Annals of Operations Research 63(1996)29–55

Generalized speculative computation of parallel simulated annealing

Andrew Sohn

Department of Computer and Information Science,
New Jersey Institute of Technology, Newark, NJ 07102-1982, USA

E-mail: sohn@cis.njit.edu

Simulated annealing is known to be highly sequential due to dependences between iterations. While the conventional speculative computation with a binary tree has been found effective for parallel simulated annealing, its performance is limited to ($\log p$)-fold speedup due to parallel execution of $\log p$ iterations on p processors. This report presents a new approach to parallel simulated annealing, called *generalized speculative computation* (GSC). The GSC is synchronous, maintaining the same decision sequence as sequential simulated annealing. The use of two loop indices encoded in a single integer eliminates broadcasting of central data structure to all processors. The master-slave parallel programming paradigm simplifies controlling the activities of p iterations which are executed in parallel on p processors. To verify the performance of GSC, we implemented 100-city to 500-city Traveling Salesman Problems on the AP1000 massively parallel multiprocessor. Execution results on the AP1000 demonstrate that the GSC approach can indeed be an effective method for parallel simulated annealing as it gave over 20-fold speedup on 100 processors.

1. Introduction

Simulated annealing (SA) is an efficient method to optimization problems [10]. It is based on the analogy that the way metals cool and anneal as temperatures change. At high temperatures, the molecules of a metal move freely with respect to one another. As the metal is slowly cooled, they lose thermal mobility, thereby limiting their movements. At low temperatures, the molecules may then line themselves up to reach the state of minimum energy. Simulated annealing closely follows the above analogy by controlling temperature in an attempt to reach the minimum energy state for the given problem. Simulated annealing is typically implemented in two nested loops. The outer loop controls temperatures, while the inner loop iterates a fixed number of times for the given temperature. The inner loop typically consists of three steps: evaluation, decision, and modification (or move). The evaluation step evaluates the next possible state by generating random numbers and using evaluation criteria. The decision step decides if the evaluation performed is acceptable. If the decision is found

acceptable, the modification step will modify the central data according to what the evaluation step suggests. The SA technique has been applied to various optimization problems such as VLSI layout [4, 11, 13], the Traveling Salesman Problem [1,3,5], grid partitioning for computational fluid dynamics, satisfiability problems, etc.

The major hurdle of simulated annealing, however, is a large computation time due to its sequential nature of "slow" annealing process. To be more precise, simulated annealing is highly sequential due to dependences between iterations, or loop-carried dependences. This sequentiality limits its parallel execution in multiprocessor environments. Various approaches have therefore been taken to execute simulated annealing faster in multiprocessor environments which can be classified into two categories: (1) *synchronous* approach and (2) *asynchronous* approach [8]. The synchronous approach maintains the *same* decision sequence as sequential simulated annealing while trying to execute simulated annealing in parallel [11, 15, 17]. Those algorithms using the asynchronous approach violate the convergence properties of sequential simulated annealing. The errors introduced by this modified decision sequence will likely decrease solution quality and will likely converge to a bad local minimum [8].

This study concerns parallelizing synchronous simulated annealing. Among synchronous algorithms is *speculative computation* [17]. The idea is to simultaneously compute multiple decision sequences of a binary speculative tree using multiple processors. As a synchronous algorithm, speculative computation maintains the same decision sequence as sequential simulated annealing. Each processor has a role assigned to it to reflect the binary decision sequence of the speculative tree, which we call *binary speculative computation* (BSC). A processor can be either an accept or a reject processor. The reject processor directly proceeds to the next iteration, whereas the accept processor modifies the data before proceeding to the next iteration.

A major difficulty in binary speculative computation is that it requires a large number of processors [17]. A multiprocessor with p processors can execute a maximum of up to $(\log p)$ iterations in parallel. The maximum speedup is, therefore, limited to $(\log p)$-fold, where p is the number of processors. The second difficulty of BSC is complex communication and synchronization patterns due to controlling loop iterations. There must be a systematic and consistent way of keeping track of which processor is working on what iterations. The third difficulty also stems from independent function invocation on children or grandchildren processors. Sending a central data structure, or DATA, to children processors is costly for problems such as the Traveling Salesman Problem with 1000 cities. Sending DATA to a child processor entails sending 1000 integers, which would cause a substantial amount of communication overhead. Since each processor sends DATA to children processors *independent* of other processors, a good synchronization mechanism between the two processors is needed to ensure that all data are received before any computation proceeds.

It is precisely the main objective of this paper to attempt to solve the above three problems. This report introduces a new approach to parallel synchronous

simulated annealing. Our approach attempts to solve the problems by using an n-ary speculative tree, called *generalized speculative computation* (GSC). We generalize the speculation of (log p) iterations to p iterations. We use loop indices to execute p iterations in parallel on p processors. The use of loop indices reduces communication overhead as opposed to sending data to processors. To demonstrate this new approach, we implement the 100-city to 500-city Traveling Salesman Problem (TSP) on the 1024-processor AP1000 message-passing multiprocessor [15].

We begin our discussion in section 2 by giving a brief description of simulated annealing and binary speculative computation. Section 3 presents a new method to parallel SA. Section 4 lists implementation details and execution results of the TSP on the AP1000 multiprocessor. Section 5 analyzes our experimental results and compares them with binary speculative computation. We conclude our report in the last section and give future directions on parallel speculative computation of simulated annealing.

2. Simulated annealing with speculative computation

Simulated annealing is briefly described in this section, along with it sequentiality. Conventional binary speculative computation is reiterated as a solution to the sequential nature of simulated annealing. We identify several problems associated with binary speculative computation in multiprocessor environments.

2.1. SIMULATED ANNEALING

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 the temperature based on the cooling speed

```
T←T0                                ;initial temperature
for i=1 to n do {                   ;i-loop
    for j=1 to m do {               ;j-loop
1:      ΔE ← evaluate(DATA,i,j)     ;Select random numbers and compute ΔE
2:      flag ← decide(ΔE,Tj)        ;Decide if it is acceptable. flag = {OK,NOK}
3:      DATA ← modify(DATA,flag)    ;Modify DATA if flag is OK.
    }
    T ← T*CS                        ;Lower temperature. CS = cooling speed
}
```

Figure 1. Sequential simulated annealing.

(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 finds 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 typically uses the Metropolis equation to decide [12]. *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.

The above algorithm terminates either when all the n iterations of the outer loop are completed or when no iterations of the inner loop produce an OK flag. The inner j-loop iterates m times for a fixed temperature. If none of the m iterations of the j-loop produces an OK flag, then the whole process may terminate since it is unlikely that the next m iterations will produce an OK flag. Our experience tells us that when none of the m iterations of the j-loop produces an OK flag, executing another m iterations for lower temperature is very unlikely to improve the solution quality. This termination condition is still, however, implementation dependent and can be modified to suit to different problems.

