ANKARA YILDIRIM BEYAZIT UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OPTIMIZING LOAD BALANCING AND TASK SCHEDULING ALGORITHMS IN CLOUD COMPUTING

M.Sc. Thesis by

Alperen AKMAN

Department of Computer Engineering

September, 2023

ANKARA

OPTIMIZING LOAD BALANCING AND TASK SCHEDULING ALGORITHMS IN CLOUD COMPUTING

A Thesis Submitted to

The Graduate School of Natural and Applied Sciences of

Ankara Yıldırım Beyazıt University

In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering, Department of Computer Engineering

by

Alperen AKMAN

September, 2023

ANKARA

M.Sc. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"OPTIMIZING LOAD BALANCING AND TASK SCHEDULING ALGORITHMS IN CLOUD COMPUTING"** completed by **Alperen AKMAN** under the supervision of **ASST. PROF. DR. Mustafa YENİAD** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Mustafa YENİAD

Supervisor

Prof. Dr. Fatih Vehbi ÇELEBİ

Jury Member

Assoc. Prof. Dr. Yakup KUTLU

Jury Member

Prof. Dr. Sadettin ORHAN

Director

Graduate School of Natural and Applied Sciences

ETHICAL DECLARATION

I hereby declare that, in this thesis which has been prepared in accordance with the Thesis Writing Manual of Graduate School of Natural and Applied Sciences,

- All data, information and documents are obtained in the framework of academic and ethical rules,
- All information, documents and assessments are presented in accordance with scientific ethics and morals,
- All the materials that have been utilized are fully cited and referenced,
- No change has been made on the utilized materials,
- All the works presented are original,

and in any contrary case of above statements, I accept to renounce all my legal rights.

Date: 15/09/2023 **Signature:**

Name & Surname: Alperen AKMAN

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Assist. Prof. Dr. Mustafa YENİAD, for his support and motivation during my study. His valuable advice and guidance helped me a lot while doing this study.

I would also like to thank my dear father Ahmet Akman, my dear mother Tülin Akman and my dear sister Tuğçe Akman, who have never denied their support and trust in me.

2023, 15 September **Alperen AKMAN**

OPTIMIZING LOAD BALANCING AND TASK SCHEDULING ALGORITHMS IN CLOUD COMPUTING

ABSTRACT

The load balancing and task scheduling are important problems that need to be optimized in cloud computing in terms of meeting the expectations of both the user and the provider. A poorly optimized scheduling method harms the customer and the provider due to non-fulfillment of Quality of Service (QoS) and violation of the Service Level Agreement (SLA). It is possible to meet customer and provider demands in the best way with a well-optimized task scheduling algorithm. Metaheuristic algorithms produce good outcomes in solving NP-Hard problems such as cloud task scheduling problem.

In this study, comparative performance analysis of metaheuristic algorithms such as Clonal Selection Algorithm (CSA), Genetic Algorithm (GA), Differential Evolution (DE), and Particle Swarm Optimization (PSO) are presented to optimize load balancing and task scheduling in cloud computing environments. Contrary to methods such as hybridization of metaheuristic algorithms and adaptive hyperparameters methods used in the literature to increase the search performance of algorithms, a parallelization method is proposed to increase search performance in this study. The proposed hybrid parallelization method is created by taking advantage of the global population master-slave and multiple-deme methods to be independent of the metaheuristic algorithm and to be implementing flexible. The proposed parallelization model is applied separately to CSA, DE, PSO, and GA algorithms, and detailed search performance analyses are presented.

The results showed that when the normal sequential versions of the algorithms are compared, the CSA algorithm achieves more successful results than other algorithms, even though it gives close results to GA. The proposed parallelization model improves the performance of all tested algorithms, and the Parallel Clonal Selection Algorithm (PCSA) that parallel version of the CSA, reached better results than other compared algorithms.

Keywords: Load balancing, cloud task scheduling, clonal selection, parallelization.

BULUT BİLİŞİMDE YÜK DENGELEME VE GÖREV PLANLAMA ALGORİTMALARININ OPTİMİZE EDİLMESİ

ÖZ

Yük dengeleme ve görev zamanlama, bulut bilişimde hem kullanıcının hem de hizmet sağlayıcının beklentileri karşılaması açısından optimize edilmesi gereken önemli bir problemdir. Yeterli seviyede optimize edilememiş görev zamanlama süreçleri, QoS yerine getirilememesi ve SLA ihlali nedeniyle hem müşteri ve hem de hizmet sağlayıcı açısından olumsuzluklara neden olur. Görev zamanlama algoritmasının optimizasyonu ile müşteri ve sağlayıcı taleplerinin en yüksek seviyede karşılanması mümkün olabilir. Metasezgisel algoritmalar bulut görev zamanlama gibi NP-Zor problemlerde iyi sonuçlar üretir.

Bu çalışmada, bulut bilişim ortamlarında yük dengeleme ve görev zamanlamayı optimize etmek için Klonal Seçim Algoritması (CSA), Genetik Algoritma (GA), Diferansiyel Evrim (DE) ve Parçacık Sürü Optimizasyonu (PSO) gibi metasezgisel algoritmaların karşılaştırmalı performans analizleri yapılmıştır. Algoritmaların arama performansını artırmak için literatürde kullanılan metasezgisel algoritmaların hibritleştirilmesi ve adaptif hiperparametre optimizasyonu gibi yöntemlerin aksine, arama performansını artırmak için bir paralelleştirme yöntemi önerilmiştir. Önerilen hibrit paralelleştirme yöntemi, metasezgisel algoritmadan bağımsız ve esnek bir şekilde uygulanabilmesi için küresel popülasyon master-slave ve çoklu-deme yöntemlerinden yararlanılarak oluşturulmuştur. Önerilen paralelleştirme modeli CSA, DE, PSO ve GA algoritmalarına ayrı ayrı uygulanmış ve detaylı arama performans analizleri incelenmiştir.

Sonuçlar, algoritmaların normal sıralı sürümleri ile karşılaştırıldığında, CSA algoritmasının GA'ya yakın sonuçlar vermesine rağmen diğer algoritmalara göre daha başarılı sonuçlar elde ettiğini göstermiştir. Önerilen paralelleştirme modeli, test edilen tüm algoritmaların performansını artırmış ve Paralel Klonal Seçim Algoritması (PCSA), karşılaştırılan diğer algoritmalardan daha iyi sonuçlar vermiştir.

Anahtar Kelimeler: Yük dengeleme, bulut görev planlama, klonal seçim, paralelleştirme.

CONTENTS

NOMENCLATURE

Acronyms

ABC	Artificial Bee Colony
ACO	Ant Colony Optimization
AIS	Artificial Immune System
C-JSO	Cloneable Jellyfish Search Optimizer
CIS	Cloud Information Service
CPU	Central Processing Unit
CS	Cuckoo Search
CSA	Clonal Selection Algorithm
DAPSO	Dynamic Adaptive Particle Swarm Optimization
DB	Datacenter Broker
DE	Differential Evolution
DI	Degree of Imbalance
EA	Evolutionary Algorithms
ECT	Estimated Computation Time
EXP	Deterministic Exponential Time
FCFS	First Come First Serve
FIFO	First in First Out
GA	Genetic Algorithm
GPMS	Global Single Population Master-Slave
IaaS	Infrastructure as a Service
L	Deterministic Log Space
LJFP	Longest Job to Fastest Processor
MCT	Minimum Completion Time
MD	Multiple Deme
MI	Million Instructions
MIPS	Million Instructions Per Second
MP	Massively Parallel
MS	Makespan
NEXP	Nondeterministic Exponential Time
NL	Nondeterministic Log Space
NP	Nondeterministic Polynomial

NP-Complete	Nondeterministic Polynomial Complete
NP-Hard	Nondeterministic Polynomial Hard
P	Polynomial
PCSA	Parallel Clonal Selection Algorithm
PDE	Parallel Differential Evolution
PGA	Parallel Genetic Algorithm
PPSO	Parallel Particle Swarm Optimization
PSO	Particle Swarm Optimization
PSPACE	Polynomial Space
PaaS	Platform as a Service
QoS	Quality of Service
SA	Simulated Annealing
SJFP	Smallest Job to Fastest Processor
SLA	Service Level Agreement
SaaS	Software as a Service
TET	Total Execution Time
VET	Virtual Machine Execution Time
VM	Virtual Machine
VMM	Virtual Machine Monitor

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 General Overview

In today's digital world, cloud computing systems are a rapidly developing field by providing services that meet information technology needs. Needs in academia or industry such as data to be processed, problems to be solved, big data to be stored increase the demand and needs for cloud computing systems due to features of these systems, that are the scalability, speed, flexibility, resource pooling, on-demand self-servicing, and broad network accessing.

In cloud computing, a contractual agreement called a Service Level Agreement (SLA) is made between the service provider and the user [15]. A certain Quality of Service (QoS) is guaranteed by this agreement [16]. Although cloud systems seem to promise unlimited resources, there is a limit to the resources that can be used. Maintaining system resource performance at a level that satisfies QoS is critical for the cloud provider. In particular, the increased number of users, the size of user tasks, and the number of concurrent requests are factors that can affect system performance. Therefore, it is very important to distribute customer tasks to optimum resources through task scheduling. Customer and provider expectations can be met by maintaining QoS with a good cloud task scheduling algorithm [15, 18]. The cloud scheduling algorithm can optimize user-oriented requirements such as completion time, deadlines, and priority of tasks, as well as provider-oriented requirements such as load balancing, energy consumption, and cost.

The cloud task scheduling problem is a combinatorial NP-Hard problem. Unlike classical optimization methods, metaheuristic algorithms give good results in solving these problems. Although classical optimization methods deterministically give the same result in the same time based on a certain formula, they cannot find results in a reasonable time range in problems where it is not possible to predict how to reach the best result because the best result cannot be determined at the beginning or it is not

possible to scan all the solutions because the search area is too large [30, 32]. The computational difficulty brought about by the fact that the problem is a member of the NP-Complete or Hard class forces researchers to find the most appropriate solution method to solve the problems in these classes using classical methods. In addition, classical optimization methods require mathematical continuous and differentiable function extraction, but it may not be able to be expressed mathematically for every problem or it may not be possible to express the formula obtained for the problem as continuous or differentiable [31-32]. Although metaheuristic algorithms do not promise to reach the global optimum or to give the same result in every run, since they are stochastic methods, they can be easily applied to NP-Complete and NP-Hard problems and find reasonable solutions in reasonable time compared to classical optimization methods.

Metaheuristic algorithms are generally developed by being inspired by methods that exist in nature. Algorithms such as the genetic algorithm, particle swarm optimization, and clonal selection algorithm can be given as examples of these methods. Randomness-based natural inspired processes such as natural selection, random population generation of metaheuristic algorithms are important in solving NP-Hard problems. The amount of influence of these random processes is determined by the initial constants, and the optimum initial constant is important as it affects the convergence speed of iterations, getting stuck in local optima, and finding the global optimum. However, despite optimal initial constants, it may be difficult for metaheuristic algorithms to converge to the global optimum in problems with large search spaces. At this point, as an alternative to methods such as hybridization of metaheuristic algorithms and adaptive hyperparameters methods to improve the performance of the algorithms for problems with slow convergence and large search spaces, parallelization of metaheuristic algorithms may be a good method to increase convergence to the global optimum by increasing search capacity.

1.2 Aim of the Study

In this study, comparative performance analysis of metaheuristic algorithms such as clonal selection algorithm (CSA), differential evolution (DE), particle swarm optimization (PSO), and genetic algorithm (GA) is presented to optimize load balance for cloud task scheduling. Contrary to methods such as hybridization of metaheuristic algorithms and adaptive hyperparameters methods used in the literature to increase the search performance of algorithms, a parallelization method is proposed to increase search performance in this study. The proposed hybrid parallelization method is created by taking advantage of the global population master-slave and multiple-deme methods to be independent of the metaheuristic algorithm and to be implementing flexible. This model can also be seen as isolated multi-deme parallelization. In this way, it is aimed to increasing the search performance of algorithms by increasing the number of search workers in the search space. The proposed parallelization model is applied separately to CSA, DE, PSO, and GA algorithms, and detailed search performance analyses are presented.

1.3 Related Studies

In the literature, there are various publications in which the cloud task scheduling problem is solved successfully with metaheuristic algorithms according to different optimization goals such as load balancing, makespan, energy consumption, resource utilization, etc.

Ge et al. [65] propose a GA for load balancing. In the study, GA is used to optimize the load balance to ensure a more even distribution of tasks to the nodes in Hadoop MapReduce. While adapting the GA to the problem, the candidate solutions are expressed as a one-dimensional array such that the size of the array is equal to the number of tasks, while the value of the task belonging to the index of the array represents the source to which the task is assigned. As a fitness function, it is aimed to increasing the load balance by ensuring that the tasks are completed in the shortest time by looking at makespan. The results show that GA achieves better results than the first in first out (FIFO) scheduler in Hadoop.

Naithani [6] proposes a GA for the optimization of energy consumption when assigning tasks to VMs. In the study, reducing energy consumption by making resource utilization optimization is aimed. While adapting the GA to the problem, the number of tasks is encoded using a long, one-dimensional array in which the indices represent the VMs. GA results were compared with first come first serve (FCFS), and energy consumption was reduced with GA.

For makespan optimization, Attiya et al. [14], Al-maamari et al. [13] and Bürkük et al. [16] performed research. Attiya et al. [14] used a simplified version of the PSO algorithm as social only, which eliminated the cognitive part (personal memory) in the formula and used the inertia and social parts. In their comparison with GA and their proposed PSO, the proposed PSO method achieves better makespan optimization results than GA. Al- Maamari et al. [13] suggest dynamic adaptive PSO (DAPSO) which is an improved version of PSO. In the DAPSO algorithm, the inertia value is made dynamic, so that in iterations of the problem where global search is more important, a large inertia value is assigned, and in iterations where local search is important, a small inertia value is assigned. The DAPSO algorithm achieves better makespan optimization results than standard PSO. Bürkük et al. [16] proposes a cloneable jellyfish search optimizer (C-JSO) to optimize makespan. The proposed C-JSO algorithm includes such enhancements which are controlled dynamic population growth to avoid local minimums and a cloning mechanism to avoid producing similar candidates. The C-JSO algorithm proposed in the study achieves better makespan values than the standard JSO and ant colony optimization (ACO) algorithms.

It is possible to optimize multiple cloud optimization objectives together with metaheuristic algorithms. Wang et al. [19] performed multiple optimizations, including load balancing and makespan, with GA. In the study, individuals with less load variance and time consuming are evaluated as good by defining two fitness functions. In addition, probability-based operations such as crossover and mutation in GA are provided in an adaptive way, preventing the possibility of making major changes in cases where the algorithm is close to the optimum solution. The results show that the algorithm successfully optimizes total time, average time, and load balance at the same time. Similarly, Jena [66] proposes a multi-objective clonal

selection algorithm (CSA) to optimize energy consumption and makespan. In the study, fitness functions are defined to minimize the energy consumption and makespan of datacenter. The final fitness value is obtained by taking the sum of these two fitness values to the power of e. In this way, it is aimed to meeting both user and provider expectations at the same time. While adapting the CSA algorithm to the problem, user tasks and system resources mapping are provided with binary encoding to express candidate solutions, namely antibodies. The results show that the CSA algorithm achieves better makespan and energy consumption values than GA.

There are also successful studies in which more than one algorithm is used together to improve the performance of metaheuristic algorithms. Li et al. [57] proposed an algorithm that mixes GA and DE. The purpose of hybridizing these two algorithms is that GA has good global search capabilities and DE is good at local search. Therefore, operations such as selection and crossover for global search are done with GA first, and then operations such as mutation and crossover are performed with DE to increase local search capabilities. In the study, multiple optimizations for total completion time, cost, and load balancing are also performed. These optimization approaches are multiplied by certain weight values between 0 and 1, and the rate that will affect the total is determined in this way. The proposed hybrid GA-DE algorithm for multi-objective fitness optimization achieves better search performance and convergence speed than normal GA and DE. In addition to DAPSO, Al-maamari et al. [13] also propose a model called MDPSO, which consists of a hybrid of DAPSO and cuckoo search (CS) algorithms. Although the PSO algorithm is good in global searches, it is likely to trap on local optima, since it is weak in local searches. In order to overcome the weakness of PSO in local searches, the DAPSO algorithm is improved with adaptive inertia to global and local searches, but once inertia is reduced, it is not increased again for global search. For this reason, a hybrid version of the DAPSO algorithm is created with the CS algorithm to increase the local search performance. A global search is made with DAPSO, and the result of the last iteration is given to the CS algorithm to improve the result by performing a local search. The results show that the proposed MDAPSO algorithm achieves better makespan results than PSO and DAPSO.

There are studies in which the initial population is created with various algorithms that are not fully optimized but relatively better candidates than a random generation and then better optimized with metaheuristic algorithms. Kaur [67] proposed a hybrid GA for makespan optimization, in which the initial population is formed with the longest job to fastest processor (LJFP) and smallest job to fastest processor (SJFP) algorithms. After the initial population is generated, the individuals in the population are further optimized with GA. The results show that the proposed hybrid method achieves better makespan results than standard GA. Similarly, Alsaidy et al. [9] proposed a hybrid PSO algorithm in which the initial population is generated by LJFP and minimum completion time (MCT) algorithms. Then, the results are optimized by giving the initial population to the PSO algorithm. In tests performed according to makspan, load balancing, and energy consumption metrics, it is seen that the MCT-PSO algorithm achieves the best results compared to algorithms such as normal PSO, LJFP, MCT.

Another way to improve the performance of metaheuristic algorithms are parallelization. Parallelization provides benefits such as faster solution discovery, runtime savings, more extensive search for problems with a large search space, and efficient use of resources [58]. Parallelization of metaheuristic algorithms such as GA in the literature is based on methods in which cores work collaboratively on a single global population or work independently by partitioning the population into different subpopulations and dispersing among the cores [59]. These methods are as follows, respectively: global single population master-slave (GPMS) and multiple-deme (MD). Zu et al. [68] proposed multiple-deme parallelization to increase the running speed of clonal selection for protein structure prediction (PSP) problems. In the proposed paralleling, the best solution in the cores is shared between demes with migration. The results show that the proposed parallelization improves algorithm performance. Similarly, Purbasari [69] proposed multi-deme parallelization for the traveling salesman problem (TSP). In the study, two separate message passing systems are created with master-slave communication method and multiple communication methods. In the master-slave communication model, slave processes operating on the subpopulation send the best to the master, then the master shares it with other slaves. In the multi-communication method, the processes directly send their best solutions to other processes. In comparison, multi-communication message system gives better

performance. Golub et. Al. [60] argued that reducing interprocess communication would provide better performance increase in parallel algorithms and proposed a global population asynchronous master-slave model in their study. The main challenge in this method is making it convenient to work asynchronously on the same data, as the interprocess is global between slave nodes. For this reason, the 3-torunament selection method is used in accordance with asynchronous operation. However, the individual eliminated in a slave thread may still be processed in another slave thread. This causes other slave threads to take action in vain. However, the proposed method achieves performance improvements when compared to the serial version.

CHAPTER 2

BACKGROUND THEORY

2.1 Cloud Computing

Today, cloud computing technologies are emerging as very important services that are rapidly developing and meet information technology needs in many sectors. Despite the rapid development of information processing technologies, the demand for these technologies in the industry or academia is also increasing rapidly. The growing in the size, speed, and diversity of data via the Internet, increases the need for more powerful and new information source technologies, as well as increases the demand for cloud processing technologies [1-3]. Cloud computing technology provides users with a pay-per-use basis cost-saving service with allowing resources to be scaled according to the usage scenario [3-5]. Moreover, since all resources are managed by service providers, it eliminates the cost of maintenance as well as the initial setup cost of resources [5]. In addition to the mentioned cost saving features, providing highly scalable and flexible services enables to user better integration to these services into their business models, such as, users can reduce resources that are no longer needed or increase resources across increased requirements. On-demand self-service, resource pooling, quick flexibility, broad network access, and measured service are the five fundamental attributes that make up the model of cloud computing, according to the National Institute of Standards and Technology [6].

In cloud computing, user tasks are assigned to a resource pool that is made up of powerful computing resources. These resources, which provide users with processing capability, storage space, and software services based on changing requirements, can be handled dynamically [7]. In general, the cloud computing process flows happens as follows: the user requests tasks to be processed on the cloud server. The location of performing task request of user is seen as the client side of the cloud. While the server processes and responds to user tasks, the storage stores the required data and makes it available when needed [8]. Cloud service provider’s services are classified into three

categories, which are as follows: SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service) [6]. In SaaS, software is managed and provided by a cloud vendor over web that users can directly reach software without installing locally [8-9]. All of the maintenance and support are provided by the service vendor, without the user getting involved in any technical issues related to the software [8]. Services like Gmail, and Office 365 are examples of SaaS. In PaaS, a development framework is provided to developers to make possible to develop their software for the provided platform [5-6]. By this way, users develop their applications for the provided platform and deploy over the platform. Google App Engine can be given as example as a PaaS [8]. In IaaS, computing resources like CPU, and storage, are offered. Users can purchase these resources according to their needs [6]. Amazon EC2 is the one of example of the IaaS.

Cloud computing can be classified not only by the type of services it provides but also by the locations it serves, and it can be sorted as: Public Cloud, Private Cloud and Hybrid Cloud. Public clouds are available to anyone on-demand and it can be managed by any third-party organization or business [10]. Previously mentioned cloud service types (SaaS, PaaS, IaaS) generally arise in the public cloud. Private clouds are managed by their owner and serve only for the exclusive usage of an organization [11]. Private clouds can provide to organizations with flexibility and more secure information technology, according to their needs. Hybrid cloud occurs with a combination of Public and Private Cloud [12].

2.1.1 Virtualization

Virtualization can be shown as the most important technology underlying cloud computing technology [6]. Virtualization is a technique that allows multiple virtual machines (VMs) to be run on a physical resource [13]. VMs can be configured with different sizes of CPU, memory, and any other resource according to customer needs [14]. A significant portion of the commercialization of cloud computing technologies involves virtualization [14]. The use of a physical resource by more than one virtual machine, the use of more powerful VM to obtain results faster in a large calculation problem, or low configured VM despite longer processing time are a few examples of how virtualization has helped to commercialization of cloud computing technology. A

strong physical resource can be marketed in small parts to many customers. Customers can use stronger or weaker VMs according to their needs and save cost since they only pay for their usage.

2.1.2 Service Level Agreement (SLA) And Quality of Service (QoS)

In cloud computing, a contractual agreement called Service Level Agreement (SLA) is made between the service provider and the user [15]. A particular Quality of Service (QoS) is guaranteed by this agreement [16]. The cloud user uses services like storage, and computing sources maintained by the cloud provider and pays various fees according to the amount of usage and characteristics of the VMs. Maintaining system resource performance at a level that satisfies QoS is crucial for the cloud provider. In particular, the increasing number of users, the size of user tasks, and the number of concurrent requests are all factors that may have an impact on system performance. Also, although cloud systems promise unlimited resources, there is still a limit to the available VM resources that can be used [17]. The service provider cannot guarantee optimal resource utilization without a good scheduling algorithm, which will result in SLA violations due to non-QoS fulfillment [16, 17]. Therefore, a good job scheduling algorithm in cloud systems is critical for meeting customer expectations by maintaining QoS and meeting product provider expectations by maximizing profits [15, 18]. The cloud scheduling algorithm can optimize user-oriented requirements like makespan time, deadlines, and priority of tasks as well as provider-oriented requirements like load balancing, energy consumption, and cost. Moreover, there is an indirect connection between these optimizations. As a result of poorly optimized task scheduling, that poor mapping of jobs and physical resources / VMs, increases job completion times (makespan). Prolongation of the makespan time leads increasing the cost of the customers, as it extends the working time of the service. On the other side, this situation also has a negative impact on the service provider due to increased energy consumption and maintenance costs [16]. Similarly, too much cluttering of jobs only on the same physical resources / VMs will result in not creating a good load balance, which will further increase the cost of energy consumption due to idle energy consumption of idle machines, in addition to the aforementioned problems with

increased makespan times. Therefore, effective task scheduling is required to properly meet the needs of both the provider and the user. [16].

2.2 Cloud Task Scheduling

Cloud task scheduling problems, like regular job scheduling problems, can be defined as follows:

- Let $T = \{T_1, T_2, T_3, \ldots\ldots, T_n\}$ is n number of user tasks that needs to be processed in cloud,
- Let $R = \{R_1, R_2, R_3, \ldots\ldots, R_m\}$ is m number of resources.
- Let $F = \{F_1, F_2, F_3, \ldots\ldots, F_z\}$ is z number of fitness values of defined objectives.

The task scheduling problem is finding the optimum solution, that mapping n number of tasks in the T to m number of resources in R, with maximizing or minimizing value in the set of F within a suitable time range [3, 6]. The task scheduling problem is a combinatorial optimization problem [7, 19]. A task scheduling diagram is shown in the Figure 2.1 where, similar to problem definition, n tasks are assigned to m cloud resources.

