Scheduling Microservices in a Container using Ant Colony Optimization

Michael Soebroto
Computer Science and Engineering
University of California, Santa Cruz
msoebrot@ucsc.edu

Jay Mehta
Computer Science and Engineering
University of California, Santa Cruz
jmehta1@ucsc.edu

Abstract

Efficient container scheduling is crucial for optimizing resource utilization and application performance in cloud computing environments. This paper proposes an multi-objective approach to container scheduling using Ant Colony Optimization (ACO). The ACO algorithm leverages the collective intelligence of virtual "ants" to dynamically allocate containers to available resources while considering multiple objectives. These objectives can include factors such as minimizing resource utilization, balancing load across machines, and node failure rates. We evaluate the performance of our ACO-based scheduler by comparing it with an implementation inspired by the traditional Least Loaded scheduling. The results demonstrates that our approach achieves significant improvements in resource load balancing with different sets of clusters. This research highlights the potential of ACO for container scheduling in distributed systems, offering a promising approach for efficient resource utilization and improved containerized application performance.

1 Introduction

In the realm of cloud computing, containerized applications have become a dominant force. These applications, built on the foundation of microservices architecture, rely on efficient resource management to function optimally. This is where container scheduling takes center stage.

Container scheduling is the systematic allocation of resources – CPU, memory, storage – to containers deployed across a cluster of machines. It acts as an intelligent orchestrator, ensuring each container receives the necessary resources to execute its tasks effectively. This process optimizes resource utilization, promoting several key benefits. Unlike traditional virtual machines (VMs) that require their own operating system, containers share the host machine's kernel. This lightweight nature allows container schedulers to pack containers more efficiently onto host machines, maximizing resource utilization and reducing overall cloud infrastructure costs.

Based on the author's observation in Kubernetes [8], which is an open-source platform for automating the deployment, scaling, and management of containerized applications, we can categorized all such scheduling algorithms used by these applications, mainly into four categories as described below.

1. **Generic Scheduling** - These algorithms mainly deal with efficient resource allocation and fairness among containers. Many strategies have been studied here like least square fit, priority queues, affinity rules, etc.

2. **Multi-Objective and Optimization based** - Schedulers in this category incorporates many objectives for the optimization process and aims to benefit for a group of parameters. These algorithms go a bit beyond the core generic properties, and provide a more holistic approach in an distributed environment.

3. **AI-Focused** - This category of schedulers introduces Artificial Intelligence (AI) within the main logic. By leveraging Machine Learning, schedulers are able to analyze several metrics of a cluster like usage patterns, job completion times and predict future demands proactively. This is something similar to the usage of Speculators [7] used by the authors, to achieve different synchronous states within a File System.

4. **Autoscaling algorithms** - Addressing the dynamic nature of the cloud environments, these algorithms automatically adjust resource allocation based on fluctuations in workloads. Scaling up and down based on demands, to keep costs in checks are the strategies used here.

In this paper, we will explore an multi-objective approach that utilizes a popular search algorithm, Ant Colony Optimization(ACO), in order to provide optimized schedules in terms of resource load balance and compare it with a more traditional approach of scheduling.

2 Motivation

Atul V. and Dharmendra K. Y. [4] in their paper, express their discerns regarding the needs of datacenters not being fulfilled by common single-objective scheduling algorithms. They described an distributed environment where the main job was to bind set of tasks received by the broker to the received list of Virtual Machines running on the server. They show how scheduling done by FCFS, SJF and Priority based is done efficiently in best cases but degrades the overall performance to a very low level such that it cannot be run.

Thus the main motive was to adapt to a multi-objective optimization based scheduling algorithm, that offers better performance in a real-time distributed network where priorities are not just the execution times of each job, but rather a set of parameters that can affect the processing of each job in a container.

Another review for the need of an multi-objective algorithm was displayed by Tarek M. in implementing the Kubernetes scheduling strategy [6]. The goal was to maximize number of containers being executed, minimize the makespan of executing containers and minimize power given n sets of nodes and c sets of containers. The principle of the scheduler lies in maximizing the CPU, memory and storage usage rate while minimizing power consumption and number of instances running.

