Multi-objective Task Scheduling in Heterogeneous Cloud Computing Environment

(Report)

Department of Computer Science

Group Members

Sidra Khalid	**NIM-BSCS2019-44**
Fatima Sadiqa	**NIM-BSCS2019-48**

2023

Namal Institute,

30-KM, Talagang Road, Mianwali, Pakistan.

www.namal.edu.pk

DECLARATION

The project report titled **"Multi-objective Task Scheduling in Heterogeneous Cloud Computing Environment"** is submitted in partial fulfillment of the degree of Bachelors of Science in Computer Science, to the Department of Computer Science at Namal Institute, Mianwali, Pakistan.

It is declared that this is an original work done by the team members listed below, under the guidance of our supervisor **"Mr. Shahzad Arif"**. No part of this project and its report is plagiarized from anywhere, and any help taken from previous work is cited properly.

No part of the work reported here is submitted in fulfillment of requirement for any other degree/ qualification in any institute of learning.

Team Members	**University ID**
Sidra Khalid	**NIM-BSCS2019-44**
Fatima Sadiqa	**NIM-BSCS2019-48**

Supervisor

Mr. Shahzad Arif

Co-supervisor

Ms. Tooba Tehreem

Table of Contents

9. List of tables……

Acknowledgment

I would like to extend my deepest gratitude to my Project Supervisor, Mr. Shahzad Arif, due to his availability to clarify the design of the project and whenever we got stuck. His support, guidance, and morale boost have allowed us to complete the project and build a working prototype. We would also like to take the opportunity forward in extending my gratitude towards my teachers, all my friends, and class fellows for their constant backing and reassurance which inspired us to start work with new motivation whenever we felt down or needed guidance.

Abstract

For the high-performance execution of applications in a heterogeneous distributed computing system, efficient task scheduling is crucial. The main purpose of this research is to build a dynamic task scheduling algorithm for a heterogeneous cloud computing environment. We suggested a comprehensive multi-objective algorithm based on PPEFT (Parental Prioritization Earliest Finish Time) algorithm for task scheduling to minimize make span and provides efficient resource utilization to be able to address the issue of dynamic task scheduling in heterogeneous cloud computing environments. This algorithm works in two phases. First phase is parental prioritization of tasks and the second phase is about processor's assignment to these tasks. The suggested algorithm is contrasted with HEFT, CPOP and PPEFT algorithms. This approach performs noticeably superior to alternative algorithms in terms of make span and resource utilization while providing dynamic task scheduling and preserving the other parameters within significant bounds. It uses different benchmark scientific procedures, such as Epigenomics and Montage.

Chapter 1

1. Introduction

Modern technology is greatly dependent on cloud computing. It gives boundless capacity space for any kind of data. The data is put away in different information capacity sorts. For reinforcement and reestablish purposes data can be stored within the cloud. Cloud computing empowers organizations to spare cash by lessening the require for equipment overhauls. At long last, cloud computing makes a difference companies diminish IT costs by permitting them to outsource a few of their administrations.

It is providing different reliable and secure services to the users like Infrastructure as a Service (IaaS) like Virtual machine, Servers &Storage, Platform as a Service (PaaS) like Google compute Engine, Apache Stratos and Software as a service (SaaS) like Drop box, Google Apps. All these services are provided by Cloud Service Providers (CSPs). In the form of tasks these services are performed over different set of applications. From here, the concept of task scheduling was introduced when it was required that these services should be provided to the user timely and more securely.

Task scheduling, which involves assigning tasks to available resources, is essential to effective resource use. As a result, task scheduling for heterogeneous computing systems has attracted increasing attention.

1.1. Background Information

In cloud computing, task scheduling is NP-hard to solve. The set of tasks can be broken down into smaller subtasks in task scheduling systems so they can be processed in parallel. Prioritization of supplied jobs at a specified time is caused by task scheduling. Scheduling is performed to measure the order of carrying out different jobs on different virtual machines. It also optimizes the cost, make span, energy consumption and resource utilization etc. CSPs use a variety of high-end devices deployed in various data centers to offer these services to users. Normally a data center has limited resources to handle the requests of customers, but when these requests for different services are on peak in a data center, it becomes difficult to manage them at once. Due to which multiple data centers are combined to resolve this issue of delay in service providing. This generated the concept of multi-cloud environment. Every cloud has its own way of scheduling its tasks. Thus, it is difficult to arrange the tasks in a multi-cloud environment.

Heterogeneous cloud computing involves the use of different types of computing resources, such as CPUs and GPUs, to provide cloud services. The heterogeneous nature of these resources makes task scheduling a challenging task. Dynamic task scheduling involves the distribution of computing to task with resources at runtime, based on various criteria, such as resource availability, task priority, and resource utilization.

1.1.1. Task Scheduling

The process of scheduling tasks involves assigning tasks to different resources to complete them efficiently and by optimizing the resource utilization. The main purpose of task scheduling is optimization of assets. When it comes to cloud computing task scheduling is well known for performance optimization by maximizing the resource utilization and minimizing the makespan. Task scheduling determines the order of execution of different tasks. It ensures that the tasks execution should be completed efficiently and on time. Different types of algorithms are being developed for task scheduling. Task scheduling is a complex problem and requires a lot of planning and analysis for development.

1.1.2. Scheduling levels

In a cloud computing system, scheduling is done at two different levels.

- **Host level**

At the physical host level of the cloud infrastructure, scheduling entails allocating and delivering virtual machines. The scheduler decides which host should host a particular VM, while the cloud provider oversees a pool of actual servers or hosts. This choice is often based on a set of guidelines that take performance optimization, load balancing, and resource availability into account. The scheduler strives to equally distribute the workload among the hosts while making effective use of the physical resources that are readily available.

The host level scheduling regulations could use the following methods:

- **Load balancing:** To balance resource usage and avoid any host from being overloaded or underutilized, the scheduler distributes VMs among hosts.
- **Resource Provisioning:** The scheduler assigns virtual machines to hosts with the necessary resources, such as CPU, memory, and storage, to satisfy their needs.

- **Power management:** The scheduler may reduce the number of hosts used by VMs and, when possible, put unused hosts into low-power or inactive states in order to reduce power consumption.
- **Fault Tolerance:** To achieve high availability, the scheduler may take into consideration redundancy and spread VMs across many hosts.

➢ **Virtual machine (VM) level**

The scheduling emphasis switches to the VM level after a VM has been assigned to a host. Managing the distribution of computational resources into a particular VM instance is the responsibility of VM level scheduling. The virtualization layer or hypervisor operating on the host is responsible for this scheduling.

The scheduling at the VM level may include:

- **CPU and Memory Management:** The hypervisor's scheduler divides up CPU and memory resources among the virtual machines in accordance with their needs and priorities.
- **I/O and Network Scheduling:** To ensure equitable resource sharing among various VMs running on the same host, the hypervisor controls I/O and network connections for the VM.
- **Inter-VM Communication:** To ensure effective data flow and reduce latency, the scheduler manages communication between several VMs.

The goal of VM level scheduling is to maximize each individual VM's performance and ensure effective use of the resources allotted to each VM instance.

Cloud computing systems are capable of effectively handling and allocate resources, balance workloads, optimize efficiency, and provide flexibility and scalability to meet the different demands of various applications and users within the cloud environment by carrying out scheduling at both the host and VM levels.

1.1.3. Tasks Scheduling Algorithms classifications

The following categories apply to task scheduling algorithms:

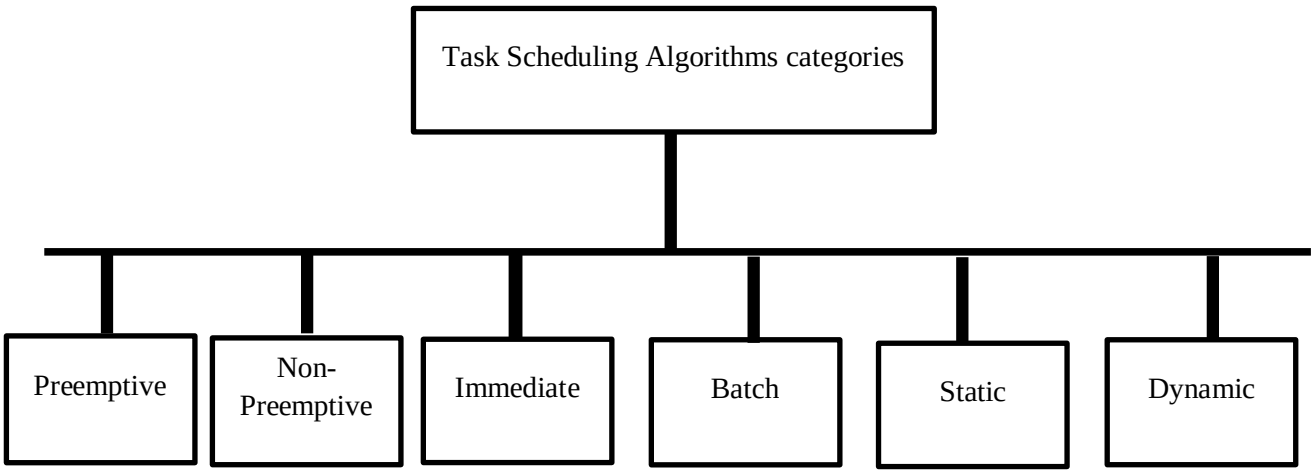


Fig. 1 Task Scheduling classes

- **Immediate Scheduling**

The term "immediate scheduling" refers to a scheduling strategy in which tasks or jobs are carried out without delay promptly as they become available. It seeks to reduce waiting time and guarantee quick task completion in a computing system. Immediate scheduling in the case of cloud computing entails allocating resources and starting task execution as soon as they are sent to the cloud environment. This method is very helpful in situations where low latency and rapid reaction times are essential, such as applications that operate in real time, online services, or workloads that must be completed in a timely manner.

Immediate scheduling has advantages like shorter wait times, better responsiveness, and effective resource management. To ensure equity and top performance across jobs, proper resource management and workload balancing are necessary. It's important to note that not all workloads or applications may benefit from quick scheduling. To meet certain needs or optimize resource consumption in various situations, some tasks could need complex scheduling algorithms, prioritization, or queuing methods. The preferred scheduling strategy is determined by the particular requirements, features, and workload in the computer environment.

- **Batch scheduling**

The process of managing and optimizing the performance of numerous jobs or activities in a batch processing system is known as batch scheduling. In batch processing, jobs are gathered together and carried out automatically, frequently according to a predetermined order or set of criteria.

Operating systems, systems for handling jobs, systems for processing data, and systems for industrial automation are just a few of the areas where batch scheduling is frequently employed. It facilitates the effective use of computing resources and the automation of repetitive operations.

The particular needs of the application, the characteristics of the workload, and the available resources all influence the decision about the batch scheduling system and algorithms. The needs for batch processing vary across different businesses and disciplines, and the scheduling strategy might change correspondingly.

- **Preemptive Scheduling**

Operating systems and task management systems use preemptive scheduling, a scheduling approach that allows a higher-priority activity to halt the execution of a lower-priority task that is currently in progress. Tasks are given priorities during preemptive scheduling, and the scheduler chooses which task to carry out depending on those priorities. Round Robin with Time Quantum, Priority Scheduling with Ageing, and Multilevel Queue Scheduling are a few examples of preemptive scheduling algorithms.

Preemptive scheduling, in general, offers more control and flexibility for handling tasks, particularly in situations where tasks must be completed within specific deadlines and with fluctuating priorities. It enables the effective use of system resources and guarantees that crucial tasks get the attention they require

- **Non-preemptive Scheduling**

When a task or process begins to run under non-preemptive scheduling, sometimes referred to as cooperative scheduling, it keeps running until it is finished or voluntarily yields the CPU. In non-preemptive scheduling, the job that is currently running is in charge of the CPU until it completes or moves into a waiting state, such as one that is awaiting I/O or an event.

First-Come, First-Served (FCFS), Shortest Job Next (SJN), and Priority Scheduling without preemption are a few examples of non-preemptive scheduling algorithms. As jobs are cooperative

and can be made to give the CPU as necessary, non-preemptive scheduling can be an effective option. Although scheduling techniques are made simpler, fairness and responsiveness must be ensured by proper programming and resource management.

- **Static Scheduling**

Static scheduling is a scheduling method where the allocation of jobs to resources and the sequence in which they are completed are decided upon in advance, usually before the activities themselves are completed. It entails developing a set schedule or strategy for carrying out tasks in accordance with limits or predetermined criteria. In static scheduling, despite the dynamic conditions of the system, the schedule does not alter throughout execution.

Even while static scheduling gives predictability and optimal resource use, its efficacy in dynamic contexts is constrained by its lack of adaptation. The schedule may need to be reassessed and modified in response to changes in task priority, resource availability, or unforeseen events. Preemptive or online scheduling are two dynamic scheduling techniques that can adjust and reassign jobs in real-time that may be more appropriate in these circumstances.

- **Dynamic Scheduling**

The term "dynamic scheduling" refers to a scheduling method in which decisions about task assignments, execution sequences, and resource allocation are made in real time. In order to respond to shifting system dynamics, task behavior, priorities, or resource availability, the schedule must be dynamically adjusted. Compared to static scheduling, dynamic scheduling offers greater flexibility and reactivity.

Here are some crucial aspects of dynamic scheduling to take into account:

1) **Real-time Adaptability:** Dynamic scheduling changes task allocations and order of execution in response to current events. To make wise scheduling decisions, it considers things like job dependencies, resource availability, deadlines, priorities, and system load.
2) **Prioritization of Tasks:** When determining the order of execution, dynamic scheduling takes task priorities into account. Critical operations can be completed quickly because high-priority tasks are prioritized over low-priority chores.
3) **Resource Optimization:** By effectively utilizing the resources that are already available, dynamic scheduling seeks to maximize resource utilization. It takes into account the state of the

available resources and dynamically distributes them to jobs according to their needs and availability.

4) **Load balancing:** Dynamic scheduling aids in distributing workload among resources in an even manner to avoid processing bottlenecks. It keeps an eye on how the system is using its resources and redistributes jobs as necessary to improve overall performance.
5) **Adaptation to Changing variables:** Dynamic scheduling can change the schedule in response to alterations in workload, failures, or resource availability, among other varying variables. It enables adaptation to unforeseen circumstances or resource limitations without interfering with overall execution.
6) **Online Scheduling:** Dynamic scheduling frequently takes place online, making scheduling decisions continually as jobs become available or as system conditions alter. It can manage the addition of new tasks and modifications to task specifications without the need for a set schedule.
7) **Algorithms for Scheduling:** Different algorithms, such as shortest job next, round-robin, priority-based, or sophisticated methods like backfilling or genetic algorithms, can be used for dynamic scheduling. The scheduling problem's specific criteria and goals influence the algorithm of choice.
8) **Trade-offs:** Responsiveness, resource use, and overhead are all subject to trade-offs in dynamic scheduling. Though they increase computing cost, frequent plans and adaptability result in a more responsive system and better resource management.

In contexts with shifting priorities, unanticipated occurrences, varied workloads, or constrained resources, dynamic scheduling is very helpful. It is appropriate for systems that need reactivity, scalability, and successful resource management because it enables efficient resource allocation, the execution of real-time tasks, and flexibility to dynamic settings.

1.1.4. Task Scheduling Systems in Cloud Computing

There are three various levels in cloud computing through which a task scheduling algorithm passes. These levels are defined as follows:

- **Task level**

Task-level scheduling, concentrates on how certain activities or processes are carried out on a machine or virtual machine. Based on a task's requirements and dependencies, task schedulers distribute resources within a computer or virtual machine to each particular task.

They oversee task prioritization, task execution order, and, if necessary, task migration or preemption.

- **Scheduling level**

In this second level the tasks are mapped to the suitable resources in order to maximize resource utilization and minimize the make span.

- **VM level**

Task scheduling systems in cloud computing must have VM level scheduling, commonly referred to as virtual machine level scheduling. To efficiently complete activities, it entails allocating and managing virtual machines (VMs).

VM level schedulers are in charge of assigning VMs to tasks depending on their availability and resource needs. They take into account variables like CPU, memory, storage, and network bandwidth to make sure that tasks have the resources they need to be executed.