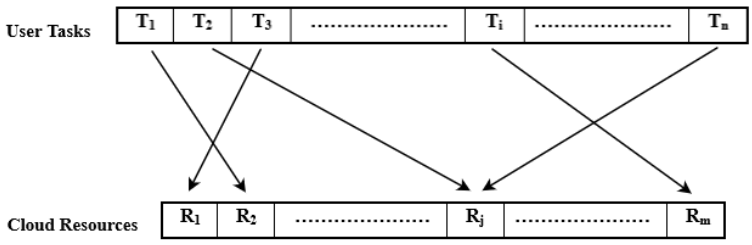

Figure 2.1 A task scheduling that assigning n tasks onto m cloud resources [3]

In a cloud environment, scheduling architecture can be explained with several important components in a general. As can be seen in Figure 2.2, these can be listed as Task Queue, Cloud Information Service (CIS), Datacenter Broker (DB), Hypervisor (a.k.a. Virtual Machine Monitor (VMM)) and Host Operating System. All of the tasks requested by users are kept in the task queue for processing on VMs. The tasks are distributed to VMs by DB with a scheduling algorithm and its objectives. The VM resource information required for the scheduling in DB is collected and provided by

the CIS. The hypervisor serves as an interface between the host operating system and the virtual machines [9, 16, 17]. This study is based on this mentioned scheduling architecture.

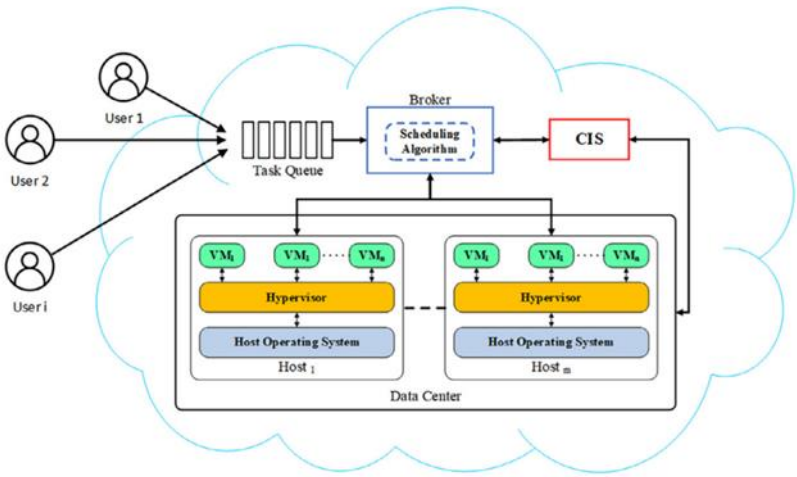

Figure 2.2 Task scheduling architecture in the cloud [9]

Cloud computing consists of different service stacks, and scheduling techniques can be applied in each layer. These service layers can be listed from bottom to top as follows: deployment (infrastructure) layer, virtualization (platform) layer, and application layer. Previously mentioned cloud service types like SaaS, PaaS, and IaaS, correspond to services of these service stacks. Scheduling in the application layer comprises the scheduling of virtual or physical resources. This means that scheduling can be optimized by assigning tasks to physical or virtual resources (such as VMs). The optimization objective of the scheduling algorithm in the application layer can be user or service provider oriented. User-oriented optimizations improve the QoS, including makespan, user cost, priority, performance, deadline while provider-oriented optimizations focus on load-balance, resource utilization, energy consumption. Scheduling in the virtualization layer include scheduling of VMs assigning to physical resources. Optimization made in this virtualization layer may focus on objectives like load-balancing, energy efficiency, cost effectiveness which can be user-oriented, provider-oriented, or both. Apart from scheduling in the application and virtualization layers, another scheduling optimization can be performed in the deployment layer. One of the difficulties faced by service providers in this layer is placing services at

appropriate locations. Scheduling optimizations can be focused on this layer, such as placing services in appropriate locations and collaborating with other providers [3].

Task scheduling algorithms aim to optimize one or several optimization objectives. Development and performance measurement of the algorithms are realized based on these optimization objectives. Previously mentioned that these optimization objectives can be user-oriented or provider-oriented. In literature, the most frequently studied user-oriented objectives can be sorted as makespan, deadline, user cost, priority [3, 17]. Makespan time focuses to optimize reducing execution time of the tasks [16]. Performance losses due to a poorly optimized scheduling algorithm can lead to increased task times and queued tasks not being completed on time. For this reason, deadline optimization is important to the user in terms of meeting the QoS in their work. Users can also request that urgent tasks be prioritized in exchange for a higher cost. Here, it can be important to prioritize urgent tasks without exceeding the deadline of pending tasks in the queue. Another important optimization is cost that, losses that occur due to performance losses are undesirable. The most studied provider-oriented optimizations in the literature are listed as follows: resource utilization, load-balancing, energy consumption, cost (maximize profitability) [3, 17, 19]. Efficient and maximum using of resources is very important in the cloud system. So that, resource utilization is an important optimization for service provider. The balanced distribution of tasks across resources is identified as load balancing [19]. Balanced load distribution among resources also contributes to better resource utilization and lower energy consumption [17, 20, 21]. Energy consumption is another optimization objective that is important to the provider in terms of both the environment (such as reducing carbon emissions) and cost. By using resources effectively and balancing the load, energy consumption can be decreased. The key point is that the resources continue to consume a certain amount of energy even if they are idle [17]. Utilizing resources effectively will reduce the amount of time that the machines are idle and will aim to complete the work as quickly as possible. In this way, energy consumption can be reduced. Another approach is to minimize the number of resources used with VM consolidation and put the unused servers on standby [17, 22]. Cost is important for the service provider as well as the user. The service provider should be able to generate

revenue by serving its customers. It is necessary to maximize profitability without compromising the quality of service.

2.3 Optimization

Optimization can be defined as the process of solving a problem to reach the minimum or maximum value of a given function in mathematics [23, 30]. Optimization can also be expressed as obtaining the most suitable solution by providing certain restrictions for the given purpose or purposes [31]. Optimization problems are evaluated by the objective function in order to find solutions with optimal values within the solution set [33]. Problems such as minimizing the space used in the problem of placing the boxes on the pallets, and minimizing the price in order to find the cheapest round-trip problem can be given as examples of the evaluation criteria of the objective function in optimization problems [33]. Optimization problems mathematically can be expressed as follows:

$$\begin{aligned} &min\, f(x) \\ &max\, f(x) \end{aligned} \tag{2.1}$$

where *f(x)* function is called as *objective* function, *x* is independent variable that can be vector or certain element. If optimization problem is a minimization problem, *f(x)* can also be called as *cost*. Otherwise, if the optimization problem is a maximization problem, then *f(x)* function can be expressed as *fitness* function [29].

Combinatorial optimization problems can be defined as finding the solution with the most suitable combination from a finite set of objects according to a certain objective function. Task scheduling problems are combinatorial optimization problems since a certain number of user tasks at a given time are assigned to the most suitable machines in accordance with the optimization goal. The number of potential solutions to the job scheduling problem can be expressed in exponential size as m^n (m number of machines, n number of jobs). Also, the solution set is exponential for most combinatorial problems [24, 33]. Combinatorial optimization problem has a finite set of independent variables. Therefore, unlike continuous optimization problems the

independent variable of combinatorial optimization problems is discrete, not continuous [29].

2.3.1 Complexity Theory

Complexity theory is a computational model used to associate and group problems with complexity classes to identify an idea about the performance of developed algorithm [25]. The performance of algorithms can be evaluated as space and time. Evaluation is performed with a Turing machine and is denoted by Big O notation. Calculating the complexity of problems is important because solving a problem belonging to one complexity class can pave the way for solving another problem belonging to the same class [33]. As can be seen in the Figure 2.3, P ≠NP time complexity classes are basically divided into P (Polynomial), NP (nondeterministic polynomial), NP-Complete and NP-Hard. But the complexity classes are not limited to these; in addition, there are other time and space complexity classes such as L (deterministic log space), NL (nondeterministic log space), PSPACE (polynomial space), EXP (deterministic exponential time), NEXP (nondeterministic exponential time) [26]. Polynomial problems are problems that can be solved by deterministic Turing machine in polynomial time [25]. NP problems can be expressed as problems that can be verified in polynomial time [26]. NP problems are difficult to solve in polynomial time, but they can be verified. In addition, the NP class encompasses the P class. NP-Complete problems are problems that cannot be solved in polynomial time since the solution time in each step is longer than the previous one. Any problem in the NP set can be reduced to the NP-Complete class in polynomial time. The reduction can be explained as follows: Let P_1 and P_2 be two separate problems, and let X be an algorithm that solves P_1 in polynomial time. If X can be used to solve the P_2 problem in polynomial time, the P_2 problem can be reduced to the P_1 problem [33]. Therefore, an algorithm that solves the problem in the NP-Complete class in polynomial time can solve every problem in the NP class in polynomial time. NP-Complete class problems are members of both the NP class and the NP-Hard class. Solutions can be verified in polynomial time due to being a member of NP class. The NP-Hard class is the most difficult problem in NP, so it cannot be proven to have a solution in polynomial time. NP-Hard class problems may not be NP class. Therefore, solutions cannot be verified

in polynomial time as in the NP class. Similarly, in this class, the problem in the NP class can be reduced to the problem in the NP-Hard class. The difference of NP-Hard problems from NP-Complete can be shown as NP-Hard problems may not belong to the NP class.

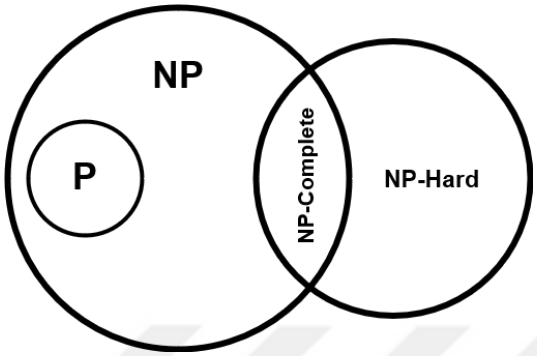

Figure 2.3 P ≠NP Complexity Classes

2.3.2 Heuristic and Metaheuristic Optimization Algorithms

In classical optimization methods, problems are based on a certain formula, and it is ensured that the best solution (maximum or minimum) is reached deterministically. Since these methods are deterministic methods, they can commit to producing the same result in the same execution time for each run [30]. On the other hand, in such optimization problems; since the optimum solution cannot be determined at the beginning of the problem and it is not possible to predict how to reach the best solution, or because the solution space is too large, it is not possible to search for all solutions, so deterministic methods cannot find the solution within a reasonable time range [32]. Optimization problems can have NP-Complete or NP-Hard computational complexity. The computational difficulty of the problems in the NP-Complete or Hard classes and the necessity of developing the best appropriate solution method for solving the problems in these classes, make classical optimization methods difficult for researchers. Classical optimization has limitations such as mathematical formula extraction, and the extracted function being continuous and differentiable [31, 32]. However, it may not be possible to express every problem mathematically or for the derived formula that expresses the problem to be continuous and differentiable. This

method produces good results for integer programming and linear programming problems [32].

The disadvantages mentioned in classical optimization have highlighted heuristic and metaheuristic methods. Heuristic and metaheuristic methods can be applied to NP-Complete and NP-Hard optimization problems successfully and solved within an acceptable time. Heuristic algorithms and metaheuristic algorithms are preferred for problems that classical methods cannot solve, or solve for long periods of time, or difficult to formula extraction [32, 70]. These algorithms do not guarantee to find the best solution every time, like classical methods, but they can find the closest solution in reasonable time. However, when a quick solution is wanted to find to the problem, it may not always be able to solve problem or when it solves given problem effectively, it may not be able to solve it in a reasonable time. The main reason for this is that classical optimization methods are deterministic methods, and since they are based on a certain mathematical formula, they produce the same result and execution time in every run with the same parameters, while heuristic methods are usually stochastic methods, so they cannot guarantee the same result and execution time since randomness and probability are involved. However, heuristic algorithms are used to produce a solution close to the reasonable best solution in a large problem since it is not possible to find the best result by trying all combinations in terms of time cost [32, 64]. It is also preferred for problems where the optimal solution is unknown. These algorithms are also effective in solving nonlinear and quadratic problems. [32].

The word heuristic in Greek means finding or discovering a new method [29, 32]. This term means discovery by trial and error in computer science. Meta means high level [32]. For this reason, as the name suggests, metaheuristic algorithms have more advanced features than heuristic algorithms, although they are from the same family as heuristic algorithms. Although the previously mentioned features are common features for heuristic and metaheuristic algorithms, there are still differences between them. While heuristic algorithms search more locally than metaheuristic algorithms, metaheuristic algorithms can produce more efficient solutions by avoiding local optima by making more extensive searches [32, 70]. While heuristic algorithms are still problem-specific algorithms, metaheuristic algorithms can be considered as an

adaptive solution framework for different problems [30, 33, 34]. Metaheuristic algorithms are generally developed by being inspired by methods that exist in nature. For example, most popular metaheuristic algorithm is the GA that developed inspired by Darvin's evolutionary theorem. While the artificial bee colony (ABC) algorithm was inspired by the foraging behavior of bees, another algorithm the ant colony algorithm (ACA) was developed by being inspired by the behavior of ants, such as the ability to find the shortest path by leaving pheromone material.

Although nature-inspired algorithms are problem-independent, it is important to define the target function according to the problem being solved. The target function should evaluate the candidate solutions in accordance with the problem criteria, and the candidate solutions should adhere with the boundaries of the solution space. In task scheduling problems, for example, if the candidate solution contains a machine description that exceeds the defined number of machines, it indicates that the candidate solution is outside the boundaries of the solution space. In addition, there are problems where the value obtained with the target function represents a value in the solution space, unlike the solution encoding representation in the task scheduling problem. For this reason, there are various methods to avoid going out of the solution space while creating or evaluating solutions. These can be listed as repair, penalty, and rejection methods. In the repair mechanism, the solution that does not meet the problem criteria is repaired and corrected. Considering the previous task scheduling example again, the exceeding of the machine description in the candidate solution can be repaired by replacing the improper value with the maximum or minimum machine value. In the penalty method, the penalty value can be added to the solution, while in the rejection mechanism, unsuitable solutions can be eliminated [28, 32].

Metaheuristic algorithms can be classified according to their common features in other different categories under the category of nature-inspired algorithms. Population-based ones, evolutionary algorithms, swarm-based algorithms, and by discovery area (bio-inspired, physics and chemistry inspired etc.) can be given as examples of these categories. Algorithms can belong to more than one mentioned category at the same time. For example, evolutionary and swarm-based algorithms are population-based, bioinspired algorithms. Bio-inspired algorithms are algorithms inspired by natural

organisms. The processes developed by living things in various situations are used by researchers to develop algorithms for solving problems [10]. Similarly, Physics and chemistry-based algorithms are algorithms inspired by physics and chemical systems instead of living organisms in nature. Swarm-based algorithms are algorithms inspired by swarm intelligence. Individual agents in the swarm can exchange information and organize themselves by exhibiting behaviors that are not centrally controlled [10]. These swarm intelligence traits inspire researchers to solve computer science problems. Evolutionary algorithms are algorithms inspired by the genetic evolution process [27]. These algorithms are based on the "survival of the fittest" Darwin's evolution theorem. Another example of bioinspired algorithms is artificial immune system algorithms. Artificial immune system algorithms are algorithms developed by being inspired by the immune system behaviors of vertebrates. The classification of nature inspired algorithms is shown in Figure 2.4.

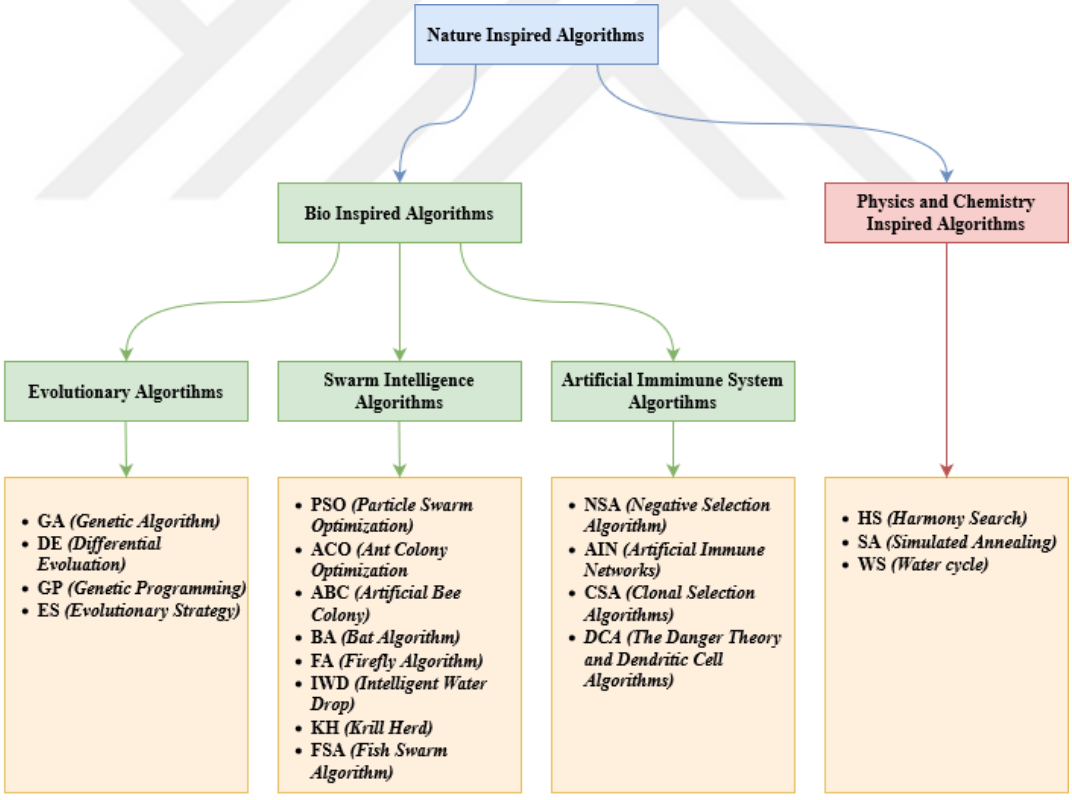

Figure 2.4 Classification of Nature Inspired Metaheuristic Algorithms [10, 27, 37]

Unlike this classification, evolutionary optimization algorithms (or EA Evolutionary Algorithms) are seen in some sources [29] shown as a more general category. More generally, evolutionary optimization algorithms also include nature-inspired algorithms such as swarm intelligent algorithms (like PSO and ACO), physics and chemistry inspired algorithms (like Simulated Annealing (SA)), apart from classical evolutionary algorithms (like GA and DE). Since evolutionary optimization algorithms do not only include nature-inspired algorithms, it can be said that they are more general than nature-inspired algorithms [29]. However, there are authors who use evolutionary optimization algorithms to express Nature inspired algorithms [29]. In addition, evolutionary optimization algorithms are also used to express metaheuristic algorithms because they mostly consist of metaheuristic algorithms. While some authors distinguish swarm intelligence algorithms from evolutionary optimization algorithms, there are also authors who accept them as a subset of evolutionary optimization algorithms because they include the same processes as evolutionary algorithms (like generation evaluation, new candidate solutions are created in each iteration). For example, the inventor of the PSO algorithm, one of the swarm intelligent algorithms, defines the PSO algorithm as a subset of the evolutionary algorithm [29, 35].

Evolutionary optimization algorithms generally involve similar processes. These processes can be shown as the creation and evaluation of candidate solutions and trying to find the optimum solution by creating a new generation with iterations up to a certain stopping criterion [29, 36]. The workflow diagram of these processes is shown in Figure 2.5.

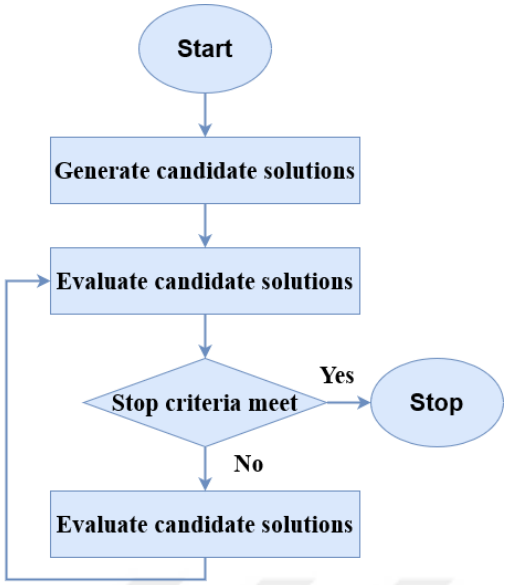

Figure 2.5 Flowchart of evolutionary optimization algorithm processes [36]

CHAPTER 3

METHODOLOGY

Nowadays, task scheduling problem is very important on cloud servers in terms of meeting the expectations of both the user and the provider. A poorly optimized scheduling method harms the customer and the provider due to non-fulfillment of QoS and violations of SLA. It is possible to meet customer and provider demands in the best way with a well-optimized task scheduling algorithm. In this section, the task scheduling problem, the evaluation object, and the metaheuristic evolutionary optimization algorithms chosen for the solution of the problem will be discussed in detail.

3.1 Problem Statement

In the cloud computing environment, the tasks that the user wants to perform on the cloud server are performed by the VM. The task scheduling problem discussed in this study can be defined as the most appropriate assignment of n number of user tasks to m number of VMs.

User tasks-VMs assignments should be made in the most appropriate way in order to meet the user and provider demands in the best way. The cloud scheduling architectural model, in which the problem is addressed, is based on the model mentioned in section 2.2, which includes components such as DB, task queue, and CIS. Tasks requested by users are held in queue and distributed to VMs by DB with scheduling algorithm. The following assumptions are also considered for the optimization of the problem in the study:

- All user tasks are independent of each other, so the incompleteness of one task does not prevent starting or completing of other user tasks.
- The size of the user tasks is not the same; they can be different from each other.
- VMs are heterogeneous in configuration, so VMs do not have to have the same processing capacity or features.

- A task assigned to a VM is completed by the assigned VM. It is not possible to migrate tasks to other VMs, interrupt started tasks, and process a task on more than one VM.

3.1.1 Optimization Objective

The assignment of tasks to VMs is performed according to user or provider-oriented objectives. The objective function is used to measure the quality of the solution produced by the scheduling algorithm. In optimization problems, these values obtained by the objective function are tried to be maximized or minimized. In this study, the main goal is to distribute the tasks to the resources in a balanced way, which is load balancing. In systems with poor load balanced, some resources will be idle, while others will be loaded more. This will lead to unnecessary energy consumption in the waiting time of idle resources, prolongation of makespan times, and increase in costs for the user and the provider, causing negativities for the user and the provider. For this reason, load balance optimization is an important optimization goal in terms of providing user and provider expectations by indirectly affecting makespan, energy consumption, and cost, although not directly. The comparison of powerful and poor load balance and makespan time is shown respectively in Figure 3.1 and 3.2.

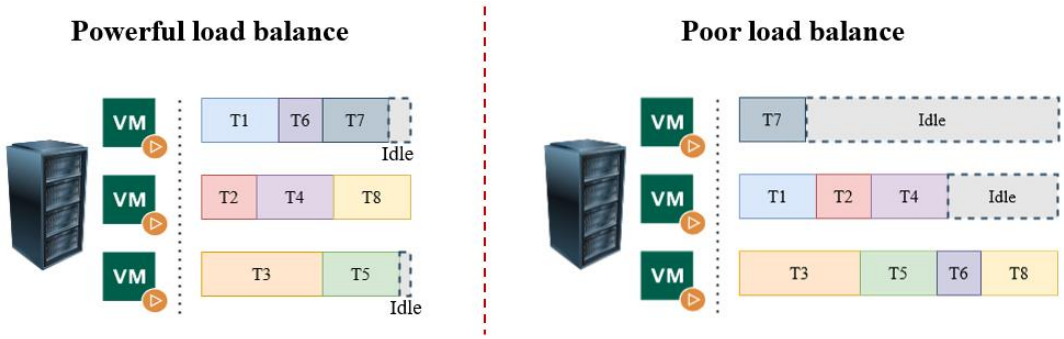

Figure 3.1 Comparison of powerful or poor load balanced cloud server

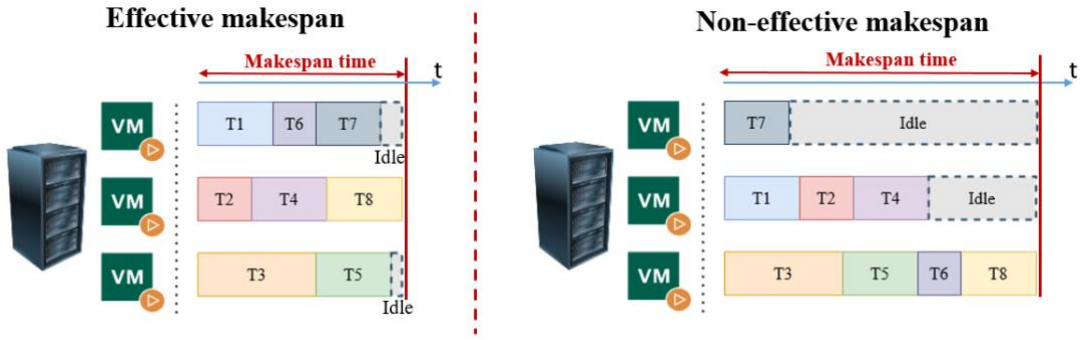

Figure 3.2 Comparison of makespan times

3.1.2 Optimization Formulation

Tasks are distributed to the appropriate VMs in accordance with the load balancing objective by the scheduling algorithm. The load balancing optimization formulation can be expressed as [9, 57]:

- Consider $T = \{T_1, T_2, T_3, \ldots, T_i, \ldots, T_n\}$, $(0 < i \leq n)$ is a task set consisting with total n number of tasks, where T_i is i-th independent task that is identified by task length with million instructions (MI) unit. Therefore, the T set can also be expressed as $T = \{taskSize_1, taskSize_2, \ldots, taskSize_n\}$. Each task in the T has a maximum T_{max} and minimum T_{min} length and is expressed as $T_{min} \leq T_i \leq T_{max}$.
- Consider $V = \{V_1, V_2, V_3, \ldots, V_j, \ldots, V_m\}$, $(0 < j \leq m)$ is a VM set consisting with total m number of VM where V_j is j-th independent VM that identified by machine execution rate (processing speed) with a million instructions per second (MIPS) unit. Therefore, the V set can also be expressed as $V = \{mips_1, mips_2, \ldots, mips_m\}$. Each task in the V has a maximum V_{max} and minimum V_{min} execution rate and is expressed as $V_{min} \leq V_j \leq V_{max}$.

