# Parallel Ant Colony Optimisation

**Ion Andy DITU**
*247196*
*ionandy.ditu@studenti.unitn.it*

**Amir GHESER**
*247173*
*amir.gheser@studenti.unitn.it*

***Abstract*—Ant Colony Optimization (ACO) is a bio-inspired metaheuristic that has shown remarkable success in solving complex combinatorial problems such as the Traveling Salesman Problem (TSP). In this report, we explore the implementation and parallelization of ACO using three strategies: MPI-based distributed parallelism, OpenMP-based shared-memory parallelism, and a hybrid approach combining both. Our MPI implementation adopts a coarse-grained master-slave model, while the OpenMP version exploits loop-level parallelism within a shared-memory context. The hybrid implementation integrates both techniques to enhance scalability. We benchmark these methods using TSPLIB datasets and evaluate their performance in terms of speedup, efficiency, and scalability. Results show that the MPI version offers strong scalability with moderate communication overhead, while the hybrid approach achieves higher speedups at the cost of reduced efficiency due to resource contention. We conclude by discussing the trade-offs between communication and computational workload across architectures and outline directions for future extensions.**

## I. Introduction

Real-world optimisation problems, such as vehicle routing [1], network routing [2, 3], and task scheduling [4], are often characterised by complexity, dynamic constraints, and large solution spaces that render traditional deterministic methods insufficient. Recently, researchers and practitioners have increasingly turned to bio-inspired computational approaches to tackle such challenges efficiently and adaptively. These methods draw inspiration from the cunning problem-solving capabilities observed in nature, leveraging evolutionary principles, neural processes, and collective behaviours to develop robust optimisation algorithms.

Among these bio-inspired paradigms, swarm intelligence has emerged as a particularly promising approach for solving complex combinatorial and continuous optimisation problems [5]. Swarm intelligence models the collective and decentralised behaviours exhibited by social insects such as ants, bees and termites. One of the most widely studied and applied swarm intelligence techniques is Ant Colony Optimisation (ACO), first introduced by Dorigo et al. [6]. ACO simulates the foraging behaviour of real ant colonies, which communicate indirectly via pheromone trails to discover shortest paths and adaptively improve their collective performance over time.

Due to its simplicity, flexibility, and effectiveness, ACO has been successfully applied to a wide range of domains, including travelling salesman problems, telecommunication networks, and dynamic logistics, and is still undergoing further research [7, 8].

Ant Colony Optimisation has undergone a brief evolution in its preliminary state and been refined in many works to exploit to farther extent its parallel computing advantages or to integrate some heuristics. Before delving into such extensions we will present an overview of the core approach.

### A. Ant Colony Optimisation

In nature, certain ant species wander randomly at the beginning of the search for food, and upon stumbling into resources they lay down a trail of pheromones along the path back to their colony. If another ant, while wandering randomly in the search for food, stumbles into the trail, it will stop the random search and pick up the trail whilst reinforcing the pheromone scent. Shorter paths will be used more frequently causing more pheromones to be left

on the trails. This behaviour, known as *stigmergy*, enables indirect communication among individuals through environmental changes.

The Ant Colony Optimisation algorithms draw major inspiration from this behaviour and performs a search through the parameter space, where multiple agents (ants) follow a specific function to choose the next node to reach based on pheromone levels. Generally the decision is a probability function of attractiveness of a choice (a priori) and the experience (pheromones) in taking that decision (a posteriori). For each ant $k$ we define such function $p_k(x, y)$ as

$$p_k(x,y) = \frac{\tau(x,y)^\alpha \eta(x,y)^\beta}{\sum_{z \in adj(x)} \tau(x,z)^\alpha \eta(x,z)^\beta} \qquad (1)$$

where $\tau(x, y)$ is a measure of *a posteriori knowledge* (usually pheromones), $\eta(x, y)$ is a measure of *a priori attractiveness*, and $\alpha$ and $\beta$ are two weight parameters to decide how to balance the contributions of each information. Pheromones play a crucial role in this optimisation algorithm, their update depends on an *evaporation factor* $\rho$ and the sum of the single pheromone contribution $\tau(x, y)$ of an ant $k$ for all $m$ ants.

$$\tau(x,y) \leftarrow (1-\rho)\tau(x,y) + \sum_{k}^{m} \Delta\tau_k(x,y) \qquad (2)$$

.

### B. Common Method Extensions

The original implementation was expanded over the years, modern ACO algorithms include hybridizations with other metaheuristics and local search, self-adaptation methods, and have been extended to continuous and multi-objective optimisation [9].

There are many approaches in the literature such as Ant System, Ant Colony System, Elitist Ant System, MinMax Ant System, and Rank-based Ant System, with tweaks on the update rules and transition functions. Divide-et-impera approaches like Recursive ACO where the search space is divided into regions which then get searched by smaller groups of ants; and finally, other approaches based on local search like Antabu or Continuous ACO.

