International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V73I7P101

Volume 73 Issue 7, 1-11, July 2025
© 2025 Seventh Sense Research Group®

Original Article

Job Scheduling and Optimization of Job-Worker Assignments in Distributed Computing Environments

Regis Donald HONTINFINDE[1*], Ariel AMOYEDJI[2], Mahugnon Geraud AZEHOUN-PAZOU[1], Marcos Thyrbus VITOULEY[1], Christian Djidjoho AKOWANOU[1]

[1]*Laboratory of Science, Engineering and Mathematics (LSIMA), National University of Science, Technology, Engineering and Mathematics (UNSTIM), Abomey, Republic of Benin.*
[2]*Woven by Toyota, Toyota Group, Tokyo, Japan.*

[*]*Corresponding Author : donald.hontinfinde@yahoo.com*

Received: 03 October 2024   Revised: 21 June 2025   Accepted: 30 June 2025   Published: 30 July 2025

Abstract - *A genetic algorithm is a metaheuristic inspired by the process of natural selection, which belongs to the large class of evolutionary algorithms. This makes it a good candidate for the development of new algorithms to solve optimization problems. In this study, we investigated the User-PC computing (UPC) system by proposing a genetic algorithm to assign jobs to workers to minimize the response time. To evaluate the effectiveness of the proposed genetic algorithm, we conducted experiments involving the execution of 72 jobs on the UPC system, using six worker PCs with varying numbers of threads and processor cores. We evaluated the algorithm in two distinct scenarios: static and dynamic job scheduling. In the static scheduling scenario, the algorithm assigns all available jobs to workers in a manner that minimizes the overall response time. The dynamic assignment scenario assigns newly arrived jobs to workers as they become available. The results demonstrate that the proposed genetic algorithm achieved a* 26.4% *reduction in response time compared to the random multiplestart local search (DRMSLS) algorithm. The results of this research have the potential to improve the performance of Next-Generation Networks (NGNs) in the telecommunications sector.*

Keywords - *Genetic Algorithm, Grid computing, Telecommunications networks, Thread job scheduling, UPC distributed computing.*

1. Introduction

The applications of grid computing are vast and varied. In the telecommunications sector, with the advent of Next-Generation Networks (NGN), grid computing appears as a promising technology in this context, potentially enabling telecom operators to manage their resources dynamically and optimally via a single platform. In response to the growing demand for cost-effective, high-performance computing solutions, this paper presents a distributed computing platform based on the master-worker model, referred to as the User-PC computing (UPC) system [1-7]. The UPC system is founded on a straightforward yet powerful concept: the master node collects and queues job requests from users, then strategically allocates these jobs to UPC workers, leveraging the idle computational resources available on the personal PCs of UPC members. The workers execute their assigned jobs, and the results are transmitted back to the master node, which coordinates the aggregation of the final output for the user [2-14]. A key performance metric for the UPC system is the makespan, or the total time required to complete all submitted jobs. Although the literature offers various task scheduling algorithms in distributed environments including greedy heuristics, round-robin, and custom heuristics, etc., the application of evolutionary approaches such as genetic algorithms remains largely unexplored in the specific context of User PC systems with quite heterogeneous workers in both static and dynamic cases. For instance, Kamoyedji et al. [2] suggested a static job-worker assignment algorithm designed for User-PC Computing (UPC) systems. Other studies, including [1, 3, 7], investigate different rule-based or cost-effective scheduling strategies, although they frequently use the assumption that worker capabilities are uniform or function in static batch settings. Although metaheuristic techniques, such as swarm optimization and genetic algorithms, have shown encouraging results in classical grid contexts [10, 12], their use is still uncommon in UPC environments, particularly those with dynamic work inflow and heterogeneous multi-core PCs. Unlike these previous methods, this paper introduces a novel job-worker assignment algorithm specifically designed for the UPC system. The algorithm builds upon prior research and is enhanced by a Genetic Algorithm framework, tailored to address the complexities of dynamic job scheduling. It accounts for crucial factors such as the number of threads each job utilizes during execution and the available CPU cores on each worker PC. This study examines two distinct scenarios: static job-worker assignment and dynamic job-worker assignment. In the static scenario, all jobs are known in advance and assigned to worker PCs with the primary goal of minimizing the overall makespan. In contrast, the dynamic scenario assumes real-time job arrivals; jobs are queued and dispatched to worker PCs as they become available after completing prior assignments. A series of experiments was conducted to assess the effectiveness of the proposed genetic algorithm-based scheduling approach. The experimental setup replicates that of Kamoyedji et al. [2], using 72 jobs and six worker PCs with varying thread and CPU core configurations. This replication ensures consistency and fairness when comparing both approaches. The rest of the paper is arranged as follows. Section II revisits the notion of a User-PC Computing (UPC) system, laying the groundwork for the suggested scheduling paradigm. Section III provides a mathematical description of the employment scheduling problem. Section IV presents the study's central concept: the genetic algorithm-based jobworker assignment technique. Section V presents and analyzes the experimental outcomes in both static and dynamic environments. Section VI closes the report by outlining potential future research directions.

