Decentralized Parallel Ant Colony Optimization for Distributed Memory Systems

Huw Lloyd
School of Computing, Mathematics and Digital Technology
Manchester Metropolitan University
Manchester, United Kingdom
huw.lloyd@mmu.ac.uk

***Abstract*—Ant Colony System (ACS) is a well-established variant of the Ant Colony Optimization (ACO) nature inspired metaheuristic for solving combinatorial optimization problems. We present the DMACS (Distributed Memory Ant Colony System) algorithm, which is a parallelization of ACS for distributed memory architectures. The system is decentralized, with each processor running an identical agent process which administers a part of the pheromone matrix used to record the movements of simulated ants over a graph. We evaluate a Message Passing Interface (MPI) implementation of the algorithm on the well-known Travelling Salesman Problem (TSP), running on a distributed memory cluster. The results show that the algorithm scales at least as well as previous agent-based distributed implementations of ACS, without the need to sacrifice core features of the algorithm such as local search. However, our results also demonstrate that scaling the ACS algorithm to large numbers of processes in distributed memory architectures remains a significant challenge.**

***Index Terms*—Ant colony optimization; Parallel algorithms; Message passing; Scalability**

I. INTRODUCTION

Ant Colony Optimization (ACO) is a population-based metaheuristic method for approximate solution of a wide range of combinatorial optimization problems. It has been successfully applied to a number of $\mathcal{NP}$-hard problems, such as the Travelling Salesman Problem (TSP), Quadratic Assignment Problem and scheduling problems [1], and more recently to real world problems such as virtual machine placement in the cloud [2]. In ACO, a population of agents construct solutions on a graph, guided by a global *pheromone matrix* which encodes solutions found by previous ants. In the case of the TSP, each entry in the pheromone matrix corresponds to an edge in the complete graph; edges which form part of a good solution receive more pheromone and are more likely to be chosen by following ants.

The ACO metaheuristic includes a number of different algorithms, which combine standard ACO operators in different ways [3]. Two of the most sucessful are Max-Min Ant System (MMAS) [4] and Ant Colony System (ACS) [5]. In this paper, we focus on parallelizing the ACS algorithm. Two features of the ACS algorithm make it difficult to parallelize. Firstly, the default parameter settings for ACS stipulate a relatively small number of ants (of order 10) which makes fine-grained parallelism difficult. Secondly, ACS includes a *local pheromone update* operator, which requires that ants have concurrent write access to the pheromone matrix while constructing their tours. Here, we address the first issue by proposing a method to change the parameter settings of ACS with the aim of maintaining perfomance while increasing the number of ants, and the second issue by distributing the pheromone matrix across multiple nodes in a distributed system.

The contributions of this paper are; 1) a novel parallelization of the ACS algorithm which includes all features of the original algorithm, notably local search which has to date been omitted from many parallel implementations, and 2) a method for modifying the ACS parameters in order to maintain the balance of local and global pheromone updates when scaling the solution by increasing the number of ants to maintain fine-grained parallelism.

II. RELATED WORK

The ACO metaheuristic offers up an obvious approach to parallelization; the algorithm uses agents which can act independently of each other in constructing solutions, so it would seem natural to decompose the calculation by assigning one ant to each process. This *task parallel* approach, often in the form of a *master-slave* system, has formed the basis of many parallel ACO implementations [6]. However, there are a number of potential issues with this approach. Firstly, there must be a large enough population of ants to provide a sufficiently fine-grained parallel decomposition; for Graphics Processing Units (GPUs), with many thousands of threads, this is not the case and a data-parallel approach is preferred [7]–[10]. In the data parallel approach, the task of selecting the next city is divided between the threads using various vectorized versions of the classical roulette wheel algorithm. Secondly, the ACS variant under consideration here includes a *local pheromone update* operator which requires ants to continually modify the pheromone matrix as they construct their tours. This means that ants can no longer complete the task of constructing a full tour of the graph independently, and synchronization is required in accessing the pheromone matrix. One approach to this is to ignore the synchronization, as in the GPU implementation of ACS presented by [10]. In this case, errors may occur in the pheromone matrix, but experiments show that in most cases there is not a large effect on solution quality (although the effect is clear in some cases).

Distributed parallel implementations of ACO have a long history; the earliest approaches used multiple colonies [11] or independent runs [12] across processes. This latter, "embarrassingly parallel" approach simply spreads the task of an ensemble of runs with different seeds across a distributed architecture. In the multiple colony approach, independent ant colonies are run in parallel, with various schemes for sharing solutions or pheromone information. A survey of parallel approaches to ACO up to 2011 can be found in [6]; of the fourteen proposals based on ACS surveyed in that paper, two used parallel independent runs , five multiple colonies , six master-slave and one a hybrid method .

