicate that our approach can satisfy 99.9% of the clauses while giving almost a 70-fold speedup on 500 processors.

1 INTRODUCTION

The Satisfiability (SAT) Problem refers to finding a truth assignment of variables which evaluates clauses to true [3,4]. Applications of the SAT Problem can be found in numerous areas, including computational complexity, graph coloring, logic, operations research, and artificial intelligence. The SAT Problem consists of a set of variables $v_1, v_2, \ldots, v_n$, a set of clauses $C_1, C_2, \ldots, C_m$, and operators $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT). A variable can either be true or false. A variable or the negation of a variable is called a literal. A clause consists of one or more literals. A clause with two literals such as $C = a \vee b$, will be true if at least one of the literals is true. The SAT Problem is to determine if there is an assignment of variables that evaluates the formula $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ to true. The Maximum Satisfiability (MAX-SAT) Problem refers to finding a truth assignment of variables such that the number of true clauses is maximized [7].

The SAT Problem with a small number of variables and clauses such as a 10-variable/20-clause instance with three variables per clause can be tested for satisfiability with reasonable effort. However, the task becomes challenging when the problem size increases to several hundred variables. It is not straightforward to find if there is an assignment that can satisfy all the clauses, or that maximizes the number of true clauses. What is even more interesting is that there is a class of problems, called Random L-SAT Problems, which are extremely difficult to evaluate and require long execution times. The report by Mitchell et al. [13] indicated that when the ratio of clauses to variables is approximately 4.25 for clause length 3, the problem becomes extremely difficult to solve. When the ratio is higher, the problem instances are not solvable. Kirkpatrick and Selman [10] explained that there exists a threshold value for a given SAT Problem that can predict the fraction of unsatisfiable clauses.

Many methods have been developed to date to solve the SAT Problem and tend to be heuristic and greedy in nature [5,7,13,14]. Franco and Paull [5] analyzed the probabilistic performance of solving randomly-generated SAT instances. Selman and Kautz [14] found that local greedy search was successful in solving some of the Random L-SAT Problem instances. Spears [17] found that simulated annealing can be an efficient method to solve the Random L-SAT instances with approximately a 50% success rate. The major drawback of these methods is the long execution time. Studies have shown that most Random L-SAT Problem instances require several hours, and often a few days, on single-processor sequential machines [17]. These long execution times clearly indicate that testing the SAT Problem on a sequential machine can be intolerable, and often impractical, for realistically-sized problem instances. It is precisely the purpose of this work to investigate the possibility of solving the SAT Problem on parallel machines.

We solve Random L-SAT Problem instances using parallel synchronous simulated annealing [16] on a large-scale distributed-memory multiprocessor. Simulated annealing is chosen because it is known to provide high solution quality for combinatorial optimization problems. We choose a synchronous method as opposed to an asynchronous one because it preserves the convergence property of simulated annealing which, in turn, can yield high-quality solutions. While synchronous simulated annealing is inherently sequential due to its dependence between iterations, we introduce an efficient parallel technique, called Generalized Speculative Computation, which can execute P different iterations in parallel on P processors. In other words, this parallel method preserves the convergence property, solution quality, and decision sequence of sequential simulated annealing. It provides exactly the same solution but much more quickly by using massively-parallel machines. This fast turnaround

To appear in the 10th ACM International Conference on Supercomputing, Philadelphia, PA, May 25-28, 1996.

would allow practitioners to accurately examine the behaviors of various SAT Problem instances in a short time.

2 SIMULATED ANNEALING FOR THE SAT PROBLEM

Simulated annealing (SA) is one of the most efficient methods for solving combinatorial optimization problems [9]. Given the search space, the method attempts to find the global minimum state. The motivation behind developing such a search method is based on the analogy the way metals cool and anneal as their temperatures decrease. A typical implementation of SA consists of two nested loops. The outer loop controls the temperature based on the annealing schedule. The inner loop performs the three steps of evaluate, decide, and modify for the given temperature. The evaluation step suggests the next possible state by generating random numbers and evaluates it by using some criteria. The decision step decides whether the suggested state is acceptable. If the suggested state is indeed acceptable, the modification step updates the data according to the specifications of the evaluation step. The sequential SA technique as well as its parallel versions have been applied to various optimization problems [6], including VLSI cell placement [1,2,11], the Traveling Salesman Problem [16], the Satisfiability Problem [17], and task assignment problems [18].

Figure 1 shows our implementation of sequential SA for the SAT Problem. While SA is usually implemented as two nested loops, our version uses three nested loops to detect those rare configurations at low temperatures. The outermost loop is a trial loop which changes the decay rate, the rate at which the temperature is lowered. The middle loop is a temperature loop which lowers the temperature based on the decay rate. The innermost loop executes the three steps of evaluate, decide, and modify, for the given temperature.

```
T = T0;                                          // initial temperature
CS = 1;                                          // decay rate
for i = 1 to ntry                                // # trials
    for j = 1 to nclause                         // # clauses
        for k = 1 to nvariable                   // # variables
1:          ΔN = evaluate(CLAUSE, i, j, k);
2:          flag = decide(ΔN, T, ε);
3:          CLAUSE = modify(CLAUSE, flag);
        T = T * (1.0 − CS * (1.0/(j + 10.0)));
    CS = 0.9 * CS;
```

Figure 1: Sequential simulated annealing.