## II. Method

Ant Colony Optimisation often undergoes different parallelization strategies, we implement the *master-slave with single colony and coarse-grain communication* model according to the taxonomy proposed by Pedemonte et al. [10]. By **master-slave model** they refer to a hierarchical parallel model where a master process manages the global information (pheromone matrix, current best solution, etc.) and controls the groups of slaves processes. They further distinguish based on the *granularity*, which is the amount of work done by each slave process. By **coarse-grain** they refer to models where the master manages the pheromone matrix and only interacts with complete solutions of the slaves. **Medium-grain** approaches require some sort of decomposition of the problem, for example the master node reconstructs information from partial slave solutions. Finally, **fine-grain** models perform tasks of minimum granularity, as processing subcomponents like medium-grain solutions, but also have more frequent communications. Yet another approach requires a single colony, where each ant is placed in a subset of nodes which constitutes a neighbourhood. Each neighbourhood has its own pheromone matrix and the neighbourhood may overlap, this is known as **Cellular model**. In the **multicolony model** several colonies explore the search space, each with their own pheromone matrix and they regularly exchange information among the colonies. We illustrate a summary in Table 1.

Table I: Characteristics of the models in the new taxonomy.

| **Model** | **# Colonies** | **# Pheromone matrices** | **Communication frequency** |
|---|---|---|---|
| Coarse-grain master-slave | One | One | Medium |
| Medium-grain master-slave | One | One | Medium-high |
| Fine-grain master-slave | One | One | High |
| Cellular | One | Many | Medium |
| Parallel independent runs | Several | Several | Zero |
| Multicolony | Several | Several | Low |

### 1. Algorithmic Overview

Our algorithm takes as input a graph and returns a list of nodes. After initializing the graph and a pheromone matrix $P_{G_i}$ with all ones for each group of $g = K/N$, where $K$ is the number of ants and $N$ is the number of computing cores, each group begins evaluating tours. Once each sub-colony has finished their tours we synchronize all pheromone matrices $P_{G_i}$ by accounting for evaporation and the single contributions of each subcolony $P_{G_i}$. Recall Eq. 2

$$P_{G_i}^{t+1} = (1-\rho)P_{G_i}^{t} + \sum_{j \in |G|} \Delta P_{G_j}^{t} \tag{3}$$

where $\rho$ is the evaporation factor and $t$ indicates the updated pheromone matrix.
We save the minimal tour and broadcast the updated master pheromone matrix $P_m$ and finally repeat the process until the max number of iteration $I$ is reached.

To determine the next node for an ant, we use a roulette selection, where the probability of moving to a town $(x, y)$ is elucidated in Equation 1.

**Algorithm 1** Algorithm Overview

```
for i = 0; i < I; i++ do
    ΔP = 0
    for k ∈ K do
        tour, ΔP_k = make_tour(G, P, k)
        ΔP += ΔP_k
    end for
    P ← (1 − ρ)P + ΔP
end for
```

**Algorithm 2** Construct Tour

```
reset_tour()
current_city ← random_city()
visited[current_city] ← True
for m = 1; m < M; m++ do
    next_city ← select_next_city(
       current_city, visited
    )
    tour[m] ← next_city
    visited[m] ← True
    current_city ← next_city
end for
```

**Algorithm 3** Select Next City

```
for m = 0; m < M; m++ do
    probabilities[m] ← 0.0
end for

for m = 0; m < M; m++ do
    if not visited[m] and current_city ≠ m then
        idx ← getIndex(current_city, i)
        if current_city < i
        else getIndex(i, current_city)
        τ ← (P_idx)^α
        η ← (dist(curr, idx))^−β
    end if
end for
normalize_probabilities(probabilities)

return roulette_selection(probabilities)
```

### 2. Complexity Analysis

The fundamental input parameters that impact the time complexity are the number of iterations $I$, the total number of ants $K$ and the number of cities in the graph $M$. A full tour assessment which is performed $I$ times, has a cost of $K$ ants exploring the $M$ nodes, however the cost of choosing the next node is $M$. Thus, the final time complexity is $\Theta(IKM^2)$. Memory-wise, a triangular matrix of $M$ nodes has a size of $\frac{M(M-1)}{2}$ and we need to allocate one for each group of ants $G_i = \frac{K}{N}$. Lastly, we are allocating the graph of cities for the master and each slave, meaning the total memory requirement is $\Theta\left(\frac{M(M-1)}{2} \cdot \left(\frac{K}{N} + 1\right)\right)$.

In the following we delve into the implementation details of three different implementation: one based on MPI, one on OpenMP and the last one as a Hybrid of the two.

## A. Parallelization with MPI

MPI is a standardized message-passing system designed to enable communication among processes in a distributed memory environment. It allows multiple processes, potentially running on different nodes of a cluster, to exchange data through explicit messages. MPI provides a set of communication functions, either point-to-point or collective, making it suitable for building scalable parallel applications. In order to parallelize the code for ACO, as previously said, we implemented the *master-slave* approach, where the

master node, node 0, processes global information (i.e. best cost, best solution) based on the results of the subordinate slave processes. For *coarse-grain communication* we intend that the master manages complete solutions from the slaves; this means that each slave process constructs a full solution, i.e. tours of the graph, using all its assigned ants and communicates the results back to the master. We chose this category of parallelization because it is the most direct modification of the serial implementation of ACO and it has a discrete amount of communication overhead, although we lose some information in the form of pheromones "released" by the other ants on the parallel processes until we finish the iteration and update the pheromone matrix.