More recently, agent-based approaches to ACO have been evaluated using middleware solutions for agent-based programming [13]–[16]. In [13]'s implementation of the *Ant Colony System* (ACS) algorithm, each agent is responsible for a part of the computational domain, and ants are passed between agents as they navigate the graph. This algorithm is asynchronous, so that ants do not finish constructing their tours at the same time. This approach allows for local pheromone update, since each agent is responsible for its own part of the pheromone matrix, but the asynchronous nature of the calculation means that an important feature of the ACS algorithm, local search, is not implemented. [16] present another agent-based implementation of ACS, on the Siebog platform. Results are presented for small instances of the TSP (up to 150 cities) which demonstrate that representing ants as messages, rather than as agents, gives the best performance.

A decentralized, distributed ACO algorithm for TSP with an adaptive fuzzy parameter controller is presented in [17]. Although the algorithm is decentralized, one process is designated a 'master' process and is responsible for coordination tasks, but not tour construction. Results are presented for small instances of the TSP (up to 299 vertices) on up to 32 processes. Results are given for solution quality but not timing, making it difficult to judge the efficiency or scalability of this approach.

In this work, we present a novel parallelization of the ACS algorithm which retains all the original features of the algorithm, including local pheromone update and local search. The solution shares some features with agent-based algorithms, such as the distribution of the pheromone matrix between agent processes, and the passing of ant movements between agents; however our system is synchronous and the distributed processes carry out tour construction in step.

III. ALGORITHM

A. Ant Colony System (ACS)

The ACS algorithm [5] is one of the most effective variants of ACO for the TSP. The algorithm differs from earlier ACO algorithms in the use of the *random proportional rule* for selecting vertices during tour construction, the introduction of a *local pheromone update* operator, and the replacement of pheromone evaporation with a novel pheromone update operator which updates only the edges which form part of the best-so-far tour.

The algorithm starts by initializing the *pheromone matrix*, a square matrix ($n \times n$, where n is the number of vertices in the problem instance) to a constant value $\tau_0 = 1/C^{nn}$, where C^{nn} is the length of a tour found using the nearest-neighbour heuristic. The elements τ_{ij} of the pheromone matrix represent the amount of pheromone associated with the edges connecting vertices i and j.

The algorithm then proceeds in iterations. During each iteration, each of the ants in the system constructs a tour of the graph. This is carried out in parallel; that is, the system repeatedly iterates over the ants, performing one step of the tour construction for each ant. This is important for the functioning of the *local pheromone update* mechanism, in which the pheromone is reduced on edges as they are traversed, to discourage following ants from taking the same path, and hence improve diversity in the solution set.

At the start of the tour construction phase, each of the m ants is placed on a random city, and then constructs a tour of the graph by making random choices of vertices to visit next. These random choices are weighted, using weights derived from a combination of pheromone values and heuristic information (the edge costs). The weight assigned to the edge connecting vertex i to vertex j is

$$w_{ij} = \tau_{ij}^{\alpha}\eta_{ij}^{\beta} \tag{1}$$

where the *heuristic* $\eta_{ij} = 1/d_{ij}$ where d_{ij} is the length of edge ij and τ_{ij} is the pheromone value associated with the edge. The constants α and β therefore control the relative importance of pheromone and heuristic information in constructing the edge weights. For an ant placed on vertex i, the next vertex to visit is chosen using the *random proportional rule*. With probability $q_0 \in [0, 1]$, the ant is moved to the vertex connected to i by the highest weighted edge, subject to the constraint that the vertex is not already in the tour. That is, after generating a random deviate $q \in [0, 1]$, the selected vertex i_{sel} is

$$i_{\text{sel}} = \operatorname*{argmax}_{j \in N_i^k} w_{ij} \;\text{ if }\; q < q_0 \tag{2}$$

where N_i^k is the *feasible region* for ant k on vertex i (i.e., the set of vertices not already visited by ant k in its current tour). If $q \geq q_0$, the next vertex is selected using the *roulette wheel* method, with the probability of selecting vertex j given by

$$p_{i,j}^{k} = \begin{cases} w_{ij} / \sum_{i \in N_i^k} w_{ij} & i \in N_i^k \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

After each step in the tour construction process, the *local pheromone* operator is applied. For the edge just traversed, the pheromone is modified according to

$$\tau_{ij} \leftarrow \tau_{ij}(1 - \xi) + \tau_0 \xi \tag{4}$$

where $\xi \in [0, 1]$ is a constant parameter.

