IEEE Transactions on Parallel and Distributed Systems,
October 1995, pp.997-1005

Parallel *N*-ary Speculative Computation of Simulated Annealing on the AP1000 Massively Parallel Multiprocessor

Andrew Sohn
Dept. of Computer and Information Science
New Jersey Institute of Technology
Newark, New Jersey 07102-1982, sohn@cis.njit.edu

Abstract

Simulated annealing is known to be an efficient method for combinatorial optimization problems. Its usage for realistic problem size, however, has been limited by the long execution time due to its sequential nature. This report presents a practical approach to *synchronous* simulated annealing for massively parallel distributed-memory multiprocessors. We use an *n*-ary speculative tree to execute *n* different iterations in parallel on *n* processors, called *Generalized Speculative Computation* (GSC). Execution results of the 100- to 500-city Traveling Salesman Problems on the AP1000 massively parallel multiprocessor demonstrate that the GSC approach can be an effective method for parallel simulated annealing as it gave over 20-fold speedup on 100 processors.

Index terms: Parallel simulated annealing, combinatorial optimization, synchronous simulated annealing, speculative computation, Traveling Salesman Problem, and simulated annealing

1 Introduction

Simulated annealing (SA) is one of the highly efficient methods for combinatorial optimization problems [6]. 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 that the way metal cool and anneal as temperatures change. A typical implementation of simulated annealing consists of two nested loops; The outer loop controls temperatures based on the annealing schedule. The inner loop performs the three steps of evaluate, decide, and modify (or move) for the given temperature. The evaluation step suggests the possible next state by generating random numbers and evaluates it by using evaluation criteria. The decision step decides if the suggested state is acceptable. If the decision step found the suggested state indeed acceptable, the modification step would actually modify the central data according to what the evaluation step suggested. The simulated annealing technique has been applied to various optimization problems including VLSI cell placement, the Traveling Salesman Problem [1], satisfiability problem, and graph partitioning. Among the applications is cell placement which is to arrange a very large number of circuit elements on a tiny silicon subtrate while minimizing wire interference. Simulated annealing has been found highly successful for such cell placement [2, 3, 7, 11, 14]. A large set of literature indicates the success of simulated annealing in cell placement.

The major hurdle of simulated annealing, however, is the long computation time due to its sequential nature of 'slow' annealing process. To be more precise, simulated annealing is highly sequential due to a loop-carried dependence between iterations. Iteration i depends on iteration $i-1$ because iteration i-1 may modify the central data based on which iteration i should proceed. This inherent sequential execution takes much time on a single processor and has thus spurred much research in simulated annealing in parallel machine environments. Parallel simulated annealing can be classified into two categories: *synchronous* and *asynchronous* [4]. The synchronous approach maintains the same decision sequence as sequential simulated annealing [7, 14] while the asynchronous approach does not [8]. Our objective in this study is not to find a new method but to perform simulated annealing fast while preserving the algorithmic behavior of sequential simulated annealing. We therefore limit ourselves to synchronous simulated annealing. Among parallel synchronous algorithms is *speculative computation* [14]. The idea is to simultaneously compute multiple decision sequences of a binary speculative tree using multiple processors. Each processor has a role assigned to it to reflect the binary decision sequence of a speculative tree, which we call *binary speculative computation* (BSC). BSC has been reported to exploit parallelism at both low and high temperatures. Discussion on parallelism exploitation and experimental results of BSC are presented in [14].

This reports introduces a new practical approach to parallel synchronous simulated annealing for large scale massively parallel distributed-memory multiprocessors. Our approach generalizes binary speculation to N-ary speculation, called *generalized speculative computation* (GSC). N processors simultaneously perform N different evaluations and decisions. All processors simultaneously modify DATA locally with a simple control mechanism. Two loop indices are used to simplify the controlling of iterations and communication. Using an encoded single integer of two loop indices completely eliminates sending DATA to processors. To demonstrate this massively parallel approach to synchronous simulated annealing, we implement the 100- to 500-city Traveling Salesman Problem (TSP) on the 1024-processor AP1000 message-passing multiprocessor [5, 12]. We begin our discussion in section 2 by giving a brief description of simulated annealing and binary speculative computation. Section 3 presents a new practical method to synchronous simulated annealing for massively parallel distributed-memory multiprocessors. Section 4 lists implementation parameters and execution results of the TSP on the AP1000 multiprocessor. Section 5 analyzes experimental results of the Generalized Speculative Computation. We conclude our report in the last section.

2 Binary Speculative Computation

Simulated annealing can be implemented in two nested loops. Figure 1 shows a typical simulated annealing algorithm implemented in two nested loops. The outer loop is a temperature loop which lowers temperature based on the cooling speed (or annealing schedule). The inner loop executes three steps for the given temperature: evaluate, decide, and modify. The evaluation step of line 1 generates random numbers and computes the difference between the previous solution and the new solution. The decision step of line 2 decides if the new solution is acceptable. It generates *flag* which takes on OK (accept) or NOK (reject). This step uses the Metropolis equation [9]. *Flag* is OK if $\Delta E < 0$ or $r < e^{-\Delta E/T}$, where r is a random number and $0 < r < 1$. If *flag* is OK, the modification step of line 3 will modify DATA based on the proposal from the evaluation step, resulting in DATA′. Otherwise, no data modification will take place.

```
T←T0                                ;initial temperature
for i=1 to ncity do                 ;i-loop
    for j=1 to ncity^2 do           ;j-loop
1:      ΔE ← evaluate(DATA,i,j)      ;Compute ΔE.
2:      flag ← decide(ΔE,Tj)         ;Decide.
3:      DATA ← modify(DATA,flag)     ;Modify DATA if OK.
    T ← T*CS                        ;Lower temperature.
```

Figure 1: Sequential Simulated Annealing. *flag* = {OK, NOK} CS = cooling speed.