2.2. BINARY SPECULATIVE COMPUTATION

The simulated annealing algorithm shown in figure 1 is sequential in two facts: 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) [17].

BSC assigns a role to each processor: root, accept, or reject. Figure 2 shows a typical binary speculative tree. Each processor iterates an altered sequence of modification, evaluation, and decision steps. The root processor which is working on iteration i sends a central data-structure (or DATA) to two (accept and reject) children 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.

The 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 $i + 1$. The 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 are no more processors left in the processor pool. When the processor pool is exhausted, the leaf processors send

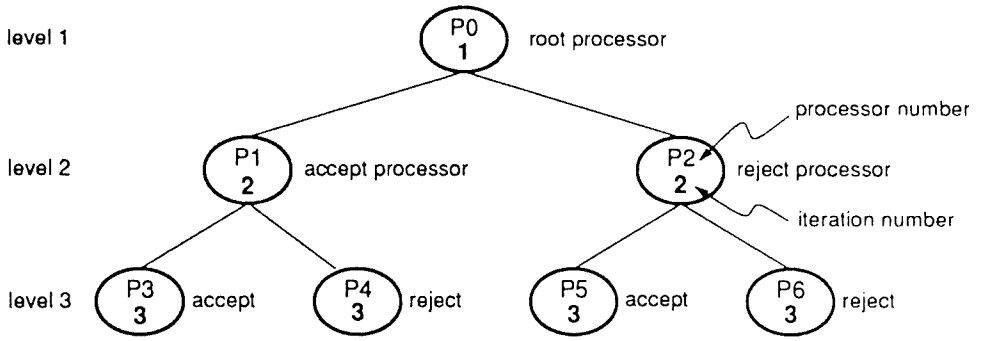

Figure 2. A binary speculative tree with three levels. p_0 is a root processor, p_1 an accept processor, and p_2 a reject processor. Seven processors execute up to three iterations in parallel. In general, BSC allows p processors to execute up to $\lceil \log p \rceil$ iterations in parallel.

their current *solutions* to the root processor, provided that the leaf nodes have finished their assigned computation. The root node than initiates another round of assigning roles to processors for newly initiated iterations. Details on communication and processor control are presented in [17].

Figure 3 shows a mapping of a binary speculative tree to a multiprocessor, where seven processors execute three iterations in parallel. Assume that processor 0 (P_0 labeled as root) has already performed evaluation and decision steps of the very first iteration [1, 0], which are not shown in the figure. Before and after the modification step of iteration [1, 1], DATA are sent to two children processors, P_1 and P_2. P_1 is labeled as reject since it will initiate iteration [1, 2] based on the same data used in iteration [1, 1]. P_1 assumes that the previous iteration will not modify DATA. P_2 is labeled accept since it initiates iteration [1, 2] based on the assumption that iteration [1, 1] will modify DATA. Three processors involved in computing two iterations therefore comprise one level of a speculative tree, or *level* 1.

The second level of a binary speculative tree starts whenever each child processor receives DATA, regardless of its status. When P_1 receives DATA, it immediately sends DATA to two grandchildren processors, P_3 and P_4, before and after the modification step. Similarly, P_2 sends DATA to P_5 and P_6 before and after the modification step. If P_0 decides OK (accept) in iteration [1, 1], then based on the new DATA P_2 has almost finished iteration [1, 2]. If P_2 decides NOK, then P_5 is near completion of iteration [1, 3]. If the decision sequence of the three iterations is accept-reject-reject (or OK-NOK-NOK), the three iterations [1, 1]...[1, 3] will take approximately half the time taken on a single processor.

3. Generalized n-ary speculative computation

We generalize binary speculative computation using an n-ary speculative tree, which we call generalized speculative computation. A detailed example is presented to show how the new method is mapped to a multiprocessor.

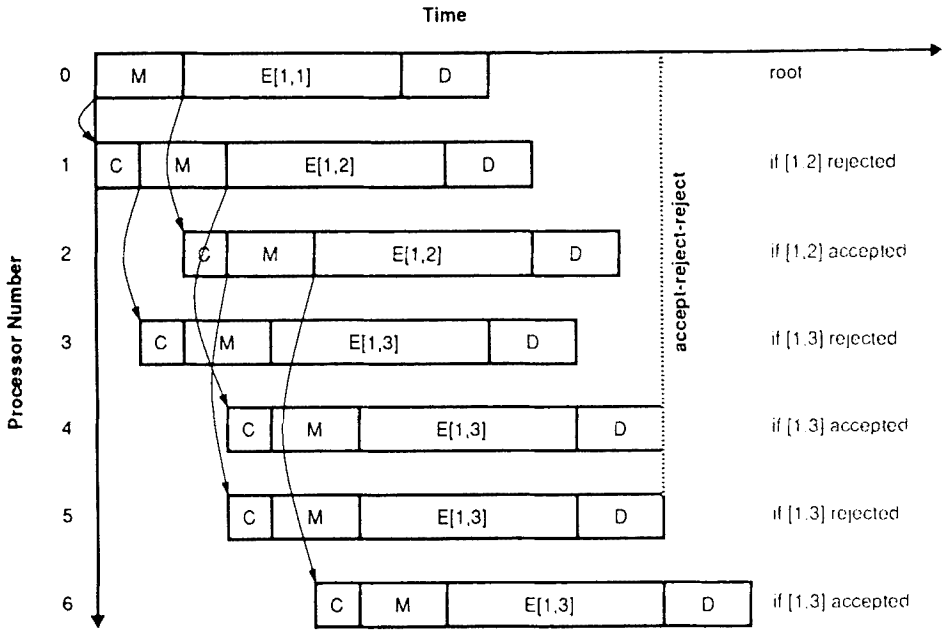

Figure 3. Speculative computation, where seven processors execute three iterations. E = evaluation, D = decision, M = modification, C = communication step. $E[i,j]$ indicates that the processor performs an evaluation step of $[i,j]$th iteration, where i = outer loop index and j = inner loop index.

3.1. n-ARY SPECULATIVE TREE

Generalized speculative computation uses an n-ary tree. Figure 4 shows an n-ary speculative tree with three levels. There is no notion of an accept or reject processor. No 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 resulting from the lowest numbered loop index is selected among the flags 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 nm seeds, where n is the maximum number of outer loop iterations and m is the maximum number of inner loop iterations. A loop index $[i,j]$ used as a seed therefore generates a unique random number. Figure 4 shows a total of 11 (*not* 16) iterations executed in three levels, as explained below.