The execution time of the tasks on VMs is calculated by the estimated computation time $ECT(i,j)$ formula that calculation of i-th task on j-th VM is given by:

$$ECT(i,j) = \frac{T_i}{V_j}, \quad (0 < i \leq n), \text{ and } (0 < j \leq m) \tag{3.1}$$

The total execution time of the tasks assigned to a virtual machine is calculated with the virtual machine execution *VET(j)* formula that calculation of *j*-th VM total execution time expressed as:

$$VET(j) = \sum_{i=0}^{n} ECT(i, j) * CONV(i, j) \quad \textbf{(3.2)}$$

where *CONV(i, j)* is a function that determines whether the *i*-th task is assigned to the *j*-th VM. If the *i*-th task is assigned to the *j*-th VM then it returns 1, otherwise it returns 0.

Makespan (tasks compilation time) is calculated by *MS* formula that expressed as:

$$MS = MAX\{VET(j),\ 0 < j \leq m\} \quad \textbf{(3.3)}$$

The total execution time of each task on assigned VMs is calculated by the total execution time *TET* formula that expressed as:

$$TET = \sum_{j=1}^{m} VET(j) \quad \textbf{(3.4)}$$

The degree of imbalance *DI* on the cloud server is calculated by following formula:

$$DI = m\left(\frac{MS - min\{VET(j),\ 1 < j \leq m\}}{TET}\right) \quad \textbf{(3.5)}$$

Load balancing optimization *O* is represented by the following:

$$O = min\{DI\} \quad \textbf{(3.6)}$$

3.1.3 Problem Encoding

The most important step of metaheuristic evolutionary optimization algorithms is how to express candidate solutions for problem solving. Since the metaheuristic evolutionary optimization algorithms discussed in this study are problem independent algorithms, the most important step in adapting the algorithms to the problem is to

encode the problem correctly. The nomenclature of candidate solutions may vary from algorithm to algorithm; for example, antibody in the CSA, chromosome in the GA, and particle in PSO, while they all represent an encoded solution. But as a more general expression, the candidate solutions are called as individuals in this section. The indirect value encoding method is used in this study. According to this method, each individual is encoded with a sequence or vector as long as the number of tasks to be assigned [57]. As a result, the length of the individual is equal to the number of tasks, and the values it can accept are equal to the number of VMs. In short, the index of the directory represents the job, the value of which is assigned to the VM. An example of an individual consisting of 8 tasks and 3 VMs is shown in Figure 3.3.

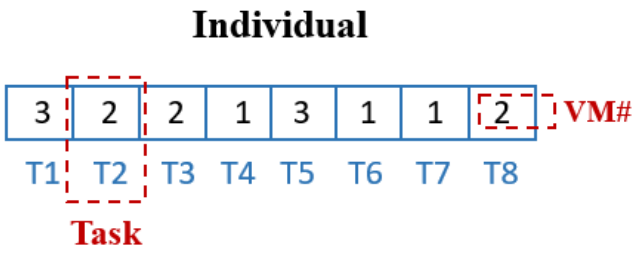

Figure 3.3 An illustration of individual that consisting of 8 tasks, 3 VMs

For example, in Figure 3.3, in the individual consisting of 8 tasks and 3 VMs, the first task will go to the third virtual machine and the second task to the second virtual machine. This mutual decode and encode operation example is also shown in Figure 3.4.

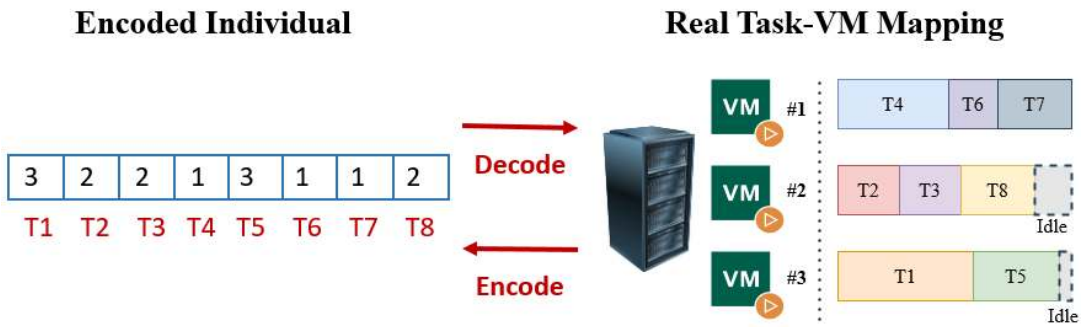

Figure 3.4 An illustration of mutual decode, encode operation

Encoded individuals can also be expressed with a two-dimensional CONV matrix consisting of 0s and 1s. One dimension of the CONV matrix represents tasks, and the other dimension refers to VMs. If the task in the column is assigned to the VM in the row, it takes the value 1, otherwise 0 [57]. Here, the number of columns is equal to the number of tasks, and the number of rows is equal to the number of VMs. Because each task is assigned to only one virtual machine, there is only a single value of 1 in each column [57]. CONV matrix conversion example is shown in Figure 3.5.

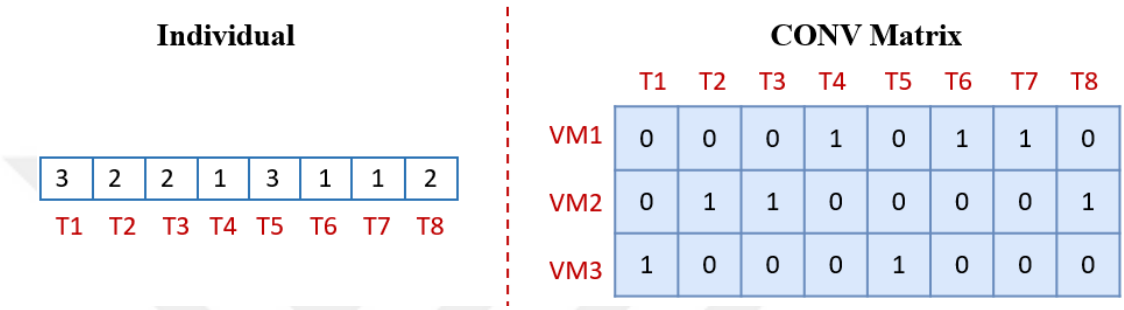

Figure 3.5 An example of CONV matrix conversation

3.2 Selected Evolutionary Optimization Algorithms

3.2.1 Clonal Selection Algorithm

Clonal selection algorithm (CSA) is a member of the artificial immune system (AIS). The immune system of living things ensures the detection and protection of potentially harmful foreign element [39]. The immune system consists of three components: the physical barrier, the innate immune system, and the adaptive immune system [37]. While physical barriers are the first protective mechanism faced by external factors such as skin and tear secretion, the innate immune system is a ready defense system that recognizes microbes. The adaptive immune system, on the other hand, is a defense system that is shaped by the microorganisms encountered and produces antibodies specific to them. The adaptive immune system inspires the majority of artificial immune system models that have been developed [37]. CSA was developed by De Castro and Van Zuben (2000) to solve nonlinear functions [71, 72]. CSA is based on clonal selection in the immune system and has similar processes as other evolutionary optimization algorithms. Antigens are harmful molecules in the bodies of individuals.

B-lymphocytes, a component of the adaptive immune system, form the defense system by producing antibodies specific to the antigens they encounter. Each antibody has different affinities for the antigen. The best affined antibody against the antigen is used as a memory cell to be cloned by plasma cells. These plasma cells undergo clonal expansion to combat invading antigens and may undergo hypermutation during this expansion. Hypermutation may increase the affinity of progeny cell receptors for the trigger antigen, resulting in better antibody production [40].

The clonal selection algorithm is based on this mentioned biological motivation. The population consists with set of candidate solutions that are called in the CSA as antibody. The result obtained as a result of the evaluation of the antibodies with the fitness function in the algorithm corresponds to the similarity ratio between the antibody and the antigen. The degree of closeness to the desired result of the fitness score indicates the highness degree of the affinity ratio. CSA includes processes such as: clonal selection, clonal expansion, and hypermutation from the immune system [40]. First of all, a set of candidate solutions (antibodies) is generated. The affinity value of the antibodies in the population is calculated according to a problem specific objective function. A certain number of antibodies with the highest similarity ratio are selected, and cloning is performed. The cloning of antibodies is performed in direct proportion to the affinity value. In other words, since the number of antibodies to be formed by cloning is certain, the antibodies with the highest affinity value will be copied more than the other selected antibodies. The determination of the replication numbers of the selected antibodies is carried out with the following formula, called the cloning factor:

$$Qf_c = \left(\frac{af_c}{\sum_{c=1}^{n} af_c} \right) * R,\ c \in \{1,2,\ldots,n\} \qquad \textbf{(3.7)}$$

where c represents the number of selected antibodies, Qf_c represents the value of cloning factor of c. af_c identify value of affinity of c and $\sum_{c=1}^{n} af_c$ is summation of affinity values of all selected n number of antibodies. R expresses number of antibodies to be cloned.

The cloned antibodies then undergo hypermutation. The critical point here is that cloned antibodies with the highest affinity mutate less, while antibodies with low affinity mutate more. The logic here is that antibodies with high affinity need small changes to increase the similarity ratio, while antibodies with low affinity need more changes to reach the optimum solution. The calculation rate of hypermutation can vary depending on the problem type such as minimization or maximization problem. Shui et. al. [38] in the solution of the transportation timing problem with the clonal selection algorithm, they calculate the mutation factor with the following formula:

$$Mr_i = b + \left(\frac{a - b}{p}\right) * i,\ i \in \{1,2,\ldots,p\} \quad \textbf{(3.8)}$$

where Mr_i represents the mutation ratio group of antibody clones. In order to apply this formula, cloned antibodies are sorted according to their affinity value and divided into p groups. In the formula, i indicates the group number, while p indicates the number. a and b represent maximum and minimum mutation ratios, respectively. While calculating the mutation rate of each group with this formula, it is ensured that the mutation rates of the antibodies belonging to the same group are the same and the mutation rates of the antibodies with a high affinity group are smaller. Whether a mutation will occur in the gene on the antibody belonging to a group is determined by this mutation rate of that group. For example, if the mutation rate is 0.5, it means that there is a fifty percent chance that that gene will be mutated.

The CSA performs following steps:

Step 1: Generate set of antibodies that candidate solution

Step 2: Select predefined number of antibodies that have best affinity values

Step 3: Clone antibodies that cloning amount is proportional to best affinity values

Step 4: Perform hypermutation inversely proportional to the best affinity values

Step 5: Repeat steps 2-5 until a certain number of iterations or stopping conditions are met

3.2.1.1 CSA Algorithm Design to the Problem

CSA implementation in this study starts with the definition of constants as, population size, number of iterations, selection size, and minimum and maximum mutation rate constant values. The population of the candidate antibodies is generated randomly according to the indirect encoding method mentioned in Section 3.1.3 with a given population size initially. The degree of imbalance value in Formula 3.5 is used to calculate the fitness or cost value, which is the similarity ratio of each antibody in the randomly generated population. The population is sorted from the best to the worst according to this similarity ratio, and the first antibodies are selected from the population as much as the number of selection sizes. The cloning numbers of the selected antibodies for clonal expansion are calculated with the cloning factor in Formula 3.7, and copies are created by the number of cloning factor values that antibodies have. The generated replica antibodies are hypermutated according to the mutation factor calculation in Formula 3.8. New antibodies are produced randomly as the number of selected antibodies. A new population is formed by combining randomly generated antibodies with cloned antibodies. Therefore, all antibodies are eliminated in the population, and newly cloned antibodies are included to the new population in proportion to the number of antibodies eliminated. These steps continue until the number of iterations that given at the beginning is reached or an antibody with an imbalance degree value of 0 is found. The pseudocode of the algorithm is given in Figure 3.6.

```
CSA {
Initialize algorithm constants (population size, iteration size,
selection size, min, and max mutation ratio)
Generate population of random antibodies
Calculate similarity (fitness) values of each antibodies
while (i < iteration size and not reached to global optimum) do
{
   Sort population best to worst with respect to similarity
   Select antibodies as much as selection size
   Calculate cloning factors of selected antibodies
   Perform clonal expansion with cloning selected antibodies
with their cloning factors
   Calculate mutation factors
   Perform hypermutation to cloned antibodies with mutation
factors
   Generate new population with combining randomly generated
antibodies and new cloned antibodies
   Calculate similarity (fitness) values of each antibody in the
new population
   i++
} End while
```

Figure 3.6 Pseudo code of the CSA for the problem

3.2.2 Differential Evolution Algorithm

The differential evolution (DE) algorithm is a combinatorial optimization algorithm that was developed by Storn and Price (1995) for continuous value optimization problems [29, 36]. In addition to being a population-based algorithm, the DE algorithm has processes that correspond to the selection crossover and mutation operations in the genetic algorithm, although they are not identical. Although it contains procedures similar to the genetic operators in the GA, it may not be possible to say that it is entirely motivated by biological factors [29]. The population of the DE algorithm consists of candidate solutions called vectors. Vectors are d-dimensional according to the domain size of the problem [36]. Although the DE algorithm was originally designed for continuous value problems, it was later adapted for discrete value problems by researchers due to its success.

The DE algorithm is based on the creation of a new trial vector as a result of operations such as vector subtraction, scaling and addition with three vectors selected differently from each other [29]. The algorithm basically consists of selecting three different vectors, creating the trial vector, and replacing it with a parental vector. The population

of candidate solutions called vectors is generated according to the size of the problem domain. Candidate solutions such as individual or chromosome in other evolutionary optimization algorithms correspond to vector in this algorithm. The size of the vector is determined by the problem size, while the population size is determined by the hyperparameter *n* given at the beginning as an initial value. Three different vectors are randomly selected from the population to generate the mutant vector. The generation of mutant vector is first step of the generation of trial vector. Mutant vector generation is performed following formula:

$$Vt_i = x_i + F * (y_i - z_i),\ i \in \{0,1,2,\ldots,d\} \quad \textbf{(3.9)}$$

where i represents number of genes in the vector and d size of the vector. x_i, y_i, z_i represent three different vectors randomly selected from the population. F is the differential weight that identifies the magnitude of variation in the genes of the newly created trial vector. It can take values from 0 to 2 and is initially set as a constant. If the formula is to be examined in more detail, after taking the difference of y_i and z_i, it is multiplied by the differential weight to create a scaled difference vector. The mutant vector Vt_i is obtained as a result of the sum of the scaled difference vector and x_i [29]. This process is shown in Figure 3.7 for two-dimensional space where d is 2.

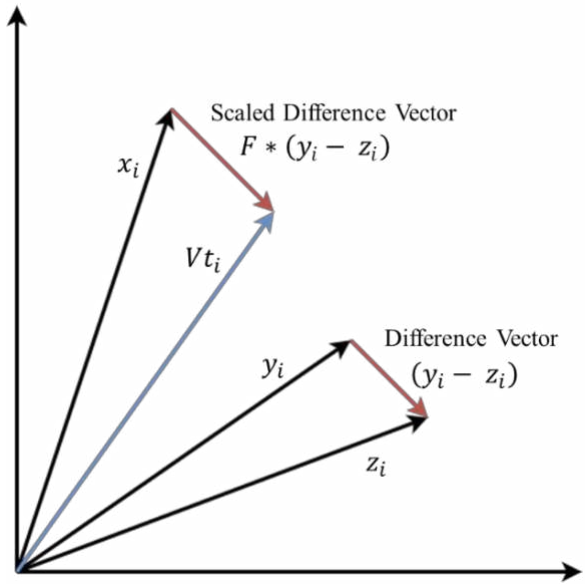

Figure 3.7 Generation of mutant vector in the 2-dimensional domain [29, 41].

Replacing the gene of each generated mutant vector with the gene of the parental vector is probability dependent. This probability is determined by the initially determined crossover ratio *CR* constant. *CR* can take values from 0 to 1. For example, a *CR* value of 0.5 means that each of the parental vector genes has a fifty percent probability of replacing with mutant vector genes. This process can be represented mathematically as the following statements [41]:

$$\begin{aligned} rand(0,\ 1) < CR \ &\rightarrow Vp_i = Vt_i \\ rand(0,\ 1) \geq CR \ &\rightarrow Vp_i = Vp_i \end{aligned} \qquad \textbf{(3.10)}$$

where Vp_i represents the parental vector, as a new parameter to the previous formula. At this point, the creation period of the trial vector is completed. Mutant vector creation, defined by formula 3.9, is a mutation operation of three different vectors excluding the parental vector. The process of replacing the genes of the created mutant vector with the parental vector genes according to the *CR* ratio, that defined in formula 3.10, is the binomial crossover process [36]. Unlike the binomial crossover, another well-known crossover method is the exponential crossover [42]. Exponential crossover is accomplished by selecting contiguous genes from the mutation vector, completing the remaining sites from the parental vector to forming the trial vector [42]. The starting point *s* is randomly selected from the mutant vector. Then, a random *e* endpoint is determined according to the *CR* probability value [43]. In both $s,\ e \in [1,\ d]$ where *d* is size of problem domain that vector size. With this value, the genes to be added to the trial vector can be determined. The effect of the difference between binomial and exponential crossover is shown in the Figure 3.8.

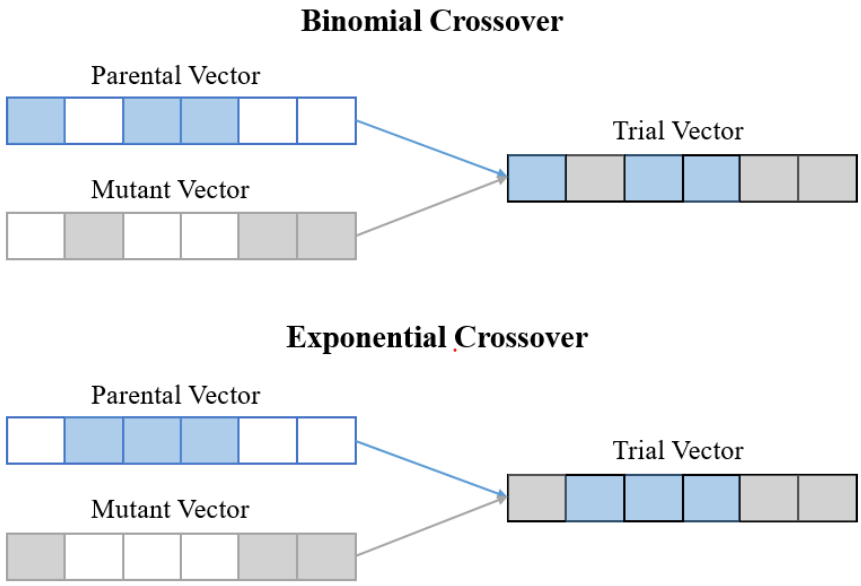

Figure 3.8 Comparison between binomial and exponential crossover [42, 43]

The selection process is applied between the created trial vector and the parental vector. The entry of the trial vector into the population is determined by the survival principle of the fittest. In other words, the vector with the best fitness score is selected from the trial vector with parental and added to the population. According to the minimization optimization problem, this process can be expressed mathematically with the following formula [41]:

$$\begin{aligned} f(Vp_j) \leq f(Vt_j) &\rightarrow \quad Vp_j^{t+1} = Vp_j^{t} \\ f(Vt_j) < f(Vp_j) &\rightarrow \quad Vp_j^{t+1} = Vt_j^{t} \end{aligned} \tag{3.11}$$

where f is the minimization optimization (cost) function. j represents the number of vectors in the population that $j \in [1, n]$. As can be seen, the DE algorithm includes processes such as crossover, mutation, and selection, which are also present in genetic algorithms, but their application methods are different. In the differential evolution algorithm, mutation and crossover are performed with four different vectors during the formation of the trial vector. Crossover and mutation operations are handled completely differently in the GA. Another difference is that while new individuals

created as a result of processes such as crossover mutation in the GA begin to affect the population by being included in the next iteration, trial vectors created in the DE algorithm begin to affect the population in the same iteration. Three different vectors selected for the parental vector can be new vectors produced in the same iteration [36].

The DE algorithm occurs with following steps:

Step 1: Define the value of initial hyperparameter constants (*n, F, CR*)

Step 2: Generate random set of vectors that candidate solution

Step 3: Select three different vectors for the next parental vector

Step 4: Generate trial vector with randomly selected three different and parental vector

Step 5: Adding the whichever better of the parental or trial vector to the population

Step 6: Repeat steps 3-5 for each vector in the population

Step 7: Repeat step 6 until a certain number of iterations or stopping conditions are met

3.2.2.1 DE Algorithm Design to the Problem

The implementation of DE in this study starts with the definition of constant values for population size (*n*), number of iterations, differential weight (*F*), and crossover ratio (*CR*). The population of candidate vectors is randomly generated as many as the number of population size according to the indirect encoding method mentioned in section 3.1.3. The dimension of the vector is equal to the number of tasks to be assigned to the machines. Each dimension of the vector represents a machine assigned to a task corresponding to that dimension. The degree of imbalance value in Formula 3.5 is used to calculate the fitness or cost value, which is the evolution of each vector in the randomly generated population. When creating a mutant vector for each vector in the population, three different vectors are selected from the population, except for the parental vector. Mutant vector generation is carried out with the mutation formula given in Formula 3.9. However, this formula is appropriate for continuous problems and can also extend beyond the search range. Since the scheduling problem in this study is a discrete problem, the continuous value obtained as a result of the formula is rounded to the nearest discrete value. In addition, an unsuitable value outside the problem search area is replaced with the closest value within the search area. Binomial crossover in Formula 3.10 is performed between the mutant vector and the parental

vector to generate the trial vector. In the selection step, the selection process is applied between the trial vector and the parent vector according to the minimization optimization given in Formula 3.11 These mutation, crossover, and selection process is applied sequentially to all vectors in the population. These steps continue until the number of iterations that given at the beginning is reached or a solution vector with an imbalance degree value of 0 is found. The pseudocode of the algorithm is given in Figure 3.9.

```
DE {
Initialize algorithm constants (population size (n), iteration
size, differential weight (F), crossover ratio (CR))
Generate population of random vectors
Calculate cost (fitness) values of each vectors
while (i < iteration size and not reached to global optimum) do
{
   j=0
   while (j < n) do {
       Select 3 different vectors in the population except
population[j] (parental vector)
       Create mutant vector with selected vectors
       Fix values of trial vector that continues and outside of
the search space
       Create trial vector with mutant and parental vector
       Calculate cost (fitness) value of the trial vector
       Perform selection between parental and trial vector
       j++
  } End while

  i++

} End while
```

Figure 3.9 Pseudo code of the DE for the problem

3.2.3 Particle Swarm Optimization

The particle swarm optimization algorithm (PSO) is an evolutionary computational technique that was proposed by Kennedy and Eberhart at the Congress on Evolutionary Computation in 1995 for continues vector problems [9, 36, 44]. The PSO algorithm is an algorithm inspired by the social behavior of animals in the swarm. Therefore, PSO is a population-based, swarm intelligence algorithm. Although there is no central intelligence in swarm intelligence, there is intelligence in the interaction between individuals in the swarm. Grouped animals have an advantage over alone

individuals, such as foraging and predatory attacks. Predators may have difficulty focusing on a single animal in a pack [29]. Zebras, for example, are a more obvious target when they are alone, but when they form a swarm, it is difficult to distinguish them individually [29, 45]. Animals in a swarm may appear larger, make a louder noise, and become a more difficult target [29]. In addition, in swarms, animals are better at detecting threats and reacting early, as they have more eyes that spy on the environment. Although foraging in a flock does not seem advantageous in terms of sharing the found food, sneaking up on the prey, etc., foraging with more eyes is much more advantageous than foraging individually [29, 46]. These swarm advantages can be used to demonstrate swarm intelligence. PSO algorithm based on bird flock and fish schooling as biological motivation [14]. The approach taken in the PSO can be explained through the bird flock as follows: birds in a flock fly randomly to search food in a certain area. When foraging, birds do not know of the exact location of the food source, but they are aware of the best proximity they have discovered, and the best proximity found within the flock. Birds in the flock fly towards the bird closest to the source. For example, if one of them reaches a new closer position while the birds are flying towards the bird closest to the source, the other birds in the flock will orient towards that new position. In general, it can be summarized as the foraging process of birds in a flock and the tendency of birds in the flock towards the bird that finds the food. A summary illustration of this process is given in Figure 3.10. However, when orienting towards to the best bird in the flock, the position they find best is also a determining factor in their direction in the algorithm.