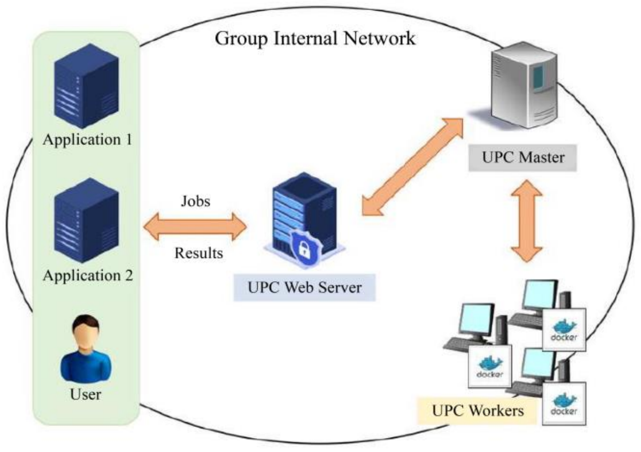

Fig. 1 Overview of the UPC system [17]

2. Review of User-PC System

The User-PC computing system (UPC) is a distributed computing platform that leverages the idle resources of Personal Computers (PCs) owned by a group of users to run computational jobs. Figure 1 provides an overview of the UPC system, which follows a master-worker architecture [17]. The system operates on a master-worker model, where the master PC manages various processes, including job scheduling, job transmission, job execution, result transmission, and result collection [1-3].

- Job Scheduling Job scheduling in the UPC system is a crucial process handled by the master PC. It involves receiving job processing requests from users and determining the most suitable worker PCs to run the jobs. Effective job scheduling algorithms consider factors such as job dependencies, CPU core utilization, and load balancing to allocate jobs efficiently. This ensures optimal resource utilization and minimizes the makespan, resulting in improved system performance.
- Job Transmission Once the job scheduling process is completed, the master PC transmits the necessary job information to the job's assigned worker PC. Job Transmission encompasses delivering all relevant information and instructions required for job execution. Efficient job transmission mechanisms ensure reliable and timely delivery of job data, enabling worker PCs to begin job execution promptly.
- Job Execution Worker PCs receive the job data from the master PC and execute their assigned jobs based on the provided instructions. Each worker PC utilizes its available computational resources, such as CPU cores and memory, to execute the jobs efficiently. Job execution may involve parallel processing, where multiple jobs are executed simultaneously to enhance overall performance.
- Result transmission Upon completion of the assigned jobs, worker PCs transmit the results back to the master PC. Result transmission entails sending the output data and any relevant information generated during job execution. Reliable and efficient result transmission ensures accurate and timely delivery of results to the master PC.
- Result Collection The master PC collects the results from the worker PCs and performs any necessary post-processing or aggregation jobs. Efficient result collection mechanisms enable accurate retrieval of job results and facilitate further analysis or presentation to the users who initiated the job processing requests.

This section provides an overview of related work in the literature. To the best of the authors' knowledge, no prior studies using genetic algorithms have explicitly considered the number of CPU cores and/or the number of job threads in the context of job scheduling.

Xu et al. [1] introduced the Deadline Preference Dispatch Scheduling (DPDS) algorithm, a dynamic scheduling method that prioritizes deadline constraints. They further developed the Improved Dispatch Constraint Scheduling (IDCS) algorithm, which integrates a risk prediction model aimed at minimizing resource wastage and maximizing job completion rates.

Kamoyedji et al. [2] proposed an O(n^3) algorithm for the dynamic allocation of jobs to worker PCs upon arrival, with the objective of minimizing the makespan in the UPC system. This algorithm is derived from a previously developed static job scheduling method based on the randomized multi-start local search technique. Although the algorithm considers

CPU core utilization, its efficacy diminishes as the number of jobs increases substantially.

Amalarathinam et al. [14] presented the Dual Objective Dynamic Scheduling Algorithm (DoDySA), which schedules jobs based on the Earliest Starting Time (EST) and Earliest Finishing Time (EFT) criteria. The algorithm aims to maximize CPU utilization while minimizing the makespan. Empirical results indicate that DoDySA outperforms existing alternatives in achieving these dual objectives.

Bhatia [4] provided a comprehensive review of job scheduling algorithms in the context of grid computing, categorizing them into heuristic and nature-inspired approaches. These algorithms primarily seek to minimize the execution time of individual jobs or to enhance the processing capacity of available computational resources. Recent research has enhanced evolutionary algorithms for difficult scheduling challenges.

For example, a 2021 study on college course scheduling developed a domain-specific local search operator and upgraded genetic operators (selection, crossover, mutation) to increase convergence speed and diversity [16]. Another 2024 work suggested a unique greedy-based seeding technique for the initial GA population, considerably enhancing solution quality and convergence time for the Travelling Salesman Problem [15].

While these techniques illustrate the adaptability and strength of genetic algorithms in a variety of situations, none of them solves the specific issues of work scheduling in User-PC computing systems with dynamically accessible, heterogeneous resources. This study attempted to overcome this gap by offering a GA-based scheduling mechanism that explicitly takes into account both thread-level task configuration and worker CPU core availability in both static and dynamic job arrival circumstances.

3. Job-Worker Assignment Problem Formulation

This section defines the mathematical symbols, constraints, and formal formulation of the job scheduling problem in the UPC system.

Let us consider the following:

- Set of jobs: $I = 1,2, \dots, n$, where n is the total number of available jobs.
- Set of workers: $U = 1,2, \dots, m$, where m is the total number of available workers.

The objective function $\mathcal{C}$, representing the overall performance metric of the job scheduling process, will be minimized.

3.1. Conditions

- Total number of assigned jobs: The total number of jobs assigned must always be less than or equal to the total number of available jobs, ensuring that all jobs are assigned and none are left unallocated.
- Total number of workers assigned: The total number of workers assigned must always be less than Equal to or greater than the total number of available workers, ensuring that the workload is distributed among the available resources.
- Job assignment uniqueness: Each job can only be assigned once, guaranteeing that jobs are not duplicated or assigned to multiple workers simultaneously.
- Resource requirements: The resource requirements of each job on any given worker PC must not exceed the usage limit specified by the user. This ensures that the assigned jobs can be executed within the available resources of the worker PCs.