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),

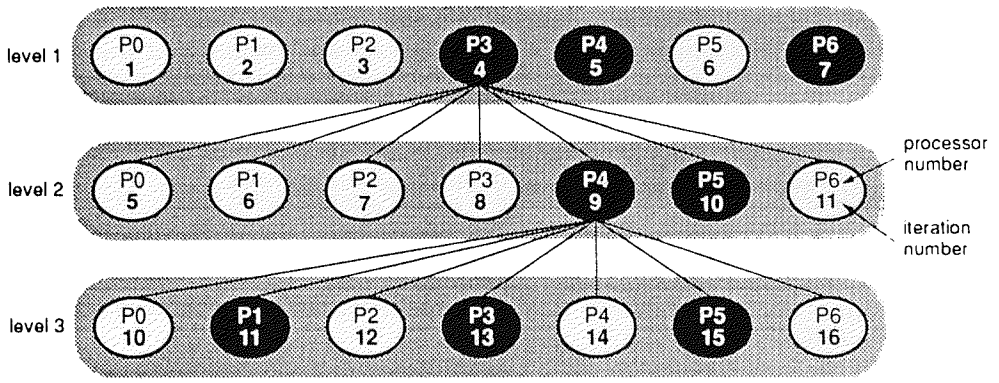

Figure 4. n-ary speculative tree, where n is the number of processors. Filled circles indicate acceptable decisions. N processors can execute up to n iterations. Each processor has its own loop index assigned to it in each level, based on which a unique random number is generated. Note that the above tree is determined only at runtime and has no relation with the interconnection network topology of the machine architecture.

processor 3, will however be 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 incorrect data.

The second level starts from iteration 5, executing iterations 5 to 11 by seven processors in parallel. Each processor receives a loop index 4, based on which it modifies the data. After modification, each processor starts evaluation and decision steps by computing its iteration index. It is straightforward to compute its own loop index 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 adds 1, resulting in 7. In other words, processor 2 now executes 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 the speculative tree. Note that the maximum number of iterations that can be computed by the same levels of the binary speculative tree is 3.

3.2. PARALLEL ALGORITHM

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-

```
p←number of processors; level ← 0; no_ok←0;   ;no_ok = number of OKs for inner loop
repeat                                        ;This is not a temperature loop
    parallel for k=1 to p {                   ;p = number of processors. p iterations are executed in parallel
1:      ΔE ← evaluate(DATA,i,j)               ;select random numbers and compute ΔE
2:      flag ← decide(ΔE)                     ;flag = OK if acceptable. Else NOK.
    }
3:  flags ← receive_from_all_processors       ;collect p flags from p processors
4:  i,j ← find_ok_index(flags);               ;Find a minimum index which gave OK
5:  no_ok ← count_ok(level,p,no_ok,flags);    ;Accumulate the number of OKs for a given temperature.
6:  send_to_all_processors(i,j)               ;Send to all processors the first OK loop index
    parallel for k=1 to p
7:      DATA ← modify(DATA,i,j)               ;p processors simultaneously modify DATA using a new index.
8:  level ← level + 1
Until (no_ok=0)                               ;number of OKs in an iteration of i-loop of Figure 1
```

Figure 5. Parallel simulated annealing using generalized speculative computation.

slave parallel programming paradigm. Given n processors, processor 0 is selected to be a master, while the rest will become slave processors. The master processor executes the algorithm shown in figure 5. Slave processors execute an instance of the repeat loop, except for the termination condition test. The repeat loop implements a level of the n-ary speculative tree. It is completely different from the temperature loop (i-loop) of figure 1 since an iteration of the repeat loop may execute j-loop instances which can come from more than one temperature.

For example, suppose that we have 500 processors and set the number of iterations for the j-loop to 150, i.e., $p = 500$ and $m = 150$, then each instance of the repeat loop can execute $\lceil p/m \rceil = \lceil 500/150 \rceil = 4$ iterations of the i-loop in parallel. A level (one repeat loop instance) activates p processors, where each processor performs one $[i, j]$ iteration. Recall that figure 1 has a maximum of nm loop iterations. One level of the n-ary speculative tree is equivalent to parallel execution of p iterations among $n * m$ iterations of figure 1.

Lines 1 and 2 of figure 5 perform evaluation and decision steps as before, but p evaluations and p decisions are executed in parallel by p processors. All p processors work on the same data but different random numbers. Since each iteration is uniquely identified by a loop index $[i, j]$, this parallel execution of p iterations is possible. Once p processors simultaneously execute p iterations, line 3 (receive_all_processors) collects *flags* to identify which processor(s) is (are) successful. Line 4 then finds among successful processors the minimum numbered-loop index. Since each processor has a unique loop index $[i, j]$ assigned to it, it is simple to identify which one among the successful iterations is the lowest numbered-loop index. The master then sends this new loop index to all processors (line 6). Note that the binary speculative version shown in figures 2 and 3 sends all DATA to child processors. Our GSC approach sends a single loop index only, which will substantially reduce communication time. In general, binary speculative computation sends $O(n)$ data to child processors, while GSC send $O(1)$ data to all processors, where n is a problem size.

Upon receiving the new loop index, all processors modify DATA in parallel by using the new loop index (line 7). Again, since each loop index $[i, j]$ uniquely determines random numbers, this parallel modification is possible. Line 5, count_ok, prepares for the termination condition test. It simply counts how many successful decisions there are in each iteration of the i-loop. Note that we do not count the number of OKs for each level, as each level may have more than one iteration of the i-loop. If there is no OK in an iteration of the i-loop (or all iterations of the j-loop), then we stop the whole process. If there is at least one OK in among m iterations of the j-loop, then the master processor continues to the next level.

3.3. IMPLEMENTATION IN MULTIPROCESSOR ENVIRONMENTS

Figure 6 gives a typical mapping of generalized speculative computation to multiprocessors. To illustrate the difference between BSC and GSC, we use the speculative tree shown in figure 4 and *seven* processors. Let P_0 be a *master* processor and $P_1,\ldots,P_6$ be *slave* processors. GSC proceeds as follows:

- At the beginning, the master processor sends the loop index [1, 1] to all slave processors. After sending, it starts working on the evaluation and decision steps.
- 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 reports its result (*flag*) and PID to the master.
- When P_0 finishes the evaluation and decision steps of [1, 1], it collects seven flags from six slave processors and itself.
- 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.
- 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.
- As soon as each processor finishes the modification step, it immediately starts the second level with no interruption.

