Parallelization Strategies for Ant Colony Optimization

Thomas Stützle

FB Informatik, FG Intellektik, TU Darmstadt
Alexanderstr. 10, D-64283 Darmstadt, Germany
stuetzle@informatik.th-darmstadt.de

Abstract. Ant Colony Optimization (ACO) is a new population oriented search metaphor that has been successfully applied to $\mathcal{NP}$-hard combinatorial optimization problems. In this paper we discuss parallelization strategies for Ant Colony Optimization algorithms. We empirically test the most simple strategy, that of executing parallel independent runs of an algorithm. The empirical tests are performed applying $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System, one of the most efficient ACO algorithms, to the Traveling Salesman Problem and show that using parallel independent runs is very effective.

1 Introduction

Ant Colony Optimization (ACO) is a new population based search metaphor inspired by the foraging behavior of real ants. Among the basic ideas underlying ACO is to use an algorithmic counterpart to the *pheromone trail*, used by real ants, as a medium for communication among a colony of artificial ants. The seminal work on ACO is Ant System [9, 11] which was first proposed for solving the Traveling Salesman Problem (TSP). In Ant System, the ants are simple agents that are used to construct tours, guided by the pheromone trail and heuristic information based on intercity distances. Since the work on Ant System, several improvements of the basic algorithm have been proposed including Ant Colony System [10], $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [24] and the rank-based version of Ant System [4]. Additionally, the performance of ACO algorithms can be significantly enhanced by adding a local search phase in which solutions are improved by a local search procedure [10, 23, 16]. Thus, the most efficient ACO algorithms are actually hybrid algorithms consisting of a solution construction mechanism and a subsequent local search phase.

Ant Colony Optimization approaches are population based, i.e., a population of agents is used to find a desired goal. Population based approaches are naturally suited for parallel processing. Yet, for population oriented search procedures several possibilities of exploiting parallelism exist and their applicability depends strongly on the particular problem they are applied to and on the hardware available. In this article we discuss possibilities of parallel processing for the most efficient ACO algorithms on MIMD architectures. In particular, we motivate and investigate the execution of parallel independent runs of ACO algorithms. We show that high quality solutions can be achieved by such an approach presenting computational results for one of the currently most efficient ACO algorithm for the TSP, $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System.

The paper is organized as follows. To make the paper self-contained, we first introduce Ant Colony Optimization and the application of $\mathcal{MM}$AS to the TSP. In Section 3 we motivate the use of parallel independent runs and discuss other possibilities of parallel processing for ACO algorithms. The computational results are presented and discussed in Section 4. The paper ends discussing related work and outlining future work.

2 Ant Colony Optimization

2.1 Ant Colony Optimization applied to the TSP

The TSP is the problem of finding a shortest closed tour through a set of n cities traversing every city exactly once. A symmetric TSP can be represented by a complete weighted graph G with n nodes, the weights being the intercity distances $d_{ij} = d_{ji}$ between cities i and j. The TSP is a $\mathcal{NP}$-hard optimization problem and is used as a standard benchmark for many heuristic algorithms [13]. Like many other general purpose approaches, Ant System, which is the seminal work on Ant Colony Optimization, has been motivated presenting its application to the TSP.

$\mathcal{MAX}$–$\mathcal{MIN}$ Ant System is one of the enhancements of Ant System and shares important features like the tour construction mechanism with Ant System. It is a population based approach using m ants. To solve TSPs each edge of the graph has associated a pheromone level τ_{ij} that is updated by the ants during algorithm execution. The pheromone level is a desirability measure, the higher is the pheromone level of an edge the higher should be the probability an ant uses this specific edge. Additionally, the ants may also use heuristic information that in case of the TSP is chosen as $\eta_{ij} = 1/d_{ij}$, giving preference to short intercity connections. To construct a tour, each ant is initially set on a randomly chosen city. Then, in each step an ant selects from its current city i one of the cities it has not yet visited according to the following probability distribution:

$$p_{ij} = \begin{cases} \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{k \; not \; visited} \tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}} & if \text{ city } j \text{ is not yet visited} \\ 0 & otherwise \end{cases} \tag{1}$$

