Some strategies for parallelizing Ant Systems

Lorenzo Cioni

Doctoral Course *Parallel Computing in Combinatorial Optimization*

June 9 - June 30, 2005

Dipartimento di Informatica, Università di Pisa

e-mail: lcioni@di.unipi.it

Abstract

The present paper is a short partial survey of some strategies that can be used to exploit the potential parallelism of the *Ant System metaheuristic*. The paper is composed of some introductory sections (that make it as self-contained as possible) followed by a description of *Ant System* applied to the *Traveling Salesperson Problem* in both the sequential version and some parallel versions.

1 Introduction

The present paper contains a short and partial survey of some of the parallelization strategies of the Ant Systems (AS) meta-heuristic with regard to the typical application of AS i.e. the Traveling Salesperson Problem (TSP). AS have been introduced as Ant Colony Optimization by Dorigo ([8]) and, since then, have undergone many changes (we cite here *Rank based Ant System Max-Min Ant System* and *Ant Colony system*,[6]) and have been successfully applied in solving different optimization problems ([11], [6], [7], [3], [12]) such as: traveling salesperson, quadratic assignment, vehicle routing, job-shop scheduling, sequential ordering, shortest common supersequence, graph coloring and frequency assignment, bin packing, constraint satisfaction, set covering and partitioning, spatial databases and telecommunication routing among the others.
So to make the paper as self-contained as possible, in the next sections we present some general concepts on metaheuristics and parallel processing then we introduce AS as a paradigm or a metaheuristic. After that we describe the TSP from the AS perspective then we describe AS in its sequential form to end with some parallel versions. As we will see shortly, AS are a population based metaheuristics and therefore ([10]) are naturally suited for parallel processing though many possibilities of exploiting parallelism exist. Such possibilities depend on the the nature of the problem at hand and on the type of available hardware. This is the main reason we focus, in this paper, on the application of AS to TSP: this allows us to put on a common ground different proposals with the aim of comparing their results.

2 Some general concepts

In this section we follow [1] and [9] to arrange AS in a general framework. The starting point is the concept of *metaheuristic*. Metaheuristics (MHs) are a set of strategies that guide the efficient exploration of the search space so to find optimal solutions (if any). MHs range from *simple local search procedures* to *complex learning processes* and make use of a well balanced mixture of *diversification* (i.e. move to unexplored areas of the search space) and *intensification* (i.e. intensively explore areas of the search space) techniques. MHs ([1]) are usually approximate, non-deterministic, non problem-specific but make use of domain specific knowledge in the form of heuristics. MHs can be classified according the following criteria:

1. **nature-inspired** vs. **non nature-inspired** depending on the origin of an algorithm: in the first family we have, among the others, *Genetic*

Algorithms and *Ant Algorithms* whereas in the second one we have algorithms such as *Tabu Search* and *Iterated Local Search*;

2. **population-based** vs. **single point search** or **trajectory methods** depending on the number of solutions used at the same time (a population or a single solution): *Genetic Algorithms* and *Ant Algorithms* are of the first type whereas *Tabu Search*, *Iterated Local Search* and *Variable Neighborhood Search* are examples of the second type;

3. **dynamic objective function** vs. **static objective function** depending on the nature of the objective function if it is kept as given or modified during the search;

4. **one neighborhood structure** vs. **various neighborhood structures** depending on the topology of the search space,

5. **memory usage** vs. **memory-less** methods depending on the the use or not of a short term and/or long term memory, for instance to implement a search history;

6. **hybrid** vs. **pure** methods depending on the fact that a method is hybridized in some way or not. Hybridization can take three forms ([1]):

   (a) the use of components of one metaheuristic into another one;

   (b) the use of various algorithms exchanging information in some way (cooperative search);

   (c) integration of approximate and systematic methods.

   As we will see shortly, some parallel implementations of AS make use of *local search* to improve performances and of a *tabu list* (as in *tabu search*) to keep track of the already visited cities.

Since in this paper we aim to present some strategies for parallelizing AS and compare their results we now introduce some concepts regarding *parallel computing* and *parallel systems*, including parameters to make parallel performance measures ([9]).
With *parallel computing* or *parallel processing* we usually mean data processing on *parallel systems.* Parallel systems can be characterized in terms of *control* (according to Flynn's classification as $SIMD$ or $MIMD$ systems), *synchronization* (as either *synchronous* or *asynchronous* or *partially asynchronous* [2] systems), *communication* (as either *shared-memory* or *message-passing* systems), *granularity* (as *fine-grained* vs. *coarse-grained* systems)

and, lastly, in terms of the *number of processors* as *small-scale systems*, if such a number is low, or *large-scale or massively parallel systems* if such a number is high. Within the scope of this paper we only point out that:

1. only with $MIMD$ systems (either in the form of networks of workstations and PCs or as grid computing or as some special architecture such as transputer) we have the concurrent execution of control flows on data flows,

2. in case of *shared-memory systems* we have conflicts due to concurrent accesses to memory locations that reduce performances;

3. in case of *message-passing systems* the exchange of messages containing data introduces a communication overhead that reduces performances.

As to the parameters we are going to introduce (and use in the next sections) a set of "classical" parameters that allows the execution of parallel performance measures. If we suppose to have p processors that can cooperate in some way in a parallel system and define as $T(p)$ the elapsed time with p processors and with $T(1)$ the time required by the "best" sequential algorithm we can define the following parameters:

1. *speedup* $S(P) = \frac{T(1)}{T(p)}$

2. *efficiency* $E(P) = \frac{S(p)}{p}$

3. *overhead* $O(P) = T(p)(1 - E(p)) = T(p) - T(1)/p$

4. *scalability* that is ([9]) how should the size of a problem vary in order to maintain constant efficiency irrespective of the number of processors used.

Usually we have $1 \leq S(p) \leq p$. In [2] the authors introduce also the following parameters:

1. *efficiency* $\eta = S(p)E(p)$

2. the ratio of *computation*, *communication* and *idle* times in relation to the *total simulated execution time*.

The first of such parameters allows the identification of the maximum number of processors that can be used to carry out useful work whereas the ratios can be used within simulation models. In [2], indeed, the authors claim that in order to evaluate the behaviour of parallel programs three tools can be used:

1. *analytical techniques* or methods;
2. *simulation models*;
3. *measurement experiments* on a real implementation.

Since analytical methods prove insufficient ([2]) "due to the complexity of estimating the communication overhead" and since "the characteristics of the particular parallel machine will bias the performance of a real implementation" they decided to use *simulation models* discarding measurement experiments too whereas, as will be examined in next sections, in [10] and in [4], for instance, experimental results are obtained by using typical data sets for the TSP.