Figure 6 shows that the GSC method executes 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

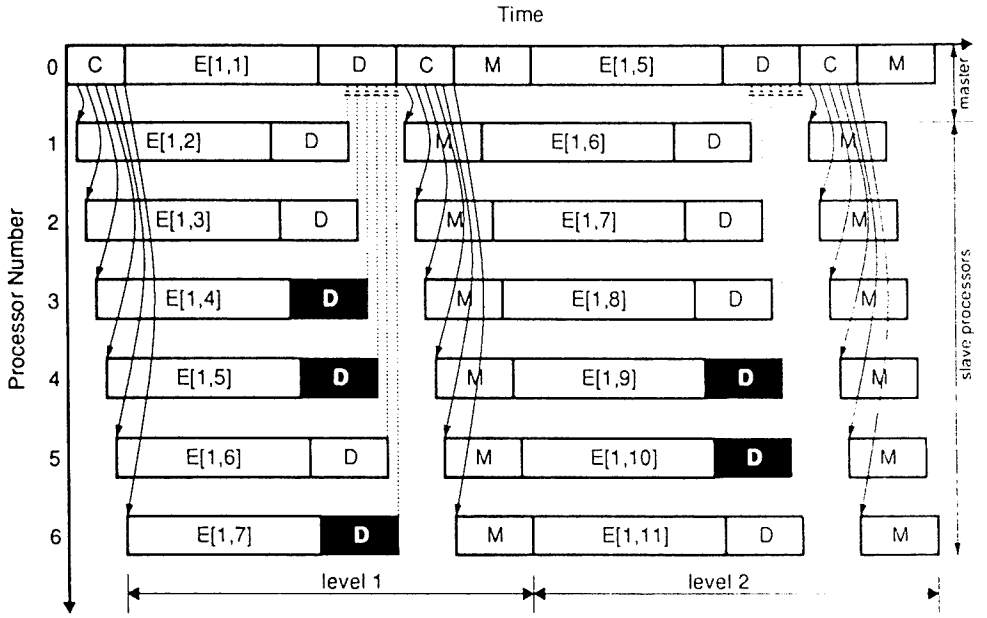

Figure 6. GSC on seven processors. P_0 is a master, while $P_1, \ldots, P_6$ are slaves. Solid arrows indicate broadcasting of a single integer $[i, j]$ to all processors, while dotted arrows send *flags* to the master. Filled squares indicate that the decisions are acceptable, i.e., *flag* = OK. The decision made by the lowest-numbered loop index is used for the 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.

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. 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}, $1 \leq k \leq n - i - 1$, are invalid due to the loop-carried dependence between iterations.

4. Experimental results

This section presents our implementation of generalized speculative computation on the AP1000 multiprocessor. We give a brief explanation of the AP1000 architecture and its programming environment, followed by a list of actual execution results.

4.1. THE AP1000 MULTIPROCESSOR AND ITS PROGRAMMING ENVIRONMENT

Our target problem is the well-known Traveling Salesman Problem (TSP), where the salesman is to find the shortest path while visiting c cities once each. We

have used the 100-city 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 multicomputer built and operational at the Fujitsu parallel computing laboratory. The main reasons we used the machine are because (1) it is available to us, and (2) it has a large number of processors which would allow us to perform full-scale experiments. Figure 7 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

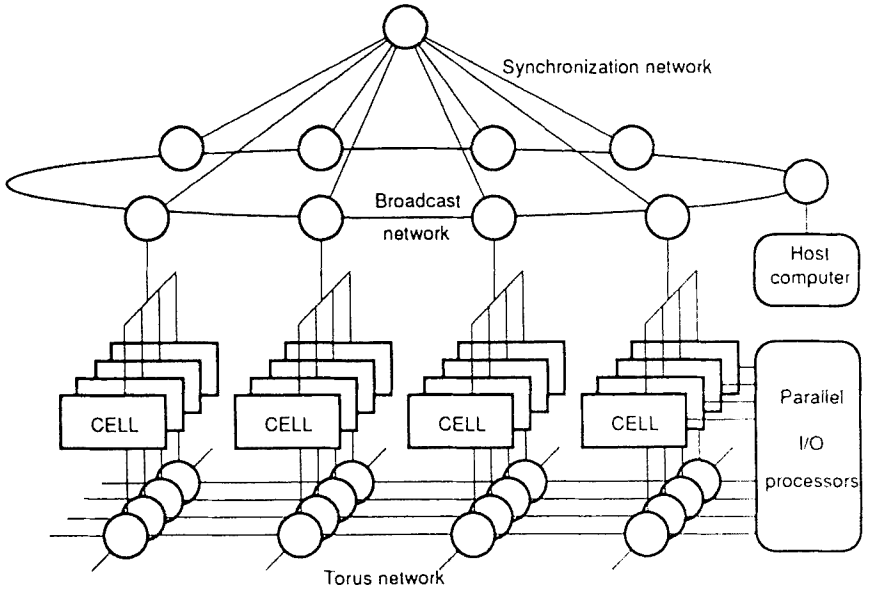

Figure 7. The AP1000 architecture. The broadcast network allows to broadcast data to all cells from cells or host. The torus network is designed for point-to-point communication between cells, while the synchronization network allows synchronization between cells or groups of cells.

network (S-net). A host computer, Sun 4, 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 cell. The S-net is used to synchronize various activities of cells, while the T-net allows to communicate between cells. Each cell of the AP1000 consists of an integer unit, a floating point unit, a message controller, network interface units, a 128 KB cache, and a 16 MB of memory. More details on the processor architecture are presented in [14].

All the programs 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 that the CASIM simulator running on workstations is particularly helpful in developing our programs since we do not have to directly deal with the real machine for debugging. What we did was that we wrote a program and debugged using CASIM on a workstation. Once the program was verified that it was free of bugs, we executed the program on the real machine. There are certainly limitations to using the simulator for debugging since some subtle bugs, such as synchronization bugs, are not caught in the simulation environment. This type of synchronization bugs is usually detected in the real machine environment because of very subtle timing. We were fortunate that we did not have such bugs in our implementation, mainly because of the master-slave parallel programming paradigm.

4.2. EXECUTION RESULTS

We have selected five different problem sizes, 100-city to 500-city TSP. For each problem size, we use the initial temperatures of 0.001 to 100. All these problem sizes are executed on 1 to 100 processors. 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}: c, where c is the number of cities.
- Number of iterations for inner loop j_{max}: c^2.

Tables 1 to 3 list execution results for the initial temperatures of 0.1, 1 and 10. Table 4 lists execution times of the individual steps for the 100-city TSP with an initial temperature of 0.1. Recall that each processor executes a unique iteration consisting of four steps: communicate, modify, evaluate, and decide. Figure 5 defined a level in which p processors execute p different iterations. The total number of processors used for a particular run uniquely determines the total number of levels. The 100-city TSP on 10 processors, for example, executed 4283 levels of a 10-ary speculative tree. 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 4 is due to the measurement of individual execution time. To measure the individual time, we

Table 1

Execution time (in seconds) of the TSP on the AP1000 multiprocessor for $T_0 = 0.1$.

