IJCI V11-1(2024) 30–43
International Journal of Computers and Information (IJCI)
Available online at *https://ijci.journals.ekb.eg/*

Comparative Study of Intelligent Scheduling Algorithms for Heterogeneous Systems

Abla Saad[a], Osama Abdelraouf [a], Ahmed Kafafy [b]

[a] Machine Intelligence Dept, Faculty of Artificial Intelligence, Menoufia University, Egypt.
[b] Operations Research & Decision Support Dept, Faculty of Computers and Information, Menoufia University, Egypt.
abla.saad@ci.menofia.edu.eg, osamaabd@ci.menofia.edu.eg, ahmed.kafafi@ci.menofia.edu.eg

Abstract

Scheduling tasks in a heterogeneous computing environment can be a challenging problem due to the diverse range of hardware and software resources available. In this comparative study different approaches are investigated for solving multitask scheduling in the heterogeneous computing environment, reviewing the literature on the topic, highlighting the strengths and weaknesses of different scheduling algorithms then, formulate a hypothesis about how multitask scheduling can be optimized in a heterogeneous computing environment and design an experiment to test this hypothesis. This study involves running a variety of scheduling algorithms as GRASP, Tabu Search, SA, GA, HEFT and FCFS on a heterogeneous computing platform. This study yields valuable insights on the efficacy of various optimization algorithms for scheduling problems and emphasizes the significance of selecting suitable algorithms based on the problem's specific features. The result of this study indicates that the GRASP algorithm outperforms other scheduling algorithms as HEFT Ranked up, Tabu Search, SA, GA, HEFT Ranked down, and FCFS on producing schedules with shorter completion times. This is a critical factor when evaluating scheduling algorithms. The exceptional performance of GRASP can be credited to its effective navigation of the solution space and its adept utilization of a blend of greedy constructive heuristics and randomized local search methods, which enable it to achieve top-notch solutions.

Keywords: Task Scheduling, Heterogeneous Computing Environment, Metaheuristics;

I. Introduction

Task scheduling in a heterogeneous computing environment involves allocating tasks to available hardware and software resources in a way that maximizes the overall efficiency and performance of the system. However, this can be a complex problem due to the wide range of resources that may be available, including different types of processors, memory, and storage. In addition, tasks may have different requirements in terms of their resource needs and deadlines, further adding to the complexity of the scheduling problem. As a result, finding an optimal schedule for tasks in a heterogeneous computing environment can be a challenging task. In this study, heuristic and static scheduling algorithms are investigated for solving multitask scheduling in such environments. This study also examines the factors that impact the performance of scheduling algorithms in heterogeneous computing environments by understanding the strengths and weaknesses of different scheduling approaches, this research hope to provide insights into how multitask scheduling can be optimized in such environments [1], [2].

Traditional priority techniques and heuristic techniques are two different types of algorithms that can be used for scheduling tasks in a heterogeneous computing environment. Traditional priority techniques involve assigning a priority to each task and scheduling the tasks based on their priorities. These techniques are simple and easy to implement, but they may not always produce an optimal schedule. Heuristic techniques, on the other hand, are strategies that are designed to find a good, but not necessarily optimal, solution to a problem. Heuristic techniques for scheduling tasks in a heterogeneous computing environment may use a variety of strategies, such as simulating natural phenomena or using metaheuristics, to find a good schedule. Heuristic techniques may be

more computationally intensive than traditional priority techniques, but they can often find a better schedule in a reasonable amount of time [3] [4].

In this comparative study, a series of novel scheduling algorithms are implemented to optimize the allocation of tasks across different computing resources, considering factors like task characteristics, resource capabilities, and system load. This study focuses on two distinct categories of scheduling algorithms, each aimed at enhancing system performance and contributing to the advancement of high-performance computing. One category comprises heuristic algorithms, which encompass metaheuristic approaches like Greedy Randomized Adaptive Search Procedure (GRASP), Tabu Search, Genetic Algorithm (GA), and Simulated Annealing (SA). The other category comprises static algorithms, including Heterogeneous Earliest Finishing Time (HEFT) and First Come First Served (FCFS). By evaluating and comparing the performance of these algorithms in heterogeneous computing environments, this research seeks to identify the most effective scheduling strategies for optimizing system efficiency and resource utilization.