The above algorithm terminates either when all the *ncity* iterations of the outer loop are completed or when no iterations of the inner loop produce an OK flag [10]. The inner *j*-loop iterates $ncity^2$ times for a fixed temperature. If none of the $ncity^2$ iterations of the *j*-loop produces an OK flag, then the whole process terminates. This termination condition however is implementation dependent and can be modified to suit different problems. The algorithm in Figure 1 is highly sequential. First, the inner loop (*j*-loop) has loop-carried dependence. Iteration *j* may modify DATA, based on which iteration *j*+1 must proceed. Second, each iteration of the *j*-loop has *true* data dependencies. The three steps within each iteration are executed one at a time. It is clear from Figure 1 that line 2 uses ΔE computed in line 1. Line 3 uses both the results of line 1 and line 2. It is therefore difficult to overlap or simultaneously execute *j*-loop iterations. One way to parallelize simulated annealing is a speculative computation technique with a binary speculative tree, which we call binary speculative computation (BSC) [14]. The binary speculative computation assigns a role to each processor: root, accept, or reject, as shown in Figure 2.

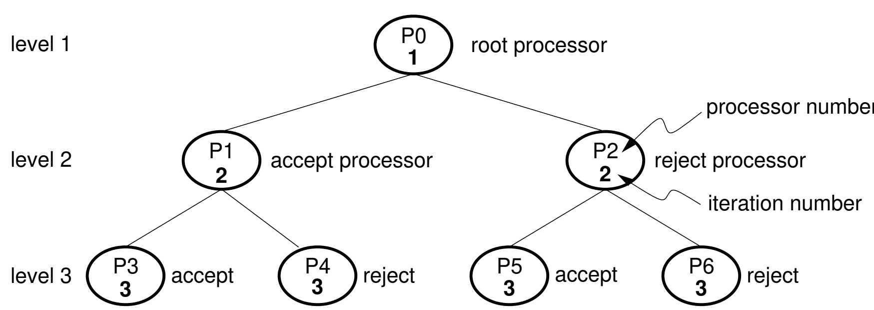

Figure 2: Binary speculative tree with three levels. Seven processors execute three iterations in parallel [14].

Each processor iterates an altered sequence of modify, evaluate, and decide. The root processor which is working on iteration *i* sends a central data-structure (or DATA) to two (accept and reject) child processors before and after the modification step of iteration *i*. Each child processor, when receiving DATA regardless of its correctness, starts working on iteration *i*+1 which goes through the same sequence as the root. Accept processor receives a modified data, DATA′. Once received, it again sends DATA′ to two (accept and reject) processors before and after the modification step of iteration *i*+1. After sending DATA′ to two children, it starts working on iteration *i*+1. Reject processor receives the same data as the root processor. Assuming that the root processor will not modify DATA, it sends DATA to two processors before and after the modification step of iteration *i*+1. This type of sending DATA, DATA′, DATA″, etc. to children processors continues until there is no more processors left in the processor pool. Speculative computation using a binary tree has been reported to exploit parallelism at both low and high temperatures [14].

3 Generalized *N*-ary Speculative Computation

Generalized speculative computation uses an *N*-ary tree. Figure 4 shows an *N*-ary speculative tree with three *levels*. There is no notion of accept or reject processor. No two processors execute the same iteration. All processors perform *different* iterations. The idea is straightforward. Each processor receives a loop index at runtime and performs the three steps: modify, evaluate, and decide. At the end of the three steps, it raises a flag to indicate its decision result. The decision results are then collected and the decision result of the lowest numbered loop index is selected to initiate the next level. To ensure that the parallel version generates the *same* decision sequence as sequential simulated annealing, we use the same sequence of seeds to generate pseudo-random numbers. There are $ncity^3$ seeds, where *ncity* is the maximum number of outer loop iterations and $ncity^2$ is the maximum number of inner loop iterations. A loop index $[i,j]$ used as a seed therefore generates a unique random number.

Figure 3 shows the GSC executing 11 (*not* 16) iterations in three levels. The first level of an *N*-ary speculative tree shows seven processors executing seven iterations simultaneously. Suppose that the three processors 3, 4 and 6 find that their decisions are acceptable (filled circles indicate acceptable decisions). Among the three, the decision made by the lowest numbered processor (or lowest loop index), processor 3, is accepted since the decisions made by the higher numbered processors are wrong. The data that processor 4 used to decide OK should have been modified by processor 3. Processor 4 can not decide based on the current data. Similarly, the decision made by processor 6 is again wrong since it is based on the wrong data.

The second level starts from iteration 5, executing iterations 5 to 11 by seven processors in parallel. Each processor receives loop index 4 based on which it modifies DATA. After modification, each processor starts evaluation and decision steps by computing its next iteration index. Next iteration index can be easily obtained by simply adding its processor number to the successful index 4. For example, processor 2 adds its processor number 2 to the successful loop index 4, and add 1, resulting in 7. Processor 2 now starts executing iteration 7 in the second level of the speculative tree. Other processors similarly compute their own iteration numbers. Assume that the second level finds that the two decisions made by processors 4 and 5 are acceptable. Again, the decision made by processor 4 will be finally selected since it is the lowest numbered processor. The third level then starts from iteration 10, executing seven iterations in parallel. The whole process terminates when there is no OK decision in the inner loop. The method presented above can therefore compute 11 iterations in three levels of speculative tree while maintaining the same decision sequence as sequential simulated annealing does. Note that binary speculative computation can execute a maximum of up to only 3 iterations in three levels of binary speculative tree.