The probability distribution for the selection of the next city is biased by parameters α and β which determine the relative influence of the trail strength and the heuristic information. To keep track of the cities already visited, every ant maintains a list, that is used to store its partial tour. After all ants have constructed a complete tour and have calculated the tour length, the trails are updated. Similar to ACS, in $\mathcal{MM}$AS only one ant (corresponding to the ant with the iteration best tour or the ant with the best tour found during the run of the algorithm) is allowed to update the trails whereas in Ant System all ants lay down some pheromone. The trail intensities are updated according to:

$$\tau_{ij}^{new} = \rho \cdot \tau_{ij}^{old} + \Delta\tau_{ij} \tag{2}$$

where $\rho,$ with $0 < \rho < 1$, is the persistence of the trail, thus $1 - \rho$ models the trail evaporation. The lower ρ, the faster the information gathered in previous iterations is forgotten. The amount $\Delta\tau_{ij}$ is equal to $1/L_{\text{best}}$, if edge $[i, j]$ is used by the updating ant on its tour, otherwise zero. L_{best} is the tour length of the trail updating ant. Thus, frequently used edges receive a higher amount of pheromone and will be selected more often in future cycles of the algorithm. The two basic steps *tour construction* according to (1) and *trail update* according to (2) are then repeated for a given number of iterations or for some maximally allowed computation time. We refer to the complete cycle of tour construction and trail update as one iteration.

The main differences between $\mathcal{MM}$AS and Ant System are that in $\mathcal{MM}$AS only one ant is allowed to provide a feedback mechanism by updating the trails and that the trails are limited to an interval between some maximum and minimum possible values τ_{max} and τ_{min}. A minor difference is that in $\mathcal{MM}$AS the trails are initialized to their maximum value $\tau_{\max}$. Minimum and maximum trail limits are used to keep the exploration of new tours on a sufficiently high level. In particular, the trail limits are used to counteract premature stagnation of the search. Stagnation may occur if the differences between the trail intensities on the edges get so high that the same tours are constructed again and again. On the other side, the trail limits should not be too tight to allow exploitation of the search experience accumulated by the pheromone trails.

2.2 Adding Local Search

The nowadays most efficient ACO algorithms use local search algorithms to improve the solutions constructed by the ants, i.e., they are hybrid algorithms. The performance of hybrid algorithms often depends crucially on the kind of local search procedure that is used to improve solutions. For example, when applying genetic local search algorithms to the TSP, it has been shown that the best performance is obtained using the sophisticated Lin-Kernighan local search algorithm [28]. Yet, an efficient implementation of the Lin-Kernighan heuristic is rather involved, thus, for simplicity we use `3-opt` as a local search procedure [15]. Our implementation of `3-opt` is sped up using standard techniques as described in [2, 13]. In particular, we perform a fixed-radius nearest neighbor search and use *don't look bits* for the outer loop optimization, see [2] for details on these techniques.

In Table 1 we present results for the sequential version of $\mathcal{MM}$AS on some TSP instances taken from TSPLIB [20]. All ants are allowed to improve their solution after every iteration. As parameter setting we used $\alpha = 1, \beta = 2$, $\tau_{\max} = L_{\text{best}}/(1-\rho)$, $\tau_{\min} = \tau_{\max}/2n$. For instances with $n < 500$ we use 10 ants, for larger instances 25. The runs are performed on a Sun UltraSparc II Workstation with two UltraSparc I 167MHz processors with 0.5MB external cache. Due to the sequential implementation of the algorithm only one processor is used. The computational results are one of the best obtained so far with an ACO approach applied to the TSP. Compared to other algorithms, the genetic algorithm of [19] and those using the sophisticated Lin-Kernighan algorithm [17, 13, 18] perform better.

Table 1. Performance of the sequential implementation of $\mathcal{MM}$AS on several symmetric TSP instances. Averages over 10 runs. Given are the best solution found, the average solution quality and the worst solution found. The number in the instance descriptor is the number of cities in each instance, see text for more details. Additionally we give the average time to find the best solution, the maximally allowed computation time and the average number of iterations to find the best solution in a run.

