Chapter 5

Hybrid Task Scheduling Algorithms in Cloud

5.1 Introduction

In the evolving landscape of cloud computing, task scheduling plays a pivotal role in optimizing resource utilization, minimizing costs, and ensuring timely task completion. This chapter explores the realm of Hybrid Algorithms for Task Scheduling in the cloud, with a specific focus on the integration of three innovative approaches: Hybrid Jaya Particle Swarm Optimization (Jaya-PSO), Improved Jaya Harmony Search (IJHS) and Merged CSO PSO(MCSO)

Hybrid algorithms, combining the strengths of different optimization techniques, have gained prominence for addressing the intricate challenges of cloud task scheduling. The Hybrid Jaya-PSO algorithm amalgamates the Jaya algorithm, known for its rapid convergence, with the Particle Swarm Optimization (PSO), a self-adaptive and collaborative optimization technique. This synergistic blend aims to harness the strengths of both algorithms, offering enhanced efficiency and effectiveness in cloud task scheduling.

Additionally, the Improved Jaya Harmony Search (IJHS) algorithm introduces a harmony search optimization process, guided by the insights obtained from the Jaya algorithm. This innovative integration seeks to achieve load balancing, minimize costs, and ensure timely task completion within a cloud environment. The collaborative nature of Harmony Search, coupled with the optimization prowess of Jaya, contributes to an algorithmic framework designed to excel in the complexities of cloud task scheduling.

Continuing further, it introduces a novel approach—merged CSO (MCSO)—aimed at leveraging the respective advantages of CSO and PSO to achieve superior results. The primary objective is to minimize both execution cost and runtime, leading to an optimized

cost mapping. The algorithm's steps are comprehensively outlined, and its implementation in the CloudSim simulator showcases remarkable improvements. Comparative analyses highlight the enhanced performance of the MCSO algorithm in terms of execution time and convergence when compared against individual PSO and CSO approaches.

Throughout this chapter, we will explore the conceptual foundations, algorithmic intricacies, and empirical evaluations of these hybrid approaches. Comparative analyses against established algorithms, experimental setups, and results will provide a comprehensive understanding of the capabilities of Hybrid Jaya-PSO, IJHS and MCSO in the domain of cloud task scheduling. This exploration aims to contribute valuable insights into the ongoing quest for efficient and robust task scheduling solutions in cloud computing environments.

5.2 Hybrid Jaya PSO Algorithm for Task Scheduling

In this chapter, we present an innovative Hybrid Jaya-Particle Swarm Optimization (PSO) algorithm, integrating the strengths of both Jaya and PSO to achieve enhanced performance. The proposed algorithm is subjected to a comprehensive evaluation, focusing on critical metrics such as execution cost and execution time. The comparative analysis showcases superior results when compared to established algorithms, including Genetic Algorithm (GA), PSO, Honey Bee, Cat Swarm Optimization (CSO), Ant Colony Optimization (ACO), and Jaya. This hybrid approach aims to capitalize on the distinct advantages of Jaya and PSO, providing a synergistic solution for optimizing workflow scheduling in cloud computing environments.

5.2.1 Problem Statement

In the context of workflow scheduling, the representation of a schedule is facilitated through a dependency matrix denoted as $D_{i,j}$, where x signifies the level of a task within the workflow, and y indicates the quantity of tasks available to be scheduled at that particular level. The design of a parser is instrumental in generating an output in the form of this D matrix. Our consideration involves three distinct types of costs: (i) Processor-to-processor communication cost, (ii) Execution cost of a task on a processor, and (iii) Data that necessitates communication between various tasks.

The problem at hand is succinctly framed as follows: "*To generate a workflow schedule that yields the least execution cost within the minimum running time, taking into account all three aforementioned costs.*"

To mathematically formalize this problem statement, the subsequent section introduces a fitness function, which serves as an analytical representation of the problem, featuring

equations that encapsulate the intricacies associated with processor communication, task execution, and data communication.

Fitness Function

A fitness function serves as a crucial measure to assess and optimize the performance of an algorithm. To ensure fairness and eliminate bias, the same fitness function has been employed across all algorithms. The function incorporates various costs, including communication cost, execution cost of each task on each processor, and the size of data that needs to be communicated between tasks.

Let's denote a schedule obtained in a particular iteration as M, where P represents the set of processors and T represents the set of tasks. The execution cost $C_{exe}(M_p)$ (Equation 5.1) is the cost associated with executing a specific task on the allotted processor according to M. Here, u_{tp} denotes the running cost for a task on a processor. Consequently, $C_{exe}(M_p)$ aggregates the running costs, providing an overall measure of execution cost.

The access cost $C_{ac}(M_p)$ (Equation 5.2) encompasses the cost of accessing a processor, which includes data transfer between two processors. Here, $comm_{M(t1),M(t2)}$ signifies the communication cost between processors for tasks $t1$ and $t2$ according to M, while $data_{t1t2}$ represents the size of data that needs to be transferred.

The total cost $C_{total}(M_p)$ (Equation 5.3) is the sum of the execution cost and access cost. The objective is to minimize both the maximum cost and the maximum running time required to obtain the optimal mapping, as depicted in the following equations.

$$C_{exe}(M_p) = \sum_{t} u_{tp} \qquad \forall M(t) = p \tag{5.1}$$

$$C_{ac}(M_p) = \sum_{t1 \in T} \sum_{t2 \in T} comm_{M(t1),M(t2)} data_{t1t2} \qquad \forall M(t1) = p \quad M(t2) \neq p \tag{5.2}$$

$$C_{total}(M_p) = C_{exe}(M_p) + C_{ac}(M_p) \tag{5.3}$$

$$FitnessFunction : Minimize(Cost(M)) + Minimise(T_{run}) \tag{5.4}$$

5.2.2 Proposed Methodology

In this section, we proposed a hybrid workflow scheduling algorithm in Cloud Computing environment based on Jaya and PSO.

Hybrid Algorithm based on PSO and Jaya

In this section, the proposed approach is shown. First, PSO is explained and then Jaya algorithm.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) was introduced by Kennedy and Eberhart in 1995 [21]. This self-adaptive algorithm draws inspiration from a swarm of particles searching for food. In PSO, each particle possesses its own position and velocity and retains knowledge of its best position ($pbest$). The global best position ($gbest$) is determined as the best position vector among all the particles. In each iteration, the particle's velocity and position are updated based on its $pbest$ and the $gbest$, as illustrated in Equations 5.5 and 5.6. Here, the position corresponds to the processor ID to which a task is assigned.

$$v_i^{k+1} = \omega v_i^k + c \times rand \times (pbest_i - x_i^k) + c \times rand \times (gbest - x_i^k) \quad (5.5)$$

$$x_i^{k+1} = x_i^k + v_i^{k+1} \quad (5.6)$$

where,

v_i^k velocity of particle i at iteration k

v_i^{k+1} velocity of particle i at iteration $k+1$

ω inertia weight

c acceleration coefficient

$rand$ random number between 0 and 1

$pbest_i$ best position of particle i

x_i^k current position of particle i at iteration k

$gbest$ position of best particle in a population

x_i^{k+1} current position of particle i at iteration $k+1$

There are some advantages and disadvantages of PSO. Advantages involve that PSO does not have any generic operator such as crossover and mutation in GA but at the same time it takes more time to converge. Its particles have an internal memory which keeps on updating in each iteration but it has only a one way shared mechanism as only $gbest$ is shared with the entire population.

Advantages and Disadvantages:

- PSO does not have any generic operator such as mutation and crossover in Genetic algorithm.
- PSO has some particles which have an internal memory and they keep on updating themselves according to the internal velocity.
- It is a one way shared mechanism as only $gbest$ is shared with the population for updation.
- It takes more time to converge as compared to GA.

Algorithm 6 Particle Swarm Optimization (PSO) for Task Scheduling

1: Initialize population of particles with random positions and velocities representing task allocations to virtual machines.
2: Initialize $pbest$ for each particle as its current task allocation.
3: Find the fitness value for each particle's current task allocation (e.g., based on fitness function Equation 5.4).
4: Set $gbest$ to the task allocation of the particle with the best fitness.
5: **while** stopping criteria not met **do**
6: **for** each particle **do**
7: Update velocity and task allocation using equations:

$$\begin{aligned} \text{velocity} &= \omega \times \text{velocity} \\ &\quad + c_1 \times \text{rand\_coeff} \times (\text{pbest} - \text{position}) \\ &\quad + c_2 \times \text{rand\_coeff} \times (\text{gbest} - \text{position}) \\ \text{position} &= \text{position} + \text{velocity} \end{aligned}$$

8: Find fitness value for the new task allocation.
9: **if** fitness value is better than current $pbest$ **then**
10: Update $pbest$ for the particle.
11: **if** fitness value is better than current $gbest$ **then**
12: Update $gbest$.
13: Return the $gbest$ as the solution.

Jaya Algorithm

Jaya is a parameterless algorithm that dynamically adjusts solutions, moving them closer to the best solution and away from the worst solution [77]. Derived from the Sanskrit word for "victory," Jaya strives for success in various scenarios. The algorithm utilizes a fitness function to evaluate both the best and worst solutions. Researchers have applied Jaya to solve optimization problems, including workflow scheduling in a cloud environment, as demonstrated in Chapter 3.

$$X'_{ij} = X_{ij} + |rand[0,1](best_j - X_{ij}) - rand[0,1](worst_j - X_{ij})| \tag{5.7}$$

where,

X_{ij} the value of i^{th} candidate in j^{th} iteration

$best_j$ the best candidate

$worst_j$ the worst candidate value in j^{th} iteration.

Unlike PSO, Jaya does not have any algorithm specific parameters. Also, both the $best$ and $worst$ solution is shared with entire population for updating its memory. Its victorious nature also make it more efficient in terms of convergence. Jaya has not been applied to

any real life problem which makes it interesting to see how it will behave if applied[59].
Advantages and Disadvantages:

- It does not need any algorithmic specific parameters as compared to other algorithms which require proper tuning of these parameters.
- Its victorious nature also make it more efficient in terms of convergence.
- The best and worst solution is shared in each iteration to entire population.