Finally, when all ants have completed their tours, the best tour found so far is used to update the pheromone matrix using the global pheromone operator:

$$\tau_{ij} \leftarrow \tau_{ij}(1 - \rho) + \Delta\tau_{ij}^{bs}\rho \forall (i, j) \in T^{bs} \tag{5}$$

where T^{bs} is the best-so-far tour with length C^{bs}, and $\Delta\tau_{ij}^{bs} = 1/C^{bs}$.

1) Nearest-neighbour lists: We use a standard extension to the basic ACO algorithm in which each vertex selection is first made by considering vertices in the *nearest neighbour list* of m_{nn} vertices of the current vertex. If all of the vertices in the nearest-neighbour list have already been visited, the highest weighted vertex is chosen from all the other unvisited vertices.

B. Distributed Memory Ant Colony System (DMACS)

The DMACS system operates in *Single Program, Multiple Data* (SPMD) mode; that is, each process runs an identical program. We call this program the *DMACS Agent*. The system comprises N processes in a distributed system, each running a DMACS Agent, and we enforce the constraint that the number of ants, m, evenly divides N. We assume that the number of vertices $n > N$. Apart from input/output operations (which are carried out by a single process), no process has a special status or role in coordinating the others. After some initial setup, during which one agent is responsible for loading the problem instance and broadcasting it to the others, the N agents execute the same program.

1) Overview: Each agent is responsible for a subset of the n vertices of the TSP instance; that is, it stores the rows of the pheromone matrix corresponding to edges which start at vertices in its vertex set, and is responsible for determining any ant movements which start at vertices in this set. During tour construction, each agent knows the position of all ants in the system and their partial tours. At the start of each step, an agent will determine the next moves of all ants in its own domain. At the end of each step, information on ant movements is shared between all agents, and each agent applies the local pheromone update operator on any of the new tour edges which lie in its domain. The process is repeated until all tours are complete. Each agent then applies a local search operator to an equal share of the ants' tours, after which the agents communicate collectively to determine if there is a new best solution. Finally, all agents apply the global pheromone operator to their parts of the pheromone matrix. The process is run for a fixed number of iterations.

2) Initialization: Once all agents have a copy of the problem instance (for a Euclidean TSP, this is the set of x, y coordinates for the n vertices of the instance) each agent calculates the range of vertices for which it is responsible. Since N does not necessarily evenly divide n, we distribute the remainder $r = n \mod N$ over the first r processes. The inclusive range of vertices $[j_0, j_1]$, where vertices are labelled $j \in [0 \ldots n-1]$, administered by process $i \in [0 \ldots N-1]$ is therefore

$$[j_0, j_1] = \begin{cases} [i\lfloor n/N \rfloor + i, (i+1)\lfloor n/N \rfloor + i] & i < r \\ [i\lfloor n/N \rfloor + r, (i+1)\lfloor n/N \rfloor + r - 1] & \text{otherwise} \end{cases} \quad (6)$$

Each agent process allocates memory for rows j_0 to j_1 of the pheromone matrix, initializing the pheromone values to τ_0.

3) Tour Construction: The ants are initially placed on random vertices of the graph. Each agent randomly allocates m/N ants to starting cities, and the positions of all ants are then distributed among the processes. This is achieved as follows: each process allocates an array of integers of length m to represent the ant positions, and initializes each element to -1. The vertices for any ants placed by an agent are written into the array, and the positions are shared globally by performing a reduction (with operator max) on this array using the `MPI_Allreduce` function. Each agent then has a complete list of the positions of all ants.

Tour construction proceeds a step at a time, with each agent iterating over all ants. If an ant is currently placed in an agent's vertex range, the pheromone matrix is used to select the next vertex in its tour. Edge lengths are calculated directly from the (x, y) coordinates of the TSP instance vertices. The new positions of all ants are then shared using the same reduction mechanism as the initial placement. Each agent then updates the ants' tours, and applies the local pheromone update to any edges that lie in its part of the pheromone matrix.

The tour construction process is represented schematically in Figure 1. In this example, an instance of 51 vertices, such as the TSPLIB [18] instance `eil51`, is solved by four agents using 8 ants. The instance is shown in Figure 2; the vertex labels correspond to those in Figure 1, and the optimum tour is shown as a line connecting the vertices. The current vertices of the ants are stored in the array `curVerts`. Here, Agent 0 is responsible for moving two of the ants (on vertices 3 and 8), Agent 1 has four ants (vertices 19, 14, 24 and 22) and so on. The agents determine new vertices for their ants based on the ACS process, and populate the array `nextVerts` with these values. Values for ants outside an agent's domain remain at -1. After the reduction process, all agents have the positions of all ants, and the process repeats. Note that since `MPI_Allreduce` includes an implicit barrier, the tour construction is synchronous.