The evaluation step in line 1 of Fig. 1 randomly selects a variable to flip. By pretending to flip the variable, the step evaluates ΔN, the difference between the previous true clauses and the new true clauses. The decision step in line 2 decides if the new solution is acceptable. It generates a *flag* which is set either to OK (accept) or NOK (reject). This step uses the Metropolis equation [12] in order to decide. The *flag* is OK if $\Delta N > \epsilon$ and NOK if $\Delta N < -\epsilon$, where ϵ is a predefined error threshold (we chose $\epsilon = 10$). For $-\epsilon \leq \Delta N \leq \epsilon$, *flag* is OK if $1/(1 + e^{-\Delta N/T}) > r$, where $0 < r < 1$. If *flag* is OK, the modification step in line 3 actually flips the variable, which appears in several clauses, resulting in new clauses. If *flag* is NOK, no data modification takes place.

For Random L-SAT Problem instances where the clause-to-variable ratio is 4.25 for a clause length of 3, a variable appears in about 30 clauses. The modify step therefore updates approximately 30 clauses if the decision is accepted. The algorithm in Fig. 1 terminates either when all the clauses are satisfied or all the iterations are exhausted. For our implementation of the SAT Problem, the outermost loop is set to $ntry = 10$ trials. This allows us to change the decay rate, enabling us to detect rare configurations of the search space at the tail of SA.

The SA algorithm shown in Fig. 1 is highly sequential because of two reasons. First, the innermost loop has a loop-carried dependence. Iteration k may modify the data, based on which iteration $k+1$ must proceed. Second, each k iteration has true data dependencies. The three steps within each iteration are executed sequentially. It is apparent from Fig. 1 that line 2 uses the ΔN computed in line 1. Line 3 uses the result of line 2. It is thus not possible to execute individual k iterations simultaneously. In the next section, we describe how this innermost loop can be executed in parallel to a certain extent by using a method called speculative computation.

3 GENERALIZED SPECULATIVE COMPUTATION

This section describes how we parallelize SA for solving the hard SAT Problem. We generalize binary speculative computation using an n-ary speculative tree, which we call Generalized Speculative Computation (GSC). A detailed example is presented to demonstrate how the new method is mapped to a multiprocessor.

3.1 THE IDEA

GSC uses an n-ary tree. Figure 2 shows an n-ary speculative tree with three levels. Note that the speculative tree is dynamically determined at runtime and is unrelated to the interconnection network topology of the machine. Each processor receives an unique k index at runtime and performs the three steps: modify, evaluate, and decide. It then sets a flag to indicate its decision. The lowest-numbered loop index returning an accept decision is selected to initiate the next level. To ensure that the parallel version generates the same decision sequence as sequential SA, we use an identical sequence of seeds to generate random numbers. There is a total of $ntry \times nclause \times nvariable$ seeds. A loop index $[i, j, k]$ used as a seed, therefore, generates a unique random number for selecting a variable. Figure 2 shows a total of 11 (not 16) iterations executed in three levels, as explained below.

The first level of the n-ary speculative tree shows seven processors executing seven iterations simultaneously. P processors can execute a maximum of P iterations in parallel. Suppose that processors P3, P4, and P6 indicate that their decisions are acceptable (filled circles in Fig. 2 denote

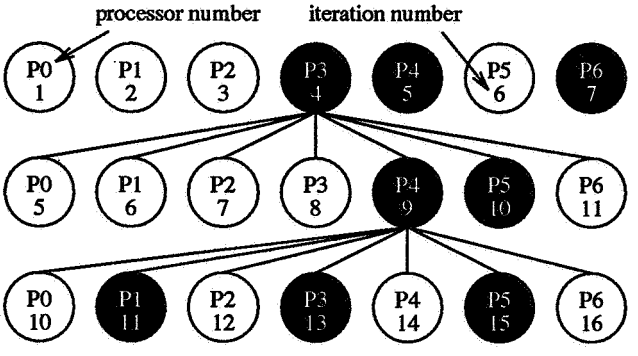

Figure 2: An 7-ary speculative tree with three levels.

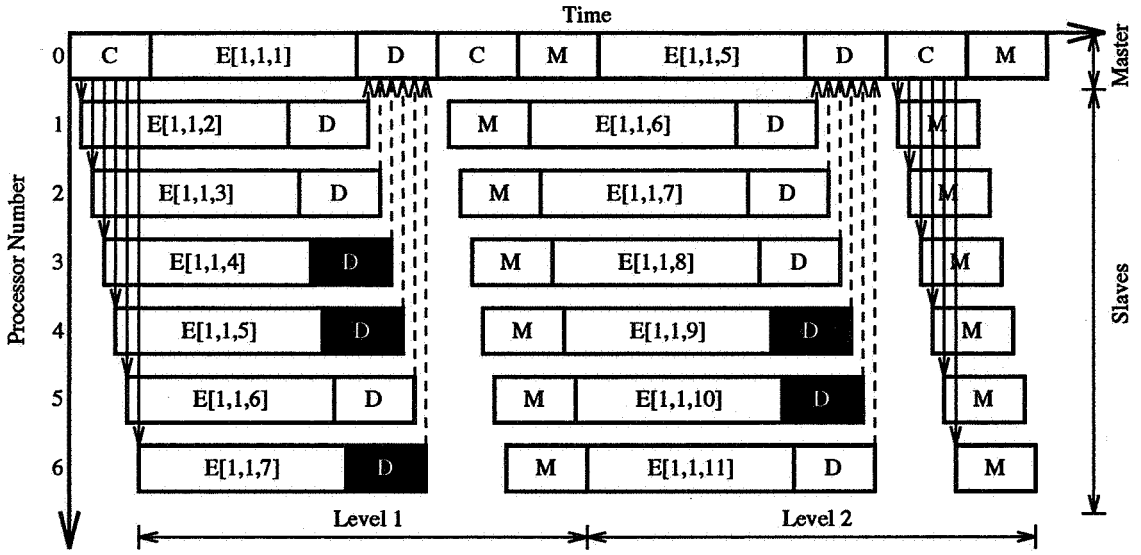

Figure 3: GSC on seven processors. The letters C, D, E, and M denote communicate, decide, evaluate, and modify, respectively. Solid arrows indicate broadcast while dashed arrows indicate point-to-point communication. Filled squares represent acceptable decisions.