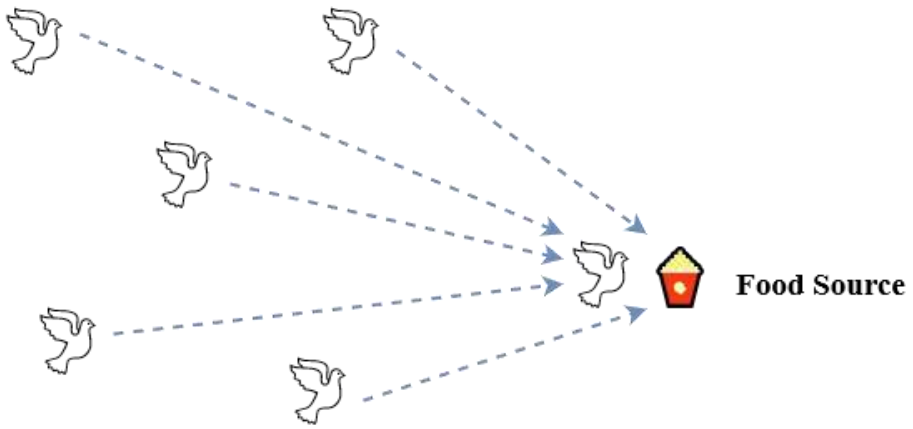

Figure 3.10 An illustration of the foraging which a flock of birds orients towards the best position.

PSO consists with swarm of particles that are candidate solutions [36]. Here, swarm corresponds to the population in other evolutionary optimization algorithms. Particles express the solution as a position vector [14]. The size of the vector may vary depending on the size of the associated problem. In addition to the current position vector, particles contain velocity vector and memory position vector information [14]. The velocity vector determines the positions of the particles to move in each iteration [36]. The memory position vector, on the other hand, contributes to the new position determination by keeping the best position of the particle. This vector can be named as *pbest*, short for particle best. The sample particle illustration with the vectors of the particle is given in Figure 3.11.

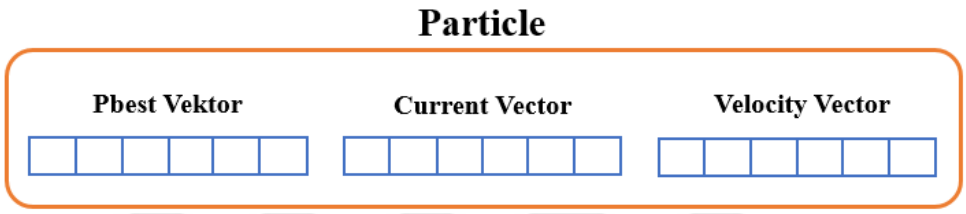

Figure 3.11 Illustration of the particle with vectors

The algorithm starts with a random distribution of particles in the search space. This means that the swarm of particles is randomly generated. In each iteration, the position of the particles is updated relative to the velocity vector. Also, the velocity vector is recalculated at each iteration. The velocity vector is calculated with values such as global best point (*gbest*), *pbest*, initial inertia, and current velocity [36]. *gbest* is the best position discovered by the particles in the swarm, while *pbest* is the best position discovered by the particle. The contribution amount of these best values also depends on the randomness and the initial value coefficients, such as the cognitive and social weights. These coefficients are also called acceleration coefficients [14, 47]. When the current position is subtracted from *pbest* and *gbest* and multiplied with these acceleration coefficients, the *gbest* part is called social cognition or social-learning term, and the *pbest* part is called self-cognition or self-learning term [36, 47]. Particles far from the best locations will have a higher velocity value because they need to be displaced more, while particles that are closer will have a smaller value. But here, initial inertia, coefficients, and random values are assumed to have the same value.

Calculations of velocity value and position update formulas are given in 3.12 and 3.13 [14, 47].

$$\vec{v}_i(t+1) = w\vec{v}_i(t) + c_1 r_1(\vec{p}_i - \vec{x}_i(t)) + c_2 r_2(\vec{g}_i - \vec{x}_i(t)) \quad \textbf{(3.12)}$$

$$\vec{x}_i(t+1) = \vec{x}_i(t) + \vec{v}_i(t+1) \quad \textbf{(3.13)}$$

where meaning of the formulas can be detailed as: $\vec{v}_i(t)$ current velocity vector, $\vec{v}_i(t+1)$ updated velocity vector, $\vec{x}_i(t)$ current position vector, $\vec{x}_i(t+1)$ updated position vector, $\vec{p}_i$ *pbest* position vector and $\vec{g}_i$ identifies *gbest* position vector. t represents the current iteration that $0 \leq t \leq t_{max}$, while i identify index of the particle vector that $0 \leq i \leq d$ [14, 47]. r_1 and r_2 are two random variables that can be either 0 or 1 [14]. c_1 and c_2 are two initial positive constants [14] that called as cognitive and social coefficients. w is the initial constant that called as inertia value [14, 47]. A large inertia value supports global search, while a small value supports local search [48]. Increasing the local search with a small value may cause a more detailed search locally, and very long iterations may cause the global optimum to be found late or never. Performing a global search with a large value can cause large displacements that moving away from the global optimum and preventing access to the global optimum. For this reason, it is important to give a balanced w value in terms of both global search and local search [14]. It is also possible to set the dynamic w value, which is initially set to large and gradually reduced [14, 49, 50]. Vectorial illustration of Formula 3.12 and Formula 3.13 is given in Figure 3.12.

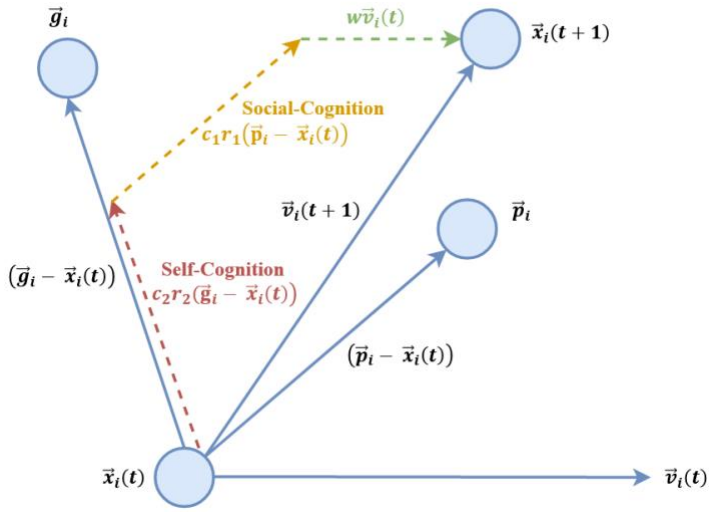

Figure 3.12 Vectorial geometric illustration of particle swarm position update [47]

After the position is updated, the *gbest* value of the swarm and the *pbest* values of the particles are checked. If a better location than *gbest* is available in swarm, the best location is updated as new *gbest*. The current positions of the particles that have discovered a better position than the previous *pbest* will be updated as the new *pbest* values.

The PSO performs the following steps:

Step 1: Define initial constants w, c_1 and c_2

Step 2: Generate swarm that set of particles which are random candidate solutions

Step 3: Evaluate the fitness value of each particle in the swarm

Step 4: Compare and find *pBest* for each particle, and *gBest* for the swarm

Step 5: Perform position update for each particle (calculation of new velocity and position)

Step 6: Repeat steps 3-5 until a certain number of iterations or stopping conditions are met

3.2.3.1 PSO Algorithm Design to the Problem

Since the PSO algorithm is suitable for the continues problems, the Binary PSO algorithm, which is suitable for the discreate problem in this study, is preferred. The implementation of Binary PSO in this study starts with the definition of constant values

for population (swarm) size (n), number of iterations, cognitive (c_1) and social (c_2) coefficients and w inertia weight. Particles consists of velocity, *pbest* and current position vectors shown in Figure 3.11. The population of particle, that velocity and current position vectors, are randomly generated as many as the number of population sizes. Unlike the standard PSO algorithm, the positions in the particle are expressed as binary in the Binary PSO algorithm. Therefore, the position vector is generated according to the indirect encoding method mentioned in section 3.1.3 and expressed as the binary CONV matrix shown in the Figure 3.5. Random generation of particles ensures random distribution in the search area. The velocity formula in the velocity vector Formula 3.12 is utilized in Binary PSO while the particles are moving in each iteration; however, since the positions are expressed as binary, the position update technique for binary PSO mentioned in the study of Alsaidy et al. [9] is preferred for position update.

According to the technique, velocity is expressed as a matrix of the same dimensions as the position vector. The value in the same column and row in the position vector that corresponds to the row with the highest value in the column of the task in the new velocity vector obtained with the velocity formula in Formula 3.12, is updated as 1. In this way, a new position vector is obtained by finding the virtual machine assignment of each task in the velocity vector.

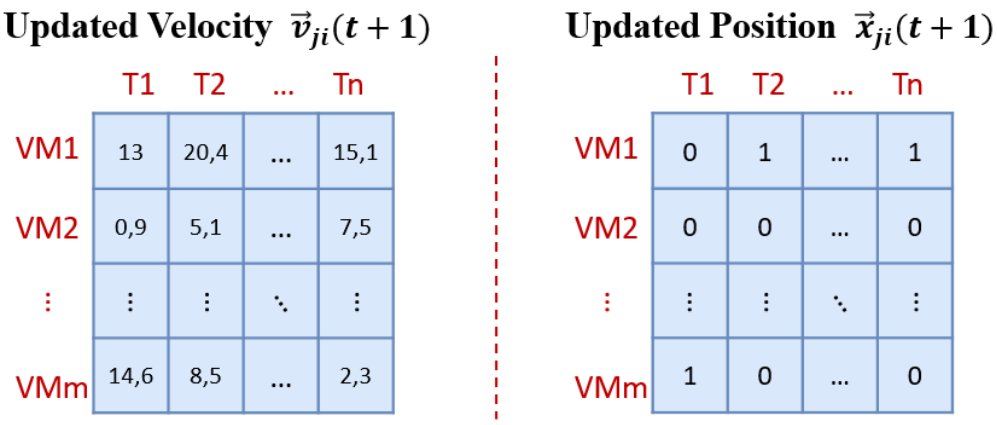

Figure 3.13 Position update method [9]

In Figure 3.13, obtaining the current position vector from the updated velocity vector matrix is given. As it can be seen, for example, since the highest value in the T_1 column

is in the VM_m row, that field is marked as 1 in the position vector so that T_1 is assigned to the VM_m. In this way, the PSO algorithm is adapted to the discreate task scheduling problem in this study. When the current position is obtained, the cost (fitness) value is calculated. The current position is compared with *pbest* and *gbest*, these vectors are updated if the current position is better than *pbest* or *gbest*. These steps continue until the number of iterations that given at the beginning is reached or a solution vector with an imbalance degree value of 0 is found. The pseudocode of the algorithm is given in Figure 3.14.

```
Binary PSO {
Initialize algorithm constants (population size (n), iteration
size, cognitive (c1), social (c2) coefficients and inertia
weight (w))
Generate swarm of particles with random velocity and position
vectors
Calculate cost (fitness) values of the particle in the swarm
Assign pbest of the particles and gbest of the swarm
while (i < iteration size and not reached to global optimum) do
{
   j=0
   while (j < n) do {
       Calculate new velocity vector
       Create new position vector from new velocity vector
       Calculate cost (fitness) value of the newly updated
particle
       Check and update pbest of the newly updated particle
       Check and update gbest of the swarm
        j++

   } End while

  i++

} End while
```

Figure 3.14 Pseudo code of the Binary PSO for the problem

3.2.4 Genetic Algorithm

The genetic algorithm (GA) is a type of evolutionary algorithm that was introduced by John H. Holland in 1975 [29, 36, 51]. In fact, Holland was not concerned with optimization problems, but with the adaptation of systems to their surroundings [29]. The detailed study of the genetic algorithm for optimization was later covered by Holland's student Kenneth De Jong in his doctoral thesis "An analysis of the behavior

of a class of genetic adaptive systems" [29]. But the history of GA dates back to the 1950s. Nils Barricelli developed an artificial life to simulate evolutionary processes in a computer environment and conducted experiments on the possibility of evolution. Barricelli was the first people to develop genetic algorithm software to simulate the evolutionary process in 1954 [29, 52]. Later, Alexander Fraser in 1957 and Hans-Joachim Bremermann in 1958 developed computer simulations to study evolution one after another [29, 53]. Also in 1957, George Box developed the evolutionary processing technique, although not exactly a GA, for solving real problems in the industry, as opposed to simulating evolution [29, 54]. In 1966, Lawrence Fogel prepared the first book on early genetic algorithms for engineering problems [29, 55]. These studies in the 1950s and 60s formed the basis of the GA for the biological evolution and optimization problem, with the studies of Holland and Jong, this field was further paved, and studies increased rapidly in the 70s and 80s.

GA developed inspired by Darvin's natural selection, or "survival of fittest", and Mendel's biological laws [30, 36]. Natural selection says that the best in nature will survive, and these good traits will be passed on to the next generation [29]. For example, hereditary characteristics that will provide advantages in abilities such as hunting among individuals will increase the chances of survival and reproduction of individuals with these characteristics. This situation will increase the probability that genes with these traits will be passed on to subsequent generations, increasing the amount of this trait in the gene pool and decreasing the number of others. GA simulates this process of natural selection to reach the global optimum in each generation by eliminating bad individuals in the generation and including better individuals to the population [33]. Simulation of natural selection in GA is performed with repetitive base genetic operators such as selection, crossover, and mutation [36]. In the GA, candidate solutions are expressed as chromosomes. As in reality, chromosomes are formed by the combination of genes. Genes can be thought of as the smallest meaningful unit that will generate the solution. For example, a gene may represent a value in one dimension of an n-dimensional problem. More than one chromosome forms the gene pool, that is, the population. The population example consisting of chromosomes and genes is given in Figure 3.15.

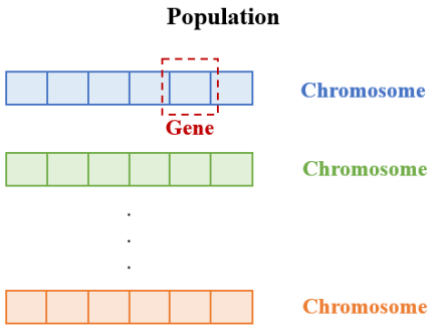

Figure 3.15 Illustration of population, chromosomes, and gene

As in other optimization algorithms discussed in this study, problem-specific encoding of the candidate solution (chromosome) is required in genetic algorithms. Commonly, binary encoding, in which genes are a combination of 0s and 1s, permutation encoding in which genes represent rank, and value encoding in which genes express problem-specific value, are used [30, 31]. Encoding examples of these classes are shown in Figure 3.16.

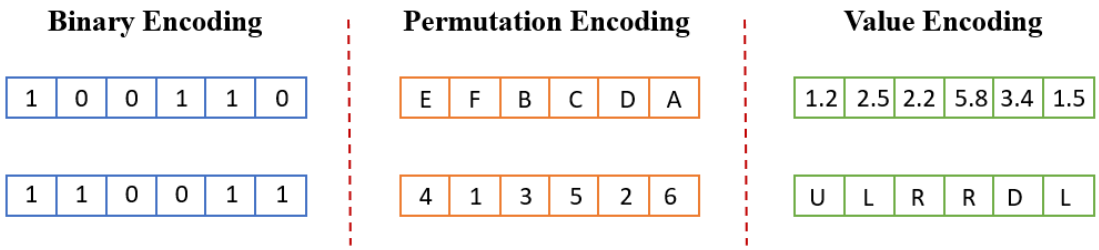

Figure 3.16 Chromosome encoding examples belongs to different encoding types

The GA begins with the random generation of the initial population. Here, there are studies that allow the initial population to start with a better initial population using heuristic algorithms or hybrid methods. Since the GA simulates natural selection, it is important to select the good individuals in the population and transfer the traits of these individuals to the next generations. The selection of the best parents with good fitness for transmission to the next generation is ensured by the selection method. The selection method increases the probability of selecting chromosomes with good fitness over chromosomes with poor fitness [36]. Selection methods are used to choose parents for mating, as well as used to decide which individuals will survive after new

offspring are created. The most common parent selection methods in the literature can be listed as roulette wheel selection, rank selection, tournament selection, and truncation selection. Parent selection methods are also shown in Figure 3.17 [56].

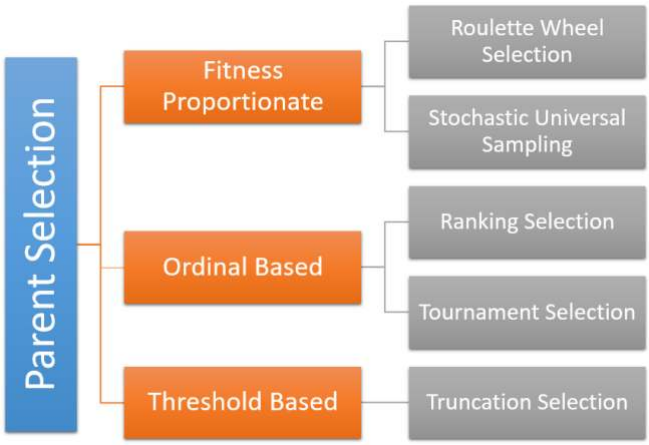

Figure 3.17 Most common parent selection methods in the literature [56]

The roulette wheel selection is a type of fitness proportionate in which the individual with a high fitness score is more likely to be selected [31, 56]. Rank selection is an ordinal based selection method that selection probability of an individual being selected depends on the rank of fitness in the population. The fitness values of individuals are normalized to the total fitness value in the population in the rank selection [56]. In the tournament selection, on the other hand, k number of chromosomes in the population are selected randomly [31, 56]. The chromosome with the best fitness value of the k-selected group is chosen as a parent. Other parents are selected with following similar way. In the truncation selection method, a certain fraction of the population is selected as parents. In this method, the chromosomes are sorted according to their fitness values, and the threshold value called *Trunc* is used to determine how many chromosomes will be chosen as the parents. The best sequenced chromosomes conforming to the *Trunc* threshold value are selected as the parent. The comparative illustration of these selection methods is given in Figure 3.18, where the k value is 3 in the tournament selection and the *Trunc* value is 2 in the truncation selection.

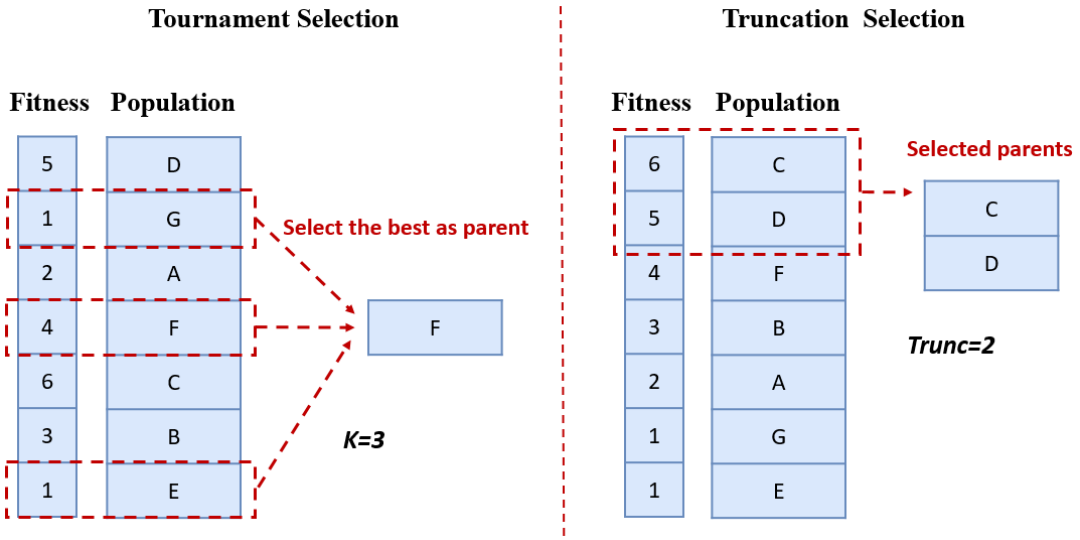

Figure 3.18 Comparative example of tournament selection and truncation selection

Survivor selection is the method used to select which individuals will survive in the population, and which of the newly produced offspring will be added to the population. Survivor selection methods encountered in the literature can be listed as age-based selection and fitness-based selection methods. Survivor selection methods are shown in the Figure 3.19 [56].

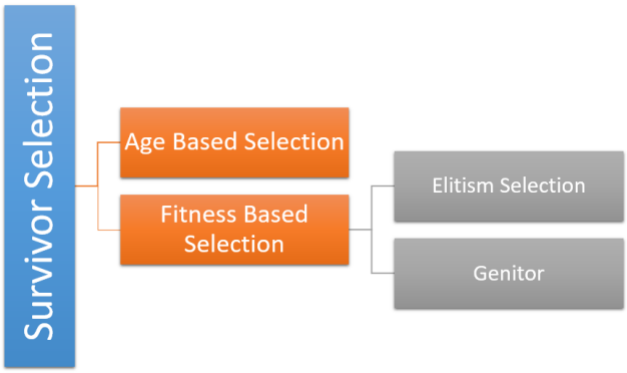

Figure 3.19 Survivor selection methods in literature [56]

In age-based selection, each individual in the population has a specific age. In the event that individuals exceed a certain age, regardless of their fitness score, they are removed from the population and replaced by new offspring. In fitness-based selection, the generated offspring are replaced with chromosomes with poor fitness value in the

population [56]. Elitism is a member of the fitness-based selection. In elitism, chromosomes with a certain number of good fitness scores are directly included in the next population [31, 32, 56]. Therefore, elitely selected individuals are directly included in the new population, while the rest of the population is complemented by new offspring. Elitism preserves the good solutions obtained, but it may also lead to a reduction in diversity by dispersing elite genes into the population. Genetic operators such as mutation can help overcome this situation.

After completing the parent selection process, crossover and mutation genetic operators are used to generate new individuals for the next generation. It is aimed to obtaining a better child by mixing the genes of the selected parents through the crossover process. Selected parents with good genes can pass these genes on to their children, resulting in a better individual. If the crossover is performed from a single point at random, it is called single-point crossover, if it is performed from more than one point at random, it is called multi-point crossover, and if the crossover of each gene is controlled depending on a certain probability, it is called uniform crossover [30]. Crossover types are shown in the Figure 3.20.

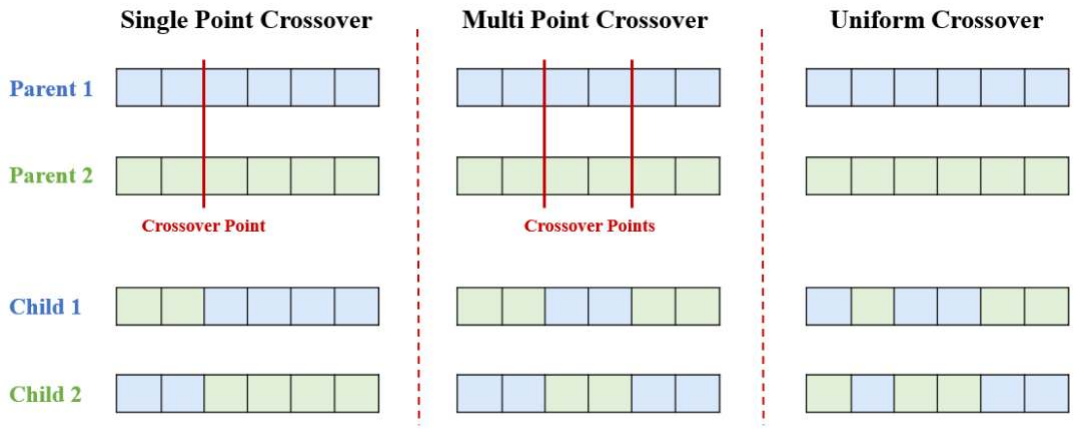

Figure 3.20 Comparison of the crossover methods

After the crossover process is applied, the mutation genetic operator is applied to prevent the individuals in the population to become increasingly similar to each other and to avoid the possibility of sticking in the local optimum [30, 36]. Mutation operation is performed by randomly changing the genes in the generated offspring

according to a certain probability [30]. The mutation process takes place according to the determined mutation rate. The operation is performed by comparing the randomly generated value with the mutation rate. In the case of a large mutation rate, the chromosome may undergo further changes and lose its good position.