3.2. Assumptions on Job-Worker Assignments

- Any worker can process one job at a time to avoid job swapping.
- Workers can have various numbers of CPU cores, and they may differ from each other in this regard.
- Any job can be assigned to any worker capable of processing it.
- All queuing jobs can be assigned to workers simultaneously.
- The arrival of future jobs is unpredictable and cannot be forecasted in advance.

3.3. Mathematical Formulation of the Assignment Problem

Let $\mathcal{T}(i,u)$ be a binary decision variable that equals 1 if task i is assigned to compute unit u, and 0 otherwise. This variable governs the assignment process in the distributed computing environment. The goal is to minimize the overall computational cost C, which reflects the total processing load distributed across all available units.

$$\mathcal{C} = \sum_{u\in\mathcal{V}} \sum_{i\in\mathcal{I}} \mathcal{T}(i,u) \cdot d_{i,u}$$

Where $d_{i,u}$ denotes the estimated execution time of task i on unit u.

Subject to the following constraints:

1. Total number of assigned tasks: $\sum_{i\in\mathcal{I}} \sum_{u\in\mathcal{V}} \mathcal{T}(i,u) = N$ Where N is the total number of tasks.
2. Uniqueness of assignment: $\sum_{u\in\mathcal{V}} \mathcal{T}(i,u) = 1 \forall i \in \mathcal{I}$
3. Resource constraint: For each task-unit pair, the resource demand must not exceed the unit's capacity. Let $R(i,u)$ be the resource demand of task i on unit u, and L_u the resource limit of unit u: $R(i,u) \leq L_u \forall i \in \mathcal{I}, \forall u \in \mathcal{V}$
4. Positive execution time: $d_{i,u} > 0 \ \forall i \in \mathcal{I}, \forall u \in \mathcal{V}$

4. Proposed Job-Worker Assignment Algorithm

This section introduces the proposed job-worker assignment algorithm designed for the UPC system. The algorithm seeks to optimize job-to-worker assignments to minimize the makespan and enhance overall system performance.

It consists of two main components: a greedy algorithm and a genetic algorithm. Before presenting the detailed implementation, an overview of the genetic algorithm is provided.

4.1. Genetic Algorithm Overview

A genetic algorithm is a metaheuristic optimization technique inspired by the process of natural selection and evolution. It operates by iteratively evolving a population of candidate solutions to a problem, mimicking the principles of survival of the fittest, crossover, and mutation. The genetic algorithm is based on the following steps:

- Initialization: The first step of the Genetic Algorithm is the population seeding phase [15]. The initialization of a population of candidate solutions. Each solution represents a potential queuing job assignment configuration. In this case, a greedy algorithm is used to generate six solutions for the initial population.
- Evaluation: Evaluate the fitness of each solution in the population based on the objective function, which measures the quality of the job assignments (the makespan). The fitness score reflects how well a solution performs in terms of the objective function.
- Selection: Select a subset of the fittest individuals from the population to serve as parents for the next generation. The selection process favors solutions with higher fitness scores, increasing their likelihood of being chosen as parents. Because the goal is to minimize the global makespan, only solutions with lower fitness scores are selected.
- Crossover: Apply crossover operators to the selected parents to generate offspring solutions. Crossover involves combining the genetic material (job assignments) from two or more parents to create new solutions. This process allows for the exploration of different combinations of job assignments.
- Mutation: Introduce random changes in the offspring solutions by applying mutation operators. Mutation alters certain elements (job assignments) within the solutions to introduce diversity and prevent premature convergence to suboptimal solutions.
- Evaluation: Evaluate the fitness of the new offspring solutions resulting from crossover and mutation.
- Replacement: Replace less fit individuals in the population with the improved offspring solutions, ensuring that the population continues to evolve towards better solutions. However, [16] shows that when the initial population is too large, the efficiency of the algorithm is reduced, and the computation time of the large algorithm execution increases greatly. On the other hand, if the initial population is too small, the diversity of the population is reduced, the sample capacity is reduced, and the overall performance of the algorithm becomes poor.
- Termination: Repeat steps 3-7 for a predetermined number of iterations (generations) or until the convergence criteria (the same offspring in over 100 generations) is reached.

By iteratively applying selection, crossover, and mutation operators, the genetic algorithm explores the search space, gradually improving the quality of solutions, and converging towards optimal or near-optimal job assignments.

4.2. Construction of the Initial Population Using a Greedy Algorithm

The development of the genetic algorithm begins with the construction of the initial population. A greedy algorithm is employed to generate the initial population of queued task assignments. This will generate a large set of usable solutions based on the workers' resource utilization. Here is the process flow:

1. Using predetermined thresholds, classify the queued workers and jobs. The classification ranges for queued workers and occupations are determined by these thresholds. The following thresholds were used in the implementation: 5, 8, 14, 17, 22, and 28. Employees and occupations that fall within the same threshold range are considered to belong to the same class.

2. For each worker class:

- The workers are ranked in ascending order based on CPU core utilization.
- Iterate through the tasks of the relevant class one by one:
    i. For each task, determine an appropriate value (relevance), such as the CPU core utilization rate among neighboring workers.
    ii. Sort the tasks in ascending order based on their relevance.
    iii. Considering the worker's available resources (disk space, RAM size), assign each task to the worker with the lowest cost.

3. Generate a (child) solution by allocating tasks to workers according to the results of the greedy algorithm. During task assignment, the greedy algorithm considers worker PC resource availability and CPU core utilization. It ensures that people and tasks with similar attributes are considered together by ranking them according to relevance, promoting a more efficient and balanced distribution of tasks.

4.3. Time Complexity

Time complexity is a measure of how the execution time of an algorithm grows as the size of the input increases. It helps us understand the efficiency of an algorithm in processing larger data sets.