acceptable decisions). However, the decision made by the lowest-numbered processor (or the lowest loop index) is accepted since decisions made by higher-numbered processors may be incorrect. In this example, the decision by P3 is accepted. The decision made by P4 could be incorrect since the data that it used should have been modified by P3. It cannot provide a reliable decision based on the current data. Similarly, the decisions made by P5, P6, and P7 are also incorrect because they are all based on wrong data.

The second level begins from iteration 5, executing iterations 5 to 11 in parallel. Each processor receives the loop index 4 based on which it modifies the data. After modifications, each processor performs the evaluation and decision steps by computing its iteration index. A processor can compute its individual index from the successful iteration index and its processor identification number (PID). We assume in Fig. 2 that at the second level, decisions made by P4 and P5 are acceptable. Once again, the decision made by P4 is selected since it is the lowest-numbered processor returning an accept decision.

The third level then starts from iteration 10, executing seven iterations in parallel. The process terminates when there is no OK decision in the innermost loop. The method presented therefore computed 11 iterations in three levels of the speculative tree. Note that a maximum of three iterations can be computed by the same number of levels of a binary speculative tree.

3.2 GSC ON A MULTIPROCESSOR

Figure 3 gives a mapping of GSC to seven processors. Let P0 be the master processor and P1, P2, ..., P6 be the slave processors. GSC proceeds as follows:

- P0 sends the loop index [1,1,1] to P1, P2, ..., P6. P0 then works on the evaluation and decision steps.
- After receiving the loop index from P0, each slave processor computes its own index by using its PID, and then performs the evaluation and decision steps. When done, each slave reports its result (*flag*) and PID to P0.
- When P0 completes its evaluation and decision steps of index [1,1,1], it collects *flags* from all the slave processors and itself.
- P0 uses the accept decision from the lowest PID to set the new loop index. Assuming that P3 is the processor with the lowest PID to return an OK *flag*, P0 sends the new index [1,1,4] to P1, P2, ..., P6.
- All processors, including P0, modify their local data using [1,1,4]. This completes level 1.
- As soon as a processor completes the modification step, it begins executing the next level.

Figure 3 shows that the GSC method completes nine iterations in two levels on seven processors. In the best case, the highest-numbered processor, P6, is the only one to return an OK *flag*. In that case, GSC can execute seven iterations at each level. In the worst case, the lowest-numbered processor, P0, returns a decision to accept. This case is identical to sequential SA, and may even be slower due to communication overhead.

4 EXPERIMENTAL RESULTS

This section presents our implementation of GSC on the AP1000 multiprocessor. We give a brief description of the AP1000 machine architecture and its programming environment, followed by a report of execution results.

4.1 THE AP1000 MULTIPROCESSOR

The Random L-SAT Problem and GSC have been completely implemented with various problem sizes and parameters. Problem instances were created by using the Random L-SAT generator [13,14], where the ratio of clauses to variables is 4.25 for a clause length of 3. We use the 1024-processor AP1000 distributed-memory multiprocessor which was built by and is currently operational at the Fujitsu Parallel Computing Research Facility in Japan.[1] The primary

[1]The 1024 processors are currently configured as a single 512-processor system and several smaller systems. The entire 1024-processor system is available to the general public once in a while.