3 AS, general view

From the definitions of section 2 we can deduce that AS are a metaheuristic

- nature-inspired,
- population-based

since the inspiring model is that of real ants (a population) searching for food: artificial ants searching through the solution space mimic real ants looking for the shortest path from the nest to food sources. The basic elements of the AS MH are the *artificial ants* as *cooperating agents* that use a set of rules to generate, update and use both local and global information so to find *good solutions* within the solution space.
Going into details we can say ([1]) that in AS artificial ants walk randomly on the connections $\mathscr{L}$ of a connected graph (*construction graph*) whose nodes are the solution components $\mathscr{C}$. If a Combinatorial Optimization (CO) problem is given its constraints are included in the ants' constructive procedure so that, at each step, only feasible solution components can be added to the current partial solution ([1]). Elements of $\mathscr{C}$ and $\mathscr{L}$ have associated parameters (the so called *pheromone trail parameters* and *heuristic or visibility values* that are used by the artificial ants to make probabilistic decisions on how to move from one vertex to the next one.
As to *pheromone trail parameters* we note that ([1]):

1. they mimic the behaviour of real ants that mark the paths they travel on with a substance called *pheromone*,

2. in general they can be assigned to both nodes and connections (only to nodes in case of TSP)

whereas with regard to *heuristic or visibility values* we only say that they too can be assigned to both nodes and connections (only to nodes in case of TSP) and represent an *a priori* heuristic information about the problem instance (in case of TSP we have the length of an arc between two cities). Given a set $\mathscr{A}$ of artificial ants, the general structure of AS (cf. sections 4 and 5) is composed of:

1. an initialization phase;

2. a loop to be repeated while termination conditions are not met.

In the *initialization phase* all the *pheromone trail parameters* are initialized to a small value whereas within the loop every artificial ant incrementally constructs a solution by adding solution components to the partial solution constructed so far ([1]). The choice of the next solution component is made on a probabilistic rule. When all the artificial ants are done a rule for the update of the pheromone trails is applied. The probabilistic rule is (cf. [2]):

$$p_{ij} = \begin{cases} \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{h\in\Omega}(\tau_{ih})^{\alpha}(\eta_{ih})^{\beta}} \;\; if\; j \;\in \Omega \\ 0 \qquad otherwise \end{cases} \tag{1}$$

where

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{2}$$

is the visibility of node v_j (or city, in case of TSP) from node v_i with d_{ij} length or cost of arc (i, j). Moreover we have:

1. τ_{ij} is the intensity of the pheromone trail between the nodes v_i and v_j,

2. α is a parameter that regulates the influence of τ_{ij},

3. β is a parameter that regulates the influence of η_{ij},

4. Ω is the set of not yet visited nodes.

The rule for the update of the pheromone trails is (cf. [2]):

$$\tau_{ij}(t+1) = \rho\tau ij(t) + \Delta\tau_{ij} \tag{3}$$

with

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{4}$$

and

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{L_k} & if\ ant\ k\ travels\ on\ \ edge\ (v_i, v_j) \\ 0 & otherwise \end{cases} \tag{5}$$

Moreover we have:

1. t is the iteration counter,
2. $\rho \in [0, 1]$ is a parameter that ragulates the evaporation of τ_{ij},
3. $\Delta\tau_{ij}$ is the total change of the trail level on the edge (v_i, v_j),
4. m is the number of the artificial ants,
5. $\Delta\tau_{ij}^{k}$ is the change of the trail level on the edge (v_i, v_j) caused by the kth artificial ant,
6. L_k is the cost of the solution found by the kth artificial ant (in case of TSP we speak of the length of a tour found by the kth artificial ant).

4 The TSP from the AS perspective

As it is known, CO problems ([9])) can be described as

$$min\{f(x) \ | \ x \in S\} \tag{6}$$

where f is the objective function and S is the finite, but very large-scale, set of the feasible solutions. Such problems are characterized by a computational complexity. Whenever such a complexity falls in the $NP-$hard class the only exact known algorithms are based on enumeration. Since, in the worst case, their complexity may be exponential with the size of the input data the only alternative is the use of $[meta-]$heuristic algorithms such as AS. TSP ([3]) is a well known CO problem that has $n!$ solutions (if n is the number of cities) so it's $NP - hard$ with exponential complexity in the worst case ([3]): this prevents the use optimal algorithms (that prove computationally inefficient if not totally unfeasible) and fosters the use of heuristic algorithms that in practice give good solutions. TSP aims at finding a constrained shortest path: given a complete weighted graph with n nodes, $G = (V, E, d)$, with

1. nodes (cities) $V = \{v_i \ : \ i = 1, \ldots, n\}$
2. arcs (roads) $E = \{(v_i, v_j)\ i, j = 1, \ldots, n\ i \neq j\ \}$
3. weights on the arcs (distances or costs between two cities) $d_{ij},\ i, j = 1, \ldots, n$

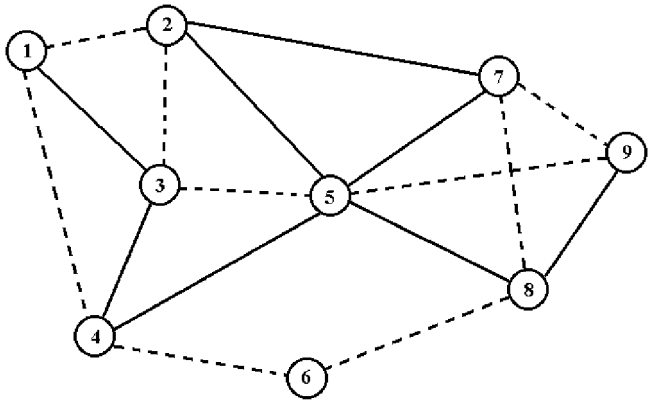

Figure 1: *Hamiltonian tour for TSP (from [3] with changes)*

the TSP requires the definition of a *minimum cost or distance Hamiltonian tour*. Figure 1 shows an example of such an Hamiltonian tour for simplicity on a non complete graph. Such a tour is formed by the following succession of the nodes: $1-2-3-5-9-7-8-6-4-1$.
Given a TSP for a graph with n cities, in order to solve it with AS we can use m artificial ants distributing them on the n cities according to some rule ([2]) (*diversification*) then we start with the algorithm (cf. section 5). The algorithm is made up of a main loop that is executed a certain number of times. At the start of each iteration all cities but the assigned ones can be visited and each ant decides independently from the others which not yet visited city to visit next. Such a decision is made according to the probabilistic rule of equation (1): the probability of selecting city j from i (p_{ij}) varies directly with the pheromone trail between i and j (τ_{ij}) and inversely with the distance d_{ij}(or directly with the visibility η_{ij}). For the first factor we speak of an *adaptive memory* with an associated parameter α whereas for the second we speak of a measure of desirability with an associated parameter β. The city selection process is repeated until all artificial ants have completed a tour. At each step of an iteration each artificial ant visits a new city (and this is always possible since the graph is connected) until it visits the last unvisited one. At this point the tour is closed ad each artificial ant evaluates the length L_k of its tour. A best tour (the one with the shortest length) is found so to update the one found at the previous iteration. Now it is time to update the trail levels of the pheromone and this is done according to equation (3). As section concluding remarks we note that ([2]):