4) Local Search and Pheromone Update: After tour construction, each agent performs local search on a number of ants' tours. Since the number of agents evenly divides the number of ants, agent i is responsible for running local search on the tours found by ants im/N to $(i+1)m/N-1$ inclusive. We use the 3-opt local search function from the ACOTSP code [19]. After local search, the final tour lengths are shared using a similar reduction process to that used in the tour construction; each agent prepares a vector of tour lengths initially filled with zeros, sets the tour lengths of the ants for which it performed the local search, then reduces this array using `MPI_Allreduce` with the max operator. All agents can then determine if there is a new best tour, and on which agent it resides. If there is a new best, this tour is communicated to all agents using a broadcast. Pheromone update in ACS is carried out using the global best tour only. Each agent updates the elements in its own part of the pheromone matrix using equation 5.

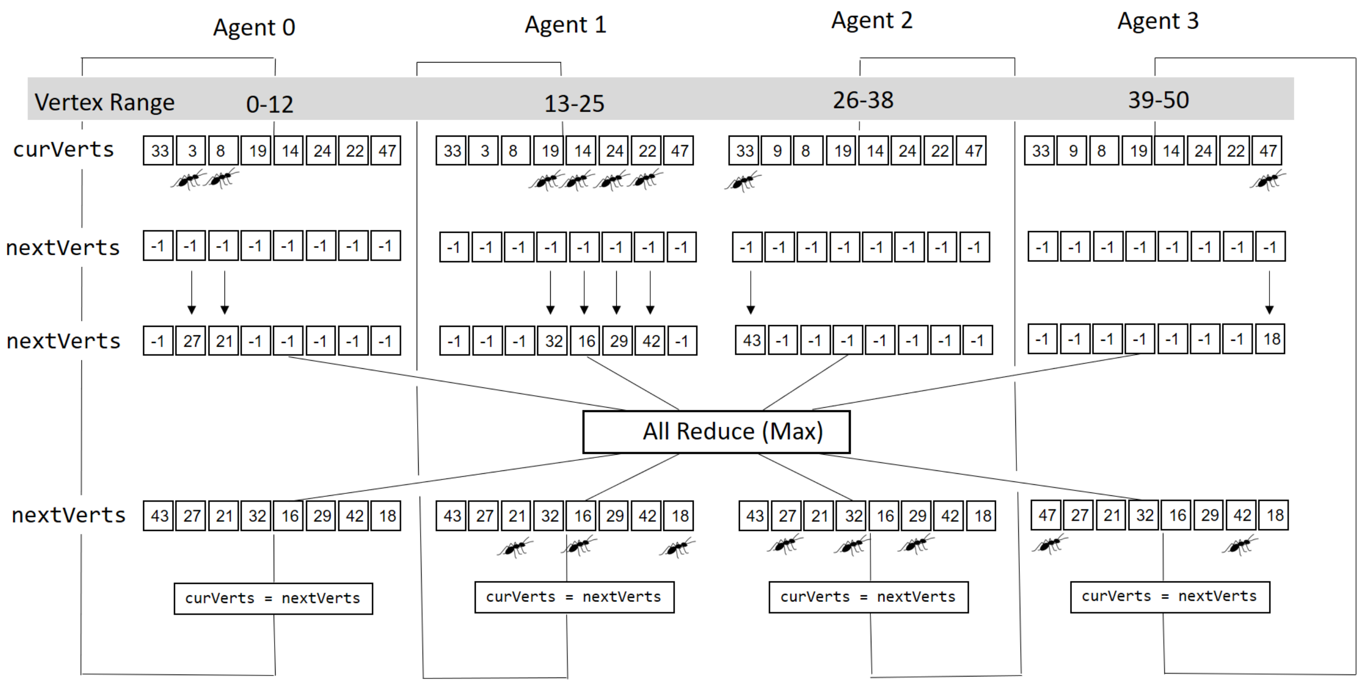

Fig. 1. Schematic representation of the distributed tour construction process in DMACS. In this case, four agents are solving an instance with 51 vertices (for example, the TSPLIB instance `eil51`), with eight ants.