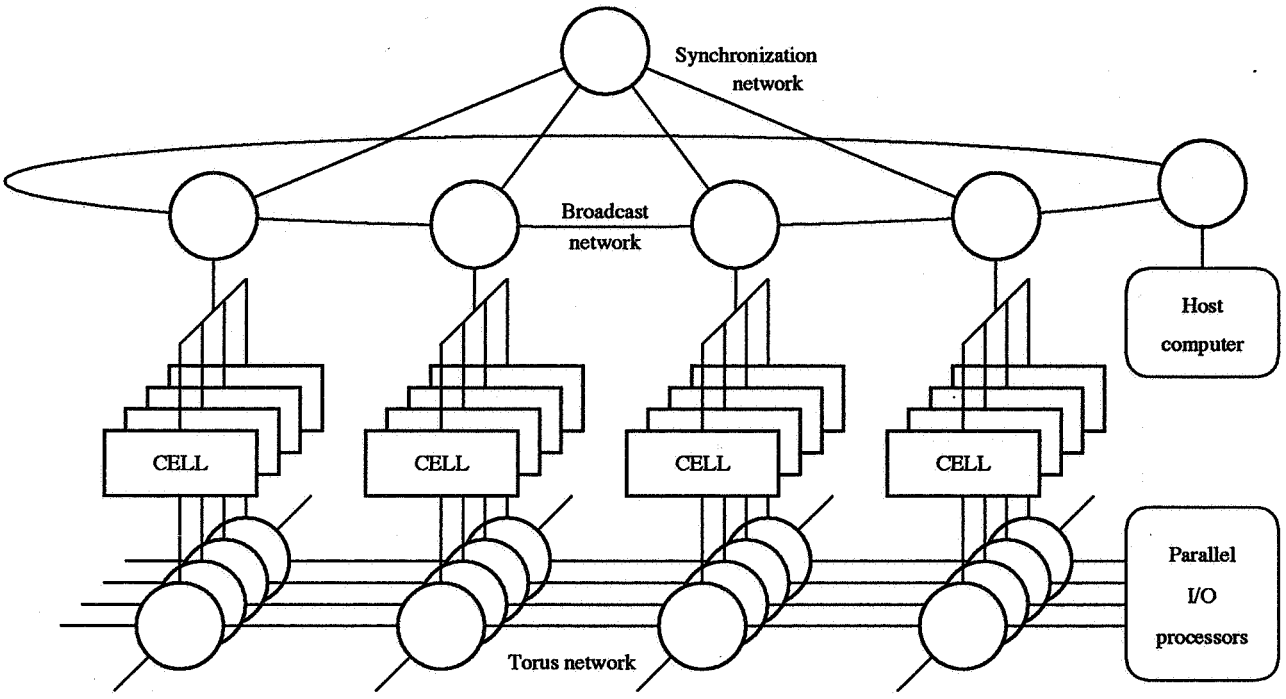

Figure 4: The AP1000 machine architecture.

reason we chose the AP1000 is that its large number of processors allows us to perform full-scale experiments.

The AP1000 is a general-purpose distributed-memory multiprocessor. Its performance on various real-world problems has been reported in Fujitsu Parallel Computing Workshops [19]. Figure 4 shows the machine architecture. Processing elements, called cells, are connected through three independent interconnection networks: a broadcast network (B-net), a torus network (T-net), and a synchronization network (S-net). A host Sun4 computer is connected to the B-net to control the AP1000. The B-net is used to broadcast data to all cells from the host or any individual cell. The S-net is used to synchronize various cell activities while the T-net allows point-to-point communication between cells. Each cell consists of an integer unit, a floating point unit, a message controller, network interface units, 128 KB of cache, and 16 MB of memory. More details about the architecture are available in [8,15].

All our computer codes are written in C with various parallel constructs for message-passing and synchronization. The AP1000 programming environment provides various debugging utilities including a runtime monitor, a performance analyzer, and the CASIM simulator which can run on workstations. We have found CASIM to be particularly helpful during code development since we did not have to directly deal with the real machine. There are certainly limitations of using the simulator since subtle bugs such as synchronization errors cannot be detected. These types of bugs only manifest on the AP1000 since they involve very subtle timing problems. We were fortunate not to have such bugs in our implementation mainly because of the master-slave parallel programming paradigm.

4.2 PARALLEL PERFORMANCE

Table 1 lists the number of levels required for initial temperatures of 10.0 and 100.0 on 1 to 500 processors. This indicates the amount of potential parallelism of the GSC method. Table 2 gives the execution times in seconds for the same cases. This indicates the amount of parallelism that is actually achieved.

Table 3 shows the average and total execution times of the individual steps for the 1000-variable/4250-clause SAT instance with an initial temperature of 10.0. Recall that each processor executes an unique iteration consisting of four steps: evaluate, decide, communicate, and modify. The time required for loop control and temperature update are included in the evaluation time. The table also shows the number of required levels for a given number of processors. The number of processors used for a particular run determines the number of levels.

This paper does not attempt to address issues related to solution quality, nor claim that GSC of SA is the most efficient method to solve Random L-SAT Problem instances. In fact, it is redundant to discuss the solution quality of this approach as it produces exactly the same solution as sequential SA. However, it is important to highlight the solution quality for the problem instances in this paper. Solution quality is defined as the percentage of clauses that are satisfied. Table 4 presents how GSC performed on large problem instances of up to 5000 variables while Table 5 lists solution quality in terms of the initial temperature.

5 PERFORMANCE EVALUATION AND DISCUSSION

Results reported in the previous section are analyzed to evaluate the performance of GSC. We first present the source of parallelism to identify potential performance. Next, execution time speedup curves are given to demonstrate achieved performance. The execution times are then analyzed to explicate how the time is spent. Finally, effects of the initial temperature are presented to identify the relationship between solution quality, initial temperature, and speedup.

# procs	$T_0 = 10.0$					$T_0 = 100.0$				
	200 vars	400 vars	600 vars	800 vars	1000 vars	200 vars	400 vars	600 vars	800 vars	1000 vars
1	1700000	6800000	15300000	27200000	42500000	1700000	6800000	15300000	27200000	42500000
10	171969	685746	1542571	2746130	4289103	171969	685746	1542571	2746131	4289104
20	87099	346089	778253	1387589	2166200	87099	346089	778253	1387590	2166202
30	58826	232903	523650	935085	1459176	58826	232903	523650	935086	1459177
40	44661	176300	396146	708443	1105041	44661	176300	396146	708444	1105043
50	36173	142343	319695	572800	892793	36173	142343	319695	572802	892795
60	30547	119725	268904	482500	751795	30547	119725	268904	482501	751797
70	26525	103594	232529	417792	650716	26525	103594	232529	417793	650718
80	23513	91453	205290	369335	575037	23513	91453	205290	369337	575039
90	21112	82075	184126	331817	516579	21112	82075	184126	331818	516581
100	19284	74496	167072	301571	469106	19284	74496	167072	301573	469108
150	13653	52038	116440	211791	328506	13653	52038	116440	211792	328508
200	10879	40769	91122	166883	258545	10879	40769	91122	166885	258547
250	9275	34103	76196	140467	217087	10191	34103	76196	140469	217089
300	8851	29702	66185	122876	189306	9161	29702	66185	122877	189308
350	8957	26592	59054	110451	169934	10124	26592	59054	110452	169936
400	9252	24254	53931	101329	155702	8235	24254	53931	101331	155704
450	9401	24746	49842	94254	144543	8724	22969	49842	94256	144545
500	9769	24923	46749	88759	135926	9960	25521	46749	88761	135928

Table 1: Number of levels for the Random L-SAT Problem instances of 200 to 1000 variables on the AP1000 multiprocessor.