Instance	Optimum	Best	Average	Worst	avg.time	max.time	avg.iterations
d198	15780	15780	15780.3	15781	43.4	300	236.1
lin318	42029	42029	42029	42029	132.7	450	494.2
pcb442	50778	50785	50886.5	50912	288.7	900	1164.5
att532	27686	27703	27707.4	27728	429.0	1800	561.2
rat783	8806	8806	8811.5	8821	935.2	2100	878.2
pcb1173	56892	56892	56960.3	57091	3728.7	4500	1837.4
d1291	50801	50801	50845.8	50909	2482.5	4500	1054.1

3 Parallelization Strategies

Ant Colony Optimization as a population-based search metaphor is as such inherently parallel. Yet, there is no golden rule for how to parallelize ACO algorithms. How to parallelize ACO algorithms efficiently, depends strongly on the available computing platform and the problem to which the algorithm is applied. Today a very common way to parallelization is given by MIMD architectures like, e.g., a cluster of workstations. In our discussion we concentrate on parallelization possibilities in such an environment.

3.1 Parallel runs of one algorithm

The most simple way to obtain a parallel version of an algorithm is the parallel independent execution on k processors of the sequential algorithm. Using parallel independent runs is appealing as basically no communication overhead is involved and nearly no additional implementation effort is necessary. Of course, using independent runs of an algorithm in parallel is only useful if the underlying algorithm is randomized, i.e., if the search process relies on random decisions. ACO algorithms and $\mathcal{MM}$AS, in particular, are such algorithms as, for example, the tour construction process is highly random. To describe the performance of a randomized heuristic algorithm, we may adopt one of the following two points of view. On the one hand, we can describe the solution quality obtained after executing an algorithm for time t by a random variable C_t with associated distribution function $F_t(c)$. Alternatively, if we fix a required solution quality, say, $q\%$ above a lower bound or a known optimal solution, we can view the algorithm dependent run-time needed to find such a solution as a random-variable T_q with distribution $G_q(t)$.[1] Knowledge of both distributions can give very helpful indication on the effectiveness of parallel independent runs.

In case of parallel independent runs the best solution of the k runs is taken as the final solution. In such a situation we are interested in the solution quality distribution of *best-of-k* runs. Given the distribution $F_t(c)$, the distribution function of C_t^k, the random variable corresponding to the final solution quality of best-of-k runs, can easily be calculated as $F_t^k(c) = 1-(1-F_t(c))^k$ in case of minimization problems. Alternatively, if we require the algorithm to reach a given bound on the solution quality, the runs are stopped as soon one process reaches such a bound. Again, the run-time distribution $G_q^k(t)$ for executing k runs in parallel can be calculated as $G_q^k(t) = 1 - (1 - G_q(t))^k$. Performing parallel independent runs is efficient if the run-time is exponentially distributed, as in this case optimal speed-up is obtained. The main effect of performing multiple independent runs is that the solution quality distributions and the run-time distribution will get more peaked and be shifted to the left.

To judge the effectiveness of using parallel independent runs, the sequential version and the parallel version should be given the same CPU-time. Thus, one should compare the solution quality distribution $F_t^k(c)$ running k times the algorithms for time t with the distribution $F_{t\cdot k}(c)$ of executing a single run of the algorithm for time $t \cdot k$. Results can more easily be compared, for example, using the expected solution quality $E[C_t^k]$ and $E[C_{kt}]$. In case $E[C_t^k] = E[C_{kt}]$, and the variances of C_t^k and C_{kt} are similar, we could talk of an "optimal" speed-up, as the time needed to reach a specific solution quality in time t with k runs is the same as the average solution quality obtained with 1 run of time

[1] Actually, we should speak of a two-dimensional random variable (T, Q). $G_q(t)$ and $F_t(c)$ are simply the marginal distributions of (T, Q).

$k \cdot t$.[2] In case of $E[C_t^k] < E[C_{kt}]$ the resulting "speed-up" could even be regarded as "super-optimal". Similar arguments apply to run-time distributions. For a discussion of how to calculate speed-up in this case we refer to [27].