Time complexity of the Greedy Algorithm Part: The time complexity of the greedy algorithm can be analyzed as follows:

- Dividing workers and jobs into classes based on the threshold: $O(n+m)$, where n is the number of workers, and m is the number of jobs.
- Sorting the workers for each job class based on CPU information: $O(m \log m)$.
- Calculating and assigning a heuristic value to each job: $O(m)$.
- Finding the most suitable worker for each job and assigning the job to the selected worker: $O(m \log n)$, where n is the number of workers. Overall, the time complexity of the greedy algorithm is $O(n+m+m\log m+m\log n)=O(m\log m+m\log n)$.

Time complexity of the Genetic Algorithm Part: The time complexity of the genetic algorithm can be analyzed as follows:

- Sorting the scores in ascending order: $O(n \log n)$, where n is the population size.
- Making selection: $O(n)$.
- Performing crossover and mutation: $O(n)$.

Overall, the time complexity of the genetic algorithm is $O(n \log n)$. As the greedy and genetic algorithms are executed sequentially, the overall time complexity is determined by the component with the higher computational cost, especially the greedy algorithm, which has a complexity of $O(m\log m + m\log n)$. Therefore, the global time complexity of the entire job scheduling process, combining both the greedy algorithm and the genetic algorithm, is $O(m\log m+m\log n)=O(m \log m\,n)$.

4.4. Dynamic Job Scheduling Algorithm Overview

The dynamic job-worker scheduling genetic algorithm repeatedly calls the previously proposed static genetic algorithm whenever jobs are still queuing and idle worker PCs are in the system. The procedure for managing dynamic job arrivals in the UPC system is the same as the one described in [2]. The pseudo-code of the dynamic job scheduling algorithm is described in Algorithm 1.

4.5. Comparison of Static vs Dynamic Algorithm

Table 1 summarizes the main differences between the static and dynamic scenarios addressed by the proposed Job Scheduling Genetic Algorithm.

5. Evaluation

5.1. Symbol

First, here are the variables used for the experimental evaluation.

$M/M/c$ is a multi-server queuing model;

```
Algorithm 1: Dynamic Job Scheduling Algorithm
  function GETIDLINGWORKERSET(workerSet)
    idlingWorkerSet ← ∅
    for each workerPC w in workerSet do
      if current worker $w$ is idling then
        Add the current worker to idlingWorkerSet.
      end if
    end for
    return idlingWorkerSet
  end function
  $U$, $I$, timer.initialize(), startTime ← 0,
  resultingJobWorkerMapping ← ∅, makespan = 0
  startTime ← timer.getCurrentTime()
  while true do
    idlingWorkerSet ← GetIdlingWorkerSet($U$)
    if idlingWorkerSet ≠ ∅ then
      resultingJobWorkerMapping ← Static Job
  Scheduling Method($U, I$)
      Assign jobs to idling worker PCs based on
  resultingJobWorkerMapping.
      Update the queue of waiting jobs $I$.
    end if
    if there are no new job arrivals then
      makespan = timer.getCurrentTime()-startTime
      return makespan
    end if
    Sleep for a short while
  end while
```

- U is a set of available worker PCs, and I is a set of queuing jobs to be scheduled.
- w is a worker PC characterized by an available CPU performance index, an available memory size, an available disk space and a CPU usage limit $\lim_w$ set by the user for the UPC system;
- j is a job characterized by some requirements in terms of CPU time, memory size and disk space;
- $\tau_{j,w}$ is the processing time associated with the processing of job j on worker PC w;
- λ is the average job arrival rate, and μ is the average job processing rate;
- $\sigma = \lambda/(|U|\mu)$ is the average utilization of each worker PC ($|U|$ is the size of the worker PCs)

5.2. Experimental Methodology

The genetic algorithm was implemented in Java and evaluated on a computer system composed of one master PC and six worker PCs, as shown in Table 1. Each chromosome encodes a task-to-worker assignment, where each gene represents a worker ID associated with a specific task. A population size of six solutions was used, with a one-point crossover probability of 0.6 and a mutation rate of 0.5 applied to the initial population.

Table 1. Overview of the differences between static and dynamic scheduling as implemented in a User-PC computing system

Aspect	Static Scenario	Dynamic Scenario
Number of jobs	Known beforehand	Arrive progressively
Job arrival behavior	All submitted at once	Follow a Poisson-like distribution
System behavior	Scheduler runs once, then stops	Scheduler stays active, waiting for new jobs
Assignment strategy	Global assignment using Greedy + Genetic Algorithm	File queuing + repeated use of the same static assignment logic
Scheduling frequency	Single-shot decision	Triggered by job arrival or job completion
Adaptability	No reallocation during execution	Jobs are dynamically matched to available workers
Main scheduling challenge	Achieving optimal balance in one pass	Minimizing makespan under uncertainty and fluctuating loads

To evaluate the genetic algorithm with an increasing number of jobs, each of the 24 jobs was executed once (= 24 jobs in total), twice (= 48 jobs in total), and three times (= 72 jobs in total). In this evaluation, jobs join the system in the same order as in Table 2.

As a performance index for evaluation, the makespan is calculated as the difference between the first job arrival time and the last job completion time. It is important to note that the experiments were repeated 50 times for 24 jobs, 100 times for 48 jobs, and 150 times for 72 jobs.

