MULTITHREADED SIMULATED ANNEALING

Leonardo Jelenković, Joško Poljak
Faculty of Electrical Engineering and Computing, University of Zagreb
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Unska 3, 10000 Zagreb, Croatia
{leonardo.jelenkovic, josko.poljak}@fer.hr

Abstract: *Simulated annealing is known to be an efficient method for combinatorial optimization problems. Its usage for real-life problems has been limited by the long execution time. This report presents a new approach to asynchronous simulated annealing for parallel thread oriented multiprocessor operating systems. Experimental results of the 100- to 1000-city traveling salesman problems on the two-processor Ultrasparc II workstation shows the efficiency of this technique.*

Keywords: *parallel simulated annealing, multithreading, traveling salesman problem*

1. Introduction

Simulated annealing is one of highly efficient methods for combinatorial optimization problems. The method is developed from the annealing process, where with slow temperature decrease metal obtains a structure with state of minimal energy. To simulate this process N. Metropolis used the following method: from the current configuration one element is randomly chosen and a slight modification is performed. If modification decreases the energy the new configuration is accepted. Otherwise, probability of acceptance is given with $\exp(-\Delta E/(k_B T))$, where ΔE is energy change, k_B is Boltzman constant and T is the current metal temperature. After many similar steps, metal obtains temperature equilibrium at the current temperature. The procedure is then repeated with lower temperature, and so on, until the metal is cooled. If the current metal configuration represents one possible state of the combinatorial optimization problem, the energy represents a cost function, and T is one of controlling parameters, the above method could be applied to reach the minimum of a given combinatorial problem. A typical implementation of simulated annealing consists of two nested loops. The outer loop controls the temperature and the inner loop performs evaluation, decision, and modification steps at a given temperature. Outer loop usually decreases the temperature by a constant factor between 0.5 and 1. The inner loop is usually performed a constant number of times that depends on the problem size and the expected result quality, typically βN^2 ($0 < \beta < 1$), where N represents the problem size.

The method has been successfully applied to various problems such as cell placement, task scheduling, traveling salesman problem and graph partitioning. The main disadvantage of the algorithm is a long computation time. Several papers propose methods for implementing the simulated annealing on parallel architectures to speedup the computation. Parallel simulated annealing can be classified into two categories: *synchronous* and *asynchronous*. The synchronous approach maintains the same decision sequence as sequential simulated annealing, while the asynchronous approach does not.

In A. Sohn paper [5] on traveling salesman problem, a 20-fold speedup on a 100 processor system (AP1000 message passing multiprocessor) is presented. The authors used N-parallel simulated annealing method (synchronous approach), where the N processors speculatively compute N iterations in one computing level.

Fig. 1 shows execution of 7 iterations in two levels. The first level of N-ary speculative tree shows seven processors executing seven iterations simultaneously. Suppose that processors 3, 5 and 6 find that their decisions are acceptable. Among the three, only the decision made by the lowest numbered processor, processor 3, is accepted since the decisions made by the higher numbered processors are wrong because they are based on wrong configuration. The next iteration level begins with fifth iteration.