Our motivation follows along those lines, we wanted to propose an optimal resource utilization in a cloud environment where several containers are instantiated which support several microservice applications. We will simulate the working of the Ant Colony Optimization Algorithm and show that it does achieve better utilization compared to straightforward Most Resource Remaining scheduler.

3 Design and Implementation

Ant colony optimization is a population-based search algorithm based on the pheromone trail laying behavior of ants in searching for food. It simulates the feeding process to complete the scheduling of microservices.

Kaewkasi and Chuenmuneewong (2017) presented the first Ant Colony Optimization algorithm for container scheduling with the goal to enhance resource utilization using proper load balancing. The algorithm was integrated and tested on Docker Swarm. The results were compared with Swarm greedy approach showing 15% performance enhancement. [1] [3]

3.1 Algorithm Overview

The algorithm can be summarized as follows. [5]

1. Ant_i is randomly placed to a microservice s_i.

2. Then ant_i then selects a $path_{i,j}$ with a certain probability p to reach a node n_j which is summarized in table 1. $Container_i$ is the number of containers for each microservice in the cluster and basically indicating the service needs to be scheduled equivalent to the count of $container_i$ times.

3. After one iteration i, ant_k returns to the next microservice and repeats step(2).

4. In each round, all the ants simulate the allocation of containers for all microservices. This process of simulating allocation by all ants is considered one iteration. The algorithm continues these iterations until it reaches a predefined maximum number of iterations.

3.2 Heuristics

Let H_k be the heuristic information which represents how likely an ant_k decides to assign the container for a particular microservice s_j to a node n_j. This likelihood is based on the calculation of the three models described above. H can be calculated as follows.

$$H_{i,j} = \left(1 - F_j\right)\left(\frac{1}{1 + \frac{R_i(t)+C_i(t)}{M_j(max)+C_j(max)}}\right) \tag{1}$$

where,
$H_{i,j}$ = Represents the heuristic value for assigning element i to target j
F_j = Represents the fail rate of target j
$R_i(t)$ = Represents the remaining processing units of element i at current time t
$C_i(t)$ = Represents the remaining storage units of element i at current time t
$M_j(max)$ = Represents the maximum processing capacity of target j
$C_j(max)$ = Represents the maximum storage capacity of target j

4 Implementation

The core logic of the ACO algorithm involves iteratively searching for the best way to assign elements (tasks, services) to targets (machines, nodes). It considers resource utilization and potentially other objectives.

Here's a brief overview of the algorithm stated in Algorithm 1.

4.1 Initialization

The algorithm starts by setting up a "pheromone matrix." This matrix keeps track of how desirable it is to assign a particular element to a specific target. Initially, all these desirability values are set to a small value. We also initialize variables to track the best solution found so far, the costs of different solutions, and the number of iterations completed.

4.2 Main Loop

This is the heart of the algorithm and repeats for a predetermined number of iterations (n_iter), simulating multiple agents and exploring different solutions. Each agent will build an assignment by probabilistically choosing targets for

elements based on pheromone trails and resource fit. They each calculate the cost of their assignments and update the best and worst solutions found so far.

Finally, pheromone trails are strengthened for assignments that led to better (lower cost) solutions, guiding future exploration towards improved assignments.

4.3 Output

After all the iterations are complete, the algorithm returns the best solution (assignment list) found with the lowest cost. This represents the optimal (or near-optimal) way to assign the elements to the targets based on the defined objectives.

Algorithm 1 Ant Colony Optimization

Input: *elements*, *targets*, *n_iter*, *n_agents*
Output: *best_solution*

pheromones ← Matrix(Elements, Targets)
best_solution ← *None*
best_cost ← ∞
worst_cost ← 0

for *iter* ← 1 *num_iterations* **do**
    Initialize *agent_cost* and *agent_sol*

    for *agent* ← 1 *num_agents* **do**
        get *assigned_list* and available resources

        for *element* ∈ Elements **do**
            Calculate probability p for each service
            Assign each service their p_s

        Calculate total cost
        Update *ant_paths* and *ant_cost*

        if *total_cost* < *best_cost* **then**
            best_solution ← assigned_list
            best_cost ← *total_cost*
        if *total_cost* > *worst_cost* **then**
            worst_cost ← *total_cost*