1. the shorter a tour is the more pheromone is left on the arcs per unit length,

2. to avoid an early convergence of the algorithm we have (in analogy with nature) a pheromone evaporation regulated by the parameter ρ.

5 AS for TSP: the sequential version

Now that we have the bricks we can build the wall and so we present a high level sequential version of AS for TSP ([2]):

```
Initialize
For t=1 to T
  Assign the m ants to the n nodes
  For k=1 to m do
    Repeat until k has completed a tour
      Select city v_j to be visited next with probability p_ij given by (1)
    Calculate the length L_k of the tour generated by artificial ant k
  Update the trail levels τ_ij on all edges according to (3)
  Update the length of the best tour
End
```

Sequential pseudo-code for TSP with AS (we call it TSP-oriented-PC*, from [2] with changes)*

In the *Initialize* phase we have:

1. the calculation of the matrix of the distances $D = [d_{ij}]$, $i, j = 1, \ldots, n$ and, therefore, of the values of visibility (η_{ij});

2. the initialization of the matrix of the initial values of pheromone on each arc $\tau_0 = [\tau_{ij}]$, $i, j = 1, \ldots, n$.

After that the algorithm enters the main loop which is repeated T times (T may be either a predefined value or a value depending on some convergence criterion): within this main loop [artificial] ants work independently one from the others to define a tour among all the cities. When all the ants are done we have an *Update* phase during which the algorithm updates the values τ_{ij}. After T iterations the algorithm returns the best generated solution. Following [2] we have:

1. if we have n cities, m ants and T iterations we have an algorithm whose computational complexity is of order $\mathcal{O}(Tmn^2)$;

2. in [2] the authors claim that if we put $m = n$ and assign one ant to each city we get good results with respect to the quality of the best solution and the rate of convergence;

3. therefore they adopt such a position (and call it *problem size*) and put $m = n$ so that, since T is independent from the problem size, the computational complexity is of order $\mathcal{O}(m^3)$.

We are going to use such results in section 6. With regard to *TSP-oriented-PC* we note that a more general structure (so we call it *Abstract-PC*) is the following ([1] and [3]):

```
Initialize
While (termination_conditions_not_met) do
  ScheduleActivities
    AntBasedSolutionConstruction()
    PherormoneUpdate()
    DaemonActions()    %Optional
  end ScheduleActivities
endwhile
```

We are going to deal with *Abstract-PC* mainly in section 7, for the moment we note that the pseudo-code contains an outer loop which encloses an abstract construct (*ScheduleActivities*) that gathers three parts without specifying how these are scheduled and synchronized, all this being a duty of the algorithm designer. The three parts include ([1]):

1. *AntBasedSolutionConstruction*(): during this phase ants build a solution by moving through the nodes of the construction graph using rules such those specified by equations (1) and (3) and keeping in memory the path each is following and its length;

2. *PherormoneUpdate*(): this is the more complicate phase and that differentiating the various versions of *AS* (cf. section 1) since we can have:

   (a) the update of the pheromone when an ant walks on arc (i, j) or the so called *online step-by-step pheromone update*,

   (b) the update of the pheromone in a delayed and backward way and only when a solution of the proper quality has been built, the so called *online delayed pheromone update*,

(c) the update of the pheromone through evaporation by which the pheromone trail intensity on the components (arcs and nodes in general) decreases with time: this allows the forgetting, favours the diversification and prevents a too rapid convergence of the algorithm on sub-optimal solutions;

3. *DaemonActions*(): such an optional phase can be used to implement centralized actions that cannot be executed by the single ants, such as ([1]):

   (a) the use of local search procedures on the solution built by the ants or

   (b) the collection of global information to decide whether deposit or not additional pheromone so to bias the search process from a nonlocal perspective.

From the previous discussion it is easy to see ([3]) how each ant is an *autonomous agent* that (in TSP) constructs a *tour* and so proposes a solution to the TSP. There is, therefore, a *natural form of parallelism* at ants level since ants (that behave independently form each other) do not need synchronization (even if, as we will see in section 6, synchronization is useful to avoid sub-optimal solutions) and more ants can be concurrently active at the same time.

6 AS for TSP: the first two parallel versions

As stated at the end of section 5 the sequential version of AS applied to TSP ([2]) contains a high level of problem-inherent parallelism since the behaviour of each ant is totally independent from that of the other ants during every iteration. From this consideration in ([3]) two distinct parallel implementations strategies are proposed: a *synchronous strategy* and a *partially asynchronous strategy* (see figure 2). In figure 2 (left) a $fork - join$ completely synchronous architecture is shown. The idea is to compute the tours of the cities in parallel and to do that we have:

1. a master process M_p,

2. a set of processes P_i, $i = 1, \ldots, m$, one for each ant.

At the very start M_p spawn processes P_i and distributes them (with a message passing protocol) some global information such as the matrix of the distances D and that of the pheromone trail intensities τ_0. At this point

