



Please cite the Published Version

Peake, J, Amos, M, Costen, N , Masala, G and Lloyd, H  (2022) PACO-VMP: Parallel Ant Colony Optimization for Virtual Machine Placement. Future Generation Computer Systems: the international journal of grid computing: theory, methods and applications, 129. pp. 174-186. ISSN 0167-739X

DOI: <https://doi.org/10.1016/j.future.2021.11.019>

Publisher: Elsevier

Version: Accepted Version

Downloaded from: <https://e-space.mmu.ac.uk/629025/>

Additional Information: This is an Author Accepted Manuscript of an article published in Future Generation Computer Systems, by Elsevier.

Enquiries:

If you have questions about this document, contact openresearch@mmu.ac.uk. Please include the URL of the record in e-space. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from <https://www.mmu.ac.uk/library/using-the-library/policies-and-guidelines>)

PACO-VMP: Parallel Ant Colony Optimization for Virtual Machine Placement

Joshua Peake^a, Martyn Amos^b, Nicholas Costen^a, Giovanni Masala^a, Huw Lloyd^{a,*}

^aDepartment of Computing and Mathematics, Manchester Metropolitan University, Manchester, United Kingdom.

^bDepartment of Computer and Information Sciences, Northumbria University, Newcastle upon Tyne, United Kingdom.

Abstract

The Virtual Machine Placement (VMP) problem is a challenging optimization task that involves the assignment of virtual machines to physical machines in a cloud computing environment. The placement of virtual machines can significantly affect the use of resources in a cluster, with a subsequent impact on operational costs and the environment. In this paper, we present an improved algorithm for VMP, based on Parallel Ant Colony Optimization (PACO), which makes effective use of parallelization techniques and modern processor technologies. We achieve solution qualities that are comparable with or superior to those obtained by other nature-inspired methods, with our parallel implementation obtaining a speed-up of up to 2002x over recent serial algorithms in the literature. This allows us to rapidly find high-quality solutions that are close to the theoretical minimum number of Virtual Machines.

Keywords: Virtual Machine Placement, Ant Colony Optimization, Swarm Intelligence, Parallel MAX-MIN Ant System, Parallel Ant Colony Optimization

1. Introduction

Cloud computing [22] is an increasingly prevalent computing paradigm, in which on-demand computing services (such as compute or storage) are provided, either privately or commercially, as a service to remote users and organizations. As well as providing the foundation for modern electronic commerce, cloud computing is a key enabler for a number of recent developments, such as the Internet of Things [6], Edge Computing [37], and Big Data Analytics [2], all of which, in turn, enable important societal developments such as Smart Cities [47] and Intelligent Transportation Systems [21]. However, data centres now represent a significant proportion of global energy usage (currently estimated at around 2%, and this is set to rise [19]). There is, therefore, an urgent need to optimize the software infrastructure underpinning modern data centres.

Resource requirements are expressed in terms of a number of Virtual Machine (VM) instances, each of which carries its own overhead. A key benefit of cloud computing for users is its *scalability*, which is derived from the ability to dynamically increase and reduce resource usage depending on demand. While this *elasticity* is beneficial for users, it presents challenges for cloud computing providers. As demand changes constantly, the assignment of VMs to servers (or Physical Machines, PMs) can quickly become inefficient, leading to sub-optimal utilisation of servers. This can cause providers to use more hardware resources than are necessary, which has both

an economic and environmental impact. The solution to this is *virtual machine consolidation*, which allocates currently in-use VMs to as few PMs as possible (essentially “packing” them into servers). This increases server utilisation and energy efficiency, and lower power consumption equates to lower energy costs for the host. This also incentivizes the efficient re-allocation of servers to ensure that they operate in an efficient configuration for a longer duration, which leads to further reductions in energy usage.

A number of algorithms have been proposed to address this problem; here, we focus on methods based on *Ant Colony Optimization*. Importantly, we focus on *parallel* Ant Colony Optimization, which takes advantage of modern multi-core hardware to significantly reduce the time required to find satisfactory solutions. We make use of the AVX2 instruction set, available on the vast majority of modern CPUs, to further reduce execution time in an already parallelized approach.

The rest of the paper is organised as follows: In Section 2 we provide background to the Virtual Machine Placement problem and existing methods for its solution; in Section 3 we describe the Ant Colony Optimization method and our own improved algorithm; in Section 4 we present the results of evaluating our algorithm against competing techniques, and in Section 5 we discuss our findings and suggest possible further work.

2. Background & Related Work

In this Section we first describe the Virtual Machine Placement Problem and discuss a range of existing methods for its solution.

*Corresponding author: huw.lloyd@mmu.ac.uk. Author Accepted Manuscript shared under Creative Commons licence CC-BY-NC-ND. Full paper: <https://doi.org/10.1016/j.future.2021.11.019>

2.1. Virtual Machine Placement Problem

Hardware virtualization in cloud computing allows for many separate machine instances to be created that are distinct from the host machine on which they are running. These instances, known as Virtual Machines (VMs), essentially act as completely separate computers, distinct from other VMs running on the same host. Each VM may have its own specific resource requirements (in terms of memory, and so on), and thus occupies a specific “footprint”. These VMs are managed by a *hypervisor* running on the host server (also known as a Virtual Machine Monitor, VMM) which creates, optimizes and monitors the performance of VMs.

Many companies, such as Amazon (AWS) and Microsoft (Azure), provide access to Virtual Machines hosted on their own servers. Due to the sheer size of these operations, a significant amount of hardware is required. In order to minimize, as far as possible, the amount of costly physical infrastructure, a process known as *Virtual Machine Migration* is often used to move Virtual Machines from one Physical Machine to another in a seamless fashion, without disruption for the user [12]. This ability to migrate VMs therefore offers the possibility of *optimization* of their placement on servers. The fundamental question we address, therefore, is as follows: given a set of VMs, each with a specific resource footprint, and a number of PMs with individual capacities, what is the best allocation of VMs to PMs, such that the number of PMs is minimised? For scenarios where a small number of servers are available, determining the most efficient allocation for a small number of VMs can be trivial. However, services such as AWS have millions of users and hundreds of thousands of servers.

Virtual Machine Placement (VMP) is an NP-hard problem [40], in which the aim is to allocate Virtual Machines (VMs) to Physical Machines (PMs) as efficiently as possible. While VM features differ between variants of the problem, the two most typical attributes are *memory* (RAM) and *processing* (CPU). RAM requirements are generally measured in Gigabytes (GB), while CPU requirements are generally measured in either processor cores or MIPS (million instructions per second). Every VM has its own specific requirement for each, which means it occupies its own resource “footprint”. The aim of the problem is to “legally” allocate every VM to a PM in such a way that the number of PMs required is minimised (that is, this version of the VMP is a variant of a bin packing problem). Here, we focus on the *static* variant of the VMP problem, where we need to allocate a fixed set of VMs to PMs.

Although the static variant of the VMP problem is relatively limited in applicability (as, in reality, cloud computing represents a dynamic situation in which virtual machines are added and removed at different times) it still offers a useful challenge, with a number of recent solutions in the literature for the purposes of comparison. Our focus here, and the key contribution of the current paper, lies in the *efficient* solution of such problems using parallelized and vectorized ACO. This work represents an evolution of our earlier work on applying ACO to the Travelling Salesman Problem (TSP) [35, 36], and future work will adapt our methods to solve the dynamic version of VMP.

We note that the pheromone evaporation mechanism of ACO could provide a suitable adaptation strategy for the dynamic problem, which has recently been shown to be effective on the dynamic version of the TSP [32].

2.1.1. Problem definition

We now formally define the variant of VMP that we consider here. An instance of the VMP is defined by a set V of virtual machines $V = \{V_i, i \in [1, N_{VM}]\}$ with *CPU requirements* and *RAM requirements* $C_i^{req}, R_i^{req} \forall i \in [1, N_{VM}]$, and a set P of physical machines $P = \{P_j, j \in [1, N_{PM}]\}$ with *CPU capacities* and *RAM capacities* $C_j^{cap}, R_j^{cap} \forall j \in [1, N_{PM}]$. A *feasible solution* to an instance of the VMP is a mapping of the indices of virtual machines i to physical machines j such that $\forall j, \sum_i C_i^{req} \leq C_j^{cap}$ and $\sum_i R_i^{req} \leq R_j^{cap}$ where the sums are taken over the indices of all virtual machines i which are mapped to the physical machine j . The optimization problem seeks to find a feasible solution which maximizes the number of *empty* physical machines, which is equal to the cardinality of the set of indices j which are not mapped from any virtual machines i .

An illustrative example of an instance of the static VMP problem is shown in Figure 1, along with its solution. We have an initially unbounded number of PMs, each with a fixed CPU and RAM resource, and a number of VMs (VM1-5) to be allocated to PMs such that the total resource requirement on each PM does not exceed its capacity, and the number of PMs is minimised (in this case, to three).

