

# Ant Colony Optimisation in Parallel

Sean Kelly

September 15, 2008

## **Abstract**

Ant Colony Optimisation algorithms have been applied successfully to solve many combinatorial optimization problems such as the Travelling Salesman Problem. Here we investigate parallel implementations of the high performing Max-Min Ant System algorithm. We implement a coarse grained parallelization with one colony of ant agents per compute node. We propose two communication strategies one with a high communication overhead in which best-so-far solutions are exchanged and the other with substantially lower communication volume in which solutions are compared in a meta-algorithmic framework applying a mixed pheromone update schedule across all process during the initial stages of the algorithm run. The latter scheme proves to be most effective against all variations considered. We test our proposed implementations against serial and parallel zero-communication schemes and give reasons for differences among the topologies considered.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>3</b>  |
| 1.1      | Ant inspired algorithms . . . . .                                     | 4         |
| 1.2      | Parallel Implementations . . . . .                                    | 5         |
| 1.3      | MMAS parallel implementation . . . . .                                | 6         |
| <b>2</b> | <b>Max-Min Ant System for the Traveling Salesman Problem</b>          | <b>8</b>  |
| 2.1      | The Travelling Salesman Problem . . . . .                             | 8         |
| 2.2      | Ant Algorithms for Travelling Salesman Problem . . . . .              | 9         |
| 2.2.1    | Tour Construction . . . . .   | 9         |
| 2.2.2    | Pheromone Update . . . . .  | 9         |
| 2.2.3    | Pheromone Re-initialisations . . . . .                                | 10        |
| 2.2.4    | Local search extensions . . . . .                                     | 10        |
| 2.3      | MMAS: convergence and pheromone bounds . . . . .                      | 11        |
| 2.4      | Summary . . . . .   | 14        |
| <b>3</b> | <b>Code Structure</b>   | <b>15</b> |
| 3.1      | Data Structures . . . . .   | 15        |
| 3.1.1    | Problem Data structures . . . . .                                     | 15        |
| 3.1.2    | Ant agent Data structures . . . . .                                   | 17        |
| 3.2      | Algorithm Implementation . . . . .                                    | 17        |
| 3.2.1    | Data Initialisation . . . . .   | 18        |
| 3.2.2    | Solution Construction . . . . .                                       | 19        |
| 3.2.3    | Local Search . . . . .  | 20        |
| 3.2.4    | Pheromone Update . . . . .  | 20        |
| 3.2.5    | Pheromone Re-initialisation . . . . .                                 | 21        |
| 3.3      | Parallel Implementation . . . . .                                     | 22        |
| 3.3.1    | Directed Ring: exchange of best-so-far solution . . . . .             | 22        |
| 3.3.2    | Fully Connected: exchange of tour length and update<br>flag . . . . . | 23        |
| 3.3.3    | Solution collection and Sorting . . . . .                             | 24        |
| 3.4      | Code Layout and Command line Input . . . . .                          | 24        |
| 3.5      | Summary . . . . .   | 26        |

|          |   |           |
|----------|---|-----------|
| <b>4</b> | <b>Experimental Setup</b>   | <b>27</b> |
| 4.1      | IITAC cluster @ TCHPC . . . . .                                       | 27        |
| 4.2      | Experimental Procedure . . . . .                                      | 27        |
| 4.2.1    | Directed Ring: exchange of best-so-far solution . . . . .             | 28        |
| 4.2.2    | Fully Connected: exchange of tour length and update<br>flag . . . . . | 28        |
| <b>5</b> | <b>Results and Analysis</b>   | <b>30</b> |
| 5.1      | Directed Ring Topology . . . . .                                      | 30        |
| 5.2      | Fully Connected Topology . . . . .                                    | 32        |
| 5.3      | Code Validation . . . . .   | 34        |
| <b>6</b> | <b>Conclusion</b>   | <b>35</b> |
|          | <b>Bibliography</b>   | <b>37</b> |
| <b>A</b> | <b>Communication code: Directed Ring</b>                              | <b>40</b> |
| A.1      | Ring Topology:best-so-far solution exchange . . . . .                 | 40        |
| A.1.1    | Exchange Phase . . . . .  | 40        |
| A.1.2    | Termination Phase . . . . .   | 41        |
| <b>B</b> | <b>Communication code: Fully Connected</b>                            | <b>43</b> |
| B.1      | Fully Connected: exchange of best solution and update flag . . . . .  | 43        |
| B.1.1    | Exchange Phase . . . . .  | 43        |
| B.1.2    | Termination Phase . . . . .   | 45        |

# Chapter 1

## Introduction

Ants complex social behaviour have many parallels to that of Human societies. Ant societies have division or specialisation of labour force, communication between individuals and have the ability to solve complex problems. Ants are typically organised within colonies ,and although they may act individually within the colony, display a collective or swarm intelligence as a whole. This cooperation allows an individual ant within a colony to far exceed its individual capabilities. Colonial activities such as foraging for food in some species such as the Argentine ant *Iridomyrmex humilis* [1] indicates an ability to solve shortest path problems. A familiar sight to many of ant behaviour is that of ant lines or streets ,channels in which ants move back and forth between their nest in search of food. This cooperation is facilitated through the spraying of chemical messages by pheromones laid down by each ant as it searches its surroundings for food. Successive ants "smell" these pheromones and influence their decisions to follow such a path on the strength of these chemical signals. Over successive generations of ants this distributed system evolves into the avenues in which we see as lines of ants, collectively searching for an optimal path between their nest and food source.

These natural optimisation techniques provide the inspiration for Ant Colony Optimisation (ACO) algorithms used today to solve many *NP-Hard* optimisation combinatorial problems and are among the best performing algorithms for many problems. ACO algorithms have been employed successfully to many real-world problems like finding a minimum cost plan to deliver goods to customers, optimal assignment of employees to tasks, optimum routing scheme for data packets in the Internet,optimal sequence of jobs which are to be processed in a production line, flight crew allocations for airlines to name but a few [2].

ACO techniques take a metaheuristic approach evolving solution quality towards optimal values rather than exhausting all possible combinations of a solution space.ACO algorithms combine a priori information about the

structure of a promising solution with a posteriori information about the structure of previously obtained good solutions. The main theme of ACO techniques is that the highly coordinated behaviour of real ants is exploited by colonies of artificial agents to solve computational problems. Different aspects of ant behaviour such as foraging, division of labour, brood sorting and collective movement have inspired many different types of ACO algorithms. In the following sections we will describe the development of ACO algorithms Sec. 1.1, recent attempts to find effective parallel implementations Sec. 1.2 and Sec. 1.3 outlines our proposed parallel implementation of the high performing MMAS. In the proceeding Chapter 2 outlines the mathematical background, Chapter 3 analyses the code structure, Chapter 4 outlines the experimental setup and procedure, Chapter 5 presents analysis of our results and Chapter 6 provides a conclusion and considers future implementations.

## 1.1 Ant inspired algorithms

The first ACO algorithm called Ant System(AS) [3, 4, 5] was applied to solve the classic combinatorial problem of Travelling Salesman Problem(TSP). TSP is a problem of a salesman starting from his home who needs to travel to a number of cities , visiting each city once in a round trip and returning home by the shortest route possible. Throughout the development of ACO algorithms the TSP has been used as a benchmark to display the behaviour of ant algorithms without obscuring them in technicalities. Good performance on this problem has often proved as an indicator of usefulness for other combinatorial problems [2]. The two main phases of AS constitute the ants solution construction and pheromone update. For the TSP the Ant System algorithm sends out ant agents to search routes on the map. Ants stochastically choose the next city to visit based on a heuristic combining the distance to the city and the amount of virtual pheromone deposited on the edge of the city. Ants deposit pheromone on each edge that they cross visiting each city once in completing a tour. Pheromone evaporation; a constant decrement in the pheromone values of the problem components, is also applied after a solution is constructed controlling the influence older solutions have on future solutions. Today many of AS direct extensions are among the best performing combinatorial optimisation algorithms. These extensions include elitist Ant System, rank-based Ant System [6], and Max Min Ant System (MMAS)[7]. These algorithms's have a more sophisticated approach in the way the pheromone update is performed and also in the management of the pheromone trails and provided a large improvement on the initial AS algorithm. Algorithm 1.1 shows a psuedocode outline of how Ant System( and extensions) can be applied to a static problem such as the Traveling Salesman.

---

**Algorithm 1** ACO algorithm applied to static Combinatorial Problem

---

*initialize pheromone trails and parameters*

**while** *termination condition not met* **do**

*ConstructAntsSolutions*

*ApplyLocalSearch*

*UpdatePheromones*

**end while**

---

The use of a Local search procedure is shown in the Algorithm 1.1. The addition of a local search procedure has been shown to improve significantly the performance of AS and its extensions particularly MMAS [7]. Local search is itself a metaheuristic which can be applied to a number of candidate solutions of a combinatorial optimisation problem. The main idea behind a local search is to take a route that may cross over itself and re-order it so that it does not. When combined with an ACO such as MMAS after ants have constructed a tour it has been shown to improve the algorithm performance although with increased computational overhead.

The high performing Max-Min Ant System (MMAS) will be the focus of our study in this report. MMAS is an extension of AS which updates the pheromone matrix; the evolving stochastic bias component of the heuristic, in a more sophisticated manner. As the name suggests limits are imposed on the values a pheromone trails can take. MMAS updates either the iteration best or the best-so-far solution (ant) only. Implementations of the algorithm often feature a mixed pheromone update strategy which alternates between iteration best solution (one colony provides one solution per ant agent per iteration) or the best-so-far solution (overall current best solution) update. Any update schedule generally allows the iteration best solution (ant) to be updated more frequently in the earlier stages of the search thus allowing more of the search space to be explored. As the algorithm progresses the best-so-far ant is updated more frequently until near convergence where only the best-so-far ant is updated. MMAS can also be implemented with occasional re-initialisations when the algorithm displays signs of stagnation behaviour. These extensions of AS implemented by MMAS significantly improved on AS performance and today it is one of the best performing combinatorial optimization metaheuristics.