No. of processors	100-city	200-city	300-city	400-city	500-city
1	12.64	39.08	87.62	105.59	165.96
2	7.67	24.04	53.53	65.30	102.39
4	4.54	14.56	32.19	39.95	62.84
6	3.35	10.93	23.97	30.27	47.74
8	2.70	8.89	19.42	24.76	39.55
10	2.31	7.69	16.74	21.64	34.24
20	1.45	5.02	10.78	14.60	23.26
30	1.14	3.87	8.26	11.33	17.99
40	0.98	3.27	6.89	9.57	15.26
50	0.91	2.97	6.37	8.79	13.86
60	0.84	2.69	5.75	8.01	12.63
70	0.75	2.38	5.16	7.20	11.31
80	0.71	2.23	4.84	6.74	10.54
90	0.67	2.10	4.59	6.38	9.89
100	0.64	2.01	4.31	6.04	9.37

Table 2

Execution time for $T_0 = 1$.

No. of processors	100-city	200-city	300-city	400-city	500-city
1	15.607	38.627	86.555	104.413	245.780
2	9.488	23.873	53.041	64.703	152.038
4	5.629	14.549	31.943	39.629	93.351
6	4.156	10.941	24.013	30.064	70.828
8	3.376	8.935	19.430	24.870	58.038
10	2.872	7.739	16.739	21.644	50.714
20	1.798	5.114	10.865	14.715	34.233
30	1.410	3.972	8.410	11.504	26.344
40	1.216	3.399	7.040	9.720	22.017
50	1.083	2.987	6.231	8.611	19.363
60	0.990	2.733	5.710	7.937	17.608
70	0.940	2.532	5.287	7.355	16.239
80	0.879	2.383	4.982	6.900	15.041
90	0.837	2.250	4.739	6.542	14.067
100	0.802	2.172	4.457	6.213	13.270

Table 3

Execution time for $T_0 = 10$.

No. of processors	100-city	200-city	300-city	400-city	500-city
1	50.207	220.814	557.386	961.458	1442.729
2	31.160	139.124	351.433	614.590	932.021
4	18.964	86.853	219.984	393.084	604.979
6	14.293	66.682	169.461	306.327	480.862
8	11.817	55.801	141.993	263.059	413.303
10	10.322	48.988	125.482	235.214	372.699
20	7.028	34.737	90.541	177.846	289.747
30	5.911	28.718	76.546	152.092	251.721
40	5.434	25.898	69.527	139.044	232.273
50	5.135	24.089	65.654	131.651	221.793
60	4.938	23.004	64.364	127.519	216.423
70	4.853	22.196	62.044	124.787	213.040
80	4.772	21.592	61.104	122.375	209.517
90	4.708	21.198	60.755	120.901	207.251
100	4.663	20.886	59.593	119.426	205.563

had to insert many system calls in the program and also needed to interact with the host. Minor program statements such as conditional constructs and loop control constructs were not included in the measurement.

5. Performance evaluation and discussion

In this section, we analyze experimental results to evaluate the performance of GSC. 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 are 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 TO THE SPEEDUP

The execution times shown in tables 1 to 3 are now converted to speedup curves to identify the effectiveness of the GSC method. Speedup is defined as the execution time on 1 processor over p processors for the given problem. Table 5 lists speedup and figure 8 shows their corresponding plots.

Table 4

Execution time of individual steps for the 100-city TSP with $T_0 = 0.1$, C = communicate, M = modify, E = evaluate, D = decide.

No. of processors	Overall execution time (seconds) for 100-city, $T_0 = 0.1$				No. 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	9.000367	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 5

Speedup of the TSP using generalized speculative computation on the AP1000.

No. of processors	$T_0 = 0.1$					$T_0 = 1$					$T_0 = 10$				
	100	200	300	400	500	100	200	300	400	500	100	200	300	400	500
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.6	1.5
3	2.3	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.1	2.1	2.1	2.0
4	2.8	2.7	2.7	2.6	2.6	2.8	2.7	2.7	2.6	2.6	2.6	2.5	2.5	2.4	2.4
5	3.3	3.2	3.2	3.1	3.1	3.3	3.1	3.2	3.1	3.1	3.1	3.0	2.9	2.8	2.7
6	3.8	3.6	3.7	3.5	3.5	3.8	3.5	3.6	3.5	3.5	3.5	3.3	3.3	3.1	3.0
7	4.3	4.0	4.1	3.9	3.9	4.3	4.0	4.1	3.9	3.9	3.9	3.7	3.7	3.4	3.3
8	4.7	4.4	4.5	4.3	4.2	4.6	4.3	4.5	4.2	4.2	4.2	4.0	3.9	3.7	3.5
9	5.1	4.7	4.9	4.6	4.5	5.1	4.6	4.8	4.5	4.5	4.6	4.2	4.2	3.9	3.7
10	5.5	5.1	5.2	4.9	4.8	5.4	5.0	5.2	4.8	4.8	4.9	4.5	4.4	4.1	3.9
20	8.8	7.8	8.1	7.2	7.1	8.7	7.6	8.0	7.1	7.2	7.1	6.4	6.2	5.4	5.0
30	11.1	10.1	10.6	9.3	9.2	11.1	9.7	10.3	9.1	9.3	8.5	7.7	7.3	6.3	5.7
40	12.9	12.0	12.7	11.0	10.9	12.8	11.4	12.3	10.7	11.2	9.2	8.5	8.0	6.9	6.2
50	13.9	13.2	13.8	12.0	12.0	14.4	12.9	13.9	12.1	12.7	9.8	9.2	8.5	7.3	6.5
60	15.1	14.5	15.2	13.2	13.1	15.8	14.1	15.2	13.2	14.0	10.2	9.6	8.8	7.5	6.7
70	16.9	16.4	17.0	14.7	14.7	16.6	15.3	16.4	14.2	15.1	10.3	9.9	9.0	7.7	6.8
80	17.9	17.5	18.1	15.7	15.7	17.8	16.2	17.4	15.1	16.3	10.5	10.2	9.1	7.9	6.9
90	19.0	18.6	19.1	16.6	16.8	18.6	17.2	18.3	16.0	17.5	10.7	10.4	9.2	8.0	7.0
100	19.6	19.5	20.3	17.5	17.7	19.5	17.8	19.4	16.8	18.5	10.8	10.6	9.4	8.1	7.0

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 increases 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 a larger 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 temperatures of 1 and 10 show a noticeable difference.