HybridJayaPSO Algorithm

In this section, the proposed algorithm is elucidated through a flowchart. Since Jaya lacks specific algorithmic parameters, it is widely applicable. In the basic Jaya algorithm, initialization is performed randomly. Here, with the incorporation of PSO, the results of PSO serve as input, contributing to a more effective reduction in the mapping's cost. Fig. 5.1 illustrates the proposed algorithm. Initially, PSO is executed to find $gbest$ due to its advantages over GA. After $n/2$ iterations, where n is the total number of iterations, the Jaya algorithm is employed to find the $best$ solution, yielding the most optimal outcome. Algorithm 3 provides a detailed explanation of the proposed algorithm.

Analysis of Algorithm: The proposed algorithm leverages both Jaya and PSO, facilitating the attainment of the most optimal solution early in the process. This approach reduces the time required to find the optimal solution. Individually, PSO and Jaya suffer from random initialization, which can delay convergence. However, in HybridJayaPSO, PSO is utilized for initialization, contributing to an overall reduction in processing time compared to other algorithms.

Algorithm 7 HybridJayaPSO Algorithm

1: Randomly initialize a set of n solutions.
2: Calculate its execution cost (fitness function).
3: **while** iteration $<$ $\frac{\text{max\_iteration}}{2}$ **do**
4: Call Algorithm 6 (Particle Swarm Optimization).
5: Create a new set of n solutions using $gbest$ as the best solution to initialize for the next step.
6: **while** iteration $<$ max_iteration **do**
7: Call Algorithm 2.

5.2.3 Implementation and Results

This section presents us with the experimental setup and simulation tool used, alongwith, the results obtained on comparing it with various other meta heuristics approach.

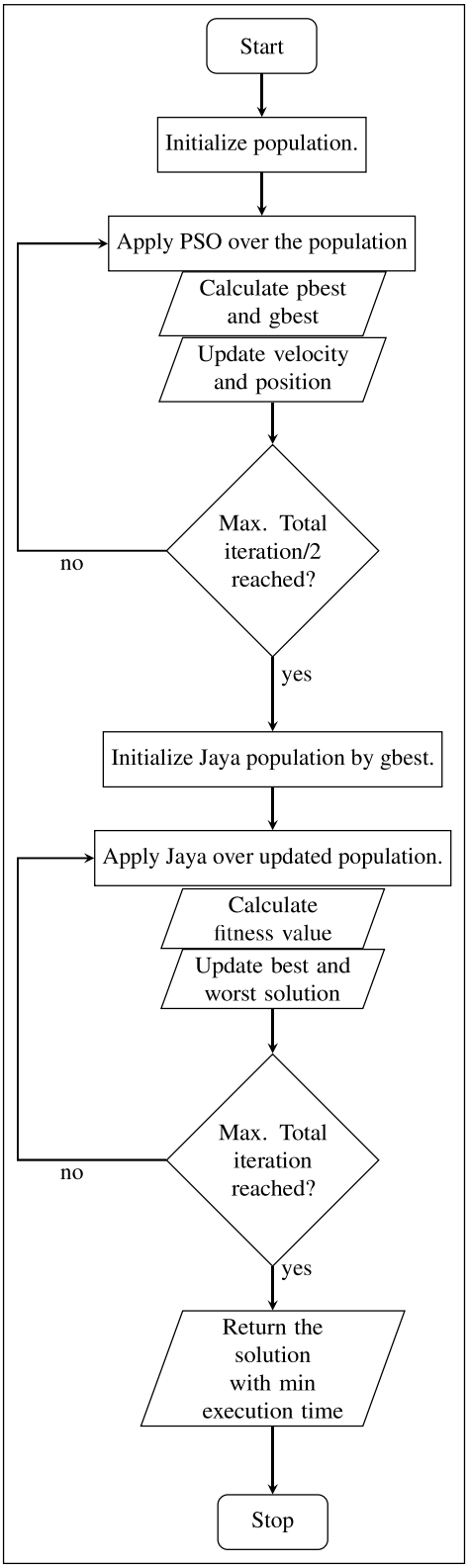

Fig. 5.1 Flowchart of HybridJayaPSO

Parameter	Value
Number of VMs	3-20
Number of datacenter	1
Transmission cost per byte	0.01$
Processing cost	3.00$
VM RAM	128, 512 and 1024 MB
File size	100-300 MB
Datacenter OS	LINUX
Number of cloudlets	25-1000
Number of CPU	1
Bandwidth	1000-2000 MB

Table 5.1 CloudSim parameters

Experimental Setup

The listed algorithms were executed on a personal computer running Windows 10 with 8GB RAM and a 64-bit processor. Java served as the programming language, with Eclipse as the chosen IDE (Integrated Development Environment), and the simulation tool CloudSim was utilized. The algorithms were implemented using a dependency matrix D and executed level by level. A set of ready tasks consisted of tasks available at a particular level, where previously dependent tasks had already been executed. A new parser was introduced to parse various workflows based on the tasks.

Table 5.1 provides details about the simulation setup parameters.

Data

Three types of matrices play a crucial role in calculating and defining the cost of a schedule:

i **TP Matrix (Task to Processor Cost Matrix):** This matrix represents the execution cost of a task on a processor.

ii **PP Matrix (Processor to Processor Matrix):** This matrix illustrates the transfer cost between two processors.

iii **Data Matrix:** This matrix provides information about the input and output data of tasks to be transferred to other dependent tasks.

The Pegasus Workflow Management System generates certain workflows, as depicted in Fig. 1.6. The parser we designed processes a dependency matrix level-wise, meaning that the next level is considered only after the previous level has been parsed.

Several matrices are used for cost determination, including TP (Task to Processor Cost) and PP (Processor to Processor Cost), which involves transferring 1 unit of data between

two processors. The number of tasks is varied to assess the algorithm's performance as the task count increases up to 1000.

The constants used in the algorithms have the following values:

- **PSO:**
  - Inertia (ω) = 1.2
  - Acceleration Coefficient (c) = 2.0
- **ACO:**
  - c = 1.0
  - α = 1
  - β = 5
  - Evaporation rate (ρ) = 0.5
  - Q (pheromone constant) = 500
  - Number of ants (m) = $no.of tasks \times numFactor$, where $numFactor$ = 0.8

Results

This section presents the results obtained from the comparison of various algorithms listed above. The evaluation criteria include the total time taken to obtain the schedule and the minimum cost of that schedule. All comparisons have been made by averaging the results to eliminate any biases and provide a clear view of the compared outcomes.

Cost

Execution cost is given top priority for evaluation, as explained in the fitness function. It considers running cost, communication cost, and the data that needs to be communicated. The graph illustrates the minimal cost obtained by an algorithm for the schedule it produces. The execution cost of the mapping obtained by the HybridJayaPSO algorithm is generally lower than that of other algorithms. Fig. 5.2 and 5.3 depict various comparisons based on cost, considering different numbers of tasks up to 1000.

Execution Time with Convergence of Algorithms

To determine the exact time complexity, we have measured the running time of the algorithms. This provides a clear view of which algorithm takes the minimum time for execution.