*1. Algorithm*

In Algorithm 4, we have the final implementation of the parallelized version of ACO using MPI. The computational workload is divided among the processes by assigning a portion of the total number of ants, whereas the number of iterations and the vertices of the graph are kept in full. Each process independently builds $g{=}k/n$ solutions and evaluates them locally. In this step there are no dependencies between the agents handled by different processes. Once this step is done, the processes cooperate to identify the globally best solution: we obtain all the locally constructed solutions on the master using the `MPI_Gather` operation. This centralization is necessary because the algorithm requires identifying the best solution globally at each iteration, a task that requires a complete view of all candidate solutions. Having a single process perform the `calc_best` function avoids race conditions or redundant computation. Afterwards, the algorithm proceeds to update the pheromone matrix: each process computes a local pheromone contribution, based on its solutions, using the `local_pher` function. However, these local contributions must be combined into a consistent global update, so we use `MPI_Allreduce` with the `MPI_SUM` clause. This operation performs a reduction, specifically the addition operation, of all local pheromone contributions and distributes the resulting global sum back to every process. Finally, the global pheromone contributions are then added to the local pheromone matrix to reinforce paths associated with better solutions. Once all iterations are completed, the best global solution (tour and cost) is printed, typically by the root process.

Overall, `MPI_Gather` and`MPI_Allreduce` are the most important operations that maintain the balance between parallel independence and global synchronization, since ACO is easily parallelizable with this design. This strategy allows the algorithm to scale efficiently while preserving its effectiveness.

**Algorithm 4** MPI implementation

```
g ← K/N
for i = 0; i < I; i++ do
    for k = 0; k < g; k++ do
        tour[k] ← construct_solution(k)
        length[k] ← evaluate_tour(tour[k])
    end for
    contribution[] ← local_pher()
    MPI_Gather(tour[g], 0)
    if comm_rank == 0 then
        tour*, cost* ← calc_best(tour[g])
    end if
    total_contribution[]
    MPI_Allreduce(contr[], total_contr[], MPI_SUM)
    evaporate_pheromones()
    pheromones[] += total_contribution[]
end for
print(tour*, cost*)
```

*2. Data structure*

To efficiently communicate the constructed tours across MPI processes, we define the custom data structure `AntTour` that contains an array of integers representing the path (*tour*) and a double representing the cost (*tourLength*). To enable MPI to transmit these composite data in a single communication call, the algorithm defines a custom MPI data type using `MPI_Type_create_struct`. The function `defineAntTourMPIType` creates this custom MPI data type by specifying: The number and type of elements (integers and double), their respective block sizes (`NUM_CITIES` for the tour and 1 for the length) and their exact memory displacements, using `offsetof`. Once committed with `MPI_Type_commit`, this MPI data type allows `MPI_Gather` to collect all ant tours across processes in an efficient manner, simplifying data exchange and reducing overheads compared to sending multiple separate buffers.

*3. Iterative modifications*

We gradually updated the algorithm until we arrived at the final implementation described in the

previous section. In the first draft, we had the master not performing tour constructions and only handling communication and the global data, but we observed that having the master as just an overseer practically reduces the efficiency since the master core is in busy waiting for the result of the slaves and effectively not performing operations in the meantime. In this version, we had the master calculate the length of all tours from the slaves to lighten their workload, however, the imbalance was still high. Previous to the use of `MPI_Gather`, we had a for loop with a `MPI_Recv` for each slave process to obtain the tours and symmetrically each slave performed a `MPI_Send` towards the master. Another major change we made, after the modification to have a "working" master, was to switch from an update of the pheromone matrix done by the master–where it had to single-handedly calculate the contribution of each tour and add them to the matrix–to the approach where each slave calculates its contribution and "shares" it with `MPI_Allreduce` where each result is summed together and then added to the local pheromone matrix.

### *B. Parallelization with OpenMP*

Differently from MPI, OpenMP is a shared-memory parallel programming model designed to simplify the development of parallel applications on multi-core systems. While MPI requires explicit communication between processes that may run on different machines or memory spaces, OpenMP leverages threads within a single shared memory space, allowing variables to be accessed directly without sending messages. OpenMP uses compiler directives (*pragmas*) to manage parallel regions, loops, and tasks, making it suitable for parallelization. For the OpenMP implementation of ACO, we mostly took the serial version and parallelized the most expensive sections of the algorithm. Similarly to the MPI approach, the first major use of OMP appears in the construction and evaluation of tours, where each ant builds a solution independently. Each thread handles one ant's data, and local arrays, e.g. visited, are declared inside the loop to prevent race conditions.

**Algorithm 5** OMP: Construct solutions

```
#pragma omp parallel for
for i = 0; i < num_ants; i++ do
    local_visited[]
    tour[i] ← construct_sol(i, local_visited[])
    length[i] ← evaluate_tour(tour[i])
end for
```

When evaluating the cost of a tour, we used a reduction to split the summation of segment distances across threads. The reduction clause ensures that each thread accumulates a partial sum without interfering with others, and OMP combines them at the end.