The subsequent sections of this paper are organized as follows: In Section 2, this study presents the task model. In Section 3, illustrate the concepts of scheduling problem. Sections 4 provide concise explanations of the implemented algorithms FCFS (First Come First Served), HEFT (Heterogeneous Earliest Finishing Time), GA (Genetic Algorithm), Simulated Annealing (SA) and the greedy randomized adaptive search method (GRASP) algorithms, respectively. The evaluation of performance and the associated parameters are discussed separately in Sections 5. Finally, our conclusions and recommendations for future research are summarized in Section 6.

II. Task Model

In a parallel and distributed computing environment, an application can be divided into a set of tasks, which are represented using a directed acyclic graph (DAG) $G = (N, E)$. The set N consists of n $nodes$, and each node n_i in N represents a task in the application. The set E is a set of e directed edges that represent dependencies between tasks. Each edge $e_{(i,j)}$ in E connects two nodes in the graph, with the first node being the parent or protector node and the second node being the child node as in figure 1. The child node cannot be executed until the parent node has completed [5]. The node with no children is known as the $exit\ node$. When two nodes are assigned to the same processor, the communication cost between them is minimal. The mapping of nodes to processors and the optimization of execution time depends on ranking, which is a measure of the importance or priority of a node.

The weight W_i of each node n_i reflects the processing expenses of the node, and the computation cost is the estimated execution time (EET) for completing task n_i on processor p_j.The average execution cost of a task n_i $EET_{avg}(n_i)$ is defined in equation (1)

$$EET_{avg}(n_i) = \sum_{p=1}^{j} {}^{W_i}/_{p_j} \quad (1)$$

The average execution cost of a task $EET_{avg}(n_i)$ is calculated as the sum of the execution costs on all processors W_i, divided by the total number of processors p_j.

The transfer time of data between processors is stored in a matrix B of size $q\ x\ q$, and the communication startup time for each processor is given in a q-dimensional vector L [1].The communication cost between two tasks n_i and n_j, which are scheduled on processors p_m and p_n, respectively, is represented by the edge between them.

Various predefined criteria such as upward ranks, downward ranks presented in [6], and others developed by different researchers can be used to prioritize all the tasks in a given DAG. Once the tasks in the DAG have been prioritized, Equations 2 and 3 can be utilized to determine the Earliest Start Time (EST) and Latest Finish Time (LFT) attributes.

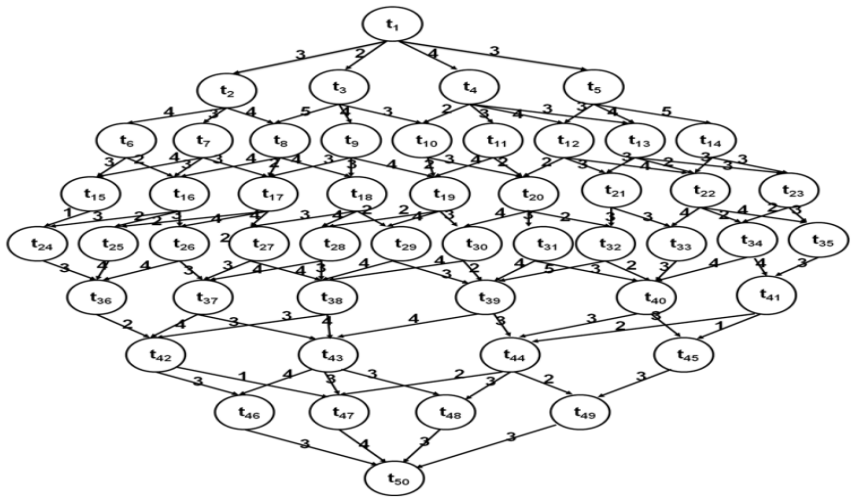

Fig. 1. Molecular DAG with 50 nodes **[7]**

$$EST(n_i, p_j) = \begin{cases} 0 & if\ n_i = Entery\ node \\ \max\{EST(n_j) + EET(n_j) + C(n_{j,i})\}_{,n_j \epsilon pred(n_i)} & ,otherwise \end{cases} \quad (2)$$