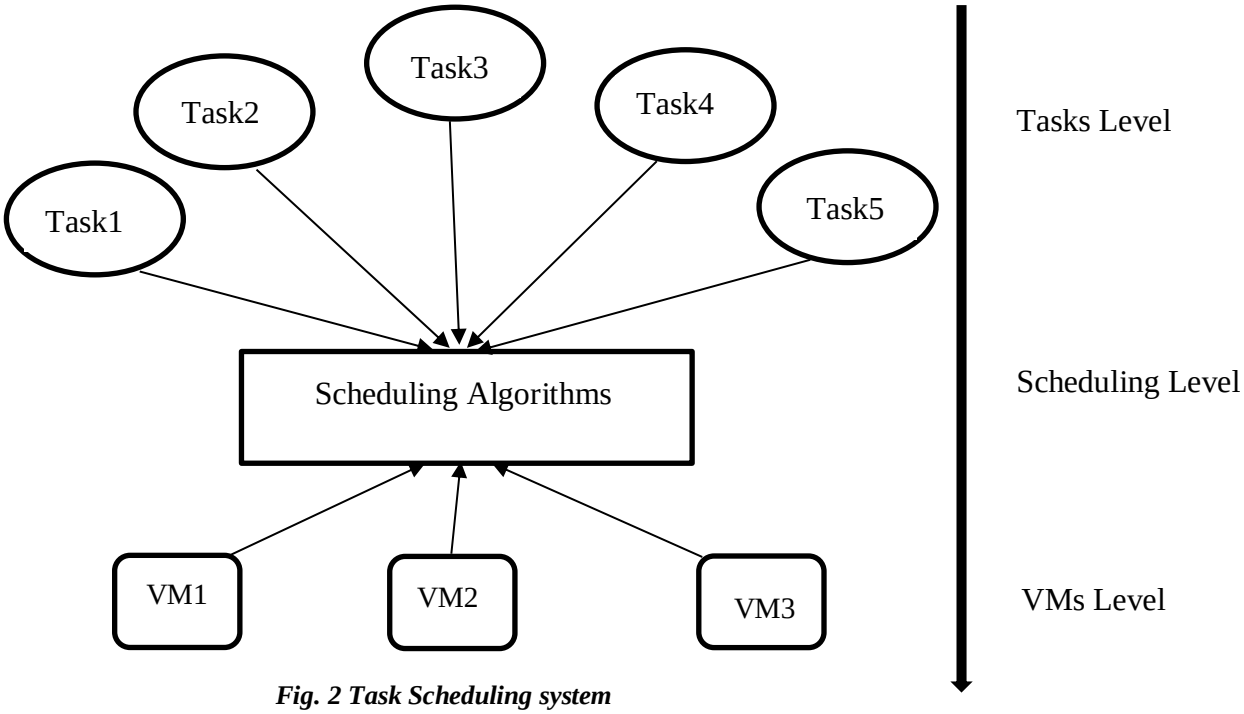


Fig. 2 Task Scheduling system

Task scheduling is crucial to achieving strong dependability, computing power, availability and scalability in a lot of studies domains, especially as the use of heterogeneous computing systems increases. Here below is brief description of homogeneous and heterogeneous distributed system.

1.1.4.1: Homogeneous Distributed System

A homogeneous distributed system, as used in cloud computing, is a network of interlinked nodes or servers that operate as a single, cohesive system and have standardized or comparable software and hardware configurations. The terms homogeneity and distribution are combined in a homogeneous distributed system, where the nodes share similar traits and work together as a unit.

The properties of a homogenous distributed system in cloud computing are as follows:

- **Hardware Consistency**

The nodes or servers which make up a homogenous distributed system have constant or comparable hardware configurations. Included in this are elements like CPU architecture, RAM size, storage capabilities, network interfaces, along with additional hardware characteristics. The distributed system's management, resource distribution, and interoperability are made simpler by the hardware's consistency.

- **Software Consistency**

In a homogenous distributed system, the software stack, includes the Operating System (OS) and system software, is constant across all nodes. This makes sure that all nodes have the same software needs, configurations, and capabilities. The deployment, upgrades, and maintenance of applications and services that are distributed are made simpler by the consistency of software.

- **Interconnectivity**

Homogeneous distributed systems are connected via a network architecture, allowing for node-to-node communication and coordination. The network connectivity enables information sharing, task coordination, and resource sharing among system nodes.

- **Load Balancing and Resource Management**

Load balancing and resource management strategies are used by homogeneous distributed systems to effectively distribute workloads and distribute resources among the nodes. To efficiently use the resources at hand, load balancing algorithms make sure that jobs or requests are allocated equitably. To ensure system performance and scalability, resource management systems track and optimize resource utilization.

- **Fault Tolerance and High Availability**

In order to provide system resiliency and high availability, homogeneous distributed systems frequently use fault tolerance methods. These technologies, which handle failures and sustain ongoing operation even if each of the nodes encounter problems, include redundant operation, replication, and identification of faults and recovery strategies.

- **Performance and Scalability**

By adding additional nodes to the system, homogeneous distributed systems can scale horizontally. The system can handle growing workloads and rising user demands thanks to the homogeneity in both software and hardware configurations. To improve system performance as a whole, performance optimizations can be done consistently across all nodes.

Homogeneous distributed systems in cloud computing offer a unified and coordinated environment for running distributed applications, services, and operations by utilizing the homogeneity in hardware, software, as well as management across remote nodes. The uniform setup and behavior of nodes make management easier, boost resource efficiency, and promote effective collaboration and interaction among the distributed system.

1.1.4.2. Heterogeneous Distributed System

When referring to a distributed computing environment in the cloud, a heterogeneous distributed system is one in which numerous interconnected nodes or servers have different or varied hardware and software configurations. A heterogeneous distributed system contains nodes with various qualities and capacities, as opposed to a homogeneous distributed system in which the nodes are comparable.

The qualities that distinguish a heterogeneous distributed system in cloud computing are as follows:

- **Hardware Diversity**

Hardware diversity refers to the many hardware configurations that the nodes or servers in a heterogeneous distributed system have. Variations in CPU designs, memory sizes, storage capabilities, interfaces for networks, and other hardware specs are included in this. The diversity in hardware can result from nodes coming from several manufacturers or employing various hardware versions.

- **Software Variability**

The operating systems, software system variants, libraries, and dependencies of the software stack on each node of a heterogeneous distributed framework may differ. Different software components, setups, or specialized software may be present on nodes depending on the task at hand.

- **Interconnectivity**

Heterogeneous systems with distributed components still depend on network infrastructure to connect their nodes together. Regardless of the hardware and software variations between the nodes, they communicate and share data through the network. Coordination, sharing of data, and distributed processing amongst heterogeneous nodes are made possible via network connectivity.

- **Compatibility Challenges**

Heterogeneous distributed systems have difficulty in achieving interoperability and compatibility between nodes that have different software and hardware configurations. When developing and running applications for such systems, system designers and developers must take changes in protocols, formats for data, and software dependencies into consideration.

- **Load Balancing and Resource Management**

Load balancing and managing resource utilization are two strategies used by heterogeneous distributed systems to efficiently balance workload and allocate resources amongst nodes with various capabilities. In order to optimize job distribution and resource utilization throughout the system, load balancing algorithms take the heterogeneity of resources and skills into account.

- **Fault Tolerance and Resilience**

Heterogeneous distributed systems employ fault tolerance techniques to deal with breakdowns and guarantee system resilience. These processes, which take into account the diversity and variability of the system's nodes, may include recurrence identification of faults, and recovery techniques.

- **Flexibility and Specialized Workloads**

Heterogeneous distributed systems provide flexibility and the capacity to manage specialized workloads that call for particular software or hardware configurations. The system can accommodate various application requirements since separate nodes can be optimized for various sorts of processing.

In comparison to managing a homogeneous system, maintaining a heterogeneous distributed system entails more difficulties. Resource discovery, software reliability, workload division, and load balancing among nodes with different capabilities must all be taken into account. Despite these difficulties, heterogeneous distributed systems are flexible, can make use of specialized resources, and have the potential to achieve improved performance and efficiency by maximizing the use of various hardware and software combinations.

As we know in heterogeneous cloud computing environment requirements and characteristics of all data centers and their resources are not same due to which task scheduling becomes more challenging in such environment. To resolve this issue different kind of algorithms are being developed.

1.2. Problem Statement

Multi-objective task scheduling is main issue in heterogeneous cloud computing environment since it has a significant impact on cloud performance. Its performance also directs impact on customer's satisfaction. So, this problem should be resolved more accurately to improve its performance.

1.3. Motivation

Cloud computing is at the heart of modern technology solutions. It is providing cost effective, secure and flexible computing and software access from anywhere in the world at any time. Most of the organizations are adopting this digital transformation to become more productive and competitive. Cloud computing has been widely adopted across a variety of businesses and industries thanks to a number of strong justifications. The following are some major drivers of cloud computing:

- **Cost effectiveness**

Businesses can switch from a capital expense (CapEx) paradigm to an operational expense (OpEx) model using cloud computing. Companies can use services in the cloud and just pay for the resources they use, saving money on hardware and infrastructure maintenance and investment costs. With the "pay-as-you-go" concept, there are fewer up-front expenses and the ability to scale charges based on actual consumption.

- **Scalability and Flexibility**

Cloud service providers provide almost infinite resources that can be quickly provided or de provisioned in response to demand. This elasticity makes it simple for businesses to scale up or down their apps and services to meet changing workloads, seasonal needs, or unanticipated traffic surges.

- **Accessibility and Availability**

With a working internet connection, cloud computing makes data and apps accessible from almost anywhere. Teams may collaborate and operate remotely more effectively thanks to this accessibility, which enables smooth cross-border cooperation.

- **High performance and dependability**

Top cloud service providers deploy data centers with redundant equipment to guarantee high availability and dependability. They provide Service Level Agreements (SLAs), which can be difficult for smaller organizations with less resources to meet, but which ensure uptime and performance levels.

- **Security and compliance**

Reliable cloud service providers make significant investments in security measures to safeguard infrastructure and data. They frequently adhere to standards and regulations that are industry-specific, giving organizations the confidence that their information is safely maintained and preserved.

- **Time-to-market and innovation**

Cloud computing provides quick application and service deployment. Without having to handle underlying infrastructure, development teams can concentrate on generating novel goods and features, speeding up the time it takes to commercialize new products.

- **Resource optimization**

When compared to conventional on-premises data centers, cloud providers manage resources more effectively and at a larger scale, which can improve resource utilization and have a less negative impact on the environment.

- **Disaster Recovery and Backup**

Backup and disaster recovery are made easier by the built-in redundancy and recovery features that cloud service providers provide. Organizations won't need to spend money on or manage complicated backup systems thanks to this capability.

- **Global Reach**

Cloud service providers have data centers located throughout the world, enabling businesses to install apps nearer to their end users, reducing latency and enhancing user experience.

- **Democratization of Computing Power**

Cloud computing makes it easier for people, small businesses, and startups to use powerful computing resources that were previously only available to established businesses with sizable budgets.

Overall, cloud computing has revolutionized the IT landscape, providing a cost-effective, scalable, and flexible platform for businesses to innovate, collaborate, and deliver services in a globalized and connected world.

1.4. Scope

Important aspects of heterogeneous task scheduling are to maximize utilization of virtual machines while reducing the operational cost of virtual machines. As a result, which improves the quality-of-service parameters and overall performance. Multi-objective heterogeneous task scheduling in cloud computing is a crucial field for exploration that aims to optimize the allocation of computational resources to different tasks in a cloud environment. It involves the efficient allocation of resources to tasks with different objectives, such as minimizing the execution time, minimizing energy consumption, maximizing resource utilization, and so on.

The scope of this area is vast and encompasses various aspects of cloud computing, such as resource management, task scheduling, load balancing, and energy efficiency. The key objectives of multi-objective heterogeneous task scheduling include minimizing the make span (i.e., the time taken to complete all tasks), maximizing resource utilization, minimizing energy consumption, and reducing the overall cost of resource usage.

The importance of multi-objective heterogeneous task scheduling in cloud computing is increasing due to the growth of cloud-based applications and services. The ability to efficiently schedule tasks across multiple cloud resources can lead to significant improvements in application performance, resource utilization, and energy efficiency, while also reducing the overall cost of resource usage.

Chapter 2

2. Literature Review

Dynamic task scheduling is a crucial concern in heterogeneous cloud computing. The interest in cloud services is growing, the need for efficient and effective task scheduling algorithms is becoming more critical. This literature review aims to investigate the recent advances in dynamic task scheduling in heterogeneous cloud computing.

Several research studies have investigated dynamic task scheduling in heterogeneous cloud computing. These studies have proposed different task scheduling algorithms based on various optimization methodologies, such as genetic algorithms, ant colony optimization, and particle swarm optimization. The following are some of the recent studies on dynamic task scheduling in heterogeneous cloud computing:

2.1. Existing Task Scheduling Algorithms

As we know that the performance of a systems depends on the scheduling of its resources and tasks. Various task scheduling algorithms are being developed to enhance the system performance and reliability. We are using some existing algorithms in comparative analysis with our proposed algorithm. Some of them are as follows:

2.1.1. Min-min Algorithm

A common job scheduling strategy in cloud computing settings is the Min-Min algorithm, which is based on heuristics. By giving priority to the completion of shorter tasks, it seeks to reduce the makespan, which refers to the total amount of time needed to accomplish all jobs. Up until all jobs are scheduled, the Min-Min algorithm continues to pick and allocate tasks iteratively. It attempts to optimally manage resources and shorten the overall makespan by giving shorter activities priority.

In order to increase its efficiency, the Min-Min algorithm is frequently used in conjunction with other scheduling methods or as a smaller scheduling algorithm within a wider scheduling framework. By taking into account a larger variety of variables, hybrid techniques that combine Min-Min with extra heuristics or metaheuristic algorithms might produce superior scheduling outcomes. In general, the Min-Min algorithm provides a straightforward and practical method for task scheduling in cloud computing, especially in situations where the makespan minimization is the primary goal and job parameters are known in advance.

2.1.2. Max-min Algorithm

The Max-min algorithm is another simple task scheduling algorithm that aims to maximize the minimum time required to complete all jobs. The algorithm works by choosing the task that takes the longest to complete and assigning it to the resource that has the least amount of load. The Max-Min algorithm assigns longer tasks to resources that have a substantially lower workload in order to balance the workload among the available resources. This minimizes the overall makespan while maximizing the minimal completion time and preventing resource overload.

For more optimal outcomes, the Max-Min algorithm can be used in conjunction with other scheduling methods or as a component of a larger scheduling framework. By taking into account a wider variety of variables, hybrid techniques that combine Max-Min with extra heuristics or metaheuristic algorithms can offer superior scheduling solutions.

The Max-Min algorithm, which prioritizes workload balancing and reduces the risk of resource saturation, provides a simple method for job scheduling in cloud computing.

2.1.3. Genetic Algorithm (GA)

In 2017, S. Wang et al. published "A Genetic Algorithm for Scheduling Task in Heterogeneous Cloud Computing Environment" in IEEE Access. A class of optimization algorithm called a genetic algorithm draws its inspiration from the ideas of natural selection. In the context of task scheduling, genetic algorithms aim to find the optimal allocation of tasks to resources by evolving a population of candidate solutions over time.

A genetic algorithm's fundamental goal is to simulate evolution via a series of generations by using genetic operators like mutation, crossover, and selection to generate new potential solutions. These potential solutions, also known as individuals or chromosomes, are rated according to their quality or applicability to the issue at hand using a fitness function.

In complicated and multi-modal optimization issues, genetic algorithms are renowned for their capacity to explore broad solution spaces and identify close to ideal solutions. They are employed throughout several industries, including machine learning, engineering, finance, and scheduling. It's crucial to remember that a genetic algorithm's effectiveness and efficiency strongly depend on elements like representation, fitness assessment, selection processes, and genetic operators, all of which must be customized for the particular situation at hand.

2.1.4. Ant Colony Optimization Algorithm (ACO)

An article by N. Zhao and colleagues titled "Task Scheduling Algorithm for Heterogeneous Cloud Computing Based on Improved Ant Colony Optimization" was released in 2018. Ant colony optimization is a metaheuristic optimization algorithm motivated by the behavior of ants. In the context of task scheduling, ant colony optimization aims to find the optimal allocation of tasks to resources by mimicking the behavior of ants searching for food. The idea of pheromone trails, which ants use to communicate with one another, is the foundation of the ACO algorithm. Artificial ants used in ACO search through a problem space by creating and updating pheromone trails in search of ideal or nearly ideal solutions.

Performance of the ACO algorithm is affected by a number of variables, such as parameter selection, pheromone update guidelines, and problem-specific heuristics. For a given situation, it is frequently required to fine-tune these parameters in order to get good outcomes. It is an effective algorithm for resolving combinatorial optimization issues. It has been successfully used in a number of fields, including vehicle routing issues, work scheduling, and routing challenges.

2.1.5. Particle Swarm Optimization Algorithm (PSO)

In 2015, Y. Zhang and colleagues published "A Particle Swarm Optimization Algorithm for Task Scheduling in Heterogeneous Cloud Computing" in the Journal of Supercomputing. A population-based metaheuristic optimization technique called Particle Swarm Optimization (PSO) is inspired by the social behavior of fish schools and bird flocks. Task scheduling, parameter optimization, and function optimization are a few of the common optimization challenges that it is used to solve. A population of particles traverses the issue space in PSO in pursuit of the best answers. Each particle represents a possible solution to the issue, and both its individual and the collective population's best-known positions have an impact on how it moves.

The strength of PSO comes in its capacity to investigate the solution space using both individual knowledge (best-known location for each particle) and social knowledge (best-known position for the entire system). The particles cooperate and modify their motions in accordance with the best solutions discovered thus far, eventually convergent to better solutions.

In general Particle Swarm Optimization is a flexible and effective approach for resolving optimization issues. Numerous domains, including task scheduling, feature selection, neural network training, and many other optimization problems, have effectively used it.