**Algorithm 6** OMP: Calculate total distance

```
#pragma omp parallel for
reduction(+:total_distance)
for i = 0; i < num_cities − 1; i++ do
    idx ← get_index(tour[i], tour[i + 1])
    total_distance += distance[idx]
end for
```

After all ants finish their tours, the best tour in the iteration is selected in parallel. A loop finds the best tour for each thread using the `nowait` clause so that threads do not have to wait at the end of the loop. Using the `critical` clause, we define a section of the code that must be executed by only one thread at a time to avoid race conditions when accessing shared resources. This allows us to safely update the shared best tour across threads.

**Algorithm 7** OMP: Calculate best tour

```
local_best[]
#pragma omp parallel
{
thread_best[]
#pragma omp for nowait
for i = 0; i < num_ants; i++ do
    if ant[i].length < thread_best[].length then
        thread_best[] = ant[i]
    end if
end for
#pragma omp critical
if thread_best[].length < local_best[].length then
    local_best[] = thread_best[]
end if
}
```

During pheromone update, evaporation is applied within a nested loop over city pairs. Then, ants contribute a quantity of pheromones based on their tour length. Because multiple threads may attempt to update the pheromone matrix, we use the `atomic` clause to ensure thread-safe accumulation without data races.

**Algorithm 8** OMP: Update pheromones

```
#pragma omp parallel for
shared(pheromones)
for k = 0; k < num_ants; k++ do
    contribution ← Q/ant_tours[k].tour_length
    [...]
    idx ← get_index(from, to)
    #pragma omp atomic
    pheromones[idx] += contribution
end for
```

### C. Hybrid Implementation

The hybrid implementation combines both MPI and OMP. From the code's point of view, the hybrid version is an aggregation of the strategies used in the previous two versions. Exactly like in the MPI version, the hybrid model handles the work division of the total ants among all processes, computing local tours and keeping the best on the master. The major improvement, coming from the combination with OMP, is the parallelization of the tour construction even among threads, shown in Algorithm 9. This additional division of the workload aids the MPI approach mainly when the number of cores is low, since part of the work is computed by the thread and it lowers the number of communications among processes. The consequence is an improved scalability overall, given that the workforce is linearly increased by the number of max usable threads. The other improvements related to the use of OMP are the same as described in the OMP Implementation section and simple parallelization of some for loops.

**Algorithm 9** Hybrid: Tour construction

```
g ← K/N
for i = 0; i < I; i++ do
    #pragma omp parallel for
    for k = 0; k < g; k++ do
        tour[k] ← construct_solution(k)
        length[k] ← evaluate_tour(tour[k])
    end for
    [...]
end for
```

### D. Implementation Details

For our TSP implementation we essentially have a non-directed clique, which in turn may constitute an intensive memory requirements with big graph sizes as it is quadratic in memory complexity $O(M^2)$, where $M$ is the number of cities. For this reason we implement both the adjacency and pheromone matrices as triangular matrices and flatten them into vectors, effectively halving the memory requirements and times requirements as the matrix that is being delivered is smaller. We work with 1D arrays as this allows to ship the information through a single `MPI_Gather` instruction.

### E. Data dependencies

Data dependency analysis in parallel programming identifies relationships that affect execution order. Its goal is to prevent race conditions, ensure data consistency, and optimize performance by determining which tasks can run in parallel and which must run sequentially. Given the simplicity of the base code of ACO and it is intuitively parallelizable, it is not well-suited for data dependency analysis; that is to say, there are no parts for which this operation is that meaningful, since most function are independent, therefore we decided to perform the analysis on the major function of the program and its subfunctions, i.e. the for loop for each ant encompassing tour construction and tour evaluation.

The data dependencies for the selected portion of the code, specifically the inner part of the `construct_solution` function and the probability computation, are summarized in the following table. Focusing first on the `ant_tours[i].tour[]` array (lines 214–215), we observe a flow dependency (read-after-write), not loop-carried, as both write and read occur within the same loop iteration. This type of de-

pendency does not hinder parallelization as long as each thread works on an independent index i, which is the case here when each ant constructs its tour separately. For `visited[]` array (lines 143,145) there is an anti-dependence (write-after-read) within the same iteration. This arises because the current city is read to check if it has been visited and then marked as visited. Since these accesses are in the same iteration and ordered, they are not problematic unless shared between threads. Cases of loop-carried dependencies: Line 145 writes to `visited[]`, and this value is read on line 143 in the next iteration. This introduces a loop-carried flow dependency, which can cause race conditions if wrongly parallelized across loop iterations. Additionally, the write to `visited[]` on line 145 also conflicts with a future write in the same line of iteration i+1, leading to a loop-carried output dependency. The `probabilities[]` array also presents both intra-iteration and loop-carried dependencies. The initial write on line 108, followed by a read on line 109, represents a non-loop-carried flow dependency, which is safe under iteration-local parallelism. However, the write at line 108 also carries into the next iteration i+1, resulting in a loop-carried output dependency. This implies that without privatizing `probabilities[]` for each thread or iteration, race conditions are likely. In summary, while some memory accesses are local to each iteration and safe for parallelism (e.g., `ant_tours[i].tour[]`), shared arrays (`visited[]`, `probabilities[]`) introduce loop-carried dependencies—particularly of the flow and output kinds. We resolved these dependencies when using OpenMP by making private arrays for each thread to ensure correctness during parallel execution.