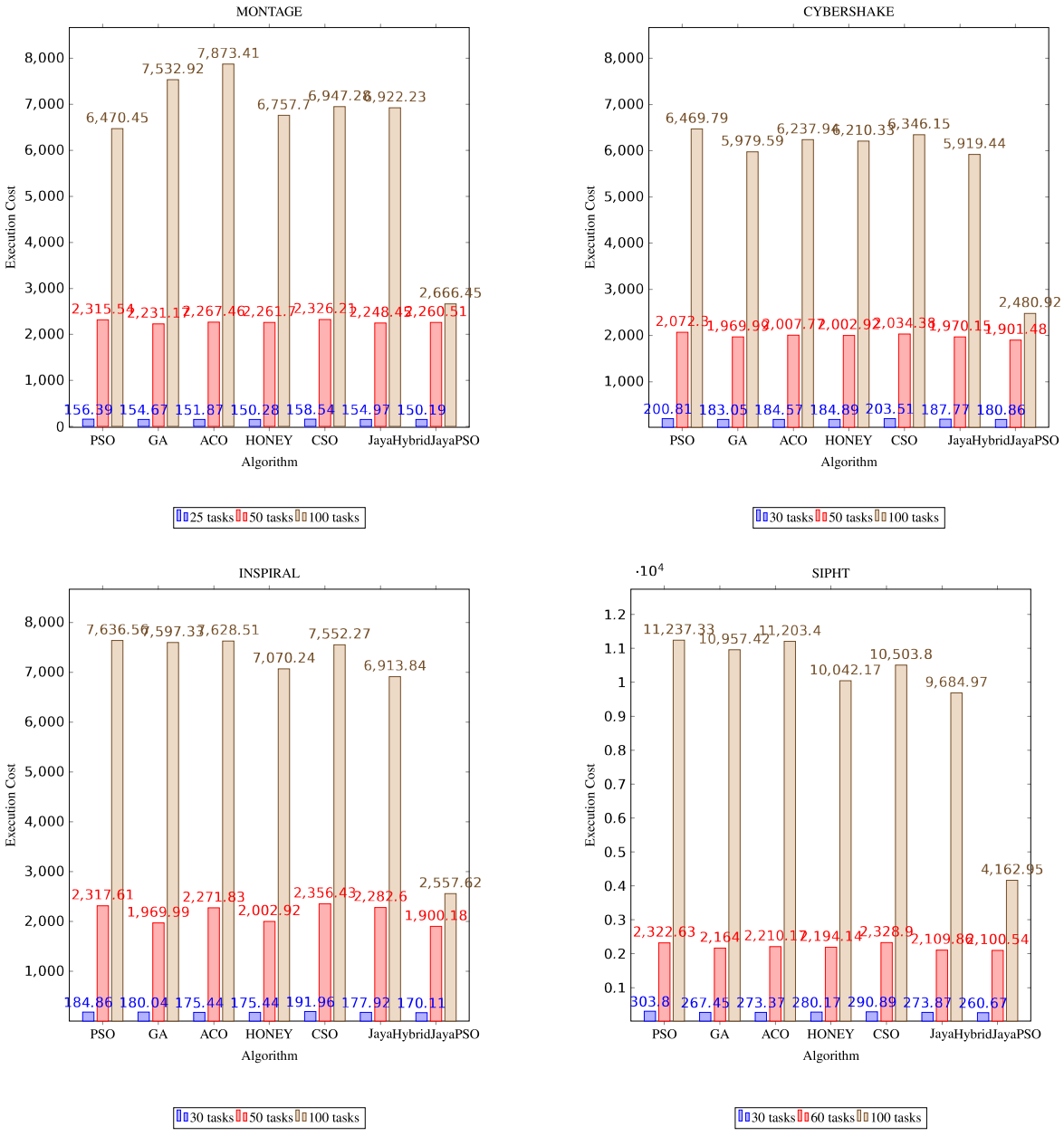

Fig. 5.2 Execution cost for various benchmark workflows

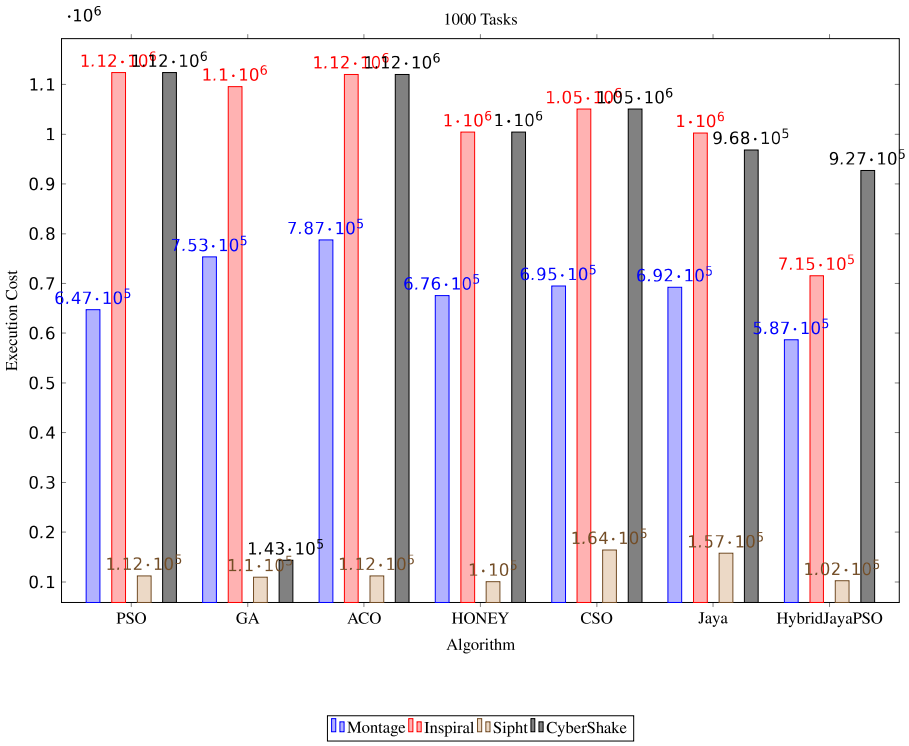

Fig. 5.3 Execution cost for various workflows with 1000 tasks

EXECUTION TIME	PSO	GA	ACO	HONEY	CSO	Jaya	HJayaPSO
Montage 25	558	2040	40032	34262	700	598	**540**
Montage 50	1198	3939	296452	142011	1201	2178	**1000**
Montage 100	1605	4676	10000.22	3775	1861	5172	**1860**
Montage 1000	160235	434676	103201.22	373275	138761	100172	**91890**
CyberShake 30	473	1717	67510	38952	538	5887	**490**
CyberShake 50	742	2554	286381	107059	775	14852	**658**
CyberShake 100	1300	4700	19223	3653	1441	51772	**1950**
CyberShake 1000	122612	570809	1789583	383653	144001	517072	**84620**
Inspiral 30	645	2479	66607	53986	11205	7904	**470**
Inspiral 50	1119	2555	2650	10259	1104	20085	**700**
Inspiral 100	10929	74530	23288	6782	2500	8091	**2030**
Inspiral 1000	192900	705325	2132288	611889	205900	840910	**10979**
Sipht 30	401	1428	67510	3538	479	4694	**580**
Sipht 60	719	2283	509958	12846	811	17112	**1720**
Sipht 100	1325	4261	125459	34628	1826	4979	**8792**
Sipht 1000	15850	231258	2499593	385628	152650	60777	**21351**

Table 5.2 Running time of the algorithms

Table 5.2 offers a detailed overview of the time taken by various algorithms for different workflows such as Sipht, Inspiral, CyberShake, etc., for various numbers of tasks. Fig. 5.4 illustrates the convergence of the algorithms, indicating that the proposed algorithm converges rapidly and produces better results in minimal time. The results presented in Fig. 5.2 demonstrate that the proposed technique ensures the minimum execution cost for all benchmark problems for workflows up to 100 tasks. Additionally, Fig. 5.3 provides insights into the execution cost for various benchmark workflows with 1000 tasks. The Jaya algorithm utilizes both the best and worst candidates to compute a new solution candidate, leading to the improvement of the worst candidate with each iteration. Consequently, the Jaya algorithm converges rapidly, producing superior results. When combined with PSO, the execution cost of the resulting mapping is further reduced, enhancing the overall outcomes. Furthermore, Table 5.2 compares the total time taken by the algorithm to produce the best results. The convergence analysis depicted in Fig. 5.4 clearly illustrates that GA, PSO, CSO, ACO, etc., take more time to converge compared to the proposed algorithm.

5.3 An Improved Hybrid Jaya Harmony Search Algorithm for Task Scheduling

In this section, we introduce the Improved Jaya Harmony Search Algorithm (IJHS), a hybrid approach combining the Jaya algorithm with Harmony Search (HS) to address the task scheduling problem in cloud computing. The IJHS algorithm leverages the Jaya

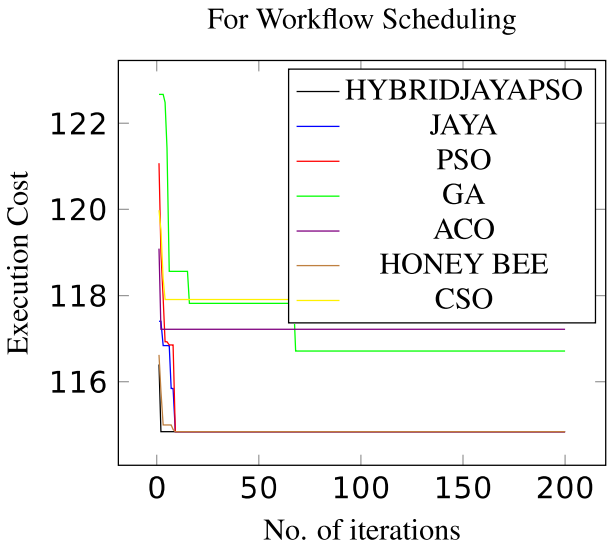

Fig. 5.4 Convergence of algorithms

algorithm to determine optimal parameters for the Harmony Search phase. Following parameter optimization, tasks are assigned to Virtual Machines (VMs) using Harmony Search, with a focus on load balancing by redistributing tasks from overloaded VMs to underloaded ones. The proposed approach considers both dependent and independent tasks, striving to achieve load balancing, minimize costs, and ensure timely completion of tasks. Through extensive experiments, we demonstrate that the IJHS algorithm surpasses existing task scheduling algorithms in terms of cost efficiency, execution time, and load balancing. Comparative evaluations are conducted against HS, Jaya, Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and First Come First Serve (FCFS) algorithms. The simulations involve diverse tasks and VM configurations within the CloudSim framework.

5.3.1 Problem Statement

Mathematical Formulation for Independent Tasks

In numerous multi-objective optimization challenges, the likelihood of discovering an optimal solution is often diminished. This challenge is particularly pronounced in vast search spaces like the Cloud, where the quest for an optimal solution becomes significantly challenging. The resolution of multi-objective problems typically involves the minimization or maximization of various components to yield an optimal solution [22], as illustrated in the equation below:

$$Optimal\_sol(n) = \{o_1(n), o_2(n), \ldots, o_m(n)\} \tag{5.8}$$

where, $o_1(n), o_2(n), \ldots, o_m(n)$ are various objective functions which needs to be minimized or maximized for finding an optimal solution.

Suppose, there is a heterogeneous set of Virtual Machines (VMs), where $V = \{v_j | m \geq j \geq 1\}$ is the set of VMs and m are the total number of machines. Also, there is a set of tasks T, where $T = \{t_k | n \geq k \geq 1\}$ and n are the total number of tasks. For each VM, v_j and task t_k, cost used is different, suppose c_{kj} represents cost of task k for j^{th} VM per time unit considered 'hour' in this paper.

Aim: The aim of the proposed algorithm is to map every task, t_k to a VM, V_j to minimise the cost and time taken for execution after shifting of the load. A schedule is said to be an assignment of Task to a VM according to an algorithm. Fitness Function is used in meta-heuristic algorithms to find an optimal solution. The fitness function used in our paper has been explained below.

Various costs are taken into consideration which are given as below. The execution cost for a task to execute on a processor according to the schedule M is calculated as follows:

$$Cost_{exec}(M) = \sum_{j \in V} proc_{tj} \forall t \in n \tag{5.9}$$

where, $proc_{tj}$ is the (task to processor) processing cost of executing a task t in j^{th} VM.

The total access cost of the schedule M is calculated as:

$$Cost_{access}(M) = \sum_{t \in T} trans_{tj} \times data_t \forall j \in m \tag{5.10}$$

where, $trans_{tj}$ is the transmission cost of a file by VM j allocated a task t and $data_t$ is the size of file to be communicated.

$Cost_{total}(M)$ is the total cost which needs to be minimised is calculated for schedule M as:

$$Cost_{total}(M) = Cost_{exec}(M) + Cost_{access}(M) \tag{5.11}$$

For time effectiveness of the schedule, a VM is determined if it is over or underloaded by following number of steps. Firstly, we calculate the Load of each VM by:

$$Load_i = \frac{N * cloudletLength}{VM_{mips}} \tag{5.12}$$

where i is the i^{th} VM, N is the number of task allocated to the VM, $cloudletLength$ is the length of tasks and VM_{mips} is MIPS rate of VM.