Our implementation is designed to simplify controlling iterations. Each level of the *N*-ary speculative tree is directly mapped to *N* processors. We use a master-slave parallel programming paradigm. Given *N* processors, processor 0 is selected to be a master and a slave while the rest will become slave processors. The master processor executes the three steps of evaluate, decide and modify, and manages communication, including broadcasting and collection of flags. Slave processors execute the three steps. Figure 4 illustrates a parallel implementation of generalized speculative

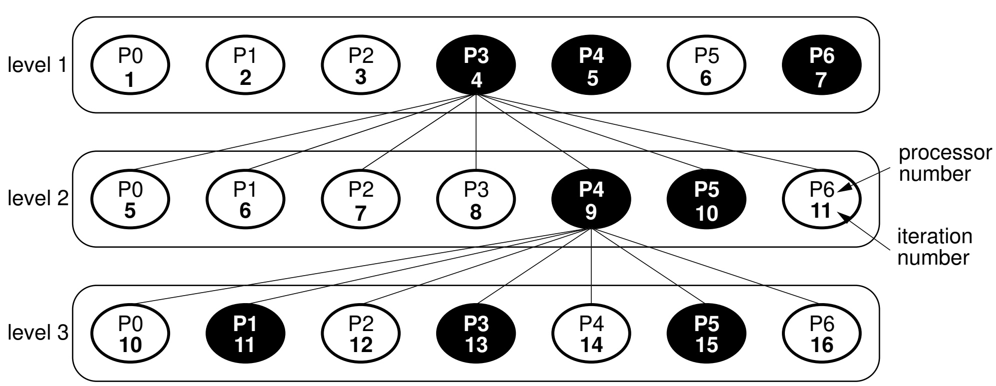

Figure 3: *N*-ary speculative tree, where *N* is a number of processors. Filled circles indicate acceptable decisions. Seven processors execute 11 iterations in parallel.

computation on seven processors (see Figure 2 for binary speculative computation). To illustrate the difference between the BSC and the GSC, we use the speculative tree shown in Figure 3 and *seven* processors, as used in binary speculative computation. Let P_0 be a *master* processor and $P_0,...,P_6$ be *slave* processors. The GSC proceeds as follows:

1. At the very beginning, the master processor sends a *single* integer, loop index [1,1], to all slave processors. After sending, it starts working on the evaluation and decision steps.
2. Once received, each slave processor computes its own index by using its processor identification number (PID), and performs evaluation and decision steps. When finishing the two steps, each slave processor reports its decision result (*flag*) and PID to the master processor. *Flag* and PID are encoded in a single integer before sending.
3. When P_0 finishes the evaluation and decision steps of [1,1], it collects seven encoded flags from six slave processors and itself. Each encoded integer is decoded back to *flag* and PID before its use.
4. Assume now that P_0 collected all seven *flags* among which three processors (P_3, P_4, and P_6) decide OK, i.e., their decisions are acceptable. The master uses the decision made by P_3, the lowest numbered-PID. The decisions made by P_4 and P_6 are *invalid* since P_3 will modify DATA and generate new DATA, based on which other higher loop iterations should proceed. The master now sends the new index [1,4] to all processors.
5. After sending [1,4] to all processors, the master will modify its own DATA using [1,4] while all other processors will do the same thing. This modification step completes level 1.
6. As soon as each processor finishes the modification step, it immediately starts the second level with no interruption.

Figure 4 shows that the GSC method execute nine iterations in two levels on seven processors, assuming that in the second level P_4 is the lowest-numbered PID to be successful in the decision step. The best case occurs if the highest numbered processor, P_6, is the only one which decides OK while all others decide NOK. The best case of GSC can execute seven iterations in each level by seven processors. The worst case occurs when the lowest numbered processor, P_0, decides OK. This worst case is the same as sequential simulated annealing, or it may be even slower than sequential simulated annealing due to communication overhead. In general, if P_i decides OK, then the decisions made by P_{i+k},

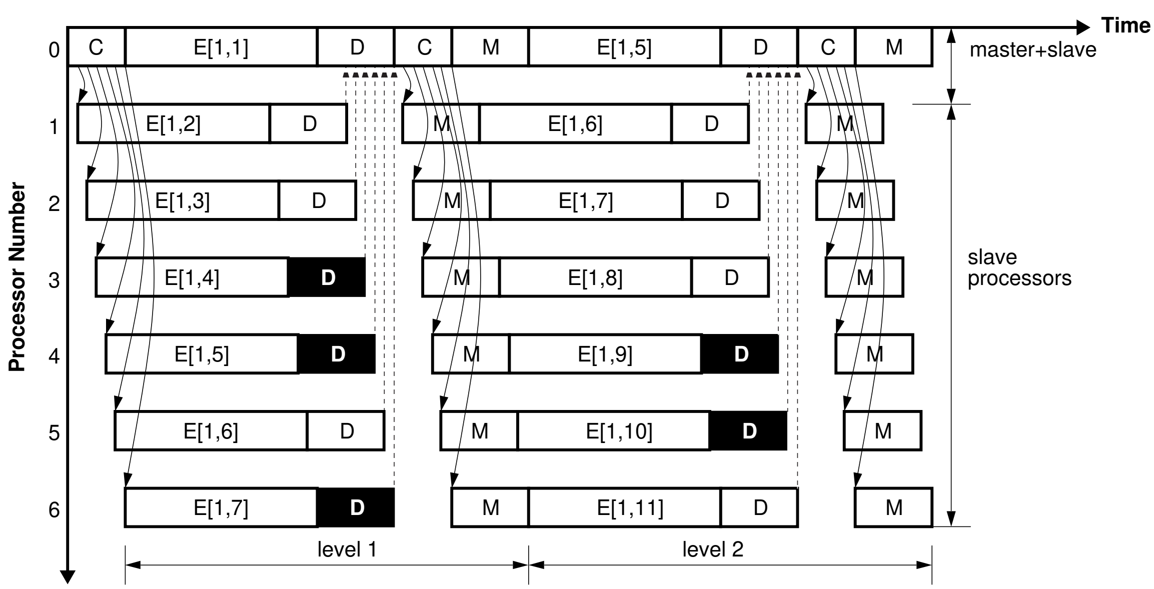

Figure 4: GSC on seven processors. P_0 is a master while $P_0,...,P_6$ are slaves. Solid arrows indicate broadcasting of a single integer [i,j] to all processors while dotted arrows send *flags* to the master. Filled squares indicate that the decisions are acceptable, i.e., *flag* = OK. The decision made by the lowest-numbered loop-index is used for next round. The second round therefore starts from iteration [1,5] while the third will start from iteration [1,10], in which case 17 iterations are executed in three rounds.