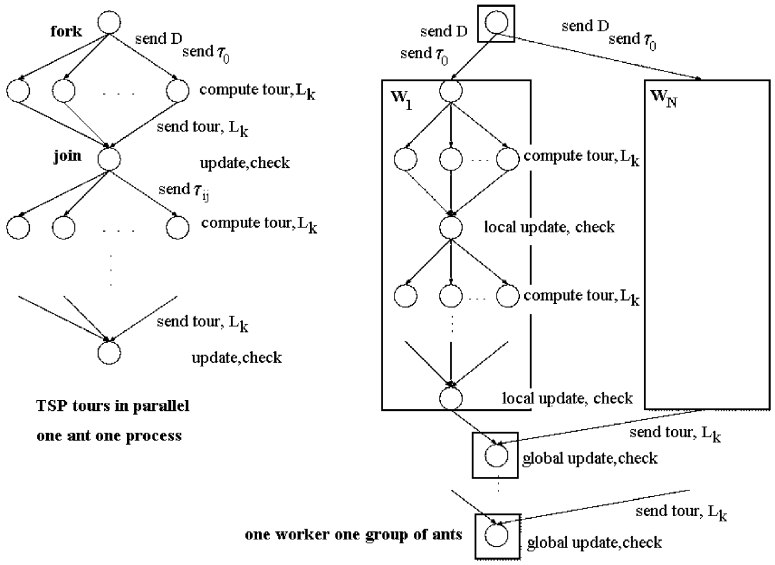

Figure 2: *Synchronous vs. Partially Asynchronous strategies (from [2])*

each ant/worker can compute a tour of the cities and, upon completion, can send the tour and its length back to M_p that, in its turn, update the trail levels (according to equation 3 so defining the new matrix τ_{ij}) and checks for the best tour found so far (that with the shortest length). To start a new iteration M_p sends to the P_i the new matrix τ_{ij} so that the processes can repeat their calculations with the new local data and this till the termination condition is met. In this case we have a synchronous structure with M_p that $fork$s processes that $join$ with M_p to exchange data and so on (see figure 2 (left)). Under the following assumptions:

1. communication overhead may be ignored,
2. we have as many processing elements (PEs) or workers as we need (and so an infinite number of PE) so that we can assign each process to a PE,

we get the following asymptotic speedup (under the assumption that the sequential version of AS is executed with as many ants as cities and so $n = m$):

$$S_{asymptotic}(m) = \frac{T_seq(m)}{T_par(m, \infty)} = \frac{\mathcal{O}(m^3)}{\mathcal{O}(m^2)} = \mathcal{O}(m) \tag{7}$$

Since communication overhead cannot be ignored and since the number of PE is N is smaller than the problem size in practice we get a lower speedup. Under the assumptions that:

1. we have as many ants as cities so that $m = n$,
2. we have N PE and $N \ll m$

we must assign, in a balanced way, a *colony* of ants to each PE so to increase the granularity of the application under the constraint that each worker gets about the same amount of processes and so the same computational load. Under these more realistic assumptions we get the following speedup:

$$S(m, N) = \frac{\mathcal{O}(m^3)}{\mathcal{O}(\frac{m^3}{N}) + T_{ovh}(m, N))} \tag{8}$$

The degradation of speedup is due to the rather high frequency and volume of communication in the synchronous approach: at each iteration M_p must exchange data with the processes P_i and this data exchange gives rise to the communication overhead $T_{ovh}(m, N)$ that strongly depends on the communication behaviour of the underlying physical architecture ([2]).
Since it is not possible,in general,to avoid the assignment of more than one process/ant to each PE (since usually $N \ll m$) the only way to get a better speedup is to reduce the impact of communication overhead. The solution proposed in [2] to this end is that shown in figure 2(right) and is called *partially asynchronous strategy*. According to such a strategy we have:

1. one PE that runs M_p (and that can coincide with one of the workers),
2. N workers W_j, $j = 1, \ldots, N$.

At the very start M_p sends (with a message passing protocol) to each worker W_j the matrices D and τ_0. Upon receiving such data the processes assigned to each worker W_j (on average we have $\lfloor m/N \rfloor$ processes for each worker) perform independently a certain number of local iterations of the sequential algorithms. When the local computations are over we have a *global synchronization* of all the workers with the M_p : the best locally found tour is sent to the M_p that performs the usual updating operations and sends to all the workers the updated matrix τ_{ij}.
The critical factor is the ratio between the number of local and global iteration: if such ratio is low there is no reduction of the communication overhead, if such ratio is too high good and promising values found locally might be ignored by the other workers. In the experiments reported in ([2]) such local/global ratio is fixed at five to one.
Having described the two strategies presented in [2] we now use the parameters we introduced in section 2 to illustrate the quantitative results presented in [2]. Such results have been derived using a discreet-events simulator. The simulator works on two basic assumptions:

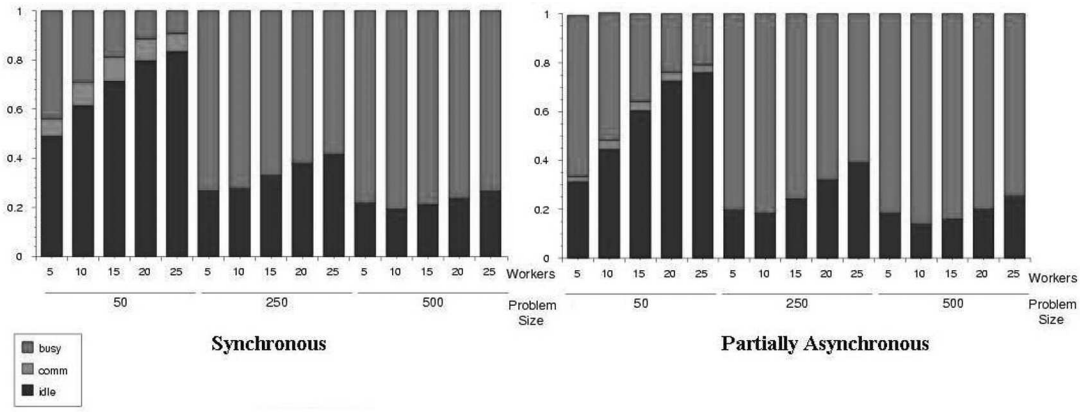

Figure 3: *Synchronous vs. Partially Asynchronous timings (from [2])*

1. the time needed to send a message is the sum of a fixed startup time and a variable time depending on the size of the message,

2. we can have multiple simultaneous communications without contention.

Moreover the simulator gets as input the description of the parallel program structure and the resource requirement specification and produces as output a trace file that contains the time stamps for the start and end of both computation and communication blocks. In [2] the experiments to compare the two strategies have been carried out on three different problem sizes:

1. **small**: $m = 50$,

2. **medium**: $m = 250$,

3. **large**: $m = 500$

with $N \in \{5, 10, 15, 20, 25\}$ so that in the *partially asynchronous strategy* there are up to 100 processes for each worker. Figure 3 shows, on the left, the busy/communication/idle times in the *synchronous* case and, on the right, the same times for the *partially asynchronous* case. According to [2] we have:

1. **busy time** is the time devoted to effective local computations,

2. **comm[unication] time** is the time spent in preparing and sending data among PE,

3. **idle time** is the time PEs do no useful work.