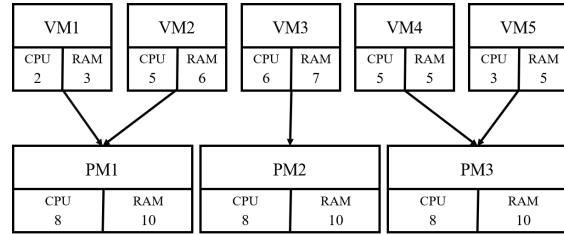


Figure 1: Instance of the Virtual Machine Placement problem, with arrows showing the solution (i.e., the allocation of VMs to PMs). Virtual Machine requests are efficiently allocated to Physical Machines

2.2. Existing Methods

As with many combinatorial optimisation problems, a wide range of techniques exist to find solutions to VMP instances. As well as heuristic-based approaches such as Next Fit, First Fit, First Fit Decreasing (FFD) and Best Fit (BF) [13], more advanced optimization techniques have been successfully applied to the VMP problem; these include Genetic Algorithms (GA) [20, 33, 42], Particle Swarm Optimization [45], Q-Learning [31] and Ant Colony Optimization (ACO) [3, 18, 20, 27].

A recently-published method for VMP, which provides one of the comparison baselines for the work presented here, is the IGA-POP genetic algorithm (GA) [1]. This frames the VMP as a Variable-Sized Bin Packing Problem (VSBPP), a variant of the Bin Packing Problem in which the container elements have differing capacities. In IGA-POP, a solution encodes an ordering of VM assignments to PMs. The fitness function for this

algorithm prioritises low power usage, and it performs competitively in terms of solution quality against the BF and First-Fit (FF) greedy algorithms, the Sine-Cosine Optimization Algorithm (SCA) [34] and a generic GA. For this reason, we select IGA-POP as being representative of the “evolutionary” algorithm class of solvers for VMP.

Our own method is based on Ant Colony Optimization (ACO), [15] which is an optimization metaheuristic modelled on the foraging behaviour of ants [14]. When ants leave the nest to look for food, they initially explore the local area; once food is located by an ant, it returns to the nest. On the return journey, the ant leaves a *pheromone trail*, which increases the probability that other members of the colony will take that path to the food source. As each ant follows a trail, it also lays its own trail, strengthening the pheromone over time and causing more ants to follow it, in a process of positive feedback reinforcement.

This phenomenon is abstractly replicated by the ACO algorithm, with problem instances generally represented by graph structures, and with multiple “ants” searching for a solution by traversing its edges according to pheromone concentrations. As an ant traverses a problem graph, it uses a weighted random process to select its next move, in which solution components with a better combination of pheromone and heuristic are more likely to be selected. Pheromone “evaporates” over time, allowing unproductive paths to be eventually removed (thus preventing premature convergence).

Feller *et al.* [18] presented an early application of ACO to VMP. This treats VMP as a multi-dimensional bin-packing problem (MDBP), with the bins being the Physical Machines, and the items to be packed being the VMs. As reducing the number of PMs is the most effective way of reducing energy usage, the objective of the algorithm is to minimize the number of bins used. The algorithm fills bins (PMs) one-at-a-time, with each bin being closed when no remaining VMs can fit inside. Pheromone is deposited on Item-Bin pairs, with VMs being linked to the specific PM to which they are allocated. The heuristic is based on the total resource utilisation of the PM if the current VM were to be assigned to it, and pheromone deposition is based on the average utilisation of all utilised PMs. The Feller ACO technique outperforms First Fit Decreasing (FFD) in terms of energy usage, saving 4.1% on average. However, execution time for the algorithm is significant, ranging from 37.46 seconds for 100 VMs to 2.01 hours for 600 VMs.

A more recent ACO-based VMP algorithm is OEMACS [27]. This adds two Local Search procedures: an *exchange* procedure similar to a local search procedure used for Bin Packing Problems [4], which swaps VMs between PMs in an attempt to find a more efficient configuration, and an *insertion* procedure, which attempts to remove a VM from one PM and insert it into another. OEMACS outperforms [18] in terms of both solution quality and execution time. We therefore compare our ACO algorithm against OEMACS, as it stands as a representative modern ACO algorithm for the VMP.

To summarize, we select OEMACS and IGA-POP as representative state of the art ACO and GA solvers for the VMP, along with a standard heuristic, First-Fit.

3. Our ACO algorithm for VMP

3.1. Overview of ACO

As we base our algorithm on ACO, we now provide a brief overview of its operation. The first ACO algorithm (Ant System) was described by [16]. Many variants and applications of the algorithm have since been developed; however, the key features of ACO, shared by all variants, are that the algorithm uses a number of *agents* (ants) which independently construct solutions guided by a global *pheromone matrix* data structure (and, in some cases, a problem-specific heuristic). The representation of a solution typically takes the form of a subset of edges of a graph, and the *solution construction* phase of the algorithm involves each ant iteratively traversing the graph, building a feasible solution by selecting from the available edges at each step. Edges are selected using a random choice, weighted by the pheromone and heuristic values associated with the available edges. The pheromone update phase of the algorithm associates higher pheromone values with edges which are included in objectively “good” solutions, and to evaporate pheromone from older, less successful, solution components. In what follows, we base our algorithm on the *MAX-MIN* Ant System (MMAS) ACO variant [41].

Due to the inherently distributed nature of ACO (many “ants” effectively act independently, informed by their environment), parallelizing the algorithm is a well-researched topic. Early implementations of parallel ACO made use of distributed systems [7, 10]. Later work on distributed ACO systems included the use of agent-based systems [23, 24, 25, 26], decentralized algorithms [28] and, most recently, desynchronized parallel ACO has demonstrated scalability on up to 400 compute nodes [39].

The development of Nvidia’s Computed Unified Device Architecture (CUDA) framework led to a significant number of GPU-based implementations [8, 9, 38]. The CUDA framework gives access to the powerful parallelization architecture offered by GPUs for graphics processing, which may be utilised for other purposes (this is known as General-Purpose computing on Graphics Processing Units (GPGPU)). Before CUDA, distributed systems were the only realistic method of parallelizing an algorithm, but CUDA allows for parallelization to be performed on a single machine. The rising thread count on modern processors, along with the availability of Single Instruction Multiple Data (SIMD) vector operations such as the AVX512 instruction set on Intel Xeon Phi and Xeon processors, has also increased the viability of parallel ACO on CPUs [11, 17, 29, 43, 44]. SIMD is a class of parallel computing in which the same operation is performed on multiple data points simultaneously, allowing multiple operations to be performed in parallel. In the case of AVX512 and the earlier AVX2 instruction set, 16 and 8 operations respectively may be performed simultaneously. These instructions may be applied to code that is already parallelized, essentially reducing the number of operations required by up to a factor of 16, allowing for an even deeper level of parallel processing.

While running ants in parallel across a graph seems like a straightforward task, certain aspects of the ACO algorithm make parallelization difficult. In particular, the fundamental

roulette-wheel selection technique used by ACO, where different paths are allocated a “slice” of a roulette wheel that is proportional to their favourability, is not amenable to parallelization. In order to overcome this issue, Cecilia *et al.* [8] developed a new data parallel approach to ACO edge selection, known as *I-Roulette*. While the exact proportionality between probability and edge weights is not fully maintained, I-Roulette still accurately replicates the behaviour of Roulette Wheel selection in a parallel-compatible manner. I-Roulette was later adapted into vRoulette [29], which made use of the AVX512 vector instructions to further increase the efficiency of the algorithm.

In what follows we describe an alternative ACO algorithm for VMP which dramatically reduces run-time by harnessing the techniques described above, along with modern processor features.

3.2. Data structures and algorithm design

In this Section we describe our algorithm for Parallel Ant Colony Optimization for Virtual Machine Placement (PACO-VMP), which combines a standard ACO variant with existing parallelization techniques and SIMD vector operations. Our method is based on the *MMAS* variant of ACO, as this is the version that is most amenable to parallelization (due to the absence of communication between ants during an iteration). We have made complete reference code available online¹. Our notation is defined in Table 1, a high-level pseudocode description of the method is supplied in Algorithm 1, and a detailed description of the algorithm is given in Figure 4.

When constructing a solution, ants visit each virtual machine in turn and allocate it to a physical machine. The graph representation of the solution is therefore a complete bipartite graph between the two sets of vertices representing the virtual and physical machines respectively. Solution construction corresponds to the ants selecting N_{VM} edges from this graph, with one edge attached to each of the N_{VM} vertices on one side of the graph representing the virtual machines. The graph is illustrated schematically in Figure 2.

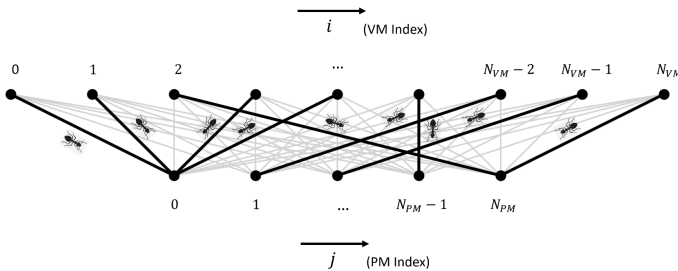


Figure 2: Schematic representation of the solution graph and the solution construction process carried out by the ants. Edges represented by bold lines correspond to edges in the solution, possible edges are in light grey.

The key data structures used by the algorithm are:

1. The *pheromone matrix*, a square ($N_{VM} \times N_{VM}$) matrix of floating point numbers which holds the pheromone values τ_{ij} corresponding to placing two VMs, i and j in the same PM in a solution.
2. An array of *ant* data structures, which each contain a representation of a *solution* (an array of arrays of integers, which lists the indices of the VMs assigned to each PM).

At the highest level of description, the algorithm proceeds as shown in Algorithm 1; each phase of the algorithm is discussed in detail in subsections which follow.