Equation 2 defines the Earliest Start time $EST(n_i, p_j)$ of a task n_i at any processor p_j. If n_i is an entry task, its EST will be zero. However, if it has immediate predecessor tasks represented by n_j, the value of EST will be determined using Equation 2. Here, $EST\ (n_j)$ represents the EST of predecessor tasks, $EET(n_j)$ denotes their execution times, and $C(n_{j,i})$ signifies the communication cost between the predecessor task and n_i.

$$LFT(n_i, p_j) = \begin{cases} DL & if\ n_i = Exit\ node \\ \min\{LFT(n_j) - EET(n_j) - C(n_{j,i})\}_{,n_j \epsilon succ(n_i)} & ,otherwise \end{cases} \quad (3)$$

Equation 3 defines the Latest Finish Time $LFT(n_i, p_j)$ of a task n_i at any processor p_j. If n_i is an exit task, its LFT will be equal to the Deadline (DL). However, if n_i has immediate successor tasks represented by n_j, the values of LFT will be determined using Equation 3. Here $LFT\ (n_j)$ denotes the LFT of successor tasks, $EET(n_j)$signifies their execution times, and $C(n_{j,i})$ represents the communication cost between task n_i.and the successor task n_j.

The Actual Start Time $AST_{(n_i)}$ of a task n_i is determined by Equation 5, which specifies the start time of the task. However, if $n_i$is the entry task n_{entery}, its AST is determined by Equation 4. It is important to note that the value of AST depends on the task and the context in which it is being evaluated.

$$AST_{(n_i)} = EST_{(n_{entery})} = 0 \quad (4)$$

$$AST_{(n_i)} = min_{(EFT)} \quad (5)$$

Equation 6 provides the definition of the Actual Finished Time (AFT) for a task n_i ($AFT_{(n_i)}$), which represents the time when the task is finished. The value of AFT is dependent on the specific task being evaluated $EET(n_i)$ and can be calculated using the equation.

$$AFT_{(n_i)} = \ AST_{(n_i)} + \ EET(n_i) \quad (6)$$

The main goal of the objective function is to minimize the makespan $MS(Schedule\ length)$. It is determined by Equation 7, which provides a definition of the time duration between the start of the earliest task and the finish of the latest task.

$$MS(Schedule\ length) = \ min\ EFT(n_{Exit}) \tag{7}$$

III. Scheduling Problem

The focus of this paper is on the static scheduling of a specific application in a heterogeneous computing system There are two main approaches to task scheduling: one of them is static and the other is dynamic. Static scheduling involves determining the task schedule before the tasks are executed, while dynamic scheduling is suitable for situations where the computing system and task parameters are not known in advance. Where dynamic scheduling algorithms are used when the workload and system status are only known at runtime, and they make decisions about task assignment during execution [8] [9]. However, these algorithms often have additional overhead compared to static scheduling methods. In contrast, static scheduling involves creating a schedule before tasks are executed, and it does not incur any additional overhead during runtime. The goal of task scheduling is to divide an application into different parts or nodes, and to determine the priority of these nodes in order to minimize the overall execution time of the application. In this paper, we will discuss lists-based scheduling heuristics (FCFS, HEFT, GA, SA, GRASP and Tabu Search) and their effectiveness in a heterogeneous computing environment. We will also examine the impact of different scheduling parameters, such as schedule length, speedup, and efficiency, on the performance of these algorithms.