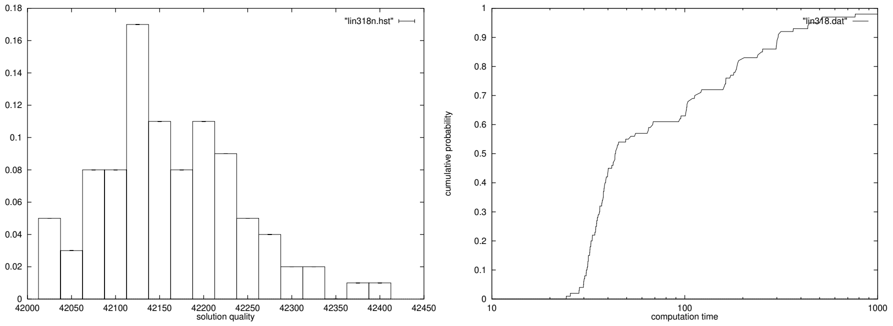

Fig. 1. On the left side histogram of the solution quality for $\mathcal{MM}$AS on `lin318` after $t = 20$sec. On the right side the cumulative run-time distribution is given for obtaining the optimal solution value of 42029. Based on 100 independent runs of $\mathcal{MM}$AS.

To give an idea on the shape of the distributions, we present in Figure 1 a histogram of the solution quality (left side) and the run-time distribution (right side) for the application of $\mathcal{MM}$AS to TSPLIB instance `lin318`. As can be observed, the histogram of the solution quality is multimodal. In this case, repeated execution of the algorithm for the given time bound will significantly increase the probability of finding high quality solutions. The run-time distribution on the right of Figure 1 indicates that, with a high probability, the optimal solution is found fast ($G(60) = 0.57$). Yet, for longer run-times the distribution is much less inclined. In such a situation parallel independent runs may be effective. For example, the probability that 5 parallel independent runs find the optimal solution within 60 sec is $1 - (1 - G(60))^5 = 0.985$. For the sequential algorithm, using the same overall run-time, we get $G(300) = 0.88$, lower than for running the algorithm five times for 60 secs.[3] Looking at the curve again, it is noteworthy that a certain time t_{init} is needed by $\mathcal{MM}$AS until an optimal solution can be found with a probability larger than zero. This effect will limit the obtainable speed-up by performing parallel independent runs. In case only very short computation times are allowed, parallelization has to be used to speed up the execution of a single run to get a reasonable solution quality, see next section for a short discussion of this issue.

Instead of running one algorithm with *one* particular parameter setting independently on k processors, it may also prove advantageous to run one algorithm with different search strategies or different parameter settings. Such an approach is specially appealing in case an algorithm's performance on different problem instances depends on the algorithm's parameter settings. In case of the application of $\mathcal{MM}$AS to the TSP, we did not notice significant dependencies of the algorithm's parameter settings on particular problem instances, one fixed setting appeared to perform generally well. Yet, such a strategy

[2] Usually, speed-up is defined w.r.t. an optimal sequential algorithm. Here we use speed-up to compare a parallel version of an algorithm with its sequential version.

[3] In this case the run-time distribution is below an exponential distribution, thus, even "super-optimal" speed-ups may be obtained for a low number of independent runs.

using different local search procedures should prove successful when applying $\mathcal{MM}$AS or other nature-inspired algorithms to the Quadratic Assignment Problem, as in this case the algorithms performance depends strongly on the instance type [22, 26].

Still, more improvement over using parallel independent runs may be gained by cooperation in form of solution exchanges among the search processes. Similar to parallel approaches for genetic algorithms like the island-model, communication among the ant colonies solving a particular problem could take place by exchanging of ants with very good solutions among the single ant colonies. These ants then may modify the pheromone trails of other ant colonies. Other possibilities would be to exchange or combine whole pheromone matrices among the ant colonies to influence the search direction of the others. A study of these possibilities is intended for future research, for the experimental investigation we concentrate on parallel independent runs.

3.2 Speeding up a single run

In case that only very low computation time is available, another possibility of parallel processing is to speed up a single run of an algorithm. A first implementation of Ant System on a transputer and a connection machine has been presented in [3]. In [5] parallelization strategies for Ant System have been discussed, presenting synchronous and asynchronous master-slave schemes for Ant System. Here, we extend the work of [5] by considering alternative schemes of speeding up runs of ACO algorithms like Ant Colony System [10] or $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [23] that rely strongly on local search applications.