# procs	$T_0 = 10.0$					$T_0 = 100.0$				
	200 vars	400 vars	600 vars	800 vars	1000 vars	200 vars	400 vars	600 vars	800 vars	1000 vars
1	123.5	493.7	1127.5	2006.3	3210.0	116.7	478.0	1101.4	1943.2	3107.3
10	21.1	83.9	190.5	342.3	544.4	20.9	84.6	193.9	345.3	549.6
20	11.8	46.5	105.4	190.3	301.9	11.7	47.0	107.6	192.5	305.7
30	8.6	33.3	75.4	136.4	215.8	8.5	33.5	76.5	137.4	217.4
40	6.8	26.1	59.0	107.1	169.3	6.7	26.2	60.0	107.9	170.7
50	7.7	29.7	66.9	121.5	191.0	5.7	22.1	50.4	90.9	143.3
60	6.7	25.8	58.2	105.7	166.0	4.9	19.0	43.4	78.5	123.5
70	4.6	17.2	38.9	71.0	111.7	4.4	16.9	38.5	69.9	109.8
80	4.1	15.4	34.7	63.4	99.6	4.0	15.1	34.3	62.4	98.0
90	3.8	14.2	31.9	58.6	91.9	3.7	13.9	31.6	57.7	90.5
100	3.5	13.1	29.5	54.3	85.1	3.4	12.9	29.3	53.5	83.7
150	2.6	9.6	21.7	40.0	62.3	2.6	9.6	21.7	40.0	62.3
200	3.2	11.5	25.6	47.6	74.0	3.1	11.4	25.6	47.5	73.7
250	2.9	10.0	22.4	41.9	64.6	3.2	10.0	22.4	41.9	64.6
300	2.8	9.2	20.4	38.5	59.3	3.0	9.0	20.2	38.0	58.5
350	3.4	8.5	18.9	35.8	55.0	3.5	8.5	18.9	35.8	55.0
400	3.5	8.0	17.8	34.0	52.2	2.9	7.9	17.6	33.5	51.3
450	3.2	7.9	16.8	32.2	49.0	3.2	7.8	16.8	32.2	49.0
500	3.7	9.5	16.6	32.0	48.9	3.7	9.2	16.2	31.1	47.4

Table 2: Execution time in seconds for the Random L-SAT Problem instances of 200 to 1000 variables on the AP1000.

# procs	# levels	Average execution time				Total execution time			
		Evaluate	Decide	Comm.	Modify	Evaluate	Decide	Comm.	Modify
10	4289103	0.000044	0.000037	0.000058	0.000095	0.004333	0.003656	0.005713	0.009376
50	892793	0.000044	0.000037	0.000138	0.000096	0.004388	0.003655	0.013635	0.009466
100	469106	0.000044	0.000037	0.000097	0.000096	0.004402	0.003678	0.009632	0.009459
200	169936	0.000045	0.000037	0.000187	0.000096	0.004412	0.003670	0.018553	0.009512
300	155704	0.000045	0.000037	0.000202	0.000096	0.004448	0.003655	0.020012	0.009532
400	144545	0.000045	0.000037	0.000215	0.000096	0.004443	0.003685	0.021316	0.009511
500	135926	0.000045	0.000037	0.000228	0.000096	0.004452	0.003669	0.022614	0.009526

Table 3: Average and total execution times in seconds of individual steps for the 1000-variable/4250-clause Random L-SAT Problem instance with $T_0 = 10.0$.

# vars	# clauses	# levels	# flips	# clauses satisfied	execution time
1000	4250	135928	86486	4236 (99.67%)	49 secs
2000	8500	513331	299341	8475 (99.71%)	179 secs
3000	12750	1137647	648178	12716 (99.73%)	399 secs
4000	17000	2212138	1438930	16962 (99.78%)	793 secs
5000	21250	3326346	2057900	21219 (99.85%)	1197 secs

Table 4: Results of large problem instances on 500 processors with three variables per clause.

T_0	200 vars	400 vars	600 vars	800 vars	1000 vars
0.1	849 (99.9%)	1692 (99.5%)	2541 (99.7%)	3379 (99.4%)	4240 (99.8%)
1.0	848 (99.8%)	1695 (99.5%)	2542 (99.7%)	3383 (99.5%)	4236 (99.7%)
10.0	848 (99.8%)	1695 (99.5%)	2542 (99.7%)	3383 (99.5%)	4236 (99.7%)
100.0	848 (99.8%)	1695 (99.5%)	2542 (99.7%)	3383 (99.5%)	4236 (99.7%)

Table 5: Number of clauses satisfied. The total number of clauses is 4.25 times the number of variables.