IV. Implemented Algorithms

FCFS (First Come First Served)

The First Come First Served (FCFS) algorithm is a simple and intuitive approach for task scheduling in heterogeneous computing platforms. The FCFS algorithm executes tasks in the order they arrive in the system as in equation (8), regardless of their priority $FCFS_{Rank(\mathrm{n_i})}$ or computing requirements [10].

$$FCFS_{Rank(\mathrm{n_i})} = \begin{cases} 0 & \text{if } PRED(n_i) = \emptyset \\ 1 + max\left\{FCFS_{Rank(n_j)} : n_j \in PRED(n_i)\right\}, & Otherwise \end{cases} \tag{8}$$

Alg.1: FCFS

Inputs:

n_i: Number of Tasks
m: Number of processors
$\overline{Cij}$: Avg communication cost
$\overline{Wi}$: Avg computation time

Output: Makespan

Begin:

1. $Schedule_{List} = \emptyset$; *//Schedule list*
2. **For** $i \in \{1, \dots, m\}$ **do:**
3. $FCFS_{Rank(\mathrm{n_i})} = \begin{cases} 0 & \text{if } PRED(n_i) = \emptyset \\ 1 + max\left\{FCFS_{Rank(n_j)} : n_j \in PRED(n_i)\right\}, & otherwise \end{cases}$
4. $Schedule_{List} \leftarrow Schedule_{List} \cup \{ FCFS_{Rank(\mathrm{n_i})}\}$.
5. **End For**
6. $Schedule_{List} \leftarrow$ ArrangASce($schedule_{List}$) *//arrange in Ascending order.*
7. **Assign(** $schedule_{List}$ **,** m**)**
8. **Return Makespan***; //return the makespan.*

End

Where, $PRED(n_i)$ is the predecessor task. This approach is easy to implement and provides a fair allocation of resources to all tasks. However, the FCFS algorithm has some disadvantages. Firstly, it can lead to long waiting times for tasks with higher priority or shorter computing requirements, as they are blocked by longer tasks. Secondly, the FCFS algorithm does not take into account the resource availability, which can result in inefficient resource utilization and low system performance. Despite its drawbacks, the FCFS algorithm can be suitable for simple and low-priority tasks, where the goal is to achieve a fair allocation of resources to all tasks without

complicated scheduling algorithms. Overall, the choice of task scheduling algorithm for heterogeneous computing platforms should be based on the specific requirements of the system and the tasks to be executed [11]. The pseudo code for FCFS algorithm is mentioned in **Alg.1** below.

HEFT (Heterogeneous Earliest Finish Time)

HEFT (Heterogeneous Earliest Finish Time) is a simple and effective scheduling technique for static task scheduling in both heterogeneous and homogeneous computing environments with a limited number of processors [6]. It consists of two stages: a prioritization phase and a processor selection stage. During the prioritization phase, HEFT calculates the priority of each task using an $upward\ ranking$ ($rank_{up}$) method and $downward\ ranking$ ($rank_{down}$) according equations 9,10 respectively.

$$Rank_{up(n_i)} = \overline{Wi} + \max_{Tj \in SUCC(Ti)} \{\overline{Cij} + \ Rank_{up(n_i)}\} \quad (9)$$

$$Ranked\_down(n_i) = \max_{Tj \in Pred(T_i)} \{Ranked\_down(n_j) + \overline{Wi} + \overline{Cij}\} \quad (10)$$

This involves traversing the application graph in an upward direction and calculating the mean communication $\overline{Cij}$ and computation costs $\overline{Wi}$ for each node n_i. The resulting list of nodes is then sorted in decreasing order of $rank_{up}$ and increasing order of $rank_{down}$. HEFT uses a tie-breaking policy to determine which node to select when there are multiple nodes with the same $rank_{up}$ value: There are several advantages to using the HEFT algorithm for solving scheduling problems in a heterogeneous computing platform. One advantage is that it is a simple and effective technique for finding good schedules in a short amount of time. Another advantage is that it is a static scheduling algorithm, which means that it does not require any runtime overhead and can be used to pre-schedule tasks. A disadvantage of HEFT is that it may not always find the optimal solution due to its reliance on mean values and the use of a Tie breaking policy. It may also be sensitive to changes in the problem, such as changes in the processing times or communication costs of the tasks. In addition, HEFT may not be suitable for dynamic scheduling scenarios, where the workload and resources are not known in advance. The pseudo code for HEFT algorithm is mentioned in **Alg.2** below.