## 1.2 Parallel Implementations

Ant algorithms are obvious candidates for parallelization. Investigations regarding the best strategy have indicated that coarse grained approaches which typically involve one colony of artificial agents per compute node have proved most effective. These are similar in many ways to the island approach for parallel implementation of the Genetic Algorithm [11, 12]. Investigations

of the appropriate object for information to exchange studied solutions, parameters and pheromone levels [8, 9, 10] They indicated that exchanging of the best-so-far solution proved the best with respect to increasing solution quality. Investigations by Manfrin et al(2006) [13] studied the exchange of colonies best-so-far solution for various communication topologies to solve the Travelling Salesman Problem(TSP) using the high performing Max-Min Ant System(MMAS) algorithm. Each topology differed in the amount of communication overhead employed.Their implementation only updated the pheromone trail of the best-so-far solution and also neglected the use of the occasional re-initialization of a colonies pheromone trail often used in any serial MMAS implementation [7].Each topology was tested against a multi-colony zero-communication scheme in which essentially  $p$  copies of the algorithm were run independently over  $p$  computing nodes. The zero-communication scheme in contrast used both a mixed pheromone update schedule ([7],Sec 3.2.4) and occasional pheromone re-initialisations when stagnation (Sec 2.2.3) of the algorithm occurred. Their findings showed that the zero-communication scheme proved to be the best performing implementation and also that no statistical difference existed between the various communication topologies implemented. Analysis showed that communication of best-so-far solutions among colonies caused stagnation of the algorithm and as a result limited the implementations searching of solution space. This lead to acceleration of the colonies towards convergence of the same solution. It proved that a mixed update schedule and occasional re-initialisations to be more important factor than best-so-far solution exchange over all the topologies for improving performance. It must be noted that the former factor , that of the mixed pheromone update schedule has been shown to have a greater influence on solution quality than re-initialisations for serial implementations of MMAS [7]. In this report we propose to implement variations on the coarse grained(one colony per compute node)parallel schemes for the MMAS.

### 1.3 MMAS parallel implementation

Our first proposed implementation exchanges the best-so-far solution between colonies as done by Manfrin et al. [13] but considers a **non-homogeneous** approach by varying the type of local search used by each colony. As discussed in Sec 1.1 coupling a local search with MMAS was found to significantly improve solution quality although resulted in significantly increased computation time for solution construction [7]. In investigations by Manfrin et al. [13] a **3-opt** local search was used. A *k-opt* local search investigates how a solution candidate can be improved by exchange of at most  $k$  arcs of a solutions component set(cities in the case of the TSP). This has a computational overhead of complexity time  $O(n^3)$  while searches such as the

2.5-opt have an overhead of approximately  $O(n^2)$ . We propose to vary each colonies local-search procedure between a 3-opt and a 2.5-opt local search for colonies exchanging a best-so-far solution. The communication topology employed here is that with the lowest possible communication overhead: a single nearest neighbour or directed ring topology. Both Synchronous and Asynchronous implementations of the topology are considered.

Our second proposed implementation differs from recent attempts in that solutions are compared but not exchanged. This results in a very small communication overhead as only single integer values are exchanged between colonies. Studies of parallel implementations of MMAS by Stutzle and Hoos [7] indicated that an optimum is more likely to be found by updating the best-so-far solution but is also more likely to result in convergence to a local optimum. This lead them to propose a mixed schedule for pheromone update between the *iteration best* (one colony provides one solution per ant agent per iteration) and *best-so-far solution* (overall current best solution). The schedule allowed the iteration best to be updated more frequently in the earlier stages of the search thus allowing more of the search space to be explored. As the algorithm progresses the best-so-far ant is updated more frequently until near convergence where only the best-so-far ant is updated. We employ a mixed pheromone update schedule across all processes for the initial stages of the algorithm run. In this schedule (Sec 4.2) the worst performing colonies update the iteration-best solution up until the schedule finishes and then all processes update the best-so-far solution regardless of performance. Our implementation includes the use of occasional re-initializations or restarts of the pheromone values when stagnation occurs by all colonies.

## Chapter 2

# Max-Min Ant System for the Traveling Salesman Problem

In this Chapter we consider the mathematical background associated with the Travelling Salesman Problem (TSP) and the Max-Min Ant System (MMAS) Ant Colony Optimisation (ACO) algorithm used to solve it. In the following Section 2.1 details the TSP. Section 2.2 details how MMAS is applied to TSP. Section 2.3 considers a convergence proof for MMAS and investigates the upper pheromone trail bound.

### 2.1 The Travelling Salesman Problem

The TSP is an *NP* hard combinatorial optimisation problem. Given a problem instance  $H$  there exists a *NP*-complete problem  $L$  such that is Turning Machine (abstract device to simulate any algorithm) reducible and solvable in polynomial time to  $H$  such that

$$L \leq H_T$$

The TSP can be represented by a complete weighted graph  $G = (N, A)$ , where  $N$  is the set of nodes representing the cities,  $A$  the set of arcs between each city. Each arc  $(i, j) \in A$  has a length values  $d_{ij}$  the distance between cities  $i$  and  $j$  with  $(i, j) \in N$ . Here we consider instances of the TSP which are symmetric such that  $d_{ij} = d_{ji}$  for all arcs in  $A$ . Any solution attempts to find the minimum length Hamiltonian of a circuit of the graph for all nodes  $n \in N$ . An optimal solution to the problem is a permutation  $\mu$  of of the nodes  $i \in \{1, 2, \dots, n\}$  such that the length function  $f(\mu)$  is minimised ,  $f(\mu)$  given by

$$f(\mu) = \sum_{i=1}^{n-1} d_{\mu(i)\mu(i+1)} + d_{\mu(n)\mu(1)}$$

Most instances of the TSP are subject to a metric constraint where distances in the plane obey the Triangle Inequality. That is, for any three cities  $A$ ,  $B$  and  $C$

$$|AC| \leq |AB| + |BC|$$

## 2.2 Ant Algorithms for Travelling Salesman Problem

In ACO implementations for the TSP each arc on the graph has an associated pheromone value  $\tau_{(i,j)}$  which reflects the desirability of visiting city  $i$  after city  $j$ . The heuristic guide is chosen as  $\eta_{(i,j)} = 1/d_{(i,j)}$ . Pheromone and Heuristic values for each city node are stored in suitable matrix form for each problem instance. The two main phases of a ACO algorithm constitute the ants solution construction Section 2.2.1 and pheromone update Section 2.2.2. In this implementation occasional pheromone re-initialisations Section 2.2.3 and local search Section 2.2.4 extensions are also included.

### 2.2.1 Tour Construction

A tour consists of one stochastic circuit of the graph guided by the meta-heuristic shown in equation 2.1. A solution is constructed firstly by choosing a start city for each ant in the colony, then each ant agent stochastically constructs a tour using the pheromone and heuristic bias and iteratively adding an unvisited city until it completes its tour. The probability that an ant arriving at city  $i$  chooses city  $j$  from the set  $S$  of cities not visited is given by the probability

$$p_{(ij)} = \frac{(\tau_{ij})^\alpha (\eta_{ij})^\beta}{\sum_{k \in S} (\tau_{ik})^\alpha (\eta_{ik})^\beta} \quad (2.1)$$

where  $\alpha$  and  $\beta$  are constants that determine the influence of the pheromone and heuristic bias of the ant.

### 2.2.2 Pheromone Update

After all ants have completed a tour the pheromone trails are updated. Firstly pheromone evaporation takes place in which pheromone values on all trails are reduced by a constant factor.

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}, \quad \forall (i, j) \in L \quad (2.2)$$

where  $\rho$  is the evaporation rate ( $0 < \rho \leq 1$ ). This parameter is used to limit the accumulation of pheromone trails and reduces the influence of bad decisions. It essentially enables the system to "forget" bad choices limiting their influence on later decisions. After evaporation pheromone deposition

takes place. Most ACO variants differ in how they update the pheromone trail. As discussed earlier we implement Max-Min Ant System(MMAS) a direct extension of Ant System(AS). In MMAS the pheromone trail values are constrained to the interval  $[\tau_{max}, \tau_{min}]$ . These limit the probability  $p_{ij}$  of selecting a city  $j$  when ant is at a city  $i$  to the interval  $[p_{min}, p_{max}]$ . It is shown in Section 2.3 that  $\tau_{max}$  the upper pheromone trail limit is bounded above by  $\frac{1}{\rho C^*}$ , where  $\rho$  is the pheromone evaporation rate and  $C^*$  is the length of the optimum tour. Each time a best-so-far solution is found MMAS uses an estimate of this value to update  $\tau_{max}$ . The lower pheromone trail limit  $\tau_{min}$  is set to a fraction of  $\tau_{max}$  with  $\tau_{min} = \frac{\tau_{max}}{a}$  where  $a = 2$  here( Sec 2.3, [8]). In order to make MMAS more exploratory in its initial phases the pheromone trails are set to an estimate of the upper pheromone trail limit along with a small pheromone evaporation rate  $\rho$  (set to 0.2 ) here see Sec 3.2.4.Implementations of the algorithm also often feature a mixed pheromone update strategy which alternates between *iteration best solution* or *the best-so-far solution* update. Any update schedule generally allows the iteration best solution(ant) to be updated more frequently in the earlier stages of the search thus allowing more of the search space to be explored. As the algorithm progresses the best-so-far ant is updated more frequently until near convergence where only the best-so-far ant is updated.

### 2.2.3 Pheromone Re-initialisations

Pheromone trail re-initialisations are implemented here when the algorithm displays stagnation behaviour. This criterion is established by calculation of the average  $\lambda$ -branching factor [19] calculated across each node on the problem graph. The  $\lambda$  branching factor shows for a city  $i$  the number of choice arcs which have a dominant pheromone bias. If  $\tau_{max}^i$  and  $\tau_{min}^i$  is the maximum/minimum pheromone trail value on arcs incident to city  $i$ , then the  $\lambda$ -branching factor is given by the number of arcs incident to  $i$  that have a pheromone trail value

$$\tau_{ij} \geq \tau_{min}^i + \lambda(\tau_{max}^i - \tau_{min}^i), \quad (2.3)$$

where  $(0 < \lambda < 1)$ . For TSP this corresponds to an interval  $(2 \geq \lambda_{branch} < n - 1)$ . An unchanging  $\lambda$ -branching factor of  $\lambda_{branch} \approx 2$  ( with  $\lambda = 0.05$  here ) is a good indication stagnation behaviour in an ACO algorithm for the TSP.

### 2.2.4 Local search extensions

The main idea behind a local search is to take a route that may cross over itself and re-order it so that it does not.Iterated local search procedures are themselves some of the best performing algorithms for solving static Combinatorial Optimization Problems such as the TSP [14].Local search

as the name implies tries to improve a current solution by local changes. Once an ant has completed their solution construction, the solutions can be taken to their local optimum by a local search procedure. The types of local changes that may be applied are defined by a neighbourhood structure which is a function which assigns neighbours to a solution  $s \in S$ . A *k-exchange* neighbourhood of a candidate solution  $s$  is the set of candidate solutions which can be obtained from  $s$  by exchanging at most  $k$  solutions. A *k-opt* local search consists of a *k-opt* neighbourhood which can be obtained from tour  $s$  by replacing at most  $k$  of its arcs.

Local search procedures of increasing strength namely **2.0-opt**, **2.5-opt** and **3.0-opt** have all been successfully applied with MMAS significantly improving the algorithm's performance [7, 15]. Note that a **2.5-opt** is an extension of a **2-opt** search which when checking for a **2-opt** improvement also checks whether inserting a city between two cities  $i$  and  $j$  (say) results in an improved tour. This only increases the computational time complexity slightly from a **2-opt** search but leads to a significant improvement [16] in solution quality. Typically computational complexity for **3.0-opt** local searches is of the order  $O(n^3)$  and for **2.5-opt** of the order  $O(n^2)$ .