5.1 SOURCE OF PARALLELISM

Recall that Table 1 listed the number of levels required for different numbers of processors. Figure 5 plots the algorithmic speedup of the approach, which is defined as the ratio of the number of levels required on one processor to that required on P processors. The graph shows that GSC can provide a significant amount of algorithmic improvement. Note that the algorithmic speedup is an ideal measure without actual execution. It simply indicates that the problems have a large amount of parallelism that can be exploited.

5.2 SPEEDUP USING EXECUTION TIME

The execution times shown in Table 2 are converted to speedup curves to identify the effectiveness of the GSC method. Speedup is defined as the ratio of the execution time on one processor to that on P processors. The execution time speedup curves in Fig. 6 demonstrate the following three features. First, speedup increases as the number of processors is increased. The maximum speedup is about 70 on 500 processors. Second, speedup changes very little with temperature. This is because the parallelism exploited in the experiments comes mostly from the tail of SA, where decisions are rarely accepted. Third, speedup changes only slightly with problem size. This is a rather unusual phenomenon because larger problems tend to possess more parallelism than smaller ones, resulting in higher speedup. However, it is not true for our experiments because the SAT Problem with SA is independent of problem size.

Recall that an iteration consists of four steps: evaluate, decide, communicate, and modify. The evaluation step randomly selects a variable to flip, independent of the number of variables and clauses in the problem. It requires a fixed amount of time to find the effect of flipping the variable. The decision step is also independent of problem size. It decides using the Metropolis equation [12] and a random number. The modification step is slightly dependent on the problem size. The number of clauses the selected variable can appear in is essentially bounded to about 30 for our Random L-SAT Problem instances. Even for the 10,000-variable/42,500-clause instance, a variable can appear in a maximum of 29 clauses. Finally, the communication time is independent of the problem size. It only depends on the number of processors used.

5.3 ANATOMY OF EXECUTION TIME

To further understand the behavior of GSC, we plot the execution times for the individual steps. Figure 7 shows how the total execution time is spent in each step. It plots a complete execution time profile of the 1000-variable/4250-clause SAT Problem instance with an initial temperature of 10.0 on 10, 100, and 500 processors. The x-axis shows the level number, but only to 10,000 levels (See Table 1 for the total number of levels required). The y-axis shows the execution time in milli-seconds for each level.

The two lower curves in each plot indicate evaluation and decision times while the two upper curves indicate communication and modification times. Note that while the evaluation and decision times remain relatively constant, the communication and modification times fluctuate at each level. The modification time fluctuates because the number of clauses modified varies. However, the maximum number of clauses modified is limited to 30. The communication time fluctuates because of the nondeterministic nature of message-passing latency and network traffic.

Figure 8 summarizes the relationship between the number of processors and the percentage of the execution time spent on individual steps. When the number of processors is greater than 100, more than half the total execution time is spent in communication. For 500 processors, the communication time is almost 60% of the execution time while modification requires only about 20%. The evaluation and decision steps account for the remaining 20% of the time.

The curves in Figs. 7 and 8 indicate that a massively-parallel or a full-scale implementation needs to be crafted to reduce the communication overhead. It should be noted that our method does not broadcast large amounts of data; only the three loop indices i, j, and k, are broadcast at each iteration. Our experience suggests that significant effort on massively-parallel implementations of synchronous SA should be devoted to the communication step.

6 SUMMARY AND CONCLUSIONS

Testing the satisfiability of large Random L-SAT Problem instances is a challenging and time-consuming task. This report has presented a practical approach using synchronous simulated annealing (SA) on the AP1000 massively-parallel distributed-memory multiprocessor. We selected problem

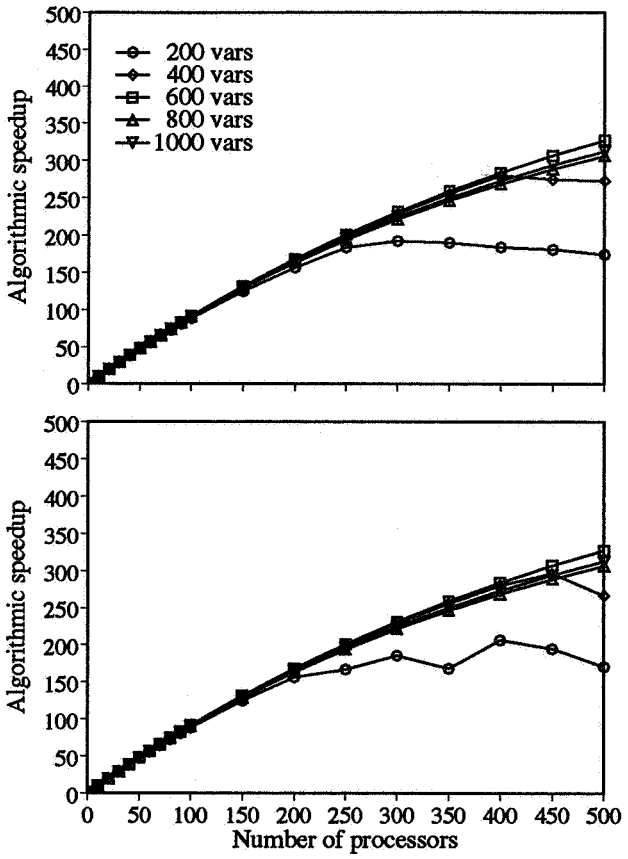

Figure 5: Algorithmic speedup on the AP1000 using GSC with $T_0 = 10.0$ (top) and $T_0 = 100.0$ (bottom).