Alg.2: HEFT

Inputs:

n_i: Number of Tasks
m: Number of processors
$\overline{Cij}$: Avg communication cost
$\overline{Wi}$: Avg computation time

Output: Makespan

Begin:

$Schedule_{List} = \emptyset$; *//Schedule list*

1. **If** $Function = Ranked\_up$
2. **For** $i \in \{1, \dots, m\}$ **do:**
3. $Rank\_up(n_i) = \overline{Wi} + \max_{Tj \in SUCC(Ti)} \{\overline{Cij} + \ Rank\_up(n_i)\}$
4. $Schedule_{List} \leftarrow \ Schedule_{List} \cup \{\ Rank\_up(n_i)\}$.
5. **End For**
6. $Schedule_{List} \leftarrow$ ArrangDesc($\boldsymbol{s}chedule_{List}$) *//arrange in descending order.*
7. **ElseIf** $Function = Ranked\_down$
8. **For** $i \in \{1, \dots, m\}$ **do:**
9. $Ranked\_down(n_i) = \max_{Tj \in Pred(T_i)} \{Ranked\_down(n_j) + \overline{Wi} + \overline{Cij}\}$
10. $Schedule_{List} \leftarrow \ Schedule_{List} \cup \{\ Rank_{down(n_i)}\}$
11. **End For**
12. $Schedule_{List} \leftarrow$ ArrangASce($\boldsymbol{s}chedule_{List}$) *//arrange in Ascending order.*
13. **Assign(** $\boldsymbol{s}chedule_{List}$ **,** m**)**
14. **Return Makespan***; //return the makespan.*

End

GA (Genetic algorithms)

Genetic algorithms (GA) are a type of optimization technique that can be used to solve scheduling problems in a heterogeneous computing platform [12]. They are inspired by the process of natural evolution and are used to find the optimal solution to a problem by simulating the process of natural selection. In a genetic algorithm, a population of schedules is initialized and is evolved through a series of generations. Each schedule is represented as a chromosome, which is a set of genes that encode a potential solution to the problem. The chromosomes are evaluated using an objective function, which measures the quality of the schedule.

The fittest chromosomes are then selected to be used in the next generation, and the less fit chromosomes are discarded. The selected chromosomes are then subjected to genetic operations, such as crossover with $Pc\ probability$ and mutation with $Pm\ probability$, to produce a new population of chromosomes. This process is repeated until a satisfactory schedule is found Sol^*, or a predetermined stopping criterion is reached as mentioned in figure 2 below. Genetic algorithms can be effective at finding good schedules, but they may not always find the optimal solution due to the probabilistic nature of the algorithm [13] [14].

Alg.3: GA

Inputs:
n_t: *number of tasks*
m: *number of processors*
N: *Pop Size.*
P_c: *probability of Crossover*
P_m: *Probability of Mutation.*
Output: *Archive: all improved solutions found over generations.*
Begin:

```
1.    Pop ← Random (schedule_List )
2.    While Stopping criterion is not satisfied do:
3.      r ← random(0,1);
4.      IF (r < Pc) then: //apply crossover
5.        X′, Y′ ← Crossover(X, Y).
6.      EndIF
7.      r ← random(0,1);
8.      IF (r < Pm) then:
9.        X″ ← Mutate(X′).
10.       Y″ ← Mutate(Y′).
11.     End IF
12.     NewPop ← NewPop ∪ {X″, Y″}
13.   End For
14.   NewPop ← Evaluate(NewPop)
15.   Pop ← UpdatePop(Pop, NewPop);
16.  End While
17.  Sol* ← FindTheBest(Pop)
18.  Return Sol*;
End
```