### 2.3 MMAS: convergence and pheromone bounds

ACO is a stochastic constructive algorithm in which the bias due to pheromone trails could possibly prevent the optimum being reached. As a result it is important to consider whether our ACO algorithm specifically the MMAS which we implement here converges. In the following section we present a proof for MMAS optimal convergence. We firstly make the distinction between convergence in value against convergence in solution. The former implies that the procedure will generate an optimum solution once and the latter that on reaching an optimum the solution continues to generate the same optimal solution. It is sufficient to proof convergence for the former in this case as once our algorithm generates an optimal solution it discontinues its search.

We consider the general case of a minimization problem given by the triplet  $(S, F, \Omega)$  where  $S$  is the set of possible solutions to a problem instance,  $f$  the cost function associated with the solution with  $f(s) \quad \forall s \in S$  and  $\Omega$  is the set of constraints which defines  $S$ . A specific problem instance has components  $C = \{c_1, c_2, \dots, c_N\}$ , where the set of all possible sequences  $X$  with  $x = (c_i, c_j, \dots, c_h, \dots) \quad \forall x \in X$ . We consider an ACO algorithm with the probabilistic construction value (independent of the heuristic value  $\eta_{ij}$ )

$$P_\tau(c_{h+1} = j | x_h) = \begin{cases} \frac{(\tau_{ij})^\alpha}{\sum_{(i,l) \in \Gamma_i^k} (\tau_{il})^\alpha} & \text{if } (ij) \in \Gamma_i^k \\ 0 & \text{o.w.} \end{cases} \quad (2.4)$$

where  $\tau_{ij}$  is the pheromone value which reflects the desirability of visiting city  $i$  after city  $j$ ,  $\Gamma_i^k$  is the neighbourhood of ant  $k$  which satisfies constraints  $\Omega$ . After a tour construction we restrict the pheromone update to the *best-so-far solution*  $S_\theta = s^{bs}$ , where  $\theta$  is the current iteration index and  $s^{bs}$  represents the best-so-far solution. We also introduce a lower limit on the pheromone trail values  $\tau_{min}$  such that  $\tau_{min} > 0$ . We will now prove convergence for this hypothetical ACO algorithm denoted  $ACO_{\tau_{min},bs}$  and then extend the proof to show convergence for the MMAS.

**Theorem 1** *Let  $P^*$  be the probability the algorithm finds an optimal solution at least once within the first  $\theta$  iterations. Then for an arbitrarily small  $\epsilon$  and a sufficiently large  $\theta$*

$$P^* \geq 1 - \epsilon \quad (2.5)$$

and

$$\lim_{\theta \rightarrow \infty} P^* = 1 \quad (2.6)$$

*Proof: Pheromone limits  $\tau_{min}$  and  $\tau_{max}$  guarantee that any feasible choice of with the probability in Equation 2.1 for any partial solution  $x_n$  is made with probability  $p_{min} > 0$ . A lower bound for  $p_{min}$  would be given by*

$$p_{min} \geq \bar{p}_{min} = \frac{(\tau_{min})^\alpha}{(N_c - 1)(\tau_{max})^\alpha + (\tau_{min})^\alpha}$$

where  $N_c$  is the number of elements in the set  $C$  of components. Any optimal solution  $s^* \in S^*$  can be found with  $\bar{p} \geq p_{min} + (\tau_{min})^{alpha}$ . A lower bound  $P^*(\theta)$  is given by

$$P^*(\theta) = 1 - (1 - \bar{p})^\theta$$

and by choosing  $\theta$  large enough then for arbitrarily small  $\epsilon$  we have

$$P^*(\theta) \geq 1 - \epsilon$$

and thus convergence is implied by the limit

$$\lim_{\theta \rightarrow \infty} P^*(\theta) = 1$$

Now that we have shown convergence for our hypothetical algorithm  $ACO_{\tau_{min},bs}$  it is easily extended to a proof of convergence for the MMAS algorithm. Firstly the addition of the heuristic value to the probability (Equation 2.1) has only the affect of changing the lower bound  $p_{min}$  since  $\eta_{ij} > 0$ ,  $\eta \in [\eta_{min}, \eta_{max}]$ . In regard to pheromone updating there are only two small differences between  $ACO_{\tau_{min},bs}$  and *MMAS*. *MMAS* uses an explicit value rather than an implicit value used by  $ACO_{\tau_{min},bs}$  for calculating  $\tau_{max}$ . *MMAS* updates either updates the best-so-far or the iteration best ants pheromone trail unlike  $ACO_{\tau_{min},bs}$  which just updates the best-so-far but as *MMAS* algorithm progresses the iteration best is chosen

less and less until a situation exists after the initial phase near convergence where it is no longer chosen. It should also be noted that the local search used does not affect the way the solution is constructed and therefore does not affect any convergence towards an optimum. It is clear that the differences between  $ACO_{\tau_{min},bs}$  and  $MMAS$  do not affect convergence of the algorithm and therefore it can be extended that  $MMAS$  converges towards an optimum due to  $ACO_{\tau_{min},bs}$  for sufficient time.

As discussed in section 2.2 we are interested in the upper bound of the pheromone trail  $\tau_{max}$ . Note that this is also used in determining the lower pheromone trail bound as outlined in [9]. After a tour has been constructed in  $MMAS$  pheromone trail  $\tau_{max}$  is recalculated from the updated pheromone values. In the following Theorem we will show that  $\tau_{max}$  is bounded above by  $\frac{1}{\rho C^*}$ .

**Theorem 1** *For any  $\tau_{ij}$  it holds that*

$$\lim_{\theta \rightarrow \infty} \tau_{ij}(\theta) \leq \tau_{max} \quad (2.7)$$

where

$$\tau_{max} = \frac{q_f(s^*)}{\rho}. \quad (2.8)$$

We define  $q_f(s^*)$  as an non-increasing quality function such that  $f(s_1) > f(s_2) \implies q_f(s_1) \leq q_f(s_2)$  for  $s_1, s_2, \dots, s_\theta \in S^\theta$  the candidate solution set up to iteration  $\theta$ . Here  $s^* \in S^*$  the set of optimal solutions.

*Proof:* The largest quantity of pheromone added to any arc  $(i, j)$  after a selection (iteration) is  $q_f(s^*)$ . We assume here that each ant in the colony moves in parallel selecting one city per iteration. At iteration 1 the maximum possible pheromone trail is  $(1 - \rho)\tau_0 + q_f(s^*)$  and at iteration 2  $(1 - \rho)^2\tau_0 + (1 - \rho)q_f(s^*) + q_f(s^*)$  and so on. Hence for any arbitrary iteration  $\theta$

$$\tau_{ij}^{max} = (1 - \rho)^\theta \tau_0 + \sum_{i=0}^{\theta} (1 - \rho)^{\theta-i} q_f(s^*).$$

With  $0 < \rho \leq 1 \implies$

$$\tau_{max} = \frac{q_f(s^*)}{\rho}$$

## 2.4 Summary

Max-Min Ant System (MMAS) algorithm provide a metaheuristic approach to solving the Travelling Salesman Problem(TSP). We have shown that convergence in solution occurs when MMAS is applied to TSP and that the pheromone limits associated with MMAS implementation are bounded ( Sec. 2.3) ensuring that an optimum solution can be found for sufficient run time of the algorithm. In the next chapter we will outline the code structure associated with both our serial and parallel implementations.

## Chapter 3

# Code Structure

In this section we outline our implementation of the Max-Min Ant System (MMAS) Ant Colony Optimisation (ACO) algorithm for the Travelling Salesman Problem (TSP). The algorithm is coded in *C* and makes use of the *Message Passing Interface (MPI)* libraries for parallel coding. The codes implementation is inspired and adapted from the online software resource written by Thomas Stutzle [17]. The termination condition for the algorithm is based on a time constraint of the algorithm run time after initialisation. In the following Section 3.1 details the important data structures used, Section 3.2 the important aspects of the serial algorithm, Section 3.3 the communication code used for the parallel implementation and Section 3.4 details the code layout, compilation information and command line inputs.

### 3.1 Data Structures

The main data structures used are for those representing the problem instance and those representing the artificial ant agents. Section 3.1.1 details the important problem data structures and Section 3.1.2 the important ant agent data structures.

#### 3.1.1 Problem Data structures

Here we outline the important data structures associated with the problem. Listings 3.1 below show the data structures outlined here.

Listing 3.1: Problem Data Structures

```

long int dist[n][n]      /* distance matrix*/
long int nn_list[n][nn] /* nearest neighbour listing*/
double pheromone [n][n] /*pheromone matrix*/
double total[n][n]      /*pheromone and
                           heuristic product*/

```

### Instance Specific

Computation of intercity distances is most efficiently implemented by pre-computation and storage of distances in a matrix of size typically  $n^2$  for TSP instance size  $n$  cities. Problem coordinates are read from file and inter-city distances are computed and stored in the matrix  $dist[i][j]$ . The file *TSP.c* contains the code associated with the reading of the TSP instances and computation of intercity distance according to the specified metric. The TSP instances used here are available from the online software resource TSPLIB [22].

### Nearest-Neighbour List

It is more computationally economical and convenient to store for each city  $i$  a list of its nearest neighbours  $r$  which is precomputed and sorted according to increasing distance from city  $i$ . The position  $p$  of a city  $j$  in city  $i$ 's nearest neighbour list  $nn\_list[i]$  is given by  $nn\_list[i][p]$ .

### Pheromone Trails and Heuristic Information

For each connection  $i, j$  we store our pheromone values  $\tau_{(ij)}$  in a matrix of size  $n^2$ , where  $n$  is the problem size. The pheromone value on the arc from city  $i$  to  $j$  is given by  $pheromone[i][j]$ . The combined stochastic bias of the heuristic and pheromone values for each ants surrounding arcs is computed and stored in the matrix  $total[i][j]$ . Since the value  $\eta_{ij}$  is constant throughout the problem, this value is precomputed and stored for repeated use as a constant.

### Truncation speedups

A large speedup in the algorithm was found by restricting the nearest neighbour list matrix  $nn\_list[i][j]$  and consequently the matrix  $total[i][j]$  to a certain size for large problem instance for TSP [15]. This speedup is implemented here where any cities nearest neighbour list is restricted to its *twenty* nearest cities. It must be noted that the use of truncated nearest neighbour may increase the possibility of a global optimum never being found [15].

### 3.1.2 Ant agent Data structures

An ant agent must be able to store its partial solution, determine its candidate neighbourhood at any point along its tour and compute its solution length. An array *visited*[*j*] of problem instance size *n*; which indicates (with a 1 or 0) if a city *j* has been visited, is updated by an ant agent during its solution construction and can be used to quickly determine an ant's candidate neighbourhood. This reduces the computational overhead which would be required with storing a partial tour and checking this to determine its candidate neighbourhood. Listing 3.2 shows the ant structure used in the implementation.