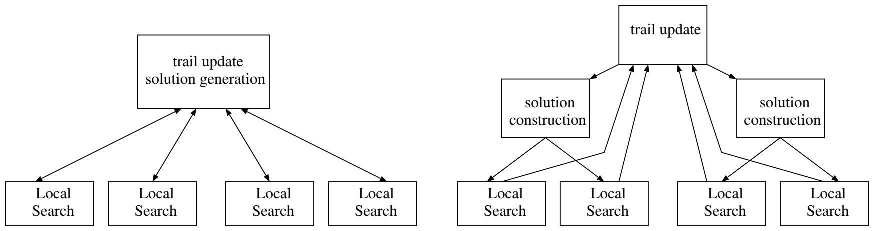

Fig. 2. Parallelization by Master–Slave approach, see text for details.

In case local search algorithms are used to improve solutions, one possibility of parallel processing consists in a master-slave approach. One master processor is used to update the main data structures for the ACO algorithm, constructs initial solutions for the local search algorithms, and sends the solutions to other processors which improve them by local search. The master collects these locally optimal solutions and in case a sufficient number of such solutions have arrived, it updates the trail matrix before constructing more solutions. This situation, also implementable in asynchronous mode, is depicted in Figure 2 on the left side. Such a parallelization scheme is particularly interesting if the update of the trails and the construction of solutions consumes much less time than the local search. This, for example, is the case when applying ACO algorithms to the Quadratic Assignment Problem [16, 22, 14]. In this case the construction of solutions and the update of the trail matrix is of complexity $O(n^2)$, whereas the local search is of complexity $O(n^3)$. Profiler data for the sequential algorithm $\mathcal{MM}$AS show that roughly 99% of the time is spent improving the solutions by local search. Yet, for the application of $\mathcal{MM}$AS or ACS to the TSP the situation is completely different. For the TSP the local search runs very fast using the implementation tricks described in [2]. In general, only roughly 70 - 80% of the time is spent by the local search, roughly 10 - 15% of the time is spent constructing tours and 5-10% of the time is needed for updating the trail matrix.[4] In this case, for a parallel implementation on several processors, a situation like that depicted in Figure 2 on the right side might be preferable. One processor keeps the main data structures for the trail matrix and updates it. One or several other processors can use the trail matrix to construct solutions and send those to the processors that local search to these solutions. The improved solutions are sent back to the main processor. Again, communication can be done in asynchronous mode. A major disadvantage of these approaches is that communication has to take place frequently. In general, the obtainable speed-up by such an architecture will be less than optimal due to the communication overhead.

4 Experimental Results

In this section we present computational results for the execution of parallel independent runs of $\mathcal{MM}$AS with up to 10 processes. It is difficult to calculate the exact speed-up for parallel independent runs, because the solutions quality has to be taken into account. To circumvent this problem we concentrate on the following experimental setting. We compare the average solution quality of $\mathcal{MM}$AS running k times for time limit t_{max}/k to that of a single run of time t_{max}. Like noted in Section 3.1, for very short run-times the solution quality for $\mathcal{MM}$AS will be rather poor, as the algorithm needs some initialization time t_{init} to have a reasonably high chance to find high quality solutions.[5] Also, t_{init} will increase with increasing dimension of the TSP instance. Therefore, the computation times were chosen roughly in such a way that $t_{\text{max}}/k \geq t_{\text{init}}$.

The experimental results are presented in Table 2 for several values of k. The two smallest instances are regularly solved to optimality, for `d198` only in few runs the optimal solution has not been found. The effect of using parallel independent runs for these two instances can be noted in the reduction of the average time needed until the optimal solution is found. For example, for `lin318` with 6 independent runs, the mean time to find the optimal solution is 22.0 sec as opposed to 132.7 for the sequential version. Thus, the obtained speed-up would be $t_1/t_6 = 6.03$, for 10 independent runs the speed-up would be 6.7. For larger problem instances, the best average results are always obtained with $k \geq 2$ runs of the algorithm. Due to the increased number of executions of the algorithm, the probability to get stuck at a rather bad solution is low, therefore the average solution quality increases.

Note, that this way to investigate the effect of parallelization is strongly related to an often posed question for Simulated Annealing algorithms: Is it preferable to execute one long Simulated Annealing run of time t_{max} or is it better to execute k short runs of time t_{max}/k [8]? The experimental results presented here suggest, as it was the case with Simulated Annealing, that an increased solution quality with $\mathcal{MM}$AS can be obtained by restarting the sequential algorithm after some given time bound. We are convinced

[4] This situation is worse, in case `2-opt` is used as local search procedure. Then roughly 50% of the time is spent for the trail update and the tour construction. Nevertheless, the situation would be better if the Lin-Kernighan heuristic is used for the local search, as it is usually more time consuming.