$1 \le k \le n-i-1$, are invalid due to the loop-carried dependence between iterations. Considering that simulated annealing spends much of its execution time on the tail where a decision is rarely acceptable, our approach is a natural approach to parallel simulated annealing for massively parallel multiprocessors.

4 Experimental Results

Our target problem is the Traveling Salesman Problem (TSP), where the salesman is to find the shortest path while visiting the *n* cities once each. We have used the 100- to 500-city TSP for our experiments. All the cities are randomly generated. The multiprocessor on which we implemented the TSP is the 1024-processor AP1000 message-passing multiprocessor built and operational at the Fujitsu parallel computing laboratory [14]. All the programs are written in C with various parallel constructs for message-passing and synchronization. The following parameters summarize our implementation on the AP1000:

- Problem size: 100-city to 500-city TSP.
- Number of processors used: 1 to 100.
- Initial temperatures T_0: 0.001 to 100.
- Error rate: 0.001
- Cooling speed and annealing schedule: $T_{i+1} = 0.9 * T_i$, i = outer loop index.
- Number of iterations for outer loop i_{max}: *ncity*, where *ncity* is the number of cities.
- Number of iterations for inner loop j_{max}: $ncity^2$.

The values to i_{max} and j_{max} are chosen respectively *ncity* and $ncity^2$ to reflect problem size in computation [10], and can be adjusted for different problem. If j_{max}=1, the GSC will execute *ncity* iterations where all iterations have *different* temperature. Table 1 lists results for the initial temperatures of 0.1, 1 and 10.

No of Processors	$T_0 = 0.1$					$T_0 = 1$					$T_0 = 10$				
	100-city	200-city	300-city	400-city	500-city	100-city	200-city	300-city	400-city	500-city	100-city	200-city	300-city	400-city	500-city
1	12.64	39.08	87.62	105.59	165.96	15.607	38.627	86.555	104.413	245.780	50.207	220.814	557.386	961.458	1442.729
2	7.67	24.04	53.53	65.30	102.39	9.488	23.873	53.041	64.703	152.038	31.160	139.124	351.433	614.590	932.021
4	4.54	14.56	32.19	39.95	62.84	5.629	14.549	31.943	39.629	93.351	18.964	86.853	219.984	393.084	604.979
6	3.35	10.93	23.97	30.27	47.74	4.156	10.941	24.013	30.064	70.828	14.293	66.682	169.461	306.327	480.862
8	2.70	8.89	19.42	24.76	39.55	3.376	8.935	19.430	24.870	58.038	11.817	55.801	141.993	263.059	413.303
10	2.31	7.69	16.74	21.64	34.24	2.872	7.739	16.739	21.644	50.714	10.322	48.988	125.482	235.214	372.699
20	1.45	5.02	10.78	14.60	23.26	1.798	5.114	10.865	14.715	34.233	7.028	34.737	90.541	177.846	289.747
30	1.14	3.87	8.26	11.33	17.99	1.410	3.972	8.410	11.504	26.344	5.911	28.718	76.546	152.092	251.721
40	0.98	3.27	6.89	9.57	15.26	1.216	3.399	7.040	9.720	22.017	5.434	25.898	69.527	139.044	232.273
50	0.91	2.97	6.37	8.79	13.86	1.083	2.987	6.231	8.611	19.363	5.135	24.089	65.654	131.651	221.793
60	0.84	2.69	5.75	8.01	12.63	0.990	2.733	5.710	7.937	17.608	4.938	23.004	63.364	127.519	216.423
70	0.75	2.38	5.16	7.20	11.31	0.940	2.532	5.287	7.355	16.239	4.853	22.196	62.044	124.787	213.040
80	0.71	2.23	4.84	6.74	10.54	0.879	2.383	4.982	6.900	15.041	4.772	21.592	61.104	122.375	209.517
90	0.67	2.10	4.59	6.38	9.89	0.837	2.250	4.739	6.542	14.067	4.708	21.198	60.755	120.901	207.251
100	0.64	2.01	4.31	6.04	9.37	0.802	2.172	4.457	6.213	13.270	4.663	20.886	59.593	119.426	205.563

Table 1: Execution time (in seconds) of the TSP on the AP1000 multiprocessor.

Table 2 lists execution time of the individual steps for the 100-city TSP with the initial temperature of 0.1. Recall that each processor executes a unique iteration consisting of four steps: communicate, modify, evaluate, and decide. Figure 4 defined level in which *N* processors execute *N* different iterations. The total number of processors used for particular run uniquely determines the total number of levels. The 100-city TSP on 10 processors, for example, executed 4283 levels of 10-ary speculative tree.

# of Processors	Overall execution time (seconds) for 100-city, T_0=0.1				# of levels	Average level execution time (seconds) for 100-city, T_0=0.1			
	Comm.	Modify	Eval. + Dec.	C+M+E+D		Comm.	Modify	Eval. + Dec.	C+M+E+D
10	0.945403	1.049772	0.361732	2.356907	4283	0.000221	0.000245	0.000084	0.000550
20	0.697500	0.586510	0.182120	1.466130	2282	0.000306	0.000257	0.000080	0.000643
30	0.592556	0.426199	0.122711	1.141466	1616	0.000367	0.000264	0.000076	0.000707
40	0.526534	0.354602	0.092783	0.973919	1282	0.000411	0.000277	0.000072	0.000760
50	0.474128	0.306135	0.075399	0.855662	1083	0.000438	0.000283	0.000070	0.000791
60	0.438558	0.272980	0.062735	0.774273	949	0.000462	0.000288	0.000066	0.000816
70	0.410689	0.249984	0.054686	0.715359	854	0.000481	0.000293	0.000064	0.000838
80	0.387143	0.236090	0.048045	0.671278	782	0.000495	0.000302	0.000061	0.000858
90	0.368060	0.218676	0.042865	0.629601	727	0.000506	0.000301	0.000059	0.000866
100	0.346610	0.212170	0.039330	0.598110	683	0.000508	0.000311	0.000058	0.000877

Table 2: Execution time of individual steps for the 100-city TSP with T_0 = 0.1. C = Communicate, M = Modify, E = Evaluate, D = Decide.