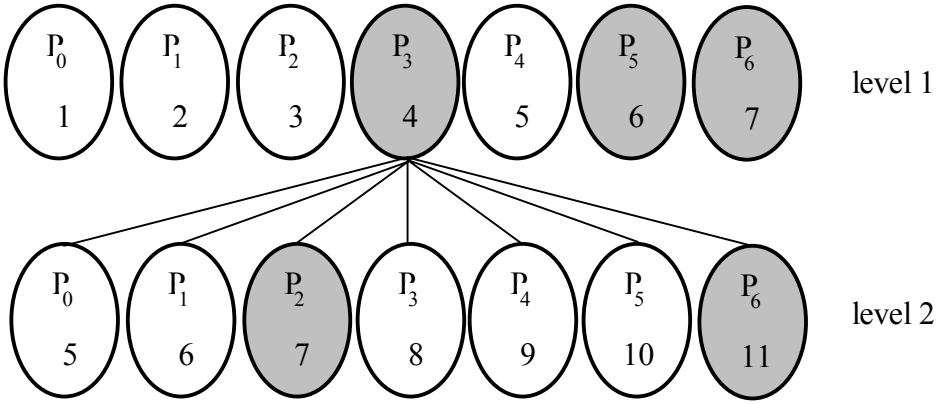

Fig. 1. Parallel N-ary speculative computation

Implementation of the same method with multiple threads instead of multiple processors is presented in [2]. Experiments on a UNIX based two-processor workstation showed that communication overhead between multiple threads is too high to obtain any speedup. Computation time for a single iteration is too short in comparison with time needed for synchronization.

```
procedure thread(i);
  repeat
    wait_start;
    compute (iter + i);
    O(i) = decision;
    signal_end;
  until forever
end.
```

```
procedure main_thread;
  iter = 1;
  for i = 1 to S do
    j = 0;
    while j < N²*DML do
      signal_start;
      compute (iter);
      O(1) = decision;
      wait_all_threads;
      k = min{l | O(l) = accept ∨ l > threads};
      if k ≤ threads then
        modify (k);
      j = j + k;
      iter = iter + k;
    decrease_temperature;
end.
```

Fig. 2. Parallel N-ary speculative computation code

Fig. 2 shows the algorithm for each thread. One thread must take care of synchronization and its code is slightly different. In every computation level threads are twice synchronized, at the beginning and at the ending. In the original implementation with message passing multiprocessors communication time is also significant, but much smaller then synchronization time between threads.

2. Proposed parallel multithreaded simulated annealing method

The need for extreme reduction of communication between threads is obvious. With a synchronous approach such a reduction cannot be achieved. Thus, the asynchronous approach is used instead. In the new proposed method each thread computes independently one random move, decides about its acceptance, and modifies the configuration if the move is acceptable.

Threads could simultaneously change the same part of the system in a different way, and this must be prevented. The simplest way to do that is to treat parts of the program where threads modify the system as critical sections and protect them by exclusive locking. Also, there is no need to make difference between threads because there is no synchronization. Every thread can modify temperature when the number of computed iterations at current temperature reaches the defined value. This part of program must be protected from simultaneous change. Fig. 3 shows the proposed algorithm for each thread. Dependence between threads is reduced to the part where change is performed and temperature decreased.

```
Procedure thread;
  Repeat
    Compute;
    Decision = make_decision;
    enter_critical_section;
    if decision = accept then
      modify;
    iter = iter + 1;
    if iter > N² * DML then
      iter = 0;
      step = step + 1;
      T = T * α;
    leave_critical_section;
  while step < S;
end.
```

Fig. 3. Proposed per-thread code

This method is applied to the *traveling salesman problem* (TSP), where the salesman has to travel to a number of cities and then return to the initial one. Each city must be visited once. This is an *NP*-complete problem, and use of approximation methods, such as simulated annealing, is the only way to get acceptable results in a reasonably short computation time. Cost function in this case is the tour distance. However, the cost of the new solution is not needed for the simulated annealing algorithm, a change in cost is enough to accept a move. In our implementation, a move exchanges two cities in the tour and the cost function change is simple to compute.

3. Experimental results

Experiments were made with 100-, 200-, 500- and 1000-city TSP at low loaded system (average load near zero) and heavy loaded system (average load above 4). All programs are written in the C++ programming language.

Table 1 shows TSP parameters used in the experiments. Parameters are chosen in such a fashion that the execution times are not too long, regarding a problem size. Result quality is, therefore, better for smaller problems, but the quality of the solution is not the primary objective. Instead, the goal was to search for some technique which will benefit from a multiprocessors on a different operating systems that support multithreading. Thus, results can

not be compared with results made on a non-generic, special message passing multiprocessors shown in papers [4] and [5].