Algorithm 1 High-level description of proposed algorithm

Initialization Phase

for each Iteration **do**

for each Ant **do**

 // Solution Construction

 Randomize order of VM list

for each VM in VM list **do**

 // allocate VM to PM

 Calculate pheromone and heuristic for each PM

if placement feasible **then**

 Place VM with weighted random selection

else

 Allocate VM to PM with most available space

end if

end for

end for

 Apply Local Search to iteration-best Ant

if Iteration-best solution is feasible **then**

 Update Global Best Solution

end if

 Pheromone Update

end for

3.2.1. Initialization Phase

In this phase, the parameters and structures required by the algorithm are created and initialized. An important first step is to ensure that any arrays used for vectorized computations are *padded* correctly, which prevents errors when they are loaded into vectors. As our implementation uses the Intel AVX2 instructions, which operate on 8 32-bit values at a time, the size of each array must be a multiple of 8. The arrays also need to be *aligned* in memory in order to be correctly loaded into AVX2 vectors. The pheromone matrix is a matrix of size $N_{VM} \times N_{VM}$, with N_{VM} being the number of Virtual Machines in the problem instance. The values of the pheromone matrix are initially set to $\tau_0 = 1/N_{PM}$, where N_{PM} is the number of Physical Machines. In this phase we also set the value of the *MMAS* constant a (see Equation 11), which is later used to determine the maximum and minimum pheromone values. The number of PMs is initially set to be equal to the number of VMs.

3.2.2. Solution Construction

The first step of the solution construction phase is to randomly shuffle the VMs. This happens at the beginning of each

¹<https://github.com/jnpeake/PACO-VMP>

iteration in order to prevent VMs being allocated to the same PM purely due to their position in the array. OpenMP is used to allocate each ant's construction process to a separate thread. As each ant only *reads* from global pheromone memory during the construction phase, and does not write to memory, synchronisation is not required. During the construction phase, the ants loop through every VM and allocate it to a PM, unless the current VM does not fit in any remaining PM. Any VMs left un-allocated at the end of the loop are then allocated to the PM with the most available capacity, creating an *infeasible* solution. A Local Search procedure, fully described in a later section, is then applied to the solution in an attempt to make it *feasible*.

	0	1	2	3	4	5	6	7
Heuristic	0.02	0.02	0.07	0.03	0.04	0.12	0.35	0.1
	x							
Pheromone	0.06	0.05	0.09	0.08	0.06	0.03	0.02	0.07
	x							
Random	0.1	0.6	0.4	0.6	0.3	0.5	0.9	0.2
	=							
Total	0.00012	0.0006	0.00252	0.00144	0.0072	0.0018	0.063	0.00014

Figure 3: A demonstration of how the vRoulette-1 technique combines the heuristic and pheromone values of a PM with a random number between 0 and 1. AVX2 instructions allows operations to be carried out on each Vector lane (numbered 0-7) simultaneously.

The selection procedure used to allocate VMs is based on the vRoulette-1 technique developed by Lloyd & Amos [29]. This is illustrated in Figure 3, which shows how the Heuristic and Pheromone values (which we describe in detail later) of the PM in each vector lane are combined with a random number between 0 and 1. This is then multiplied by a *Tabu* value, which is set to 0 or 1 (the value is only set to 0 in the instance that the “PM” in that lane is actually just a placeholder used to pad the PM list to a multiple of 8), and then masked by vectors (denoted *MaxCPUMask* and *MaxRAMMask*) that filter out any PMs that do not have enough available capacity for the current VM. This is done on a vector-by-vector basis, with 8 PMs being processed for selection in parallel. The 8 current PM values are compared lane-by-lane with a vector of the highest PM values in the current selection process. Once every PM has been processed, a parallel reduction (with the *max* operator) is carried out on this vector and the PM corresponding to the highest value is then assigned to the current VM. If the highest value is lower than 0, this indicates that no PMs had enough capacity available for the current VM, and the VM is added to the unassigned list, to be allocated once the solution construction procedure has been completed.

Relating Figure 3 to the notation of Table 1, the vector of heuristic values is populated using the values $\eta_{ij} \dots \eta_{i,j+8}$ for some PM index j . The pheromone vector contains $\tau_{ij} \dots \tau_{i,j+8}$, and the final vector produced, labelled ‘total’ contains $W_{ij} \dots W_{i,j+8}$.

While the original vRoulette-1 implementation made use of the AVX512 instruction set, which allows for 16-wide vectors and features additional instructions compared to AVX2, it is

not currently as widely-available as the AVX2 instruction set, which is available on most Intel CPUs released since 2013, and most AMD CPUs released since 2015. For the implementation evaluated here, we used AVX2.

3.2.3. Heuristic & Pheromone Definition

As with any ACO implementation, the definition of the pheromone and heuristic values is crucial for the consistent construction of good-quality solutions. The heuristic is a problem-specific value which indicates the favourability of assigning a VM to a PM. The definition of the heuristic value can differ significantly, even within ACO implementations for the same problem. A key feature of the process for calculating the heuristic value for VMP is the need for a *dynamically* calculated heuristic, which differs depending on the current state of the PM that is being assigned to. This requires the heuristic to be calculated at every step of the solution for every VM, which increases the solution time compared to static heuristic computation for problems such as Traveling Salesman.

Our heuristic definition is designed to ensure that the lowest possible number of VMs is used, by prioritising both resource utilisation *balance* and *total resource utilisation*. Resource balancing attempts to keep the available RAM and CPU levels on a PM as even as possible. The aim is to prevent PMs exhausting one resource capacity while still having a large available capacity on the other resource. The prioritisation of total utilisation makes it more likely that an ant will allocate the current VM to a PM that already contains other VMs.

First, we define f_C and f_R , the fractional usage of CPU and RAM of PM j if VM i were added to it, as

$$f_C = \frac{C_j^{\text{used}} + C_i^{\text{req}}}{C_j^{\text{cap}}} \quad (1)$$

and

$$f_R = \frac{R_j^{\text{used}} + R_i^{\text{req}}}{R_j^{\text{cap}}} \quad (2)$$

Here, C_j^{used} and R_j^{used} are, respectively, the current CPU and RAM usage of physical machine j , C_i^{req} and R_i^{req} are the CPU and RAM requirements of virtual machine i , and C_j^{cap} and R_j^{cap} are the CPU and RAM capacities of physical machine j . We then define the heuristic value, η_{ij} , associated with placement of virtual machine i on physical machine j as

$$\eta_{ij} = \frac{1 - |f_C - f_R|}{1 + f_C + f_R} \quad (3)$$

Implementations of ACO for VMP generally use one of two definitions of “pheromone trail”; the first defines the trail as connecting VMs and the PMs to which they are allocated, and the second defines trails as being connections *between VMs that are allocated to the same PM* (meaning that VMs are more likely to be allocated to a PM alongside VMs that they have previously sat alongside in good solutions). Here, we use the second definition, where the pheromone trail associates VMs

with other VMs. The pheromone distributed is based on solution quality, which in our case is measured in terms of *energy consumption*.

As the selection process of our algorithm attempts to allocate VMs to PMs, we are unable to load pheromone information directly from the matrix, as pheromone is distributed between VMs rather than between VM and PM. Instead, we calculate the mean value of pheromone linking the current VM and the VMs that are currently allocated to the PM under evaluation (the amount of pheromone between VM and PM is initially set to τ_0 , and remains at that level until a VM is added to the PM).

Therefore, the pheromone associated with a physical machine j when placing a virtual machine i is given by

$$T_{ij} = \begin{cases} \tau_0, & \text{if } N_{VM}^j = 0 \\ \frac{1}{N_{VM}^j} \sum_k \tau_{ik}, & \text{otherwise} \end{cases} \quad (4)$$

where N_{VM}^j is the number of VMs already assigned to PM j , and the sum is taken over all VMs k which are assigned to PM j .

Finally, the weight associated with a particular choice of PM, j , for a given VM, i , during the solution construction phase is determined by combining the pheromone and heuristic values according to

$$W_{ij} = T_{ij}^\alpha \eta_{ij}^\beta \quad (5)$$

where α and β are parameters controlling the relative influence of pheromone and heuristic information. The choice of PM is then made with probability p , which is proportional to W_{ij} .

3.2.4. Local Search

Local Search is a procedure that takes a candidate solution generated by an optimisation algorithm and applies small perturbations to it in order to find a local minimum with respect to some neighbourhood. The term ‘‘Local Search’’ refers to a vast array of usually problem-specific techniques for making these small changes. Local Search techniques are widely used in conjunction with ACO to good effect, and they are essential for creating solutions that are optimal or near-optimal. Local Search takes place after the solution construction phase, once the iteration-best solution has been determined. Following [27], we perform local search only on the iteration-best solution. However, we note (from our experimental results) that the time taken by the local search phase was relatively short (less than 1% of execution time on our largest instances), and that local search could (in principle) be parallelized so that each ant performs this step on its own thread. We followed [27] for a fair comparison, but note that in future work we may obtain further improved solution performance by conducting local search on all candidate solutions.

Our Local Search is based on a technique developed by Alvim *et al.* [4] for the Bin Packing Problem, and also utilised by Liu *et al.* [27] for the VMP. In this algorithm, after each solution is found, one bin is destroyed. If a subsequent solution is then able to successfully fit all items in the remaining bins, it is considered *feasible*. However, if no feasible solution can

Table 1: List of symbols and notations used in this paper. Model Quantities refers to symbols used in the definition of the model for the problem, ACO quantities are values calculated and used during the ACO iterations, while ACO Parameters are fixed values throughout an ACO run.