Table 2. PCs specifications

PC	# cores	CPU Type	clock Rate	Memory Size
Master	4	core i5	3.20 GHZ	8 GB
PC1	4	core i3	1.70 GHZ	2 GB
PC2	4	core i5	2.60 GHZ	2 GB
PC3	4	core i5	2.60 GHZ	2 GB
PC4	8	core i7	3.40 GHZ	4 GB
PC5	16	core i9	3.60 GHZ	8 GB
PC6	20	core i9	3.70 GHZ	8 GB

5.3. Results for Static Job-Worker Assignment

This section evaluates the results of the static job-worker assignment produced by the proposed algorithm, through comparisons with the reference algorithm (a method based on the randomized multi-start local search proposed by Kamoyedji et al. in [2]). The outcome of the evaluation provides valuable insight into the effectiveness of the proposed approach in optimizing static job assignments within the User-PC computing system.

Table 3. Jobs specifications

Job #	Job Name	# Threads	Disk Usage
1	Network Simulator	1	0.392 GB
2	Optimization Algorithm	1	1.5 GB
3	DCGAN	17	1.9 GB
4	RNN	17	1.9 GB
5	CNN	17	1.9 GB
6	FFmpeg	18	2.8 GB
7	Converter	1	1.1 GB
8	Palabos	2	6.7 GB
9	Flow	4	0.438 GB
10	Blockchain Mining	1	920 MB
11	COVID Detection	23	2.95 GB
12	COVID Outbreak Prediction	4	1.84 GB
13	Multimedia Content Resizing	18	2.8 GB
14	Multimedia Content Format Changing	18	2.8 GB
15	OpenFOAM 5W	1	1.4 GB
16	OpenFOAM 10 W	1	1.4 GB
17	OpenFOAM 15W	1	1.4 GB
18	OpenFOAM 20W	1	1.4 GB
19	OpenFOAM 25 W	1	1.4 GB
20	OpenFOAM 30 W	1	1.4 GB
21	OpenFOAM 35W	1	1.4 GB
22	OpenFOAM 40 W	1	1.4 GB
23	OpenFOAM 45 W	1	1.4 GB
24	OpenFOAM 50W	1	1.4 GB

Table 5 shows makespan results in hours:minutes:seconds (H:M:S) format for static job-worker assignments. It compares the outcomes of the genetic algorithm-based approach and the reference algorithm proposed by Kamoyedji et al. in [2], across three experimental settings: 24 jobs, 48 jobs, and 72 jobs. Table 4 and Figure 2 clearly show that the proposed method outperforms the reference algorithm in terms of makespan. The "Improvement" row quantifies the efficiency gains achieved by the proposed approach compared to the reference algorithm, both in terms of time saved (H: M: S) and as a percentage. This comparison underscores the potential for the genetic algorithm to significantly reduce the makespan and enhance computational efficiency across various job instances.

5.4. Results for Dynamic Job-Worker Assignment

The following section presents an evaluation of the dynamic job-worker assignment results produced by the proposed algorithm, through comparison with a reference method based on the randomized multi-start local search, as introduced by Kamoyedji et al. [2].

Table 4. Job CPU times

Job #	PC1	PC2 & PC3	PC4	PC5	PC6
1	02:14:46	01:06:44	00:54:21	00:39:40	00:36:09
2	00:41:43	00:26:38	00:20:27	00:13:53	00:13:10
3	01:37:14	01:09:28	00:22:44	00:12:45	00:11:15
4	00:17:43	00:12:01	00:07:03	00:05:37	00:04:49
5	00:26:04	00:22:22	00:07:07	00:05:24	00:04:47
6	00:46:37	00:31:49	00:13:23	00:07:59	00:06:54
7	00:17:09	00:11:34	00:05:41	00:04:53	00:04:15
8	00:12:51	00:08:38	00:05:24	00:04:29	00:03:19
9	00:25:20	00:14:45	00:11:08	00:08:32	00:07:55
10	00:36:28	00:09:09	00:07:53	00:05:59	00:04:14
11	00:39:35	00:24:53	00:10:42	00:04:08	00:03:16
12	00:13:12	00:06:35	00:05:09	00:03:55	00:03:01
13	00:34:50	00:19:47	00:12:33	00:07:48	00:07:10
14	00:36:33	00:24:56	00:10:55	00:05:45	00:04:19
15	00:12:23	00:07:19	00:06:51	00:05:04	00:04:28
16	00:29:58	00:19:18	00:16:58	00:13:08	00:11:27
17	00:45:57	00:30:44	00:25:59	00:20:31	00:17:54
18	00:55:44	00:36:54	00:32:25	00:25:17	00:22:13
19	01:20:36	00:52:29	00:46:40	00:36:55	00:32:20
20	01:37:25	01:05:44	00:56:27	00:45:18	00:39:42
21	01:44:18	01:12:44	01:06:06	00:51:19	00:44:56
22	01:52:04	01:26:35	01:14:53	01:00:08	00:52:44
23	02:14:32	01:30:28	01:19:17	01:03:01	00:55:19
24	02:22:38	01:32:22	01:23:02	01:07:02	00:58:40

Table 5. Makespan results for static job-worker assignment (H:M:S)

Method	24 Jobs	48 Jobs	72 Jobs
RMSLS [2]	02:06:58	04:08:41	06:10:31
Proposed Method	01:49:33	03:45:05	05:37:39
Improvement (H:M:S)	00:17:25	00:23:36	00:32:52
Improvement(%)	13.7%	9.49%	8.8%

To model the dynamic scenario, a non-preemptive M/M/c queue model is employed, with a fixed number of worker PCs denoted as $c = |U| = 6$ [9]. The average job arrival rate is approximately $\Lambda \approx 1$ job per 500 seconds, consistent with the model described in [2]. Job arrivals are simulated using a Poisson distribution to reflect continuous entry into the UPC system until the final job arrives [9]. Upon arrival, each job is immediately assigned to an available worker PC if one is idle; otherwise, it waits in the queue.