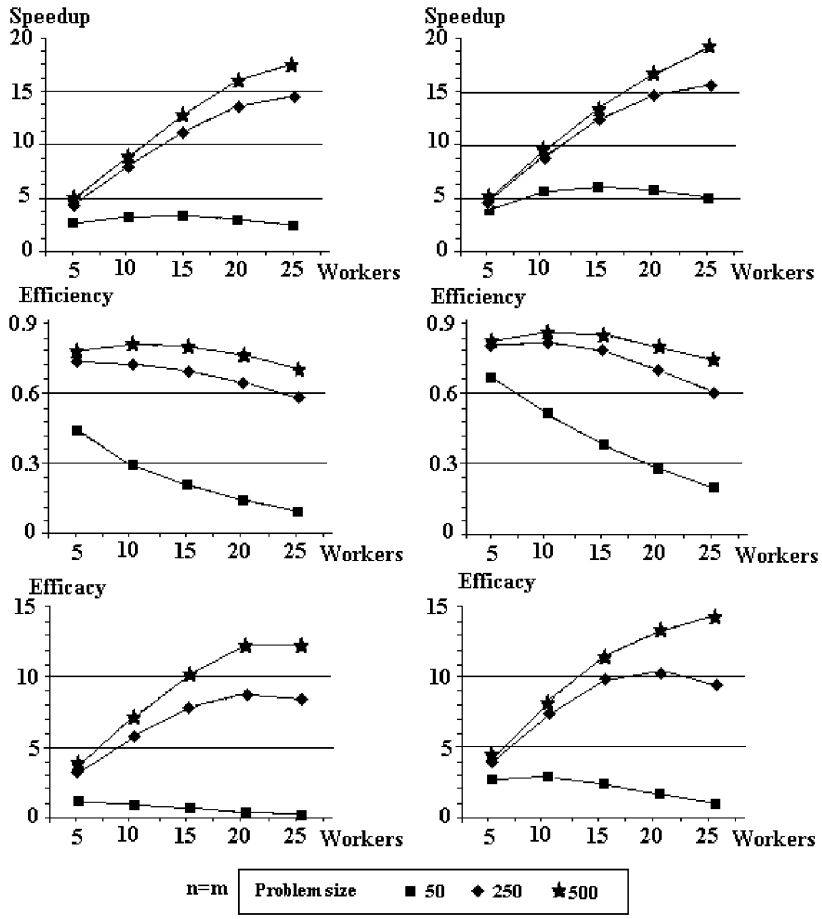

Figure 4: *Synchronous vs. Partially Asynchronous parameters (from [2])*

Data are presented in such a figure for the three problem sizes (that do not belong to any standard library of TSP such TSPLIB, see section 7) in a rather qualitative percentage form. Authors claim that:

1. for the small problem size, idle time dominates the scene and is lower in *partially asynchronous* case than in the *synchronous* case;

2. with increasing problem sizes (medium and large) communication time becomes negligible in both cases and idle time seems lower in *partially asynchronous* case than in the other (even if it is not so clear how much lower it is);

3. in both cases idle time increase with an increase in the number of workers.

Figure 4 presents, on the left, the values of the comparative parameters (i.e. Speedup, Efficiency and Efficacy) in the *synchronous* case and, on he right, the values of the same parameters for the *partially asynchronous* case. From the figure, again in a rather qualitative way, we can see that:

1. for all the problem sizes the second approach is better than the first (higher Speedup, Efficiency and Efficacy) owing to a reduced communication frequency, this factor being very important when implementing the algorithm on real parallel architectures;

2. for the small problem size we can say that increasing the number of workers above 5 (in the first case) or 10 (in the second) is a waste of computational resources;

3. in both cases/approaches the best values of the parameters are obtained for the large problem size when Speedup is nearly optimum ($S(N) \approx N$) and Efficiency decreases slowly with an increasing number of workers;

4. the Efficacy curves show that optimum number of workers that can be used with the different problem sizes is highest for the large problem size in the *partially asynchronous* case.

In the final section of [2], authors point out some corrections that could be adopted to get better performance measures. Such corrections involve the ratio local/global computations in the partially asynchronous case and the grouping of the processes to be assigned to the available PE in both cases.
As to the first point they propose a dynamic approach so to start with a low ratio, to avoid early convergence, and then switch to higher ratios when some promising solutions begin to emerge. As to the second point they highlight two aspects: assignment and dynamics.
With respect to assignment they note that processes can be assigned to the PE randomly or according to a distance criterion such that ants/processes on clustered cities are assigned to the same PE or, at the opposite, ants/processes on distant cities are assigned to the same PE. Lastly, with respect to dynamics they say that the assignment of processes to PE may be done only once at the very start of the computation or may be repeated during the computation and after a certain number of global or local synchronizations.

7 AS for TSP: further parallel versions

The next parallel implementation of AS we are going to describe in this paper is that presented in [10]. In that paper the author presents parallelization strategies for AS and empirically tests the most simple strategy, that of executing parallel independent runs of the algorithm. To do so the author uses $MAX - MIN$ AS applied to TSP noting that, since the most efficient AS algorithms are actually hybrid algorithms, he will present an algorithm enriched with a local search phase.

In order to better understand what follows we note that $MAX - MIN$ AS

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

Figure 5: *Performances of sequential implementations of $MAX - MIN$ AS (from [10])*

only the ant with the iteration best tour is allowed to update the pheromone trails. Such an update is performed in the usual way (see equation 3) and the steps of the algorithm (which are repeated for a given number of iterations or for some maximum computation time) are *tour construction* and *trail update*. Again in$MAX - MIN$ AS trails assume values within the interval τ_{min} and τ_{max} and are initialized to τ_{max}, all this to avoid a premature convergence of the algorithm.

Figure 5 presents the results of $MAX - MIN$ AS ($MMAS$) on some TSP instances taken from the standard library TSPLIB averaged on 10 runs. The numbers in the instances names are the number of cities among which the algorithm has to find a tour. To get lower execution times the author proposes a parallelization strategy of MMAS using a MIMD architecture such as a cluster of workstations. The most simple way to obtain the parallel version of an algorithm is to run it in parallel on k processors. This solution (which is effortless) avoids any communication overhead but can be adopted only if the underlaying algorithm is randomized as in the case of MMAS

where the tour construction is highly random. If we adopt k parallel independent runs we keep as the final solution the best solution of the k runs. In [10]) the author gives a justification in probabilistic terms to which we refer the reader. to compare the sequential version with the parallel version the two must have the same computation time so that if we have a time t for the k runs the sequential version must be given a time equal to kt. One