Model Quantities	
C_j^{cap}	Total CPU capacity on PM j
C_i^{req}	CPU requirement of VM i
C_j^{used}	Current CPU usage on PM j
f_C	CPU usage ratio for current PM
f_R	RAM usage ratio for current PM
N_s	The number of PMs ants are able to use
N_{gb}	The number of PMs used in S_{gb}
N_{ib}	The number of PMs used in S_{ib}
N_{PM}	The number of PMs in the current instance
N_{sol}	The number of PMs used in a solution (all algorithms)
N_{VM}	The number of VMs in the current instance
P	Power usage of current solution
P_{gb}	Power usage of the global best solution
P_{ib}	Power usage of the iteration best solution
P_j^{idle}	Idle power usage of PM j
P_j^{max}	Maximum power usage of PM j
R_j^{cap}	Total RAM capacity on PM j
R_i^{req}	RAM requirement of VM i
R_j^{used}	Current RAM usage on PM j
S_{gb}	The global best solution
S_{ib}	The best solution from the current iteration
ACO Quantities	
η_{ij}	Heuristic value between VM i and PM j
i_{cur}	The current VM
j_{cur}	The current PM
k	Current iteration number
N_{VM}^j	Number of VMs assigned to PM j
τ_{ij}	Pheromone value between VMs i and j
τ_0	The initial pheromone value
τ_{max}	Maximum pheromone value
τ_{min}	Minimum pheromone value
T_{ij}	Averaged pheromone value between VM i and PM j
W_{ij}	Weight associated with PM j for VM i
ACO Parameters	
α	Pheromone influence
β	Heuristic influence
ρ	Pheromone decay rate
a	MMAS constant value
k_{max}	Maximum number of iterations permitted
K_P	Scaling constant for power usage in pheromone calculation

be found, the local search technique is applied. There are two phases of our technique, the *swap* phase and the *insertion* phase. Any PM that has been allocated more VMs than it has capacity for is marked as *overloaded*. In the swap phase an overloaded PM is compared with every non-overloaded PM, and the algorithm attempts to swap each VM in the overloaded PM with each VM in the non-overloaded PM. This continues until either a successful swap takes place, or every non-overloaded PM has been compared to the overloaded PM. Regardless of the outcome, the process is carried out again for the next PM, and this continues until every overloaded PM has been compared. If the swap phase is unable to successfully find a feasible solution, the insertion phase is then performed. In this phase, each

overloaded PM attempts to allocate each of its VMs to a non-overloaded PM. While this is far less likely to produce positive results than the swap phase, it is still able to occasionally make progress when the swap phase cannot.

3.2.5. Pheromone Update

The final phase of our ACO algorithm is the deposition of pheromone. As our algorithm is based on the *MMAS* ACO variant, pheromone is deposited only by the global-best ant. As mentioned previously, pheromone is distributed on edges connecting VMs allocated to the same PM. The pheromone matrix is updated using

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{K_P}{P} \quad (6)$$

where K_P is a constant, for all pairs of VMs i, j which are allocated to the same PM in the global best solution, and where P is the power usage of the solution. In our runs we used $K_P = 365$. In practice, this is a constant which scales power usage into the typical range of values for τ_{\min} and τ_{\max} , defined below. The actual value of this parameter may be varied depending on the units used for power in the instance definition, and can be considered a hyper-parameter of the method.

The power usage is defined as

$$P = \sum_{j=1}^{N_{PM}} \left((P_j^{\max} - P_j^{\text{idle}}) \frac{C_j^{\text{used}}}{C_j^{\text{cap}}} + P_j^{\text{idle}} \right) \quad (7)$$

where N_{PM} is the number of PMs in the current instance and P_j^{\max} and P_j^{idle} are the maximum and idle power usage of physical machine j respectively. We choose a definition of pheromone based on power usage, as this reflects the positive impact of a lower number of PMs while still measuring differences between solutions with the same number of PMs used. The global amount of pheromone then decays by a static amount, controlled by the parameter ρ ,

$$\tau_{ij} \leftarrow \tau_{ij}(1 - \rho) \quad \forall i, j \in [1, N_{VM}]. \quad (8)$$

The choice of value of ρ will be discussed in Section 4.

MMAS utilises a clamping procedure to prevent stagnation, by restricting the level of pheromone to be between maximum and minimum values. The maximum and minimum values are defined as

$$\tau_{\max} = \frac{1}{\rho N_{\text{best}}^{\text{global}}} \quad (9)$$

$$\tau_{\min} = \tau_{\max} \frac{2(1 - a)}{(N_{VM} + 1)a} \quad (10)$$

where N_{VM} is the number of VMs in the current instance, ρ is the evaporation parameter and

$$a = \exp(\ln(0.05)/N_{VM}). \quad (11)$$

This clamping is applied to the whole matrix after evaporation.

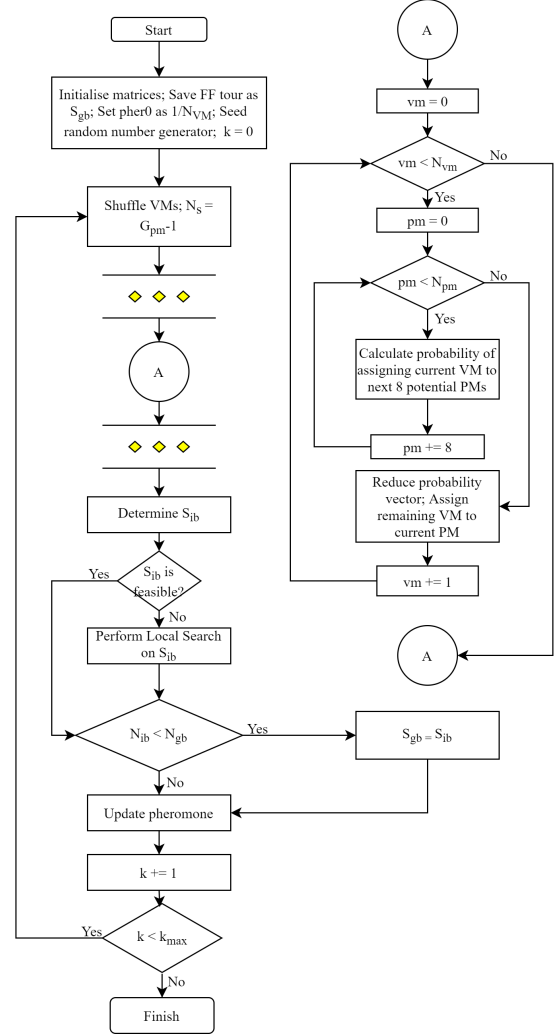


Figure 4: A flow chart detailing the PACO-VMP algorithm. Section A, between the yellow diamonds, is executed in parallel for each ant.

4. Experimental Evaluation

We investigate the performance of our algorithm by comparing it with an implementation of the OEMACS algorithm (which is an ACO-based method that generally out-performs conventional heuristics and evolutionary algorithms for this problem [27]), and a state-of-the-art genetic algorithm, IGA-POP [1]. Code for OEMACS is publicly available². All algorithms were implemented in C++, and all tests were carried out on a machine with an Intel® Xeon E5-2640 v4 processor with 20 cores running at a base frequency of 2.4 GHz and a maximum frequency of 3.4 GHz. Code was compiled using the GNU C++ compiler (g++), with *O2* optimization enabled. The initial comparative tests compare a serial implementation of PACO-VMP with the serial algorithms OEMACS and IGA-POP, with the aim of comparing the *quality* of solutions obtained. Two variants of IGA-POP are used: the first, referred to as GA1, uses the fitness function also used by PACO-VMP and OEMACS;

²<https://github.com/Budding0828/OEMACS>

the second, referred to as GA2, uses a slightly modified version of the fitness function used in the initial IGA-POP experiments [1]. Finally, to evaluate the impact of our OpenMP parallelization, we also run a parallelized PACO-VMP using OpenMP, assigning one ant to each of our 20 cores. While the execution time differs, solution quality is identical to the serial version.

4.1. Problem instances

All problem instances used in our experiments were randomly generated. For each instance, the initial number of Physical Machines is set to be equal to the number of Virtual Machines. Our dataset consists of three sets of 600 VMP instances, each further split into 6 subsets of 100 instances with 100, 200, 300, 400, 500 and 1000 VMs. The three sets (A, B and C) are described in the following subsections, and differ in terms of either the inclusion of *bottlenecks* in one or other resource, or the *homogeneity* of the physical machines in the instance setup. One run is performed using each instance. We use one run per instance with a larger number of instances, rather than multiple runs on a smaller number of instances; a proof in [5] shows that given a budget of N runs, selecting K instances and performing n runs on each with $N = Kn$ is a sub-optimal choice, and that the best statistical estimate of algorithm performance is obtained from a single run on each of N independently selected instances (contrary to popular belief).

4.1.1. Instance Set A: Homogeneous Environment

Set A is designed to evaluate the performance of the algorithms in a straightforward scenario where the PMs are identical and the demands of the VMs are fairly evenly divided between RAM and CPU. This set consists of 600 VMP instances equally divided between 100, 200, 300, 400, 500 and 1000 VM instances. This dataset is similar to the Set A data used by Liu *et al.* [27] in evaluating OEMACS, which was initially created to benchmark the Reordering Grouping Genetic Algorithm (RGGA) [46]; however, this data is no longer publicly available. In comparison to this dataset, we used a larger number of smaller instances.