The first four columns next to the number of processors indicate the overall execution time for each step. The last four columns show the average level execution time for each step. We find that the evaluation and decision time is relatively small compared to others and therefore combined the two into one. Note that some execution time may be slightly different from Table 1. This slight difference between Tables 1 and 2 is due to the measurement of individual execution time. To measure the individual time, we had to insert many system calls in the program and also needed to interact with the host. Minor program statements such conditional constructs and loop control constructs were not included in the measurement.

5 Performance Evaluation and Discussion

We analyze experimental results to evaluate the performance of GSC in this section. We first present speedup curves to identify the absolute performance of generalized speculative computation for the Traveling Salesman Problem. Second, execution time is thoroughly analyzed to explicate where the time is spent. Third, the effects of initial temperature is presented to identify the relation between solution quality, initial temperature, and speedup. We summarize our findings at the end of this section.

5.1 Effects of generalized speculative computation

Execution time shown in Table 1 is now converted to speedup curves to identify the effectiveness of the GSC method. Speedup is defined as the execution time on 1 processor over *n* processors for the given problem. Figure 5 shows speedup plots. Experimental results give the following three observations: *First, speedup increases as the number of processors increases*. For all three different temperatures, we find that the GSC gives more speedup as the number of processors increases. However, the rate at which speedup increases (acceleration) decreases as more processors are used to solve the same problem. For the low temperature of 0.1, the speedup increases relentlessly while at the high temperature of 10 it increases slowly and saturates when the number of processors reaches 100. We shall come back to this in more detail later in the section.

Second, speedup decreases as temperature increases. This is not surprising because the number of OKs increases as temperature increases. When temperature is high, there will be more number of OKs than low temperatures, resulting in highly sequential decision steps. The two temperatures of 0.1 and 1 do not show a significant difference whereas the temperature of 1 and 10 show a noticeable difference.

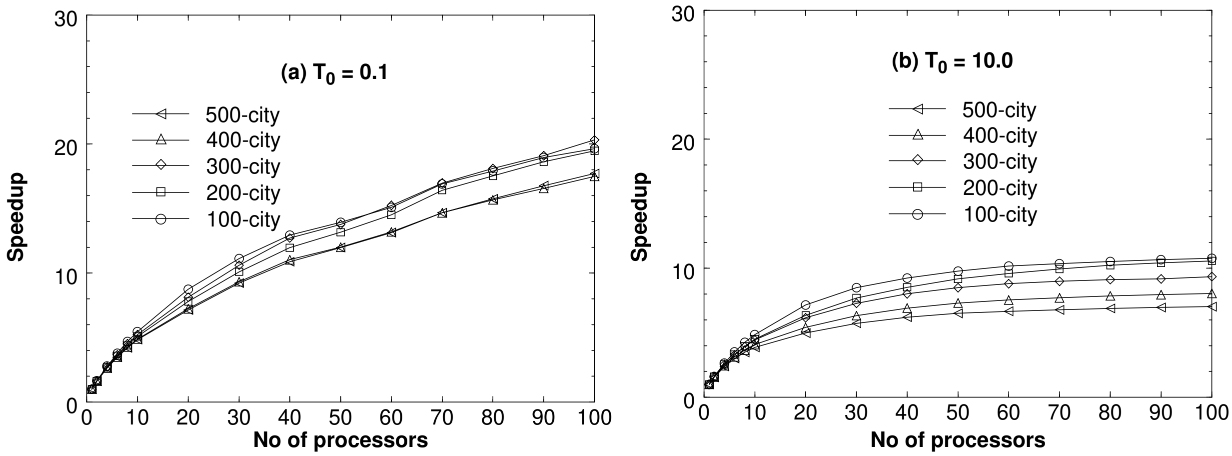

Figure 5: Speedup of the Traveling Salesman Problem on the AP1000 multiprocessor, using simulated annealing with Generalized Speculative Computation. (a) T_0 = 0.1, (b) T_0 = 10.

Third, speedup decreases as problem size increases. This is a rather unusual phenomenon because larger problems tend to possess more parallelism than smaller ones, which would result in higher speedup. For instance, the matrix multiplication of two 100x100 matrices can likely give more speedup than that of 10x10 when a large number of processors, say 100 processors, is used. In other words, for a large multiprocessor speedup is likely to increase as the problem size increases. However, it is certainly not the case for our experiments as Figure 5 suggests. It is again not surprising that we obtained such results because the Traveling Salesman Problem with simulated annealing is less dependent on problem size. Recall that each iteration consists of four steps: evaluate, decide, modify, and communicate.

The evaluation step always selects a fixed number of cities (2 or 3 cities) no matter how many cities there are in the given problem. In fact, this step will take less time for a larger number of cities because the selection of two (for reversal) or three (for transportation) cities out of 500 cities is easier than out of 50 cities. This is precisely why most of our experimental results show that speedup for larger problems is smaller than those for smaller ones. The decision step has absolutely nothing to with the problem size. It simply decides using the Metropolis equation and a random number. The total number of instructions for this step is not more than 10. The problem size however does affect the modification step to a certain extent. When modifying DATA, either reversal or transportation of a selected segment, there can be a large amount of memory operations due to array modification. For example, when a segment of 10 cities is to be reversed, the reversal operation can involve a minimum of 20 memory references (10 for reading and 10 for writing), in addition to other instructions such as a condition test for loop termination. We were particularly cautious not to increase memory references by optimizing our program. It was the transportation that we paid much attention to avoid creation of a whole new array. The last step, communication, is absolutely independent of problem size. It depends only on the number of processors used.