    Update *pheromones*

return *best_path*

5 Evaluation

For our evaluations, there were several factors we wanted too look at:

1. **Resource Allocation**: In containerized environments, one of the most significant goals in creating a good scheduling is to maintain a balanced resource utilization across all different nodes.
2. **Multi-Objective Scheduling**: The ability to optimally schedule containers based on several factors more than just resource allocation. Other important aspects for creating optimal schedules are factors such as transmission overhead and data locality.
3. **Failure Rate**: To ensure good up-time on cloud computation, we want to minimize the failure rate of the system, therefore, we want to prioritize sending containers to nodes with lower failure rates.

5.1 Experiments

For our experiments, we ran our system within a Python notebook over Google Colab for ease of access. We also created the Most Remaining Resources approach, inspired by Least Loaded scheduling, where containers are scheduled to nodes with the most available resources. Using our implementation of the ACO multi-objective function and the Most-Remaining-Resources implementation as a baseline, we generated schedules based on our created workload and cluster, detailed in Table 1 and Table 2. The above schema is inspired from the Alibaba Cluster Trace Program *v2017* [2], which is workload comprising of real production values and metrics from 1300 machines in a period of 12 hours.

Using these implementations, we tested the scheduling ability on the service workload over 3 cases:

- A balance node cluster
- A node cluster with an outlier
- A node cluster with fail rates

Both the balanced and the cluster with an outlier was tested on both the ACO and the Most-Remaining-Resources algorithms and the node cluster fail rate was performed on just the ACO.

To get an idea of the resource load balancing, we evaluated the resource utilization rate of each node given the generated schedule and the standard deviation. To show the multi-objective capability of this algorithm, we collected the resource utilization rate of the ACO algorithm with fail rate taken into account.

5.2 Balanced Node Cluster

In our first test, we ran the algorithm using the created dataset. In our test, we found that the schedule generated showed a more balanced utilization rate in comparison to the Most Remaining Resource algorithm. Our results are show in Figure 1, more nodes found higher or similar resource utilization within the ACO schedule.

To better show this finding, we also calculated the standard deviation of the computed schedules and compared it with the worst schedule generated by the ACO algorithm. In Figure 2, we show that our ACO algorithm produced an algorithm where the standard deviation was just 0.077, which is an ~11% improvement from the computed standard deviation of 0.086 for MRR.

Table 1. Description of the Microservices

Microservice	Services scheduled	Dependencies	Computational requirement	Memory Requirement	N_Containers
s_1	-	(s_2)	7	10	5
s_2	(s_1)	(s_4)	6	12	3
s_3	-	(s_4)	8	16	4
s_4	(s_1, s_2)	(s_5)	6	10	3
s_5	(s_4)	(s_6, s_7)	5	14	5
s_6	(s_5)	(s_7)	7	12	4
s_7	(s_5, s_6)	(s_8)	6	10	3
s_8	(s_6, s_7)	-	8	16	5

Table 2. Description of the Physical Nodes in Cluster

Physical nodes	Computational resources	Memory resources	Failure Rate
p_1	70	100	0.4
p_2	60	110	0.01
p_3	80	90	0.03
p_4	90	120	0.02
p_5	70	80	0.05
p_6	90	100	0.04
p_7	50	110	0.02
p_8	80	120	0.06
p_9	60	130	0.03
p_{10}	50	100	0.02

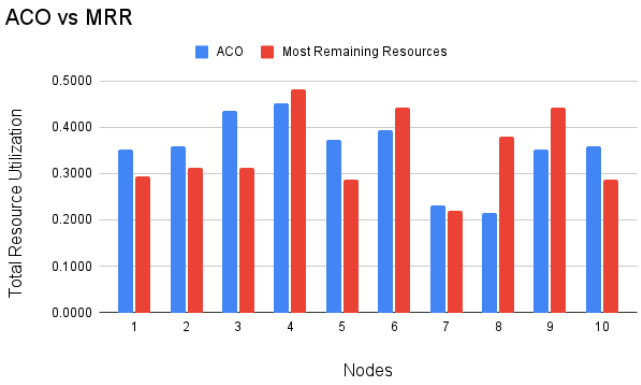

Figure 1. Resource utilization rates of ACO vs MRR on a balanced cluster without failure rate considered.