| Memory Location | Earlier Statement | | | Later Statement | | | Loop Carried? | Kind of Dataflow |
|---|---|---|---|---|---|---|---|---|
| | Line | Iteration | Access | Line | Iteration | Access | | |
| ant_tours[i].tour[] | 214 | i | write | 215 | i | read | no | Flow |
| visited[] | 143 | i | read | 145 | i | write | no | Anti |
| visited[] | 145 | i | write | 145 | i+1 | write | yes | Output |
| visited[] | 145 | i | write | 143 | i+1 | read | yes | Flow |
| probabilities[] | 108 | i | write | 109 | i | read | no | Flow |
| probabilities[] | 108 | i | write | 108 | i+1 | write | yes | Output |

## III. Experiments

### A. PBS Directives

In order to submit jobs to the cluster, we use PBS directives to set up the configuration. The code snippet in Listing 1 needs the following parameters:

- `NUM NODES`: the number of nodes to allocate in the cluster;
- `NUM CORES`: the number of processes to create;
- `STRATEGY`: the placement strategy for the processes, either pack or scatter;
- `EXECUTABLE NAME`: the path of the file to execute e.g. MPI or Hybrid version;
- `OUTPUT FILE`: a .txt file where to save the result of runs using from 2 to 64 cores.

Notably, we always allocate 64 total cores even if we do not use all of them. This is needed to execute sequentially all the runs and obtain all results more easily. To have 64 cores we use the pairs *num nodes* and *num cores* jointly, therefore resulting input pairs will be (1, 64), (2, 32) and (4, 16).

```
#!/bin/bash
#PBS -l select=[NUM NODES]:ncpus=[NUM CORES
    ]:mem=16gb
#PBS -l place=[STRATEGY: PACK/SCATTER]

#PBS -l walltime=1:00:00

#PBS -q short_cpuQ

module load mpich-3.2
NODES=(2 4 8 16 32 64)
for NODE in "${NODES[@]}"; do
    {
        mpiexec -n $NODE ./[EXECUTABLE NAME
    ];} >> "[OUTPUT FILE]"
done
echo " " >> "[OUTPUT FILE]"
```

Listing 1: Job definition

For the execution of instances where we need multiple threads, such as OMP or Hybrid, we set the total number using the `export OMP_NUM_THREADS` command as seen in the example code snippet in Listing 2.

```
THREAD=(2 4 8 16 32 64)
for T in "${THREAD[@]}"; do
    export OMP_NUM_THREADS=$T
    {
        mpiexec -n 1 ./[EXECUTABLE NAME]; }
    >> "[OUTPUT FILE]"
done
echo " " >> "[OUTPUT FILE]"
```

Listing 2: Thread usage

### B. Datasets

To run and evaluate our algorithms we used different datasets from the TSPLIB repository, that contains series of files representing graph that are typically used to solve the Traveling Salesman Problem.

These files contain details regarding the graph to explore, that can be represented with the distance matrix, spatial coordinates or others, and the type of the data can be integers or floating point numbers. We selected files that contained the x and y coordinates on the nodes, which had to be integers specifically. The files that we used were: a280.tsp, rat783.tsp, pr1002.tsp and d15112.tsp. The last file was too big to be solved completely, therefore we used just a limited number of the total ones: for some testing runs we used 2000 nodes, whereas for scalability evaluation we used 128, 256, 512, 1024 and 2048 nodes. We need to specify that the number of nodes is just one parameter for the total number of operations. For example, using 1024 ants in 10 iterations on a graph of 2000 nodes, we would perform approximately 1024 * 10 * $2000^2$ = 40.960.000.000 operations. [1]

### C. *Useful Notions and Definitions*

In parallel computing, performance may be assessed based on the following parameters:

1. **Speedup**: quantification of the gain in performance of a parallel implementation with respect to its serial counterpart. It is defined as the ratio of serial execution time $T_s$ over parallel execution time $T_p$:

$$\text{Speedup} = \frac{T_s}{T_p} \tag{4}$$

2. **Efficiency**: a measure of the efficiency of use in terms of resources when using a parallel implementation. It is defined as the speed up over the number of processors used $p$:

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{T_s}{p \times T_p} \tag{5}$$

   where $T_s$ is the serial execution time and $T_p$ is the parallel execution time.

3. **Scalability** measures the capacity of a system to handle higher workloads by adding more processor.

   (a) **Strong Scalability** inspects how well the solution time scales with the number of processors for a problem of fixed dimension. Strong scaling entails that doubling the number of processors halves the execution time, ideally speaking.

   (b) **Weak Scalability** evaluates how well the solution time changes as the problem size and the number of processors increase proportionally, whilst keeping a constant workload per processor.