[5] This is not a particular problem of $\mathcal{MM}$AS, but any optimization algorithm will need some time t_{init} to find very high quality solutions for the TSP.

Table 2. Performance of parallel independent runs for $\mathcal{MM}$AS on symmetric TSP instances. Averages over 10 runs (except for d1291, 5 runs), for 1 (sequential case) to 10 parallel independent runs. Given are the average solution quality reached, the best average solution quality is indicated in bold face. The parameter settings for $\mathcal{MM}$AS are those presented in Section 2. $t_{\max}$ is the maximally allowed time for the sequential algorithm, computation times refer to a UltraSparc I processor (167MHz).

Instance	$t_{\max}$	1	2	4	6	8	10
d198	300	15780.3	15780	15780.1	15780	15780	15780
lin318	450	42029	42029	42029	42029	42029	42029
pcb442	900	50886.5	50873.0	50875.3	**50852.7**	50862.9	50860.6
att532	1800	27707.4	27702.5	27702.1	27703.5	27702.3	**27699.0**
rat783	2100	8811.5	8810.8	**8809.5**	8810.6	8810.8	8813.1
pcb1173	4500	56960.3	56960.9	56922.4	**56912.6**	56929.7	56969.1
d1291	4500	50845.8	**50809.0**	50821.8	50825.2	50826.6	50830.0

that this observation also holds for a variety of other regularly used search metaphors like Genetic Algorithms.

5 Related Work

Parallelization strategies are widely investigated in the area of Evolutionary Algorithms, see [6] for an overview. In [21] the performance of parallel independent runs of genetic algorithms has been investigated theoretically and on some test problem "superlinear" speed-up could be observed. Most parallelization strategies can be classified into *fine-grained* and *coarse-grained* approaches. Characteristic of fine-grained approaches is that very few, often only one, individuals are assigned to one processors and individuals are connected by a *population-structure*. An example of an implementation of such an approach for the TSP is ASPARAGOS [12] that was able to find very good solutions to the TSP instance att532. A typical example of a coarse grained approach is the *island*-genetic algorithm in which the population of a genetic algorithm is divided into several subpopulations. Communication takes place by exchanging individuals at certain time-points among the subpopulations.

Using independent runs is also studied for Simulated Annealing. The question to answer is whether it is profitable to execute one long run or in the same time several short runs [8]. For an introduction to concepts of parallel Simulated Annealing see [1]. Similarly, parallel, independent runs are also investigated for Tabu Search. In [25] it is argued that in case the run-time to find an optimal solution to QAPs is exponentially distributed, optimal speed-up using independent runs of the algorithm can be obtained. In fact, empirical evidence is given that the run-time distribution is very close to an exponential distribution. In [27] parallel independent runs of Tabu Search are investigated for the Job-Shop problem. A classification of parallel Tabu Search metaheuristics is presented in [7].

6 Conclusions

For many modern randomized algorithmic approaches to combinatorial optimization, parallelization strategies have been examined. We start the discussion of parallelization possibilities of the most efficient Ant Colony Optimization algorithms. The parallelization strategy to be used depends on the particular problem one has to solve and the available hardware. Of special appeal to randomized algorithms like Ant Colony Optimization approaches is the use of parallel independent runs. Our experimental investigation of this strategy for one particular ACO algorithm, $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System, on the Traveling Salesman Problem has shown that such a simple parallelization scheme can be highly efficient. More elaborate mechanisms are only justified if they give better performance than the execution of independent runs.

One line of future research is to investigate the effectiveness of parallel ACO algorithms on other problems like the QAP. Of even larger interest is the investigation of the benefit of using cooperation among the single runs of the algorithm. Here, previous investigations on fine-grained parallelization of genetic algorithms and the island-model seem to suggest that still some performance improvement can be obtained by cooperation among parallel processes.