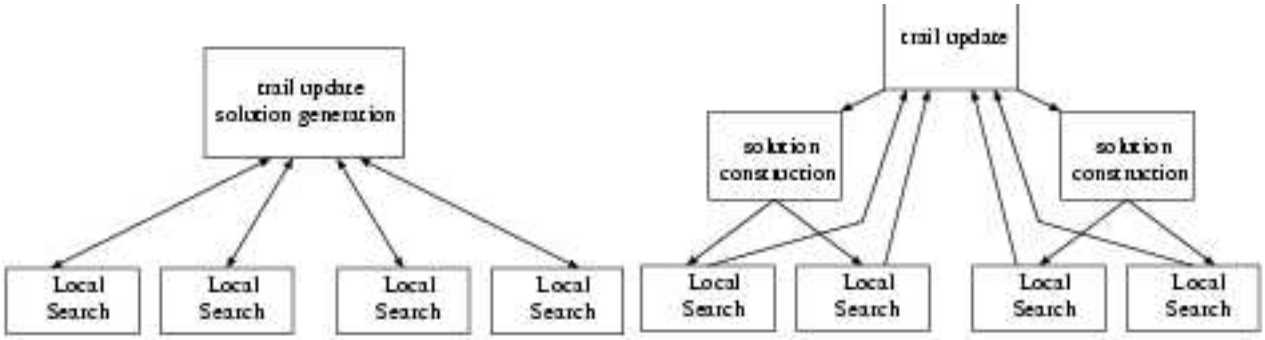

Figure 6: *Proposed architectures (from [10])*

potential problem may occur if we have only a very low computation time so that all we can do is speeding up a single run of an algorithm. It this case the author (extending the work of [2]) presents the two architectures (basically of a master-slave type) shown in figure 6 where is assumed the use of a local search to improve solutions found by pure MMAS. In figure 6 (left) we have one master processor that update the main data structures for MMAS, constructs the initial solutions and sends them to the slave processors so that they can improve them with local search algorithms. The master collects such improved solutions, updates global data structures (essentially the pheromone trails matrix) before passing to the evaluation of new and better solutions. Such a solution is suitable if the update of the trails and the computation of a solution is quicker than the execution of a local search. if this is not the case we can adopt a solution like that in figure 6 (right) where (with a solution similar to that found in [5]) we have one processor that keeps the trail matrix and updates it, one or more processors that use such a matrix to compute solutions and several other processors that receive the solutions and improve them by local search and send them back to the main processor. Such a solution suffers of a communication overhead that lowers speedup with respect to the optimal values.
The solution shown in figure 6 (right) is suitable in the case of MMAS for TSP where $70 - 80\%$ of the time is spent by the local search, $10 - 15\%$ is

spent constructing local tours and the remaining is spent updating the trail matrix. Figure 7 shows the computational results of the execution of parallel

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

Figure 7: *Performances of parallel independent runs of MMAS for TSP (from [10])*

independent runs of MMAS with a maximum of 10 processors. In order to calculate the exact speedup in case of parallel independent runs taking into account the solutions quality the author compares the average solution quality of MMAS running k times for a time t_{max}/k with that of a single run for a time t_{max} taking care of the initialization time t_{init} needed to an algorithm to find high quality solutions so that we have to be sure that $t_{max}/k \geq t_{init}$. Experimental results are shown in figure 7 where the best average solutions are in boldface and t_{max} is the maximum execution time for a sequential algorithm. We note that for *dl*98an optimal solution has not been found.
The next parallel implementation of AS for TSP we are going to examine is the one proposed in [4]. In that paper the authors develop what they call *Parallel Ant Colony Systems* or $PACS$ and propose three communication methods for updating the pheromone levels among groups in PACS. The aim of PACS in not limited at reducing computation time since they develop a parallel formulation which reduces the computation time and gives a better solution.
The first step is the generation of a set of artificial ants and their subdivision in groups. To each group we apply the sequential algorithm AS whereas communication among the groups occurs on a fixed scheduling: communication aims at updating the pheromone level for each route according to the best route found either by neighbouring groups or by all groups.
The PACS the authors propose is shortly described below, for further details we refer the reader to [4].

Data sets	*AS*	*ACS*	*PACS1*	*PACS2*	*PACS3*
EIL101	718.5917	670.8689	646.4660	648.6656	645.9640
TSP225	4536.8308	4199.8483	3939.4237	3934.4802	3916.0839
ST70	711.4132	696.1061	677.4363	677.1765	678.5531

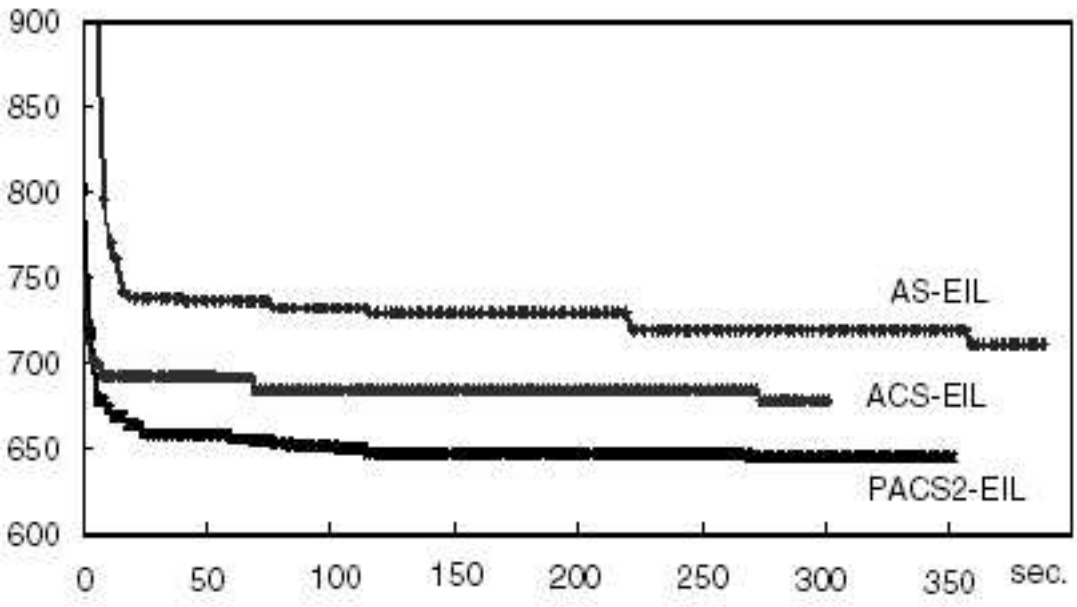

Figure 8: *Performance comparisons for EIL101 data sets (from [4])*

1. **Step 1: Initialization** generate N_j [artificial] ants for jth group of G groups then randomly select an initial city for each ant, initialize the pheromone level on every arc to a small positive value τ_0 and set the cycle counter to 0;