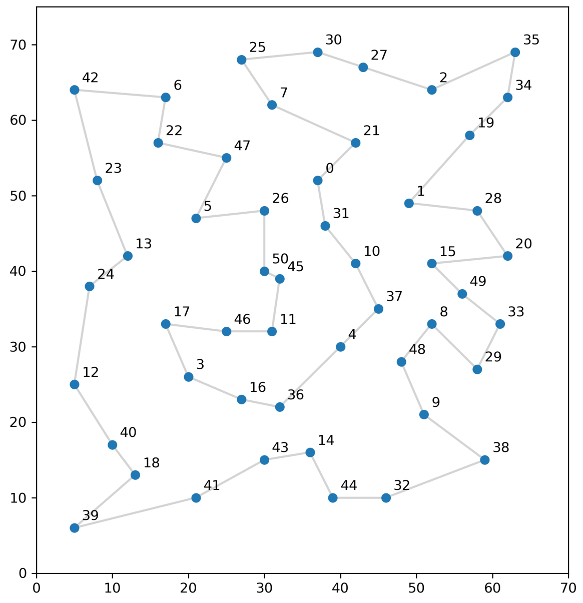

Fig. 2. The TSPLIB instance `eil51`, with the optimum tour shown as a line connecting the vertices.

IV. EVALUATION

We have implemented the DMACS algorithm in C++ using the Message Passing Interface (MPI) library to handle the communication between agents. We conducted experiments with the code on a distributed memory cluster, running on up to 32 cores. The cluster nodes are equipped with Intel Xeon ES2650 processors running at 2.6GHz, and nodes are connected with QDR Infiniband. We evaluated the code on standard instances from the TSPLIB [18] library, with vertex counts n between 1002 and 11849. The code was run with numbers of MPI ranks from 1 to 32, while maintaining the total number of tour evaluations as a constant. To achieve this, we scaled the number of ants with the number of processes while reducing the total number of algorithm iterations by the same factor.

A. Algorithm Parameters

We adopt standard values for the ACS parameters ($\alpha = 1$, $\beta = 2$, $q_0 = 0.9$, $\rho = 0.1$, $\xi = 0.1$, $m_{\text{nn}} = 20$), however we modify ρ and ξ to compensate for the varying number of ants. We do this because with a larger number of ants, the effect of local pheromone update is increased relative to global update (since with more ants there are fewer global updates per tour evaluation). In order for the effect of local pheromone update per iteration to remain constant when increasing the number of ants by a factor N, it is straightforward to show that the value of ξ should be replaced by ξ_N, given by

$$\xi_N = 1 - (1 - \xi)^{1/N} \tag{7}$$

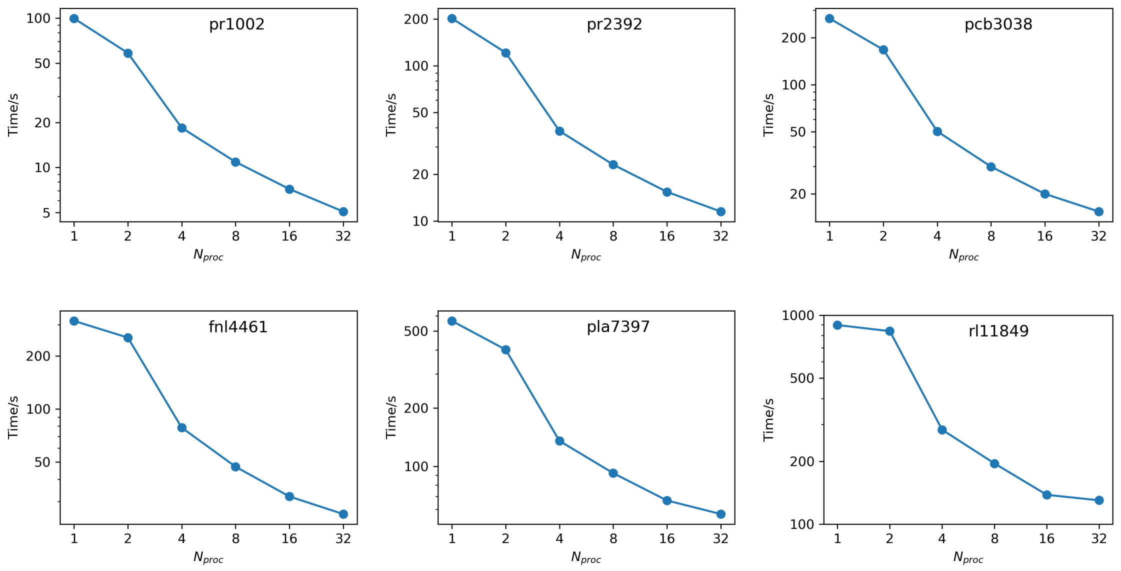

Fig. 3. Plots of execution time versus number of MPI ranks for the parallel runs. Data points represent the mean value from 10 runs per instance for each configuration.