References

1. E. Aarts and J. Korst. *Simulated Annealing and Boltzman Machines – A Stochastic Apporach to Combinatorial Optimization and Neural Computation*. John Wiley & Sons, 1989.
2. J.J. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.
3. M. Bolondi and M. Bondanza. Parallelizzazione di un Algoritmo per la Risoluzione del Problema del Commesso Viaggiatore. Master's thesis, Politecnico di Milano, 1993.
4. B. Bullnheimer, R.F. Hartl, and C. Strauss. A New Rank Based Version of the Ant System — A Computational Study. Technical report, University of Viena, 1997.
5. B. Bullnheimer, G. Kotsis, and C. Strauss. Parallelization Strategies for the Ant System. Technical Report POM 9/97, University of Vienna, 1997.
6. E. Cantú-Paz. A Survey of Parallel Genetic Algorithms. Technical Report IlliGAL 97003, University of Illinois at Urbana-Champaign, 1997.
7. T.G. Crainic, M. Toulouse, and M. Gendreau. Toward a Taxonomy of Parallel Tabu Search Heuristics. *INFORMS Journal on Computing*, 9(1):61–72, 1997.
8. N. Dodd. Slow Annealing Versus Multiple Fast Annealing Runs – An Empirical Investigation. *Parallel Computing*, 16:269–272, 1990.
9. M. Dorigo. *Optimization, Learning, and Natural Algorithms*. PhD thesis, Politecnico di Milano, 1992.
10. M. Dorigo and L.M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.
11. M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 26(1):29–41, 1996.
12. M. Gorges-Schleuter. Comparison of Local Mating Strategies in Massively Parallel Genetic Algorithms. In *PPSN-II*, pages 553–562, 1992.
13. D.S. Johnson and L.A. McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*. John-Wiley and Sons, Ltd., 1997.
14. E.D. Taillard L. Gambardella and M. Dorigo. Ant Colonies for the QAP. Technical Report IDSIA-4-97, IDSIA, 1997.

15. S. Lin. Computer Solutions of the Traveling Salesman Problem. *Bell Systems Technology Journal*, 44:2245–2269, 1965.
16. V. Maniezzo, M. Dorigo, and A. Colorni. The Ant System Applied to the Quadratic Assignment Problem. Technical Report IRIDIA/94-28, Université Libre de Bruxelles, Belgium, 1994.
17. Olivier C. Martin and Steve W. Otto. Combining Simulated Annealing with Local Search Heuristics. *Annals of Operations Research*, 63:57–75, 1996.
18. P. Merz and B. Freisleben. Genetic Local Search for the TSP: New Results. In *Proc. of the IEEE Conf. on Evol. Comp. (ICEC'97)*, pages 159–164, 1997.
19. Y. Nagata and S. Kobayashi. Edge Assembly Crossover: A High-power Genetic Algorithm for the Traveling Salesman Problem. In *Proc. of ICGA'97*, pages 450–457, 1997.
20. G. Reinelt. TSPLIB — A Traveling Salesman Problem Library. *ORSA Journal On Computing*, 3:376–384, 1991.
21. R. Shonkwiler. Parallel Genetic Algorithms. In *Proceedings of ICGA'93*, 1993.
22. T. Stützle. $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System for the Quadratic Assignment Problem. Technical Report AIDA–97–4, TH Darmstadt, July 1997.
23. T. Stützle and H. Hoos. The $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System and Local Search for the Traveling Salesman Problem. In *IEEE Conf. on Evol. Comp. (ICEC'97)*, pages 309–314, 1997.
24. T. Stützle and H. Hoos. Improvements on the Ant System: Introducing $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System. In *Proceedings of ICANNGA'97*. Springer Verlag, Wien, 1997.
25. É.D. Taillard. Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.
26. É.D. Taillard. Comparison of Iterative Searches for the Quadratic Assignment Problem. *Location Science*, pages 87–105, 1995.
27. H.M.M. ten Eikelder, B.J.M. Aarts, M.G.A. Verhoeven, and E.H.L. Aarts. Sequential and Parallel Local Search Algorithms for Job Shop Scheduling. Paper presented at the Metaheuristics conference 1997, 1997.
28. N.L.J. Ulder, E.H.L. Aarts, H.-J. Bandelt, P.J.M. van Laarhoven, and E.Pesch. Genetic Local Search Algorithms for the Traveling Salesman Problem. In *Proceedings PPSN-I*, number 496 in LNCS, pages 109–116. Springer Verlag, 1991.