Then the $Proc\_Cap$ is calculated as:

$$Proc\_Cap = PE_n * PE_{MIPS} + VM_{BW} \tag{5.13}$$

We can estimate Processing Time using these two values as:

$$Time^i_{proc} = \frac{Load_i}{Proc\_Cap} \tag{5.14}$$

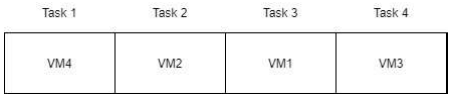

Fig. 5.5 Representation of a schedule

Calculate the overloaded and underloaded VM by using Standard Deviation as:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{5.15}$$

where, X_i is the Processing time as Calculated above, $\overline{x}$ is the Average Processing Time of VM. If any overloaded VM is found, then the task is allocated to under loaded VM[57]. After the load balancing of the VMs, the total execution time is calculated.
$Time_{exec}$ is the time taken by the last task to execute which is calculated as:

$$Time_{exec}(M) = Max\{FinishTime_k\} \forall k \in n \tag{5.16}$$

Fit_Func is taken to be multi-objective as a combination of execution cost and execution time:

$$Fit\_Func(F) = Minimum\Big(Minimise(Cost_{total}(M)) + Minimise(Time_{exec}(M))\Big) \tag{5.17}$$

where, M is the schedule which is represented as shown in Fig. 5.5 where $Task_1$ is scheduled to VM_4, $Task_2$ is scheduled to VM_2 and so on.

Mathematical Formulation for Dependent Tasks

The problem is to schedule a workflow on available resources in a cloud computing environment in a way that minimizes the makespan, cost, and load balancing.

Process

i **Task Decomposition:** The workflow is decomposed into a set of tasks, and the data dependencies among the tasks are identified. Each task is characterized by its processing time, input data size, output data size, and priority.

ii **Resource Selection:** The available resources in the cloud environment are identified, and the characteristics of each resource are collected, such as processing speed, memory capacity, and network bandwidth. Each resource is characterized by its availability, cost, and type.

iii **Load Balancing:** The load balancing of tasks is achieved by dividing the tasks into groups based on their priority, and allocating them to the available resources in a way that minimizes the load imbalance.

iv **Jaya Harmony Search Algorithm:** The Jaya Harmony Search algorithm is used to schedule the tasks on the available resources based on the task dependencies, resource characteristics, and load balancing. The Jaya Harmony Search algorithm is a metaheuristic algorithm that is used for optimization problems. It combines the benefits of Jaya algorithm and Harmony Search algorithm to improve the optimization process. The Jaya Harmony Search algorithm uses a harmony memory to store the best solutions found so far, and uses randomization to explore new solutions.

v **Cost Optimization:** The cost optimization of the workflow is achieved by minimizing the cost of resource allocation while ensuring that the makespan and load balancing constraints are satisfied. The cost of resource allocation is calculated based on the cost of each resource and the duration of resource usage.

vi **Makespan Optimization:** The makespan optimization of the workflow is achieved by minimizing the time required to complete all the tasks while ensuring that the load balancing and cost constraints are satisfied. The makespan is calculated as the maximum time required to complete all the tasks.

5.3.2 Proposed Methodology

This section explains the proposed algorithm for both dependent and independent tasks, along with basic algorithms. Jaya and Harmony Search both are meta heuristics algorithms and give optimal results. The first section describes the Jaya Algorithm, and the next explains Harmony Search.

Methodology for Independent Tasks

Jaya Algorithm

Jaya algorithm has been developed by R. V Rao in 2016 [62]. It is a metaheuristic algorithm which helps in solving optimisation problems irrespective of the fact that they are constrained or unconstrained. It is different from other algorithms as it does not contain any parameters which make it easier to apply.

Working of Algorithm: Jaya algorithm starts with the initialisation of the Candidate solution randomly. Here, the candidate solution is the value of HS parameters which are $HMCR$ and PAR. After random initialisation is done, the $best$ and $worst$ candidate from the solutions are selected according to the $fitnessfunction$ i.e f as explained in the section above after providing the calculated $HMCR$ and PAR and determining the fitness of the schedule obtained. Then, a new set of solution is generated using equation 5.7 which is also explained in Chapter 3 as 3.5. If the new solution is better than

$Candidate\_Solution$ i.e. X_{ij}, then the X_{ij} is updated. The above steps are repeated until the maximum number of iterations is reached. The Jaya Algorithm has been written in Algorithm 2.

Harmony Search Algorithm

Harmony Search algorithm has been developed by Zong Woo Geem et al. in 2001 [24]. In Harmony Search algorithm, a solution is called Harmony. It has a Harmony Memory(HM) which stores all the harmonies taken as different schedules in our work. It is used to maximise or minimise the function to obtain the optimal solution. $HMCR$ is the probability of selecting a component from HM. PAR is defined as the probability of the component from HM for mutation. In GA, only one or two chromosomes are used for mutation whereas HS takes into consideration the full HM for mutation.

Working of the Algorithm: The algorithm starts with random initialisation of HM with various schedules. It takes some random number $rand_1$, and if it is less than $HMCR$ then a new harmony is generated by $x_{new} = lower + rand_1 \times (upper - lower)$, else one Harmony is randomly selected from HM. After this, the improvisations are done based on another random number $rand_2$ and PAR. If $rand_2 \geq PAR$, then improvisation is done according to the equation $x_{new} = x_{new} + bw \times (rand_2 - 0.5) \times (upper - lower)$ else no improvisation is done. In this, $lower$ and $upper$ are 0 and $maximumnumberofVMs$ respectively. After this, the fitness function is calculated and checked if new Harmony can be added to HM or not. The above steps are repeated until the end of stopping criteria.

Analysis of HS algorithm: It randomly generates harmonies such as $x_1, x_2 \ldots, x_{HMS}$ to form harmony memory (HM), whose size is HMS. Each harmony is composed of W different tone components which are the number of cloudlets in our algorithm, i.e., if the harmony number is $j | j = 1, 2, \ldots, HMS$, the harmony is represented as $x_j = (x_{j1}, x_{j2}, \ldots, x_{jW})$. The time complexity of HS is $O(T_{max} \times W)$ where $T_(max)$ is the maximum iterations in HS which is taken as 10000 for all algorithms.

IJHS-proposed algorithm

IJHS algorithm uses both Jaya and HS algorithms. HS algorithm has many parameters which need to be determined precisely for optimal results. In IJHS, we have used Jaya to determine HS's parameters, which includes both $HMCR$ and PAR. Jaya is a parameter-less algorithm, so it does not take much time to evaluate the parameters, hence not adding much to the complexity of the algorithm. Jaya and HS have been explained individually in the previous sections. We have taken $HMCR$ as $\{0.9, 1.0\}$ and PAR as $\{0.01, 0.1\}$. As, HS depends basically on these parameters, so we have taken average over several

Algorithm 8 Harmony Search: for Task Scheduling in proposed Algorithm

Input: $HMS, W : No.ofCloudlets\ max_{iterations}, f, HMCR, PAR$
Output: x_{ij}

procedure HS
  for $i = 1; i \leq HMS; i++$ **do**
    for $j = 1; i \leq W; i++$ **do**
      Randomly initialise x_{ij}
  while Termination not reached **do**
    for $j = 1; i \leq W; i++$ **do**
      if $rand_1(0,1) \leq HMCR$ **then**
        $x_{new} = lower + rand_1 \times (upper - lower)$
      if $rand_2(0,1) \geq PAR$ **then**
        $x_{new} = x_{new} + bw \times (rand_2 - 0.5) \times (upper - lower)$
      Calculate f for x_{new}.
      if $f(x_{new})$ is better than worst $f(x_{ij})$ **then**
        Add x_{new} to x_{ij}.
  Return x_{ij}

iterations and taken that as $HMCR$ and PAR. The complexity of the algorithm has been discussed in the next section along with an example to show how it works.

Methodology for Dependent Tasks

This section gives the algorithm for Dependent Tasks or workflow which uses both the Harmony Search and Jaya algorithm.
Input: A workflow consisting of a set of tasks with their processing times, input data sizes, output data sizes, and priorities A set of resources in the cloud environment with their processing speeds, memory capacities, network bandwidths, availabilities, costs, and types
Output: A schedule of tasks on the available resources that minimizes the makespan, cost, and load balancing of the workflow.

Algorithm based on IJHS for Workflow Scheduling

i Decompose the workflow into a set of tasks and identify the data dependencies among the tasks.

ii Collect the characteristics of each resource, such as processing speed, memory capacity, and network bandwidth.

iii Divide the tasks into groups based on their priority, and allocate them to the available resources in a way that minimizes the load imbalance.

Algorithm 9 Improved Jaya-Harmony Search(IJHS): for task scheduling in Cloud

Input: $max_{iterations}, f, P, V$
Output: x_{ij}

```
1:  procedure IJHS
2:      for i = 1; i ≤ P; i++ do
3:          for j = 1; i ≤ V; i++ do
4:              Randomly initialise X_ij for finding HMCR and PAR.
5:      while Stopping Criteria Not Reached do
6:          Call Algorithm 2.
7:      Initialise HMCR and PAR.
8:      while Stopping Criteria Not Reached do
9:          Call Algorithm 8.
10:     Return x_ij.
```

iv Initialize the Jaya Harmony Search algorithm parameters, such as the harmony memory size, pitch adjustment rate, and bandwidth.

v Set the initial solution by assigning the tasks randomly to the available resources.

vi Evaluate the initial solution by calculating the makespan, cost, and load balancing of the workflow.

vii Loop until a stopping criterion is met:

- Update the harmony memory with the best solutions found so far.
- Generate new solutions by using the harmony memory and randomization to explore new solutions.
- Evaluate the new solutions by calculating the makespan, cost, and load balancing of the workflow.
- Update the current solution by selecting the best solution from the current and new solutions.

viii Return the best solution found.

Explanation: The Jaya Harmony Search algorithm combines the Jaya algorithm and the Harmony Search algorithm to improve the optimization process. The Jaya algorithm is used to update the current solution by using a pitch adjustment rate and a randomization process. The Harmony Search algorithm is used to generate new solutions by using a harmony memory and a bandwidth parameter to control the randomness of the search.