2. **Step 2: Movement** determine from city r the next unvisited city s to be visited by ant i in the group j with a rule that accounts for either the pheromone level and visibility between city r and s in group j or the transition probability from city r to s of the ant i in group j;

3. **Step 3: Local Pheromone Level Updating Rule** is a rule that update the pheromone level between pairs of cities for each group and accounts, through a convex combination, an evaporation factor and of the approximate distance of the route between all cities;

4. **Step 4: Evaluation** of the total length of the route for each ant in each group;

5. **Step 5: Global Pheromone Level Updating Rule** is a rule that updates the pheromone level between cities for each group;

6. **Step 6: Updating from communication** with three methods called **Method 1** and **Method 2** and that differs on the number of cycles they are applied and on the rule by which the pheromone trail on each arc for each group is updated whereas **Method 3** is a mixture of the other two;

7. **Step 7: Termination** increment the cycle counter, move the ants to their initial positions and go back to step 2 until a termination condition is met (maximum number of cycles reached or a stagnation condition satisfied, for instance when all ants take the same route.

In figure 8 we quote a table and a diagram. The table is relative to performance comparisons of three standard data sets ($EIL101$, $TSP225$ and $ST70$) with regard to AS, ACS and PACS with the three method of updating from communications (i. e. PACS1, PACS2 and PACS3) whereas the diagram is relative to performance comparisons of AS, ACS and PACS2 (i.e. PACS with **Method 2**) for $EIL101$ data set.
$EIL101$, $TSP225$ and $ST70$ are data sets with 101, 70 and 225 cities respectively. For AS, ACS they use 80 ants and the same is true for PACS where the ants are divided in 4 groups of 20 ants each. All results in both table and diagram are averaged on 5 runs. In the experiments the number of cycles between updates of the pheromone level due to communications with methods 2 and 3 in PACS have been set respectively to 80 and 30.
The very last example of parallel implementations we outline in this paper is that proposed in [5]. In that paper the authors present Ant Colony optimization algorithms with an application to TSP and two frameworks:

1. a *master-slaves* framework,

2. a *pyramidal framework*

together with the communication scheme and some experimental results. Their aim, again, is to improve the performance of the master-slaves (politically called *workers* in section 6 following [2]) paradigm without modifying the behaviour of the algorithm. The first framework they present is shown in figure 9 and is very similar to the one proposed by [10]. They speak of *parallelization paradigms* with master and slaves as the roles of the PE and

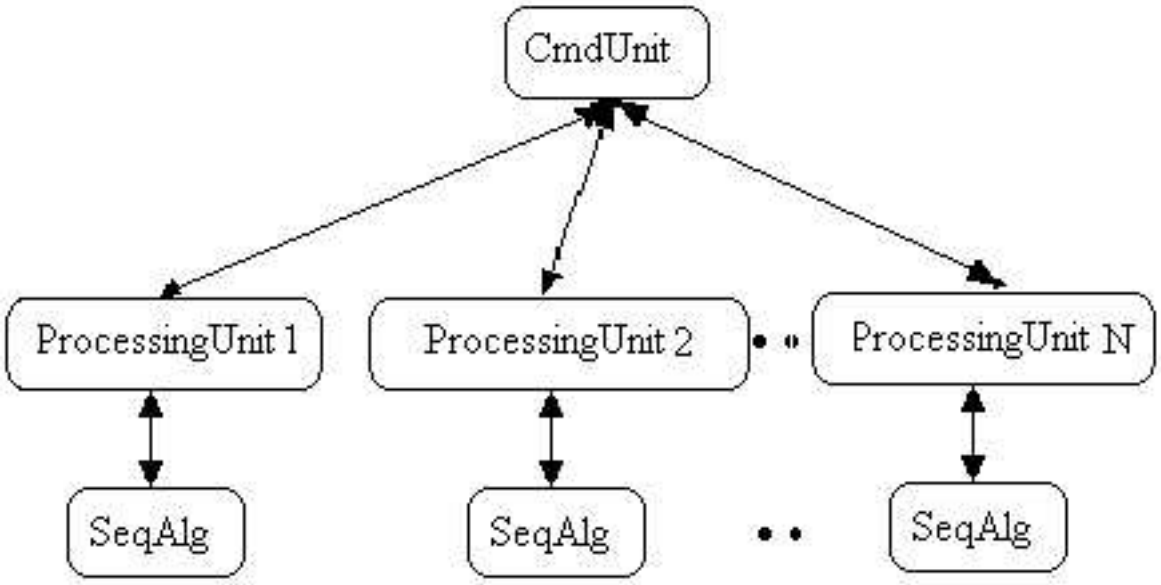

Figure 9: *master-slaves and message-passing (from [5])*

message-passing as the communication structure. The master prepares global data (the usual matrices D and τ_0) and then transfer control to the slaves: each slave runs an instance of the sequential AS and all of the slaves work on the same problem instance. At the end of each iteration the slaves synchronize with the master, send the solution back to the master that updates global data and broadcast them to the slaves for a new iteration. Since, as shown in figure 9 for a TSP instance with 229 cities ($gr229tsp$ from the TSP library), the speedup shows a degradation for a number of CPUs greater than 30 they propose a more complex framework they call *Pyramidal Framework*. Such a framework makes use of one master PE (M), several (unspecified) sub-masters (SM) and several (unspecified) slaves (SL) that communicate with message-passing. A disjoint subset of the Ss is assigned to each SM. At the very beginning M reads the problem instance (for our purposes the graph on which we have to solve a TSP), initializes global data structures, wraps all things up in a message that is broadcasted to the Ss which start the execution while M and SM wait for requests to update the (either global or locally global) data: each S runs a local instance of the sequential algorithm (AS for TSP in our case) that works on a copy of the global data structures. At the end of each local iterations the Ss interact with their SM which in turn, when all or a tunable percent of its Ss are done, synchronizes with the M. The M receives all the changes, updates global data and broadcasts