There are several advantages to using the genetic algorithm for solving scheduling problems in a heterogeneous computing platform. One advantage is that it can find good solutions in a short amount of time by simulating the process of natural selection. Another advantage is that it can handle problems with a large search space and no known algorithm for finding the optimal solution. A disadvantage of the genetic algorithm is that it may not always find the optimal solution due to its probabilistic nature, which can cause it to get stuck in local optima. It may also require a large number of generations to find a satisfactory solution, which can increase the running time. In addition, the performance of the genetic algorithm may be sensitive to the choice of parameters, such as the crossover rate and the mutation rate, which can require fine-tuning to achieve good results.

The pseudo code for GA algorithm is mentioned in **Alg.3** above.

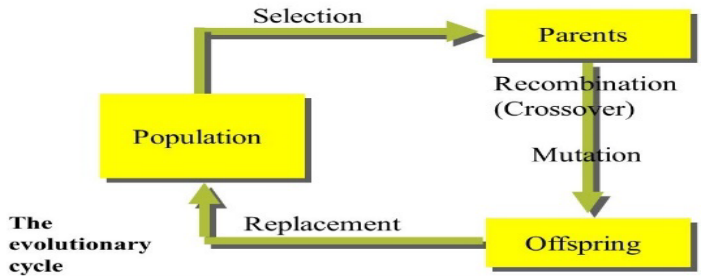

Fig. 2. GA evolutionary cycle

SA (Simulated annealing)

Simulated annealing (SA) is a heuristic optimization technique that can be used to solve scheduling problems in a heterogeneous computing platform [15]. It is a randomized search algorithm that is inspired by the annealing process used in metallurgy to harden and purify metals. In simulated annealing, a schedule is initialized S_0 with F_{S0} which F is the objective function and is modified through a series of iterations, or "temperature" steps. At each step, a new schedule S_1 is generated by making a random change to the current schedule and evaluating the resulting change in the objective function F_{S1}. The new schedule is accepted if it results in an improvement in the objective function, or it is accepted with a certain probability if it results in a worsening of the objective function. This probability is determined by a temperature parameter $Temp\ T$, which is gradually decreased as the algorithm progresses. The temperature determines the likelihood of accepting a worse solution, and it allows the algorithm to escape from local optima and explore the search space more extensively as in (equation 11). Simulated annealing can be effective at finding good schedules [16] [17].

$$P_{Accept} = \begin{cases} 1 & if\ F_{S0} \le F_{S1} \\ exp\left(\frac{F_{S0}-F_{S1}}{Temp}\right) & if\ F_{S0} > F_{S1} \end{cases} \tag{11}$$

Alg. 4: Simulated Annealing:

Begin:

1. $S_0 \leftarrow intial\ solution$.
2. $T \leftarrow T_0$ *//initial temperature.*
3. **While** *termination conditions not satisfied* **do**
4. $S_1 \leftarrow generation\ solution\ S_1$.
5. **If** $F(S_1) < F(S_0)$ **then**:
6. .....$S_0 \leftarrow S_1$
7. ..**ElSE**
8. $S_0 \leftarrow S_1$ **with acceptance probability** $\boldsymbol{P}(T, S_0, S_1)$
9. **EndIf**
10. **Update** (T)
11. **END While**

Return *the best solution*

There are several advantages to using the simulated annealing algorithm for solving scheduling problems in a heterogeneous computing platform. One advantage is that it can escape from local optima and explore the search space more extensively. Another advantage is that it can handle problems with a large search space and no known algorithm for finding the optimal solution. A disadvantage of simulated annealing is that it may be slower than other optimization techniques due to the need to evaluate the objective function at each iteration. It may also require a large number of iterations to find a satisfactory solution, which can increase the running time. In addition, the performance of simulated annealing may be sensitive to the choice of parameters, such as

the initial temperature and the cooling schedule, which can require fine-tuning to achieve good results. The pseudo code for SA algorithm is mentioned in **Alg.4** above.