Table 1. TSP parameters

Problem size (No of Cities)	100	200	500	1000
No of iteration in inner loop	3333	8000	25000	50000
No of cooling steps (outer loop)	50			
Initial acceptance probability	0.8			
Temperature decrease factor	0.9			

The execution times and used system times are shown in Table 2. Execution time is actual, real time elapsed while the program was executing. The used system time is a sum of user time and kernel time consumed on program execution. The real time is the main representative for quality of technique because it is a measure of a speedup. In a message passing multiprocessor systems, as those presented in [4] and [5], speedup is the only relevant attribute since the observed program is the only one running.

Table 2. Execution time (sec) of the TSP on the Ultrasparc 2

Low loaded system								
No of Threads	100-City		200-City		500-City		1000-City	
	t_R	t_S	t_R	t_S	t_R	t_S	t_R	t_S
1	3,67	3,68	8,94	8,89	27,67	27,54	52,42	55,38
2	2,87	5,36	6,69	12,28	20,59	38,24	40,08	75,29
4	2,94	5,79	6,67	12,92	20,14	39,15	40,03	78,21
Heavy loaded system								
No of Threads	100-City		200-City		500-City		1000-City	
	t_R	t_S	t_R	t_S	T_R	t_S	t_R	t_S
1	6,27	3,77	15,84	9,31	47,77	28,02	98,37	56,03
2	4,63	3,97	11,42	9,89	35,08	30,41	72,02	61,6
4	3,93	4,81	10,19	12,18	30,67	36,23	59,26	69,8

LEGEND: t_R - execution time t_S - consumed system time (both user and kernel time)

In a standard multiprocessor systems there are other processes, and influence of the program to the whole system is of our interest. A measure of such influence is the amount of used system time. This time is the sum of user time and kernel time spent on behalf of the program. The user time is time consumed on actual execution of the program. It can be greater than elapsed real time if there is more than one processor on the system. The kernel time is time spent on different system calls, page fault handling, context switching and similar actions. It does not include actual program instructions. In the current implementation, most of kernel time is used up on lock calls and context switching when locking fails.

Used multithreaded programs are composed only of worker threads. Consequently, they overload system many times more than a single-threaded programs and must be used with care. If a program consists of many worker threads, it can cause system to become too slow in response and unusable for other users. Therefore, the experiments were made not only at low, but also at high loaded system.

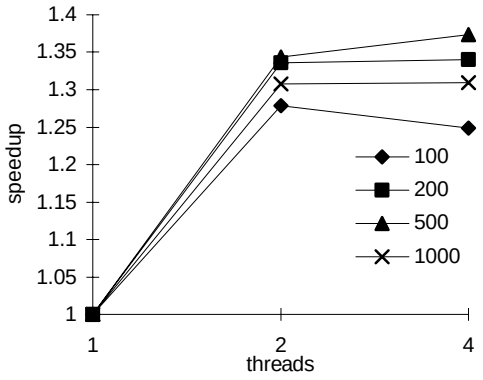

Fig. 4. Speedup at low loaded system

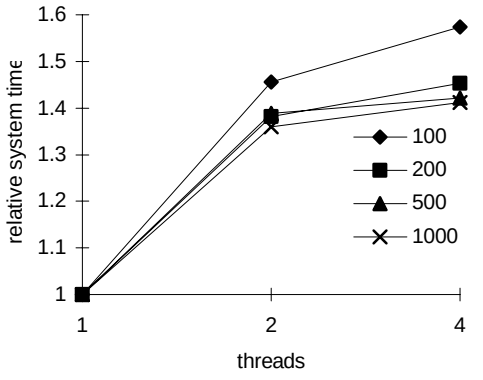

Fig. 5. System time consumed at low loaded system

Fig. 4 shows speedup achieved on a low loaded system. It is almost the same for two and four threads because there are only two processors. The speedup is slightly greater at larger problems, as a result of smaller possibility of two threads trying to work with the same object (one city) at the same time.