The VM requirements for this instance set are randomly generated in ranges of [1,128] for CPU and [1,100] for RAM. Each PM has a capacity of 500 for both CPU and RAM, leading to slightly higher average CPU utilisation compared to RAM utilisation (but still close to 1:1). As these instances are randomly generated, there is no known optimum, but a lower limit to the number of PMs used in the solution, N_{\min} , may be calculated:

$$N_{\min} = \max \left\{ \frac{\sum_{j=1}^{N_{\text{VM}}} C_j^{\text{req}}}{C_i^{\text{cap}}}, \frac{\sum_{j=1}^{N_{\text{VM}}} R_j^{\text{req}}}{R_i^{\text{cap}}} \right\} \quad (12)$$

where i is the index of any physical machine; as the servers in Instance Set A are homogeneous, it does not matter which physical machine is used to evaluate this quantity.

4.1.2. Instance Set B: Homogenous Environment with Bottleneck

Set B introduces a bottleneck resource to the problem instances, evaluating the performance of the algorithms in a slightly more complicated scenario which will lead to more overloaded servers. As with the previous instance set, Set B consists of 600 VMP instances equally divided between 100, 200, 300, 400, 500 and 1000 VM instances. VM requirements are randomly generated, in the range [1-4] for CPU (measured in cores) and [1-8] for RAM (measured in GB). PM capacity is 16 cores for CPU and 32GB for RAM. As the probability of a 4 core VM requirement is higher than the probability of an 8GB RAM requirement, CPU is the bottleneck resource. As with Set A, these instances have no known optimum, and a lower limit is calculated using the same formula.

4.1.3. Instance Set C: Heterogeneous Environment with Bottleneck

Set C further complicates the problem instances by introducing non-identical servers, simulating a scenario in which a cloud host has multiple server types. We define two types of server, *A* and *B*. Server type *A* has a CPU capacity of 16 cores and a RAM capacity of 32GB. Server type *B* has a CPU capacity of 32 cores and a RAM capacity of 64GB. However, type *B* servers only make up 10% of the total PMs in each problem instance, meaning that VMs will have to use both types of servers. This will evaluate the ability of the algorithms to prioritise the high capacity servers while still allocating the VMs efficiently. The VM requirements are in the range [1,8] for CPU and [1,32] for RAM, meaning that the bottleneck resource in this case is RAM. Set C utilises the same instance sizes as the previous sets.

Due to the heterogeneous servers in this instance set, an alternative formula is required for calculating the lower limit to the number of PMs:

$$N_{\min} = N_B + \max \left\{ \frac{\sum_{j=1}^{N_{\text{VM}}} C_j^{\text{req}} - N_B C_B^{\text{cap}}}{C_A^{\text{cap}}}, \frac{\sum_{j=1}^{N_{\text{VM}}} R_j^{\text{req}} - N_B R_B^{\text{cap}}}{R_A^{\text{cap}}} \right\} \quad (13)$$

where N_B is the number of type *B* servers, C_A^{cap} and C_B^{cap} are the CPU capacities of type *A* and *B* servers respectively, and R_A^{cap} and R_B^{cap} are the RAM capacities of type *A* and *B* servers respectively.

4.2. Experimental configuration

For each experiment we use 20 ants for PACO-VMP, as this allows one ant to be allocated to each core available on our hardware in the OpenMP-enabled variant. For the ACO parameters used by PACO-VMP, we select values of $\rho = 0.8$, $\alpha = 1$, $\beta = 6$ on the basis of tuning experiments on a small sample of 1000 VM instances (although we found the performance was generally insensitive to these parameters). For OEMACS we use the default values as specified in [27]. Both PACO-VMP and OEMACS are run for 50 iterations. We chose this value for fair comparison with the results in [27], which used this number of iterations, although this is a relatively small number of iterations for ACO. However, in practice we find that the solutions

do converge within this limit (see Section 4.4), but note that it may be possible to find better solutions by running over more iterations and different values of the algorithm parameters. For GA1 and GA2, we use the parameters specified in [1]; 200 iterations and a population size of the number of PMs multiplied by 4. We compare these results against the First Fit (FF) algorithm in order to provide a baseline greedy algorithm implementation. FF was selected over the more widely-used FFD algorithm due to it obtaining better results on our datasets. It should be noted that the solution construction time for FF is near-instantaneous for all instance sizes, and has therefore been omitted from all execution time plots.

4.3. Results

We compare the performance of the algorithms against two metrics, solution quality and execution time. We measure solution quality as the number of physical machines expressed as a percentage excess over the minimum number possible for the instance, N_{min} , i.e. $100(N_{sol}/N_{min} - 1)$, where N_{sol} is the number of non-empty physical machines in a solution. Execution time is measured as *wall clock time* on a machine which is otherwise idle apart from the experimental run.

The results for instance Set A in terms of solution quality are shown in Figure 5. FF shows good results throughout, improving as the problem instances get larger, which indicates that it is fairly straightforward for a greedy solver to create good-quality solutions for the non-bottlenecked version of the VMP problem. In all but one instance, OEMACS is able to match or exceed the solutions created by FF. Likewise, PACO-VMP outperforms or matches OEMACS on 5 sizes of instances, including the largest instances. We note that the PACO-VMP algorithm utilises the FF result as its initial best tour, so it is not able to find worse tours than FF. A distinction between the results of set A and our other instance sets is that FF is competitive with the two ACO algorithms. For our other instance sets this is not the case, but as the non-bottlenecked problem is fairly straightforward, it allows FF to find good quality solutions. GA2 also performs well on this dataset, outperforming PACO-VMP on all but a single dataset. On the other hand, GA1 struggles, remaining moderately competitive for the smaller instances, but performing dramatically worse on the 400, 500 and 1000 VM instances.

Execution time results for instance Set A are shown in Figure 6. It is clear that PACO-VMP has a significant advantage over OEMACS when it comes to execution time, beginning at around 1 order of magnitude for the size 100 instances, and increasing to an advantage of around 3 orders of magnitude for the 1000 VM instance sets. An even larger advantage is demonstrated over the two IGA-POP algorithms, beginning at around 2 orders of magnitude for the 100 VM instances and increasing to around 3 orders of magnitude for the 1000 VM instances. Interestingly, despite beginning with a sizeable time advantage over IGA-POP, OEMACS performs similarly to GA1 for the 1000 VM instance set. The gap between the sequential and parallelized versions of PACO-VMP grows from a speedup of 2.2 \times for the 100 VM instances, to a speedup of 3.47 \times for 1000 VM instances.

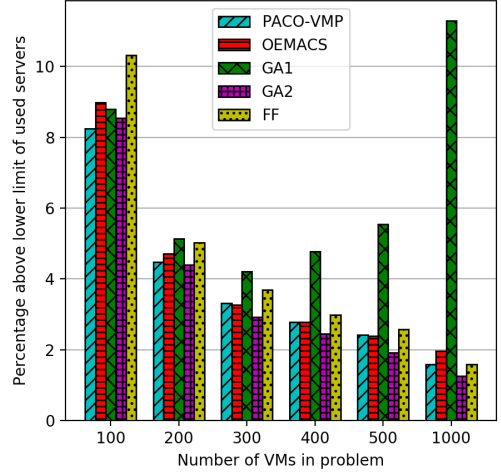


Figure 5: Solution difference measured as percentage over theoretical optimum for PACO-VMP, OEMACS, GA1, GA2 and FF for instance set A.

Unlike instance Set A, the results for instance Set B (shown in Figure 7) reveal a clear difference between PACO-VMP and OEMACS. FF’s poor results also indicate that a greedy solver has more difficulty in finding a good solution for the bottlenecked VMP than for the non-bottlenecked variant. While OEMACS significantly outperforms FF, PACO-VMP outperforms it for every problem size, finding solutions that range from 1%-2% closer to the theoretical lower limit. Additionally, solution quality is actually worse for the size 1000 instances with OEMACS, whereas PACO-VMP continues to improve. In contrast to Set A, GA1 performs very well in this bottle-necked scenario, with PACO-VMP returning better results for the 100 VM instances but then returning very slightly worse results for the larger instances. Conversely, GA2 performs poorly, initially returning similar results to OEMACS before worsening on the larger instances, and even being outperformed by FF for the 1000 VM instances.

In terms of execution time (displayed in Figure 8), the results for PACO-VMP are near-identical to the results for instance Set A, demonstrating that the bottleneck leads to no additional execution time. This is also the case for OEMACS, which also achieves near-identical execution times to the instance set A results. The execution time advantage held by PACO-VMP is maintained, with OEMACS once again losing the advantage it holds over IGA-POP as the solution size increases. The difference between sequential and parallel PACO-VMP is also near-identical to instance Set A, though the speedup increase is slightly smaller (from 2.2 \times to 3.27 \times).