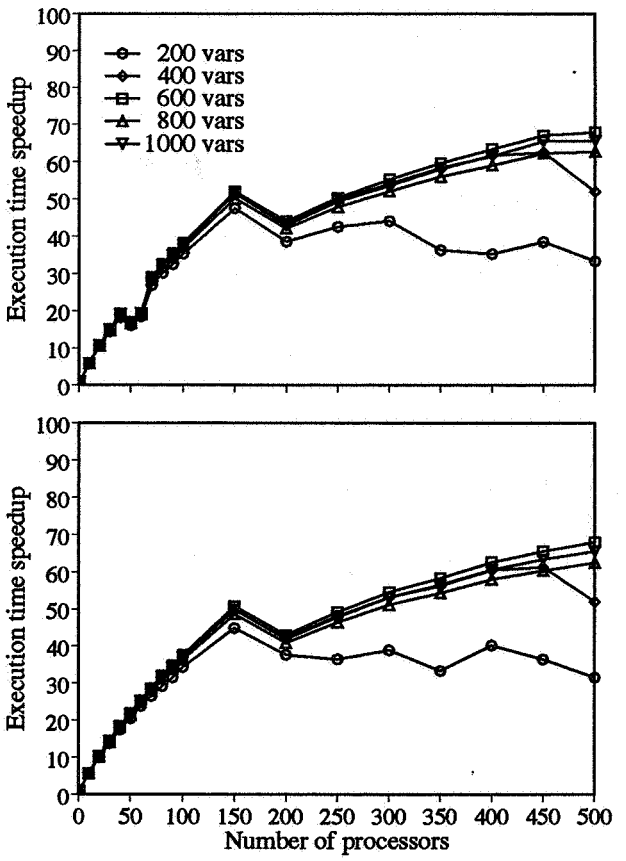

Figure 6: Execution time speedup on the AP1000 using GSC with $T_0 = 10.0$ (top) and $T_0 = 100.0$ (bottom).

instances between 100-variables/425-clauses and 5000-variables/21,250-clauses with initial temperatures of 0.001 to 100.0. These test cases have been executed on 1 to 500 processors and a part of the results have been reported here.

We used Generalized Speculative Computation (GSC) which is able to execute P different iterations in parallel on P processors using loop indices. This simplifies program control, making it suitable for massively-parallel distributed-memory multiprocessors. The GSC technique also simplifies communication by sending loop indices to all the processors. The three loop indices uniquely determine the random numbers for the entire SA process. In addition, the GSC approach is synchronous, giving the same decision sequence and solution quality as sequential SA regardless of the number of processors used.

Experimental results have demonstrated that GSC is effective for testing the Random L-SAT Problem. The 1000-variable/4250-clause problem instance for an initial temperature of 10.0 required almost 54 minutes on a single processor but only 49 seconds on 500 processors, while satisfying 4236 clauses. The 5000-variable/21,250-clause problem instance required about 20 minutes while satisfying 21,219 clauses. Overall, we have obtained nearly a 70-fold speedup on 500 processors. This is encouraging, considering that SA is highly sequential due to loop-carried dependence.

We found that the GSC method is effective for both low and high initial temperatures of up to 100.0; however, increasing the initial temperature rarely improved the solution quality. This indicates that high initial temperatures are not always practical nor preferable.

Finally, we have found that the communication time dominates the computation time. The evaluation and decision steps account for only 20% of the total time, and the modification step for another 20%. On the other hand, almost 60% of the total execution time is spent in communication. This dominance of the communication time certainly prefers problems with longer computation times. Reducing the communication overhead will result in an even better performance by GSC.

ACKNOWLEDGEMENTS

The authors would like to thank the Fujitsu Parallel Computing Laboratory for providing access to the AP1000 multiprocessor. Special thanks go to Toshiyuki Shimizu, an AP1000 group member, who introduced the first author to the AP1000. The first author would also like to thank Bill Spears of Naval Research Laboratory who explained the important issues in solving the SAT Problem with simulated annealing, and Bart Selman of AT&T Bell Laboratories who generously allowed us to use the Random L-SAT Problem generator. The work of the second author was supported by NASA under Contract Number NAS 2-13721 with the Universities Space Research Association.

REFERENCES

[1] P. Banerjee, M. H. Jones, and J. Sargent, "Parallel Simulated Annealing Algorithms for Cell Placement on Hy-