Fig. 5 presents a comparison of consumed system time between single-threaded and multithreaded executions. System usage is moderately greater than it should be if we watch speedup, and it is the result of time wasted on unsuccessful locking and switching between multiple threads.

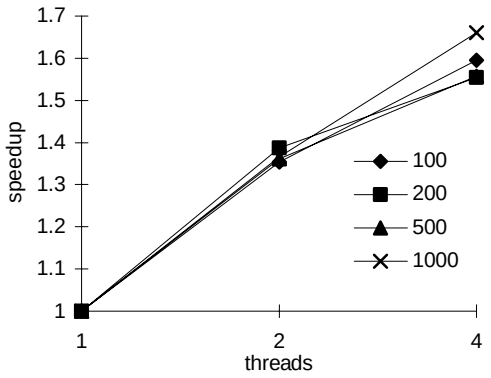

Fig. 6. Speedup at heavy loaded system

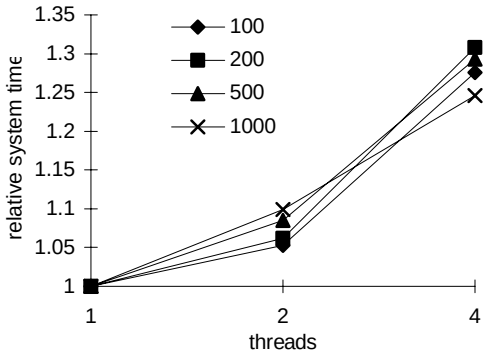

Fig. 7. System time consumed at heavy loaded system

Fig. 6 shows the speedup on a heavy loaded system. As a surprise, this speedup is significantly greater than the speedup at a low loaded system and grows linearly with the number of threads. Of course, it takes more time than on a low loaded system, but the speedup (execution time of single-threaded divided by execution time of multithreaded program) is greater here. The explanation is very simple. More threads get more system time, and even if they are not running concurrently on both processors, they get much more system time than single-threaded program.

From Fig. 7 it is obvious that speedup is not just the result of concurrent execution, because the speedup is greater than the relative system time (system time of multithreaded program divided by system time of single-threaded program).

4. Conclusion

With the proposed parallel simulated annealing technique we obtained 30 % speedup over sequential technique, which is a considerable improvement. However, for practical use the number of system processors must be greater than two. TSP is an extreme case of combinatorial optimization problems where computation time of one iteration is very short. Thus, the presented technique can be easily applied to many other combinatorial problems with presumably better performance than achieved on the TSP. Although speedup is obtained on a heavy loaded system, its use on such system is not recommended because it significantly increases load on already heavy charged system.

5. Acknowledgments

This work was done within the research project "*Problem-Solving Environments in Engineering*", supported by the Ministry of Science and Technology of the Republic of Croatia.

6. References

1. Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D. (1998) “Solving problems on concurrent processors: Volume I, General Techniques and Regular Problems”, California Institute of Technology, Prentice-Hall International, Englewood Cliffs, N. J.
2. Jelenković, L. (1996), *Collection of Functions for Multithreaded Programs*, Undergraduate dissertation, Faculty of El.Engineering and Computing, University of Zagreb. (in Croatian)
3. van Laarhoven, P.J.M., Aarts, E.H.L. (1987), *Simulated annealing: Theory and Applications*, D. Reidel Publishing Company, Dordrecht.
4. Nabhan, T., Zomaya, A. (1995), “A Parallel Simulated Annealing Algorithm with Low Communication Overhead”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 1226-1233.
5. Sohn, A. (1995), “Parallel N-ary Speculative Computation of Simulated Annealing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 997-1005.
6. SunSoft, (1994), *Solaris 2.4: Multithreaded Programming Guide*, Sun Microsystems, Mountain View, California.