The GA the performs following steps:

Step 1: Define initial constants like population size, selection constant (*k*, *Trunc*), mutation ratio

Step 2: Generate population that set of chromosomes which are random candidate solutions

Step 3: Evaluate the fitness value of each chromosome in the population

Step 4: Perform parent selection to be used for chromosome generation

Step 5: Perform crossover to selected parents to generate new offspring

Step 6: Perform mutation to offspring

Step 7: Perform survivor selection to decide which chromosome will be in the new population

Step 8: Repeat steps 3-7 until a certain number of iterations or stopping conditions are met

3.2.4.1 GA Algorithm Design to the Problem

The implementation of GA in this study starts with the definition of constant values for population size (*n*), number of iterations, selection size (*Trunc*), crossover ratio. The population of candidate chromosomes is randomly generated as many as the number of population sizes according to the value indirect encoding method mentioned in section 3.1.3. The size of the chromosome is equal to the number of tasks to be assigned to the machines. Each gene on the chromosome represents a machine assigned to the corresponding task. The degree of imbalance value in Formula 3.5 is used to calculate the fitness or cost value, which is the evolution of each chromosome in the randomly generated population. As a parent selection, truncation selection method shown in Figure 3.18 is used. The selection method creates a parent list as much as the selection size (*Trunc*). Two chromosomes randomly selected from the parent list, that created by parent selection, are crossed with a uniform crossover, and mutated with the mutation rate, resulting in two new offspring. The cost values of these

new individuals are calculated, and the best one is entitled to enter the new population. Elitism is used as survivor selection, so the parents selected by the selection method are added directly to the new population. Therefore, the new population is completed with newly created offspring in addition to the individuals directly transferred from the old population until the number of populations is reached. These steps continue until the number of iterations that given at the beginning is reached or a solution chromosome with an imbalance degree value of 0 is found. The pseudocode of the algorithm is given in Figure 3.21.

```
GA {
Initialize algorithm constants (population size (n), iteration
size, selection size (Trunc), crossover ratio)
Generate population of random chromosomes
Calculate cost (fitness) values of each vectors
while (i < iteration size and not reached to global optimum) do
{
   Create new empty population
   Select and create parent list according to selection size
(Trunc)
    Add selected parents to new population
    j = 0
   while (j < (n – selection size) do {
      Randomly select two parents from parent list
      Perform uniform crossover
      Perform mutation
      Calculate cost value of the generated offspring
      Check and add best offspring to new population
      j++
  } End while
  i++
} End while
```

Figure 3.21 Pseudo code of the GA for the problem

3.3 Parallelization of the Selected Algorithms

Metaheuristic algorithms discussed in this study in the previous chapters, as in other metaheuristic algorithms, have processes based on randomness such as natural selection, random population generation, crossover of individuals, mutation, herd, and social behavior. These processes are important in solving NP-Hard problems such as task scheduling, which are very difficult or impossible to solve with classical optimization methods. The amount of influence of these random processes affects the convergence speed of iterations, getting stuck in local optimums, and finding the

global optimum. These influence ratios are determined by initial constants in metaheuristic algorithms. For this reason, the optimum initial constant value is important for issues such as the convergence and reaching the global optimum. However, despite the optimal initial constant values, it may be more difficult to approach the global optimum because the convergence speed of the iterations may be slow for the problems with large search space of metaheuristic algorithms. Researchers develop methods to increase the search performance of metaheuristic algorithms; for example, the parameters that affect local and global search are made adaptive parameters, or in hybrid methods, algorithms with good global search performance and good local search performance are combined. Parallelization provides benefits such as faster solution discovery, runtime savings, more extensive search for problems that have a large search space, and efficient use of resources [58]. At this point, the parallelization of metaheuristic algorithms for problems with slow convergence and a large search space may be a good method to increase convergence to the global optimum by increasing the search capacity.

In the following sections, parallel methods for applying metaheuristic algorithms in the literature and the algorithm-independent parallelization method proposed for metaheuristic algorithms will be discussed.

3.3.1 Parallelization

Parallelization can be defined as dividing a task into subtask clusters and running concurrently on multiple processors [59, 60]. Parallelization can be performed as data, or control parallelism. Running the same procedure simultaneously on multiple data chunks can be expressed as data parallelism. The simultaneous execution of different procedures on multiple processors with their own data sets is called control parallelism. Therefore, p number of processors executes different procedures at the same time [58, 61].

3.3.2 Parallelization of Metaheuristic Algorithm

There are many studies in the literature on the parallelization of metaheuristic algorithms, especially genetic algorithms. In this study, since the other metaheuristic

evolutionary optimization algorithm, such as the genetic algorithm, is a population-based algorithm chosen for the solution of the problem, the parallelization methods applied to the genetic algorithm in the literature will be examined in order to shed light on the parallelization of other metaheuristic algorithms. The parallelization of the genetic algorithm in the literature is based on methods that a single global population or divide population into multiple subpopulations [59]. The method based on a single global population is the standard parallel approach [59, 60]. In this approach, the algorithm runs sequentially on a single population, while steps such as evolution and reproduction are executed in parallel. In the decomposition approach based on multiple subpopulations, the population is divided into subpopulations, and genetic algorithm processes are executed on separate processors at the same time [60]. In addition, terms such as deme, migration, synchronous/asynchronous, topology are encountered in the parallelization of genetic algorithms in the literature. Deme means a separated subpopulation [58]. In methods executed by distributing subpopulations (demes) to separate processors, the individual transfer between processors at regular intervals is expressed as migration. Migration operation can be performed according to the topology in which the processors are neighboring. Synchronous parallelization refers to the parallel coordinated operation of subtask execution with processors by migration with data transfer at specific intervals, whereas asynchronous parallelization refers to the independent operation of processors with limited data transfer [59]. The genetic algorithm parallelization methods commonly encountered in the literature are shown in Figure 3.22. Parallel genetic algorithms can be classified as global single population master-slave (GPMS) (or distributed fitness evaluation), massively parallel (MP) (or single population fine gradient), and multiple deme (MD) (or distributed, island model) [60].

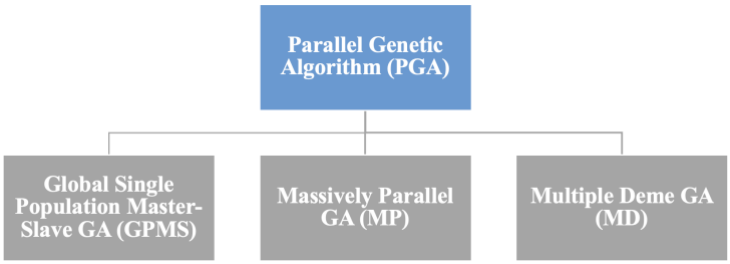

Figure 3.22 Classification of the parallel genetic algorithm

The global single population master-slave algorithm consists of a global single population as the name suggests. While sequential operations are performed on the global population, operations such as fitness evolution and mating are parallelized to the improve performance of the algorithm. This method is based on the master-slave paralleling method shown in the Figure 3.23. The global population is stored on the master, and the algorithm operations are performed there. In the fitness evolution part, the master assigns a certain number of individuals in the population to calculate the fitness values of the slaves [60]. Parallelization is generally preferred in the fitness evolution part because of not requiring communication [59]. GPMS can be performed in both synchronous and asynchronous modes. In synchronous, the master waits for slaves to complete their tasks before moving on to the next generation. Synchronous GPMS is faster than GA and simpler to implement, but the slowest processor could be a bottleneck. In asynchronous, master do not require to wait to slaves but differ in GA by requiring tournament selection to cover missing individuals [59, 61]. GPMS also has no restrictions on the use of computer architectures such as shared-memory and distributed memory computers.

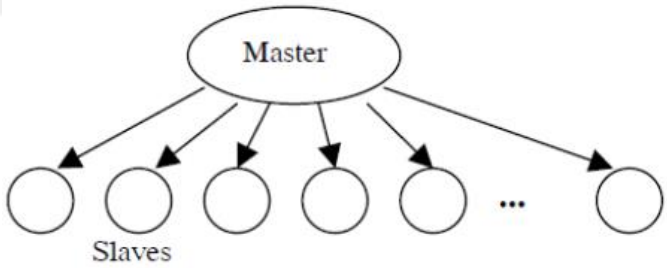

Figure 3.23 Master-slave architecture [60]

MP, like GPMS, has a single population but is designed for massively parallel computers. It restricts instruction between individuals, and the selection and crossover phases are carried out by processors in a small neighborhood. Individuals migrate in the neighborhood to increase diversity. [59, 60]. Different neighborhoods are shown with different sizes (4, 8, 12) and shapes in the Figure 3.24.

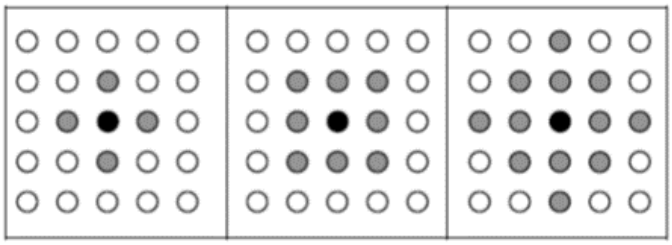

Figure 3.24 Different neighborhoods with different sizes (4, 8, 12) and shapes [60].

Multiple Deme GAs consist of numerous, relatively isolated subpopulations (deme). Individuals in the demes occasionally migrate to another deme as part of this migration process. The configuration of the algorithm determines the number and size of demes, as well as the frequency and rate of migration. Assume that, if a problem has 800 individuals in a population that will run on 8 core processors of system, population can be divided equally across to each processor so that, each of them includes 100 sized demes. Normal genetic algorithm is performed on these demes in each processor separately, and after a specific number of generations i.e., 5, some selected individuals are swapped between demes (migration) to increase genetic diversity [62]. In this way, 8 independent core searches are performed in a parallel way.

3.3.3 Parallelization Method

In order to increase the search performance of metaheuristic algorithms, unlike the hybrid metaheuristic methods or other methods in the literature, a hybrid parallelization method is proposed. This method is created by taking advantage of the global population master-slave and multiple-deme methods to be independent of the metaheuristic algorithm and to implement flexible. The fact that the global population is common in the GPMS method requires coordinated work when slave communication is synchronous, whereas it requires the use of independent and collaborative techniques to ensure that slave processors work independently and properly when slave communication is asynchronous. At this stage, the concept of a separate subpopulation in MD can improve the search performance of the algorithm by allowing slaves to search entirely separately. In this way, search workers are created that search independently from each other in the search space and compete with each other to achieve the targeted best result. In addition, the algorithm will be more flexible

in adapting to the problem compared to synchronous/asynchronous GPMS. Since algorithms discussed in the study are already stochastic methods, each core with its own sub-population already has a sufficient variety of solutions. Although the selected candidate solutions from the neighboring core with migration are aimed to increasing the population diversity in the cores, the steps of the algorithms, such as clonal expansion, hypermutation, crossover, and mutation, already increase the diversity in each iteration. Therefore, migration between subpopulations was not preferred to make the method algorithm-independent and flexible.

The pseudocode of the parallelization method proposed in this study is shown in Figure 3.25. As can be seen, the parallelization model is a model with isolated deme (subpopulation) working as a master-slave. Unlike the GPMS model, each slave has its own subpopulation instead of a common population. The master process starts with the definition of various algorithm constants such as iteration size, population size, algorithm-specific constants such as mutation ratio, and parallelization constants such as the number of processors. The master process then creates slave processes. Slave processes create their own subpopulations and apply sequential algorithm steps. The first process to reach the global optimum sends a message to other processes to stop the search. Other slave processes that receive the message stop searching and return to the master. Since this proposed parallelization model is algorithm-independent, it has been applied jointly to the chosen metaheuristic optimization algorithms in this study.

```
Master Process {
Initialize algorithm constants (population size, iteration size,
etc.)
Initialize parallelization constant (number of processors)
while (i < number of processors) do {
   Create slave processes to perform metaheuristic algorithm
   i++
} End while
} End Master Process

Slave Process to perform chosen metaheuristic algorithm {
while (termination criterion is not met) do {
   Generate subpopulation
   Perform steps of chosen metaheuristic algorithm
   if (global best is found) do {
     Send a found message to other processes
     Stop iteration and return master process
   } End if
   else if (found messages, that send by other slaves, exist) do
{
     Stop iteration and return master process
   } End else if
} End while
} End Slave Process
```

Figure 3.25 Pseudo code of the proposed parallelization method

CHAPTER 4

EXPERIMENTS & RESULTS

In this section, the search performances of the proposed CSA, GA, DE, and PSO and their parallel versions with different scenarios are evaluated and discussed. The serial and parallel versions of the algorithms will be compared in different scenarios, and subsequently, all algorithms will be compared in general. But first of all, more general and common information about the tests will be given.

4.1 Preliminary General Information

Algorithms are implemented with the Java Fork/Join Framework. It allows dividing a problem into sub-problems with fork. A fork can be seen as an independent task that running on a thread. The solutions of sub-problems are joined to obtain an overall solution [63]. The cloud environment parameters created for the numerical test simulation of the algorithms are given in Table 4.1. Task (MI) and VM (MIPS) sequences are randomly generated and saved according to the parameters given in Table 4.1 as datasets. Moreover, all algorithms are tested with the same datasets to make the comparison fairer.

Table 4.1 Cloud simulation parameters

Parameters	Values
Number of Tasks	64-256
Number of VMs	14-28
Size of Tasks (MI)	10000-80000
Execution Rate of VMs (MIPS)	1000-4000

The task sequence includes the MI values of the task size between 10000 and 80000, with the length between 64 and 256 task numbers given in Table 4.1. The task sequence can also be thought of as tasks in the queue to be assigned to the appropriate VMs by the scheduling algorithm in the datacenter broker, which indicates the size of the user task requests. Similarly, the VM sequence includes the MIPS values of the

execution rate between 1000 and 4000, with the length between 14 and 28 VM numbers. Therefore, the VM sequence identifies the execution rate of each VM.

Since the tested algorithms (CSA, GA, DE, and PSO) are stochastic, they may not find similar results in every execution. Therefore, the test results shown in the graphs and tables are the average cost and iteration values obtained as a result of running the algorithms 10 times. In a single run, the cost value is obtained with the best value that is found with the execution of the given number of iterations, while the iteration values are obtained with the iteration that found the best cost. Parallel algorithms have four slave processes, and all slaves have the same parameters (population size, selection size, iteration size, mutation ratio, etc.) with compared to the sequential version. The parallel test results in the figures are the first iteration and cost value that found the best result among the slaves before averaging in a single run. For example, after the best cost value is found by a slave in one iteration, the iteration is not updated for the cost value that another slave finds in further iterations, which is the same or worse, and the first found iteration by first slave is accepted as the result. However, if a better cost than the previous one is found by any slave in further iterations, the new iteration value is obtained as the result. In other words, although each slave searches independently in its own subpopulations in the parallel algorithm, the results that achieve the best between the searching slaves are taken into account when evaluating the algorithm as a whole.

Figure 4.1 The coverage graph of the parallel algorithm with 4 slaves

The coverage graph of the parallel algorithm with 4 slaves and each slave running 25 iterations is given as an example in Figure 4.1. Although process 0 in the first 11 iterations generally achieves better results than other slaves, the result obtained by process 2 is taken as the final result, since the result found by process 2 in the 19th iteration is the best compared to the others. The cost and iteration results given when comparing the serial and parallel versions of the algorithms are evaluated as follows: If the cost value of the parallel algorithm improved compared to the serial version, it was considered successful regardless of the increase or decrease of the iteration found, because even if the iteration of the result increased, the slaves of the parallel version under similar conditions obtained a better solution than the serial algorithm.

4.2 Experiments & Comparisons

This section compares serial and parallel versions of the algorithms one by one in a variety of experiments. Furthermore, after serial and parallel comparisons of the algorithms, all algorithms are compared in general at the end. Since the test results are achieved by running the algorithms 10 times, the statistical results of the produced cost and iteration values is detailed in the tables, and the average values are contrasted in visual graphics.

4.2.1 CSA and PCSA Comparisons

In this section, in order to compare the performance of CSA and PCSA algorithms, various experiments will be performed according to population size change, selection size change, and dataset that different problem size. The best found cost values and the iteration values that found the best cost value will be given and compared statistically.

4.2.1.1 Experiment 1 (Population Size Change)

In this scenario, the number of populations ranges from 500 to 4000, while other parameters are kept fixed. The values of the parameters that are kept fixed during the scenario, are as follows: 128T 14M problem size, 500 iterations, 5 selection size, 0.01-0.2 mutation ratio. Statistical cost and iteration results are given in Table 4.2, and

average cost and iteration comparisons are given in Figure 4.2 and Figure 4.3, respectively.

Table 4.2 Statistical result of cost and iteration values of CSA and PCSA respect to population size change

	Population	**500**		**1000**		**2000**		**3000**		**4000**	
		CSA	**PCSA**	**CSA**	**PCSA**	**CSA**	**PCSA**	**CSA**	**PCSA**	**CSA**	**PCSA**
Cost	Mean	0.08	**0.063**	0.052	**0.043**	0.01	**0**	**0**	**0**	**0**	**0**
	Min	0.07	**0.04**	**0.04**	**0.04**	**0**	**0**	**0**	**0**	**0**	**0**
	Max	0.11	**0.07**	0.07	**0.05**	0.05	**0**	**0**	**0**	**0**	**0**
	Median	**0.07**	**0.07**	0.05	**0.04**	**0**	**0**	**0**	**0**	**0**	**0**
	Std.	0.013	**0.011**	0.012	**0.004**	0.02	**0**	**0**	**0**	**0**	**0**
Iteration	Mean	**203.7**	205.1	**262.3**	279.7	262.2	**181.2**	163.5	**111.9**	197.4	**88.6**
	Min	**45**	56	80	**39**	85	**62**	73	**62**	45	**25**
	Max	461	**425**	**486**	488	498	**481**	248	**175**	464	**173**
	Median	162.5	**125**	**244.5**	290.5	252.5	**139**	168.5	**112**	218	**72**
	Std.	147.1	**145.5**	**137.2**	159.4	153.5	**124.9**	52.2	**40.1**	132.3	**52.9**

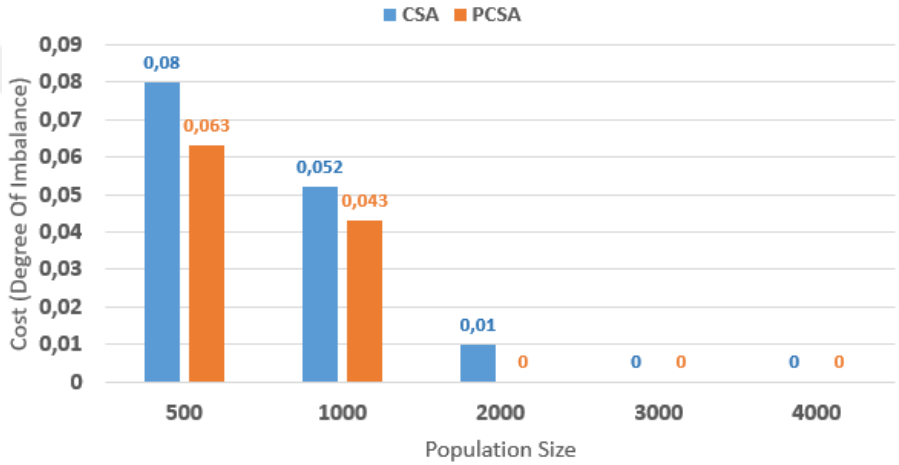

Figure 4.2 Change of cost values of CSA and PCSA respect to population size change

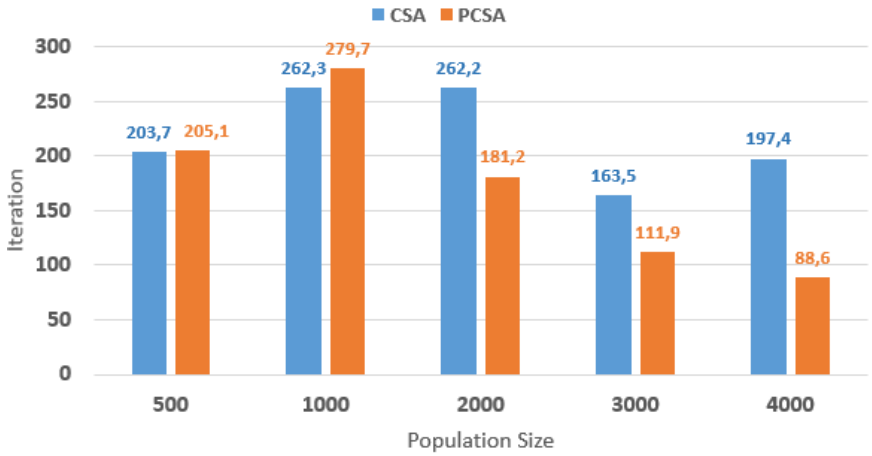

Figure 4.3 Change of iteration values of CSA and PCSA respect to population size change

4.2.1.2 Experiment 2 (Selection Size Change)

In this scenario, the number of selection size ranges from 5 to 25, while other parameters are kept fixed. The values of the parameters, that are kept fixed during the scenario, are as follows: 128T 14M problem size, 500 iterations, 500 population size, 0.01-0.2 mutation ratio. Statistical cost and iteration results are given in Table 4.3, and average cost and iteration comparisons are given in Figure 4.4 and Figure 4.5, respectively.

Table 4.3 Statistical results of cost and iteration values of CSA and PCSA respect to selection size change

Selection		**5**		**10**		**15**		**25**	
		CSA	**PCSA**	**CSA**	**PCSA**	**CSA**	**PCSA**	**CSA**	**PCSA**
Cost	Mean	0.08	**0.063**	0.009	**0**	**0**	**0**	0.017	**0**
	Min	0.07	**0.04**	**0**	**0**	**0**	**0**	**0**	**0**
	Max	0.11	**0.07**	0.05	**0**	**0**	**0**	0.17	**0**
	Median	0.07	**0.07**	**0**	**0**	**0**	**0**	**0**	**0**
	Std.	0.016	**0.011**	0.018	**0**	**0**	**0**	0.051	**0**
Iteration	Mean	**191.7**	205.1	347.2	**112.3**	193.7	**74.7**	121.4	**92.2**
	Min	**28.0**	56	159	**82**	78	**17**	22	**21.0**
	Max	442.0	**425**	487	**150**	305	**176**	341	**146.0**
	Median	**90.0**	125.5	353	**112.5**	186	**63.5**	**64**	95.0
	Std.	162.6	**145.5**	105.2	**20.4**	78.8	**44.3**	102.6	**45**

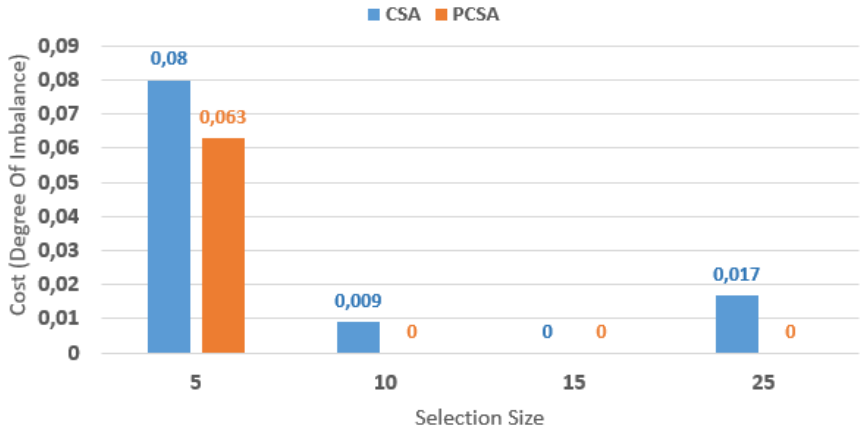

Figure 4.4 Change of cost values of CSA and PCSA respect to selection size change

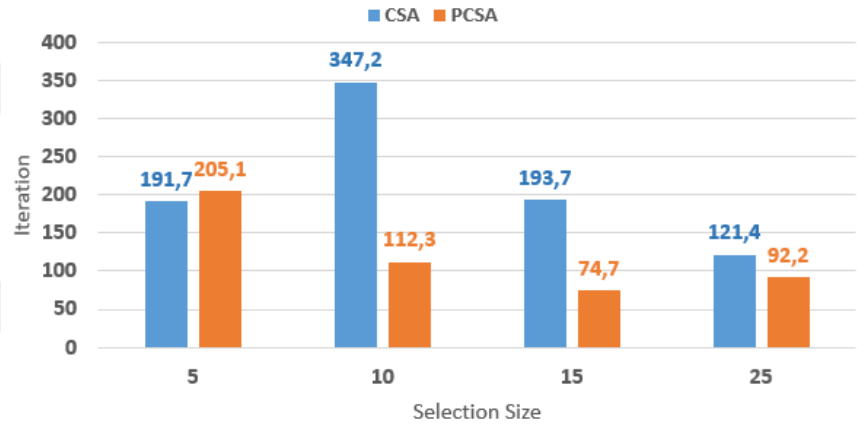

Figure 4.5 Change of iteration values of CSA and PCSA respect to selection size change

4.2.1.3 Experiment 3 (Problem Size Change)

The datasets with different problem sizes in the range of 64-256 tasks and 14-28 VMs are changed, while other parameters were kept fixed during this scenario. The values of the parameters fixed in the scenario are as follows: 15 selection size, 500 iterations, 2000 population size, 0.01-0.2 mutation ratio. Statistical cost and iteration results are given in Table 4.4, and average cost and iteration comparisons are given in Figure 4.6 and Figure 4.7, respectively.