Listing 3.2: Ant Data Structure

```
ant_structure {
long int  *tour;      /* sequence of cities visited */
char      *visited;  /* indicates if city visited */
long int  tour_length; /* stores tour length */
}
```

## 3.2 Algorithm Implementation

As outlined in Algorithm 1 the main run of the serial algorithm consists of the solution construction, local search and pheromone updating procedure. Additionally we need to parse the command line instructions, initialise and set the parameters before the algorithm run and also update some statistics throughout the procedure. Algorithm 2 shows a high level view of the serial implementation. In the following Section 3.2.1 outlines the data initialisation procedure, Section 3.2.2 the solution construction phase, Section 3.2.3 local search extensions employed, Section 3.2.4 the pheromone updating, Section 3.2.5 the occasional pheromone re-initialisations and Section 3.4 gives information regarding associated files and command line input.

---

**Algorithm 2** Serial implementation of MMAS for TSP

---

```
Parse Commandline
Initialise Data
Initialize pheromone trails and parameters
while Termination condition not met do
    ConstructAntsSolutions
    ApplyLocalSearch
    UpdateStatistics
    UpdatePheromones
end while
```

---

---

**Algorithm 3** Data Initialisation of MMAS for TSP

---

*Read Coordinates for TSP Instance*  
*Compute Intercity Distances*  
*Compute (Truncated) Nearest Neighbour Lists*  
*Initialise Heuristic Pheromone bias*  
*Initialise Ant Structures*  
*Initialise Parsed Parameters*  
*Initialise additional variables to monitor algorithm*

---

### 3.2.1 Data Initialisation

The Data initialisation procedure is outlined in Algorithm 3. It includes the initialisation of data structures outlined in Section 3.1 and also some additional structures required for monitoring statistical aspects of the algorithm.

---

**Algorithm 4** `construct_solutions()`

---

*k* {counter variable}  
*step* {counter of the number of construction steps}  
{Empty Ants memory and Mark all cities as unvisited}  
**for** *k* = 0 to *n\_ants* **do**  
    *ant\_empty\_memory()*  
**end for**  
*step* ← 0  
{Now Place the ants on some initial city}  
**for** *k* = 0 to *n\_ants* **do**  
    *place\_ant()*  
**end for**  
{Main construction procedure Action Choice Rule implemented}  
**while** *step* < *n* - 1 **do**  
    **for** *k* = 0 to *n\_ants* **do**  
        *neighbour\_choose\_and\_move\_to\_next()*  
    **end for**  
**end while**  
*step* ← *n*  
{Ants move back to initial city and tour length completed}  
**for** *k* = 0 to *n\_ants* **do**  
    *ant[k].tour[n]* ← *ant[k].tour[0]*  
    *compute\_tour\_length()*  
**end for**

---

### 3.2.2 Solution Construction

The tour construction procedure for ants solution construction phase consists of three distinct stages. Firstly an ant is assigned a starting city. This assignment is done randomly drawing without replacement from the pool of available cities for each ant. Each ant then constructs a complete tour of the graph. This procedure is synchronised so that each ant constructs one circuit of the problem in parallel moving from city to city per iteration. Finally the ants are moved back to their initial starting city and the tour length is computed. Algorithm 4 shows the psuedocode layout for this procedure.

---

**Algorithm 5** neighbour\_choose\_and\_move\_to\_next()

---

```

k {ant identifier}
i {counter for construction step}
c ← ant[k].tour[i - 1]
sum_probabilities ← 0.0
for j = 1 to nn do
  if ant[k].visited[nn_list[c][j]] then
    selection_probability[j] ← 0.0
  else
    selection_probability[j] ← total[c][nn_list[c][j]]
    sum_probabilities ← selection_probability[j] + sum_probabilities
  end if
end for
if sum_probabilities > 0.0 then
  choose_best_next(k, i)
else
  r ← random[0, sum_probabilities]
  j ← 1
  p ← selection_probability[j]
  while p < r do
    j ← j + 1
    p ← selection_probability[j]
  end while
  ant[k].tour[i] ← nn_list[c][j]
  ant[k].visited[nn_list[c][j]] ← true
end if

```

---

When choosing the next city to move to each ant applies a stochastic choice biased by the metaheuristic guide as discussed. This can be illustrated analogously to the use of a roulette wheel selection procedure where certain outcomes of a roulette wheel spin are weighted more heavily than others [15]. The roulette wheel choice is implemented using the truncation speedups as discussed in Sec 3.1.1 above. This is done by firstly sum-

---

**Algorithm 6** *choose\_best\_next()*

---

```
k {ant identifier}
i {counter for construction step}
v  $\leftarrow$  0.0
c  $\leftarrow$  ant[k].tour[i - 1]
for j = 1 to n do
  if j! = ant[k].visited[j] then
    if total[c][j] > v then
      nc  $\leftarrow$  j
      v  $\leftarrow$  total[c][j]
    end if
  end if
end for
ant[k].tour[i]  $\leftarrow$  nc
ant[k].visited[nc]  $\leftarrow$  true
```

---

ming the probability weights of the various choices  $c$  available using the  $total[j][c]$  matrix. Then a random number  $r$  is drawn from the interval  $[0, sum\_probabilities]$ . The procedure then goes through each candidate city until the sum of the probabilities up to that point is greater than  $r$ . The function *neighbour\_choose\_and\_move\_to\_next()* shown in Algorithm 5 outlines the pseudocode for implementation of this roulette wheel procedure. When using truncated speedups for nearest neighbour candidate lists the situation may arise that an ant  $k$  has visited all of its nearest neighbours on the truncated candidate list. In this case the function *choose\_best\_next()* is used to identify the city outside the Nearest Neighbour list with the highest metaheuristic value. Algorithm 6 displays the pseudocode procedure for the function *choose\_best\_next()*.

### 3.2.3 Local Search

After the solution construction phase a local search procedure is applied. A 2.5-opt and a 3-opt local search (as discussed in Sec. 2.2.4) are applied here. The version of local search implemented here is not the most efficient available [17] but is sufficient to demonstrate the gains to be achieved by applying various local search techniques to MMAS. Several standard speed-up techniques such as Truncated Nearest-Neighbour lists, fixed radius nearest neighbour search and don't look bits [16] are used here.

### 3.2.4 Pheromone Update

The two main stages of the pheromone update consist of pheromone evaporation and pheromone deposition. Algorithm 7 outlines the pheromone

---

**Algorithm 7** pheromone\_update()

---

```
for  $k = 1$  to n_ants do  
    mmas_evaporation_nn_list()  
    mmas_update()  
    compute_nn_list_total_info()  
end for
```

---

update procedure. When local search is used with truncation speedups only evaporating pheromone trails on the solutions nearest neighbour list reduces computational overhead. This speedup when tested for MMAS with local search showed no significant decrease in solution quality was observed when this was implemented for large problem instances [18]. The function *mmas\_evaporation\_nn\_list*() performs this procedure and also bounds the pheromone trails in the interval  $[\tau_{max}, \tau_{min}]$ . Pheromone deposition is carried out by the function *mmas\_update*(). Pheromones are updated either on the iteration best or the best-so-far solution. We use the mixed update schedule given by Stutzle et Hoos [7] for implementations involving mixed updates. This schedule is given in figure below. Finally the product of the heuristic values and the new pheromone values is calculated for the nearest neighbour list (since these are the only pheromone values changed). This is performed by the function *compute\_nn\_list\_total\_info*() which stores the values in the *total*[*i*][*j*] matrix.

The mixed update schedule used by our serial implementation of MMAS (Sec. 2.2.2, [7]) is shown in Equation 3.1 where  $l_{bs}(\theta)$  is the frequency at which the best-so-far solution is updated at iteration  $\theta$ .

$$l_{bs}(\theta) = \begin{cases} 0 & \text{if } 0 < \theta \leq 25, \\ 5 & \text{if } 25 < \theta \leq 75, \\ 3 & \text{if } 75 < \theta \leq 125, \\ 2 & \text{if } 125 < \theta \leq 250, \\ 1 & \text{if } \theta > 250 \end{cases} \quad (3.1)$$

### 3.2.5 Pheromone Re-initialisation

As discussed in Sec. 2.2.3, pheromone re-initialisations are implemented when stagnation behaviour is observed in the algorithm. The function *search\_and\_control\_statistics*() calculates  $\lambda$ -branching factor occasionally (approx every 100th iteration) and if it is below a  $\lambda$ -branching factor of  $2.0001^1$  for more than 250 iterations it restarts the pheromone trails and sets them to the initial value of  $\tau_{max}$  as is done initially for maximum exploitation of the search space.

---

<sup>1</sup>Note that the  $\lambda$ -branching factor is normalised in this implementation so the resulting criteria for re-initialisation is in fact for less than a  $\lambda$ -branching factor of 1.0001.

### 3.3 Parallel Implementation

---

**Algorithm 8** Parallel implementation of MMAS for TSP

---

```
Parse Commandline
Initialise Data
Initialize pheromone trails and parameters
while Termination condition not met do
    UPDATE_COMM_DATA
    ConstructAntsSolutions
    ApplyLocalSearch
    COMM_EXCHANGE_PHASE
    UpdateStatistics
    UpdatePheromones
end while
COMM_TERMINATE_PHASE
COLLECT_SOLUTIONS_AND_RETURN_BEST
```

---

The parallel implementation considered here runs  $p$  colonies on each of  $p$  processors. Communication occurs at a rate input at the command line (Sec. 3.4). The termination for the algorithm is a time constraint of the run time of the algorithm main after initialisation procedures have completed. A pseudocode implementation of our parallel code is shown in Algorithm 8. The communication code is divided into two main stages: the *exchange phase* and the *termination phase*. Before the algorithm exits a third communication phase collects solutions, sorts and prints the best-overall solution. In the following Section 3.3.1 and Section 3.3.2 outline the communication schemes employed and the objects of information exchange for each topology considered. Section 3.3.3 considers solution gathering and sorting before the algorithm terminates.

#### 3.3.1 Directed Ring: exchange of best-so-far solution

Our first implementation is a Directed ring topology (Figure 3.1) which exchanges the best-so-far solution between colonies and considers a non-homogeneous colonial approach by varying the type of local search used by each colony. Each colony has a local-search procedure of either a *3-opt* and a *2.5-opt* local search. This designation is carried out equally (except in the case of an unequal number of processes) by the function *local\_search\_vary()* before the algorithm begins its main run.