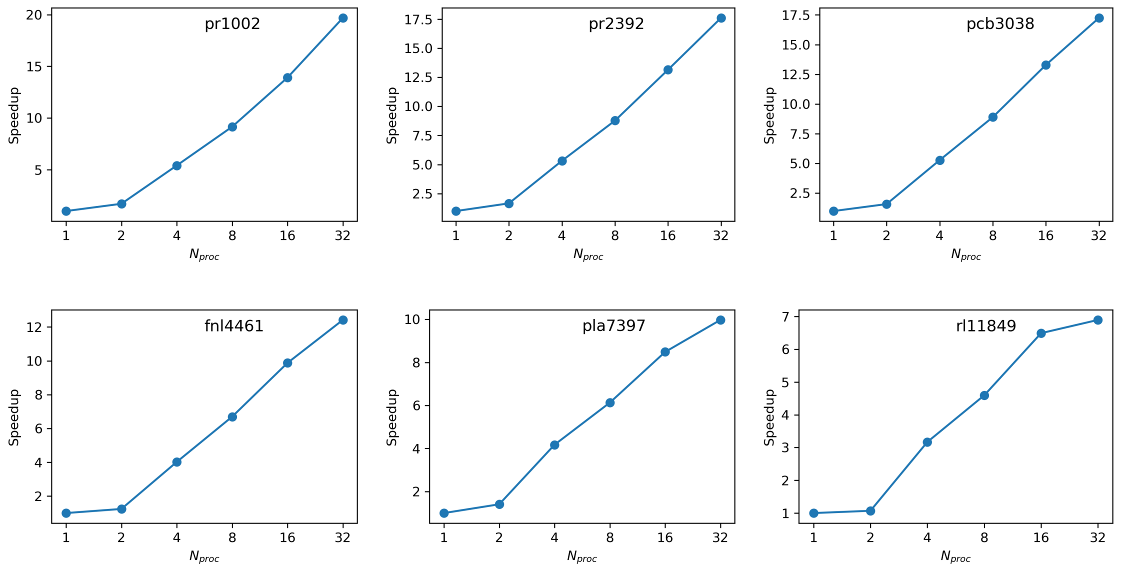

Fig. 4. Plots of speedup versus number of ranks for the parallel runs. Data points represent the mean value from 10 runs per instance for each configuration.

Alternatively, we could keep ξ constant, and modify ρ to account for the reduced number of global pheromone updates per tour evaluation according to

$$\rho_N = 1 - (1 - \rho)^N. \tag{8}$$

A third alternative, which we use in the experiments which follow, is to change both ξ and ρ by amounts which maintain the balance between local and global pheromone update. We choose to apply the correction for a factor $\sqrt{N}$ to both parameters, which is equivalent to the correction for a factor N in either parameter, and which maintains the balance between the local and global updates while minimizing the change in either parameter from their default values. The modified values of ξ and ρ are given by

$$\xi' = 1 - (1 - \xi)^{1/\sqrt{N}}; \quad \rho' = 1 - (1 - \rho)^{\sqrt{N}} \tag{9}$$

B. Experimental Setup

Apart from the runs on a single core, we ensured that all runs were split over at least two nodes of the cluster, so that network communication is necessary; running MPI in a effective shared memory environment where all the ranks are on the same physical machine would give an unrealistically low