Table 4.4 Statistical results of cost and iteration values of CSA and PCSA respect to problem size change

	Problem	256T 28M		256T 14M		128T 28M		128T 14M		64T 28M		64T 14M	
		CSA	PCSA	CSA	PCSA	CSA	PCSA	CSA	PCSA	CSA	PCSA	CSA	PCSA
Cost	Mean	0.122	**0.094**	**0**	**0**	0.338	**0.213**	0	**0**	0.906	**0.559**	0.195	**0.141**
	Min	**0.07**	**0.07**	**0**	**0**	**0.21**	**0.21**	0	**0**	0.63	**0.27**	0.17	**0**
	Max	0.18	**0.11**	**0**	**0**	0.8	**0.24**	0	**0**	1.31	**0.93**	**0.21**	**0.21**
	Median	**0.11**	**0.11**	**0**	**0**	**0.21**	**0.21**	0	**0**	0.915	**0.575**	0.21	**0.17**
	Std.	0.031	**0.02**	**0**	**0**	0.18	**0.009**	0	**0**	**0.23**	**0.23**	0.018	**0.07**
Iteration	Mean	**216**	229.4	125.9	**62.3**	252.6	**243.2**	85.7	**26**	**120.2**	124.6	**44**	173.3
	Min	78	**69**	44	**29**	41	**19**	17	**15**	**9**	12	**4**	6
	Max	476	**347**	316	**133**	**410**	475	348	**40**	495	**483**	**272**	440
	Median	**201**	226.5	108.5	**48.5**	278.5	**250**	67.5	**26**	**56.5**	66.5	**7**	99
	Std.	112.1	**83.9**	74.1	**31.2**	**119.1**	138.1	92.2	**7.9**	159.4	**134.5**	**78.4**	157.4

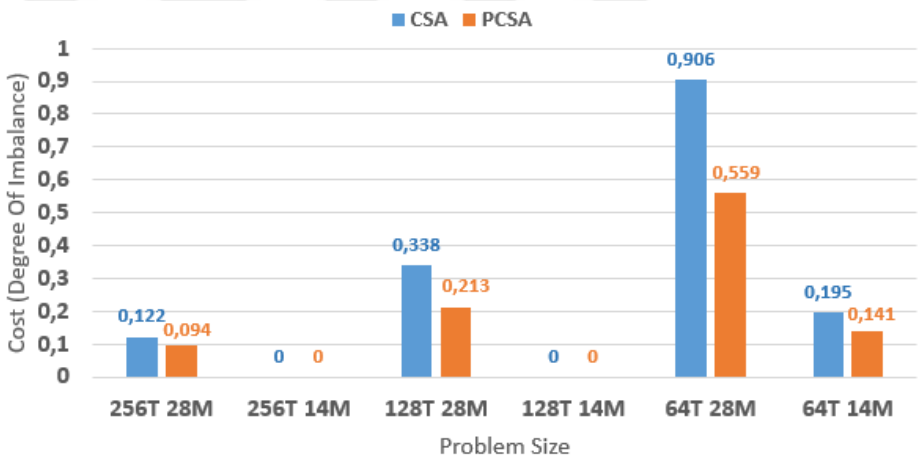

Figure 4.6 Change of cost values of CSA and PCSA respect to problem size change

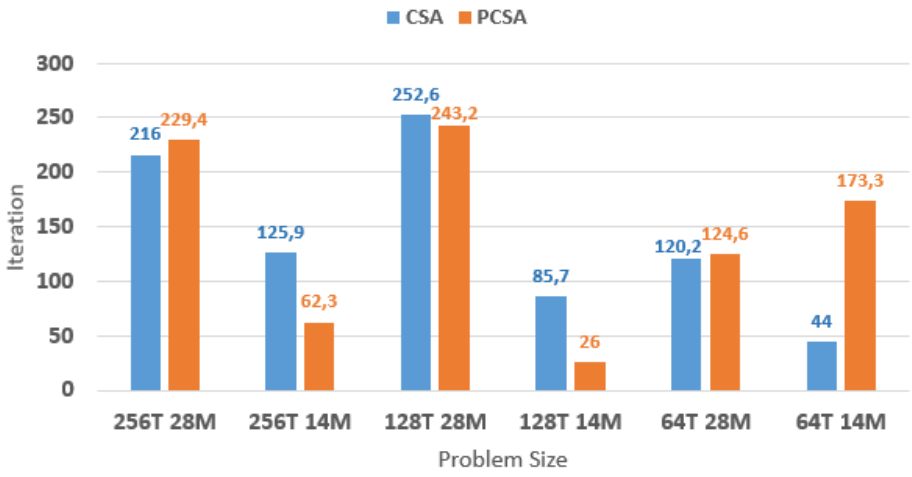

Figure 4.7 Change of iteration values of CSA and PCSA respect to problem size change

4.2.1.4 Evolution of the CSA Experiments

While the statistical results of cost and iteration values obtained in the experiments performed in this section are given in Table 4.2, Table 4.3 and Table 4.4, the average values of cost and iteration values are compared in the visual charts between in the Figures 4.2-4.7, respectively. As seen in Figure 4.2, Figure 4.4, and Figure 4.6, PCSA achieved better cost results than CSA in all experiments. Although both reach the global optimum when the population number is 3000 and 4000 in the first experiment, it is seen in Figure 4.3 that PCSA has reached the previous iterations. Also, although the average number of iterations with the best cost values of PCSA seems to increase in Figure 4.3 when the population is 500 and 1000, the cost values found by PCSA are better than CSA. Similar results are seen in Figure 4.4, Figure 4.5, Figure 4.6 and Figure 4.7 in Experiment 2 and Experiment 3. For example, although all algorithms reach the global optimum, when the selection size is 15 in Figure 4.4 or the problem size is 256T 14M, 128T 14M in Figure 4.6, it is seen in Figure 4.5 and Figure 4.7 that PCSA reaches the result in earlier iterations for the same values. Again, in Figure 4.5 and Figure 4.7, PCSA achieved better results in terms of cost compared to CSA in cases where iteration increased when selection size was 5 in Experiment 2 or problem sizes are 256T 28M, 64T 28M and 64T 14M in Experiment 3. When comparing CSA and PCSA algorithms in the tables, the better value is shown in bold in the own sections. The results in Table 4.3 and Table 4.4 in Experiment 2 and Experiment 3 are

not very different except for the 64T 14M iteration results in Table 4.4. In the tables, especially for the cost values, PCSA obtained results closer to the mean than CSA and a lower standard deviation. In addition to the fact that PCSA achieves better results in terms of cost compared to CSA, this result also shows that PCSA achieves more consistent results.

4.2.2 DE and PDE Comparisons

In this section, in order to compare the performance of DE and PDE algorithms, various experiments will be performed according to population size change, F size change, and dataset that different problem size. The best found cost values and the iteration values that found the best cost value will be given and compared statistically.

4.2.2.1 Experiment 1 (Population Size Change)

In this scenario, the number of populations ranges from 40 to 200, while other parameters are kept fixed. The values of the parameters, that are kept fixed during the scenario, are as follows: 128T 14M problem size, 15000 iterations, 0.05 F, 0.03 CR. Statistical cost and iteration results are given in Table 4.5, and average cost and iteration comparisons are given in Figure 4.8 and Figure 4.9, respectively.

Table 4.5 Statistical results of cost and iteration values of DE and PDE respect to population size change

Population		**40**		**80**		**120**		**160**		**200**	
		DE	**PDE**	**DE**	**PDE**	**DE**	**PDE**	**DE**	**PDE**	**DE**	**PDE**
Cost	Mean	0.038	**0.018**	0.039	**0.009**	0.024	**0.004**	0.018	**0**	0.02	**0**
	Min	0	**0**	0.0	**0**	0	**0**	0	**0**	0	**0**
	Max	0.07	**0.05**	0.05	**0.05**	0.05	**0.04**	0.05	**0**	0.04	**0**
	Median	0.05	**0**	0.04	**0**	0.02	**0**	0	**0**	0.02	**0**
	Std.	0.026	**0.022**	**0.014**	0.018	0.024	**0.012**	0.022	**0**	0.02	**0**
Iteration	Mean	**8639.5**	9805.5	**8806**	9064.4	**8277.8**	9682	9044.4	**6862.6**	8940.3	**5568**
	Min	**2542**	3544	3952	**2099**	**3647**	6036	**1834**	3037	**1721**	3223
	Max	**14211**	14985	**13518**	14738	**13529**	14387	13310	**10495**	14588	**8028**
	Median	**7553**	11886	**9072.5**	9099	**6928**	9117	9761.5	**5983.5**	9067.5	**5497.5**
	Std.	**3996.2**	4210.9	**3057.9**	3962.6	3585.7	**2350.8**	3611.7	**2468.9**	4184.2	**1538.5**

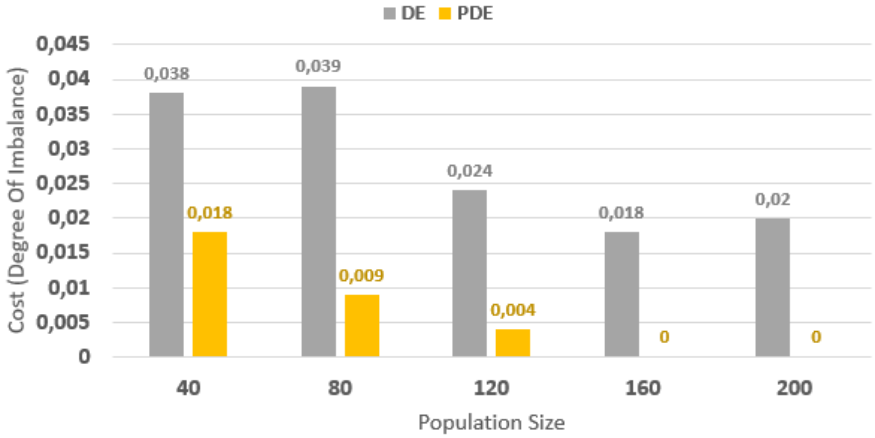

Figure 4.8 Change of cost values of DE and PDE respect to population size change

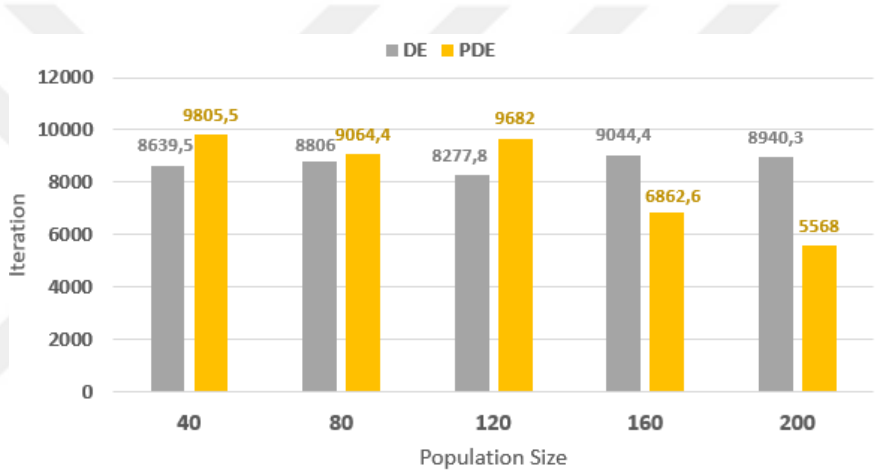

Figure 4.9 Change of iteration values of DE and PDE respect to population size change

4.2.2.2 Experiment 2 (F Size Change)

In this scenario, the number of F ranges from 1 to 0.005, while other parameters are kept fixed. The values of the parameters, that are kept fixed during the scenario, are as follows: 128T 14M problem size, 20 population size, 15000 iterations, 0.03 CR. Statistical cost and iteration results are given in Table 4.6, and average cost and iteration comparisons are given in Figure 4.10 and Figure 4.11, respectively.

Table 4.6 Statistical results of cost and iteration values of DE and PDE respect to F size change

	F Size	1		0.5		0.05		0.005	
		DE	**PDE**	**DE**	**PDE**	**DE**	**PDE**	**DE**	**PDE**
Cost	Mean	0.066	**0.056**	0.062	**0.024**	0.052	**0.018**	0.058	**0.029**
	Min	0.05	**0.04**	0.04	**0**	0	**0**	0	**0**
	Max	0.11	**0.07**	0.11	**0.05**	0.07	**0.05**	0.11	**0.07**
	Median	0.07	**0.05**	0.06	**0.02**	0.07	**0**	0.05	**0.04**
	Std.	0.017	**0.012**	**0.019**	0.024	0.027	**0.022**	0.029	**0.025**
Iteration	Mean	10433.7	**8005.3**	**8312.6**	11411.6	**5934.8**	9379	**8439.8**	9632.3
	Min	6149	**1665**	**2804**	5613	**944**	4047	**1161**	4858
	Max	**14110**	14228	**14196**	14680	13209	**12555**	**13398**	14722
	Median	10374	**8084**	**9335.5**	12481.5	**5053**	9657	8593	**8117.5**
	Std.	**2628.7**	3795.5	3617.2	**3034.8**	4242.5	**2746.4**	3847.2	**3771.2**

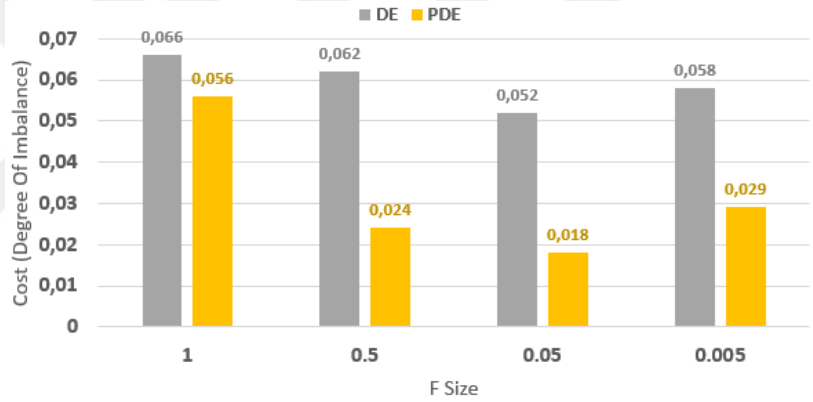

Figure 4.10 Change of cost values of DE and PDE respect to F size change

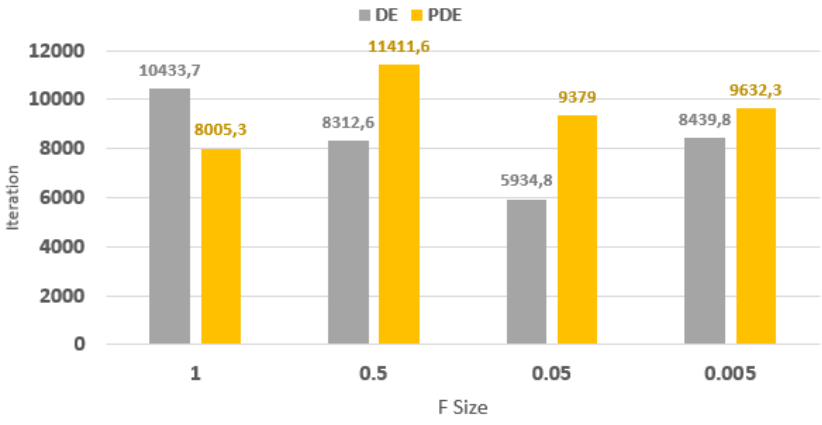

Figure 4.11 Change of iteration values of DE and PDE respect to F size change

4.2.2.3 Experiment 3 (Problem Size Change)

The datasets with different problem sizes in the range of 64-256 tasks and 14-28 VMs are changed, while other parameters were kept fixed during this scenario. The values of the parameters, that are kept fixed during the scenario, are as follows: 200 population size, 20000 iterations, 0.05 F, 0.03 CR. Statistical cost and iteration results are given in Table 4.7, and average cost and iteration comparisons are given in Figure 4.12 and Figure 4.13, respectively.

Table 4.7 Statistical results of cost and iteration values of DE and PDE respect to problem size change

	Problem	**256T 28M**		**256T 14M**		**128T 28M**		**128T 14M**		**64T 28M**		**64T 14M**	
		DE	**PDE**	**DE**	**PDE**	**DE**	**PDE**	**DE**	**PDE**	**DE**	**PDE**	**DE**	**PDE**
Cost	Mean	0.153	**0.14**	0.029	**0.025**	0.968	**0.882**	0.004	**0**	1.423	**1.334**	**0.196**	0.2
	Min	0.14	**0.14**	0.02	**0.02**	**0.58**	0.6	0	**0**	1.23	**1.12**	**0.14**	**0.14**
	Max	0.21	**0.14**	0.03	**0.03**	1.11	**1**	0.04	**0**	1.47	**1.45**	**0.21**	**0.21**
	Median	0.14	**0.14**	0.03	**0.025**	1.01	**0.9**	0	**0**	1.445	**1.35**	**0.21**	**0.21**
	Std.	0.023	**0**	**0.003**	0.005	0.147	**0.103**	0.012	**0**	**0.065**	0.1	0.028	**0.022**
Iteration	Mean	**14345**	15614	**11639**	13473	**8892**	11312	10833	**7869**	6846	**6360**	**3622**	3656
	Min	**9179**	9731	**4669**	8349	**1108**	1145	4719	**3438**	**850**	1487	**664**	848
	Max	**19022**	19825	**17011**	18842	**19316**	19987	19513	**10932**	19759	**15781**	**11015**	11708
	Median	14704	**5820.5**	**12012**	14313	**8576.5**	10917	10929.5	**8236.5**	**1970**	3281	**2080**	2353.5
	Std.	3111.9	**2894.3**	4153.9	**3574.9**	6551.4	**6434.8**	4119.9	**2544.4**	7220.9	**5073.8**	**3382.9**	3521.2

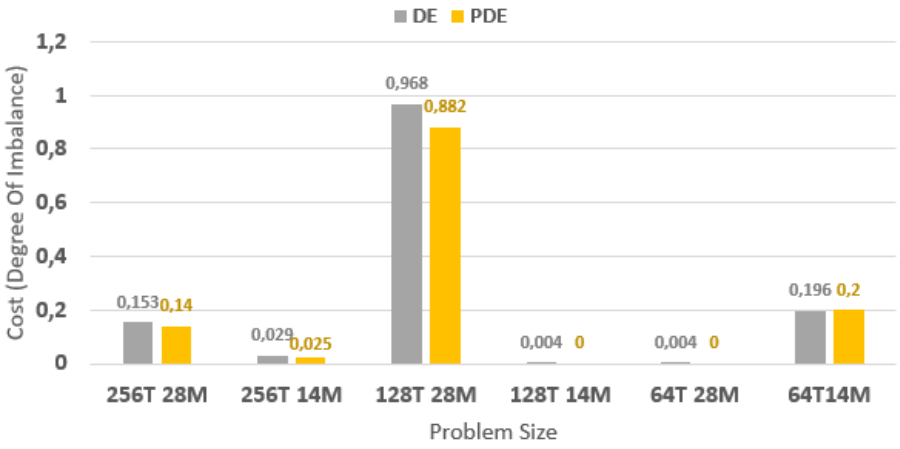

Figure 4.12 Change of cost values of DE and PDE respect to problem size change

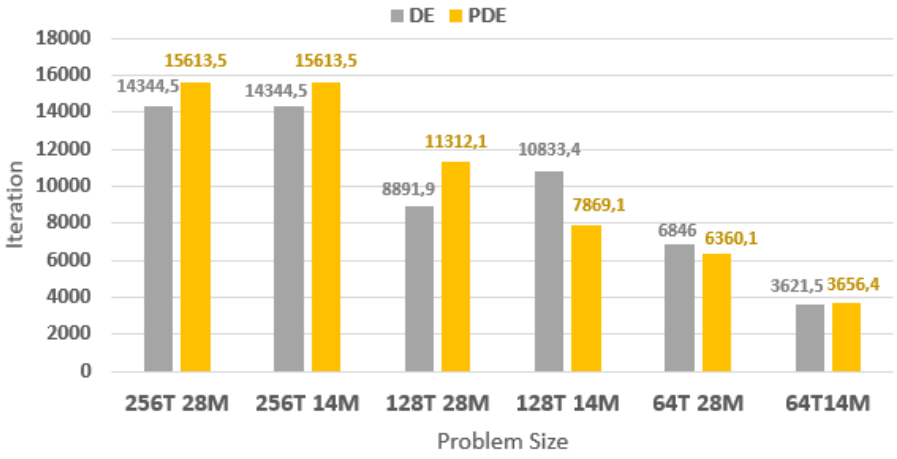

Figure 4.13 Change of iteration values of DE and PDE respect to problem size change

4.2.2.4 Evolution of the DE Experiments

The statistical outcomes of the cost and iteration values obtained in the tests carried out in this part are presented in Tables 4.5, Table 4.6, and Table 4.7, while the average cost and iteration values are contrasted in the visual charts between Figures 4.8-4.13. As can be seen in Figure 4.8, Figure 4.10, and Figure 4.12, PDE achieved better cost results than DE in all experiments, except for dataset 64T 14M in Figure 4.12. In the first experiment, it is seen that the results obtained by PDE especially get better as the number of population increases. When the population number is 160 and 200, it seems that the PDE has reached the global optimum and has reduced the average number of iterations. Although the average number of iterations with the best cost values of PDE seems to increase in Figure 4.9 when the population is 40, 80, or 120, it is seen that the cost values found by PDE for the same population values are better than DE in the Figure 4.8. Similar results are seen in Experiment 2. For example, although the average number of iterations of PDE with the best cost values seems to be increasing in Figure 4.11, it is seen that the cost values found by PDE are better than DE for the same F values in Figure 4.10. It is also seen that the value of 0.05 is the most optimal F value for DE and PDE. Although the results in Experiment 3 are similar to the previous experiments, the results of the 64T 14M dataset in Figure 4.12 show that PDE could not improve the mean cost value compared to DE, but the cost statistical results for this dataset in Table 4.7 are not worse except for the mean value. When comparing DE

and PDE algorithms in the tables, the better values in their respective sections are shown in bold. The results in Table 4.5, Table 4.6 and Table 4.7 show that PDE is generally better, especially in terms of cost values. In addition, the cost values of PDE generally obtained results closer to the mean than DE and achieved a lower standard deviation. This also shows that PCSA achieves more consistent results.

4.2.3 PSO and PPSO Comparisons

In this section, in order to compare the performance of PSO and PPSO algorithms, various experiments will be performed according to population size change, inertia change, and dataset that different problem size. The best found cost values and the iteration values that found the best cost value will be given and compared statistically.

4.2.3.1 Experiment 1 (Population Size Change)

In this scenario, the number of population ranges from 20 to 300, while other parameters are kept fixed. The values of the parameters, that are kept fixed during the scenario, are as follows: 128T 14M problem size, 1000 iterations, 2 c_1-c_2, 0.1 inertia. Statistical cost and iteration results are given in Table 4.8, and average cost and iteration comparisons are given in Figure 4.14 and Figure 4.15, respectively.

Table 4.8 Statistical result of cost and iteration values of PSO and PPSO respect to population size change

Population		**20**		**60**		**100**		**200**		**300**	
		PSO	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**
Cost	Mean	0.231	**0.122**	0.16	**0.104**	0.148	**0.095**	0.097	**0.064**	0.127	**0.054**
	Min	0.14	**0.07**	**0**	0.05	0.11	**0.05**	**0**	**0**	0.07	**0**
	Max	0.64	**0.18**	0.31	**0.11**	0.27	**0.21**	0.17	**0.11**	0.21	**0.11**
	Median	0.205	**0.11**	0.14	**0.11**	0.125	**0.09**	0.11	**0.07**	0.125	**0.05**
	Std.	0.14	**0.027**	0.09	**0.02**	0.051	**0.049**	0.058	**0.03**	0.05	**0.036**
Iteration	Mean	**479.6**	646.6	**405.8**	429.5	349.3	**315.9**	424.9	**410.1**	**157.9**	403.9
	Min	**36**	358	**100**	157	**21**	61	**24**	59	**15**	96
	Max	**982**	991	**663**	707	682	**650**	986	**667**	**499**	973
	Median	**370.5**	620.5	405.5	**353**	351.5	**342**	365	**358**	**88.5**	340
	Std.	314.5	**2,4**	1,8	**1,7**	206.3	**156.9**	305.7	**218.9**	**161.1**	230.4

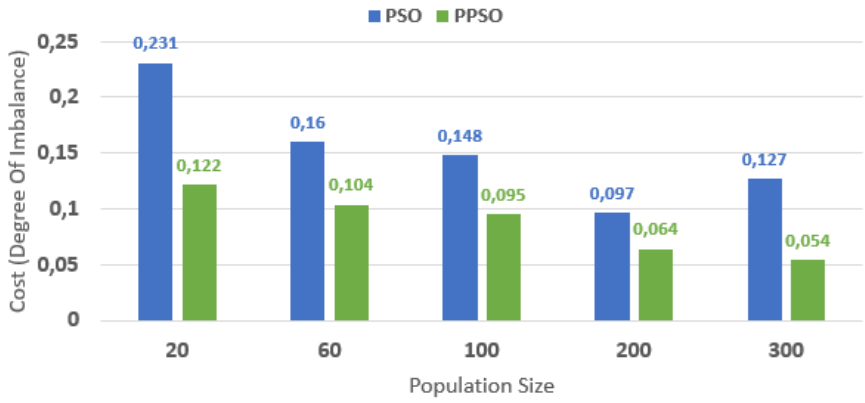

Figure 4.14 Change of cost values of PSO and PPSO respect to population size change

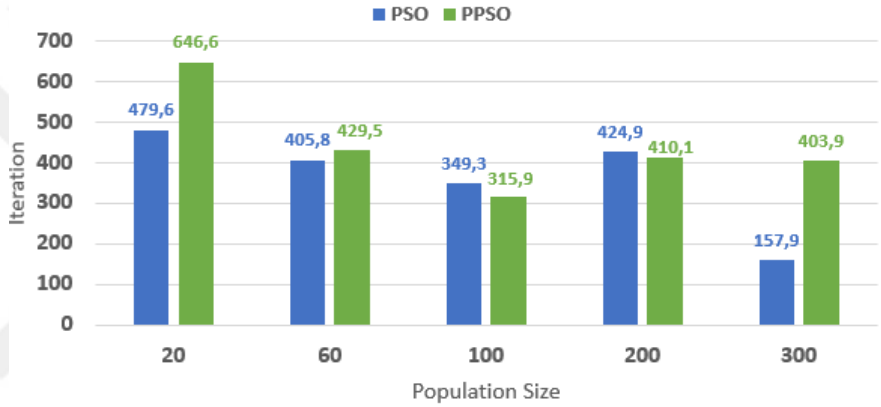

Figure 4.15 Change of iteration values of PSO and PPSO respect to population size change

4.2.3.2 Experiment 2 (Inertia Change)

In this scenario, the number of inertia ranges from 0.9 to 0.05, while other parameters are kept fixed. The values of the parameters, that are kept fixed during the scenario, are as follows: 128T 14M problem size, 70 population size, 1000 iterations, 2 c_1-c_2. Statistical cost and iteration results are given in Table 4.9, and average cost and iteration comparisons are given in Figure 4.16 and Figure 4.17, respectively.