2.1.6. First Come First Service (FCFS) Algorithm

This algorithm works on FIFO (First In First Out) method. In this algorithm, tasks look for the queue where the waiting time is smallest All tasks are placed in a queue. First task from the queue is assigned to the available virtual machine. If the large task is on the front of the queue, then all other smaller tasks will have to wait for the end of execution of that larger task. The main disadvantage of this algorithm is large waiting time. Apart from arrival order, there are no other considerations in the FCFS algorithm. It is easy to install and has no administrative burden with respect to scheduling choices. However, it might not necessarily lead to a good schedule or effective use of resources.

In general, the First-Come, First-Served algorithm is a straightforward scheduling strategy that is simple to comprehend and put into practice. Other algorithms for scheduling that take job priorities, due dates, or resource optimization into account may be more appropriate in more complex settings.

2.1.7. Round Robin (RR) Algorithm

This algorithm works in the form of a ring queue. Each task is assigned with a quantum of time and these sorted tasks are assigned to the available virtual machines in a circular form. If a task is not executed in the given time, then it will be interrupted and the next task will start its execution and the previous task will wait again for its turn. This algorithm continues until each task is assigned to at-least one virtual machine. This algorithm maintains fairness by giving each process an equal chance to succeed. It stops one process from using the CPU exclusively for a long period of time. Even if a process takes longer than predicted to finish, it will be interrupted once the time quantum has passed and put back in the queue so that other processes can run in the interim. Quick responsiveness and the appearance of simultaneous execution are made possible by this preventive behavior. When numerous users or jobs compete for CPU time in time-sharing systems, round robin scheduling is frequently utilized. It guarantees that resources are distributed fairly and that each process gets some of the processor's time.

The disadvantage of this algorithms is server overloading because if a task requires larger execution time, then it will not be completed in the given time and it will cause many switching until it is fully executed.

2.1.8. Shortest Job First (SJF) Algorithm

In this algorithm, all tasks are sorted in a queue. Small tasks are arranged in the start of the queue and long tasks are at the conclusion of the queue. The waiting time of the small tasks which are at the start of queue will be smaller and the waiting period of larger tasks that are at the end of the queue will be larger. This will increase the waiting time of longer tasks.

SJF scheduling can produce the best outcomes in terms of minimizing wait times and turnaround times, but it needs precise predictions of all process burst times. Accurately predicting burst times can be difficult in practice, and differences in actual execution durations can impact the algorithm's effectiveness.

It is important to keep in mind that the SJF algorithm is not always the best option when burst times are erratic or unknown beforehand. Additionally, it may experience a condition known as "starvation," in which lengthy procedures are repeatedly postponed in favor of lesser ones.

2.1.9. Heterogeneous Earliest Finish Time (HEFT) Algorithm

In heterogeneous computing systems, the Heterogeneous Earliest Finish Time (HEFT) algorithm is used to schedule tasks. By taking into account the computing and communication costs of jobs operating on different resources, it seeks to optimize assignment and scheduling decisions.

Task execution times in heterogeneous computing systems vary depending on the properties of the resources they are allocated to. When scheduling tasks, the HEFT method takes into account both the computing time of a task on a particular resource and the interaction time between tasks.

The makespan, or total amount of time needed to accomplish all jobs in the schedule, is what the HEFT method seeks to reduce. It attempts to balance the load among several resources and shorten the overall execution time by taking into account both computation and communication expenses.

2.1.10. Critical Path on A Processor (CPOP) Algorithm

The HEFT algorithm is very similar to this one. The task prioritization phase comes first, followed by the task selection phase. Priority queues are employed during the task prioritizing phase, and the crucial path to the processor is identified during the task execution phase. EFT will be employed if a critical path is not accessible. The heterogeneous cloud computing environment uses this algorithm.

2.1.11. Synthesized Heuristic Task Scheduling (HCDPPEFT) Algorithm

This algorithm's foundation is based on list base and duplication base techniques. It consists of two phases. One is task prioritization by resource selection and second is optimization of replication of different tasks.

From all the research, it has been clear that task scheduling is a major concern in cloud computing. We have studied many other existing task scheduling algorithms through different research articles for comparative analysis of our algorithm. Some of those articles are as follows:

- Parental Prioritization-Based Task Scheduling in Heterogeneous Systems
- HTD: heterogeneous throughput-driven task scheduling algorithm in MapReduce
- Task scheduling and resource allocation in cloud computing using a heuristic approach
- Task Scheduling Optimization in Cloud Computing (Applying Multi-Objective Particle Swarm Optimization)
- Multi-objective Task Scheduling in Cloud Environment using Decision Tree Algorithm

We have also studied about scientific workflows (Montage and Epigenomics) which are available for open use in task scheduling algorithms.

2.1.12. Differential Evolution (DE)

2016 saw the publication of a study by C. Wu and colleagues titled "Differential Evolution-Based Task Scheduling for Heterogeneous Computing Systems in Cloud Computing" in the IEEE Transactions on Cloud Computing. In diverse cloud environments, DE is a population-based optimization approach for work scheduling. By using mutation and crossover procedures, it continually develops a population of potential solutions. DE takes into account a number of goals, including reducing makespan, consuming less energy, and maximizing resource utilization. It has been successful at resolving issues with dynamic job scheduling in heterogeneous cloud computing.

3. Problem Formulation

In heterogeneous distributed systems of cloud computing (HeDSCC), a parallel application's planning for dynamic work is described by a DAG. It is presented as a tuple and comprises n tasksof set T and the number of edges e. (T, E). Each job *ti* T in a parallel processing, and the communication link among *ti* and *tj* is symbolized by an edge (*ti, tj*). If a (*ti, tj*) E, then the execution of *ti* T must be complete before the execution of *tj*, T may begin. The job *tj* is a child job of edge (*ti, tj*) while *ti* is a parental task. A DAG task is referred to as an initial task if task *ti* does not have a parent, and a final task if task *tj* does not have a child.

So, each edge (*ti, tj*) has a value (*CCij*) associated with it that represents the cost of communication across jobs (*ti, tj*). The HeDSCC have m processors of set P, each with a different set of calculation capabilities. The cost of the computation, m*n, is contained in the matrix C, where m represents the total amount of processors and n, the number of tasks.

Since each processor is assumed to be fully linked, each element of matrix *CTij* C displays the computing complexity of task *ti* on processor *pj*. The autonomous conversation unit is utilized for the execution of concurrent jobs on various processors and aids in processor communication. The amount ofdata communicatedon the edge (*ti, tj*) when two tasks are arranged on multiple processors equals the communication cost (*CCij*)from task (*ti*) to task (*tijs*). The communication cost between tasks *ti* and *tj* will be considered to be zero if theyare carried out onthe same processor.

Only once the parent task has finished running and all necessary data has been made available to the processor can the child task begin running. In order to execute jobs in parallel, each task must be given to the processor in a schedule that allows for a minimum overall makespan (runtime).

A DAG for the relying task is demonstrated in Fig.3 along with a matrix representing the computation time.

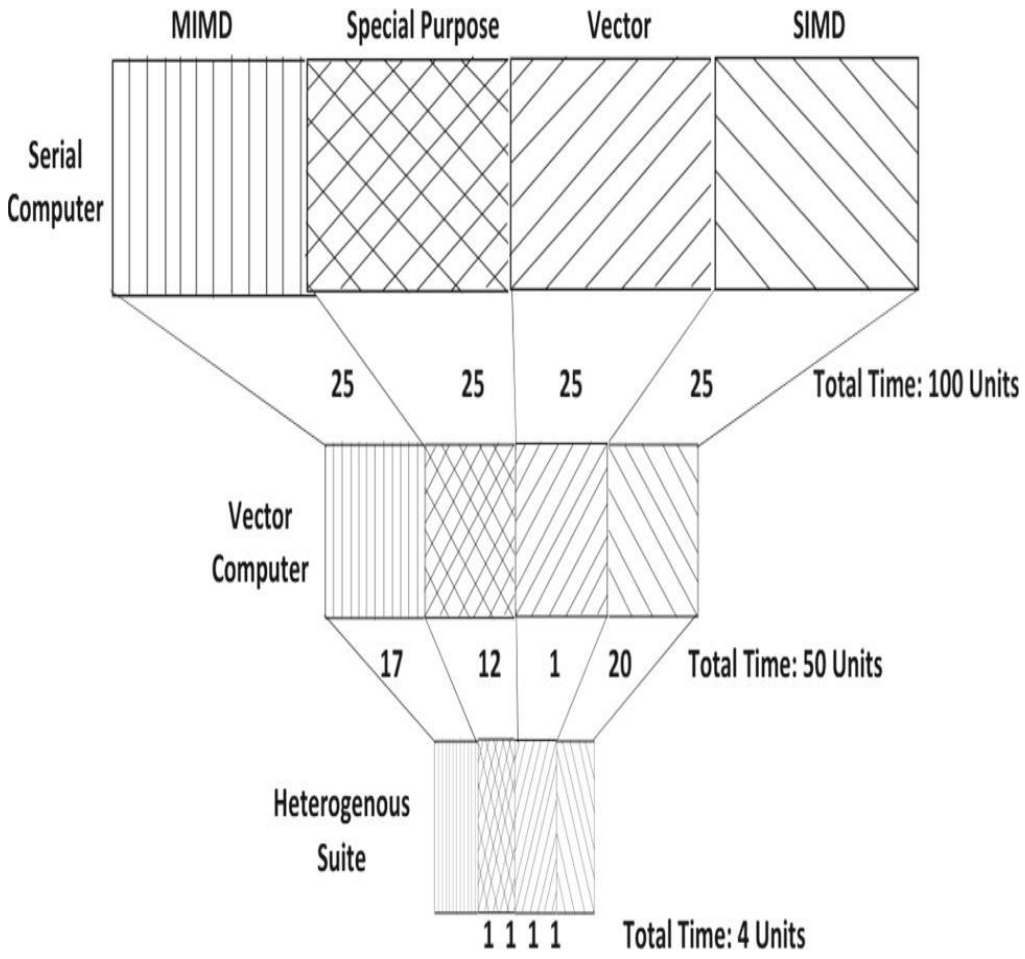


Fig. 3 Hypothetical homogeneous and heterogeneous systems

With t0 serving as the first task and t10 serving as the last job, Fig. 2 illustrates a DAG with 11 tasks. The completion of task t0 and the availability of all data are prerequisites for the execution of tasks t1, t2, t3, t4, t5, and t6. The communication cost of an edge (t0, t1) is five, while that of an edge (t1, t7) is seven. Tasks t7, t8, and t9 are prerequisites for task t10.

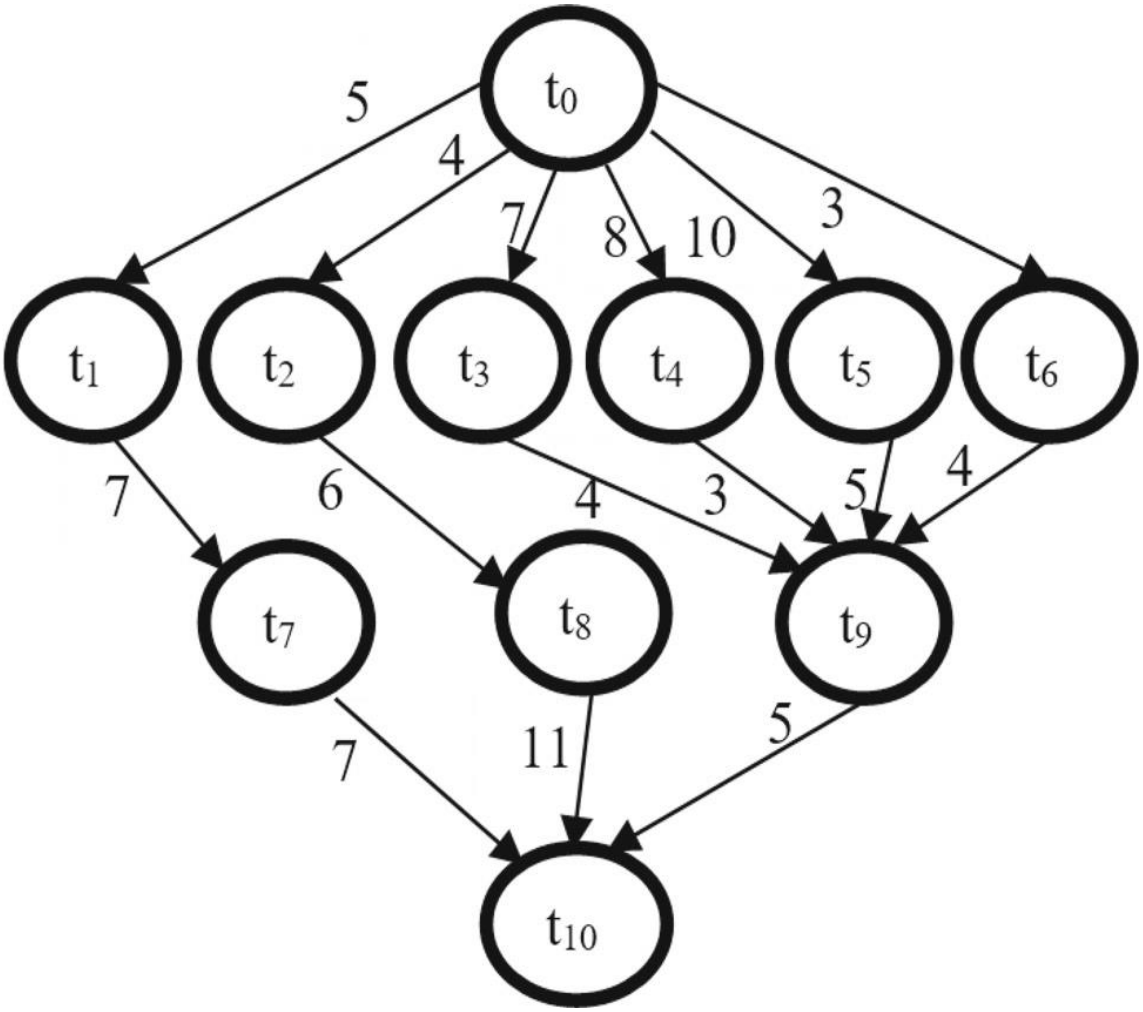


Fig. 4 DAG example for dependent tasks

4. The Proposed Algorithm Dynamic Parental Prioritization Earliest Finish Time (DPPEFT)

The DPPEFT algorithm is suggested for scheduling interdependent jobs in a heterogeneous setting. The DPPEFT algorithm comprises two stages: the task prioritization stage and the processor assignment stage for task execution. The step of prioritizing tasks assigns a ranking to each task based on parental priority. The task is given to the processors during the processor selection step based on the earliest finish time.

4.1. Task Prioritization Phase

The task prioritizing phase (TTP) is a critical stage in the task scheduling process. In this stage, each task's priority is determined based on ranks determined by parental prioritizing. By ranking each task, a list of tasks is created. A DAG's ranks are determined top-down, beginning with the first job. The order of tasks that depend on the original job is then determined. Combining mean computation time and communication costs (CCs), the rank ratings of each task are determined. If the job on the next layer has a lower rank value and is not relying on any other tasks for whichorder value is not determined, the parental prioritization has the advantage of scheduling it aheadof the tasks on the current layer in the DAG. The maximum communication cost of the parent job,the rank value, and the average computation time (*CTi*) of all processors of the given tasks can beused to calculate the rank of the node *ni*. To allocate to the processor, all of the estimated scoresare arranged in a queue after being filtered by descending order.

The definition of task *ni* is used to determine its rank as

$$rank\ (n_i)=\text{CTi}+\ (CC_{ij}\ +\ rank(n_j))\quad (1)$$

where *ni* is the active node, *CTi* is the task's mean computation time, par(*ni*) are the task's parent tasks, and *CCij* is the edge's communication cost (*ti , tj*).Since ranks are determined by moving from task to task in the tasks graph, it begins with the first task. Initial mission n_{ini}'s rank is equivalent to

$$rank\ (n_{ini})\ =\ CT_{ini}\quad (2)$$

Where CT_{ini} is the initial task's average computation time and n_{ini} is the first node.

4.2. Processor Assigning Phase

According to the tasks set in the tasks prioritizing phase, the processor allocates each task during this step. Most task scheduling algorithms take into account the earliest accessible time, which isthe moment the processor finished running. However, the DPPEFT algorithm takes into account the placement strategy, which assigns jobs to the processor based on their earliest start time (EST)and earliest finish time (EFT). The task should be started as soon as possible after EST. One may figure out the EST of node *ni* by

$$EST\ (\text{n}_\text{i},\ p_j)\ =\ \max[\text{available}(p_j),\ (\text{CT of}\ p_j\ +\ CC_{ij})]\qquad (3)$$

where *ni* is the active node, *pj* is the processor's total processing time, avail(*pj*) is the processor's availability, and *CCij* is the communication cost. Because computation time on each processor begins at zero, the EST for the starting task is considered to be zero for each CPU.