The arrays *recv\_buffer*[ $n + 1$ ] and *send\_buffer*[ $n + 1$ ] are used to send and receive the cities visited by a colonies best-so-far solution. Both synchronous and asynchronous implementations of this topology are considered here. Appendix A includes the C with MPI libraries for the exchange and ter-

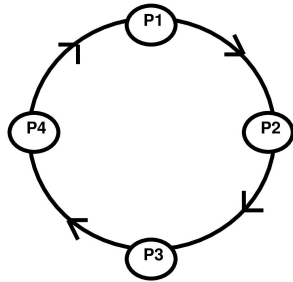


Figure 3.1: Directed Ring Topology for four processors .

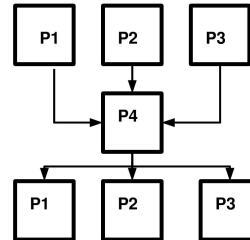


Figure 3.2: Fully Connected Topology for four Processors.

mination phase for both the synchronous and asynchronous implementations of this topology. The asynchronous scheme uses the non-blocking `MPI_Isend` and `MPI_Irecv` functions coupled with `MPI_Wait` blocking function and the `MPI_Test` non-blocking function. The functions `RING_AS_LS_EXCHANGE()` and `RING_AS_LS_TERMINATION()` perform the exchange and termination communication. The synchronous scheme uses the blocking `MPI_Recv` and `MPI_Send` functions for synchronous communication. The functions `RING_S_LS_EXCHANGE()` and `RING_S_LS_TERMINATION()` perform the exchange and termination communication for this topology.

Pheromone updating is only carried out on the best-so-far solution for this scheme and occasional algorithm restarts(pheromone re-initialisations) are not performed.

### 3.3.2 Fully Connected: exchange of tour length and update flag

Our second implementation considers a Fully Connected topology(Figure 3.2) in which one process is designated as the master (process zero here). It receives from all other processes its best solution tour length. The master process then sorts these lengths and sends to each process an update flag `recv_update` specifying whether a process should perform pheromone updating on the iteration-best or best-so-far solution. Appendix B includes the C and MPI code for the exchange and termination phase for both the

Fully connected Asynchronous scheme used. From the MPI libraries the non-blocking `MPI_Isend` and `MPI_Irecv` functions coupled with `MPI_Wait` blocking function and the `MPI_Test` non-blocking function. The functions `FC_AS_EXCHANGE()` and `FC_AS_TERMINATION()` perform the exchange and termination communication.

The sorting code used here is a simplistic bubble sort algorithm [20] `my_bubblesort()`. This simplistic sorting algorithm is highly inefficient but due to the the relative large run times is deemed not to infer any large computational overhead as a result of its use and is sufficient for the demonstrative purposes required here.

### 3.3.3 Solution collection and Sorting

A third communication phase occurs before the code terminates when each process sends to a designated master process its best solution. In order to compare performance across different instances we calculate the percentage from the optimum for each problem instance. The master process collects, sorts and then prints the percentage distance of the overall best solution from the optimum value of the particular TSP instance being solved. This is done by the function `collect_and_print_solution()`.

## 3.4 Code Layout and Command line Input

Here we outline the associated files and command line input to do with our code.

### Associated Files

The main control (main and communication code for parallel version) routines are in `acotsp.c`. Routines associated with input and output from the program and some statistical functions are contained in `InOut.c` and `In-Out.h`. Parsing code for the command line is contained in `parse.c` and `parse.h`. Procedures to do with the ant agents behaviour are contained in `ants.c` and `ants.h`. Routines associated with TSP are contained in `TSP.c` and `TSP.h`. Local search functions are in `ls.c` and `ls.h`. External variables and data structures associated with the parallel implementation are contained in `parallel.h`. Time measurement code and additional functions are contained in `timer.c`, `timer.h`, `utilities.c` and `utilities.h`.

Files associated with TSP instances (\*.tsp) used here (Sec. 4.2.1) are contained in the source code folders for both the serial and parallel versions. Refer to *TSPLIB* [22] for further problem instances.

### Command line Input

The parsing code here has been adapted from ACOTSP software resource [17]. The following flags are compulsory for a successful run of both the

Table 3.1: Command line inputs

|    |  |
|----|--|
| -t | algorithm run time   |
| -i | TSP instance input file (TSPLIB format necessary [13])     |
| -o | optimum solution for particular instance                   |
| -m | number of ants per colony                                  |
| -a | alpha (influence of pheromone trails)                      |
| -b | beta (influence of heuristic information)                  |
| -e | rho: pheromone trail evaporation                           |
| -l | local search: 0:no local search ; 3:3-opt                  |
| -u | update no. (Directed Ring Topology) <sup>2</sup>           |
| -x | apply MAX-MIN ant system(no value entered after this flag) |
| -j | communication frequency for parallel implementations       |

Table 3.2: Command line "-p" flag :Communication Topology

| Implementation                                | Acronym | "-p" flag value |
|---|---------|-----------------|
| Parallel Independent Runs                     | PIR     | 0               |
| Ring Asynchronous, Local Search               | RASLS   | 1               |
| Ring Synchronous, Local Search                | RSLS    | 2               |
| Fully Connected Asynchronous mixed schedule 1 | FCAS_1  | 3               |
| Fully Connected Asynchronous mixed schedule 2 | FCAS_2  | 4               |

serial and parallel versions of the code. Table 5.2 and Table 5.2 outline the input flags required in the command line. Table 5.2 show the communication flag values for the "-p" compulsory option. To illustrate command line use we give a sample command line for each of the serial and parallel run of the code.

### Serial Example Command line

```
./acotsp -t 900 -m 25 -a 1 -b 2 -e 0.2 -l 3 -x -o 259045 -i pr1002.tsp
-p 1 -j 1 -u 1
```

Serial run of MMAS for time=900 secs , with parameters  $\alpha = 1$ ,  $\beta = 2$ ,  $\rho = 0.2$  to solve the TSP instance "pr1002" with TSPLIB file pr1002.tsp with a 3-opt local search extension applied. Note the inclusion of the redundant parallel flags -p , -j and -u. These are required for a successful parsing of the command line but do not effect the serial parameters in any way.

### Parallel Example Command line

```
mpiexec ./acotsp -t 900 -m 25 -a 1 -b 2 -e 0.2 -l 3 -x -o 259045  
-i pr1002.tsp -p 3 -j 5 -u 1
```

Parallel run of MMAS for time=900 secs , with parameters  $\alpha = 1$ ,  $\beta = 2$ ,  $\rho = 0.2$  to solve the TSP instance "pr1002" with TSPLIB file pr1002.tsp. The communication topology is that of the Fully Connected asynchronous topology with schedule 1 (FCAS\_1) "-p 3" (See Table 5.2) with a communication rate of every 5Th iteration "-j 5". Note the inclusion of the redundant -u parallel only used for Directed communication schemes<sup>3</sup>. This is required for a successful parsing of the command line but does not effect the other parameters in any way. Also note for the Directed Ring Topology local search is varied as determined by "-u"<sup>4</sup>flag therefore making the "-l" local search flag redundant but compulsory for successful command line parsing.

## 3.5 Summary

We implement the MMAS to solve the TSP based on a time constraint termination condition. Here we outlined the important data structures used (Section 3.1 ) in both the serial and parallel implementations of our algorithm (Section 3.2, Section 3.3) along with information regarding the running of the code ( Section 3.4). In the following two chapters we describe our experimental procedure, results and data analysis.

---

<sup>4</sup>For the Directed Ring topology the percentage of ranks employing a local search can be varied using this flag. For  $p$  nodes, a number  $i$  where  $0 \leq i \leq p$  and  $i$  is an integer reflects the number of process employing a 2.5 opt local search. Ex.  $p = 8$  , for "-u 4" 50% of the processors or 4 nodes are employing a 2.5opt local search while the rest employ a 3-opt local search.

## Chapter 4

# Experimental Setup

### 4.1 IITAC cluster @ TCHPC

The code is tested and executed on the *IITAC* cluster facilitated by *Trinity Center for High Performance Computing* located at Trinity College Dublin. The facility consists of 712 AMD Opteron 2.4GHz processors with a theoretical peak performance of 3.4 TFlops run on 356 Dual AMD Opteron nodes using Voltaire Infiniband interconnection [21].

### 4.2 Experimental Procedure

We consider ten instances [13] of the TSP from *TSPLIB* [21] with termination criteria based on run time. These are shown in Table 4.2.1 below. We run our parallel algorithm variations for 8 processes on the cluster (Sec. 4.1) over 6 runs for each. For reference we run a serial implementation (**SERIAL**) of our code for the wall time of one process of our parallel algorithm. We also test our algorithm against a zero communication scheme for all instances, namely for Parallel Independent Runs (PIR) of the algorithm. Note that **SERIAL** and PIR implementations include use of occasional pheromone re-initialisations or restarts when stagnation behaviour occurs in the algorithm.

The parameters<sup>1</sup> of MMAS are chosen as recommended by Stutzle and Hoo's [7] for best performance. In order to compare performance across different instances we calculate the normalised percentage from the optimum for each problem instance. This is given in Equation 4.3.

$$\% \text{ Dist. from Optimum} = \frac{\text{Instance Solution} - \text{Optimum}}{\text{Optimum}} * 100. \quad (4.1)$$

In the following Section 4.2.1 considers the Directed Ring topology and Section 4.2.2 the fully connected topology implementation.

---

<sup>1</sup>With the exception of the mixed pheromone update schedule for the Fully Connected scheme as discussed in Chapter 2 and 3

Table 4.1: TSP instances with time in seconds,available from TSPLIB [21].

| TSPinstance | Run time(secs) |
|-------------|----------------|
| pr1002      | 900            |
| 1060        | 900            |
| pcb1173     | 900            |
| d1291       | 1200           |
| nrv1379     | 1200           |
| fl1577      | 1500           |
| vm1748      | 1500           |
| rl1889      | 1500           |
| d2103       | 1500           |
| pr2392      | 1500           |

Table 4.2: Ring Topology variations.

| Implementation(with % of processes with 2.5-opt local search) | Acronym |
|---|---------|
| Ring Asynchronous, Local Search:2.5-opt 50%                   | RASLS   |
| Ring Synchronous, Local Search:2.5-opt 50%                    | RSLS    |

#### 4.2.1 Directed Ring: exchange of best-so-far solution

For the Directed Ring communication topology we vary the local search across the processes between a 2.5-opt and a 3-opt local search. We investigate for this implementation different proportions of processes applying a 2.5-opt local search. We test this for 50% of the processes employing a 2.5-opt local search. It is also beneficial for an initial delay in communication to make the algorithm more exploratory in the initial stages. Here we have no communication in the first 100 iterations. We implement all these variation with a communication frequency of best-so-far solution exchange every 25 iterations. Table 4.2.1 shows the variations considered for this topology.

#### 4.2.2 Fully Connected: exchange of tour length and update flag

For the Fully connected topology we perform two different mixed pheromone update schedules across all processes for the initial stages of the algorithm run for our asynchronous implementation. Equation 4.2 and Equation 4.3 outline the two mixed pheromone update schedules used here implemented across all processes. They show the percentage of the worst performing processors used to update the iteration best solution for the first 250 and 500 iterations of the algorithm run.