The results shown in Figure 9 indicate that FF performs very poorly on instance Set C, with the heterogeneous servers causing issues for the greedy technique. OEMACS significantly outperforms FF once again, but is itself outperformed by PACO-VMP, with solution quality improvement ranging from 5% for 100 VM instances to 10% for 1000 VM instances. While OEMACS performs significantly worse on instance Set C than

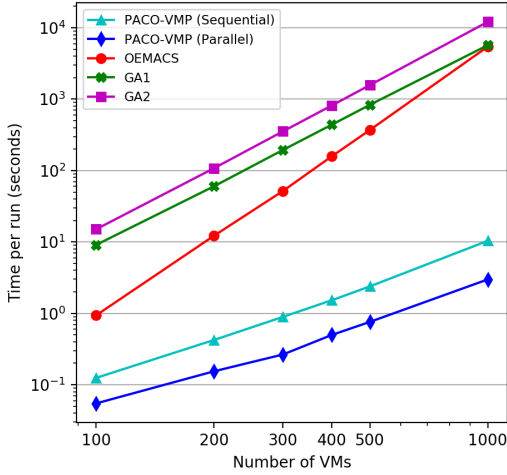


Figure 6: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set A.

on the other sets, PACO-VMP is able to capably solve the heterogeneous instances. As with instance Set B, while OEMACS begins to return worse solution qualities for the size 1000 instances, PACO-VMP continues to improve as the instance size increases. The performance of the GA variants is also consistent with results on Set B, with GA1 slightly outperforming PACO-VMP in all but one instance size, and GA2 performing poorly, showing even poorer results on instance Set C.

Execution times for instance Set C (Figure 10) are similar to the other instance sets, with the execution time of PACO-VMP being near identical. However, OEMACS takes slightly longer to solve the instances in Set C, further increasing the execution time advantage held by PACO-VMP. Additionally, the execution time of OEMACS is now closer to the time of GA2 than GA1 for 1000 VM instances, emphasising the increased difficulty that OEMACS has when trying to solve the bottlenecked, homogeneous problem. As with the previous instance sets, the time difference between the sequential and parallel PACO-VMP increases slightly as the instance sizes increase, from $2.6\times$ to $3.78\times$.

4.4. Discussion

Through the use of parallelization and vectorization techniques, we have demonstrated a significant execution time reduction for our ACO-based algorithm for VMP, demonstrated on a wide range of different problem instance sets that represent three realistic Cloud Computing scenarios. PACO-VMP outperforms OEMACS in each instance set; very slightly on Set A, and significantly on Sets B and C. While it is matched by GA1 on Set B and C, it performs significantly better on instance Set A. The opposite is true of GA2, which outperforms PACO-VMP on Set A, but significantly underperforms on Sets B and C. This consistency demonstrates the versatility of ACO compared to IGA-POP; while IGA-POP performs well, it requires two separate fitness functions in order to match PACO-

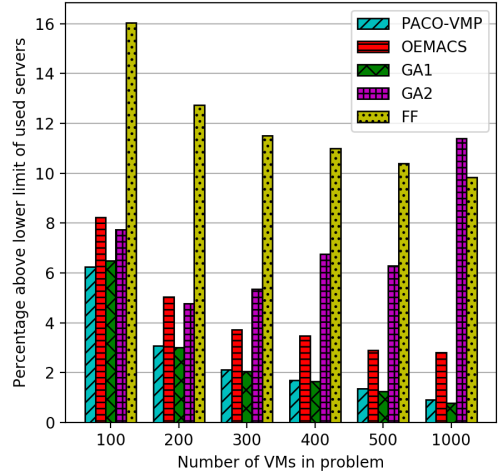


Figure 7: Solution difference measured in percentage over theoretical optimum for PACO-VMP, OEMACS, GA1, GA2 and FF for instance set B.

VMP. Further analysis of the IGA-POP results reveals that the issue stems from the tendency of the algorithm to assign VMs to empty PMs even when currently used PMs have enough capacity remaining, which happens regardless of fitness function. Both PACO-VMP and OEMACS enforce a limit on the number of PMs that can be used (the previous best number of PMs) which prevents this behaviour. Additionally, it is worth noting that despite a 10x increase in instance size across our experiments, the *quality* of the solutions produced by PACO-VMP remains consistent. The percentages above the lower limit of PM utilization decrease with each increase in instance size, which (in terms of raw numbers) indicates a fairly consistent number of PMs over the minimum. This suggests that our implementation could still produce good results for even larger VMP instances. This is an advantage over OEMACS, where solution quality degrades for size 1000 instances, a trend which would potentially continue as instance sizes increase.

The main focus of PACO-VMP is to improve execution time, and it succeeds at this objective. While PACO-VMP and OEMACS use similar pheromone definitions and local search techniques, PACO-VMP produces better results, both in terms of execution time and solution quality. This may be due partly to the choice of *MMAS* algorithm over ACS, and also explained by differences in selection probabilities due to the use of independent roulette. It has been shown [30] that independent roulette algorithms (such as *vRoulette*) tend to make greedier selections than the traditional roulette wheel algorithm, which may be a factor explaining the different solution qualities found between PACO-VMP and OEMACS. Clearly the areas in which PACO-VMP and OEMACS differ are significant in terms of execution time, as PACO-VMP has a time complexity of $O(n^2)$, whereas OEMACS is, experimentally, closer to $O(n^4)$. The main contributing factor to this is the probability calculation; whereas PACO-VMP uses the resource wastage formula as given in Formula 1 as the heuristic value, OEMACS uses a much

Table 2: Solution quality results of our experiments on FF, OEMACS, GA and PACO-VMP. Entries in the Set column represent the 100 instances of the specified size from the specific instance set. Solution Quality is the average percentage over the theoretical minimum for all 100 problem instances for each size within each set with values in **bold** being the best result, on the condition that it is significantly different when results are analysed with the Wilcoxon signed-rank test. Errors quoted are the standard deviation over the 100 instances in each set.

Set	Solution Quality (%)				
	FF	OEMACS	GA1	GA2	PACO
A100	10.3 ± 4.0	8.98 ± 2.76	8.80 ± 2.42	8.54 ± 1.92	8.25 ± 3.13
A200	5.03 ± 1.82	4.71 ± 1.60	5.13 ± 2.23	4.4 ± 1.31	4.47 ± 1.46
A300	3.69 ± 1.33	3.26 ± 1.15	4.21 ± 2.45	2.92 ± 0.858	3.32 ± 1.24
A400	2.98 ± 1.00	2.78 ± 0.98	4.78 ± 2.78	2.46 ± 1.04	2.78 ± 1.01
A500	2.58 ± 0.75	2.39 ± 0.78	5.54 ± 3.57	1.92 ± 0.672	2.42 ± 0.81
A1000	1.58 ± 0.31	1.96 ± 0.39	11.3 ± 4.2	1.26 ± 0.479	1.58 ± 0.31
B100	16.0 ± 3.7	8.24 ± 3.33	6.49 ± 1.41	7.75 ± 5.57	6.24 ± 1.90
B200	12.7 ± 2.2	5.05 ± 2.31	3.01 ± 1.02	4.78 ± 4.11	3.08 ± 1.04
B300	11.5 ± 1.5	3.72 ± 1.35	2.05 ± 0.47	5.36 ± 4.12	2.11 ± 0.62
B400	11.0 ± 1.4	3.47 ± 1.24	1.64 ± 0.384	6.77 ± 4.51	1.69 ± 0.41
B500	10.4 ± 1.1	2.89 ± 1.06	1.25 ± 0.35	6.30 ± 3.88	1.37 ± 0.52
B1000	9.83 ± 0.72	2.82 ± 0.68	0.79 ± 0.29	11.4 ± 4.02	0.91 ± 0.32
C100	34.8 ± 14.1	13.7 ± 4.7	6.96 ± 2.29	23.9 ± 7.79	7.67 ± 2.89
C200	31.6 ± 10.5	11.9 ± 2.6	5.03 ± 1.18	31.4 ± 6.25	6.10 ± 1.46
C300	32.1 ± 9.4	11.7 ± 2.4	4.46 ± 0.97	37.2 ± 6.56	5.39 ± 1.09
C400	31.0 ± 7.6	12.1 ± 2.4	4.07 ± 0.84	41.3 ± 5.4	4.66 ± 0.86
C500	29.2 ± 6.7	12.0 ± 2.0	3.93 ± 0.61	43.0 ± 4.45	4.44 ± 0.62
C1000	26.3 ± 5.2	12.8 ± 1.5	3.7 ± 0.48	54.5 ± 3.67	3.62 ± 0.32

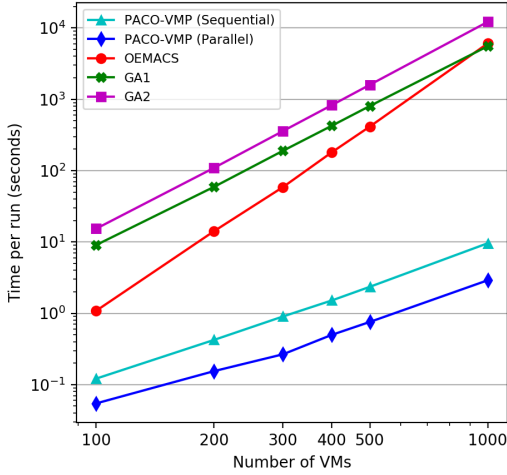


Figure 8: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set B.