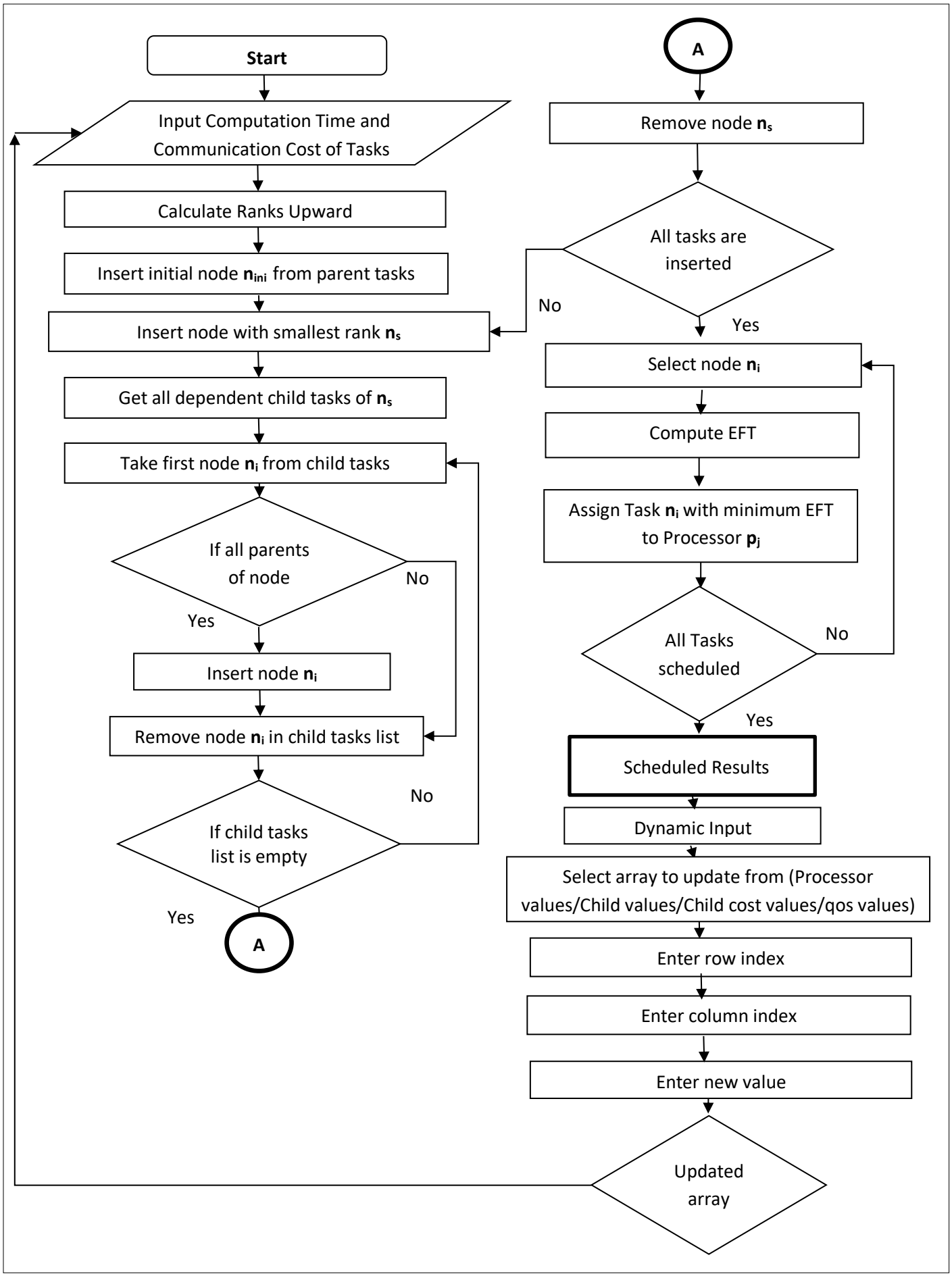
EFT can be determined by using

$$EFT\ (n_i) = CT_i + EST\ (n_i, p_j) \qquad (4)$$

where *CTi* is the average processing time across all processors. EFT, which takes into account EST and each processor's computation for task *ni*, is the earliest finish time of jobs on each processor. Task *ni* will be assigned on the processor with the lowest EFT value, which is considered to be the idle time slot for the task.

In below given fig.5 inputs are task-specific computation time and communication costs. Real-world application workflows (Montage and Epigenomics) and RTGG provide two different forms of input. Our proposed method carries out the scheduled tasks at the flowchart's conclusion.

Fig.5 DPPEFT algorithm (flowchart)


Start
Input Computation Time and Communication Cost of Tasks
Calculate Ranks Upward
Insert initial node n_{ini} from parent tasks
Insert node with smallest rank n_s
Get all dependent child tasks of n_s
Take first node n_i from child tasks
If all parents of node
No
Yes
Insert node n_i
Remove node n_i in child tasks list
If child tasks list is empty
No
Yes
A
A
Remove node n_s
All tasks are inserted
No
Yes
Select node n_i
Compute EFT
Assign Task n_i with minimum EFT to Processor p_j
All Tasks scheduled
No
Yes
Scheduled Results
Dynamic Input
Select array to update from (Processor values/Child values/Child cost values/qos values)
Enter row index
Enter column index
Enter new value
Updated array

Chapter 3

5. Methodology

Cloud computing is the on-demand provision of IT services through the Internet with pay-as-you-go charges. You can access innovation administrations like computing control, capacity, and databases on an as-needed basis from a cloud provider like Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), Alibaba Cloud, and Oracle Cloud rather than purchasing, owning, and maintaining physical data centers and servers.

Every size, industry, and kind of business is utilizing the cloud for a variety of use cases, including data backup, disaster recovery, email, and virtual desktops, program testing and development, large data analytics, and web apps with a focus on customers. For instance, healthcare organizations are using the cloud to develop more patient-specific medications.

In order to manage meaningful extortion recognition and anticipation, financial services providers are turning to the cloud. The cloud is being utilized by video game companies to provide online games to millions of players worldwide. Further advantages of cloud computing include flexibility, agility, and cost savings quickly deploy internationally.

Task scheduling divides a Framework program into smaller tasks and assigns each task to a certain set of resources based on pre-built QoS models in order to support or enhance the QoS that the client requires for the application.

The following list includes many work scheduling algorithms:

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin (RR) Scheduling
- Multiple-Level Queues Scheduling

5.1. Scientific Workflows

Montage and Epigenomics are scientific real-world methods that we use. These processes are useful for planning and arranging jobs in diverse environments. One of the most significant open-source processes that we employ for task scheduling is the Montage workflow. The Pegasus workflow management system is used in conjunction with the Epigenomics data processing pipeline to automate the various sequencing operations.

Below is a brief description of Epigenomics and Montage workflows.

- **Montage workflow:**

Montage workflows frequently include numerous actions that must be completed in a particular order and have dependencies among one another. It can take a long time to complete because of the complexity and amount of data involved. Montage workflows must be scheduled well to save entire execution time and optimize the use of resource.

Here is an explanation of a common cloud computing job scheduling montage workflow:

- **Task Identification:** Identifying and defining the specific tasks that must be completed in the cloud environment is the first stage. Every task should have a defined aim as well as specifications, including input information, expected results, and execution time restrictions.

- **Resource Discovery**: Find the cloud resources that can be used to carry out the activities, such as containers, virtual machines, or server less functions. Take into account each resource's computational power, memory, and other technical details.

- **Job-to-Resource Mapping:** Construct a mapping that allots each job to the proper resource based on the qualities of the jobs at hand and the resources that are at your disposal. Take into account elements like task importance, dependencies on data, and the resources' capacity to accomplish particular tasks.

- **Data management:** Take into account how data will be dispersed among the resources if the jobs entail processing data. To prevent information access conflicts or bottlenecks, assure information dependencies are properly maintained.

- **Load Balancing:** To equally spread the computational demand across the available resources, load balancing is essential. By doing this, resource overload is prevented and optimal resource use is achieved.

- **Task Scheduling:** Plan the tasks to be executed on the chosen resources based on the task-to-resource alignment and load balancing factors. Algorithms for task scheduling can be used to streamline resource distribution and cut down on total execution time.
- **Task Monitoring:** Implement monitoring procedures to keep tabs on the performance and progress of each task. If necessary, the task scheduling can be dynamically adjusted by reallocating tasks to alternative resources or changing priorities using this information.

- **Fault Tolerance:** Because cloud environments are prone to malfunctions, fault tolerance methods must be included. To assure job completion even in the case of resource failure, this may entail replicating tasks across various resources or employing backup mechanisms.

- **Coordination and Task Execution:** Carry out the scheduled tasks in the cloud. If the outcomes of one activity are dependent on another, make sure that there is adequate coordination and synchronization between the tasks.

- **Task completion and result aggregation:** After every task has been finished, combine the results to make sure the output is accurate and consistent.

- **Resource Release:** Release the resources after the workflow is complete to avoid wasting resources and incurring extra costs.

It is necessary to take into account elements like job characteristics, resource capabilities, and dependencies on data, load balancing, tolerance for failures, and real-time monitoring while optimizing a montage workflow for cloud computing task scheduling. The workflow can improve efficiency and resource utilization in the cloud environment by efficiently handling these variables.

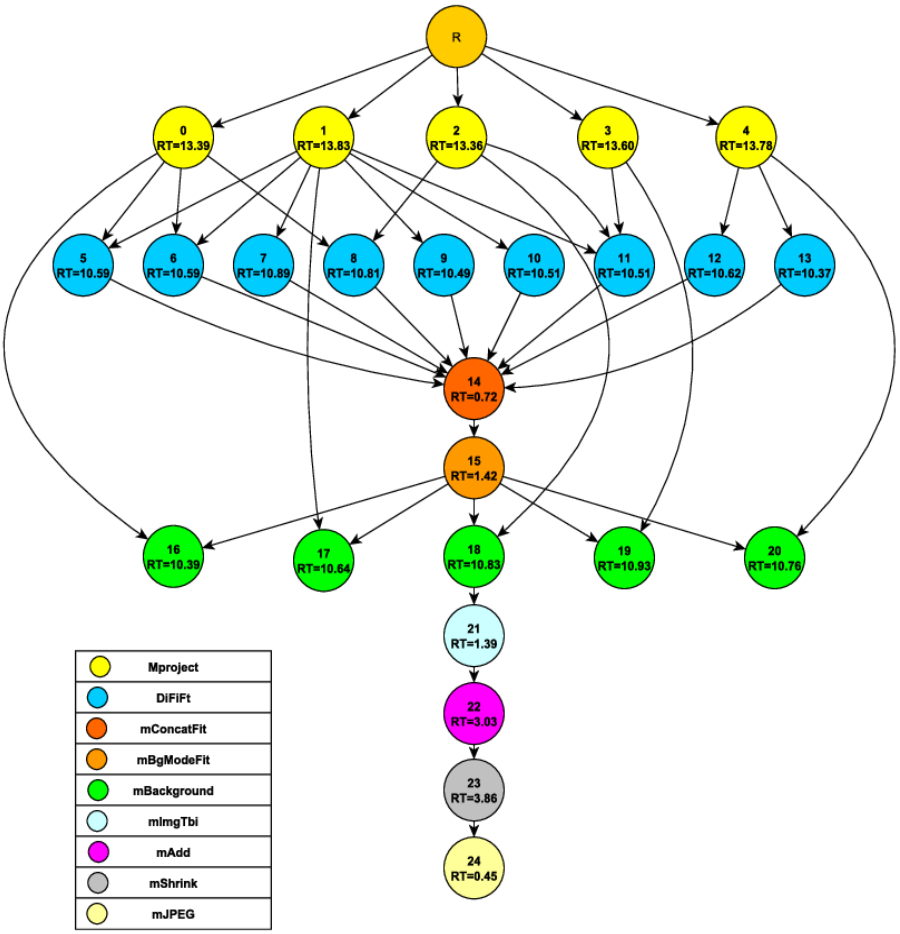


Fig. 6 Montage Workflow

➢ Epigenome Workflow

Epigenome workflows are data-intensive and demand a lot of computation, memory, and storage space. Epigenome process analysis steps frequently involve complex computational needs, data dependencies, and interdependent analysis procedures. For epigenome workflows to run quickly and make it possible to analyze genomic data on time, task scheduling is essential.

In the context of genomics research, epigenome workflows can be used as practical examples to assess the efficiency and efficacy of task scheduling algorithms. The unique needs and limits of epigenome workflows, such as data locality, data dependencies, and fluctuating resource requirements across different analysis phases, can be taken into account via task scheduling algorithms. Task scheduling algorithms can optimize resource allocation and scheduling choices to reduce execution time and improve resource utilization by knowing the features of epigenome operations.

A procedure for an epigenome in a heterogeneous cloud computing environment is described as follows:

- **Task Analysis and Profiling:** The individual jobs in the epigenome workflow should be examined to determine their computing needs, dependence on data, and resource preferences. While certain jobs could require a lot of CPU power, others might need a lot of memory or storage.

- **Resource Selection:** Determine the different resource kinds and cloud platforms that are accessible in the heterogeneous environment. This could comprise edge devices or specialized hardware, private cloud infrastructure, and public cloud service providers (such AWS, Azure, and Google Cloud).

- **Task-to-Resource Mapping:** Map each task to the most appropriate cloud resource using the task analysis and resource profiles as a guide. For instance, outstanding performance virtual machines with numerous cores may be used for CPU-intensive operations, whereas instances with sufficient RAM may be used for jobs that require a lot of memory.

- **Load Balancing:** Implement load-balancing techniques to evenly spread the burden among the various cloud resources. As a result, the chance of resource bottlenecks is reduced and optimal resource utilization is ensured.

- **Interoperability and Data Transfer:** Addressing issues with data compatibility and interoperability between various cloud systems and resource kinds. For the epigenome workflow to run smoothly, efficient data transfer and compatibility among different resources are essential.

- **Task Scheduling:** Utilize dynamic task scheduling techniques to assign jobs to the appropriate resources based on their performance and real-time availability. Because of the environment's heterogeneity, flexible scheduling systems are needed to respond to shifting circumstances.

- **Resource Orchestration:** Implement resource orchestration technologies that can handle provisioning and deallocation, manage auto-scaling based on the fluctuating workload demands, and manage the lifecycle of resources.

- **Redundancy and fault tolerance:** Include fault tolerance methods to make guarantee that the workflow can continue in the event that any of the diverse resources fail. Task repetition, backups of data, and failover techniques might be used in this.

- **Security and Data Privacy:** Address security issues in the heterogeneous cloud environment, particularly when several sources or platforms are used to store the data. Make sure that data is secure and legal compliance, and put in place the right access controls.

- **Monitoring and Performance Optimization:** To find bottlenecks and improve the workflow, it is crucial to continuously monitor task execution and resource performance. Decisions about resource allocation and task schedule optimization can be guided by performance statistics.

- **Result Aggregation and Integration:** Aggregate and combine the results into a single dataset for further analysis and interpretation when the tasks have been performed across the diverse cloud resources.

Utilizing several cloud providers and resource kinds, heterogeneous cloud computing environments have the advantages of flexibility and cost-effectiveness. However, they also present issues with interoperability, data mobility, and resource coordination. To establish a high-performance epigenome workflow in such circumstances, careful planning, effective task scheduling, and effective resource management are necessary.

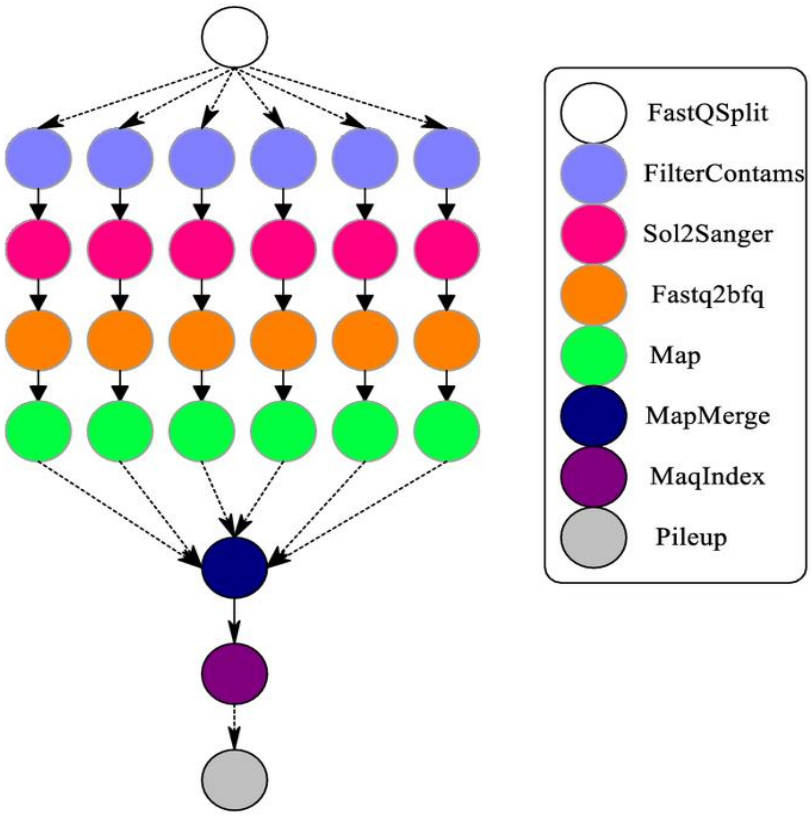


Fig. 7 Epigenomics Workflow

The various properties and requirements of Montage and Epigenome workflows make them useful in the creation and analysis of job scheduling algorithms. These workflows are examples of computationally and data-intensive tasks, where effective scheduling is essential to maximizing resource utilization and cutting down on execution time. In heterogeneous cloud computing settings, job scheduling algorithms can execute and function more efficiently by using the distinctive features of Montage and Epigenome workflows.