Table 6 presents the makespan comparison for the two algorithms under job loads of 24, 48 and 72. The results indicate that the proposed algorithm consistently outperfoms the reference approach. Furthermore, the data presented in Table 4 is used to estimate the average worker service rate through a two-step process. First, the average service rate for each worker PC is computed across all assigned jobs as follows:

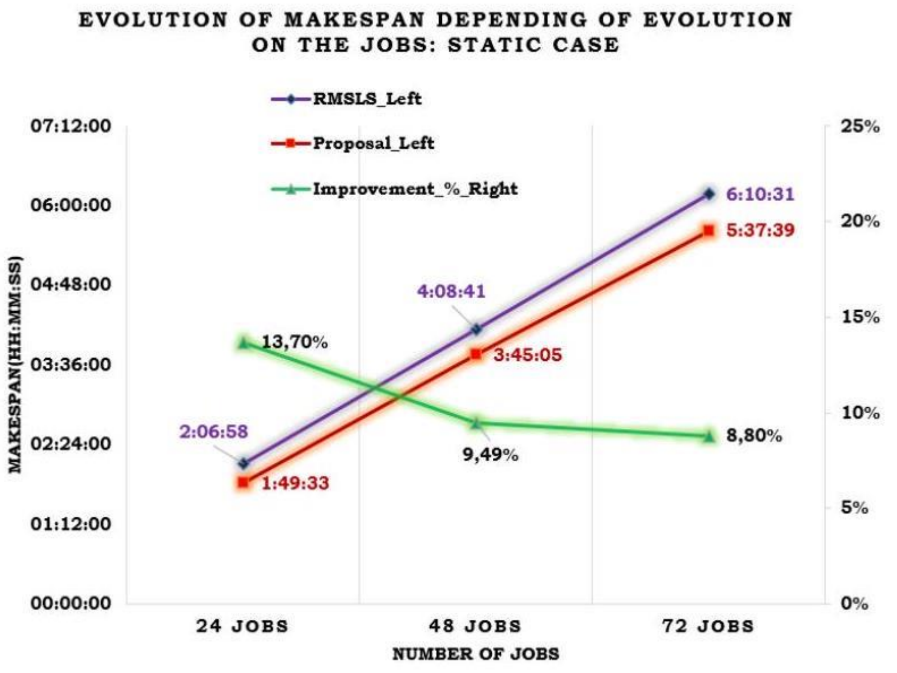

Fig. 2 Evolution of the makespan depending on the number of jobs

$$\mu_{avgw} = \frac{1}{|I|}\sum_{j\in I} \tau_{j,w} \forall w \in U \quad (1)$$

Given the use of six worker PCs, the previously computed values are averaged across all worker PCs as follows:

$$\mu_{avg} = \frac{1}{|U|}\sum_{w\in U} \mu_{argw} = 1job/2023s \quad (2)$$

The response time represents the total amount of time a job spends both in the queue and in service and is given by [9]:

$$\frac{C(|U|,\lambda/\mu)}{|U|\mu-\lambda} + \frac{1}{\mu} \quad (3)$$

Using the previous formula, the average response time of the system can be estimated by [2]:

$$AvgR_{t_1} = \frac{C(|U|,\lambda/\mu)}{|U|\mu-\lambda} + \frac{1}{\mu} \quad (4)$$

Where the response time is the total time a job spends both in the queue and in service, the probability that an arriving job is forced to join the queue, that is, all worker PCs are occupied, is given by:

$$C(|U|,\lambda/\mu) = \frac{1}{1+(1-\sigma)\left(\frac{|U|!}{(|U|\sigma)^{|U|}}\right)\sum_{k=0}^{|U|-1}\frac{(|U|\sigma)^k}{k!}} \quad (5)$$

Which is Erlang's C formula [9]. Using the Erlang C formula, and the approximate value of σ, $\sigma = \frac{1/500s}{(6*1/2023)} \approx 67\%$, as follows:

$$C(|U|,\lambda/\mu) \approx \frac{1}{1+\left(\frac{0.33*|6|!}{(4.02)^6}\right)\sum_{k=0}^{5}\frac{(4.02)^k}{k!}} \approx 0.28 \quad (6)$$

So, $AvgR_{t_1} = \frac{0.28}{6*1\,job\,/2023s - 1\,job\,/500s} + \frac{1}{1\,job\,/2023s} = \frac{0.28}{6*1\,job\,/2023s - 1\,job\,/500s} + 2023s = 290s + 2023s = 2313s = 38min33s$

Using experiment results, the average response time of the system is estimated for 24 distinct jobs as 33 min $46s$ $AvgR_{t_2} = \frac{48626}{24} = 2026s = 33\text{ min}46s$

The theoretical average response time of the system, AvgRt1, is way greater than the average response time, AvgRt2, calculated using experimental data.

This is mainly because 2/3 of the available jobs use the most powerful machines during execution and are assigned to either worker 6, worker 5, or worker 4 (half of the available worker PCs). Table 6 and Figure 3 clearly show that the proposed method outperforms the reference algorithm in terms of makespan.