5.2 Anatomy of execution time

To further explicate the behavior of GSC, we plot the execution time for individual steps. Figure 6 clarifies exactly how the total execution time is spent on each of the four steps. It plots a complete execution time profile of the 100-city TSP with the initial temperature of 0.1. The *x*-axis indicates level number, in which *N* processors execute *N* different iterations. The y-axis shows actual execution time in *milli*seconds for each level. Figure 6(a) plots execution time for 10 processors which executed 4283 levels while Figure 6(b) plots for 100 processors which executed 683 levels. Since each level consists of *N* processors speculating on *N* iterations, the number of levels becomes smaller when a larger number of processors is used (Table 2).

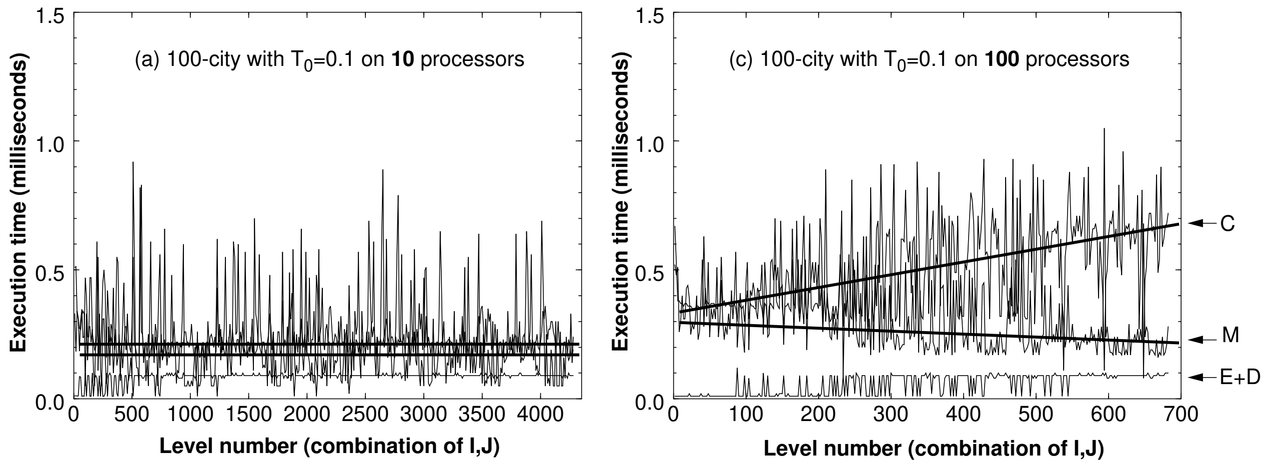

Figure 6: Execution time of individual steps (millisec.). (a) 10 processors, (b) 100 processors. C= communication time, M = modification time, and E+D = evaluation and decision time.

The bottom curve of each plot indicates evaluation and decision time while the two upper curves indicate communication and modification time, often superimposed. High spikes show the nondeterministic nature of communication time, which we believe cannot simply be formulated by an equation with parameters. The curves demonstrate the following three facts:

- Decision and evaluation time is essentially constant and small compared to communication time and modification time. This is true regardless of the number of processors.
- Modification time also remains relatively constant throughout the entire run although there is a little fluctuation at times. Modification time is superimposed in Figure 6(a) but becomes visible in Figure 6(b).
- Communication time increases as the number of processors increases. Figure 6(b) clearly shows that there is a substantial increase in communication time. Communication time dominates the four steps for 100 processors.

Figure 7 summarizes the execution time for individual steps. It shows the relationship between the number of processors and the percentage of execution time of individual steps. As the number of processors increases over 20, the communication step begins to dominate the whole process. Note that when 100 processors are used, the communication time occupies almost 60% of the whole execution time, while the modification time accounts for only 35%. Evaluation and decision time are very low as expected.

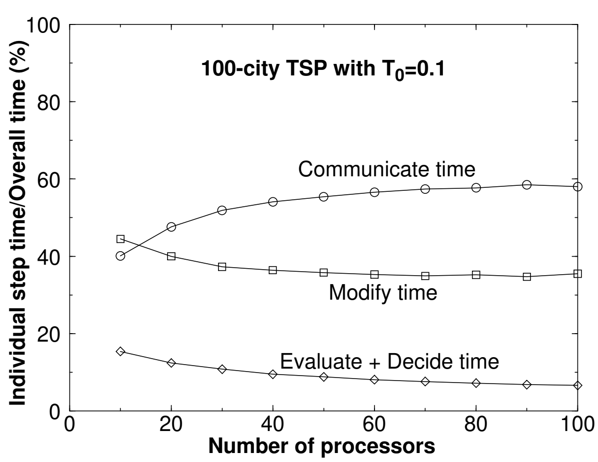

Figure 7: Comparison of individual execution time for the 100-city TSP with T_0=0.1

The curves shown in Figures 6 and 7 indicate that massively parallel or full scale implementation requires to reduce communication overhead if it is to be successful. In fact, we have spent much time to minimize communication overhead. We minimized the amount of data to broadcast in each iteration as follows: First, our implementation does not broadcast DATA in each iteration; Only two loop indices i and j are broadcast to all processors. Second, the two loop indices are packed in an integer before broadcasting. The packing can be done by simply multiplying i by the predefined constant c and add j to the result, i.e., $b = i*c + j$, where b is to be broadcast and c is a predefined constant. The receiving processor can unpack the single integer b to two loop indices and use them as a basis for its own iteration. The two loop indices can be easily recovered as follows: i = (int) b/c and $j=b-i*c$. This packing and unpacking method is also applied to *flag* and PID. We have tried other packing/unpacking techniques including the use of upper 16 bits for i and lower 16 bits for j but they made little difference in terms of execution time. Our experience suggests that the further improvement on massively parallel implementation of synchronous simulated annealing should be made on communication reduction if they are to be successful.