### D. *Parallelization Results*

In this section, we will outline the various **execution times** of our approaches under varying processes and varying threads. In Figure I we can see that as the number of nodes surges, the time decline hits a plateau around 20 processes, keep in mind that this is an average across performance with multiple threads. On the contrary, the MPI approach hits a plateau further on around 32 processes, and we will delve deeper into the roots of this behavior later in this section. On the other hand, Figure II shows the times taken as we increase the number of threads per process, we see that for OMP the time decreases until a plateau around 32 threads. Our results for HYBRID, once again show an average but across multiple nodes; we can observe that the model reaches a plateau around 32 threads, however in the early stages there is very high variance. In light of this, we selected 4 threads as the optimal amount for the HYBRID algorithm in order to be able to measure performance metrics fairly.

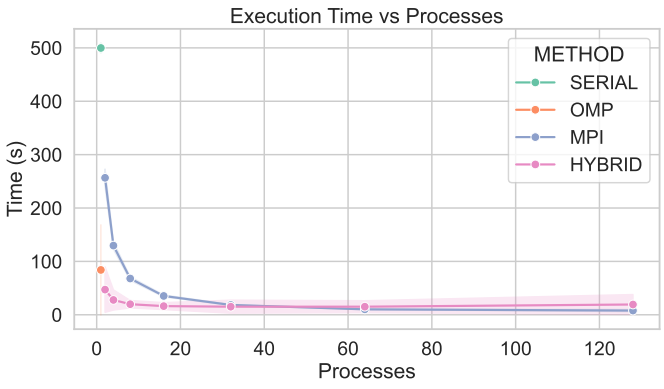


FIGURE I: EXECUTION TIMES OVER PROCESSES
In this Figure we can observe the time taken for the various methods as we increase the number of processes. For hybrid we took an average and plot the standard deviation across different number of threads.

[1] The result is obviously not meant to be treated as a precise complexity analysis

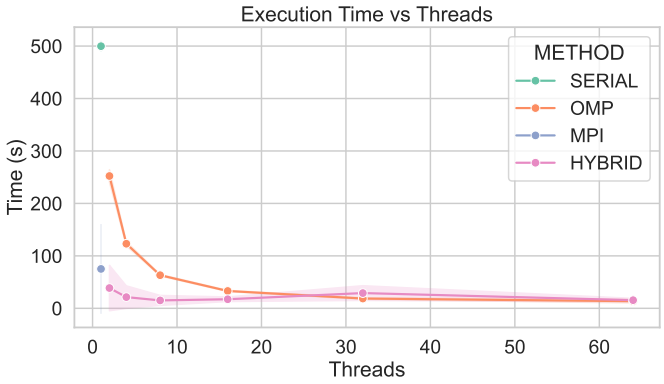


FIGURE II: EXECUTION TIMES OVER THREADS
In this Figure we can observe the time taken for the various methods as we increase the number of threads. For hybrid we took an average and plot the standard deviation across different number of processes.

## E. Evaluation

In this section we present and comment the results of our experiments, the main objective was to assess the performance of this parallel algorithm. To evaluate the performance of all approaches we will compare **speedup** and **efficiency** across a varying number of nodes (1,2 and 4). This analysis compares two communication strategies, namely *pack* and *scatter*. We conducted experiments on 4 graphs, however to ease the presentation of results we will be reporting the results on a network with 783 nodes. As such plots show results with increasing number of processes we do not include the serial implementation nor the OMP implementation, the hybrid implementation presented here is working with 4 threads.

### 1. SpeedUp

From the insights with packed strategy coming from Figure III we can observe that the speedup reached is around 40-60 and for MPI it grows linearly with the number of processes whereas the HYBRID implementation shows a plateau around 16 processes and starts slowly decaying. This is due to excessive competition between threads for resources since they are all packed on fewer nodes. It is worth noting that we would expect the speedup for HYBRID with one node to decrease faster than others, given that there are more resources that need to be shared on the single node, however this could be explained by the high variance in execution time when running on the cluster and the increased time for communication between nodes. Regarding **scatter** results we can see from Figure IV that our HYBRID algorithm performs significantly better than before approaching 80-120 in speedup. In fact, often the threads/processes on the same node will be competing for resources on that node, meaning that if scattered each node will have fewer competing threads and this explain the boost in performance.

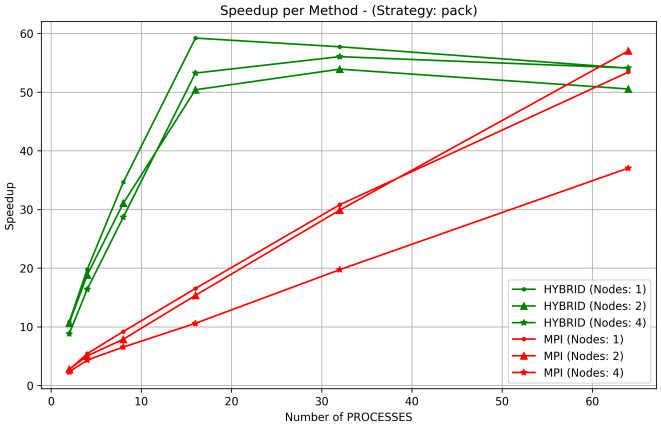


FIGURE III: SPEEDUP PACK
Speedup measurement with the **pack** strategy whilst experimenting with different amounts of nodes. The results have been run on a graph with 783 nodes.