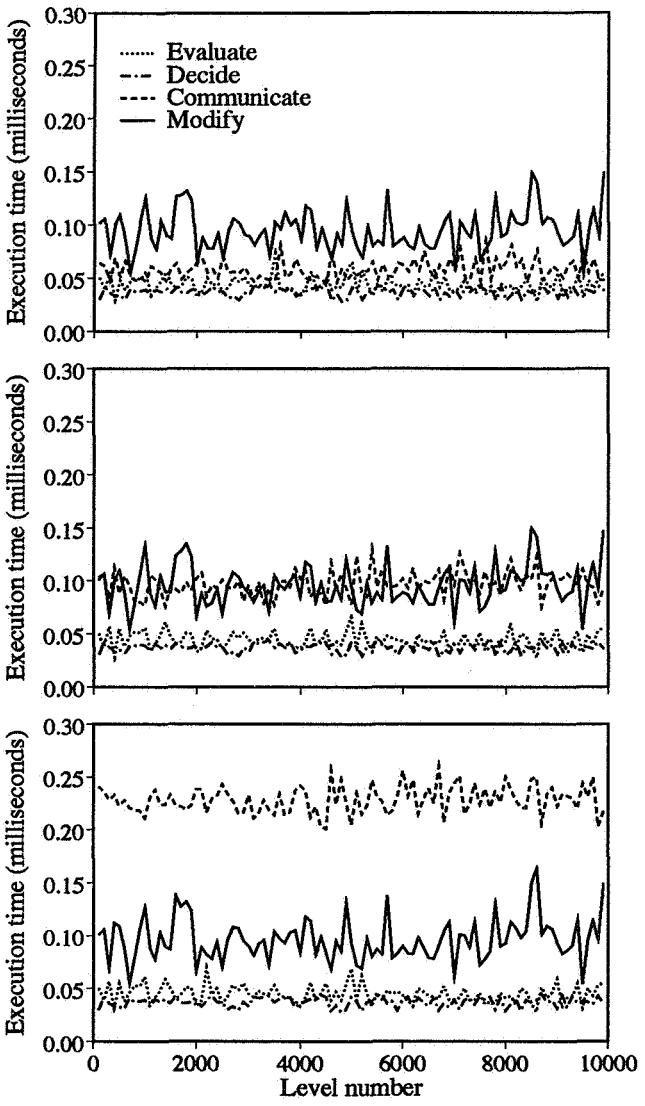

Figure 7: Execution time of individual steps for the 1000-variable/4250-clause problem instance with $T_0 = 10.0$ on 10 (top), 100 (middle), and 500 (bottom) processors.

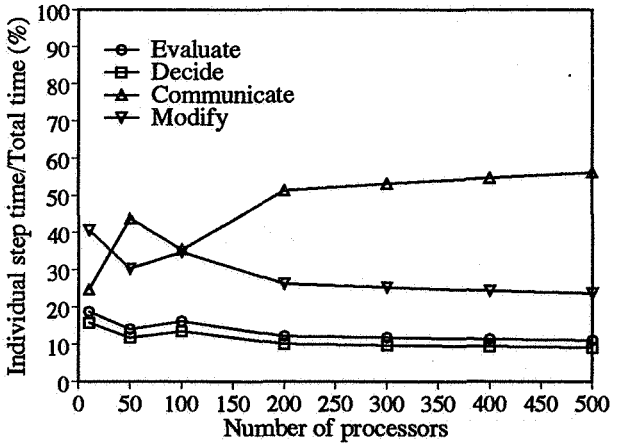

Figure 8: Comparison of individual execution times for the 1000-variable/4250-clause problem instance with $T_0 = 10.0$.

percube Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems* 1 (1990) 91–106.

[2] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *IEEE Trans. on Computer Aided Design* 6 (1987) 838–847.

[3] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of ACM* 7 (1960) 201–215.

[4] Dimacs Algorithm Implementation Challenge, "Satisfiability and Maximum Satisfiability Descriptions, Readings, and Problems," 1992.

[5] J. Franco and M. Paull, "Probabilistic Analysis of the Davis-Putnam Procedure for Solving the Satisfiability Problem," *Discrete Applied Mathematics* 5 (1983) 77–87.

[6] D. R. Greening, "Parallel Simulated Annealing Techniques," *Physica D* 42 (1990) 293–306.

[7] P. Hansen and B. Jaumard, "Algorithms for the Maximum Satisfiability Problem," *Computing* 44 (1990) 279–303.

[8] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata, "Improving AP1000 Parallel Computer Performance with Message Communication," *Proc. 20th ACM Intl. Symposium on Computer Architecture*, San Diego, CA, 1993, 314–325.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science* 220 (1983) 671–680.

[10] S. Kirkpatrick and B. Selman, "Critical Behavior in the Satisfiability of Random Boolean Expressions," *Science* 264 (1994) 1297–1301.

[11] S. A. Kravitz and R. A. Rutenbar, "Placement by Simulated Annealing on a Multiprocessor," *IEEE Trans. on Computer Aided Design* 6 (1987) 534–549.

[12] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equations of State Calculations by Fast Computing Machines," *Journal of Chemical Physics* 21 (1953) 1087–1091.

[13] D. Mitchell, B. Selman, and H. Levesque, "Hard and Easy Distributions of SAT Problems," *Proc. Natl. Conf. on Artificial Intelligence*, San Jose, CA, 1992, 459–465.

[14] B. Selman and H. Kautz, "An Empirical Study of Greedy Local Search for Satisfiability Testing," *Proc. Natl. Conf. on Artificial Intelligence*, Washington, DC, 1993, 46–51.

[15] T. Shimizu, H. Ishihata, and T. Horie, "Low-Latency Message Communication Support for the AP1000," *Proc. 19th ACM Intl. Symposium on Computer Architecture*, Gold Coast, Australia, 1992, 288–297.

[16] A. Sohn, "Parallel N-ary Speculative Computation of Simulated Annealing," *IEEE Trans. on Parallel and Distributed Systems* 6 (1995) 997–1005.

[17] B. Spears, "Simulated Annealing for Hard Satisfiability Problem," Naval Research Laboratory, TR AIC-93-015, 1993.

[18] E. E. Witte, R. D. Chamberlain, and M. A. Franklin, "Parallel Simulated Annealing Using Speculative Computation," *IEEE Trans. on Parallel and Distributed Systems* 2 (1991) 483–494.

[19] Workshop Series, Fujitsu Parallel Computing Research Facility, Kawasaki, Japan, 1992, 1993, 1994, and London, UK, 1995.