Project Flow chart

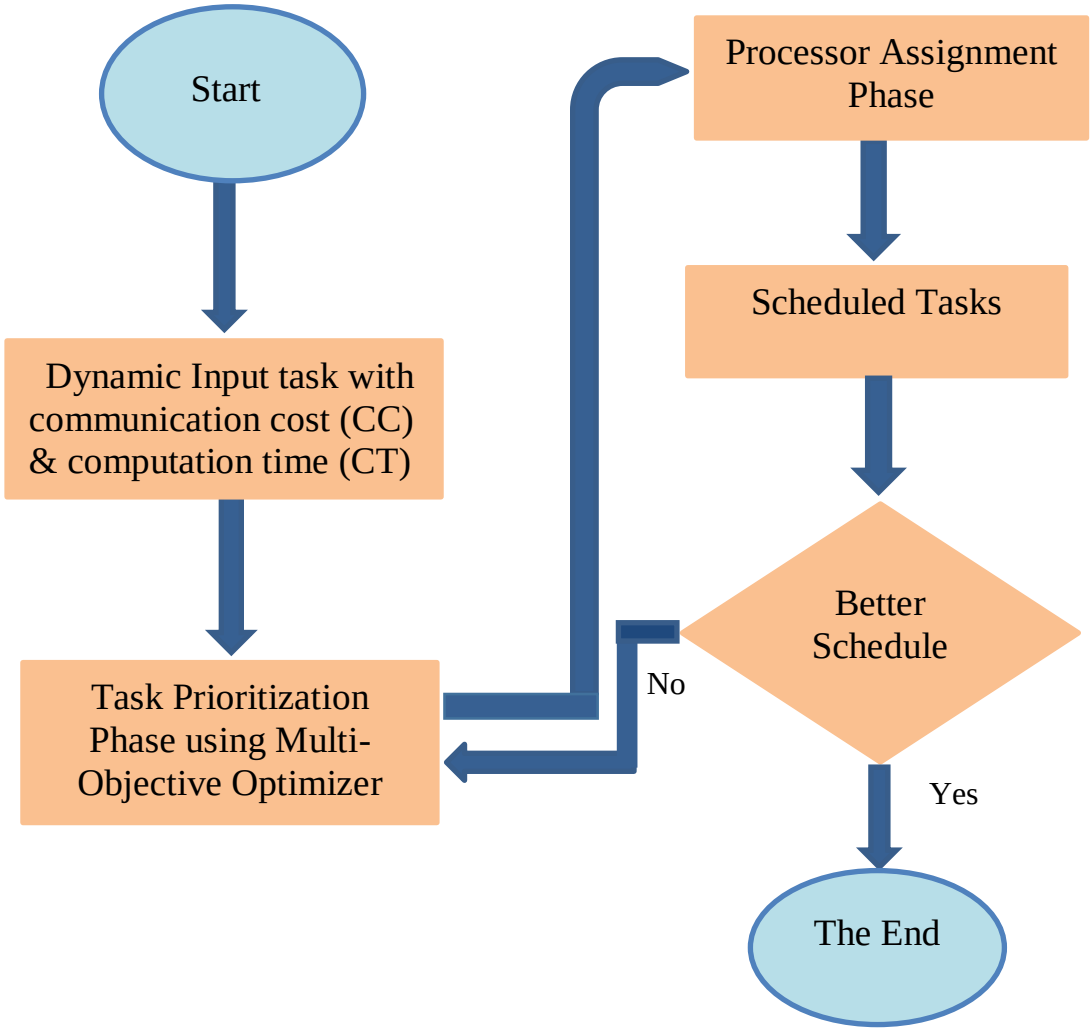


Fig. 8 Project Flowchart

5.2. Experimental Setup

CloudSim 3.0.3 is being utilized to simulate the suggested approach.

5.3. Objectives

The proposed algorithm will:

- Minimize make span.
- Maximize the resource utilization.
- Provide Dynamic task Scheduling (Handles dependencies that are unknown at compiled time)

5.4. Experimental results and Analysis

Here we are comparing the effectiveness of our suggested algorithm DPPEFT against widely recognized scheduling algorithms like HEFT and CPOP. On Cloud Sim 3.0.3, a simulation environment is utilized and Workflow Sim 1.0 with graphs produced using a python module to examine the performance of job scheduling methods. With random and manually applied DAGs, several sets of dependent task graphs are formed. We also conducted experiments utilizing conventional scientific procedures. To produce the schedules of the tasks that are given, each task scheduling algorithm is applied to the tasks DAG.

5.4.1. Comparison Metrics

The following performance indicators are used to compare algorithms:

5.4.1.1. Makespan

The makespan is the most popular comparative metric for a single DAG. The make span is the amount of time it takes for the final task in the input DAG to finish running. The execution time of the exit node is, in other words, the algorithms make span for a DAG.A definition of an algorithm's make span is

$$\text{makespan}=\max \text{AFT (last node)}$$

Where AFT (last node) is the input graph's exit node's actual finish time. The longest finish time across all exit nodes is taken into account when there are many exit nodes in the graph.

5.4.1.2. Scheduling length ratio (SLR)

The makespan is represented by the Scheduling Length Ratio (SLR), which is the lower bound normalised to it.As outlined by the SLR,

$$SLR = \frac{makespan}{\min\{CPi,j\}}$$

The Critical Path Including Communication (CP*min*) represents the critical path tasks' lowest possible cost, taking into account communication expenses. When an SLR is lower, the method is preferable since there is no makespan less than the CP*min*.

Step	EFT			Task Selected	Assigned Processor
	P0	P1	P2		
1	32	34	27	t0	P2
2	28	14	23	t9	P1
3	21	27	30	t7	P0
4	47	53	52	t1	P0
5	44	38	46	t8	P1
6	35	43	39	t6	P2
7	21	27	30	t5	P0
8	45	47	53	t2	P0
9	37	38	35	t4	P2
10	44	39	48	t3	P1

Table 1. Schedule produced by the DPPEFT algorithm in each iteration

➢ Performance Comparison

The performance of our proposed algorithm DPPEFT is compared with HEFT, CPOP and PPEFT algorithms on the basis of makespan and average SLR.

Figure (9, 10, 11, 12, 13 and 14) shows the total execution time (makespan) of each algorithm for different cases based on the number of tasks being executed. In each case the overall execution time of DPPEFT algorithm is less than HEFT, CPOP and PPEFT algorithms.

In **Figure15** Average SLR of DPPEFT, PPEFT, HEFT and CPOP algorithms is calculated. The average Schedule Length Ratio (SLR) of our proposed algorithm (DPPEFT) is smaller as Compared to other algorithms based on the number of tasks being executed.

Figure16 shows the comparison of Average SLR of 12 tasks for HEFT, CPOP, PPEFT and DPPEFT algorithms.

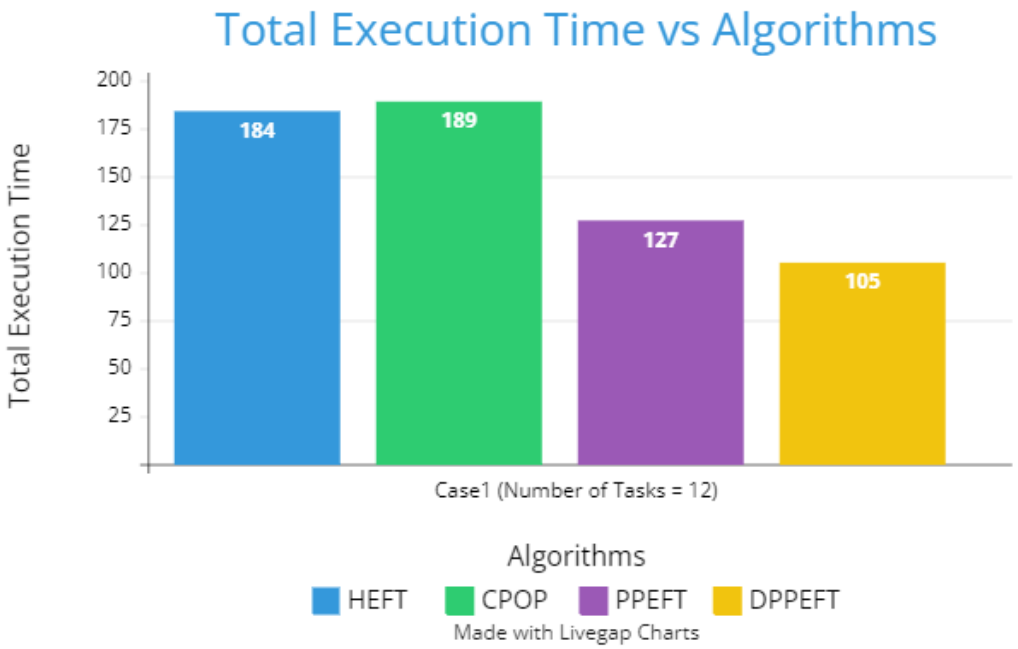

Total Execution Time vs Algorithms
Total Execution Time
200
175
150
125
100
75
50
25
184
189
127
105
Case1 (Number of Tasks = 12)
Algorithms
HEFT
CPOP
PPEFT
DPPEFT
Made with Livegap Charts

Fig.9

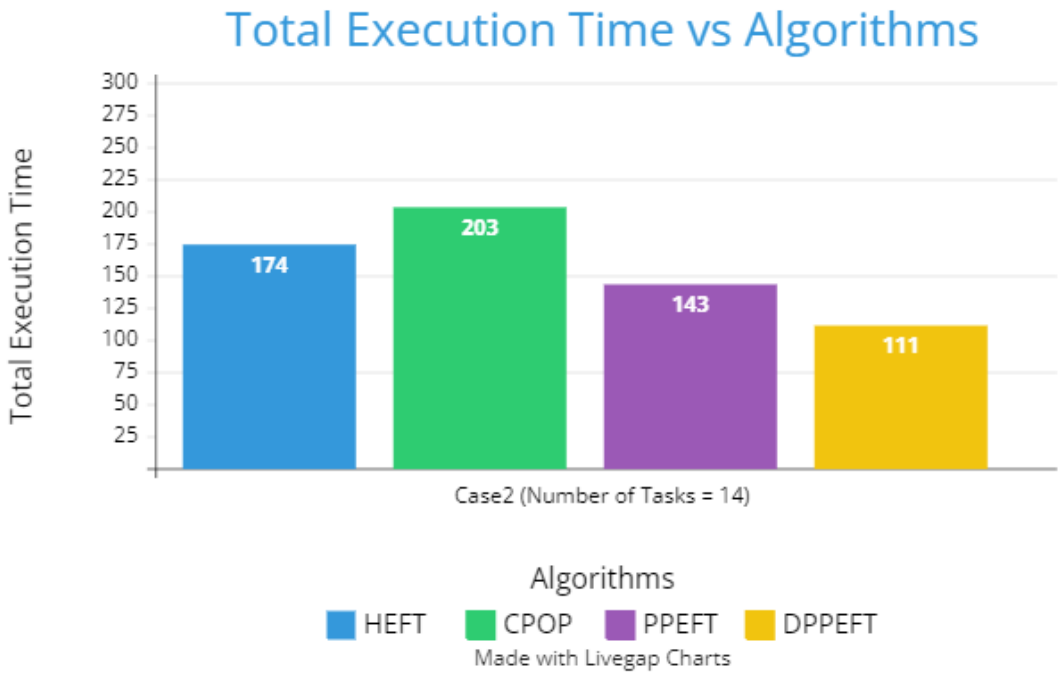

Total Execution Time vs Algorithms
Total Execution Time
300
275
250
225
200
175
150
125
100
75
50
25
174
203
143
111
Case2 (Number of Tasks = 14)
Algorithms
HEFT
CPOP
PPEFT
DPPEFT
Made with Livegap Charts

Fig.10

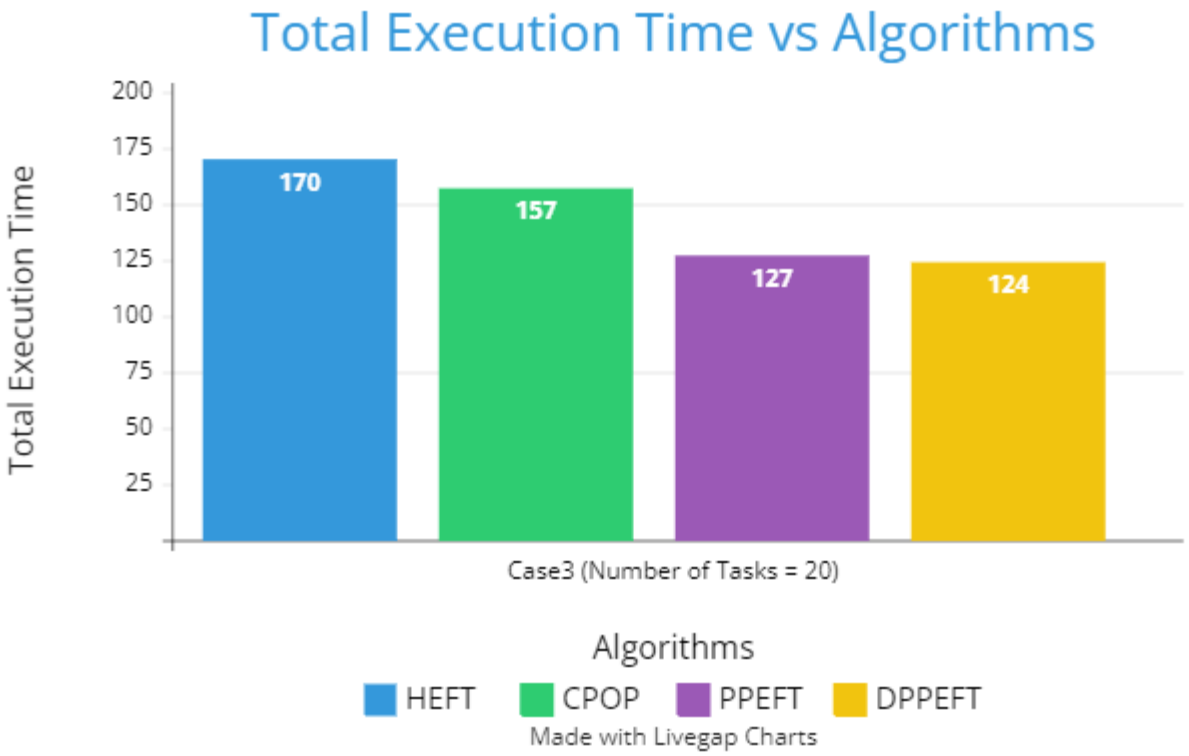

Total Execution Time vs Algorithms
Total Execution Time
200
175
150
125
100
75
50
25
170
157
127
124
Case3 (Number of Tasks = 20)
Algorithms
HEFT
CPOP
PPEFT
DPPEFT
Made with Livegap Charts

Fig.11

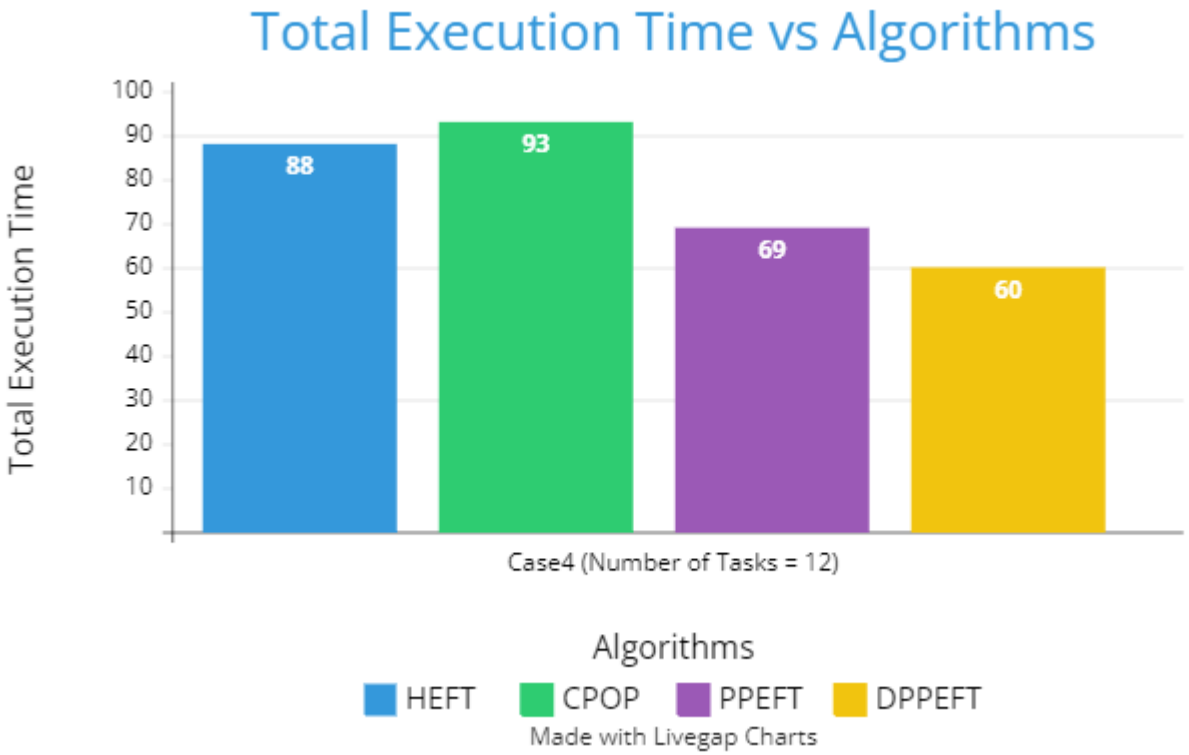

Total Execution Time vs Algorithms
Total Execution Time
100
90
80
70
60
50
40
30
20
10
88
93
69
60
Case4 (Number of Tasks = 12)
Algorithms
HEFT
CPOP
PPEFT
DPPEFT
Made with Livegap Charts