Table 4.9 Statistical results of cost and iteration values of PSO and PPSO respect to inertia change

Inertia		0.9		0.2		0.1		0.05	
		PSO	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**
Cost	Mean	0.251	**0.192**	0.135	**0.091**	0.14	**0.092**	0.14	**0.106**
	Min	0.19	**0.11**	**0**	**0**	0.11	**0**	**0.05**	**0.05**
	Max	0.41	**0.27**	0.21	**0.14**	0.21	**0.14**	**0.21**	**0.21**
	Median	0.225	**0.2**	0.14	**0.11**	0.125	**0.11**	0.125	**0.11**
	Std.	0.065	**0.04**	0.07	**0.04**	**0.036**	0.038	0.06	**0.039**
Iteration	Mean	190.1	**124.2**	**491.5**	521.6	600.4	**390.1**	**280.7**	439.6
	Min	**21**	24	48	**19**	**17**	33	**34**	34
	Max	899	**372**	**956**	964	995	**990**	**660**	928
	Median	**35**	58	**484.5**	496.5	686	**329**	**282**	402
	Std.	306.3	**126.1**	**285.4**	365.9	342.1	**319.7**	**185.4**	259.6

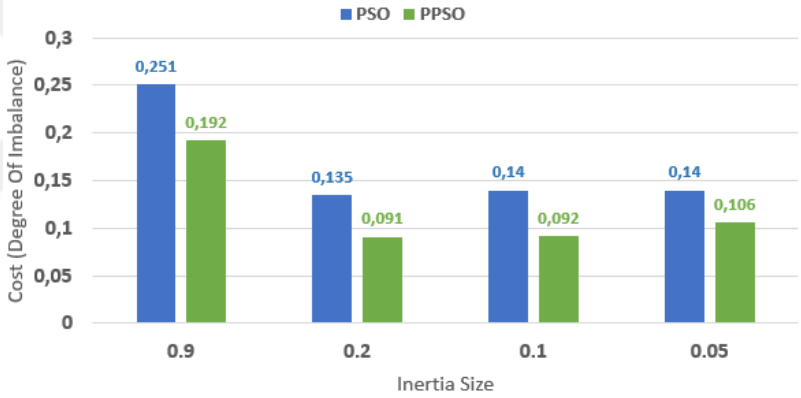

Figure 4.16 Change of cost values of PSO and PPSO respect to inertia change

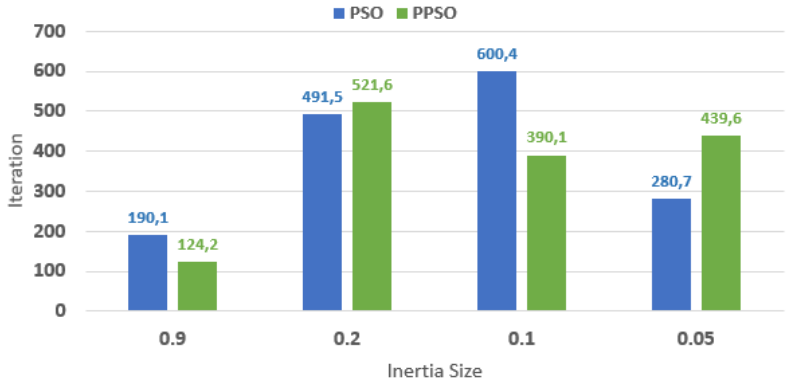

Figure 4.17 Change of iteration values of PSO and PPSO respect to inertia change

4.2.3.3 Experiment 3 (Problem Size Change)

The datasets with different problem sizes in the range of 64-256 tasks and 14-28 VMs are changed, while other parameters were kept fixed during this scenario. The values of the parameters, that are kept fixed during the scenario, are as follows: 200 population size, 1000 iterations, 2 c_1-c_2, 0.1 inertia. Statistical cost and iteration results are given in Table 4.10, and average cost and iteration comparisons are given in Figure 4.18 and Figure 4.19, respectively.

Table 4.10 Statistical results of cost and iteration values of PSO and PPSO respect to problem size change

	Problem	**256T 28M**		**256T 14M**		**128T 28M**		**128T 14M**		**64T 28M**		**64T 14M**	
		PSO	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**	**PSO**	**PPSO**
Cost	Mean	0.464	**0.307**	0.051	**0.039**	0.829	**0.583**	0.097	**0.064**	1.24	**1.1**	0.26	**0.21**
	Min	0.28	**0.17**	0.03	**0**	0.62	**0.44**	0	**0**	**0.94**	0.95	**0.21**	**0.21**
	Max	0.68	**0.44**	0.09	**0.05**	1.05	**0.65**	0.17	**0.11**	1.48	**1.36**	0.4	**0.21**
	Median	0.44	**0.31**	0.05	**0.045**	0.875	**0.6**	0.11	**0.07**	1.365	**1.075**	0.22	**0.21**
	Std.	0.118	**0.08**	0.016	**0.015**	0.13	**0.064**	0.058	**0.03**	0.21	**0.13**	0.072	**0**
Iteration	Mean	**540**	622	527.3	**524.6**	**294.5**	425.9	424.9	**410.1**	**189.8**	256.1	289.2	**259.8**
	Min	**29**	335	118	**68**	**17**	34	**24**	59	**34**	36	**15**	21
	Max	**944**	972	**911**	982	**921**	983	986	**667**	**481**	686	994	**653**
	Median	**530**	637	**523**	537.5	**106**	346	365	**358**	**82.5**	132.5	**53**	321.5
	Std.	246.6	**198.9**	**238.1**	249.7	**321.9**	324.2	305.7	**218.9**	**166.2**	239.6	352.5	**194.6**

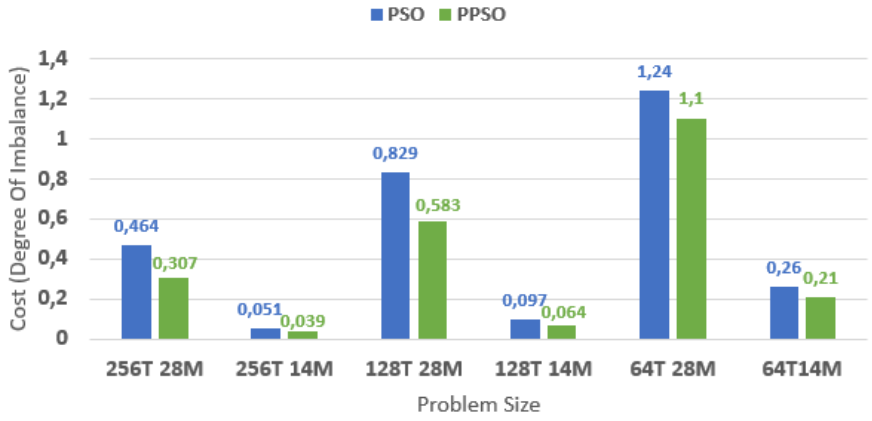

Figure 4.18 Change of cost values of PSO and PPSO respect to problem size change

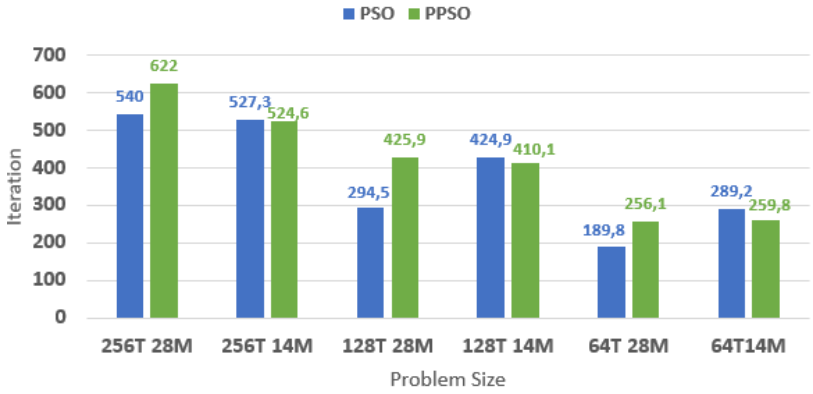

Figure 4.19 Change of iteration values of PSO and PPSO respect to problem size change

4.2.3.4 Evolution of the PSO Experiments

The statistical results of cost and iteration values obtained in the experiments performed in this section are given in Table 4.8, Table 4.9, and Table 4.10, while the average values of cost and iteration values are compared in the visual charts between in the Figures 4.14-4.19, respectively. As can be seen in Figure 4.14, Figure 4.16, and Figure 4.18, PPSO achieved better cost results than PSO in all experiments. In the first experiment, the results obtained with PSO and PPSO seem to get better as the population number increases, but the PSO gets worse again after 200. It seems that the average number of iterations of PPSO decreases when the population numbers are 100 and 200 in the Figure 4.15. Although the average number of iterations with the best cost values of PPSO seems to increase in Figure 4.15 when the population is 20, 60, or 300, it is seen in Figure 4.14 that the cost values found by PPSO for the same population values are better than PSO. Similar results to the experiment 1 are seen in the experiment 2. In Figure 4.17, although the average number of iterations with the best cost values of PPSO seems to be increasing, it is seen that the cost values found by PPSO are better than PSO in Figure 4.16. In addition, when the inertia values are 0.2, 0.1, and 0.05, the cost values obtained by PSO and PPSO do not change much, but when the inertia value is 0.1, PPSO decreases both the average cost value and the iteration value. Although the average number of iterations of PPSO in Figure 4.19 increased in Experiment 3, the results were similar to other experiments in Experiment 3, since the average cost values in Figure 4.18 got better. When comparing the PSO and PPSO algorithms in the tables, the better values in the respective sections are

shown in bold. The results in Table 4.5, Table 4.6 and Table 4.7 show that PPSO is generally better, with a few exceptions, especially in terms of cost values. It also shows that PPSO achieves more consistent results as its cost values generally achieve a lower standard deviation.

4.2.4 GA and PGA Comparisons

In this section, in order to compare the performance of GA and PGA algorithms, various experiments will be performed according to population size change, selection size change, and dataset that different problem size. The best found cost values and the iteration values that found the best cost value will be given and compared statistically.

4.2.4.1 Experiment 1 (Population Change)

In this scenario, the number of populations ranges from 50 to 1000, while other parameters are kept fixed. The values of the parameters, that are kept fixed during the scenario, are as follows: 128T 14M problem size, 5000 iterations, 10 selection size, 0.5 crossover ratio. Statistical cost and iteration results are given in Table 4.11, and average cost and iteration comparisons are given in Figure 4.20 and Figure 4.21, respectively.

Table 4.11 Statistical results of cost and iteration values of GA and PGA respect to population size change

Population		**50**		**100**		**250**		**500**		**1000**	
		GA	**PGA**	**GA**	**PGA**	**GA**	**PGA**	**GA**	**PGA**	**GA**	**PGA**
Cost	Mean	0.114	**0.077**	0.102	**0.065**	0.083	**0.042**	0.072	**0.036**	0.058	**0.015**
	Min	0.09	**0.04**	0	**0**	0.04	**0**	0	**0**	0	**0**
	Max	0.14	**0.11**	0.14	**0.11**	0.11	**0.07**	0.17	**0.07**	0.11	**0.05**
	Median	0.11	**0.07**	0.11	**0.07**	0.09	**0.05**	0.06	**0.04**	0.06	**0**
	Std.	**0.014**	0.028	0.035	**0.03**	0.028	**0.021**	0.05	**0.025**	0.037	**0.023**
Iteration	Mean	**1530.7**	2045.4	**760**	1286	**1772.5**	2075.4	**430.6**	1489.8	**1381.7**	1433.3
	Min	**283**	676	**72**	182	**109**	132	**60**	113	99	**73**
	Max	4449	**4359**	4263	**2965**	**4758**	4780	**1101**	4970	4856	**3657**
	Median	**961**	1555	**361.5**	1115	**1092.5**	1450.5	**380.5**	771.5	**1047**	1346.5
	Std.	**1299.3**	1309.2	1195.9	**875.7**	1823.9	**1450.8**	**330.3**	1463.6	1366.9	**1191.3**

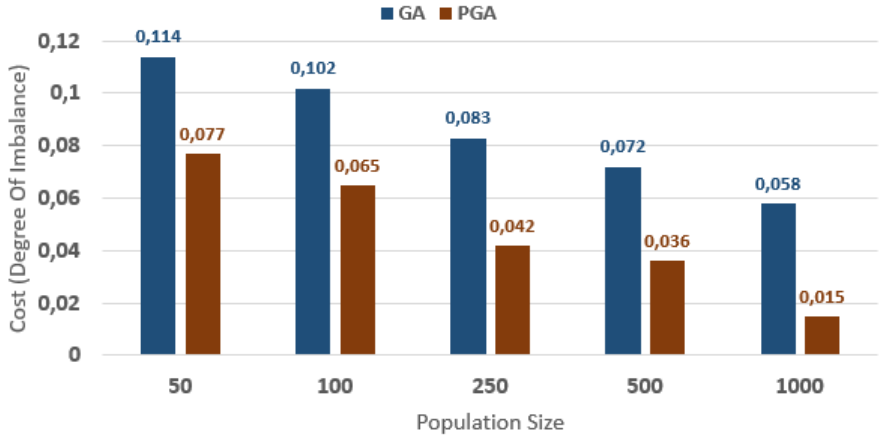

Figure 4.20 Change of cost values of GA and PGA respect to population size change

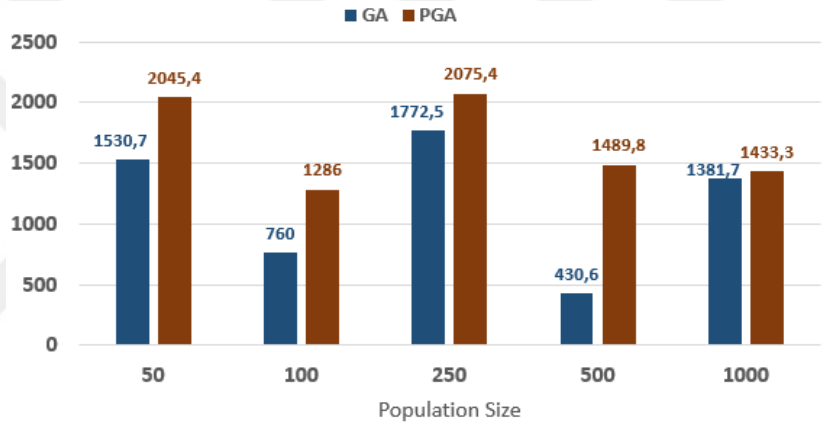

Figure 4.21 Change of iteration values of GA and PGA respect to population size change

4.2.4.2 Experiment 2 (Selection Size Change)

In this scenario, the number of populations ranges from 40 to 200, while other parameters are kept fixed. The values of the parameters, that are kept fixed during the scenario, are as follows: 128T 14M problem size, 250 population size, 5000 iterations, 0.5 crossover ratio. Statistical cost and iteration results are given in Table 4.12, and average cost and iteration comparisons are given in Figure 4.22 and Figure 4.23, respectively.

Table 4.12 Statistical results of cost and iteration values of GA and PGA respect to selection size change

Selection		100		50		30		15	
		GA	**PGA**	**GA**	**PGA**	**GA**	**PGA**	**GA**	**PGA**
Cost	Mean	0.106	**0.03**	0.073	**0.034**	0.068	**0.03**	0.095	**0.057**
	Min	0.04	**0**	0	**0**	0.0	**0**	0.05	**0**
	Max	0.18	**0.05**	0.18	**0.07**	0.17	**0.05**	0.18	**0.09**
	Median	0.11	**0.05**	0.06	**0.04**	0.06	**0.04**	0.1	**0.05**
	Std.	0.037	**0.024**	0.047	**0.024**	0.05	**0.02**	0.036	**0.024**
Iteration	Mean	**2404.2**	3859.4	**1997.8**	3665.5	**1163.2**	2313.2	**1281.4**	1730.1
	Min	**1224**	1536	**340**	1109	**157**	494	187	**137**
	Max	**4508**	4991	**4856**	4923	**3945**	4134	**3303**	4757
	Median	**2051**	4190	**848.5**	3717.5	**631.5**	2405.5	**818.5**	1729.5
	Std.	1018.7	**976.4**	1804.1	**1045.5**	1163.3	**1028.9**	**1138.2**	1452.5

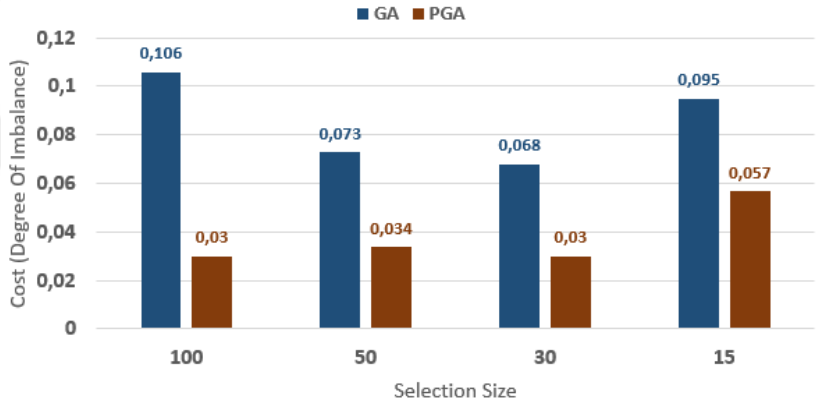

Figure 4.22 Change of cost values of GA and PGA respect to selection size change

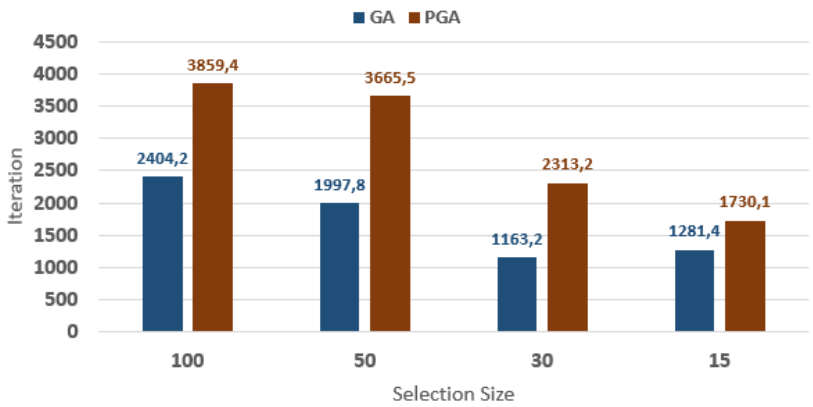

Figure 4.23 Change of iteration values of GA and PGA respect to selection size change

4.2.4.3 Experiment 3 (Problem Size Change)

The datasets with different problem sizes in the range of 64-256 tasks and 14-28 VMs are changed, while other parameters were kept fixed during this scenario. The values of the parameters, that are kept fixed during the scenario, are as follows: 1000 population size, 5000 iterations, 30 selection size, 0.1 crossover ratio. Statistical cost and iteration results are given in Table 4.13, and average cost and iteration comparisons are given in Figure 4.24 and Figure 4.25, respectively.

Table 4.13 Statistical results of cost and iteration values of GA and PGA respect to problem size change

	Problem	**256T 28M**		**256T 14M**		**128T 28M**		**128T 14M**		**64T 28M**		**64T 14M**	
		GA	**PGA**	**GA**	**PGA**	**GA**	**PGA**	**GA**	**PGA**	**GA**	**PGA**	**GA**	**PGA**
Cost	Mean	0.168	**0.126**	0.032	**0.013**	0.34	**0.277**	0.058	**0.005**	0.788	**0.61**	0.206	**0.161**
	Min	0.14	**0.11**	0	**0**	0.27	**0.21**	0	**0**	0.4	**0.27**	0.17	**0**
	Max	0.27	**0.14**	0.05	**0.04**	0.4	**0.4**	0.11	**0.05**	1.32	**0.94**	0.21	**0.21**
	Median	0.14	**0.13**	0.045	**0**	0.34	**0.28**	0.05	**0**	0.76	**0.53**	0.21	**0.19**
	Std.	0.045	**0.014**	0.022	**0.016**	0.061	**0.05**	0.038	**0.015**	0.25	**0.18**	**0.012**	0.065
Iteration	Mean	856.5	**482.9**	**1225.5**	1903.4	**425.5**	773.7	837.9	**533.8**	**442.6**	516.6	**687.9**	867.4
	Min	72	**62**	**33**	164	33	**24**	**14**	65	**20**	38	18	**13**
	Max	2646	**1732**	**3823**	4245	**2247**	2814	3793	**1955**	**1349**	3444	4770	**4696**
	Median	333.5	**172.5**	**811**	1799.5	**111.5**	459	**230**	320.5	**54**	95	106.5	**40.5**
	Std.	937.4	**554.1**	**1182.9**	1451	**696.5**	872.5	1253.5	**553.5**	**530.6**	1013.8	**1416.9**	1571.6

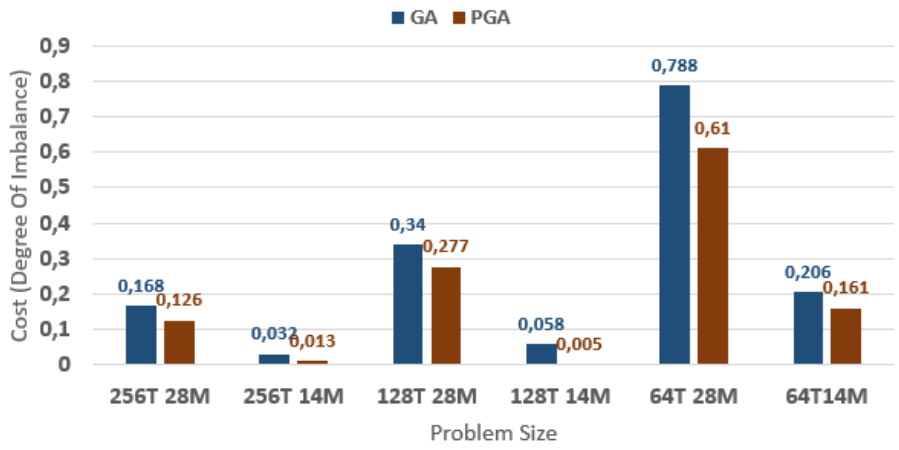

Figure 4.24 Change of cost values of GA and PGA respect to problem size change

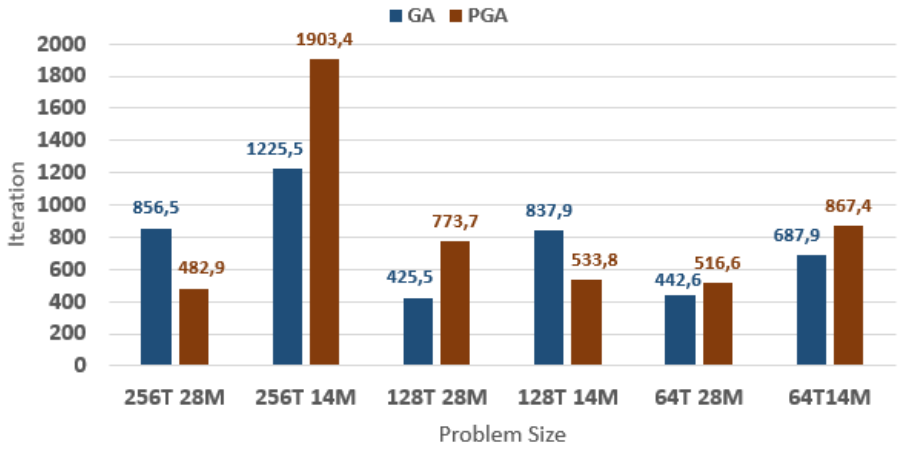

Figure 4.25 Change of iteration values of GA and PGA respect to problem size change

4.2.4.4 Evolution of the GA Experiments

The average cost and iteration values are compared in the visual charts between Figures 4.20-4.25, while the statistical results of the cost and iteration values acquired in the experiments performed in this section are shown in Tables 4.11, Table 4.12, and Table 4.13. As seen in Figure 4.20, Figure 4.22 and Figure 4.24, PGA achieved better cost results than GA in all experiments. In the first experiment, it is seen that the results obtained with GA and PGA get better as the number of population increases. Although the average number of iterations of PGA with the best cost values seems to increase in Figure 4.21, it is seen in Figure 4.20 that the cost values found by PGA for the same population values are better than GA. Similar results are seen in Experiment 2. Although the average number of iterations of PGA with the best cost values appears to be growing in Figure 4.23, the cost values found by PGA are better than GA in Figure 4.22. In addition, when the selection values are 30, GA and PGA give the most optimal values in this experiment. In Experiment 3, it is seen in Figure 4.24 and Figure 4.25 that, in addition to the improvement in cost value for the 256T 28M and 128T 14M dataset results, the average number of iterations with the best cost values also improved. Although the average number of iterations of the PGA increases in the other results in Figure 4.25, the average cost values in Figure 4.24 are getting better, and similar results are also seen in the Experiment 3 to the other experiments. When comparing GA and PGA algorithms in the tables, the better values in the respective sections are shown in bold. The results in Table 4.11, Table 4.12 and Table 4.13 show

that the PGA is generally better with a few exceptions, especially in terms of cost values. It also shows that the PGA achieves more consistent results as its cost values generally achieve a lower standard deviation.