5.3 Effects of temperature on solution quality and speedup

Initial temperature is one of the most important parameters in simulated annealing. If temperature is too hot, it is more likely to accept many moves. If temperature is too cold, then it rarely accepts a new move. Figure 8 plots the relationship between initial temperature and speedup. When the initial temperature is below 10, the performance maintains over 20-fold speedup on 100 processors. However, as the initial temperature reaches 100, the performance becomes very poor, resulting in mere several-fold speedup. This initial temperature-speedup relation seems to suggest that the temperature of over 10 should not be used unless the solution quality of higher temperatures is much better than that of lower initial temperatures. It is found indeed true that raising the initial temperatures over 10 does not always give higher solution quality. To support this claim, we define the solution quality, $q = (l_0 - l_f)/l_0$, where l_0 is the initial tour length and l_f is the final tour length. The solution quality q is 1 for the perfect solution, i.e., the final tour length is 0.

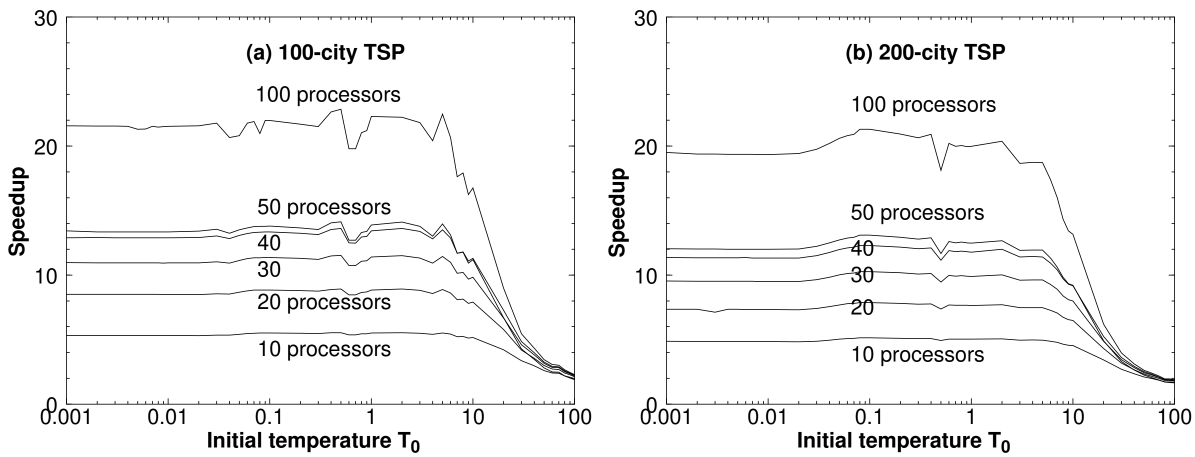

Figure 8: Speedup in terms of initial temperature. (a) 100-city, (b) 200-city

It should be noted that our parallel implementation has no relation to the solution quality. *The GSC gives the same solution quality as sequential version, regardless of the number of processors used.* Figure 9(a) demonstrates that solution quality increases very little even though initial temperature increases to 100. We find from the curves that for the 100-city TSP the solution quality in the high initial temperature range of 10 to 100 is different by only 1%. The solution quality for other two problems of 200-city and 300-city also gives a mere 1% difference. The 1% increase in solution quality requires very high price. The execution time at high initial temperatures is extremely high. Figure 9(b) plots execution time for the 200-city TSP on 1 to 100 processors. When the initial temperature is less than 5, the exe-

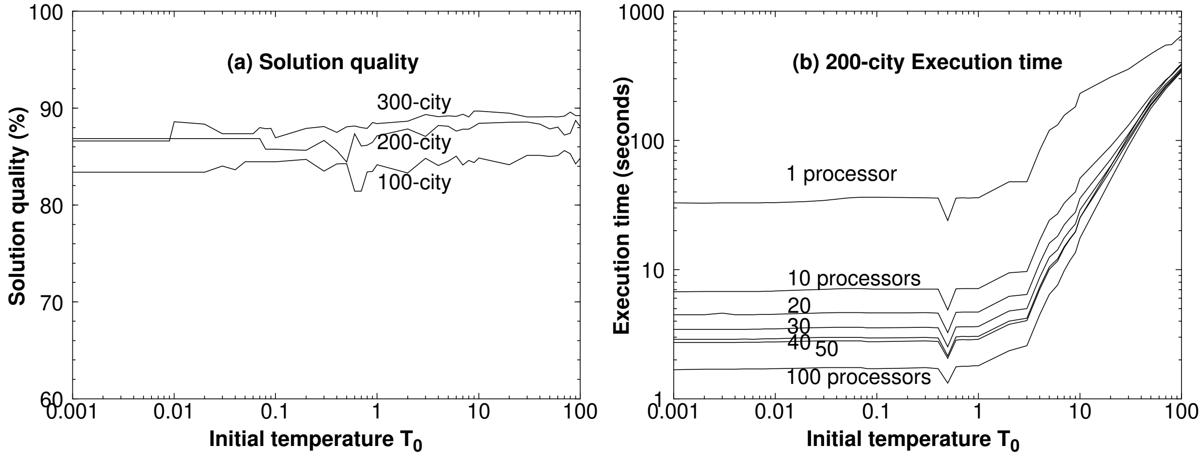

Figure 9: Comparison of solution quality and execution time. (a) solution quality, (b) execution time.

cution time remains fairly constant, approximately *one* second. However, when the initial temperature reaches over 5, execution time exponentially increases. To be more precise, execution time reaches 340 seconds when the initial temperature reaches 100 on 100 processors. The execution time has increased over 300 times! The solution quality, on the other hand, increased by merely 1% to 88% from 87%. This 300-fold increase in execution time for 1% increase in solution quality clearly suggests that the use of high initial temperatures is not necessarily a good strategy in simulated annealing unless the 1% improvement is worthier than the 300 times slower response.