Fig.12

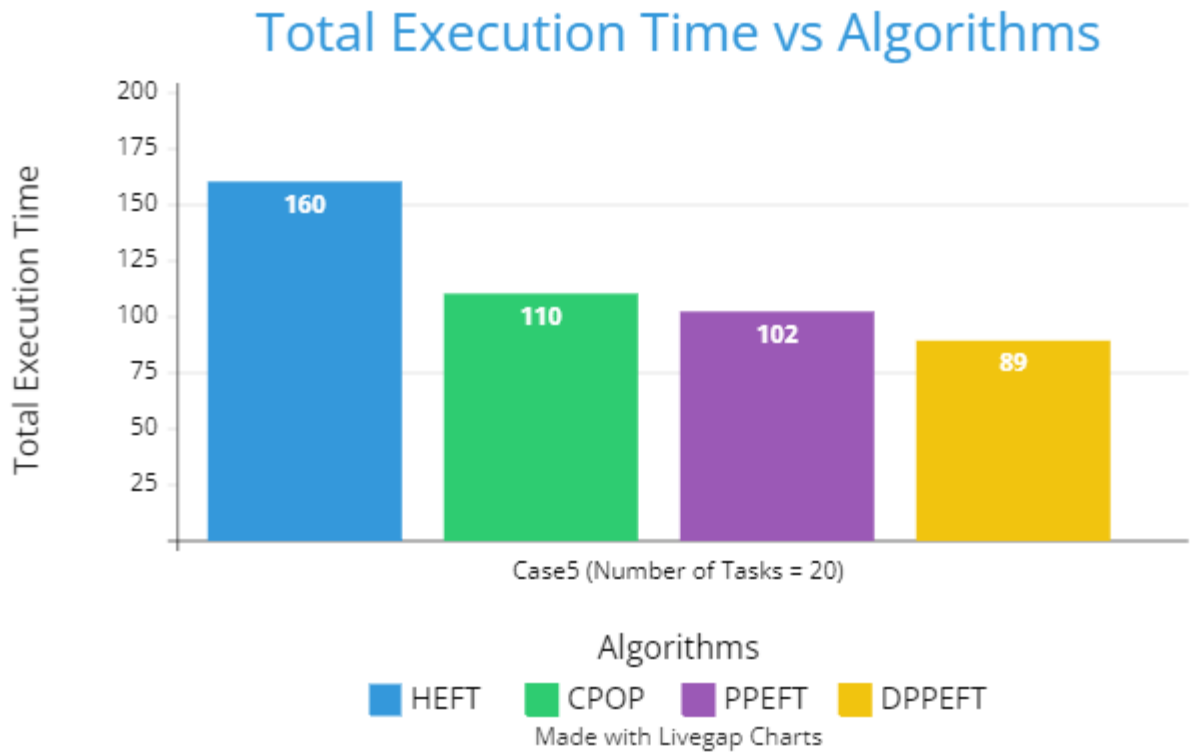


Fig.13

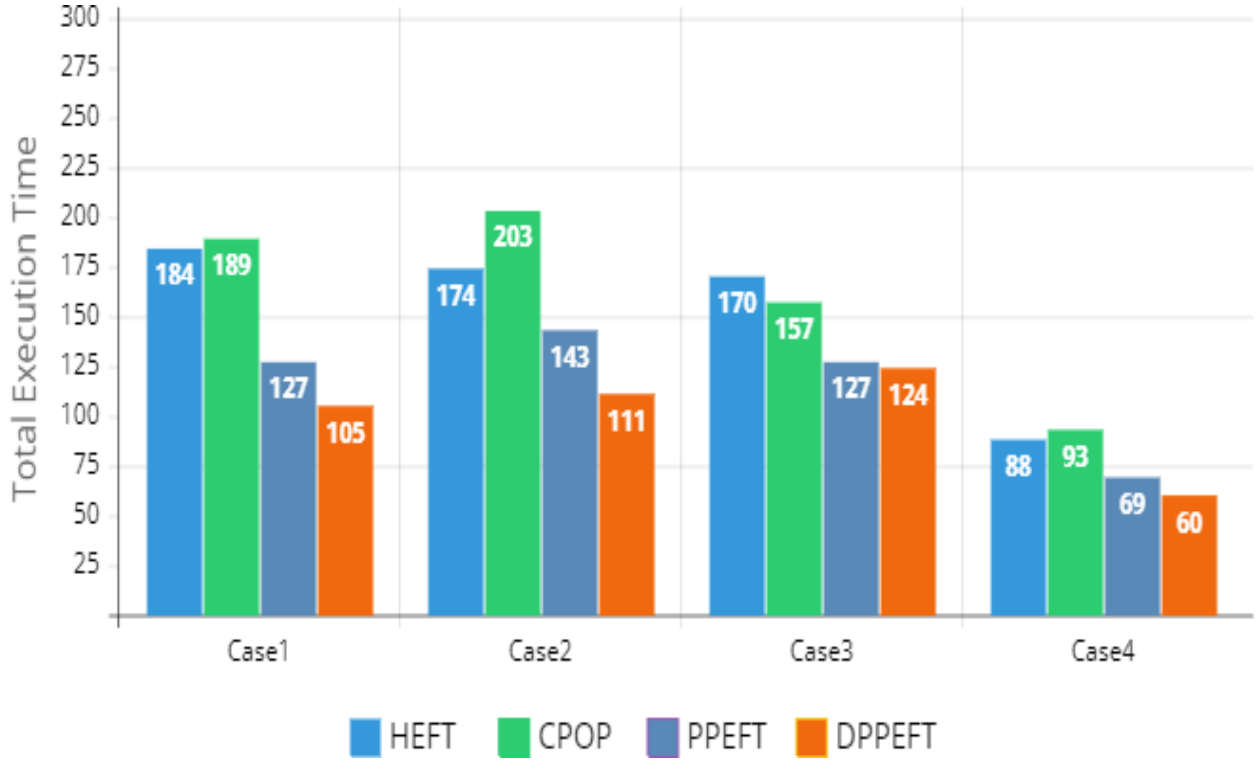


Fig.14: Total Execution Time of each algorithm (4 cases)

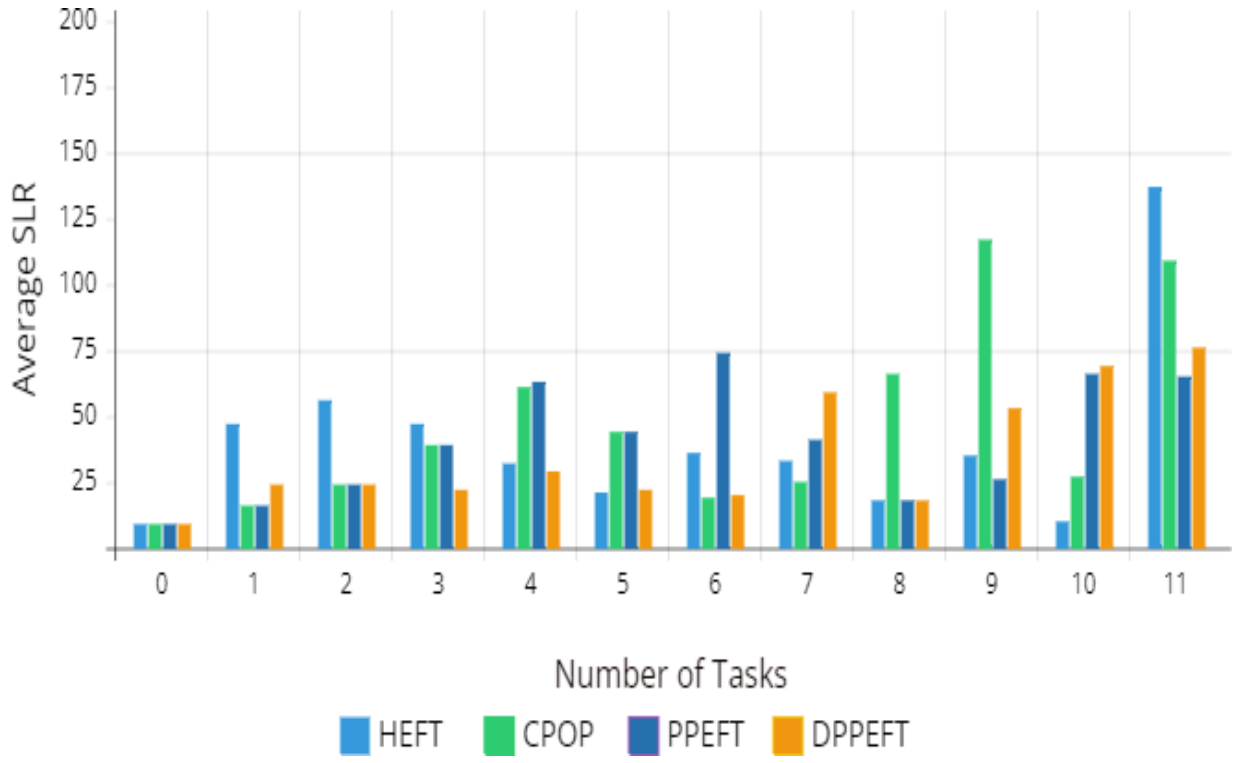


Fig15: Average SLR of 12 tasks

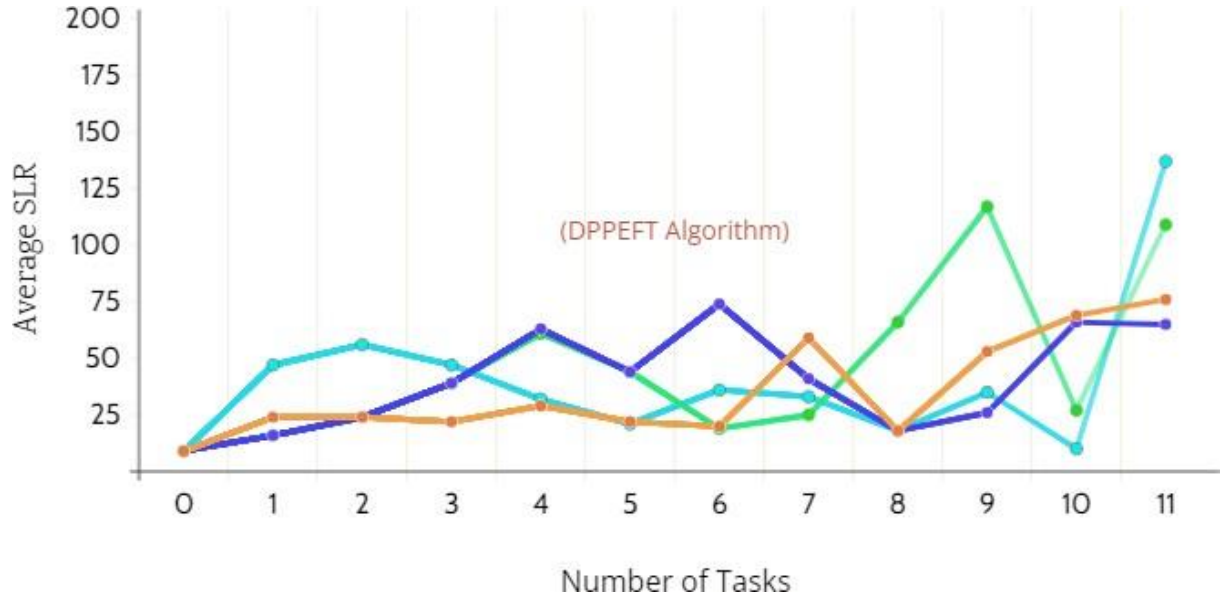


Fig16: Average SLR with 12 Tasks

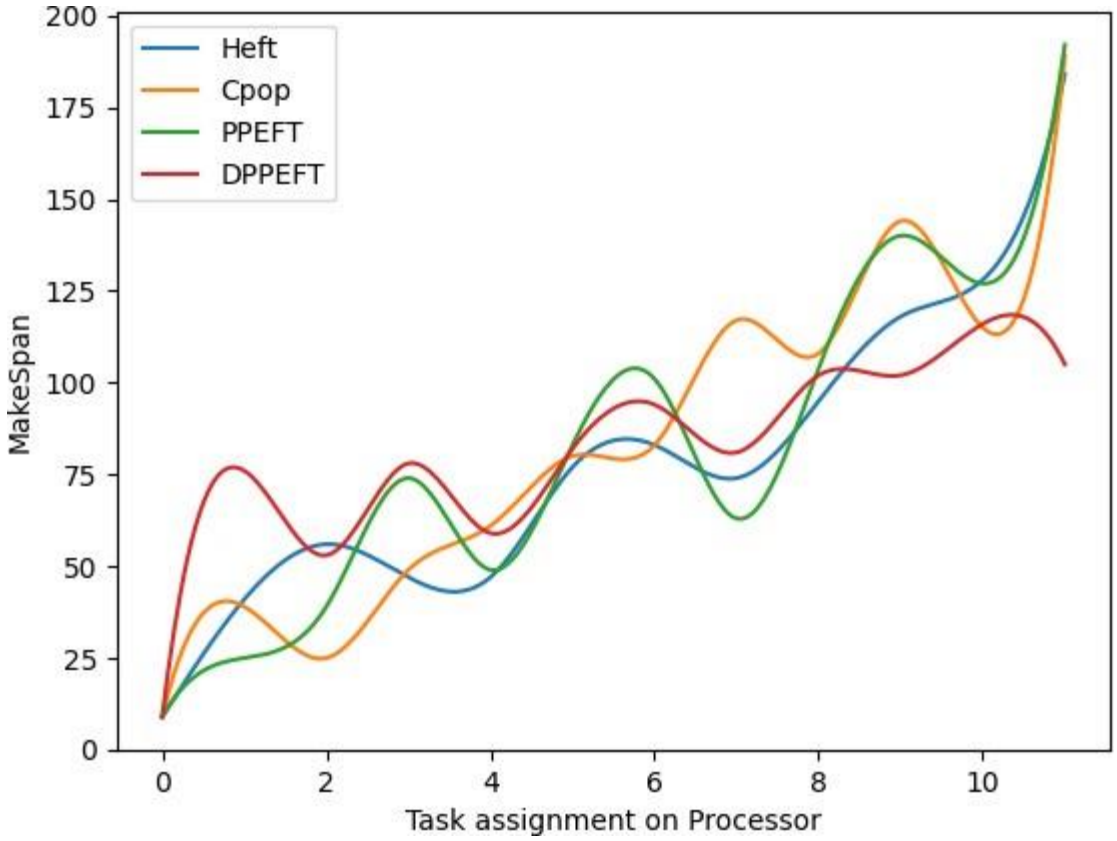


Fig17: Task Assignment on Processor with Makespan

These results show that the total execution time of DPPEFT algorithm is less than the CPOP, HEFT and PPEFT algorithms. Average SLR of each algorithm is also compared based on the number of tasks in each case because with an increase in the number of tasks, the SLR value also changes. **Figure(17)** shows the assignment of tasks to different processors based on their makespan(execution time).

6. Conclusion& Future work

In this paper, we introduce DPPEFT, a dynamic task scheduling technique for heterogeneous computing systems. Task prioritizing and processor assignment are the two phases of this novel technique. In this algorithm we are taking input dynamically. In the task prioritizing phase, tasks are planned based on parental priority and descending ranks. DPPEFT schedules jobs using the parental prioritization queue (PPQ). PPQ is used to schedule lower communication cost dependent tasks from the next row of the task graph (DAG) before the present job.

The list of tasks derived from the PPQ that require the least amount of computing time is assigned to the processor duringthe processor assignment phase. As a result, DPPEFT produces results for

work scheduling whichare more effective than those of other task scheduling algorithms. Our newly developed algorithm DPPEFT's performance is compared to that of the widely recognized algorithms HEFT, CPOP and PPEFT. In terms of makespan and resource utilization the DPPEFT algorithm produces better results. For future we want to extend this DPPEFT algorithm to respond better in terms of network load and efficiency.