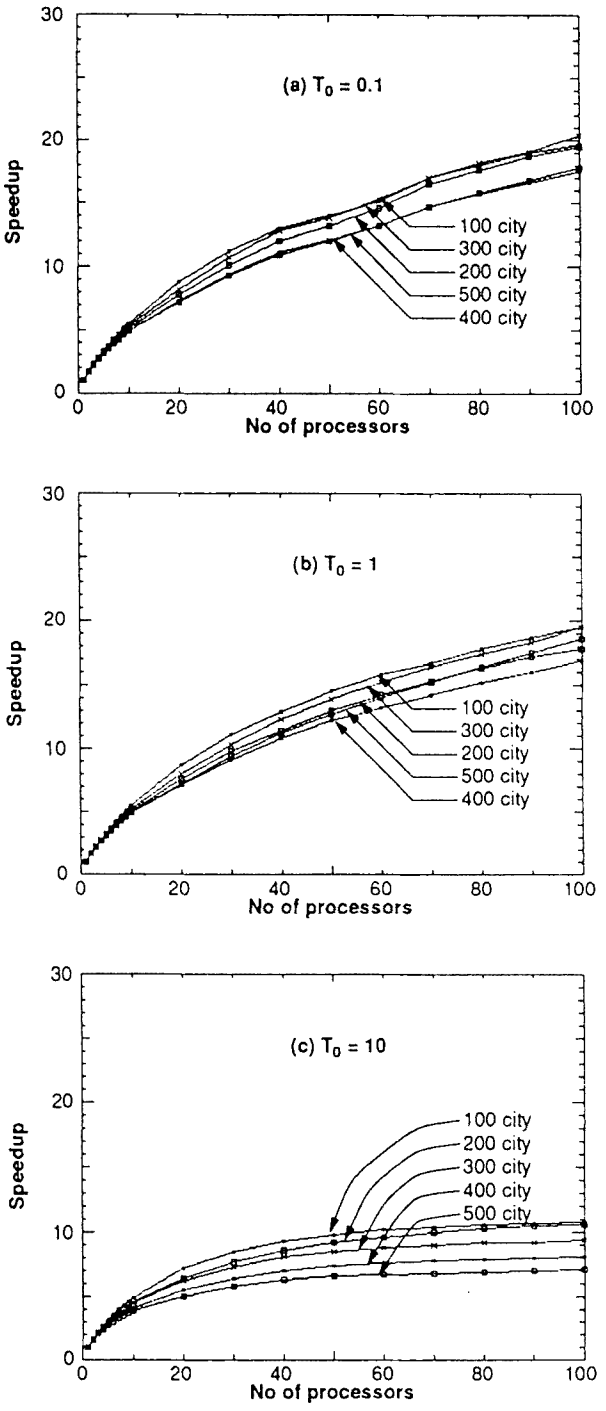

Figure 8. Speedup of the Traveling Salesman Problem on the AP1000 multiprocessor, using simulated annealing with generalized speculative computation. (a) $T_0 = 0.1$, (b) $T_0 = 1$, (c) $T_0 = 10$.

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 100×100 matrices can likely give more speedup than that of 10×10 when a large number of processors, say 100, is used. In other words, for a large multiprocessor, speedup is likely to increase as the problem size increases. However, this is certainly not the case for our experiments, as figure 8 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) 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 do 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 the 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 times for individual steps. Figure 9 shows 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 p processors execute p different iterations. The y-axis shows actual execution time in *milli*seconds for each level. Figure 9(a) plots execution time for 10 processors which executed 4283 levels, while figure 9(b) plots for 100 processors which executed 683 levels. Since each level consists of p processors speculating on p iterations, the number of levels becomes smaller when a larger number of processors is used (table 4).

The bottom curve of each plot indicates evaluation and decision time, while the two upper curves indicate communication and modification time, often super-

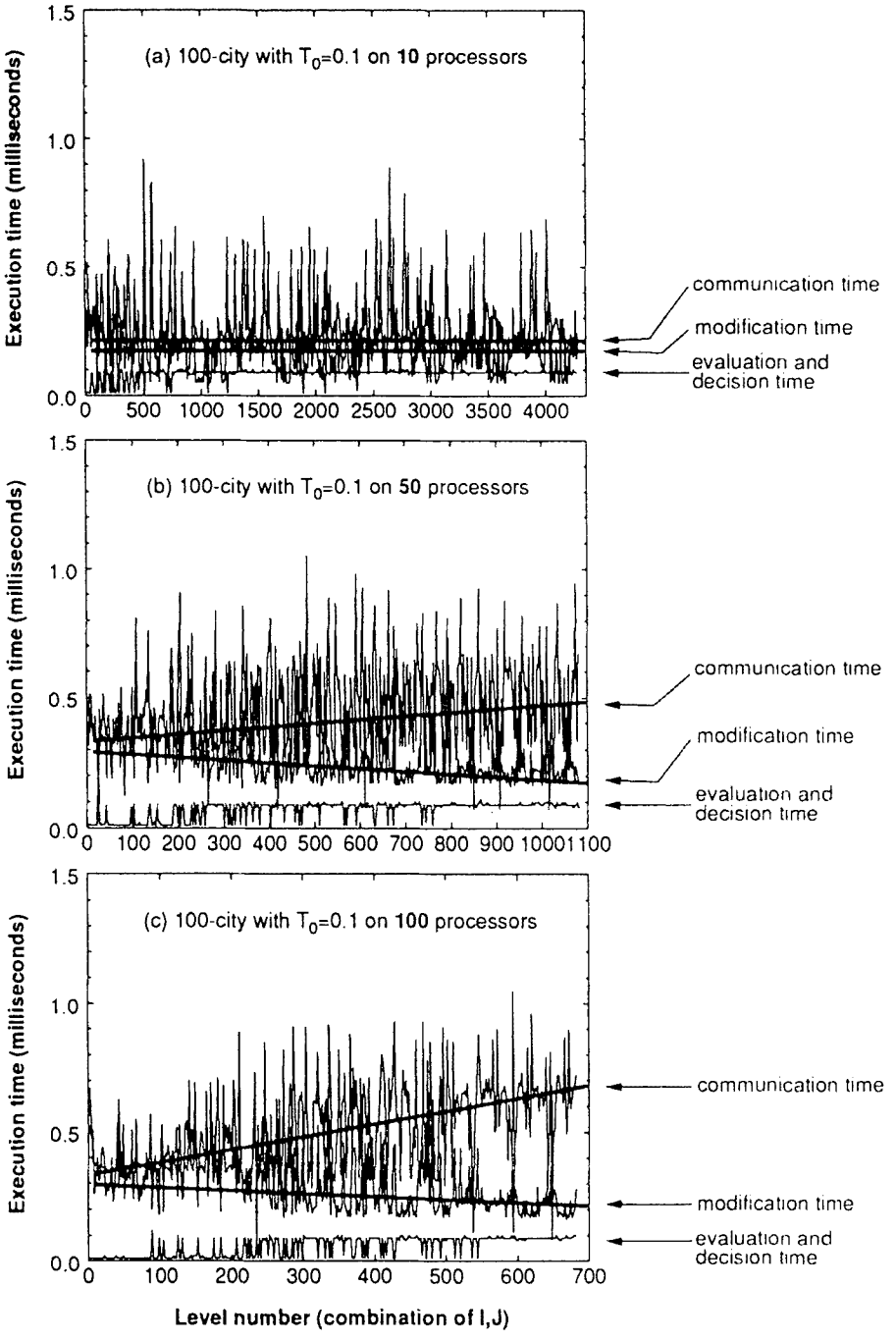

Figure 9. Execution time of individual steps (millisec).
(a) 10 processors, (b) 50 processors, (c) 100 processors.

imposed. 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 9(a) but becomes visible in figures 9(b) and 9(c).
- Communication time increases as the number of processors increases. Figure 9(c) clearly shows that there is a substantial increase in communication time. Communication time dominates the four steps for 100 processors.