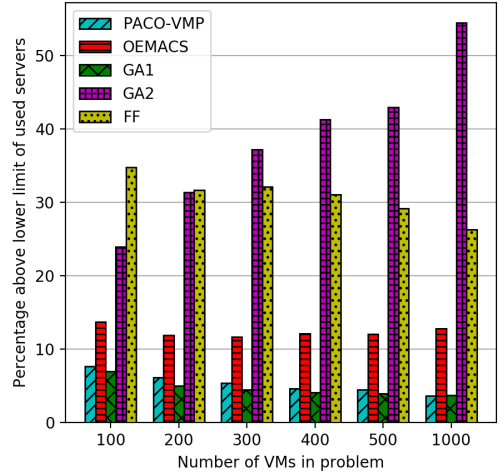


Figure 9: Solution difference measured in percentage over theoretical optimum for PACO-VMP, OEMACS, GA1 and GA2 and FF for instance set C.

more complex formula that includes resource wastage, but also has extra sums over the VMs in both the numerator and denominator of the formula. Experimentally, the time complexity of the IGA-POP variants is approximately $O(n^3)$.

We summarize our results in Tables 2 and 3. These show results for solution quality and execution time; for the solution quality results we indicate which results are statistically significant. The results of each algorithm on each instance of a

given size can be paired, and compared to each other using the Wilcoxon signed-rank test (a non-parametric test which can be used to compare paired sets of readings). Since we perform an all-vs-all comparison of three tests (all possible pairs of algorithms) we apply the Bonferroni correction, and divide the significance threshold by the number of tests (in this case 3). Bold values in the table show solution quality values that are significantly better than the other four algorithms, using a sig-

Table 3: Execution time results from our experiments on OEMACS, GA and PACO-VMP. Entries in the Set column represent the 100 instances of the specified size from the specific instance set. Execution Time is the average time per run in seconds for all 100 problem instances for each size within each set. Errors quoted are the standard deviation over the 100 instances in each set.

Set	Execution Time (seconds)				
	OEMACS	GA1	GA2	PACO(S)	PACO(P)
A100	0.9364 ± 0.0655	9.098 ± 0.166	15.05 ± 0.27	0.1246 ± 0.0217	0.05473 ± 0.00447
A200	12.22 ± 0.60	60.23 ± 0.57	107.9 ± 3.2	0.4234 ± 0.0596	0.1542 ± 0.0052
A300	51.81 ± 1.60	194.4 ± 1.9	354.3 ± 6.4	0.8922 ± 0.0772	0.2653 ± 0.0029
A400	158.8 ± 4.4	440.0 ± 3.8	817.7 ± 4.1	1.528 ± 0.099	0.4989 ± 0.0133
A500	369.8 ± 8.7	832.8 ± 6.0	1575 ± 9	2.387 ± 0.117	0.7591 ± 0.0158
A1000	5451 ± 233	5711 ± 200	12120 ± 310	10.37 ± 0.62	2.986 ± 0.476
B100	1.082 ± 0.116	9.044 ± 0.164	15.32 ± 0.19	0.1211 ± 0.0202	0.05445 ± 0.00463
B200	14.14 ± 0.59	59.15 ± 0.54	109.2 ± 3.1	0.4228 ± 0.0595	0.1544 ± 0.0045
B300	58.75 ± 1.97	189.7 ± 1.6	358.6 ± 3.8	0.9016 ± 0.0871	0.266 ± 0.003
B400	181.1 ± 5.7	426.2 ± 2.4	829.1 ± 8.0	1.514 ± 0.094	0.4988 ± 0.0105
B500	414.8 ± 6.6	805.4 ± 4.4	1596 ± 22	2.351 ± 0.114	0.7561 ± 0.0137
B1000	6080 ± 208	5547 ± 190	12190 ± 280	9.594 ± 0.506	2.904 ± 0.426
C100	1.646 ± 0.242	9.733 ± 0.220	16.98 ± 0.30	0.1349 ± 0.0232	0.0556 ± 0.0049
C200	22.14 ± 3.00	63.66 ± 0.95	116.7 ± 2.6	0.4644 ± 0.0514	0.1576 ± 0.0048
C300	88.56 ± 9.21	205.0 ± 3.1	383.7 ± 7.1	0.9796 ± 0.0841	0.2742 ± 0.0037
C400	270.3 ± 23.0	456.8 ± 4.9	874.7 ± 14.7	1.663 ± 0.0968	0.5112 ± 0.0068
C500	633.0 ± 40.0	858.4 ± 8.0	1665 ± 20	2.623 ± 0.140	0.7709 ± 0.0212
C1000	8874 ± 509	5754 ± 195	12490 ± 430	10.69 ± 0.57	2.874 ± 0.486

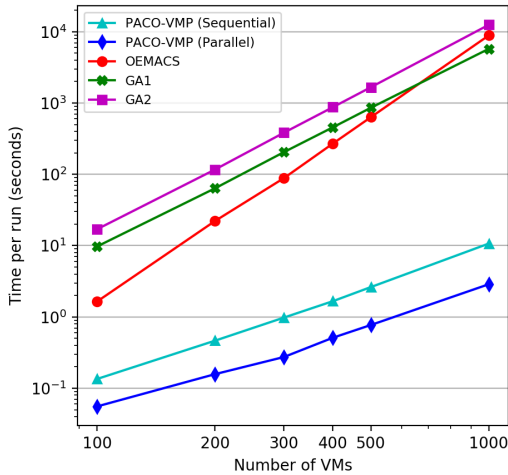


Figure 10: Average time to solve (seconds) for 100 instances of a given instance size for PACO-VMP, OEMACS, GA1 and GA2 for instance set C.

nificance threshold of 0.002 (that is, 0.01 after application of the Bonferroni correction).

In general, one of the two GA versions tends to produce the best solutions; however, although the differences are in many cases statistically significant, the magnitude of the effect is small. For example, in the case of the A1000 instances, a comparison between GA2 and PACO shows that - out of the 100 trials - GA2 is superior for 46 instances, whereas ACO is superior for 3, with 51 ties. Although this is a statistically highly significant differ-

ence, the *magnitude* of the difference is only 0.32% in terms of solution quality. This demonstrates that the experiments are very sensitive in detecting significant, but small, differences in performance.

Qualitatively, the results show that one of the two GAs generally performs the best for any set of instances, but this is often accompanied by the other GA performing the worst. Since we used the GA with the recommended parameters for the population size and number of generations, this performance also comes at a significant cost; for example in the 1000 VM instances, GA1 and GA2 perform 4000 evaluations per generation for 200 generations, compared to 20 evaluations for 50 iterations in PACO. Furthermore, the difference in performance between the two cost functions is very clear; using the original cost function proposed by [1] leads to poor performance on the B and C instance sets. On the other hand, PACO-VMP achieves solution quality close to best (or best) across all instance types, without any sensitivity to the algorithm parameters, and achieves better average solution quality than the other ACO algorithm (OEMACS) in 15 out of the 18 instance categories. There is also a clear advantage for PACO-VMP in terms of both scalability and execution time. The computational complexity of PACO is superior to both OEMACS and GA1/2, and the execution time of the parallel version is several orders of magnitude lower in most cases. For the C1000 instances, the most challenging instance set, PACO-VMP achieves the best solution quality of all algorithms, in an average time of 2.873s, while GA1/2 and OEMACS require several hours of CPU time to reach a solution.

As noted in Section 4.2, the number of iterations used here (50, consistent with [27]) is relatively small for ACO. We found

that convergence was rapid, and the number of iterations was sufficient. Figure 11 shows the mean solution quality averaged over 10 runs of PACO-VMP on instances from the A1000, B1000, C1000 sets. The hardest instances (those from the C set) still show marginal improvement at 50 iterations, but are largely converged. For the easier instances, the solutions clearly converge much earlier than this.

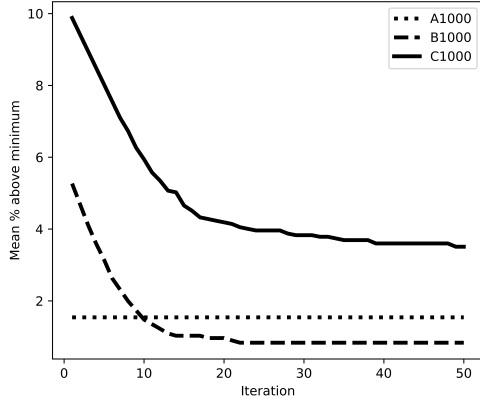


Figure 11: Mean solution quality as a function of iteration averaged over 10 instances each from the A1000, B1000 and C1000 sets, using the PACO-VMP algorithm.

5. Conclusions & Future Work

In this paper we presented PACO-VMP, a parallelized and vectorized implementation of MMAS for solving the Virtual Machine Placement problem. The method is several orders of magnitude faster than two current state-of-the-art ACO solvers (OEMACS and IGA-POP) while producing comparable or superior results. Since virtual machine placement in the real world is a problem in which reducing time to solution can have significant cost benefits, the improved execution time performance of PACO-VMP is potentially important.

While PACO-VMP is capable of solving the static VMP problem, in reality this problem is rarely static. Real-world cloud workloads have constantly changing demand, with Virtual Machines being added and removed from the workload constantly. As with the static VMP, execution time is vital for dynamic VMPs in order to minimise time spent in an inefficient configuration, and PACO-VMP's positive results on the static problem indicate that it could also be effectively used to solve the dynamic problem. This is an area for further investigation.

The parameter tuning phase of our experiments revealed that the performance of the algorithm is relatively insensitive to the parameter governing the importance of pheromone information, further suggesting that analysing and improving our pheromone definition may lead to better solution quality from the underlying ACO mechanism. This is a potentially fruitful area of further work.