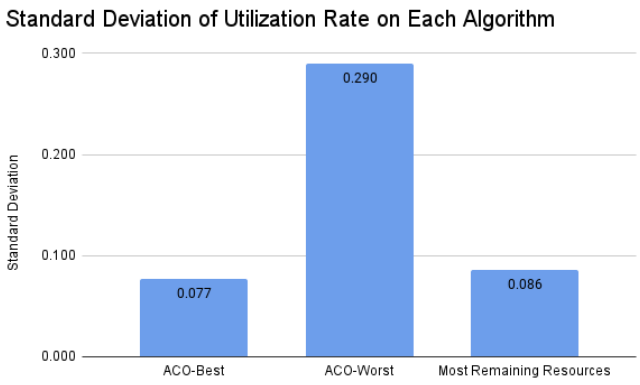

Figure 2. Standard deviation of recorded node resource utilization rates in ACO and MRR.

5.3 Unbalanced Node Cluster

For this experiment, we wanted to demonstrate the ability of the algorithm to maintain balanced resource utilization in the event that a node within a cluster is deemed a lot more favorable by the algorithm. To do so, we modify the cluster in Table 2, such that we set Node 6 to have a maximum computational resource of 180 and maximum memory resources of 200. By doing so, we create a situation in the cluster where a scheduling algorithm might try to pile as many containers as possible onto that node and this behavior is shown in the MRR algorithm.

In Figure 3, we can see that the MRR algorithm causes high resource utilization within Node 6, causing an unbalanced resource allocation on the cluster. With our ACO algorithm, we were able to normalize the usage and take more advantage of resources across all the nodes in the cluster. This is beneficial in containerized environment where machines are susceptible being overloaded by containers, leading to system crashes and bottlenecks.

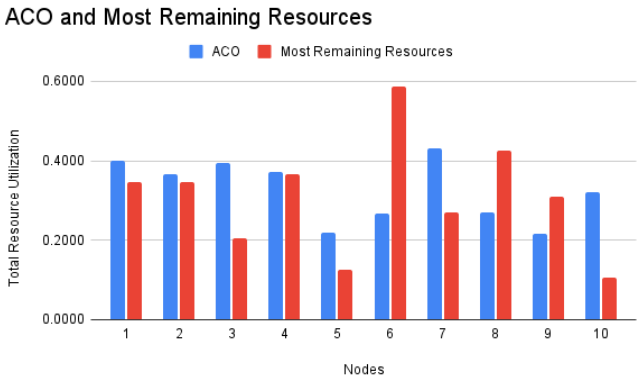

Figure 3. Resource utilization rates of ACO vs MRR on an unbalanced cluster without failure rate considered.

5.4 Multi-Objective: Failure Rates

Using the ACO algorithm gives us flexibility over the objective that you can achieve when designing a scheduling algorithm. To demonstrate this fact, we take node failures into account as machines in real-life are always susceptible to failure. The failure rates for each node are described and then taken to account in the design of our ACO algorithm.

In Figure 4, we graph the resource utilization rates of each node given a new schedule that takes into account the failure rates of each node. The failure rates are represented by the red line, and as we can see, the ACO algorithm adjust the load that is being put onto Node 1 which has an much greater failure rate in comparison to the other nodes in the cluster, exemplifying the ability of the ACO algorithm to handle the multi-objective landscape of container scheduling.

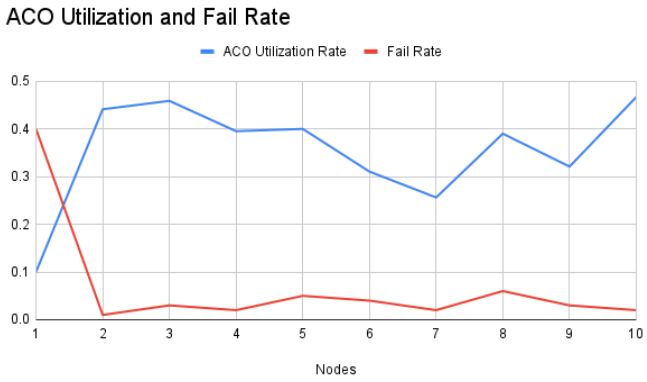

Figure 4. Resource utilization rates of ACO with failure rate consideration.