Table 6. Makespan results for dynamic job-worker assignment (H:M:S)

Method	24 jobs	48 jobs	72 jobs
RMSLS [2]	05:39:48	10:35:38	14:40:01
Proposal	04:09:53	09:01:55	12:15:22
Improvement (H:M:S)	01:29:55	01:33:43	02:24:42
Improvement (%)	26.46%	14.74%	16.4%

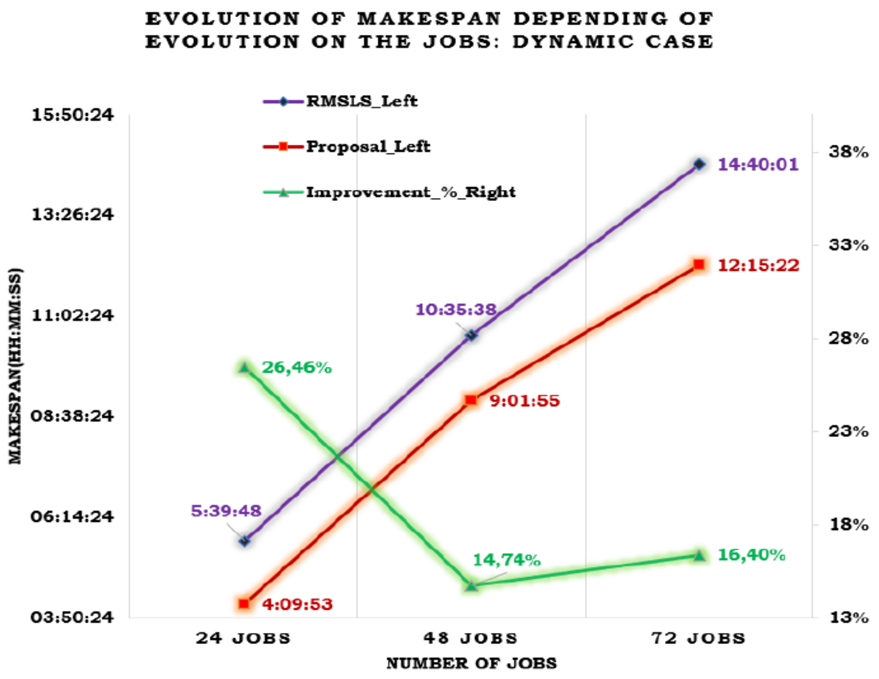

Fig. 3 Evolution of the makespan depending on the number of jobs

Table 7 reports the mean makespan and standard deviation observed across these experiments.

Table 7. Statistical summary of makespan results over multiple iterations for each job set

Number of Jobs	Mean Makespan (s)	Standard Deviation (s)	Number of Iterations
24 Jobs	4:34:40	0:27:31	50
48 Jobs	8:23:58	0:35:23	100
72 Jobs	11:48:33	0:41:27	150

5.5. Discussion of Performance Gains

The superior performance of the proposed genetic algorithm, up to 26.4% reduction in makespan compared to the reference algorithm, can be attributed to several core design improvements. The approach takes into account both job thread counts and available CPU cores, allowing for efficient, contention-free allocation, which was often overlooked in previous studies. A greedy initialization method is employed instead of random seeding, improving the quality of the initial solutions and accelerating convergence. Crossover and mutation were designed to assure both variety and convergence: crossover mixes successful patterns from top candidates, whilst mutation avoids premature convergence by exploring new options.

6. Conclusion

This study proposed a genetic job scheduling algorithm that incorporates both the number of threads used by jobs during execution and the number of CPU cores available on each worker PC, with the objective of minimizing the overall makespan. The algorithm was validated through a series of experiments involving up to 72 jobs and six worker PCs. The results indicate an improvement of up to 26.4% in makespan compared to the reference algorithms. Aside from theoretical contributions, this approach has practical implications for telecommunications carriers. Specifically, the suggested UPC-based scheduling technique can be used to dynamically offload non-critical computing jobs (such as billing procedures, log analytics, and traffic monitoring) to underutilized machines across dispersed infrastructures. However, the current study has significant shortcomings that provide opportunities for further research.

First, it presupposes that job execution timings are known in advance, which may not be practical in unpredictable contexts. Second, the tuning of GA parameters (population size, crossover, and mutation rates) was empirical and may be improved using automated or adaptive procedures. In particular, the present population size was moderate, which may limit the investigation of the solution space and result in a lack of globally optimum solutions. Finally, the proposed technique is currently centralized; expanding it to accommodate multi-master or decentralized coordination may improve scalability and fault tolerance. Future studies will address these issues by combining predictive modeling of job durations and investigating reinforcement learning methodologies for more adaptable scheduling.

References

[1] Ling Xu et al., "Dynamic Task Scheduling Algorithm with Deadline Constraint in Heterogeneous Volunteer Computing Platforms," *Future Internet*, vol. 11, no. 6, pp. 1-16, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[2] Ariel Kamoyedji et al., "A Proposal of Job-Worker Assignment Algorithm Considering CPU Core Utilization for User-PC Computing System," *International Journal of Future Computer and Communication*, vol. 11, no. 2, pp. 40-46, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[3] Manjot Kaur Bhatia, "Task Scheduling in Grid Computing: A Review," *Advances in Computational Sciences and Technology*, vol. 10, no. 6, pp. 1707-1714, 2017. [Google Scholar] [Publisher Link]

[4] Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour, "Economic Scheduling in Grid Computing," *8th International Workshop on Job Scheduling Strategies for Parallel Processing*, Edinburgh, United Kingdom, pp. 128-152, 2002. [CrossRef] [Google Scholar] [Publisher Link]

[5] Tao Xie, Andrew Sung, and Xiao Qin, "Dynamic Task Scheduling with Security Awareness in Real-Time Systems," *19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, USA, 2005. [CrossRef] [Google Scholar] [Publisher Link]

[6] Man Wang et al., "The Dynamic Priority-Based Scheduling Algorithm for Hard Real-Time Heterogeneous CMP Application," *Journal of Algorithms and Computational Technology*, vol. 2, no. 3, pp. 409-427, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[7] Zahra Pooranian et al., "GLOA: A New Job Scheduling Algorithm for Grid Computing," *International Journal of Artificial Intelligence and Interactive Multimedia*, vol. 2, no. 1, pp. 59-64, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[8] Ye-In Seol, and Young-Kuk Kim, "Applying Dynamic Priority Scheduling Scheme to Static Systems of Pinwheel Job Model in Power-Aware Scheduling," *The Scientific World Journal*, vol. 2014, no. 1, pp. 1-9, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[9] Leonard Kleinrock, *Queueing Systems, Volume I Theory*, Wiley Interscience, 1975. [Google Scholar] [Publisher Link]