6 Conclusions

Parallelizing synchronous simulated annealing for a massively parallel multiprocessor is a challenging task. The prohibitively long execution time of simulated annealing due to its sequential nature hinders its application for realistically sized problems. A simple-minded parallelization will likely result in poor performance. For small scale shared-memory multiprocessors, parallel implementation of synchronous simulated annealing does not address practical issues such as data distribution, communication, etc. for realistically sized problems. If the implementation is to be successful and efficient, those issues which are thought to be trivial for small scale shared-memory machines become critical. This report has presented a practical approach to parallel synchronous simulated annealing, called *generalized speculative computation* (GSC).

The generalized speculative computation approach is synchronous, maintaining the same decision sequence as sequential simulated annealing. We have discussed various practical issues of simulated annealing for massively parallel distributed-memory multiprocessors. The GSC method is designed to be able to execute up to N iterations in parallel. It uses an N-ary speculative tree, where N iterations can be executed simultaneously by N processors. The two loop indices, i and j, are used to uniquely determine the pseudo-random numbers. The use of a single integer which encodes two loop indices has completely eliminated broadcasting of central data structure to all processors and substantially simplified the communication between processors. The GSC approach has spent little time synchronizing loop iterations to maintain the same decision sequence as sequential simulated annealing. The master-slave parallel programming paradigm has simplified controlling the activities of N iterations which are executed in parallel on N processors. To demonstrate the performance of GSC, we have implemented the Traveling Salesman Problem on the AP1000 massively parallel multiprocessor.

We have selected the problem size of 100- to 500-city TSP for the initial temperatures of 0.001 to 100. All these problem sizes have been executed on 1 to 100 processors. Experimental results have demonstrated that the GSC method is effective for the problem instances we have implemented. The 500-city TSP for the initial temperature of 10 took roughly an half hour on a single processor and three minutes on 100 processors. We have obtained over 20-fold speedup on 100 processors. Experimental results have shown that increasing the initial temperature to 100 has improved the solution quality by merely 1% while it has increased the execution time by over 300 times! This has clearly indicated that the higher initial temperatures are not always practical, considering the 300 times increase in execution time for the 1% increase in solution quality. We have found that the communication time dominated the total computation time, almost 60% of the total execution time. The evaluation and decision time accounted for only 6% and the modify time for 35%. Experimental results have indicated that if the combined modify and evaluation time is greater than the communication time, the GSC approach will give even higher performance. In the near future, we will apply GSC to grid partitioning for computational fluid dynamics and structural analysis to solve real world problems.

Acknowledgments

The author would like to thank Fujitsu Parallel Computing Laboratory of Japan for providing access to the AP1000 multiprocessor. Special thanks go to Toshiyuki Shimizu of Fujitsu. Sequential version at high temperatures took a long time to execute. Without his timely help executing these programs, this work would still be in speculation. The author would like to acknowledge anonymous referees for various comments which helped improve the quality of this report.

References

[1] J. R. A. Allwright and D.B. Carpenter, "A Distributed Implementation of Simulated Annealing for the Traveling Salesman Problem," *Parallel Computing* 10, 1989, pp.335-338.

[2] P. Banerjee, M. H. Jones, and J. Sargent, "Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems* 1, January 1990, pp.91-106.

[3] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *IEEE Transactions on Computer Aided Design* 6, September 1987, pp.838-847.

[4] D. R. Greening, "Parallel Simulated Annealing Techniques," *Physica D* 42, 1990, pp.293-306.

[5] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata, "Improving AP1000 Parallel Computer Performance with Message Communication," in *Proc. ACM International Symposium on Computer Architecture*, San Diego, CA, May 1993, pp.314-325.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science* 220, 1983, pp.671-680.

[7] S. A. Kravitz and R. A. Rutenbar, "Placement by Simulated Annealing on a Multiprocessor," *IEEE Transactions on Computer Aided Design* 6, June 1987, pp.534-549.

[8] S. Y. Lee and K. G. Lee, "Asynchronous Communication of Multiple Markov Chains in Parallel Simulated Annealing," in *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1992, pp.III169-176.

[9] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, *Journal of Chemical Physics* 21, 1953, pp.1087-1091.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes in C, Cambridge University Press, Cambridge, U.K., 1992.

[11] J. S. Rose, W. M. Snelgrove, Z. G. Vranesic, "Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing," *IEEE Transactions on Computer Aided Design* 7, March 1987, pp.387-396.

[12] T. Shimizu, H. Ishihata, and T. Horie, "Low-Latency Message Communication Support for the AP1000," in *Proc. ACM International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp.288-297.

[13] A. Sohn, "Parallel Speculative Computation of Simulated Annealing," in *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1994, pp.III-8-11.

[14] E. E. Witte, R. D. Chamberlain, and M. A. Franklin, "Parallel Simulated Annealing Using Speculative Computation," *IEEE Transactions on Parallel and Distributed Systems* 2, October 1991, pp.483-494.

Biography

Andrew Sohn received a BS degree in Electrical Engineering (1985) and MS (1986) and Ph.D (1991) degrees in Computer Engineering, all from the University of Southern California. He is currently an assistant professor in the Computer and Information Science Department at New Jersey Institute of Technology. His research interests are in parallel computing, fine-grain multithreading, functional programming languages, and the solution of grand challenge problems on distributed-memory multiprocessors.

As a NASA/University Joint Venture Faculty Fellow, Sohn has been collaborating with researchers at the Numerical Aerodynamic Simulation Facility of the NASA Ames Research Center on dynamic load balancing for distributed-memory multiprocessors. He also has been selected three times as a research fellow to the Electrotechnical Laboratory of the Ministry of International Trade and Industry in Japan. His contribution at the Laboratory has been the investigation of the effects of multithreading on data and workload distribution for the EM-4 and EM-X distributed-memory machines.

Sohn is a member of IEEE, ACM, AAAI, and Eta Kappa Nu National Electrical Engineering Honor Society.