Table 4.3: Fully Connected topology variations

| Implementation(with mixed update schedule)    | Acronym |
|---|---------|
| Fully Connected Asynchronous mixed schedule 1 | FCAS_1  |
| Fully Connected Asynchronous mixed schedule 2 | FCAS_2  |

$$s_{is}(\theta) = \begin{cases} 100\% & \text{if } 0 < \theta \leq 25, \\ 75\% & \text{if } 25 < \theta \leq 75, \\ 50\% & \text{if } 75 < \theta \leq 125, \\ 25\% & \text{if } 125 < \theta \leq 250, \\ 0\% & \text{if } \theta > 250 \end{cases} \quad (4.2)$$

$$s_{is}(\theta) = \begin{cases} 100\% & \text{if } 0 < \theta \leq 75, \\ 75\% & \text{if } 75 < \theta \leq 125, \\ 50\% & \text{if } 125 < \theta \leq 250, \\ 25\% & \text{if } 250 < \theta \leq 500, \\ 0\% & \text{if } \theta > 500 \end{cases} \quad (4.3)$$

Table 4.2.2 shows the variations considered for this topology. In the above schedules all processes not updating the iteration best solution are updating the best-so-far solution. After these schedules have been completed only best-so-far solutions are updated and no further communication takes place until termination. Due to the low communication overhead employed here ( only integers being exchanged) we have a high comm rate of every 5th iteration.

## Chapter 5

# Results and Analysis

Our hypothesis is that under the two communication topologies considered the solution quality is increased significantly for parallel implementation of MMAS. As discussed in Section 4.2 in order to have a measure independent of each TSP instance we calculate the normalised percentage from the optimum solution for each instance (see Equation 4.3). We use statistical techniques in order to evaluate if our implementations show a statistically significant increase in solution quality. Figure 5.1 shows the master box plot<sup>1</sup> of our results for all parallel models (see Tables 4.2.1 and 4.2.2) and includes those for our zero-communication scheme (PIR, Sec. 4.2) and also for our serial algorithm (SERIAL, Sec. 4.2). The box plot indicates that all variations except for RSLs (where the synchronisation of communication appears to largely reduce quality) improve solution quality with a lower median when compared with SERIAL. The fully connected asynchronous scheme with mixed update schedule 1 and 2 (FCAS\_1, FCAS\_2) are the best performing overall with FCAS\_1 displaying the smallest median over the sample space measured. In order to check whether the differences in performance between the parallel variations are statistically significant we perform non-parametric distribution tests using the *Wilcoxon rank sum test* and for multiple hypothesis adjust the *p-values* using *Holm's method*<sup>2</sup>[22].

### 5.1 Directed Ring Topology

The Directed Ring implementations with best-so-far solution exchange and local search variations namely RASLS and RSLs display significantly different behaviour with RASLS giving the best performance of the two (see Figure 5.2). This can be accounted for as the gains due to the reduced computational overhead of implementing the 2.5-opt local search on (50 % of the processes) appear to be reduced significantly by the synchronous communication implemented by RSLs requiring each colony to synchronise the communication of the best-so-far solution exchange. Synchronisation

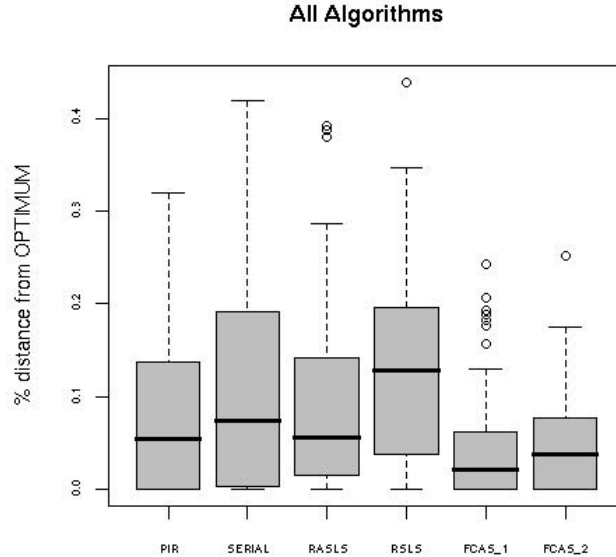


Figure 5.1: Box plot for normalised results for all algorithms

impends the 2.5-opt processes from benefiting from the increased solution construction time given by the reduced computational overhead from applying a simpler local search procedure to 50% of the processes. This results in a large decrease in solution quality and displays a median larger than SERIAL implementation indicating the degree to which synchronisation of communication impends the algorithm performance in this scheme.

RASLS displays a median approximately the same as PIR( Table 5.3). We test the single null hypothesis that RASLS is statistical no different from PIR for the normalised % difference of solution from optimum measurement. The significance level at which we reject the null hypothesis is 0.05. We find a  $p - value = 0.9564$  using the *Wilcoxon rank sum test* which implies the acceptance of the null hypothesis proposed. We conclude that there is no statistically significant advantage in choosing this scheme over PIR. Manfrin et al. [13] showed that a reduction in communication frequency for this scheme without the local search variation resulted in no statistical difference when compared with PIR. Thus further experimentation with reduced frequency of communication may prove to significantly increase performance for RASLS with respect to PIR. Analysis by Manfrin et al. also showed that the use of occasional pheromone re-initialisations or restarts when stagnation occurs as used in our PIR scheme may benefit this implementation further.

<sup>2</sup>All data analysis was performed using the R statistical package [23]

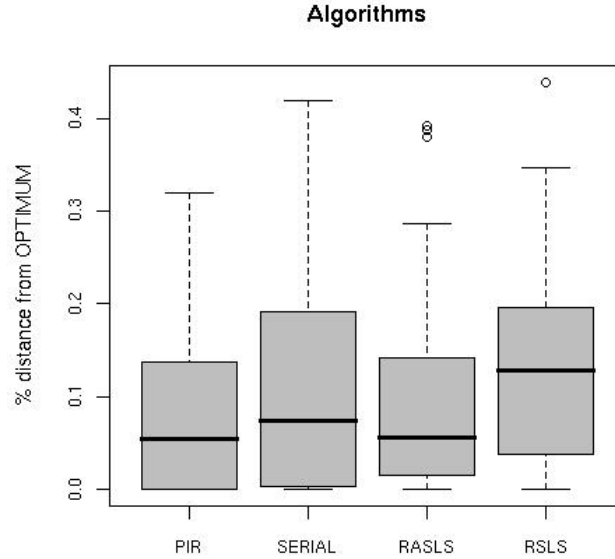


Figure 5.2: Box plot for normalised results for ring topology with solution exchange

## 5.2 Fully Connected Topology

The Fully Connected Asynchronous topology with exchange of tour lengths and update flags scheme is implemented with two update schedules. For our implementations namely **FCAS\_1** and **FCAS\_2** we see a significant increase in solution quality when compared with **PIR** (Figure 5.3). In order to test if these implementations are statistically different from all others considered we propose two null hypothesis. The first null hypothesis is that **FCAS\_1** is statistically no different from all other variations considered given the measured values of normalised % from the optimum. Table 5.2 shows the  $p$ -values associated with this hypothesis test using the *Wilcoxon rank sum test* and adjusting the  $p$ -values using *Holm's method*. We take a significance level of 0.05 and reject the null hypothesis that **FCAS\_1** is the same as all other instances except in the case of **FCAS\_2** in which the null hypothesis holds. Our second null hypothesis repeats the procedure for **FCAS\_2**. We propose the null hypothesis that **FCAS\_2** is statistically no different from all other variations considered given the measured values of normalised % from the optimum. Table 5.2 shows the  $p$ -values associated with this measurement. Again the null hypothesis can be rejected with significance level 0.05 except for the case of **FCAS\_1**. We thus conclude that either **FCAS\_1** or **FCAS\_2** significantly improve solution quality when compared to the other variations

considered. Also we conclude that no statistical difference appears between FCAS\_1 or FCAS\_2 for the schedules considered.

This performance improvement is in large due to the the increased exploration of solution space in the initial phase of MMAS run given by the FCAS communication framework. Results for serial implementations of MMAS [7] indicate that optimum convergence is more likely to occur by updating the best-so-far solution but has increased chances of the algorithm converging to local optima. By allowing the worst performing colonies to only update the iteration best solution and the best performing the best-so-far solution in the mixed schedule framework we increase both the exploration of solution space while also focusing the search for global optima by updating the best-so far solutions in the initial stages of the algorithm. The success of FCAS over zero communication schemes (PIR) highlights the importance of exploitation of the solution space in the initial stages of the MMAS run preventing acceleration of convergence towards same solutions.

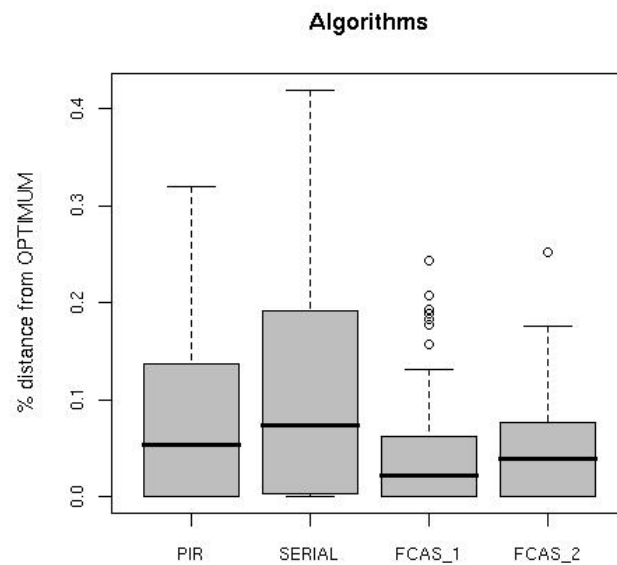


Figure 5.3: Box plot for normalised results for fully connected topology with no solution exchange

Table 5.1: P-values for Null Hypothesis: The performance of FCAS\_1 is not statistically different from all other implementations ( for normalised % from optimum measurement)

| Algorithm | p-value    |
|-----------|------------|
| PIR       | 5.2380e-06 |
| SERIAL    | 1.2932e-07 |
| FCAS_2    | 7.5580e-01 |
| RASLS     | 8.4550e-04 |
| RSLs      | 2.4315e-08 |

Table 5.2: P-values for Null Hypothesis: The performance of FCAS\_2 is not statistically different from all other implementations ( for normalised % from optimum measurement)

| Algorithm | p-value    |
|-----------|------------|
| PIR       | 1.2792e-05 |
| SERIAL    | 1.6076e-07 |
| FCAS_1    | 7.5580e-01 |
| RASLS     | 3.9620e-04 |
| RSLs      | 1.3660e-08 |

### 5.3 Code Validation

Both our serial and parallel implementation variations display solutions expected for the TSP for the run times considered. Table 5.3 shows the median values for all algorithms considered for the measurement of percentage normalised distance from the optimum. Comparatively they give average solutions within the range 0.04% to 0.12% of the normalised percentage distance from the optimum. We conclude that the code solves the *NP-Hard* TSP to within acceptable accuracy within reasonable run times and is comparable to previous investigations by Manfrin et al. [7] using similar TSP instances.