Figure 10 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.

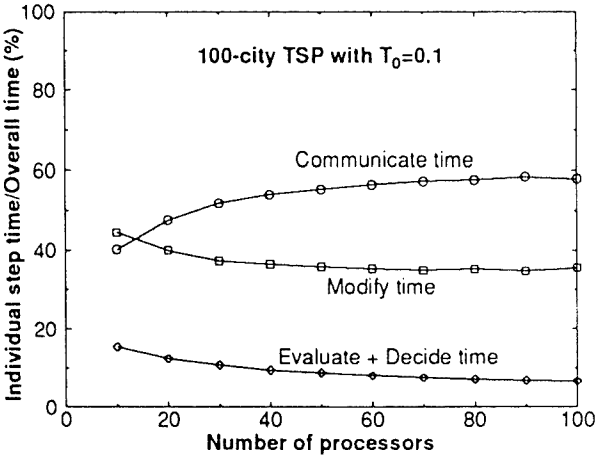

Figure 10. Comparison of individual execution time for the 100-city TSP with $T_0 = 0.1$.

The curves shown in figures 9 and 10 indicate that a massively parallel or full-scale implementation requires the reduction of communication overhead if it is to be successful. In fact, we have spent much time to minimize communcation 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 adding 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: $i = (\text{int})b/c$ 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 the upper 16 bits for i and the lower 16 bits for j, but they made little difference in terms of execution time. Our experience suggests that further improvements 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

The initial temperature is one of the most important parameters in simulated annealing. If the temperature is too hot, it is more likely to accept many moves. If the temperature is too cold, then it rarely accepts a new move. Figure 11 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 a 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 temperature 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, which is impractical.

It should be noted that our parallel implementation has no relation to solution quality. *The GSC gives the same solution quality as the sequential version, regardless of the number of processors used.* We could have used the approximation technique presented in [3], but did not do so because of many practical differences such as annealing schedule, error rate, initial random city configurations, etc. Instead, we present plots to explicate the relation between initial temperature, solution quality against initial configuration, execution time, and speedup. Figure 12(a) demonstrates that solution quality increases very little even though the 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 the other two problems of 200-city and 300-city also gives a mere 1% difference.

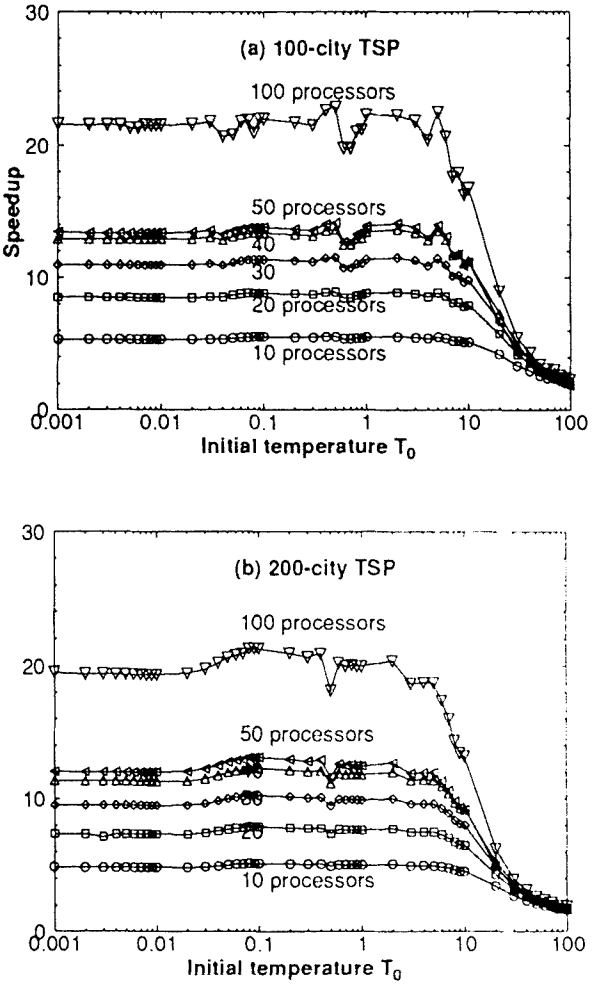

Figure 11. Speedup in terms of initial temperature.
(a) 100-city, (b) 200-city.

The 1% increase in solution quality requires a very high price. The execution time at high initial temperature is extremely high. Figure 12(b) plots the execution time for the 200-city TSP on 1 to 100 processors. When the initial temperature is less than 5, the execution time remains fairly constant, approximately *one* second. However, when the initial temperature exceeds 5, execution time increases exponentially. 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 a 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.

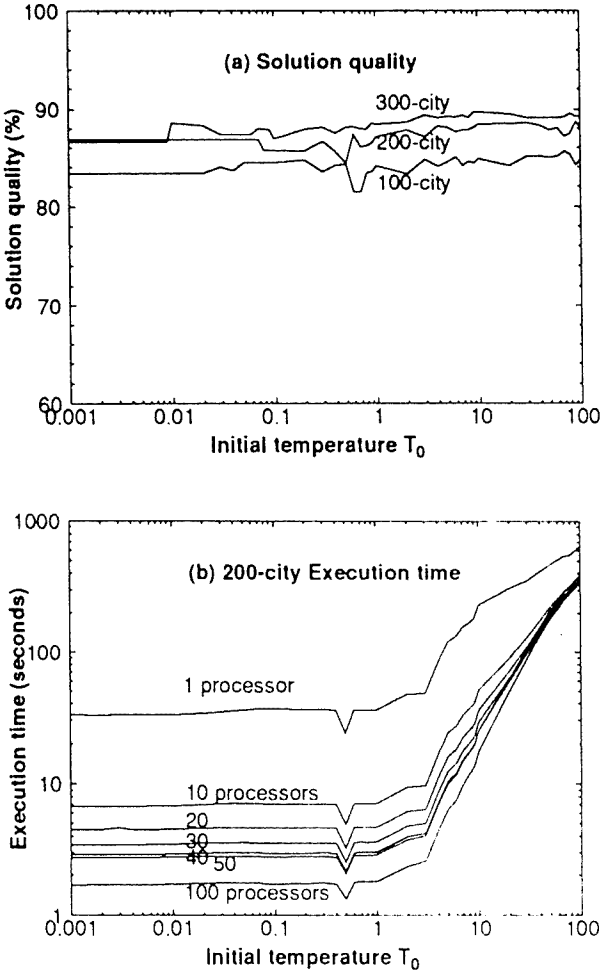

Figure 12. Comparison of solution quality and execution time.
(a) Solution quality, (b) execution time.