Cost (in Rupees)	VM_0	VM_1	VM_2
Cloudlet 1	1.226	0.222	1.652
Cloudlet 2	1.996	0.723	1.202
Cloudlet 3	1.111	0.119	0.942
Cloudlet 4	0.269	0.763	1.335
Cloudlet 5	0.824	1.127	0.056
Cloudlet 6	0.442	0.3	0.969
Cloudlet 7	1.043	1.189	1.435
Cloudlet 8	0.418	0.829	0.271
Cloudlet 9	1.222	1.482	0.949
Cloudlet 10	1.569	1.032	1.366

Table 5.3 Sample data for cost of each task on each VM(Cloudlet=10 and VM=3)

The cost, makespan, and load balancing of the workflow are optimized by evaluating the solutions generated by the Jaya Harmony Search algorithm.

Analysis of IJHS

The time complexities of both the algorithms have been explained above individually(See Section 3.2.2 and Section 5.3.2). It includes the advantage of the HS algorithm's way of finding the optimal results and Jaya algorithm's nature of being parameter-less, which reduces its complexity. In our algorithm, Jaya algorithm has been used for the pre-processing step, which will help in giving the optimal parameters for HS algorithm which results in the complexity as $O\Big(P \times V\big(T_{max} \times W\big)\Big)$. For HS, it is $O\big(T_{max} \times W\big)$. So, the overall complexity of IJHS algorithm after substituting both the complexities in the Algorithm 3 becomes $O\Big(P \times V\big(T_{max} \times W\big)\Big)$ where P is the population size and V is the number of variables in each candidate solution which is Number of Cloudlets in our algorithm.

Working of Algorithm

Let us take an example for explanation of working of algorithm as shown in Table 5.3. After receiving requests, we need a Task Scheduling Algorithm which will finally allocate various tasks to Various VMs. We have used IJHS for this. In the first step of Algorithm 3, we randomly initialise $HMCR$ and PAR, suppose $HMCR = 0.9945599283176121$ and $PAR = 0.09500483279718877$.
Now, after this, we call Algorithm 1, which requires random initialise of $candidate\_solution$ in step 1 to 4 as shown below in Table 5.4: In step 5, Take the $best$ be 2.09, with $hmcr$ as 0.9248 and par as 0.028. Take the $worst$ from X_{ij} be 2.09, with $hmcr$ be 0.9889

values	x0	x1	x2
hmcr	0.924	0.979	0.988
par	0.028	0.046	0.037

Table 5.4 Random initialisation of X_{ij}

values	x0	x1	x2
hmcr	0.987	0.967	0.995
par	0.0607	0.071	0.072

Table 5.5 Values of $HMCR$ and PAR

and par as 0.03725.

Here, X_{ij} has been used for $HMCR$ and PAR, so the equation becomes: $hmcr = hmcr + r1 * (hmcr - best\_hmcr) - r2 * (hmcr - worst\_hmcr)$. where $r1 = 0.58$ and $r2 = 0.86$ in our example. Similarly, for par, this equation is used.

For $i = 0$,

In step 7, after applying the above equation, X_0 has $hmcr$ as 0.987 and par as 0.0607. Calculation is done for other solutions too which is taken as 2 in this example as shown in Table 5.5. Now, in step 9, new $fitness$ is better than previous then $best\_hmcr$ is updated to 0.976 and $worst\_hmcr$ is updated to 0.988. Similarly, $best\_par$ becomes 0.046.

Similarly, after all iterations are done, the $best\_hmcr$ and $best\_par$ are calculated as 0.9552 and 0.0347. This is returned as the initialisation of Algorithm 2 in step 12.

In algorithm 2, for steps 2 to 4, we have done random initialisation as shown below where W are the number of cloudlets as 10 and HM is also taken as 10 which is shown in Table 5.6: In step 6, for each cloudlet, we take a random number suppose for cloudlet 1, $rand_1 = 0.7747$, $HMCR = 0.9055$, $bw = 0$ and $PAR = 0.063$ which satisfies the condition in Step 7, which makes x_new as 2 and after Step 10 using above values, it remains to be 2.

VM Allocated	cloudlet 1	cloudlet 2	cloudlet 3	cloudlet 4	cloudlet 5	cloudlet 6	cloudlet 7	cloudlet 8	cloudlet 9	cloudlet 10
hm1	2	0	0	0	1	0	2	2	1	0
hm2	1	1	1	1	0	2	2	2	1	2
hm3	0	0	1	1	2	2	1	0	2	0
hm4	0	0	1	1	0	2	2	0	2	2
hm5	0	2	1	0	1	1	0	1	1	0
hm6	2	1	2	1	2	2	2	1	1	1
hm7	0	2	2	2	1	1	0	0	1	2
hm8	2	2	2	1	1	0	0	0	1	1
hm9	2	2	0	2	2	1	0	2	1	0
hm10	2	1	1	2	2	1	0	1	1	0

Table 5.6 Value of harmony memory after random initialisation

Cloudlet	1	2	3	4	5	6	7	8	9	10
VM_number	2	2	0	0	0	1	1	0	0	2

Table 5.7 New generated harmony

Cloudlet	1	2	3	4	5	6	7	8	9	10
VM_number	1	1	1	0	2	0	0	2	2	1

Table 5.8 Final output

New x generated after this improvisation is as shown in Table 5.7. After checking if this exist in previous harmony or not, we add it in the previous harmony.
Now, this is repeated for various iterations which leads to final best harmony output as shown in Table 5.8. This final output results in final allocation which is done by Data Center Broker in CloudSim as there is no overloading of VMs. This can be seen in the screenshot of the output in Fig. 5.6.

5.3.3 Implementation and Results

This section explains the simulation parameters, along with the results of the implementation. The results have been compared with HS, Jaya, GA, PSO and FCFS algorithms. All these algorithms have been run on the same platform to avoid any biasing and an average of 10 iterations is taken for best optimal results. It starts with a single objective of Cost and Makespan individually. The section ends by comparing IJHS algorithm with other algorithms for a multi-objective solution, i.e. Cost and Makespan. The effciency of algorithms has also been compared. The efficiency has been calculated as :

$$Efficiency\_of\_IJHS\_over\_other\_Algo = \left(\frac{Algorithm_{Cost} - IJHS_{Cost}}{IJHS_{Cost}}\right) \times 100 \tag{5.18}$$

Simulation Parameters

CloudSim [16] is used to evaluate results. The parameters taken have been presented in Table 5.9, which involve VM configurations, Cloudlet Configurations, and other configurations used to implement the algorithms.

Simulation Results by varying Cloudlets and VMs

The evaluation is done on cost, makespan and multi-objective (cost and makespan both). It has been evaluated by varying cloudlets and VMs. The results have been shown in the following Tables.
Table 5.10 shows various variants of HS algorithm with different parameters value as

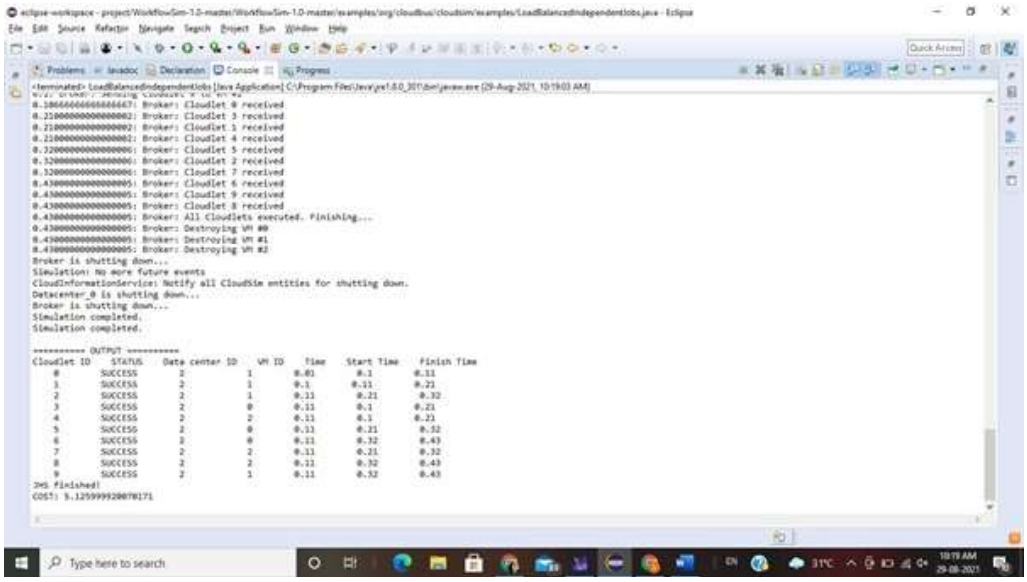

Fig. 5.6 Output of sample data after load balancing

Parameter	**Value**
Scheduling algorithms	FCFS, Jaya, Harmony search, PSO, GA, Improved Jaya–Harmony search
Number of VMs	3,5,10,15,20
Number of datacenter	1
Transmission cost per byte	0.01$
Processing cost	3.00$
VM RAM	128, 512 and 1024 MB
File size	100-300 MB
Datacenter OS	LINUX
Number of cloudlets	10,50,100,500,1000
Number of CPU	1
Bandwidth	1000-2000 MB

Table 5.9 CloudSim parameters

		Improvisations 10000	**HMS 10**			
	FCFS	HS				IJHS
		HMCR 0.9 *PAR* 0.1	*HMCR* 0.9 PAR 0.2	*HMCR* 0.8 PAR 0.1	*HMCR* 0.8 *PAR* 0.2	
100 cloudlets						
5 vms		56799	57359	55710	62529	40019
		51866	61169	60799	65980	42889
		49899	61549	64229	67989	39879
		50188	58559	59929	65169	41078
		52079	60819	62169	63559	40120
Average	**93670**	**52166.2**	**59891**	**60567.2**	**63560**	**40797**
50 cloudlets						
3 vms		29950	33130	31129	34000	29075
		29580	30630	28820	34620	28169
		30460	31750	31070	35020	29749
		29840	32100	32170	33870	29119
		30909	31899	31770	34040	25909
Average	**56620**	**30147.8**	**31901.8**	**31770**	**34310**	**28404**