Table 5.3: Average of normalised % from optimum measurement for each algorithm

| Algorithm | median of normalised % Distance from Optimum for sample space |
|-----------|---|
| PIR       | 0.054 %   |
| SERIAL    | 0.073 %   |
| RASLS     | 0.056 %   |
| RSLs      | 0.121 %   |
| FCAS_1    | 0.021 %   |
| FCAS_2    | 0.038 %   |

## Chapter 6

# Conclusion

In this report we study the parallelization of the high performing Ant Colony Optimisation (ACO) Max-Min Ant System (MMAS) algorithm applied to solve the *NP* hard Traveling Salesman Problem (TSP) static combinatorial optimisation problem. The TSP has often proved as a benchmark problem to display performance for algorithms of this type without obscuring them in technicalities. Here we propose two communication strategies one with a high communication overhead and the other with substantially lower communication volume. We implement a coarse grained parallelization with one colony of ant agents per compute node. Our first strategy considers a larger communication overhead exchanging best solutions between colonies (processes) using a Directed Ring communication topology throughout the algorithm run. Our second strategy considers a lower communication overhead comparing best solution values between colonies in a meta-algorithmic framework imposed during the initial stages of the algorithm run. Both implementations use local search extensions to improve MMAS performance. Our Directed Ring Topology considers a non-homogeneous approach varying the type of local search and thus the computational overhead used by each colony.

All parallel implementations except for the Synchronous Ring Topology (RSLS) where synchronising the communication seems to be detrimental to the algorithm performance perform better on average than our serial implementation SERIAL. Our Asynchronous Ring Topology scheme with local search variation proved to statistically perform no better than our zero communication scheme (PIR). As indicated by Manfrin et al. [13] this implementation may benefit from reduced communication frequency and occasional pheromone re-initialisations.

The fully connected topology which implements a meta-algorithmic mixed update schedule in the initial stages of the algorithm run proves to be the best performing implementation (FCAS\_1, FCAS\_2). This scheme significantly increases solution quality when compared with the Asynchronous

Ring (RASLS) scheme and our zero communication scheme (PIR). Best performing serial implementations of MMAS consider an update schedule which generally allows the iteration best solution (ant) to be updated more frequently in the earlier stages of the search thus allowing more of the search space to be explored. As the algorithm progresses the best-so-far solution is updated more frequently until near convergence where only the best-so-far ant is updated. This implementation considers a mixed pheromone update schedule across all processes in which a designated master process collects and ranks processes by solution quality. The master then sends to each colony an update flag telling each colony to update either the iteration best or best so far solution. The worst performing colonies are allowed to update only the iteration best solution while the best performing only the best-so-far solution. The number of colonies updating either solution is controlled by the master processes using a mixed schedule similar to those implemented in serial versions of MMAS. We considered two mixed update schedules for this scheme, one which performs mixed updating for the first 250 iterations of the algorithm run and an extended version for the first 500 iterations. We could not find any statistical benefit for using one schedule over the other although the shorter scheme did display a smaller median over the sample space considered.

The initial phase for MMAS is crucial for determining the algorithms speed of convergence towards an optimum solution. It is clear that any parallel communication strategy must exploit this initial highly exploratory phase. The exchange of best solution strategy (RASLS and RSLs) used here demonstrates the detrimental effect of solution exchange pushing colonies convergence towards same solution and limiting the exploration of the solution space. Using a mixed update strategy across all processes proved successful in preventing early convergence and showed significant improvement against zero-communication implementations which were previously the best-performing parallel implementation of MMAS reported [13]. It is believed that further investigation of implementations of this nature are needed. Introducing greater heterogeneity among colonies may prove successful in extending the search space exploration. Also coupling of mixed update strategies with more sophisticated best solution exchange strategies may prove fruitful and merits further investigation.

# Bibliography

- [1] Goss, S., Aron, S., Deneubourg, J. L., and Pasteels, J. M. (1989). Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76, 579581.,
- [2] Dorigo(1992).Optimization, Learning and Natural Algorithms(Italian). Phd Thesis, Dipartimento di Elettronica, Politecnico di Milano,Milan.
- [3] Dorigo, M., Maniezzo, V., and Colorni, A. (1991a). *Positive feedback as a search strategy*. Technical report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan.
- [4] Colorni, A., Dorigo, M., and Maniezzo, V. (1992a). Distributed optimization by ant colonies. In F. J. Varela and P. Bourguine (Eds.), *Proceedings of the First European Conference on Artificial Life* (pp. 134142) .Cambridge, MA, MIT Press.
- [5] Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms*. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan.
- [6] Bullnheimer, B., Hartl, R. F., and Strauss, C. (1999c). *A new rank-based version of the Ant System: A computational study*. Central European Journal for Operations Research and Economics, 7(1), 2538.
- [7] Stutzle, T., and Hoos, H. H. (2000). *MAX-MIN Ant System*. Future Generation Computer Systems, 16(8),pg 889914.
- [8] Bullnheimer, B., Kotsis, G., and Strauss, C. (1998). Parallelization strategies for the Ant System. In R. D. Leone, A. Murli, P. Pardalos, and G. Toraldo (Eds.), *High Performance Algorithms and Software in Non-linear Optimization*, No. 24 in Kluwer Series of Applied Optimization (pp. 87100). Dordrecht, Netherlands, Kluwer Academic Publishers.
- [9] Middendorf, M., Reischle, F., and Schmeck, H. (2002). *Multi colony ant algorithms*. *Journal of Heuristics*,8(3), 305320.

- [10] Piriya Kumar, D.A.L., Levi, P.: *A new approach to exploiting parallelism in ant colony optimization*. In: International Symposium on Micromechatronics and Human Science (MHS) 2002, Nagoya, Japan. Proceedings, IEEE Standard Office (2002) 237243
- [11] Whitley D., S. Rana, R.B. Heckendorn. (1999). *The Island Model Genetic Algorithm: On Separability, Population Size and Convergence*. Journal of Computing and Information Technology- CIT 7,1,33-47.
- [12] Tanese, R.: *Parallel genetic algorithms for a hypercube*. In: Proceedings of the second international conference on Genetic Algorithms and their Applications, Hillsdale, NJ, Lawrence Erlbaum Associates, Inc. (1987) 177183
- [13] Manfrin M., Birattari, Stutzle, Dorigo M. (2006): *Parallel Ant Colony Optimization for the Travelling Salesman Problem*. ANTS 2006, Lecture Notes in Computer Science 4150 pg 224-234 2006. Springer-Verlag Berlin Heidelberg 2006.
- [14] Lourenco, H. R., Martin, O., and Stutzle, T. (2002). *Iterated local search*. In F. Glover and G. Kochenberger (Eds.), Handbook of Metaheuristics, vol. 57 of International Series in Operations Research and Management Science (pp. 321-353). Norwell, MA, Kluwer Academic Publishers.
- [15] Dorigo, M., Stutzle, T.: *Ant Colony Optimization*. MIT Press, Cambridge, MA (2004)
- [16] Bentley, J. L. (1992). Fast algorithms for geometric traveling salesman problems. ORSA Journal on Computing, 4(4), 387-411.
- [17] Thomas Stutzle. ACOTSP, Version 1.0 (GPL). Available from <http://www.aco-metaheuristic.org/aco-code>, 2004.
- [18] Stutzle, T. (1999). *Local Search Algorithms for Combinatorial Problems: Analysis, Improvements, and New Applications*, vol. 220 of DISKI. Sankt Augustin, Germany, Infix.
- [19] Dorigo, M., and Gambardella, L. M. (1997b) *Ant Colony System: A cooperative learning approach to the traveling salesman problem*. IEEE Transactions on Evolutionary Computation, 1(1), 53-66.
- [20] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 106-110 of section 5.2.2: Sorting by Exchanging.
- [21] Trinity Center for High Performance Computing <http://www.tchpc.tcd.ie>

- [22] Reinelt, G.: TSPLIB95 [http://www.iwr.uni-heidelberg.de/groups/comopt/-software/tsplib95/index.html](http://www.iwr.uni-heidelberg.de/groups/comopt/software/tsplib95/index.html) (2004)
- [22] P. Dalgaard ,*Introductory Statistics wiht R*, Springer-Verlag,New York (2002). ISBN 0-387-95475-9.
- [23] The R statistical package (GPL) <http://www.r-project.org/>

### **Acknowledgements**

I would like to thank my supervisor Dr. Geoff Bradley, HPC course co-ordinator Dr. Richard Timoney and Max Manfrin for their help and input throughout this project.

# Appendix A

## Communication code: Directed Ring

### A.1 Ring Topology:best-so-far solution exchange

#### A.1.1 Exchange Phase

##### Asynchronous

Listing A.1: Asynchronous Directed Ring best-so far solution Exchange

```
int RING_AS_EXCHANGE(void){
    next_rank = ((rank+1)%size);
    prev_rank = ((rank-1+size)%size);

    if (send_flag) { // Initially set = 0 , till 1st Send made
        MPI_Wait(&request1, &stat);
        // Waits for an MPI send or receive to complete
        for (i = 0; i < n+1; i++)
            send_buffer[i] = best_so_far_ant->tour[i];

//RANK SENDS TO NEXT RANK
MPI_Isend(send_buffer, n+1, MPI_LONG, next_rank, 111,
          MPI_COMM_WORLD, &request1);
        send_flag = 1;

//Note flag initially set = 1, till 1st test
if ( 0 == flag ) { // Receive waiting if flag = 0
    MPI_Test(&request1, &flag, &stat);
    // returns flag > 0 if receive completed
    if ( 0 != flag ) {
        //Receive has completed if Test returns flag non-zero
        if (-1 != recv_buffer[0]) {
            recv_tour_length = compute_tour_length(recv_buffer);
            if ((recv_tour_length < best_so_far_ant->tour_length)
                && (recv_tour_length <= ant[iteration_worst_ant].tour_length)) {
                for (i=0; i < n+1; i++)
                    ant[iteration_worst_ant].tour[i] = recv_buffer[i];

                ant[iteration_worst_ant].tour_length = recv_tour_length;
                update_statistics(); // update best-so-far ant
            }
        }
        else if ( (recv_buffer[0] == -1))
        {
// The message received is a TERMINATE ie. recv_buffer[0] = -1
            terminate_flag = 1;
        }
    }
}
// Recieves Completed when flag non-zero
while ( (0 != flag) && (0 == terminate_flag) ) {
```

```

//RANK RECIEVES FROM PREVIOUS RANK
MPI_Irecv(recv_buffer, n+1, MPLLONG, prev_rank, 111,
          MPLCOMM_WORLD, &request1);

MPI_Test(&request1, &flag, &stat);
if ( 0 != flag) { //Recieve Completed
    if ( -1 != recv_buffer[0] ) {
        recv_tour_length = compute_tour_length(recv_buffer);
if ((recv_tour_length < (best_so_far_ant->tour_length))
    && (recv_tour_length < ant[iteration_worst_ant].tour_length)) {
        for (i = 0; i < n+1; i++)
            ant[iteration_worst_ant].tour[i] = recv_buffer[i];
        ant[iteration_worst_ant].tour_length = recv_tour_length;
        update_statistics(); // update best-so-far ant
    }
    }
    else if ( (recv_buffer[0] == -1) ) {
        // The message received is a TERMINATE
        terminate_flag = 1;
    }
}
}
return(1);
}