7. References

[1] Task scheduling for heterogeneous computing system,
https://link.springer.com/article/10.1007/s11227-016-1917-2

[2] Parental Prioritization-Based Task Scheduling in Heterogeneous Systems,
https://link.springer.com/article/10.1007/s13369-018-03698-2

[3] Task Scheduling Optimization in Cloud Computing Based on Genetic Algorithms,
https://www.techscience.com/cmc/v69n3/44156

[4] Task scheduling and resource allocation in cloud computing using a heuristic approach,
https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-018-0105-8

[5] Multi-objective Task Scheduling in Cloud Environment using Decision Tree Algorithm,
https://ieeexplore.ieee.org/document/9745131

[6] Task Scheduling and Resource Allocation in Cloud Computing using a Heuristic Approach,
https://doi.org/10.1186/s13677-018-0105-8

[7] Task Scheduling in Cloud Computing based on Meta-heuristics: Review, Taxonomy, Open Challenges, and Future Trends,
https://www.sciencedirect.com/science/article/abs/pii/S221065022100002X

[8] Enhanced Max-min Task Scheduling Algorithm in Cloud Computing,
https://www.academia.edu/download/34344253/min_max_algo.pdf

[9] Task scheduling mechanisms in cloud computing: A systematic review,
https://onlinelibrary.wiley.com/doi/full/10.1002/dac.4302

[10] A Priority based Job Scheduling Algorithm in Cloud Computing, https://www.researchgate.net/profile/Shamsollah-Ghanbari/publication/257726215_A_Priority_Based_Job_Scheduling_Algorithm_in_Cloud_Computing/links/5a5fa392a6fdcc21f4858947/A-Priority-Based-Job-Scheduling-Algorithm-in-Cloud-Computing.pdf

[11] Hybrid Job Scheduling Algorithm for Cloud Computing Environment, https://link.springer.com/chapter/10.1007/978-3-319-08156-4

Code:

Two main files of code for our algorithm are as follows.

Input file:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;
import java.util.regex.Pattern;

import org.cloudbus.cloudsim.network.datacenter.Switch;

public class Input {

static int example;
static int qosflag=0;

public static Integer[][] processorsvaluearray;
public static Integer[][] childvalues;
public static Integer[][] childcostvalues;
public static Integer[][] qosvales;
public static Integer[][] temporary;

static ArrayList<Integer[][]> templist= new ArrayList<Integer[][]>() ;
ArrayList<Integer[][]> templist2= new ArrayList<Integer[][]>() ;


static ArrayList<ArrayList<Integer>> processorsvalue = new ArrayList<ArrayList<Integer>>();
    static ArrayList<ArrayList<Integer>> childvalue = new ArrayList<ArrayList<Integer>>();
    static ArrayList<ArrayList<Integer>> childcostvalue = new ArrayList<ArrayList<Integer>>();
    static ArrayList<ArrayList<Integer>> qosvalue = new ArrayList<ArrayList<Integer>>();

    static int totaltask;
static int totalprocessors;


 Input(int examplenumber) {
    example=examplenumber;
   templist2= chooseexample(example);
   processorsvaluearray= templist2.get(0);
   childvalues= templist2.get(1);
   childcostvalues= templist2.get(2);
   qosvales= templist2.get(3);
   totaltask=templist2.get(0).length;
   temporary=  templist2.get(0);
```

```
  totalprocessors= temporary[0].length;
  }

public static void giveqos() {
givechildcost();

qosvalue = new ArrayList<>(qosvales.length);
      for (Integer[] index : qosvales)
      {
         ArrayList<Integer> currentSubList = new ArrayList<>(index.length);
         Collections.addAll(currentSubList, index);

         qosvalue.add(currentSubList);

      }
}

public static void givechildcost() {

childcostvalue = new ArrayList<>(childcostvalues.length);
      for (Integer[] index : childcostvalues)
      {
         ArrayList<Integer> currentSubList = new ArrayList<>(index.length);
         Collections.addAll(currentSubList, index);

         childcostvalue.add(currentSubList);

      }
}
  public static void giveprocessorcost()
   {
   processorsvalue = new ArrayList<>(processorsvaluearray.length);
      for (Integer[] index : processorsvaluearray)
      {
         ArrayList<Integer> currentSubList = new ArrayList<>(index.length);
         Collections.addAll(currentSubList, index);

         processorsvalue.add(currentSubList);

      }
   }
   public static void givechild()
   {
   childvalue = new ArrayList<>(childvalues.length);
      for (Integer[] index : childvalues)
      {
         ArrayList<Integer> currentSubList = new ArrayList<>(index.length);
         Collections.addAll(currentSubList, index);

         childvalue.add(currentSubList);
      }
   }
   public static ArrayList<ArrayList<Integer>> getcost() {
   giveprocessorcost();
   return processorsvalue;
   }
```

```
   public static ArrayList<ArrayList<Integer>> getchild() {
   givechild();
   return childvalue;
   }
   public static ArrayList<ArrayList<Integer>> getchildcost() {
   givechildcost();
   return childcostvalue;
   }
   public static ArrayList<ArrayList<Integer>> getqos() {
   giveqos();
   return qosvalue;
   }
   public static int gettotalprocessors() {
   return  totalprocessors;
   }
   public static int gettotaltasks() {
   return totaltask;
   }
   public static int getqosflag() {
   return qosflag;
   }
   public static ArrayList<Integer[][] > chooseexample(int option){
   Integer[][] processorsvaluearray1;
   Integer[][] childvalues1;
   Integer[][] childcostvalues1;
   Integer[][] qosvales1;



switch(option){
case 1:

processorsvaluearray1 = new Integer[][] { {18,29,11,9}, {12,24,11,16}, {31,35,33,24}, {22,43,30,35},
{40,27,29,32}, {21,22,34,40}, {31,19,20,34}, {25,29,41,33}, {18,35,38,16}, {23,51,19,28}, {28,30,10,25},
{44,45,46,47}  };
childvalues1= new Integer[][]{ {1,2,3,4}, {6}, {5}, {6}, {7}, {8,10}, {9,10}, {9,10}, {11}, {11}, {11}, {-1}  };
//processorsvaluearray1 = new Integer[][] { {21,7,16}, {18,12,20}, {13,8,17}, {5,11,14}, {12,13,10}, {13,16,9},
{7,15,11}, {11,13,19}, {13,19,18}, {14,16,9}  };
  //childvalues1=  new Integer[][]{ {7,8,9}, {3,4,5,6}, {2}, {1}, {0}, {7}, {9}, {9}, {9}, {-1}  };

  childcostvalues1= new Integer[][]{ {14,16,8,25} , {4}, {13}, {16}, {21}, {16,27}, {18,19}, {25,26}, {19}, {21},
{23} , {-1}    };
qosvales1= new Integer[][]{ {0}, {1}, {0}, {1}, {0}, {0}, {1}, {1}, {1}, {1}, {1}, {0} };

templist.add(processorsvaluearray1);
templist.add(childvalues1);
templist.add(childcostvalues1);
templist.add(qosvales1);

break;
case 2:

processorsvaluearray1 = new Integer[][] {{14, 18, 22 }, {15,19,10}, {27,29,21}, {35,12,27}, {40,11,10},
{15,23,16}, {27,19,30}, {35,41,49}, {40,13,20},  {11,44,27},  {15,19,17},  {21,23,28}, {31,33,39}, {24,27,20}   };
childvalues1=  new Integer[][] { {1,2,3,4,5}, {6}, {7}, {8}, {8,9}, {9}, {10}, {10}, {11}, {12}, {13}, {13}, {13}, {-
1}  };
```

```
childcostvalues1= new Integer[][]  { {10,24,20,15,10} , {12}, {15}, {21}, {20,11}, {12}, {22}, {20}, {10}, {18},
{20}, {24}, {16} , {-1}    };
qosvales1=  new Integer[][] { {0}, {0}, {0}, {2}, {1}, {0}, {1}, {0}, {1}, {0}, {1}, {0}, {2}, {1} };

templist.add(processorsvaluearray1);
templist.add(childvalues1);
templist.add(childcostvalues1);
templist.add(qosvales1);

break;
case 3:
processorsvaluearray1 =new Integer[][] { {14,25,42}, {30,20,13}, {37,15,20}, {17,25,30}, {15,32,20}, {19,21,12},
{23,44,20}, {41,47,45}, {37,29,11}, {14,13,15}, {42,44,25}, {12,24,30}, {24,25,26}, {12,14,15}, {41,40,32},
{44,40,33}, {30,16,20}, {26,18,25}, {16,30,31}, {49,40,50} };
childvalues1= new Integer[][]{ {1,2,3,4,5,6}, {7}, {8,9}, {9}, {10}, {11}, {11}, {12}, {12,13}, {13,14}, {14},
{15}, {16}, {17}, {17},{18}, {19}, {19}, {19}, {-1}  };
childcostvalues1=new Integer[][] { {11,13,15,19,12,25} , {10}, {18,14}, {20}, {21}, {14}, {13}, {20}, {21,11},
{26,17}, {27}, {19}, {11}, {17}, {23}, {21}, {20}, {28}, {26}, {-1}    };
qosvales1=new Integer[][] { {0}, {1}, {0}, {1}, {0}, {1}, {0}, {0}, {1}, {0}, {1}, {1}, {1}, {0}, {0}, {0}, {1}, {1},
{0}, {1} };

templist.add(processorsvaluearray1);
templist.add(childvalues1);
templist.add(childcostvalues1);
templist.add(qosvales1);

break;
case 4:

processorsvaluearray1 = new Integer[][] { {18,29,11}, {18,29,11}, {31,35,33}, {22,43,30}, {40,27,29}, {21,22,34},
{31,19,20}, {25,29,41}, {18,35,38}, {23,51,19}, {28,30,10}, {44,45,46} };
childvalues1= new Integer[][]{ {1,2,3,4}, {6}, {5}, {6}, {7}, {8,10}, {9,10}, {9,10}, {11}, {11}, {11}, {-1}  };
childcostvalues1= new Integer[][]{ {14,16,8,25} , {4}, {13}, {16}, {21}, {16,27}, {18,19}, {25,26}, {19}, {21},
{23} , {-1}    };
qosvales1= new Integer[][]{ {0}, {1}, {0}, {2}, {0}, {0}, {1}, {2}, {1}, {1}, {1}, {0} };

  templist.add(processorsvaluearray1);
templist.add(childvalues1);
templist.add(childcostvalues1);
templist.add(qosvales1);

break;
case 5:

processorsvaluearray1 =new Integer[][] { {14,25}, {30,20}, {37,15}, {17,25}, {15,32}, {19,21}, {23,44}, {41,47},
{37,29}, {14,13}, {42,44}, {12,24}, {24,25}, {12,14}, {41,40}, {44,40}, {30,16}, {26,18}, {16,30}, {49,40} };
childvalues1= new Integer[][]{ {1,2,3,4,5,6}, {7}, {8,9}, {9}, {10}, {11}, {11}, {12}, {12,13}, {13,14}, {14},
{15}, {16}, {17}, {17},{18}, {19}, {19}, {19}, {-1}  };
  childcostvalues1=new Integer[][] { {11,13,15,19,12,25} , {10}, {18,14}, {20}, {21}, {14}, {13}, {20}, {21,11},
{26,17}, {27}, {19}, {11}, {17}, {23}, {21}, {20}, {28}, {26}, {-1}    };
  qosvales1=new Integer[][] { {0}, {1}, {0}, {1}, {0}, {1}, {0}, {0}, {1}, {0}, {1}, {1}, {1}, {0}, {0}, {0}, {1},
{1}, {0}, {1} };

templist.add(processorsvaluearray1);
templist.add(childvalues1);
templist.add(childcostvalues1);
```

```
templist.add(qosvales1);

break;
case 6:
   processorsvaluearray1 =new Integer[][] { {12,24,15}, {20,40,23}, {27,25,20}, {19,27,60}, {35,32,20},
{15,21,18}, {23,44,20}, {41,47,45}, {37,29,11}, {14,13,15}, {42,44,25}, {12,24,30}, {24,25,26}, {12,14,15},
{41,40,32}, {44,40,33}, {30,16,20}, {26,18,25}, {16,30,31}, {49,40,50} };
   childvalues1= new Integer[][]{ {1,2,3,4,5,6}, {7}, {8,9}, {9}, {10}, {11}, {11}, {12}, {12,13}, {13,14}, {14},
{15}, {16}, {17}, {17},{18}, {19}, {19}, {19}, {-1}  };
childcostvalues1=new Integer[][] { {11,13,15,19,12,25} , {10}, {18,14}, {20}, {21}, {14}, {13}, {20}, {21,11},
{26,17}, {27}, {19}, {11}, {17}, {23}, {21}, {20}, {28}, {26}, {-1}    };
qosvales1=new Integer[][] { {0}, {1}, {0}, {1}, {0}, {1}, {0}, {0}, {1}, {0}, {1}, {1}, {1}, {0}, {0}, {0}, {1}, {1},
{0}, {1} };

templist.add(processorsvaluearray1);
templist.add(childvalues1);
templist.add(childcostvalues1);
templist.add(qosvales1);
break;
case 7:
processorsvaluearray1 = new Integer[][] { {14,16,9}, {13,19,18}, {11,13,19}, {13,8,17}, {12,13,10}, {13,16,9},
{7,15,11}, {5,11,14}, {18,12,20}, {21,7,16}};
childvalues1=  new Integer[][]{ {1,2,3,4,5}, {7,8}, {6}, {7,8}, {8}, {7}, {9}, {9}, {9}, {-1}  };
childcostvalues1=new Integer[][] { {18,12,9,11,14} , {19,16}, {23}, {27,23}, {13}, {15}, {17}, {11}, {13},  {-
1}    };
qosvales1= new Integer[][]{ {0}, {0}, {1}, {2}, {2}, {0}, {0}, {0}, {2},{1} };
templist.add(processorsvaluearray1);
templist.add(childvalues1);
templist.add(childcostvalues1);
templist.add(qosvales1);
break;
}

return templist;
   }

// new choose_case function to add new case
public static int choose_case(int choosencase){
Scanner scanner = new Scanner(System.in);

      // user will enter indexes and new value
      System.out.print("Enter the array to update (0 for processorsvaluearray1, 1 for childvalues1, 2 for
childcostvalues1, 3 for qosvales1): ");
      int arrayIndex = scanner.nextInt();
      System.out.print("Enter the row index to update: ");
      int rowIndex = scanner.nextInt();
      System.out.print("Enter the column index to update: ");
      int colIndex = scanner.nextInt();
      System.out.print("Enter the new value: ");
      int newValue = scanner.nextInt();
      System.out.println("choosen case number is " + choosencase);
      ArrayList<Integer[][]> valuesList = chooseexample(choosencase);

      Integer[][] processorsvaluearray1 = valuesList.get(0);
      Integer[][] childvalues1 = valuesList.get(1);
      Integer[][] childcostvalues1 = valuesList.get(2);
```

```
        Integer[][] qosvales1 = valuesList.get(3);

        // Value will be updated at specied index of specified array of specified case
        if (arrayIndex == 0) {
    processorsvaluearray1[rowIndex][colIndex] = newValue;
} else if (arrayIndex == 1) {
childvalues1[rowIndex][colIndex] = newValue;
} else if (arrayIndex == 2) {
    childcostvalues1[rowIndex][colIndex] = newValue;
} else if (arrayIndex == 3) {
    qosvales1[rowIndex][colIndex] = newValue;
} else {
    System.out.println("Invalid array index.");

}
        System.out.println("processorsvaluearray1 = new Integer[][] { " +
Arrays.deepToString(processorsvaluearray1).replace("[[", "{").replace("], [", "}, {").replace("]]", "}") + " };");
        System.out.println("childvalues1=  new Integer[][] { " + Arrays.deepToString(childvalues1).replace("[[",
"{").replace("], [", "}, {").replace("]]", "}") + " };");
        System.out.println("childcostvalues1= new Integer[][] { " +
Arrays.deepToString(childcostvalues1).replace("[[", "{").replace("], [", "}, {").replace("]]", "}") + " };");
        System.out.println("qosvales1= new Integer[][] { " + Arrays.deepToString(qosvales1).replace("[[",
"{").replace("], [", "}, {").replace("]]", "}") + " };");


        // Value will be updated at specied index of specified array of specified case
        try (BufferedReader br = new BufferedReader(new
FileReader("C:\\Users\\HP\\Downloads\\CloudSim3\\cloudsim3.0\\Thesis Codes\\Input.java"))) {

            // read the lines from the file
            List<String> lines = new ArrayList<>();
            String line;
            while ((line = br.readLine()) != null) {
                lines.add(line);
            }

            // identify the switch statement and its index
            Pattern switch_pattern = Pattern.compile("^\\s*switch\\s*\\(.*\\)\\s*\\{\\s*$");
            int switch_start_index = -1;
            int switch_end_index = -1;
            for (int i = 0; i < lines.size(); i++) {
                if (switch_pattern.matcher(lines.get(i)).matches()) {
                    switch_start_index = i;
                    break;
                }
            }
            if (switch_start_index == -1) {
                // the file doesn't have a switch statement
                System.out.println("Error: There is no switch statement in the file.");
           //     return;
            } else {
                for (int i = switch_start_index + 1; i < lines.size(); i++) {
                    if (lines.get(i).trim().equals("}")) {
                        switch_end_index = i;
                        break;
                    }
```