[10] Lale Özbakir, Adil Baykasoğlu, and Pınar Tapkan, "Bees Algorithm for Generalized Assignment Problem," *Applied Mathematics and Computation*, vol. 215, no. 11, pp. 3782-3795, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[11] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed., Addison-Wesley Professional, 1994. [Google Scholar] [Publisher Link]

[12] Ritu Garg, and Awadhesh Kumar Singh, “Adaptive Workflow Scheduling in Grid Computing Based on Dynamic Resource Availability,” *Engineering Science and Technology, An International Journal*, vol. 18, no. 2, pp. 256-269, 2015. [CrossRef] [Google Scholar] [Publisher Link]
[13] Michel Barbeau, and Evangelos Kranakis, *Principles of Ad-Hoc Networking*, 1st ed., John Wiley, 2007. [Google Scholar] [Publisher Link]
[14] D.I. George Amalarathinam, and A. Maria Josphin, “Dual Objective Dynamic Scheduling Algorithm (DoDySA) for Heterogeneous Environments,” *Advances in Computational Sciences and Technology*, vol. 10, no. 2, pp. 171-183, 2017. [Google Scholar] [Publisher Link]
[15] Esra’a Alkafaween et al., “An Efficiency Boost for Genetic Algorithms: Initializing the GA with the Iterative Approximate Method for Optimizing the Traveling Salesman Problem-Experimental Insights,” *Applied Sciences*, vol. 14, no. 8, pp. 1-19, 2024. [CrossRef] [Google Scholar] [Publisher Link]
[16] Jing Xu, “Improved Genetic Algorithm to Solve the Scheduling Problem of College English Courses,” *Complexity*, vol. 2021, no. 1, pp. 1-11, 2021. [CrossRef] [Google Scholar] [Publisher Link]
[17] Xudong Zhou et al., “A Static Assignment Algorithm of Uniform Jobs to Workers in a User-PC Computing System using Simultaneous Linear Equations,” *Algorithms*, vol. 15, no. 10, pp. 1-15, 2022. [CrossRef] [Google Scholar] [Publisher Link]

Appendix

1. Pseudo-Code of the Static Job Scheduling Algorithm

```
Algorithm 2 Job Assignment Algorithm: Greedy Part
  procedure BUlldInITIALPOPULATION(workers, jobs)
    Initialize an empty population
    Define the predefined thresholds as an array
    for threshold in predefined thresholds do
      Create two classes of workers and jobs
      for $w$ in workers do
        if w.nbOfCores $\geq$ threshold then
          Add w to the 1st worker class
        else
          Add w to the 2nd worker class
        end if
      end for
      for j in jobs do
        if $j \geq$ threshold then
          Add j to the 1st job class
        else
          Add j to the 2nd job class
        end if
      end for
      for $k$ in [0,1] do
        Sort workerClass[k] based on CPU information
        Get the current job class jobClass [$k$]
        for each job in the current job class do
          Calculate and assign a value θ to the job
        end for
        Sort the jobs in the class based on their θ value
        for each job in the sorted job class do
          Find the most suitable worker for the job based
on cost and resource constraints
          if a suitable worker is found then
            Assign the job to that worker
          end if
        end for
      end for
      Initialize an empty solution
      for jobClass in jobClasses do
        for workerClass in workerClasses do
          Apply the greedy algorithm to the current
jobClass and workerClass
          Add the job-worker assignments to the solution
        end for
      end for
      if the solution contains all jobs then
        Add the solution to the population
      end if
    end for
    return the population
  end procedure
```

2. Pseudo-code of the Genetic Algorithm

```
Algorithm 3 Job Assignment Algorithm: Genetic Part
  procedure STARTGENETICALG(P (population),
  nbStopCriteria)
    conv ← 0
    solutionFinal ← $null$
    while conv < nbStopCriteria do
      Calculate scores
      Sort scores in ascending order
      MAKESELECTION
      DOCROSSOVER
      DOMUTATION
      score1 ← P.child1.getScore ()
      score2 ← P.child2.getScore()
      if (score1 < scores[0] or scores[1]) then
        Remove the last parent from $P$
        Add child1 to $P$
      end if
      if (score2 < scores[0] or scores[1]) then
        Remove the last parent from $P$
        Add child2 P
      end if
      if (child1 or child2 isAdded) then
        conv ← 0
      else
        $conv \leftarrow conv + 1$
      end if
      solutionFinal ← p.solutions[0]
```

```
        end while
        return solutionFinal
    end procedure
    procedure MAKESELECTION
        Choose the best two solutions as parents, let them be
child 1 and child 2
    end procedure
    procedure DOCROSSOVER
        c ← random(0, child1.content.length)
        s1 ← child1.content.clone()
        s2 ← child2.content.clone()
        for i←c up to the length of child1.content-1 do
            Swap s1[i] and s2[i]
        end for
        Set child1.content to s1
        Set child2.content to s2
    end procedure
    procedure DOMUTATION
        s1 ← child1.content.clone()
        s2 ← child2.content.clone()
        m ← random(0, child1.content.length)
        Swap s1[m] and s1[m+1]
        m ← random(0, child1.content.length)
        Swap s2[m] and s2[m+1]
        Set child1.content to s1
        Set child2.content to s2
    end procedure
```