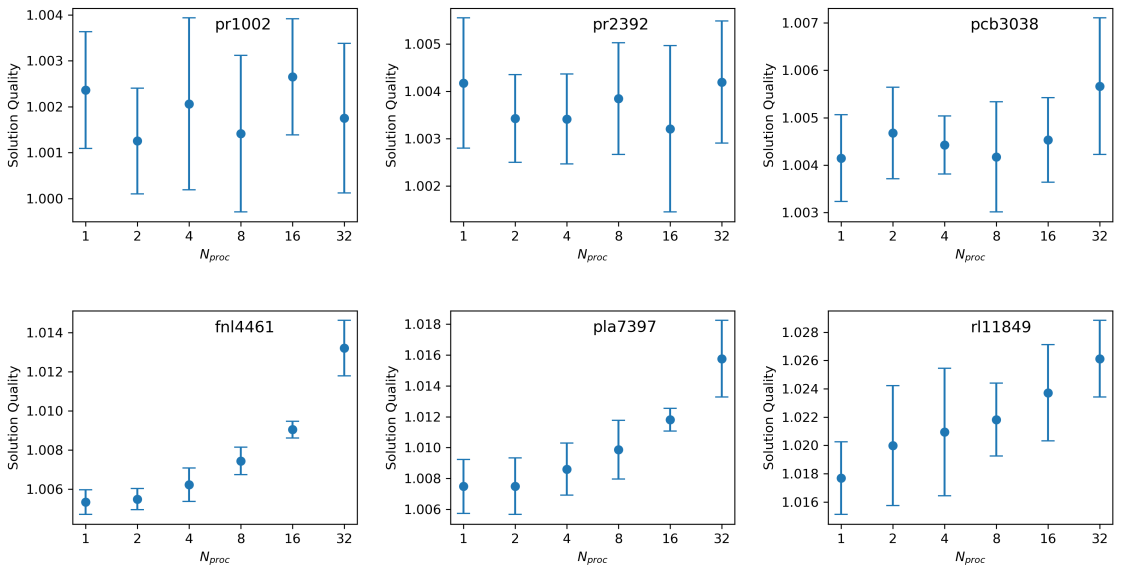

Fig. 5. Plots of solution quality versus number of MPI ranks for the parallel runs. Data points represent the mean value from 10 runs per instance for each configuration; error bars show the standard deviation. The solution quality is defined as the ratio of the length of the best tour found to the known optimum for the instance.

TABLE I
CONFIGURATION OF RANKS (N_{proc}), NODES (N_{node}) , PROCESSORS PER NODE (N_{ppn}), AND ACS PARAMETERS FOR THE PARALLEL RUNS. N_{iter} IS THE NUMBER OF ACS ITERATIONS.

N_{proc}	N_{node}	N_{ppn}	ξ	ρ	m	N_{iter}
1	1	1	0.1	0.1	8	4096
2	2	1	0.072	0.138	16	2048
4	2	2	0.051	0.19	32	1024
8	2	4	0.037	0.258	64	512
16	4	4	0.026	0.344	128	256
32	4	8	0.018	0.449	256	128

communication time. The configurations used for the runs on 1, 2, 4, 8, 16 and 32 ranks are given in Table I, along with the values of the ACS parameters. We used six instances from the TSPLIB library of TSP instances [18], `pr1002, pr2392, pcb3038, fnl4461, pla7397` and `rl11849`, and carried out ten runs on each instance for each number of ranks.

C. Results

Figures 3, 4 and 5 show the mean total runtime, mean speedup and mean solution quality for each instance over the 10 runs per process count. Error bars represent the standard deviation. We see that in all cases, adding more processes leads to a decrease in execution time, although the gains are less for the larger instances. For the smaller instances, `pr1002`, `pr2392` and `pcb3038` there is a speedup factor of approximately 17-20 between 1 and 32 processes, whereas for the larger instance this speedup ranges from around 7 to 12, with the lowest value in the case of `rl11849`, the largest instance. For all instances, the speedup between 1 and 2 processes is small compared to the speedup from 2 to 4. This is to be expected since the single-rank runs are in shared memory, with effectively zero communication overhead.

Solution quality is roughly constant for the three smaller instances, but in the larger instances we see a steady decline in quality of solution as more processes are added, although in the worst case this is a change of only approximately 1%. This is maybe due to the restricted number of iterations in these instances; even though the number of tour evaluations is constant, the pheromone feedback mechanism has less iterations in which to establish solutions, despite the increased value of ρ. Overall the system generates very high quality solutions (in many cases hitting the optimum value. The inclusion of 3-opt local search in the algorithm is essential in producing nearly optimal solutions in these relatively large TSP instances.

D. Comparison with Agent-Based Approaches

The results compare favourably with those found by [13]–[15]. Firstly, we note that DMACS shows an increasing speedup with number of ranks up to 32, whereas in [13], the maximum speedup is found using 11 processes with increasing execution time for larger numbers of processes. In [14], the maximum speedup is with 15 processes, with a slight slowdown in moving to 20 processes. The only instance in common with the present work is `pr1002`, for which results are given in [14] and [15]. The maximum speedup found for this instance by [15] is 14.9, on 15 processes, which suggests that their approach scales very well for small number of processes, but not beyond 10-15 processes. The best average solution qualities for `pr1002` found by [14] and [15] are 1.17 and 1.19 respectively, compared to 1.001 in this work. This discrepency is due to the inclusion of local search in our solution, which is omitted in the agent-based systems due to the asynchronous nature of the tour construction in those approaches.

V. Conclusion

In this paper, we describe a decentralized, distributed version of the Ant Colony System (ACS) algorithm which retains all the distinctive features of the original algorithm including local pheromone update and local search. We present experiments with an MPI implementation of the algorithm which shows that the computation scales well on up to 32 processes, although on large instances there is some loss of solution quality. We scale the computation by increasing the size of the ant population, while maintaining a constant number of tour evaluations. In order to enable the increase in population size, we present a simple mechanism for modifying the algorithm parameters which maintains the balance between the local and global pheromone update processes. Scaling the computation to larger numbers of processes remains a challenge since, to keep the size of the problem constant by reducing the number of iterations while increasing the number of ants, too few iterations will be carried out to converge to a solution.