5.5 Takeaways

From the experiments that we ran, we can see that our implementation of the ACO algorithm is capable of promoting a more balanced resource allocation across nodes on a cluster. It should be noted that more experiments are required to show capability in a real-world setting, however, the approach shows improvement in our desired heuristics of resource load distribution while also taking into account other objectives such as failure rate.

6 Conclusion and Future Work

In conclusion, this paper presented an approach to container scheduling using Ant Colony Optimization (ACO) for multi-objective optimization. The proposed method effectively allocates containers to available resources while considering these objectives. Our results demonstrate that the ACO approach achieves significant improvements in scheduling performance compared to our MRR approach which draws ideas from the traditional Least Loaded scheduling algorithm. This research highlights the potential of ACO for container scheduling in large-scale distributed systems, paving the way for further exploration of its application in dynamic and resource-constrained environments. By incorporating multi-objective optimization into the scheduling process, ACO offers a promising approach for ensuring efficient resource utilization and improved containerized application performance.

As for future work, one of the more important things we to consider is a test on real-world applications. Due to our resource constraints, we were only able to perform a limited evaluation using a small dataset. Testing on a more practical workload would be to test out the ACO approach on a larger dataset with real-time metrics involved and conclude upon its viability based on the performance. But due to the the limited resources available at hand, this experimentation is performed on a much smaller scale.

An interesting implementation was done by Chanwit K. and Kornrathak C. [3], where they proposed a modification of the SwarmKit version of Docker 1.13. The main algorithm was written in Go, and was evaluated on a cluster with 1 manager and 5 worker nodes. Their greedy approach maximised application performance rather than containers being fully packed, keeping different hueristics and outcomes.

Another factor to consider is that container scheduling includes much more constraints than the ones we propose in our approach, which would be great to include within an improved ACO approach, however, this work shows that even with a limited amount of heuristics, we can see an improvement in the quality of schedules. Other avenues to look into is to improve the ACO algorithm itself, as although ACO is often regarded as one of the better optimization algorithms out there, it is known to be prone to getting stuck in local optima as well as not converging as quickly as other known search algorithms. There are many cases where people have used techniques from other algorithms to improve on some of the limitations of the ACO approach, and

as with any other search algorithm, there also requires some level of parameter fine-tuning for optimal performance.

References

[1] Imtiaz Ahmad, Mohammad Gh AlFailakawi, Asayel AlMutawa, and Latifa Alsalman. 2022. Container scheduling techniques: A survey and assessment. *Journal of King Saud University-Computer and Information Sciences* 34, 7 (2022), 3934–3947.

[2] Alibaba. [n. d.]. Alibaba/clusterdata: Cluster data collected from production clusters in Alibaba for Cluster Management Research. https://github.com/alibaba/clusterdata

[3] Chanwit Kaewkasi and Kornrathak Chuenmuneewong. 2017. Improvement of container scheduling for Docker using Ant Colony Optimization. In *2017 9th International Conference on Knowledge and Smart Technology (KST)*. 254–259. https://doi.org/10.1109/KST.2017.7886112

[4] Atul Vikas Lakra and Dharmendra Kumar Yadav. 2015. Multi-objective tasks scheduling algorithm for cloud computing throughput optimization. *Procedia Computer Science* 48 (2015), 107–113.

[5] Miao Lin, Jianqing Xi, Weihua Bai, and Jiayin Wu. 2019. Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud. *IEEE access* 7 (2019), 83088–83100.

[6] Tarek Menouer. 2020. KCSS: Kubernetes container scheduling strategy. *The Journal of Supercomputing* 77 (2020), 4267 – 4293. https://api.semanticscholar.org/CorpusID:225011380

[7] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2008. Rethink the sync. *ACM Trans. Comput. Syst.* 26, 3, Article 6 (sep 2008), 26 pages. https://doi.org/10.1145/1394441.1394442

[8] Khaldoun Senjab, Sohail Abbas, Naveed Ahmed, and Atta ur Rehman Khan. 2023. A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing* 12, 1 (2023), 87.