4.2.5 Comparisons of the All Algorithms

CSA, DE, PSO, GA, and parallel versions of all algorithms examined in this study will be compared with the cost values obtained according to the change in dataset size. The comparisons in this section are based on the results of experiment 3 obtained for all algorithms in the previous sections. Therefore, for the statistical results of the cost and iteration obtained according to the change in the problem size of the algorithms in this section, the tables in the experiment 3 section of the relevant algorithm in the previous sections can be consulted. In Figure 4.26, cost comparisons of the sequential versions of the algorithms are given. As can be seen, the CSA algorithm gives the best results in all except the 64T 28M dataset. It is seen that the results of the CSA algorithm and the GA algorithm generally give close results in all cases. Although it is seen that the DE algorithm generally gives results close to the CSA and GA algorithms, it has experienced a noticeable performance loss in 128T 28M and 64T 28M datasets. The PSO algorithm performs poorly, obtaining results farther than CSA and GA.

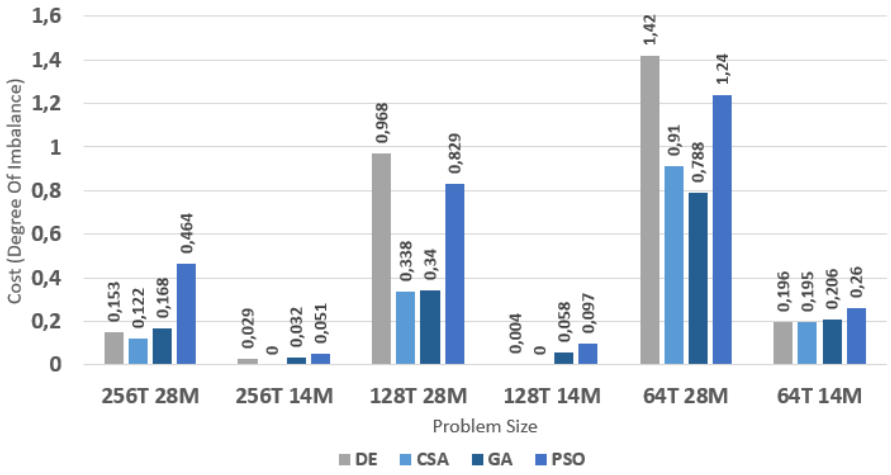

Figure 4.26 Cost comparisons of the sequential algorithms

In Figure 4.27, cost comparisons of parallel versions of the algorithms are given. Similar to the sequential versions, the PCSA algorithm gives the best results in all of

them, and the results of the PCSA algorithm and the PGA algorithm are generally close to each other in all cases. Similarly, although it is seen that the PDE algorithm generally gives results close to the PCSA and PGA algorithms, it has experienced a noticeable performance loss in 128T 28M and 64T 28M datasets. As in the sequential version, the PPSO algorithm performs poorly, obtaining farther results than PCSA and PGA.

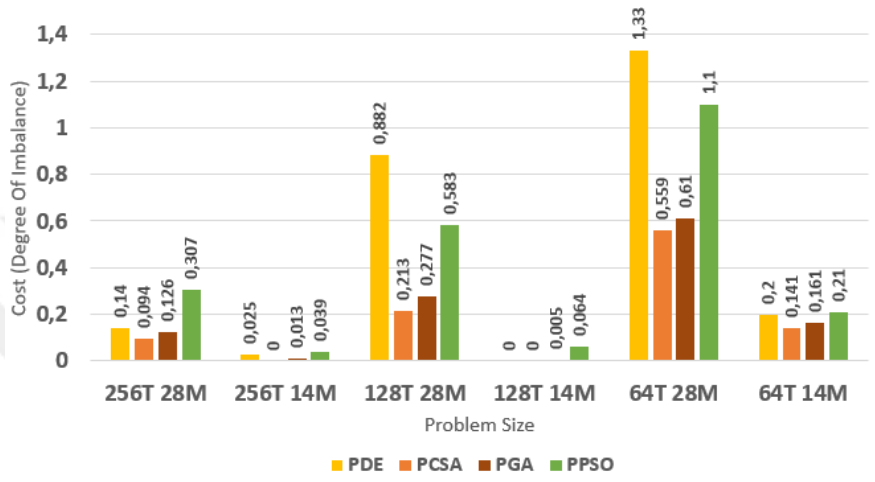

Figure 4.27 Cost comparisons of the parallel algorithms

In Figure 4.28, comparisons of cost values obtained by all algorithms according to the change in dataset size are given. The PCSA algorithm seems to give the best cost performance among all algorithms. Then, although the results are close, PCSA is followed by PGA, CSA, and GA algorithms, respectively. Although it is seen that parallelization improves the solution quality in all results, especially in the tests of 128T 28M and 64T 28M datasets, while the sequential versions of the algorithms are difficult to optimize, it is seen that the parallelized versions of the algorithms, especially the PCSA and PGA, optimize results better than the others.

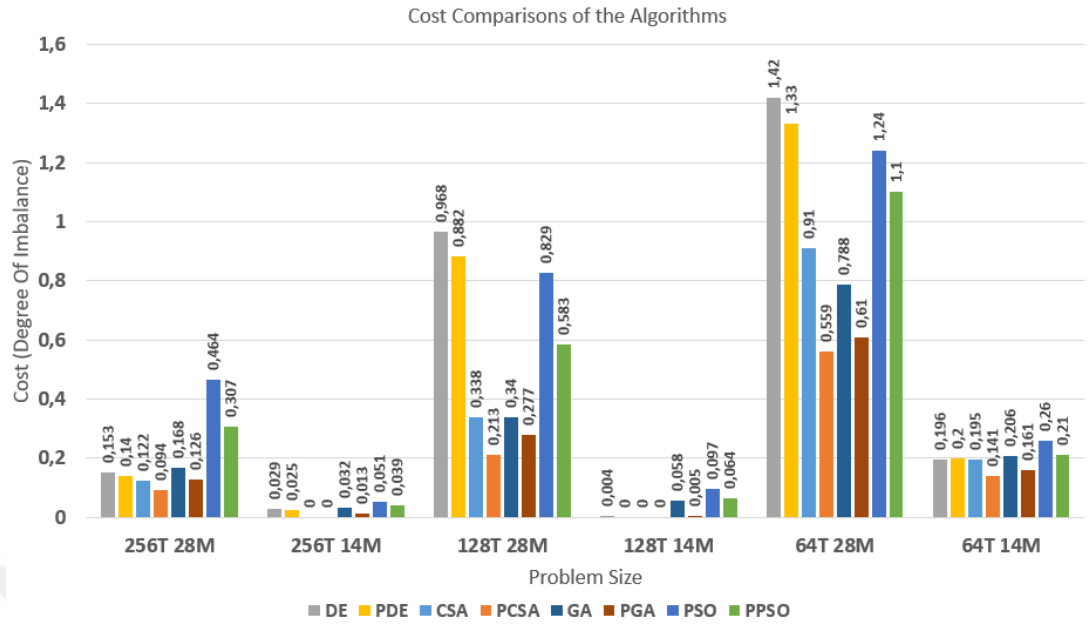

Figure 4.28 Cost comparisons of all algorithms

In order to rank the algorithms according to their performance, the cost values obtained by the algorithms in the tests in Figure 4.28 were summed, and their order from the best to the worst, that is, from the smallest to the largest, is given in Figure 4.29. As can be seen, the PCSA algorithm achieved the lowest cost value in all tests. Although DE and PDE algorithms gave results close to CSA and GA algorithms in most of the tests, the performance loss they experienced in the 128T 28M and 64T 28M datasets caused them to be ranked in the lower ranks. However, there are still cases where DE and PDE algorithms give close results to CSA, GA, PCSA and PGA algorithms, according to the dataset. For example, when the data set is 128T 14M, the PDE algorithm reached the global optimum, while the PPSO algorithm could not. However, the cost value of PDE in the 64T 28M dataset is much higher than PPSO.

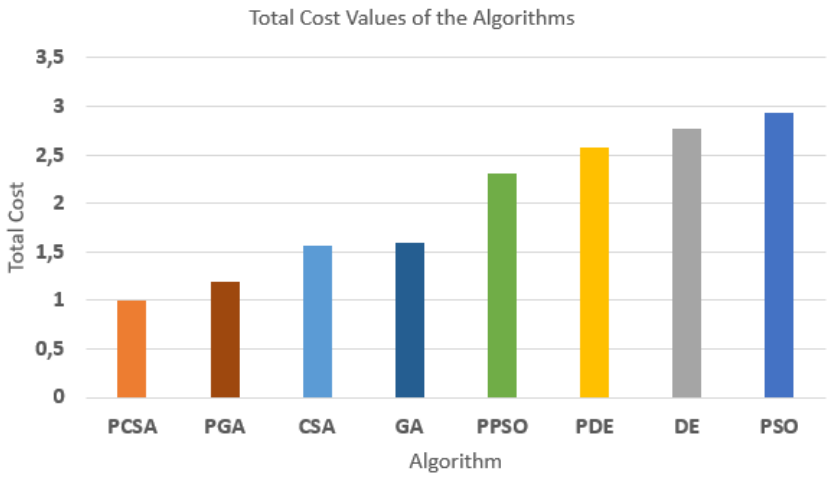

Figure 4.29 Total cost ranks of all algorithms

CHAPTER 5

CONCLUSIONS

In a cloud system with poor load balance, since the tasks are stacked on certain physical resources / VMs, situations such as the increase in makespan and the energy consumption cost of idle resources, are negatively affect both the customer and the provider. In this study, the CSA, DE, PSO and GA algorithms are used to optimize load balance and task scheduling in cloud computing environment, and a performance comparison is presented. In order to increase the search performance of metaheuristic algorithms, unlike the hybrid metaheuristic methods in the literature, a parallelization method is proposed. A hybrid parallelization method is created by taking advantage of the global population master-slave and multiple-deme methods to be independent of the metaheuristic algorithm and to implement flexible. The search performance of the algorithms is increased with this method by increasing the number of search workers in the search space. A performance comparison is presented by applying this proposed parallelization method to the metaheuristic algorithms discussed in this study.

In the tests, the parallel versions of the algorithms were first compared with the normal serial versions of the algorithms, and except for one case in the DE algorithm, the proposed parallelization method increased the search performance of the metaheuristic algorithms discussed in this study. In difficult problems where the search performance of the metaheuristic algorithm needs to be increased, in addition to the hybrid methods used to increase the search performance, the parallelization method proposed in this study is also an option. In addition, distributed parallelization with many slaves can increase performance even more since the number of search workers more increased. When compared to the normal series versions of the algorithms, CSA achieved better results even though it achieved results close to GA. Although the DE algorithm achieved results close to the CSA algorithm in most tests, dramatic performance losses in some datasets negatively affected the ranking. The PSO algorithm was the algorithm with the worst performance in the comparison. When the parallel versions of the algorithms were included in the comparison, the PCSA algorithm further improved the

performance of the CSA and became the algorithm that achieved the best performance. PGA was the algorithm that gave the second best performance, further improving GA. Although the PSO algorithm is the worst algorithm, the PPSO algorithm is better in ranking than PDE and DE. Similar to DE, although PDE outperformed the PPSO algorithm in most tests, the dramatic performance loss of DE in some datasets negatively affected the ranking of both.

In the future, the results obtained in this study can be compared with hybrid methods, and the search performance can be further increased by parallelizing the hybrid methods. In addition, the algorithm performance can be increased further by implementing the proposed model in a distributed architecture and increasing the number of nodes.

REFERENCES

[1] Dragland, A. Big data, for better or worse: 90% of world's data generated over last two years. Science Daily. Accessed 20 April 2023. Available at: https://www.sciencedaily.com/releases/2013/05/130522085217.htm

[2] Toosi, A. N., Calheiros, R. N., & Buyya., R. Interconnected cloud computing environments: Challenges, taxonomy, and survey. ACM Computing Surveys, *47*, (7) 1–47. 2014

[3] Zhan, Z., Liu, X., Gong, Y., Zhang, J., Chung, H. S., & Li, L. Cloud computing resource scheduling and a survey of its evolutionary approaches. ACM Computing Surveys, *47*, (63) 1-33. 2015

[4] Liu, C. -Y., Zou, C. -M., & Wu, P. A Task Scheduling Algorithm Based on Genetic Algorithm and Ant Colony Optimization in Cloud Computing. *2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, DCABES 2014,* 2014

[5] Kumar, P., & Verma, A. Independent Task Scheduling in Cloud Computing by Improved Genetic Algorithm. International Journal of Advanced Research in Computer Science and Software Engineering, *2*, 111-114. 2012

[6] Naithani, P. Genetic Algorithm Based Scheduling to Reduce Energy Consumption in Cloud. *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC), PDGC 2018,* 2018

[7] Zhao, C., Zhang, S., Liu, Q, Xie, J., & Hu, J. Independent Tasks Scheduling Based on Genetic Algorithm in Cloud Computing. *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing, WICOM 2009,* 2009

[8] Jain, A., & Kumari, R. An Efficient Resource Utilization Based Integrated Task Scheduling Algorithm. *2017 4th International Conference on Signal Processing and Integrated Networks (SPIN), SPIN 2017,* 2017

[9] Alsaidy, S. N., Abbood, A. D., & Sahib, M. A. Heuristic initialization of PSO task scheduling algorithm in cloud computing. Journal of King Saud University - Computer and Information Sciences, *34*, (6) 2370–2382. 2022

[10] Yahia, H. S., Zeebaree, S. R. M., Sadeeq, M. A. M., Salim, N. O. M., Kak, S. F., AL-Zebari, A., et al. Comprehensive Survey for Cloud Computing Based

Nature-Inspired Algorithms Optimization Scheduling. Asian Journal of Research in Computer Science, *8*, (2) 1–16. 2021

[11] Shukur, H., Zeebaree, S. R., Ahmed, A. J., Zebari, R. R., Ahmed, O., Tahir B. S. A., et al. A State of Art Survey for Concurrent Computation and Clustering of Parallel Computing for Distributed Systems. Journal of Applied Science and Technology Trends, *1*, 148-154. 2020

[12] Maulud, D. H., Zeebaree, S. R., Jacksi, K., Sadeeq, M. A. M., & Sharif, K. H. A State of Art for Semantic Analysis of Natural Language Processing. Qubahan Academic Journal, *1*, 21-28. 2021

[13] Al-maamari, A. & Omara, F. Task Scheduling Using PSO Algorithm in Cloud Computing Environments. International Journal of Grid and Distributed Computing, *8*, 245-256. 2015

[14] Attiya, I., & Zhang, X. A Simplified Particle Swarm Optimization for Job Scheduling in Cloud Computing. International Journal of Computer Applications, *163*, 34-41. 2017

[15] Jang, S., Kim, T., Kim, J. -K., & Lee, J. The Study of Genetic Algorithm-based Task Scheduling for Cloud Computing. International Journal of Control and Automation, *5*, 157-165. 2012

[16] Bürkük, M. & Yıldırım, G. Cloneable Jellyfish Search Optimizer Based Task Scheduling in Cloud Environments. Türk Doğa ve Fen Dergisi, *11*, (3) 35-43. 2022

[17] Alla, H. B., Alla, S. B., Touhafi, A., & Ezzati, A. Deadline and Energy Aware Task Scheduling in Cloud Computing. *2018 4th International Conference on Cloud Computing Technologies and Applications (Cloudtech), Cloudtech 2018,* 2018

[18] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J. & Brandic, I. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems, *25*, (6) 599-616. 2009

[19] Wang, T., Liu, Z., Chen, Y., Xu, Y., & Dai, X. Load Balancing Task Scheduling Based on Genetic Algorithm in Cloud Computing. *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC), DASC 2014,* 2014

[20] Gao, Y., & Yu, L. Energy-aware Load Balancing in Heterogeneous Cloud Data Centers. *Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences*, *ICMSS 2017*, 2017.

[21] Mishra, S., Sahoo, B., & Parida, P. Load Balancing in Cloud Computing: A big Picture. Journal of King Saud University - Computer and Information Sciences, *32*, (*2*) 149-158. 2018

[22] Farahnakian, F., Pahikkala, T., Liljeberg, P., Plosila, J., Hieu, N., & Tenhunen, H. Energy-aware VM Consolidation in Cloud Data Centers Using Utilization Prediction Model. IEEE Transactions on Cloud Computing, *7*, (2) 524-536. 2019

[23] Boyd, S., Vandenberghe, L. *Convex Optimization*. Cambridge University Press, ISBN: 978-0521833783, 701 pages, 2004

[24] Hoos, H., Stützle, T. *Stochastic Local Search*. Morgan Kaufmann, ISBN: 978-1-55860-872-6, 2005

[25] Bovet D. P., Crescenzi P. *Introduction to the Theory of Complexity*. New Jersey, ISBN: 9780139153808, 330 pages, New Jersey, 1994

[26] Allender E., Loui M., Regan K. Complexity Classes. *Algorithms and theory of computation handbook: general concepts and techniques*. pp. (675-697). Boca Raton: CRC Press. 1999

[27] Rai, Deepak & Seth, K. Bio-inspired optimization techniques: a critical comparative study. ACM SIGSOFT Software Engineering Notes, (*38*), 1-7. 2013.

[28] Özçetin, E. Metaheuristic Algorithm Design and Application for Open Vehicle Routing Problem, Unpublished doctoral dissertation. Eskisehir Technical University. 2019

[29] Simon D. *Evolutionary Optimization Algorithms*. John Wiley & Sons, ISBN: 9780470937419, 784 pages, 2013

[30] Aktas, M. Optimal Pool-Based Test Design Using Swarm-Based Optimization Methods. Unpublished master dissertation, Mersin University. 2020

[31] Tural, B. Time-Plan Optimization with Genetic Algorithm for Regain of Energy from Train Tracks. Unpublished master dissertation, Istanbul Commerce University. 2020

[32] Atagun, E. Development of the Genetic Algorithm Based Conference Session Schedule Program. Unpublished master dissertation, Duzce University. 2020

[33] Khudeer, H. Hybrid Crow-Genetic Algorithm for Solving 3d Bin Packing Problem. Unpublished master dissertation, Kırıkkale University. 2020

[34] Özçetin, E. Metaheuristic Algorithm Design and Application for Open Vehicle Routing Problem. Unpublished doctoral dissertation, Eskişehir Technical University. 2019

[35] Shi, Y., & Eberhart, R. Empirical study of particle swarm optimization. *IEEE Congress on Evolutionary Computation*, *CEC99,* 1999

[36] Kachitvichyanukul, V. Comparison of three evolutionary algorithms: GA, PSO, and DE. Industrial Engineering and Management Systems, *12*, 215-223. 2012

[37] Lopez, G., Morales, L., Nino, L. Immunological Computation. *Autoimmunity From Bench to Bedside*. (1st Edition) pp (373-376). El Rosario University Press. 2013

[38] Shui, X., Zuo, X., Chen, C., & Smith, A. A clonal selection algorithm for urban bus vehicle scheduling. Applied Soft Computing Journal *36*, 36-44. 2015

[39] Ulker, D. E., & Ulker, S. Comparison Study for Clonal Selection Algorithm and Genetic Algorithm. International Journal of Computer Science and Information Technology, *4*, (4) 107–118. 2012

[40] Brownlee, J. Clonal selection algorithms. Complex Intelligent Systems Laboratory, Swinburne University of Technology. 2007

[41] Das, S., Abraham, A., & Konar, A. Particle Swarm Optimization and Differential Evolution Algorithms: Technical Analysis, Applications and Hybridization Perspectives. Studies in Computational Intelligence, (*116*), 1–38. 2008

[42] Bujok, P. On the Performance and Complexity of Crossover in Differential Evolution Algorithm. *19th International Conference Artificial Intelligence and Soft Computing (ICAISC), ICAISC 2020*, 2020

[43] Latorre, A. A Framework for Hybrid Dynamic Evolutionary Algorithms: Multiple Offspring Sampling (MOS). Unpublished doctoral dissertation, Universidad Politécnica de Madrid. 2009

[44] Kennedy, J., & Eberhart, R. Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, *ICNN 1995,* 1995

[45] Stone, L. *Zebras*. Lerner Publications Company, ISBN: 978-0822575115, 48 pages, 2009

[46] Pitcher, T. J. (Ed.). *Behaviour of Teleost Fishes* (363-439). Springer Dordrecht. 1993

[47] Zhang, J., Sheng, J., Lu, J., & Shen, L. UCPSO: A Uniform Initialized Particle Swarm Optimization Algorithm with Cosine Inertia Weight. Computational intelligence and neuroscience. 2021

[48] Shi, Y., & Eberhart, R. C. Empirical study of particle swarm optimization. *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99, CEC 1999,* 1999

[49] Kennedy, J., & Mendes, R. Population structure and particle swarm performance. *Proceedings of the 2002 Congress on Evolutionary Computation, CEC 2002,* 2002

[50] Liu, H., Abraham, A., & Hassanien, A. E. Scheduling Jobs on Computational Grids Using Fuzzy Particle Swarm Optimization Algorithm. Future Generation Computer Systems, (*26*), 1336-1343. 2010

[51] Mitchell, M. Genetic Algorithms: An Overview. Complexity, *1*, (1) 31-39. 1995

[52] Barricelli, N. Esempi numerici di processi di evoluzione (Numerical models of evolutionary processes). Methodos, *6*, 45-68. 1954

[53] Fraser, A. Simulation of genetic systems by automatic digital computers. Australian Journal of Biological Sciences, *10*, (3) 484-491. 1957

[54] Box, G. Evolutionary operation: A method for increasing industrial productivity. Journal of the Royal Statistical Society, Series C (Applied Statistics), *6*, (2) 81-101. 1957

[55] Fogel, L., Owens, A., and Walsh, M. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, ISBN: 978-0471265160, 170 pages, 1966

[56] Immanuel, S., & Chakraborty, U. Genetic Algorithm: An Approach on Optimization. *2019 International Conference on Communication and Electronics Systems, ICCES*, 2019

[57] Wang, Y. L. S., Hong, X., & Li, Y. Multi-objective Task Scheduling Optimization in Cloud Computing based on Genetic Algorithm and Differential Evolution Algorithm. *37th Chinese Control Conference (CCC*), *ChiCC*, 2018

[58] Konfrst, Z. Parallel genetic algorithms: advances, computing trends, applications and perspectives. *18th International Parallel and Distributed Processing Symposium*, *IPDPS*, 2004.

[59] Cantú-Paz, E. A Survey of Parallel Genetic Algorithms. Calculateurs paralleles reseaux et systems repartis, *10*, (2) 141-171. 1998

[60] Golub, M. & Budin, L. An Asynchronous Model of Global Parallel Genetic Algorithms. *Second ICSC Symposium on Engineering of Intelligent Systems EIS2000,* 2000

[61] Nourah, A., & Abdullatif, A. Multiprocessor Scheduling Using Parallel Genetic Algorithm. International Journal of Computer Science, *9*. 2012

[62] Darrell, W. A Genetic Algorithm Tutorial. Statistics and Computing, *4.* 1998

[63] Liang, Y. D. *Introduction to Java Programming and Data Structures Comprehensive Global Version*. Pearson, ISBN: 978-1-292-07001-8, 1342 pages, 2015

[64] Cordeau, J. F., Gendreau, M., Laporte, G., Potvin, J. Y., & Semet, F. A Guide to Vehicle Routing Heuristics. The Journal of the Operational Research Society, *53*, (5) 512-522. 2002.

[65] Ge, Y., & Wei, W., GA-Based Task Scheduler for the Cloud Computing Systems. *2010 International Conference on Web Information Systems and Mining*, *WISM*, 2010.

[66] Jena, R. Energy Efficient Task Scheduling in Cloud Environment. *4th International Conference on Power and Energy Systems Engineering. CPESE'17*, 2017

[67] Kaur, S., & Verma, A. An Efficient Approach to Genetic Algorithm for Task Scheduling in Cloud Computing Environment. International Journal of Information Technology and Computer Science, *4*, (10) 74–79. 2012

[68] Zhu, H., Chen, S., & Wu, J. Paralleling Clonal Selection Algorithm with OpenMP. *International Conference on Intelligent Networks and Intelligent Systems*, *ICINIS'10,* 2010

[69] Purbasari, A. Clonal Selection Algorithm parallelization with MPJExpress. *8th Computer Science and Electronic Engineering, CEEC*, 2016

[70] Durmuş, B. Comparison of The Results of Classical and Greedy Heuristic Algorithms in Integer Programming. Unpublished master dissertation. Muğla Sıtkı Koçman Üniversitesi. 2018.

[71] De Castro, L. N., & Von Zuben, F. J. The Clonal Selection Algorithm with Engineering Applications. *Genetic and Evolutionary Computation Conference GECCO'00,* 2000

[72] De Castro, L. N., & Von Zuben, F. J. Learning and Optimization Using the Clonal Selection Principle. IEEE Transactions on Evolutionary Computation, *6*, (3) 239-251. 2002

CURRICULUM VITAE

PERSONAL INFORMATION

Name Surname :

E-mail :

EDUCATION

High School :

Bachelor :

Master Degree :

WORK EXPERIENCE

TOPICS OF INTEREST