```

## Synchronous

Listing A.2: Synchronous Directed Ring best-so far solution Exchange

```

int RING_S_EXCHANGE(void){
    next_rank = ((rank+1)%size);
    prev_rank = ((rank-1+size)%size);

    for (i = 0; i < n+1; i++)
        send_buffer[i] = best_so_far_ant->tour[i];

    //RANK SENDS TO NEXT RANK
    MPI_Send(send_buffer, n+1, MPLLONG, next_rank,
            10001, MPLCOMM_WORLD);

    //RANK RECIEVES FROM PREVIOUS RANK
    MPI_Recv(recv_buffer, n+1, MPLLONG, prev_rank, 10001,
            MPLCOMM_WORLD, &stat);

    if (-1 != recv_buffer[0]) { //coupled with else below
        recv_tour_length = compute_tour_length(recv_buffer);

if ((recv_tour_length < best_so_far_ant->tour_length)
    && (recv_tour_length <= ant[iteration_worst_ant].tour_length)) {
        for (i=0; i < n+1; i++)
            ant[iteration_worst_ant].tour[i] = recv_buffer[i];
        update_statistics();
    }
    }
    else { // The message received is a TERMINATE
        // Set the terminate flag to TRUE
        terminate_flag = 1;
    }
}
return(1);
}

```

## A.1.2 Termination Phase

### Asynchronous

Listing A.3: Asynchronous Directed Ring best-so far solution Exchange-Termination

```

int RING_AS_TERMINATION(void){
    next_rank = ((rank+1)%size);

```

```

    prev_rank = ((rank-1+size)%size);

    // fill buffer with terminate signal
    for ( i = 0; i < n+1; i++)
        send_buffer[i] = -1;

    //RANK SENDS TO NEXT RANK-TERMINATION CONDITION
    MPI_Isend(send_buffer , n+1, MPLLONG, next_rank ,
              111, MPLCOMM_WORLD, &request1);

    //while I am out on my own && I have pending receive
    while ((0 == terminate_flag) && (0 == flag)) {
        // RANK RECIEVES FROM NEXT RANK -TERMINATION CONDITION
        if ( 0 != flag ) { //Recieves Completed
            MPI_Irecv(recv_buffer , n+1, MPLLONG, prev_rank ,
                      111, MPLCOMM_WORLD, &request2);
        }
        MPI_Test(&request2 , &flag , &stat );

        if ( 0 != flag) {
            if ( -1 == recv_buffer[0] ) {
                terminate_flag = 0;
            }
        } // end of if( flag != 0) above
    } // end of while loop above

    return(1);
}

```

## Synchronous

Listing A.4: Synchronous Directed Ring best-so far solution Exchange-Termination

```

int RING.S.TERMINATION(void){
    next_rank = ((rank+1)%size);
    prev_rank = ((rank-1+size)%size);

    for ( i = 0; i < n+1; i++)
        send_buffer[i] = -1;
    // fill buffer with terminate signal

    //RANK SENDS TO NEXT RANK-TERMINATION CONDITION
    MPI_Send(send_buffer , n+1, MPLLONG, next_rank,10001, MPLCOMM_WORLD);

    // RANK RECIEVES FROM NEXT RANK -TERMINATION CONDITION
    if ( terminate_flag == 0 ){
        MPI_Recv(recv_buffer , n+1, MPLLONG, prev_rank,10001, MPLCOMM_WORLD, &stat);
    } // end of if(terminate_flag == 0)

    return(1);
}

```

## Appendix B

# Communication code: Fully Connected

### B.1 Fully Connected: exchange of best solution and update flag

#### B.1.1 Exchange Phase

##### Asynchronous

Listing B.1: Asynchronous Fully Connected

```
int FC_AS_UPDATE_EXCHANGE(void){
//BEGIN NON-MASTER//
    if(rank != master){

// send flag intially =0 till first send made by master
        if (send_flag) {
            MPI_Wait(&request1, &stat1);
        }

        send_length = best_so_far_ant->tour_length;
//RANK SENDS TO MASTER BEST TOUR LENGTH
        MPI_Isend(&send_length, 1, MPILONG, master, 111,
                MPLCOMM_WORLD, &request1);

        send_flag = 1;
        if ( 0 == flag ) {
            MPI_Test(&request2, &flag, &stat2);

            if ( 0 != flag) { // The receive has completed
                if (-1 == update_flag) {
                    //The message received is a TERMINATE
                    terminate_flag=1;
                }
            } //end of if(flag != 0) above
        } //end of if(flag == 0) above

        while ( (0 != flag) && (0 == terminate_flag) ){
// while loop: Make receive and if not complete
//after testing below (ie. flag == 0) break loop..

// RANK RECIEVES FROM MASTER RANK
            MPI_Irecv(&update_flag, 1, MPILONG, master, 222,
                    MPLCOMM_WORLD, &request2);

            MPI_Test(&request2, &flag, &stat2);

            if ( 0 != flag) { // Recieve completed

                if ( -1 == update_flag){
```

```

        // The message received is a TERMINATE
        terminate_flag = 1;
    }
} //end of if(flag!=0)
} // end of while loop above
} // end of if(rank != master)
//END NON-MASTER //

//MASTER CODE//
if((rank == master) && (terminate_flag == 0)) {
    recv_length[master]= best_so_far_ant->tour.length;
i=1;
while(i < size){
    if ( 0 == flag2 ) {
        MPI_Test(&request1, &flag2, &stat1);

        if ( 0 != flag2 ) {
            source = stat1.MPLSOURCE;
            tag = stat1.MPLTAG;
            if(recv_update != -1){

                recv_length[source]=recv_update;
            }
            else{
                terminate_flag = 1;
            }
        } //end of if(flag2 != 0) above
    } // end of if(flag2 == 0) above

    MPI_Irecv(&recv_update, 1, MPLLONG,i,111,
              MPLCOMM_WORLD, &request1);

    MPI_Test(&request1, &flag2, &stat1);

    if ( 0 != flag2 ) { /* recieve completed */
        source = stat1.MPLSOURCE;
        tag=stat1.MPLTAG;
        if(recv_update != -1){

            recv_length[source]=recv_update;
        }
        else{
            terminate_flag = 1;
        }
    } // end of if(flag2!=0) above
    i++;

} // end of while loop above

//sort recieved lengths //
for(p=0;p<size;p++)
    position[p]= p;
my_bubblesort();

i=1;
while(i<size){
    if (send_flag2) {
        MPI_Wait(&request2, &stat2);
    }
//find where current rank position is in sorted array
    for(j=0;j<size;j++){
        if((position[j] == i) ) placing=position[j];
        //set = 1 if restart criteria
        if(placing < update.no) send_update = 1;
        else send_update = 0;
    }
// Send update_flag to all processes not master
    MPI_Isend(&send_update, 1, MPLLONG, i,222,
              MPLCOMM_WORLD, &request2);

    MPI_Test(&request1, &flag2, &stat1);
    send_flag2=1;
    i++;

} // end of while loop above

for(j=0;j< size;j++){
    if( master == position[j] ) placing=position[j];
}
// find where current rank position
//is in sorted array for MASTER
if(placing < update.no ) update_flag = 1;
else update_flag = 0;
} // end of if( (rank == master)

```

```

//      && (terminate_flag == 0) ) above
//END OF MASTER CODE //
return(1);
}

```

## B.1.2 Termination Phase

### Asynchronous

Listing B.2: Asynchronous Fully Connected-Termination

```

int FC_AS_UPDATE_TERMINATION(void){
if(rank != master){
    send_length = -1;
        MPI_Isend(&send_length , 1, MPLLONG, master ,
                111, MPLCOMM_WORLD, &request1);

    while ( (0 == flag) ) {
        if ( 0 != flag ) {
// I don't have other pending Irecv,
//if do skip to MPI_test and receive code below
        MPI_Irecv(&recv_term , 1, MPLLONG, master ,
                222, MPLCOMM_WORLD, &request2);
}
        MPI_Test(&request2 , &flag , &stat2);

        if ( 0 != flag ) {
            if ( -1 == recv_term ) {
                terminate_flag = 1;
            }
        } // end of while loop above
    }
    MPI_Test(&request2 , &flag , &stat2);

    while ( ( terminate_flag == 0 ) && ( 0 == flag ) ) {
// while out on own && receives pending
        if ( 0 != flag ) {
            MPI_Irecv(&recv_term , 1, MPLLONG, master ,
                    222, MPLCOMM_WORLD, &request2);
        }
        MPI_Test(&request2 , &flag , &stat2);

        if ( 0 != flag ) {
            if ( -1 == recv_term ) {
                terminate_flag = 1;
            }
        } // end of while loop above
    } // end of if(rank != master) above
}
//Termination Phase for Master Process//
if((rank == master)){
    i=1;
    while ( i < size )
    {
MPI_Test(&request1 , &flag2 , &stat1);

    while ( ( terminate_flag == 0 ) && ( flag2 != 0 ) ){
        // If you have a pending Irecv

        if ( 0 != flag2 ) {
            MPI_Irecv(&recv_term , 1, MPLLONG,i,111, MPLCOMM_WORLD, &request1);
        }
        MPI_Test(&request1 , &flag2 , &stat1);

        if ( 0 != flag2 ) {
            source = stat1.MPLSOURCE;

            if ( -1 == recv_term ) {

```

```

        terminate_flag = 0;
    }
} // end of while loop above
i++;
}

    i=1;
    while( ( i < size) ){
        if (send_flag2) {
            MPI_Wait(&request2, &stat2);
            source=stat2.MPLSOURCE;
            tag=stat2.MPLTAG;
        }

        send_length = -1;

        MPI_Isend(&send_length, 1, MPLLONG,i,
                222, MPLCOMMWORLD, &request2);

        MPI_Test(&request1, &flag2, &stat1);
        i++;
    } // end of while loop above
} // end of if(rank == master) above

return(1);
}

```