```
            }
        }

        // identify the last case in the switch statement
        Pattern case_pattern = Pattern.compile("^\\s*case.*$");
        int last_case_index = 0;
        int case_number_counter= 1;
        for (int i = switch_start_index + 1; i < switch_end_index; i++) {
            if (case_pattern.matcher(lines.get(i)).matches()) {
                last_case_index = i;
                case_number_counter = case_number_counter + 1;
            }
        }

        // find the index of the last break statement in the switch statement
        int last_break_index = -1;

        for (int i = last_case_index + 1; i < switch_end_index; i++) {
            if (lines.get(i).contains("break;")) {
                last_break_index = i;

            }
        }
    // append the new case after the last break statement
    String new_case = "case "+ case_number_counter + ":\n" +
                "  processorsvaluearray1 = new Integer[][]  { " +
Arrays.deepToString(processorsvaluearray1).replace("[[", "{").replace("], [", "}, {").replace("]]", "}") + " }; \n" +
                "  childvalues1 = new Integer[][] { " + Arrays.deepToString(childvalues1).replace("[[", "{").replace("], [",
"}, {").replace("]]", "}") + " }; \n" +
                "  childcostvalues1 = new Integer[][] { " + Arrays.deepToString(childcostvalues1).replace("[[",
"{").replace("], [", "}, {").replace("]]", "}") + " }; \n" +
                "  qosvales1 = new Integer[][] { " + Arrays.deepToString(qosvales1).replace("[[", "{").replace("], [", "},
{").replace("]]", "}") + " }; \n"  +
                "  templist.add(processorsvaluearray1);\n" +
                "  templist.add(childvalues1);\n" +
                "  templist.add(childcostvalues1);\n" +
                "  templist.add(qosvales1);\n" +
                "  break;\n";


    if (last_break_index == -1) {
        // the file doesn't have any break statements
        System.out.println("Error: There is no break statement in the switch statement.");
      //  return;
    } else {
        // add the new case after the last break statement
        lines.add(last_break_index + 1, new_case);
    }
    // write the updated file
    try (BufferedWriter bw = new BufferedWriter(new
FileWriter("C:\\Users\\HP\\Downloads\\CloudSim3\\cloudsim3.0\\Thesis Codes\\Input.java"))) {
        for (String updatedLine : lines) {
            bw.write(updatedLine);
            bw.newLine();
        }
    }
```

```
  } catch (IOException e) {
    e.printStackTrace();
  }

     System.out.println("Case number "+ choosencase +" has been executed and added as a new case after editing its
values");
return 0;
}

}
```

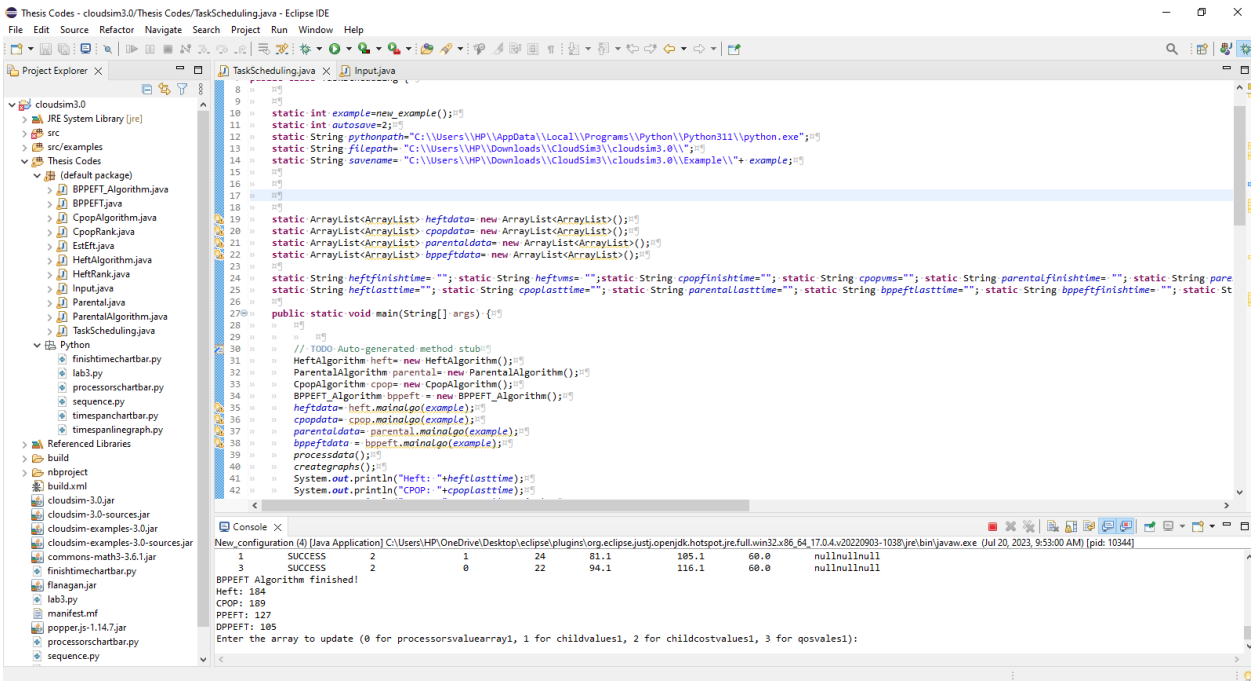
Task Scheduling file:

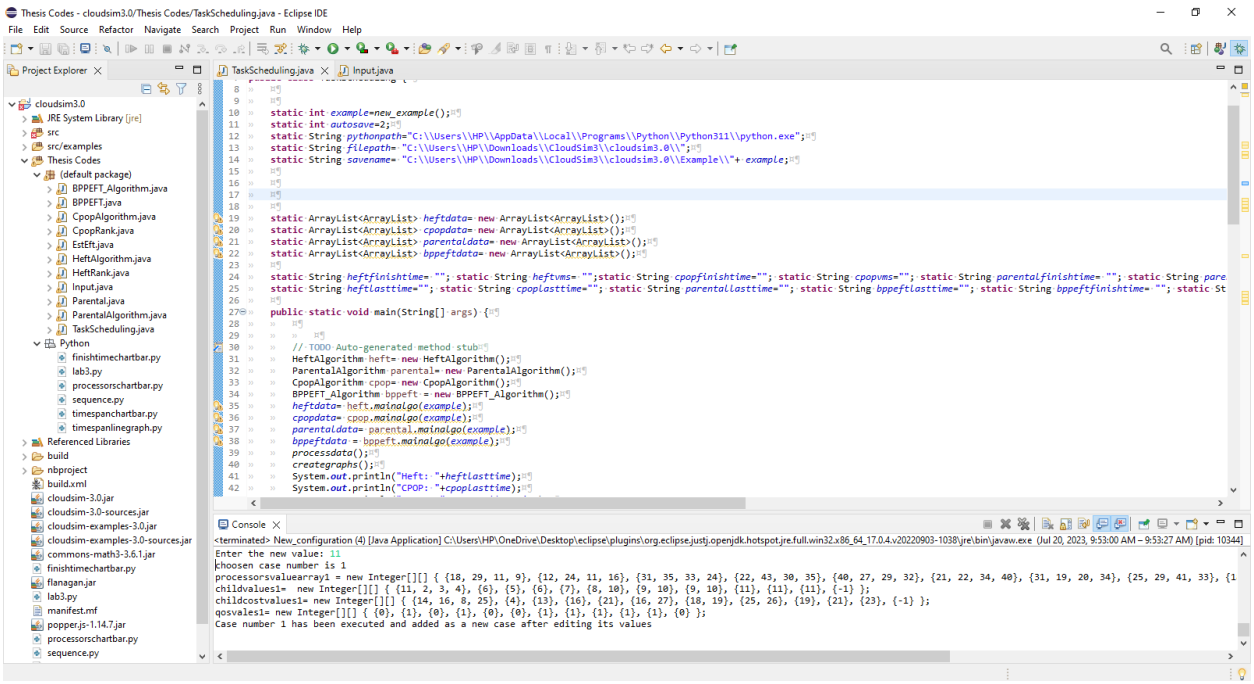
```
import java.util.ArrayList;
import java.util.Scanner;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class TaskScheduling {

static int example=new_example();
static int autosave=2;
static String pythonpath="C:\\Users\\HP\\AppData\\Local\\Programs\\Python\\Python311\\python.exe";
static String filepath= "C:\\Users\\HP\\Downloads\\CloudSim3\\cloudsim3.0\\";
static String savename= "C:\\Users\\HP\\Downloads\\CloudSim3\\cloudsim3.0\\Example\\"+ example;

static ArrayList<ArrayList> heftdata= new ArrayList<ArrayList>();
static ArrayList<ArrayList> cpopdata= new ArrayList<ArrayList>();
static ArrayList<ArrayList> parentaldata= new ArrayList<ArrayList>();
static ArrayList<ArrayList> bppeftdata= new ArrayList<ArrayList>();

static String heftfinishtime= ""; static String heftvms= "";static String cpopfinishtime=""; static String cpopvms="";
static String parentalfinishtime= ""; static String parentalvms= ""; static String graph1xaxis="";
static String heftlasttime=""; static String cpoplasttime=""; static String parentallasttime=""; static String
bppeftlasttime=""; static String bppeftfinishtime= ""; static String bppeftvms= "";

public static void main(String[] args) {

// TODO Auto-generated method stub
HeftAlgorithm heft= new HeftAlgorithm();
ParentalAlgorithm parental= new ParentalAlgorithm();
CpopAlgorithm cpop= new CpopAlgorithm();
BPPEFT_Algorithm bppeft = new BPPEFT_Algorithm();
heftdata= heft.mainalgo(example);
cpopdata= cpop.mainalgo(example);
parentaldata= parental.mainalgo(example);
bppeftdata = bppeft.mainalgo(example);
processdata();
creategraphs();
System.out.println("Heft: "+heftlasttime);
System.out.println("CPOP: "+cpoplasttime);
System.out.println("PPEFT: "+parentallasttime);
System.out.println("BPPEFT: "+bppeftlasttime);
      Input.choose_case(example);
}
```

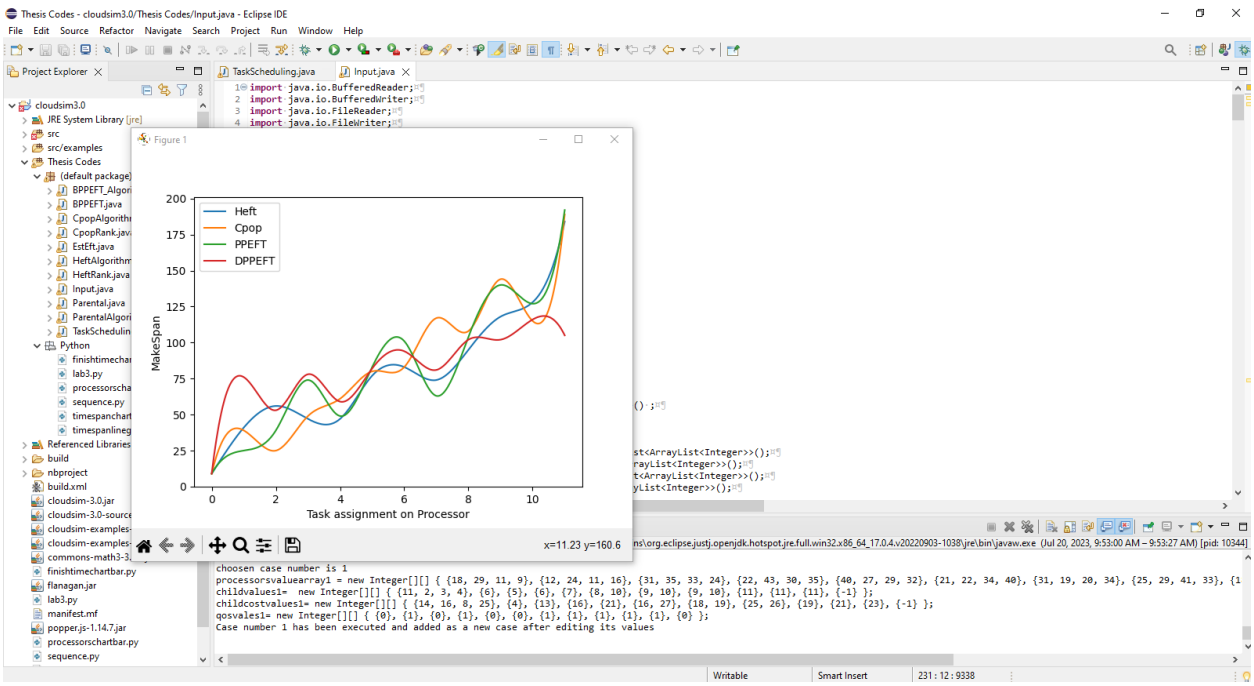
```
public static int new_example() {
Scanner scanner = new Scanner(System.in);
System.out.println("Enter case number to execute");
int example2 = scanner.nextInt();
return example2;
}
public static void creategraphs() {
try{
// First Graph
@SuppressWarnings("unused")
Process linegraphwithoutmarks= Runtime.getRuntime().exec(pythonpath+ " "+ filepath+"timespanlinegraph.py"+ "
" +graph1xaxis+" " +heftfinishtime+ " "+cpopfinishtime+" "  +parentalfinishtime+" "  +bppeftfinishtime+" "+
heftvms+" "+cpopvms+ " "+parentalvms+ " 0"+ " "+bppeftvms+ " 0"+ " "+autosave+"
"+savename+"timespanlinegraph");
Process finishtimechartbar= Runtime.getRuntime().exec(pythonpath+ " "+ filepath+"finishtimechartbar.py" +"
"+heftlasttime+ " "+cpoplasttime +" "+parentallasttime+ " "+bppeftlasttime+ " "+autosave+"
"+savename+"finishtimechartbar");
Process timespanchartbar= Runtime.getRuntime().exec(pythonpath+ " "+ filepath+"timespanchartbar.py"+ " "
+graph1xaxis+" " +heftfinishtime+" "+ cpopfinishtime+" "  +parentalfinishtime+ " "  +bppeftfinishtime+ "
"+autosave+" "+savename+"timespanchartbar");
Process processorchartbar= Runtime.getRuntime().exec(pythonpath+ " "+ filepath+"processorschartbar.py"+ " "
+heftfinishtime+ " "+cpopfinishtime+" "  +parentalfinishtime+" "  +bppeftfinishtime+" "+ heftvms+" "+cpopvms+ "
"+parentalvms+ " "+bppeftvms+ " "+autosave+" "+savename+"processorschartbar");
Process sequence= Runtime.getRuntime().exec(pythonpath+ " "+ filepath+"sequence.py"+ " " +heftfinishtime+" "+
heftvms+ " "+autosave+" "+savename+"sequence");
// BufferedReader in = new BufferedReader(new InputStreamReader(p.getInputStream()));
// int ret = new Integer(in.readLine()).intValue();
//String ret= in.readLine();
// System.out.println("Done");
}catch(Exception e){System.out.println(e);}
}

public static void processdata() {
heftlasttime= ""+ heftdata.get( heftdata.size()-1).get(0);
parentallasttime= ""+ parentaldata.get(parentaldata.size()-2).get(0);
cpoplasttime=""+ cpopdata.get(cpopdata.size()-1).get(0);
bppeftlasttime=""+ bppeftdata.get(bppeftdata.size()-1).get(0);
for(int i=0; i<heftdata.size(); i++) {

if(i!=0) {
heftvms+=",";
heftfinishtime+=",";
cpopvms+=",";
cpopfinishtime+=",";
parentalvms+=",";
parentalfinishtime+=",";
bppeftvms+=",";
bppeftfinishtime+=",";
graph1xaxis+=",";

}
graph1xaxis+= ""+i;
for(int j=0; j<heftdata.get(0).size();  j++) {

if(j==0) {
heftfinishtime+=""+(int)  heftdata.get(i).get(j);
```

```
parentalfinishtime+= ""+ (int) parentaldata.get(i).get(j);
cpopfinishtime+=""+ (int) cpopdata.get(i).get(j);
bppeftfinishtime+= ""+ (int) bppeftdata.get(i).get(j);
}
if(j==1) {
heftvms+=  ""+(int) heftdata.get(i).get(j);
parentalvms+= ""+parentaldata.get(i).get(j);
bppeftvms+= ""+bppeftdata.get(i).get(j);
cpopvms+= ""+cpopdata.get(i).get(j);

}
}
}
}
}
```

Code Snippet:




Thesis Codes - cloudsim3.0/Thesis Codes/TaskScheduling.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
TaskScheduling.java
Input.java
Console
<terminated> New_configuration (4) [Java Application] C:\Users\HP\OneDrive\Desktop\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (Jul 20, 2023, 9:53:00 AM – 9:53:27 AM) [pid: 10344]
Enter the new value: 11
choosen case number is 1
Case number 1 has been executed and added as a new case after editing its values


Thesis Codes - cloudsim3.0/Thesis Codes/Input.java - Eclipse IDE
Figure 1
Heft
Cpop
PPEFT
DPPEFT
MakeSpan
Task assignment on Processor
x=11.23 y=160.6
Writable
Smart Insert
231 : 12 : 9338

The End