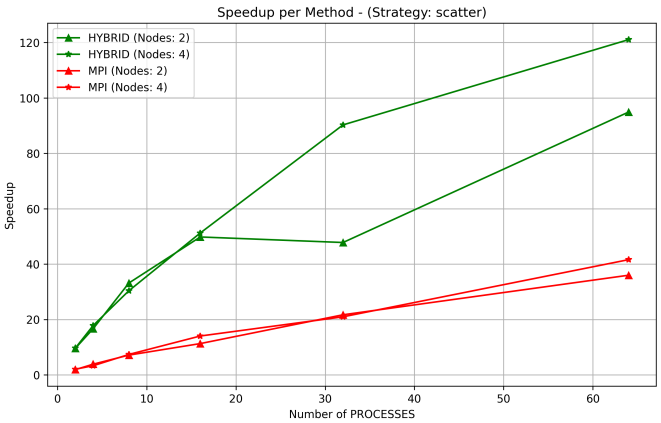


FIGURE IV: SPEEDUP SCATTER
Speedup measurement with the **scatter** strategy whilst experimenting with different amounts of nodes. The results have been run on a graph with 783 nodes.

### 2. Efficiency

The plots show results as we increase the amount of processes however we have taken into account the fact that HYBRID is using 4 threads. When looking into the **pack strategy**, it is indeed evident that the efficiency is slightly lower when using a limited amount of processes, highlighting the small commu-

nication overhead. However, as the number of processes increases we see a substantial degradation in the efficiency and this is due to the many threads competing for memory access(caches and memory bandwidth). In the **scatter** strategy we can see that this is less evident, especially when having more compute nodes for hybrid as the resources are shared between less processes. On the other hand we see that with scatter the overall efficiency is quite high for MPI, despite the extra inter-node communication overhead.

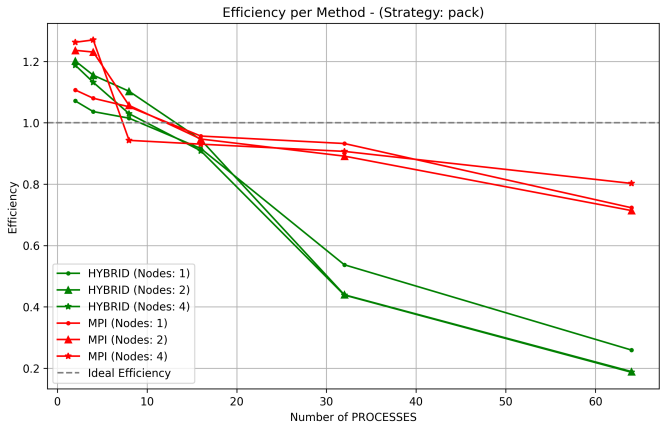


FIGURE V: EFFICIENCY OVER PROCESSES WITH PACK
Efficiency measurement with the **pack** strategy whilst experimenting with different amounts of nodes. The results have been run on a graph with 783 nodes.

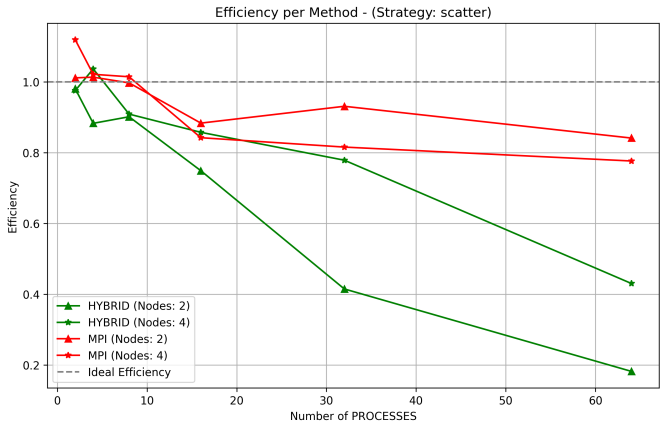


FIGURE VI: EFFICIENCY OVER PROCESSES WITH SCATTER
Efficiency measurement with the **scatter** strategy whilst experimenting with different amounts of nodes. The results have been run on a graph with 783 nodes.

### *3. Scalability*

The last assessment to be conducted is over scalability properties. It is worth mentioning that we do not expect the algorithm to showcase any linear scalability properties as doubling the size does not imply doubling the time, but to make it 4 times what it was, approximately. As can be observed both for the MPI and the HYBRID method, there are some efficiency values over 1.0. Clearly, efficiencies above one are likely artifacts of measurement errors or variability in execution time of the serial version.