Scaling the algorithm to larger numbers of processes may be possible by including some ideas from earlier attempts at parallel ACO, such as multiple colonies. This may enable the solution of very large instances of the TSP, such as the *Art TSP* instances [20], with vertex counts of order 10^5. The distribution of the pheromone matrix over multiple nodes is also a benefit here, since the memory requirement for this matrix scales with n^2.

References

[1] T. Stützle, M. López-Ibáñez, and M. Dorigo, *A Concise Overview of Applications of Ant Colony Optimization*. John Wiley & Sons, Inc., 2010. [Online]. Available: http://dx.doi.org/10.1002/9780470400531.eorms0001

[2] X. Liu, Z. Zhan, J. D. Deng, Y. Li, T. Gu, and J. Zhang, "An energy efficient ant colony system for virtual machine placement in cloud computing," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 1, pp. 113–128, Feb 2018.

[3] M. López-Ibáñez, T. Stützle, and M. Dorigo, *Ant Colony Optimization: A Component-Wise Overview*. Cham: Springer International Publishing, 2016, pp. 1–37. [Online]. Available: https://doi.org/10.1007/978-3-319-07153-4_21-1

[4] T. Stützle and H. Hoos, "MAX-MIN ant system and local search for the Traveling Salesman Problem," in *Evolutionary Computation, 1997., IEEE International Conference on*, Apr 1997, pp. 309–314.

[5] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the Traveling Salesman Problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, Apr 1997.

[6] M. Pedemonte, S. Nesmachnow, and H. Cancela, "A survey on parallel ant colony optimization," *Appl. Soft Comput.*, vol. 11, no. 8, pp. 5181–5197, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.asoc.2011.05.042

[7] J. M. Cecilia, A. Nisbet, M. Amos, J. M. García, and M. Ujaldón, "Enhancing GPU parallelism in nature-inspired algorithms," *The Journal of Supercomputing*, vol. 63, no. 3, pp. 773–789, 2013.

[8] J. M. Cecilia, A. Llanes, J. L. Abelln, J. Gmez-Luna, L.-W. Chang, and W.-M. W. Hwu, "High-throughput ant colony optimization on graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 113, pp. 261 – 274, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517303337

[9] L. Dawson and I. A. Stewart, "Improving Ant Colony Optimization performance on the GPU using CUDA," in *2013 IEEE Congress on Evolutionary Computation*, June 2013, pp. 1901–1908.

[10] R. Skinderowicz, "The GPU-based parallel Ant Colony System," *Journal of Parallel and Distributed Computing*, vol. 98, no. Supplement C, pp. 48 – 60, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731516300284

[11] M. Middendorf, F. Reischle, and H. Schmeck, "Multi colony ant algorithms," *Journal of Heuristics*, vol. 8, no. 3, pp. 305–320, May 2002. [Online]. Available: https://doi.org/10.1023/A:1015057701750

[12] T. Stützle, "Parallelization strategies for Ant Colony Optimization," in *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*. London, UK: Springer-Verlag, 1998, pp. 722–731.

[13] S. Ilie and C. Bdic, "A comparison of the island and acoda approaches for distributing aco," in *2013 17th International Conference on System Theory, Control and Computing (ICSTCC)*, Oct 2013, pp. 757–762.

[14] S. Ilie and C. Bdic, "Multi-agent approach to distributed ant colony optimization," *Science of Computer Programming*, vol. 78, no. 6, pp. 762 – 774, 2013, special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011) & Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642311001717

[15] ——, "Multi-agent distributed framework for swarm intelligence," *Procedia Computer Science*, vol. 18, pp. 611 – 620, 2013, 2013 International Conference on Computational Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050913003682

[16] A. Kaplar, M. Vidakovi, N. Luburi, and M. Ivanovi, "Improving a distributed agent-based ant colony optimization for solving traveling salesman problem," in *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2017, pp. 1144–1148.

[17] J. Collings and E. Kim, "A distributed and decentralized approach for ant colony optimization with fuzzy parameter adaptation in traveling salesman problem," in *2014 IEEE Symposium on Swarm Intelligence*, Dec 2014, pp. 1–9.

[18] G. Reinelt, "TSPLIB - a Traveling Salesman Problem library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.

[19] T. Stützle, "ACOTSP," Available at http://iridia.ulb.ac.be/~mdorigo/ACO/downloads/ACOTSP-1.03.tgz (2005/06/12).

[20] "TSP Art Instances," http://www.math.uwaterloo.ca/tsp/data/art/, accessed: 2019-03-29.