Table 5.10 Variants of HS algorithm as compared to IJHS algorithm

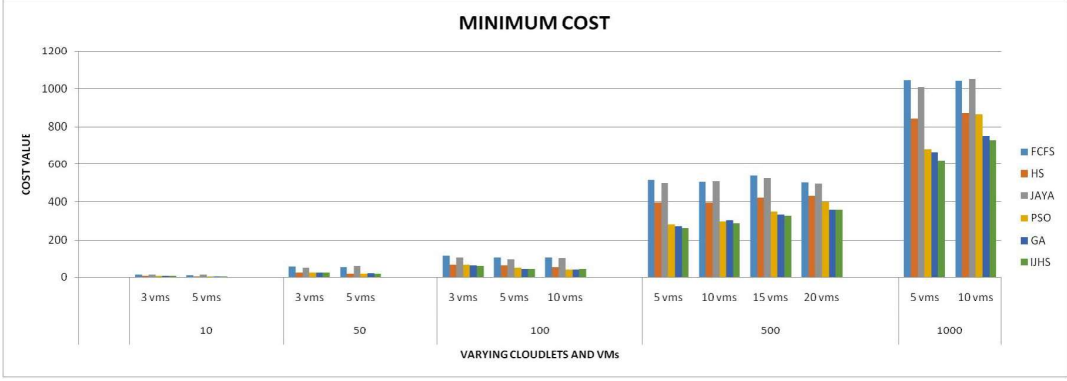

Fig. 5.7 Minimum cost by varying cloudlets and VMs

compared to IJHS algorithm. From the results, it can be seen that IJHS algorithm performs better than any variant of HS algorithm. Fig. 5.7 shows the cost results on various variations of Cloudlets and VMs. As the results show, IJHS algorithm executes with the least cost for all the tasks while variations of tasks and VMs are made. FCFS algorithm performs the worst. Jaya and Harmony algorithms have good results, but GA and PSO algorithms have the most comparable results with IJHS algorithm. Table 5.11 shows the exact values obtained.

As shown in Fig. 5.8,the makespan results reveal that IJHS algorithm has the least makespan for almost every schedule. GA algorithm performs comparably with IJHS algorithm, whereas FCFS algorithm performs the worst. PSO, Jaya and HS algorithms perform

		MINIMUM COST					
no_of_Tasks	no_of_Vms	FCFS	HS	JAYA	PSO	GA	**IJHS**
10	3 vms	11.55	4.91	11.28	4.91	4.91	**4.91**
	5 vms	10.03	3.39	12.661	3.45	3.45	**3.45**
50	3 vms	55.83	25.66	50.73	25.45	25.72	**25.43**
	5 vms	52.18	19.73	59.8	20.21	21.26	**19.91**
100	3 vms	112.06	64.46	102.44	65.22	61.24	**59.61**
	5 vms	104.36	61.12	92.07	48.75	43.39	**42.97**
	10 vms	104.12	53.28	98.93	41.2	41.23	**42.18**
500	5 vms	517.68	394.62	501.62	280.48	270.96	**260.29**
	10 vms	508.94	396.54	510.93	295.65	303.86	**285.27**
	15 vms	540.11	419.92	525.96	350.04	333.17	**326.49**
	20 vms	503.4	430.05	498.5	399.56	358.24	**359.83**
1000	5 vms	1045.79	841.26	1008.61	678.02	662.81	**621.73**
	10 vms	1043.3	869.47	1053.95	864.25	751.1	**724.88**

Table 5.11 Cost comparisons for various algorithms

worse than IJHS algorithm. Table 5.12 shows the average makespan values obtained. When both the objectives are combined, some trade-off can be seen. The minimum cost and minimum makespan have a trade-off compared to the individual objectives, whereas IJHS algorithm still shows the best results compared to other algorithms. When the tasks were less, the difference was less and when tasks increased the difference between cost and makespan results of various algorithms grew more, which can be seen in Table 5.13. The values have also been compared graphically as shown in Fig. 5.9. It can be said that IJHS performs the best with GA and PSO as second best algorithms, HS algorithm gets the third best with Jaya algorithm. FCFS behaves the worst scheduling algorithm.

The results show that the IJHS algorithm performs better as compared to other in terms of Cost and Makespan. However, a trade off is there due to pre-processing overhead added in IJHS because of initialisation of parameters by other algorithm called Jaya. This section describes the trade off between Cost and Execution Time. The graph shows that adding Jaya to Harmony Search gives much better results in terms of Cost. Although, the Execution Time it takes is more but as the number of tasks increase, there is a huge difference between the cost of IJHS and HS and Jaya(without pre processing) which can be clearly seen in Fig. 5.10. The difference in the cost of both algorithms is much significant

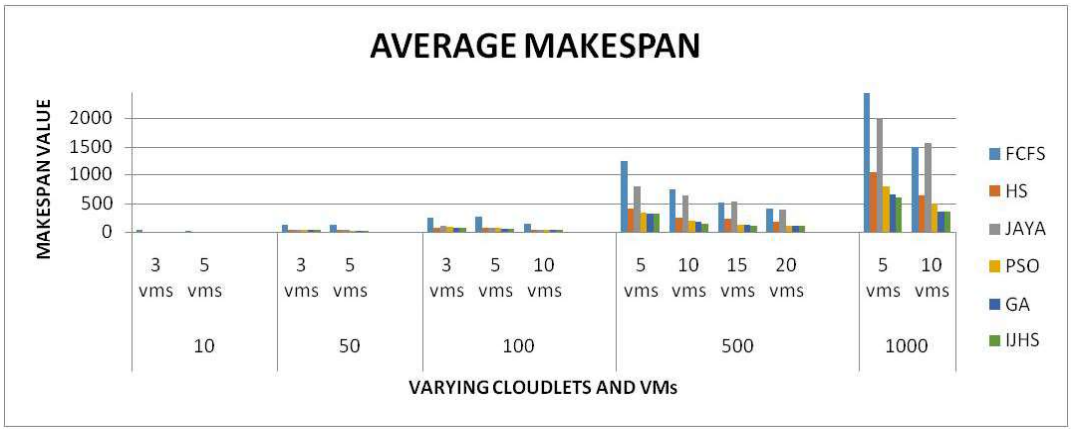

Fig. 5.8 Average makespan by varying cloudlets and VMs

AVERAGE MAKESPAN							
no_of_Tasks	no_of_Vms	FCFS	HS	JAYA	PSO	GA	IJHS
10	3 vms	32.3	9.03	11.23	8.76	8.76	**8.76**
	5 vms	26.01	6.87	9.34	6.87	7.64	**7.14**
50	3 vms	126.3	42.58	50.27	42.58	42.91	**42.72**
	5 vms	124.8	34.3	43.98	32.56	32.31	**32.97**
100	3 vms	241.96	82.54	110.58	92.56	80.63	**82.42**
	5 vms	263.39	69.48	75.25	69.12	65.39	**66.15**
	10 vms	147.17	39.29	52.1	38.45	36.04	**35.04**
500	5 vms	1239.44	410.46	800.75	333.15	328.38	**328.21**
	10 vms	742.59	257.96	645.65	198.52	179.76	**155.26**
	15 vms	518.34	240.38	528.31	138.45	135.54	**120.75**
	20 vms	402.48	174.67	400.27	110.85	106.86	**108.39**
1000	5 vms	2449.5	1043.99	2005.96	800.25	659.26	**602.04**
	10 vms	1490.79	643.65	1578.25	500.15	364.24	**360.96**

Table 5.12 Makespan comparisons for various algorithms

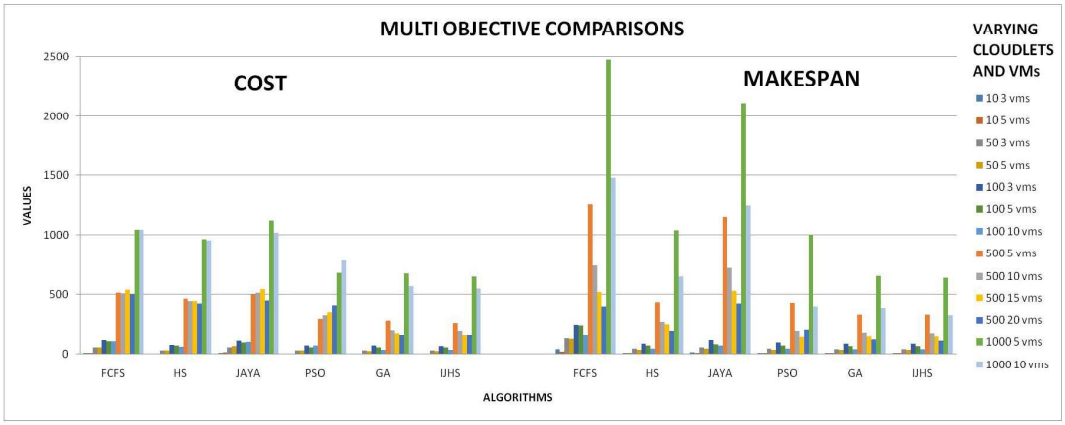

Fig. 5.9 Multi objective (cost and makespan) comparisons by varying cloudlets and VMs

		COST						MAKESPAN					
no_of_Tasks	no_of_Vms	FCFS	HS	JAYA	PSO	GA	**IJHS**	FCFS	HS	JAYA	PSO	GA	**IJHS**
10	3 vms	11.55	6.28	11.82	7.25	7.06	**6.98**	40.23	9.16	15.32	10.5	9.62	**8.65**
	5 vms	10.03	4.95	13.07	4.96	4.95	**4.95**	20.69	8.42	11.75	9.25	8.44	**7.96**
50	3 vms	55.83	28.86	52.12	30.14	29.31	**29.12**	129.81	42.08	54.7	42.58	40.69	**40.36**
	5 vms	52.18	27.71	62.12	28.45	25.98	**25.02**	123.21	34.35	45.12	35.85	33.05	**32.73**
100	3 vms	112.06	73.18	108.85	69.14	68.72	**66.49**	248.95	83.43	112.08	92.56	83.94	**82.99**
	5 vms	104.36	68.37	95.45	55.1	52.61	**53.1**	243.69	68.84	80.45	69.12	65.8	**65.16**
	10 vms	104.12	60.02	100.12	68.2	36.22	**34.2**	155.78	44.02	70.25	42.52	37.76	**38.88**
500	5 vms	517	463	502.45	295	283	**260**	1259	438	1154	430	332	**330**
	10 vms	508	445	513.25	325	204	**198**	747	273	726	199	183	**180**
	15 vms	540	446	545.25	350.04	179	**152**	522	253	531	140	142	**145**
	20 vms	504	428	450.89	409	160	**165**	403	198	425	208	119	**110**
1000	5 vms	1045	967	1125.5	690	683	**660**	2475	1041	2110	1005	661	**650**
	10 vms	1043	956	1022.12	789	570	**550**	1481	656	1246	400	384	**325**

Table 5.13 Multi-objective (cost and makespan) comparisons for various algorithms

that the difference in execution time can be handled.