1. ***MPI***: We show that MPI shows much better scalability which is aligned with our expectations and previous observations regarding communication overhead. Data regarding MPI shows that the algorithm is **strongly scalable** as there is not much decrease in performance e.g. by looking at Table II we see that the single columns more or less show the same efficiency values without much variability. Despite this, we see that the algorithm is not weakly scalable, this is due to the quadratic nature of the problem. Nevertheless, by doubling the problem size and quadrupling the amount of resources we expect to see **weak scalability**. However, we have not yet ran the correct experiments to show this.
2. ***HYBRID***: On the other hand, the hybrid implementation shows severe degradation in performance when approaching higher number of processes (Table III). This is explained by the high competition between threads to access memory resources which causes strong overhead and severe performance degradation as these runs were conducted on a single node. Indeed, as previously explore, with a higher amount of nodes such competition is less severe however the program is subject to more inter-node communication overhead. We conclude that the algorithm is scalable until 16 processes. However by looking at Figure VI we see that with more nodes and a scatter strategy we get better results.

TABLE II: MPI EFFICIENCY

| **Processes** | **128** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|---|
| 2 | 1.11 | 1.15 | 1.18 | 0.96 | 0.87 |
| 4 | 1.09 | 1.14 | 1.18 | 0.96 | 0.87 |
| 8 | 1.07 | 1.14 | 1.18 | 0.94 | 0.86 |
| 16 | 1.08 | 1.14 | 1.16 | 0.92 | 0.86 |
| 32 | 0.96 | 1.08 | 1.09 | 0.88 | 0.84 |

Table III: Hybrid Efficiency

| Processes | 128 | 256 | 512 | 1024 | 2048 |
|---|---|---|---|---|---|
| 2 | 1.00 | 1.13 | 1.18 | 1.16 | 1.09 |
| 4 | 0.98 | 1.09 | 1.18 | 1.15 | 1.00 |
| 8 | 0.94 | 1.08 | 1.11 | 1.07 | 0.91 |
| 16 | 0.89 | 0.97 | 0.96 | 0.79 | 0.70 |
| 32 | 0.27 | 0.48 | 0.43 | 0.39 | 0.39 |

## IV. Conclusion

We have implemented a coarse-grained master-slave parallel implementation of Ant Colony Optimization which, in light of our previous results presented in Section III, show that the implemented approach with MPI can be said to be strongly scalable; we suggest that with further experimentation we can confirm that the algorithm is weakly scalable if taking into consideration the quadratic nature of the problem. On the other hand, despite yielding better speedups, the hybrid approach is less efficient given that there is more competition for resources, especially when the processes are packed on the same node. The packed strategy has revealed to be more effective for the MPI implementation, as this imposes a lower demand on the single nodes. When working with multiple nodes we observe an inter-node communication overhead which imposes a loss in efficiency in the MPI implementation. Nevertheless, for the hybrid implementation this shows itself as an opportunity to reduce the intense demand of node resources.

## V. Future work

Despite achieving an approach which is strongly scalable with MPI, we argue that with appropriate experiments we can demonstrate it is weakly scalable as well whilst considering the quadratic complexity of the task and leave this as an opportunity for further experimentation. On the other hand, our hybrid implementation despite some speedup, shows unsatisfactory efficiency, potentially leaving room for redesigning the usage of threads to make it less competitive in memory access as this often constitutes a severe bottleneck for scalable performance. On a second note, an additional lead for research would be to implement a different parallelization of the ACO algorithm as extensively outlined at the beginning of Section II, as more sophisticated strategies could be more advantageous when exploiting the hybrid approach with an increased number of threads.

## References


[1] D. M. Chitty, E. F. Wanner, R. Parmar, and P. R. Lewis, "Applying partial-aco to large-scale vehicle fleet optimisation," *CoRR*, vol. abs/1904.07636, 2019. [Online]. Available: http://arxiv.org/abs/1904.07636

[2] G. D. Caro, "Ant colony optimization and its application to adaptive routing in telecommunication networks," Ph.D. dissertation, Universite Libre de Bruxelles, 2004.

[3] C. M. and S. Thakur, "Ant-net: An adaptive routing algorithm," in *2016 IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)*, 2016, pp. 1–4.

[4] J. Elcock and N. Edward, "An efficient aco-based algorithm for task scheduling in heterogeneous multiprocessing environments," *Array*, vol. 17, p. 100280, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S259000562300005X

[5] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 10 1999. [Online]. Available: https://doi.org/10.1093/oso/9780195131581.001.0001

[6] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[7] R. Priyadarshi and R. R. Kumar, "Evolution of swarm intelligence: a systematic review of particle swarm and ant colony optimization approaches in modern research," *Archives of Computational Methods in Engineering*, pp. 1–42, 2025.

[8] M. A. Awadallah, S. N. Makhadmeh, M. A. Al-Betar, L. M. Dalbah, A. Al-Redhaei, S. Kouka, and O. S. Enshassi, "Multi-objective ant colony optimization," *Archives of Computational Methods in Engineering*, vol. 32, no. 2, pp. 995–1037, 2025.

[9] G. Iacca, "Swarm intelligence ii," Bio-Inspired Artificial Intelligence, 2024.

[10] M. Pedemonte, S. Nesmachnow, and H. Cancela, "A survey on parallel ant colony opti-

mization," *Applied Soft Computing*, vol. 11, no. 8, pp. 5181–5197, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S156849461100202X