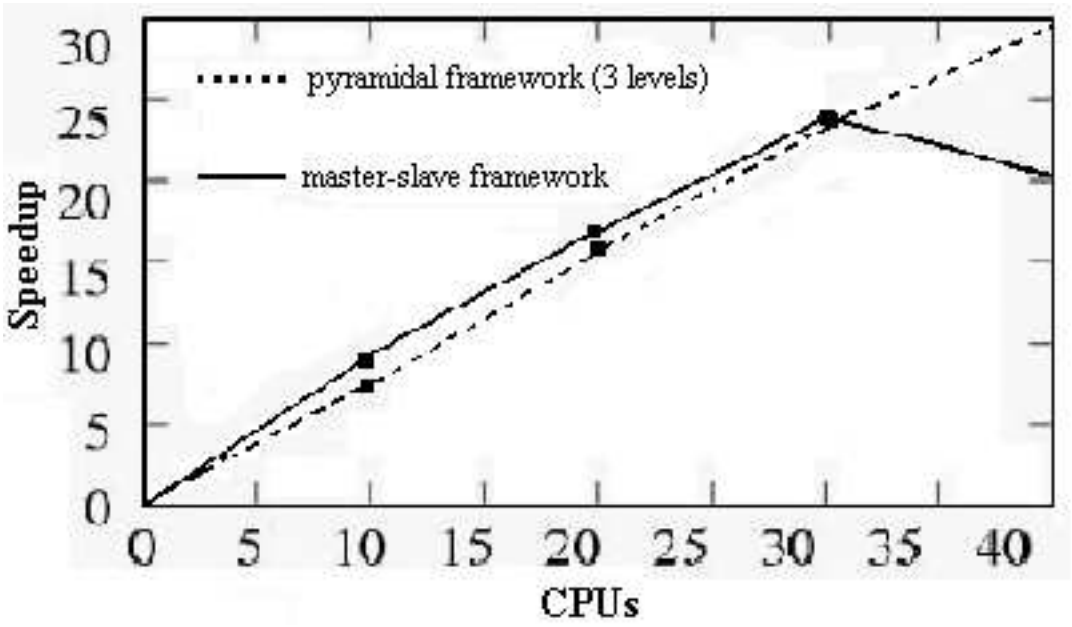

Figure 10: *Speedup in the proposed frameworks (from [5])*

them to the Ss and the SMs (this point is really not clear in the paper). The proposed framework seems similar to that proposed in [2] but since results (in [5] limited to *Speedup*) have been obtained on non homogeneous data the results cannot be compared. One of the critical points of the proposed frameworks is that of the communications (as pointed also in [2]. In [5] two solutions are proposed:

1. serialize the data transfer to the same processor (*scheduling*),
2. send only the data that have changed (*logical clocks*).

The use of *scheduling* greatly reduces communication time (and so communication overhead) and this reduction varies directly with the number of PEs whereas with *logical clocks* only data that have changed (for instance an arc on which the amount of pheromone has been increased by an ant or has evaporated) are exchanged among the various types of PEs. For further details we refer the reader to [5].
As already stated, figure 10 shows the experimental results in case of a TSP instance with 229 cities that has been run on a *Sun Fire 15K HPC* service with a backend with 48 processors and the tests runs have been carried out with the number of processors varying from 10 to 40. From figure 10 we see that the master-slave framework gets an acceptable speedup for up to 26

PEs whereas the pyramidal framework maintains an approximatively linear speedup also for a greater number of PEs.

8 Conclusions

When i started planning this paper my intention was to examine (and compare amongst themselves) as many parallel implementations of AS as i could find in literature. After only a few searches i found that mine was a "mission impossible": too many papers in too many areas of application in a too short time. Therefore, having as a starting point [2], i decided to confine myself to parallel implementations of AS for TSP. I, then, started a new search (not a really completely new one) and began finding some materials among which i selected the papers that have been discussed here. My intention, as i stated in the *Introduction* was to "to put on a common ground different proposals with the aim of comparing their results". Unfortunately, as it has been shown in this same paper, such intention has remained unfulfilled simply because, as an even superficial examination reveals, the results cannot be compared amongst themselves either because they have been obtained using ad hoc problem instances or because they have been obtained by using different and disjoint generally available and typical data sets for TSP. The paper is, therefore, simply a partial survey of some parallel implementations of AS for TSP whereas it is not clear if the original intention deserves further investigations or not, but time is a gentleman.

References

[1] Christian Blum and Andrea Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. In *ACM Computing Surveys*, volume 35 n^0 3, pages 268–308. ACM, September 2003.

[2] Bernd Bullnheimer, Gabriele Kotsis, and Christine Strauβ. Parallelization Strategies for Ant System. In *HPSNO 1997*. Kluwer Series of Applied Optimization, 1997.

[3] Gianni Di Caro. *Swarm Intelligence. Nature's way to system engineering.* slides of a Doctoral Course, IDSIA, USI/SUSPI, Lugano (CH), April 26-27 2005.

[4] Shu-Chuan Chu, John F. Roddick, Jeng-Shyang Pan, and Che-Jen Su. Parallel Ant Colony Systems. In *ISMIS 2003*, volume LNAI 2871, pages 279–284. Springer-Verlag Berrlin Heidelberg, 2003.

[5] Mitica Craus and Laurentiu Rudenau. *A Pyramidal Parallel Framework for Ant-Like Algorithms.* Course material, BCRI-UCC, Cork, Ireland, July 6, 2004.

[6] Karl F. Doerner, Richard F. Hartl, Guenter Kiechle, Maria Lucka, and Marc Reimann. Parallel Ant Systems for the Capacitated Vehicle Routing Problem. In *EvoCOP 2004*, volume LNCS 3004, pages 72–83. Springer-Verlag Berrlin Heidelberg, 2004.

[7] Karl F. Doerner, Richard F. Hartl, and Maria Lucka. A parallel version of the D-Ant algorithm for the Vehicle Routing Problem. In *Parallel Numerics*, pages 109–118, 2005.

[8] Marco Dorigo. *Optimization, learning, and natural algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992.

[9] Bernard Gendron. *Parallel Computing in Combinatorial Optimization.* Course material, Doctoral Course on Parallel Computing in Combinatorial Optimization, Pisa, June 9-June 30 2005.

[10] Thomas Stützle. Parallelization Strategies for Ant Colony Optimization. In *PPSN V*, volume LNCS 1498, pages 722–731. Springer-Verlag Berrlin Heidelberg, 1998.

[11] El-Ghazali Talbi, Olivier Roux, Cyril Fonlupt, and Denis Robilliard. Parallel Ant Colonies for Combinatorial Optimization Problems. In

IPPS/SPDP Workshops, volume LNCS 1586, pages 239–247. Springer-Verlag Berrlin Heidelberg, 1999.

[12] Jitian Xiao and Huaizhong Li. Sequential and Parallel Ant Colony Strategies for Cluster Scheduling in Spatial Databases. In *ISPA 2004*, volume LNCS 3358, pages 656–665. Springer-Verlag Berrlin Heidelberg, 2004.