5.4. SUMMARY

Our goal in this study is to execute simulated annealing quickly using a massively parallel multiprocessor. We have introduced generalized speculative computation which can speculate p different iterations in parallel on p processors. To verify the performance of GSC, we have implemented our approach on the AP1000 massively parallel multiprocessor. The following summarizes our findings:

- *The GSC approach is synchronous, giving the same decision sequence and solution quality as the sequential simulated annealing but 20 times faster.* The solution quality depended only on the initial temperature. Various experimental

results demonstrated that the approach can give over 20-fold speedup on 100 processors while maintaining the same decision sequence and solution quality as the sequential version.

- *The GSC approach is designed to be able to execute up to p iterations in parallel on p processors.*
- *The GSC approach simplifies communication.* By sending a single encoded integer of two loop indices to all processors, communication is reduced to the simplest. The loop index uniquely determines the pseudo-random numbers of the entire simulated annealing process. In general, our approach sends $O(1)$ data, regardless of problem size.
- *The GSC approach simplifies controlling the parallel execution of p loop iterations and is suitable for massively parallel distributed-memory multiprocessors.* The simplicity in iteration control is made possible by using the master-slave parallel programming paradigm and an encoded loop index.

We find that the GSC method is effective for both low and high initial temperatures of up to 10. Considering that simulated annealing is highly sequential due to loop-carried dependence, the speedup of over 20 is encouraging. However, experimental results also indicate that the Traveling Salesman Problem spends nearly 60% of the total execution time in communication. We believe that the performance of GSC will proportionally increase for problems which spend more time in computation. To be more specific, if the ratio of computation to communication is greater than one, i.e., more computation than communication, it is very likely that the GSC will perform better than what is presented in this report. We are currently investigating the possibility of applying generalized speculative computation to other problems such as satisfiability problems, VLSI cell placement, grid partitioning for computational fluid dynamics [16], etc. Implementing other problems with realistic problem sizes, we will be able to further identify the effectiveness of the generalized speculative computation presented in this report.

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-scaled 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 p iterations in parallel. It uses a p-ary speculative tree, where p iterations can be executed simultaneously by p 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 p iterations which are executed in parallel on p 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 half an hour on a single processor and three minutes on 100 processors. We have obtained over 20-fold speedup on 100 processors. While the performance of conventional speculative computation has been limited to $(\log p)$-fold speedup on p processors, the GSC method has demonstrated that it can give a maximum of p-fold speedup. To achieve the 20-fold speedup, the conventional speculative computation needs a minimum of 1048576 (2^{20}) processors! The GSC method, on the other hand, needed only 100 processors.

Experimental results have shown that increasing the initial temperature to 100 improved the solution quality by merely 1%, while it has increased the execution time by over 300 times! This has clearly indicated that 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. The evaluation and decision time accounted for only 6% and the modify time for 35%. The communication time, on the other hand, has reached almost 60% of the total execution time. This domination of communication time certainly prefers problems with longer modification time, and evaluation and decision time. Experimental results have indicated that if the ratio of the combined modify and evaluate time to the communication time is greater than 1, the GSC approach will give even better performance.

The generalized speculative computation can be a *practical* approach to parallel simulated annealing on *massively parallel* distributed-memory multiprocessors for

realistically sized problems. We believe that the GSC approach can be applicable to different problems, including satisfiability problems, VLSI cell placement, graph partitioning, etc. If a massively parallel machine can solve a problem with simulated annealing which used to take a day on a sequential machine in an hour, we believe it is practical and it will be able to find applications in real-world situations. We are currently planning on investigating the possibility of applying the GSC to other problems and at the same time developing more efficient synchronous simulated annealing algorithms for realistic applications.

Acknowledgements

The author would like to thank the members of the Fujitsu Parallel Computing Laboratory of Japan for providing access to the AP1000 mutiprocessor. Special thanks go to Toshiyuki Shimizu of Fujitsu Laboratory. Without his timely help, this work would still be in speculation.

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)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(1990) 91–106.

[3] E. Bonomi and J.-L. Lutton, The *N*-city Traveling Salesman Problem: Statistical mechanics and the Metropolis algorithm, SIAM Review 26(1994)551–568.

[4] A. Casotto, F. Romeo and A. Sangiovanni-Vincentinelli, A parallel simulated annealing algorithm for the placement of macro-cells, IEEE Transactions on Computer-Aided Design 6(1987)838–847.

[5] E. Felton, S. Karlin and S. Otto, The Traveling Salesman Problem on a hypercube, MIMD Computer, in: *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, IL (1985) pp. 6–10.

[6] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors*, Vol. I (Prentice-Hall, 1988).

[7] S.B. Gelfand and S.K. Mitter, Simulated annealing with noisy or imprecise energy measurements, Journal of Optimization Theory Application (1989).

[8] D.R. Greening, Parallel simulated annealing techniques, Physica D 42(1990)293–306.

[9] L.K. Grover, Simulated annealing using approximate calculation, Progress in Computer-Aided VLSI Design 6(1989).

[10] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, Optimization by simulated annealing, Science 220 (1983)671–680.

[11] S.A. Kravitz and R.A. Rutenbar, Placement by simulated annealing on a multiprocessor, IEEE Transactions on Computer-Aided Design 6(1987)534–549.

[12] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller, Journal of Chemical Physics 21(1953)1087–1091.

[13] J.S. Rose, W.M. Snelgrove and Z.G. Vranesic, Parallel standard cell placement algorithms with quality equivalent to simulated annealing, IEEE Transactions on Computer-Aided Design 7(1987) 387–396.

[14] T. Shimizu, H. Ishihata and T. Horie, Low-latency message communication support for the Ap1000, in: *Proceedings of the 19th ACM International Symposium on Computer Architecture*, Gold Coast, Australia (May 1992) pp. 288–297.

[15] A. Sohn, Parallel speculative computation of simulated annealing, in: *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL (August 1994) pp. III-8-11.

[16] A. Sohn, R. Biswas and H. Simon, A dynamic load balancing framework for unstructured adaptive computations on distributed-memory multiprocessors, in: *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy (June 1996).

[17] E.E. Witte, R.D. Chamberlain and M.A. Franklin, Parallel simulated annealing using speculative computation, IEEE Transactions on Parallel and Distributed Systems 2(1991)483–494.