The results show that IJHS is almost 10-20% better than PSO and almost 5-10% better than GA which can be clearly seen in the tables above. The results show that IJHS works better for task scheduling than HS and Jaya without any pre processing by almost 18-25%. FCFS performs worst which is 40-50% less efficient than IJHS.

5.4 HybridCSOPSO based Task Scheduling

Scheduling algorithms based on Swarm Intelligence have gained prominence. Each algorithm brings unique strengths and weaknesses to the table. This section introduces a novel approach—merged CSO (MCSO)—aimed at leveraging the respective advantages of CSO and PSO to achieve superior results. The primary objective is to minimize both execution cost and runtime, leading to an optimized cost mapping. The algorithm's steps are comprehensively outlined, and its implementation in the CloudSim simulator showcases remarkable improvements. Comparative analyses highlight the enhanced performance of the MCSO algorithm in terms of execution time and convergence when compared against individual PSO and CSO approaches.

5.4.1 Problem Statement

In the realm of cloud computing, the surge in user demands for a myriad of services necessitates effective task scheduling. This not only benefits Cloud Service Providers (CSPs) but also enhances the overall user experience. Task scheduling objectives in this context include minimizing execution costs, meeting deadlines, optimizing makespan, and managing energy consumption. While existing works have predominantly focused on makespan, the actual running time to attain an optimized schedule has been overlooked.

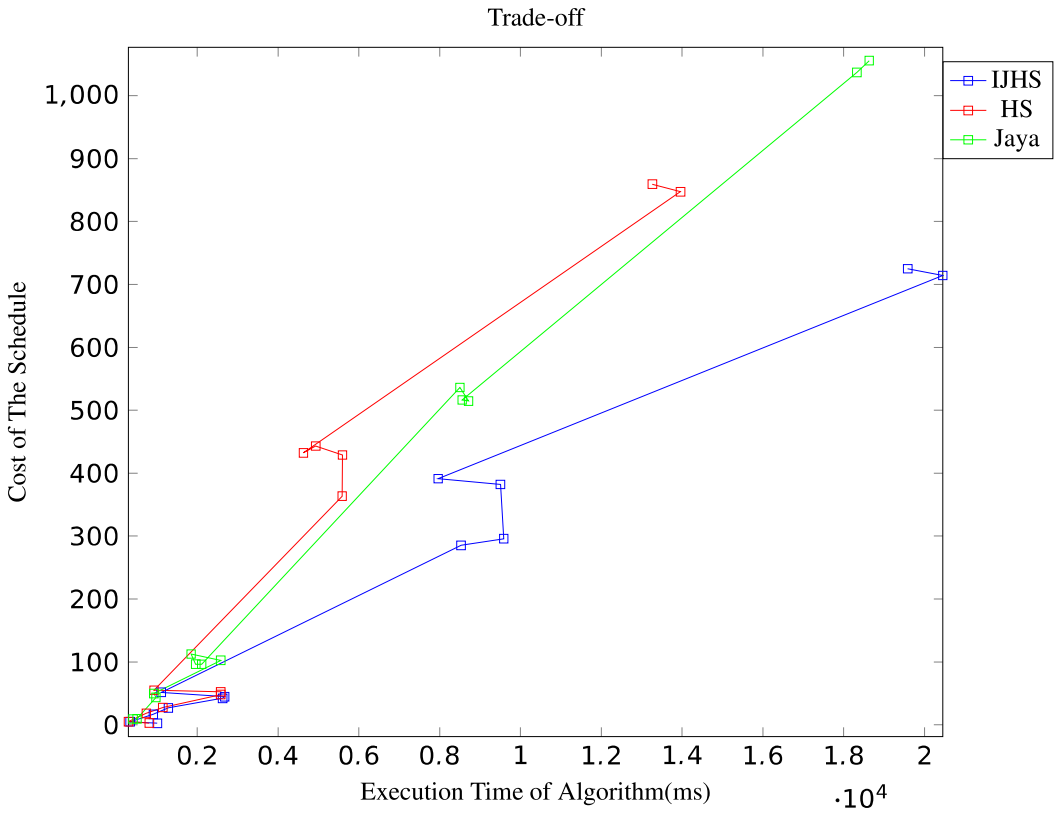

Fig. 5.10 Trade off between execution cost and time

The core problem can be articulated as follows: **"To determine a task-to-processor mapping that minimizes the total execution cost, taking into account both communication and execution costs. Additionally, the goal is to minimize the time required to reach this optimized mapping."** This problem encapsulates the multifaceted nature of cloud task scheduling, balancing cost efficiency and time optimization for enhanced performance.

Fitness Function

The fitness function incorporates parameters such as communication cost, execution cost of each task on each processor, and the file size of tasks to be communicated. The fitness function, as defined in [60], is utilized. Let M represent a mapping assigned in a specific iteration. P denotes the set of processors, and T is the set of tasks. $C_{exe}(M_p)$ signifies the execution cost of running tasks on the allocated processors according to M. u_{tp} represents the running cost of task t on processor p. $C_{ac}(M_p)$ denotes the access cost of communicating data between processors to which tasks t_1 and t_2 are assigned. $comm_{M(t1),M(t2)}$ represents the communication cost between the processors where tasks are mapped, and $data_{t1t2}$ is the file size to be communicated. $C_{total}(M_p)$ is the sum of access cost and execution cost. The fitness function is defined as the minimization of the maximum cost (Cost(M)) for each mapping along with the time taken to reach that mapping (T_{run}).

$$C_{exe}(M_p) = \sum_{t} u_{tp} \qquad \forall M(t) = p \tag{5.19}$$

$$C_{ac}(M_p) = \sum_{t1 \in T} \sum_{t2 \in T} comm_{M(t1),M(t2)} data_{t1t2} \qquad \forall M(t1) = p \quad M(t2) \neq p \tag{5.20}$$

$$C_{total}(M_p) = C_{exe}(M_p) + C_{ac}(M_p) \tag{5.21}$$

$$FitnessFunction : Minimize(Cost(M)) + Minimise(T_{run}) \tag{5.22}$$

5.4.2 Proposed Methodology

In this section, we outline the methodology for addressing the task scheduling problem in cloud computing using the Merged Cat Swarm Optimization (MCSO) algorithm. The objective is to minimize both execution cost and makespan by efficiently mapping tasks to processors. The algorithm is designed to leverage the advantages of both Particle Swarm Optimization (PSO) and Cat Swarm Optimization (CSO) to achieve improved results. The model shown in Fig 5.11 shows the approach in which the meta-heuristic optimization techniques are implemented.

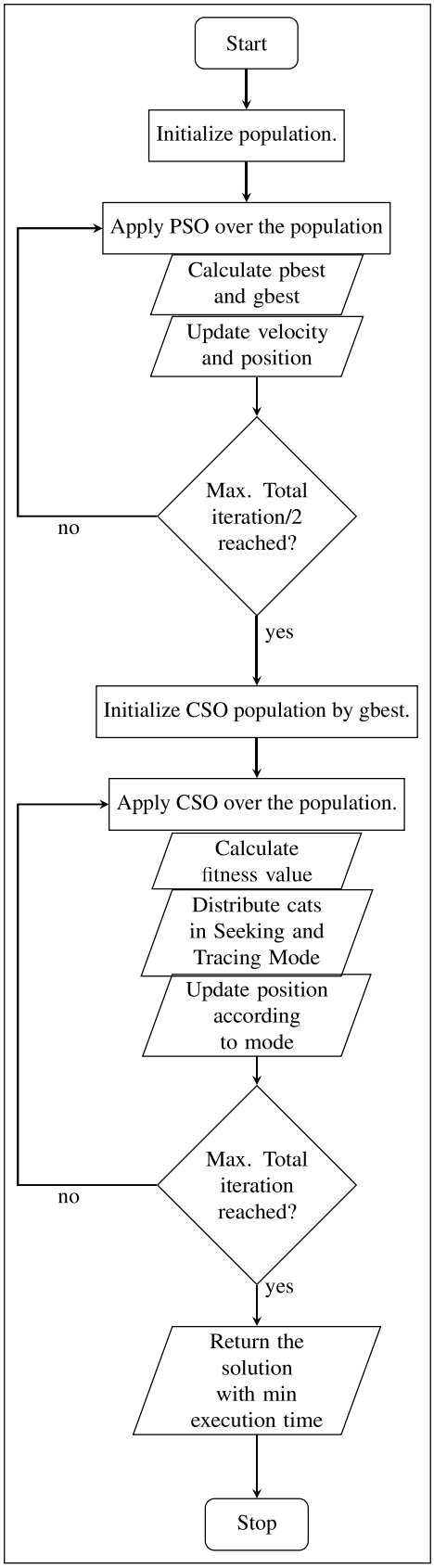

Fig. 5.11 Flowchart of HybridJayaPSO

Algorithm

This section explains both the algorithm and then shows the proposed approach.

Cat Swarm Optimization (CSO)

Cat Swarm Optimization (CSO) is an optimization algorithm rooted in swarm intelligence (SI), a concept borrowed from the behavior of social organisms in nature. In SI, self-organized individuals come together to form a society and collaborate towards a common goal. CSO draws inspiration from the behavior of a group of cats interacting with their environment. Observations from various surveys reveal that cats spend a significant portion of their time in a resting or sleeping state. Despite this seemingly relaxed state, cats exhibit high alertness and curiosity about their surroundings, optimizing resource utilization with minimal energy expenditure.

CSO divides its population of agents into two subgroups: seeking mode and tracing mode. A majority of the population is assigned to seeking mode, where cats actively explore their environment, resembling the movement of cats chasing prey. In contrast, a smaller population is designated to tracing mode, where cats rest but remain vigilant of their surroundings. The ratio between the two modes is controlled by the mixing ratio (MR), ensuring a dynamic balance between exploration and exploitation. This inherent division enhances the algorithm's adaptability to different optimization problems.

Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) draws inspiration from the collective behavior of swarms, such as fish or birds, searching for food. Proposed by Kennedy and Eberhart in 1995, PSO has seen numerous variations and adaptations since its inception. It operates as a clustering algorithm, where particles possess position and velocity as key features. The fundamental principle of PSO lies in the self-organization of particles to collaboratively solve optimization problems.

In PSO, each particle adjusts its position and velocity based on its own experience (personal best) and the collective knowledge of the swarm (global best). Through cooperation and communication, particles converge towards optimal solutions. The simplicity of PSO's mechanisms and its ability to handle various optimization problems make it a versatile and effective algorithm . The algorithm is described in Algorithm 6.

Merged CSO Algorithm: Objectives and Advantages

The primary objective of merging the Particle Swarm Optimization (PSO) and Cat Swarm Optimization (CSO) algorithms is to enhance the overall makespan execution time reduction and minimize total cost in task scheduling within the CloudSim simulation tool.

Advantages of PSO

PSO exhibits several merits:

- The equations in PSO are straightforward, facilitating easy calculation and implementation.
- PSO makes minimal assumptions about the optimization problem, making it applicable to a wide range of particles.
- As particle positions are updated based on the best solution within the swarm, PSO guides each particle and the entire swarm towards the best solution.
- PSO, being grounded in Swarm Intelligence (SI), is versatile and can be applied to both engineering and scientific research applications.

However, PSO also has certain demerits:

- It is susceptible to local optima, particularly in high-dimensional problems.
- PSO struggles when particles are widely scattered and need optimization.

Advantages of CSO

To address these shortcomings, CSO is executed for the remaining half of the iterations. CSO offers various advantages:

- CSO exhibits superior convergence time compared to other SI algorithms like PSO or PSO with a weighting factor (PSO-WF).
- CSO outperforms other algorithms in discovering global best solutions.
- CSO performs well in scenarios with a large number of scattered agents.

Nevertheless, CSO has its disadvantages:

- Due to its complexity, pure CSO may struggle to achieve an accurate solution.

The proposed merged CSO algorithm is designed to harness the strengths of both PSO and CSO while mitigating their respective weaknesses. This hybrid approach aims to achieve optimal task scheduling by leveraging the complementary features of the two algorithms.

Algorithm 10 Merged CSO (MCSO) Algorithm

1: Initialize the processor assignment using round-robin algorithm.
2: **while** iteration < max_iteration/2 **do**
3: Call Algorithm 6.
4: Initialize the population with the $gbest$ obtained from PSO.
5: **while** maximum iterations is not reached **do**
6: Randomly assign MR values to set cats into tracing and seeking mode according to MR.
7: Perform Algorithm 11 or 12 according to previous step.
8: Calculate the fitness value of the population and memorize the best fitness to the best cat.
9: Update the position of cats according to the mode it is in.

Seeking Mode

In the Seeking Mode of Cat Swarm Optimization (CSO), cats are in a resting state, attentively observing their surroundings. They maintain awareness of various situations and potential opportunities within the environment. When sensing danger or identifying a promising position for prey, a cat decides its next move.

This mode involves four key parameters:

- **CDC (Counts of Dimension to Change):** Represents the number of dimensions to change.
- **SMP (Seeking Memory Pool):** Defines the seeking memory pool for each cat.
- **SRD (Seeking Range of the Selected Dimension):** Specifies the seeking range for the selected dimension.
- **SPC (Self-Position Consideration):** A Boolean value determining whether the cat is a candidate for movement in each iteration.

The SPC value is either true or false, indicating whether a cat is a candidate for movement. The SRD is set to 2, and SMP values vary for each cat. If the SPC value is true, the SMP value for that cat is incremented by 1; otherwise, it remains unchanged. The CDC value is set to 1, indicating a change in only one dimension (X).

The Seeking Mode algorithm proceeds as follows:

Tracing Mode

Tracing Mode in the Cat Swarm Optimization (CSO) algorithm involves the cat being in motion, and its movement is determined by the position and velocity of its prey. The

Algorithm 11 Seeking Mode Algorithm

1: Build j duplicates of the i-th cat as prescribed by SMP.
2: Randomly update CDC.
3: Calculate the execution cost (fitness function) for each cat.
4: Record the best mapping with the best fitness.

velocity and position are updated using the following equations:

$$\text{velocity}_{it} = \omega \times \text{velocity}_{it-1} + c \times r \times (\text{xbest}_{it-1} - \text{position}_{it-1}) \tag{5.23}$$

$$\text{position}_{it} = \text{position}_{it-1} + \text{velocity}_{it} \tag{5.24}$$

where,

ω is the inertia coefficient,

r is a random value ($0 \leq r \leq 1$),

c is the acceleration coefficient,

$velocity_{it-1}$ is the previous velocity,

xbest is the best location.

The Tracing Mode algorithm proceeds as follows:

Algorithm 12 Tracing Mode Algorithm

1: Find velocity for the i-th cat using Equation 5.23.
2: Update the position vector using Equation 5.24.
3: Calculate the fitness function of the obtained mappings.
4: Update the solution with the best fitness value of the obtained mapping.

5.4.3 Implementation and Results

The proposed algorithm is implemented in a cloud environment using the CloudSim simulator, where cloudlets represent tasks and virtual machines act as processors for task scheduling.

Execution Cost

The execution cost is the total cost incurred for scheduling tasks on the corresponding processor, calculated using a specified formula. It depends on the cost of interaction between processors, the cost of interaction between tasks and processors, and the cost due to file size to be communicated. The execution cost is obtained by calculating the fitness function value using the TP (Task to Processor), PP (Processor to Processor), and data matrices. Fig. 5.12 shows the execution cost of various algorithms for 100 iterations. The algorithms are run 10 times to mitigate randomization errors, and the graph represents

Fig. 5.12 Execution cost in different runs

the results of all runs. MCSO exhibits a lower total cost than CSO and comparable cost with PSO. The population is initialized using the round-robin algorithm and passed to the PSO algorithm. The solution obtained from PSO is then fed as input for CSO. The results after n iterations of MCSO outperform both PSO and CSO, mainly because MCSO avoids random population initialization, as is the case with traditional CSO.

Makespan

Makespan in task scheduling is the total time taken for a set of tasks to complete execution. Fig. 5.13 compares the makespan for different scheduling algorithms. The proposed algorithm achieves the least makespan among the compared algorithms, demonstrating its efficiency in task scheduling. It converges faster and finds the optimal solution in fewer iterations.

In this section, three task scheduling algorithms were implemented and compared based on execution cost and makespan. The execution cost of the Merged CSO algorithm is comparable to PSO but smaller than CSO. Merged CSO achieves the least makespan, demonstrating superior efficiency as it utilizes the results of PSO as input positions and velocities. Additionally, the position for PSO is initialized using the round-robin method instead of random initialization, as in traditional PSO or CSO. Therefore, MCSO outperforms normal CSO or PSO by providing near-optimal solutions in a shorter amount of time.

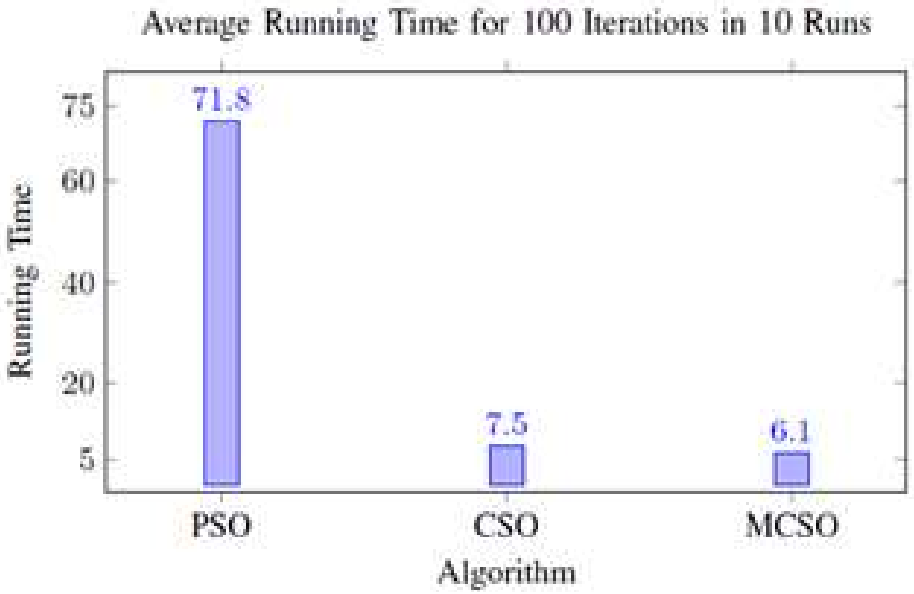

Fig. 5.13 Average running time of algorithms

5.5 Summary

Chapter 5 collectively explores the domain of hybrid algorithms for task scheduling in cloud environments. In Section 5.2, a Hybrid Jaya-Particle Swarm Optimization (PSO) algorithm was introduced to address the intricacies of task scheduling in the cloud. This hybrid approach seamlessly integrated Jaya and PSO, showcasing superior results in terms of execution cost and time when compared to established algorithms like Genetic Algorithm, Honey Bee, Cat Swarm Optimization, Ant Colony Optimization, and standalone Jaya.

Building upon this exploration, Section 5.3 proposed an Improved Jaya Harmony Search (IJHS) algorithm for task scheduling, utilizing Jaya for parameter optimization within Harmony Search. This approach aimed at achieving load balancing, cost minimization, and timely task completion. Comparative simulations against various algorithms, including Harmony Search, Genetic Algorithm, Particle Swarm Optimization, and First Come First Serve, demonstrated the efficacy of the proposed IJHS approach.

The last Section 5.4 introduced MCSO algorithm, which takes the benefits of PSO and CSO and merge them in the hybrid algorithm. The results demonstrate that MCSO works better than PSO and CSO in terms of execution cost and makespan.

Together, these sections contribute to the advancement of task scheduling in cloud environments by introducing hybrid algorithms that leverage the strengths of different opti-

mization techniques. The comparisons and evaluations highlight the superior performance of these hybrid approaches, providing valuable insights for optimizing task scheduling in cloud computing.