Many assumptions were made in our implementation regarding the VMP problem, including that there will always be

as many PMs available as VMs, that performance doesn't degrade when the PMs reach 100% capacity, and that CPU and RAM are the only requirements. These assumptions are commonly made to simplify the problem solving process rather than having to consider a vast array of additional variables. Another potentially fruitful area to investigate is the use of additional parameters for the VMP problem, rather than just CPU and RAM. Further work is required to investigate the inclusion of these additional parameters.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

JP is funded by the Centre for Advanced Computational Science at Manchester Metropolitan University.

References

- [1] A.S. Abohamama and Eslam Hamouda. A hybrid energy-aware virtual machine placement algorithm for cloud environments. *Expert Systems with Applications*, 150:113306, 2020.
- [2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.
- [3] Fares Alharbi, Yu-Chu Tian, Maolin Tang, Wei-Zhe Zhang, Chen Peng, and Minrui Fei. An Ant Colony System for Energy-efficient Dynamic Virtual Machine Placement in Data Centers. *Expert Systems with Applications*, 120:228–238, 2019.
- [4] Adriana Alvim, Fred S Glover, Celso C Ribeiro, and Dario J Aloise. Local Search for the Bin Packing Problem. 1999.
- [5] Mauro Birattari. On the estimation of the expected performance of a metaheuristic on a class of instances. how many instances, how many runs? Technical Report TR/IRIDIA/2004-001, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2004.
- [6] A. Botta, W. de Donato, V. Persico, and A. Pescapé. On the Integration of Cloud Computing and Internet of Things. In *2014 International Conference on Future Internet of Things and Cloud*, pages 23–30, 2014.
- [7] Bernd Bullnheimer, Gabriele Kotsis, and Christine Strauß. Parallelization Strategies for the Ant System. In *High Performance Algorithms and Software in Nonlinear Optimization*, pages 87–100. Springer, 1998.
- [8] José M Cecilia, José M García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing Data Parallelism for Ant Colony Optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.
- [9] José M Cecilia, José M Garcia, Manuel Ujaldón, Andy Nisbet, and Martyn Amos. Parallelization Strategies for Ant Colony Optimisation on GPUs. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 339–346. IEEE, 2011.
- [10] Ling Chen and Chunfang Zhang. Adaptive Parallel Ant Colony Algorithm. In *International Conference on Natural Computation*, pages 1239–1249. Springer, 2005.
- [11] Darren M. Chitty. Applying ACO to Large Scale TSP Instances. In Fei Chao, Steven Schockaert, and Qingfu Zhang, editors, *Advances in Computational Intelligence Systems*, pages 104–118, Cham, 2018. Springer International Publishing.

- [12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.
- [13] EG Coffman Jr, MR Garey, and DS Johnson. Approximation Algorithms for Bin Packing: A Survey. *Approximation Algorithms for NP-hard Problems*, pages 46–93, 1996.
- [14] Jean-Louis Deneubourg, Jacques M. Pasteels, and Jean-Claude Verhaeghe. Probabilistic Behaviour in Ants: a Strategy of Errors? *Journal of Theoretical Biology*, 105(2):259–271, 1983.
- [15] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant Colony Optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.
- [16] Marco Dorigo, Vittorio Maniezzo, and Alberto Colomi. Ant System: Optimization By a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.
- [17] Ivars Dzalts and Tatiana Kalganova. Accelerating Supply Chains with Ant Colony Optimization across a Range of Hardware Solutions. *Computers & Industrial Engineering*, 147:106610, 2020.
- [18] Eugen Feller, Louis Rilling, and Christine Morin. Energy-aware Ant Colony Based Workload Placement in Clouds. In *2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 26–33. IEEE, 2011.
- [19] Damián Fernández-Cerero, Alejandro Fernández-Montes, and Agnieszka Jakóbk. Limiting global warming by improving data-centre software. *IEEE Access*, 8:44048–44062, 2020.
- [20] Yongqiang Gao, Haibing Guan, Zhengwei Qi, Yang Hou, and Liang Liu. A Multi-objective Ant Colony System Algorithm for Virtual Machine Placement in Cloud Computing. *Journal of Computer and System Sciences*, 79(8):1230–1242, 2013.
- [21] J. A. Guerrero-ibanez, S. Zeadally, and J. Contreras-Castillo. Integration Challenges of Intelligent Transportation Systems with Connected Vehicle, Cloud Computing, and Internet of Things Technologies. *IEEE Wireless Communications*, 22(6):122–128, 2015.
- [22] Brian Hayes. Cloud computing, 2008.
- [23] S. Ilie and C. Bădică. A comparison of the island and acoda approaches for distributing aco. In *2013 17th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 757–762, Oct 2013.
- [24] Sorin Ilie and Costin Bădică. Multi-agent approach to distributed ant colony optimization. *Science of Computer Programming*, 78(6):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.
- [25] Sorin Ilie and Costin Bădică. Multi-agent distributed framework for swarm intelligence. *Procedia Computer Science*, 18:611 – 620, 2013. 2013 International Conference on Computational Science.
- [26] A. Kaplar, M. Vidaković, N. Luburić, and M. Ivanović. 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)*, pages 1144–1148, May 2017.
- [27] Xiao-Fang Liu, Zhi-Hui Zhan, Jeremiah D Deng, Yun Li, Tianlong Gu, and Jun Zhang. An Energy Efficient Ant Colony System For Virtual Machine Placement in Cloud Computing. *IEEE Transactions on Evolutionary Computation*, 22(1):113–128, 2016.
- [28] Huw Lloyd. Decentralized parallel any colony optimization for distributed memory systems. In *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCOM/IOP/SCI)*, pages 1561–1567, 2019.
- [29] Huw Lloyd and Martyn Amos. A Highly Parallelized and Vectorized Implementation of Max-Min Ant System on Intel® Xeon Phi™. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6. IEEE, 2016.
- [30] Huw Lloyd and Martyn Amos. Analysis of independent roulette selection in parallel ant colony optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17*, page 19–26, New York, NY, USA, 2017. Association for Computing Machinery.
- [31] Seyed Saeid Masoumzadeh and Helmut Hlavacs. Integrating VM Selection Criteria in Distributed Dynamic VM Consolidation using Fuzzy Q-Learning. In *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, pages 332–338. IEEE, 2013.
- [32] Michalis Mavrouniotis, Shengxiang Yang, Mien Van, Changhe Li, and Marios Polycarpou. Ant colony optimization algorithms for dynamic optimization: A case study of the dynamic travelling salesperson problem [research frontier]. *IEEE Computational Intelligence Magazine*, 15(1):52–63, 2020.
- [33] Haibo Mi, Huaimin Wang, Gang Yin, Yangfan Zhou, Dianxi Shi, and Lin Yuan. Online Self-reconfiguration with Performance Guarantee for Energy-efficient Large-scale Cloud Computing Data Centers. In *2010 IEEE International Conference on Services Computing*, pages 514–521. IEEE, 2010.
- [34] Seyedali Mirjalili. Sca: A sine cosine algorithm for solving optimization problems. *Knowledge-Based Systems*, 96:120 – 133, 2016.
- [35] Joshua Peake, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd. Vectorized Candidate Set Selection for Parallel Ant Colony Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '18*, page 1300–1306, New York, NY, USA, 2018. Association for Computing Machinery.
- [36] Joshua Peake, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd. Scaling Techniques for Parallel Ant Colony Optimization on Large Problem Instances. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '19*, page 47–54, New York, NY, USA, 2019. Association for Computing Machinery.
- [37] M. Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, 2017.
- [38] Rafał Skinderowicz. The GPU-based Parallel Ant Colony System. *Journal of Parallel and Distributed Computing*, 98:48–60, 2016.
- [39] Mateusz Starzec, Grażyna Starzec, Aleksander Byrski, Wojciech Turek, and Kamil Pietak. Desynchronization in distributed ant colony optimization in hpc environment. *Future Generation Computer Systems*, 109:125–133, 2020.
- [40] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and distributed Computing*, 70(9):962–974, 2010.
- [41] Thomas Stützle. MAX-MIN Ant System for Quadratic Assignment Problems. 1997.
- [42] Fei Tao, Chen Li, T Warren Liao, and Yuanjun Laili. BGM-BLA: A New Algorithm For Dynamic Migration of Virtual Machines in Cloud Computing. *IEEE Transactions on Services Computing*, 9(6):910–925, 2015.
- [43] Felipe Tirado, Ricardo J Barrientos, Paulo González, and Marco Mora. Efficient Exploitation of the Xeon Phi Architecture for the Ant Colony Optimization (ACO) Metaheuristic. *The Journal of Supercomputing*, 73(11):5053–5070, 2017.
- [44] Felipe Tirado, Angelica Urrutia, and Ricardo J Barrientos. Using a Coprocessor to Solve the Ant Colony Optimization Algorithm. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6. IEEE, 2015.
- [45] Shangguang Wang, Zhipiao Liu, Zibin Zheng, Qibo Sun, and Fangchun Yang. Particle Swarm Optimization for Energy-aware Virtual Machine Placement Optimization in Virtualized Data Centers. In *2013 International Conference on Parallel and Distributed Systems*, pages 102–109. IEEE, 2013.
- [46] D. Wilcox, A. McNabb, and K. Seppi. Solving Virtual Machine Packing With a Reordering Grouping Genetic Algorithm. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 362–369, 2011.
- [47] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014.