GRASP (Greedy Randomized Adaptive Search Procedure)

GRASP (Greedy Randomized Adaptive Search Procedure) is a heuristic optimization algorithm that can be used to solve scheduling problems in a heterogeneous computing platform [18] [19] [20]. It is a randomized search algorithm that combines elements of greedy and local search techniques. Here, we adopt two heuristic functions $Ranked\_up_{(n_i)}$ *and* $Ranked\_down(n_i)$ mentioned before in this article. In GRASP, a schedule is constructed through a sequence of iterations, or "constructive phases," in which a subset of tasks, known as the "candidate list, CL " is selected and added to the schedule. At each constructive phase, the candidate list CL is modified by adding or removing tasks based on a set of rules, known as the "construction f heuristic" according to equation (12) below.

$$f \in [\, f_{min},\ f_{min} + \alpha \times (\, f_{max} - \, f_{min})] \tag{12}$$

Alg.5: GRASP (α, $HeurFun$**)**

Inputs:
α: parameter controls greediness /randomness.
$HeurFun$: The Heuristic function used in GRASP
Output: Sol^**:* best solution found after local search.
Begin: *//begin construction phase*

1. $Sol \leftarrow \emptyset$; CL←Ø; *//initialize Sol & Candidate list*
2. **If** $HeurFun = Rank\_up$ *//in case of bottom level*
3. **For** $i \in \{1, \dots, m\}$ **do:**
4. $Rank\_up(n_i) = \overline{Wi} + \max_{Tj \in SUCC(Ti)} \{\overline{Cij} + \ Rank_{up(n_i)}\}$
5. $CL \leftarrow$ CL $\cup$ $\{Rank\_up(n_i)\}$. *//construct candidate list*
6. **End For**
7. **ElseIf** $HeurFun = Rank\_down$ *//in case of top level*
8. **For** $i \in \{1, \dots, m\}$ **do:**
9. $Ranked\_down(n_i) = \max_{Tj \in Pred(T_i)} \{Ranked\_down(n_j) + \overline{Wi} + \overline{Cij}\}$
10. $CL \leftarrow$ CL $\cup$ $\{Ranked\_down(n_i)\}$. *//construct candidate list*
11. **End For**
12. **EndIf**
13. $CL \leftarrow$ ArrangDesc(CL)*//arrange in descending order.*
14. $RCL \leftarrow$ {the first α × |CL| elements of CL}.
15. **For** $i \in \{1, \dots, |\text{RCL}|\}$ **do:**
16. Randomly pick n_i from RCL.
17. $Sol \leftarrow Sol \ \cup n_i$. *// put the task in the solution*
18. $RCL \leftarrow UpdateRCL(RCL)$*//replace item* n_i *by new one*
19. **End For**
20. **For** j ∈ $\{1, \dots, |Sol|\}$ *//begin local search phase*
21. $Sol' \leftarrow Swap(Sol, n_j, n_{j-1})$. *//get new neighborhood sol*
22. $Sol'' \leftarrow Validate(Sol')$ *//validate swap*
23. **If** $F_1(Sol'') \ < F_1(Sol^*)$ **then**:
24. $Sol^* \ \leftarrow Sol''$ *//keep Sol with the best make-span*
25. **EndIf**
26. **End For**
27. **End While**
28. **Return** Sol^**; //return the improved solution*

End

Where, the parameter $\alpha \in [0,1]$ is used to control the balance between greediness and randomness. The candidate list CL is then sorted according to a "selection function," which measures the desirability of adding a task to the schedule. The task with the highest score is added to the schedule, and the process is repeated until the schedule is complete. After the constructive phase, GRASP performs a local search to further improve the schedule. GRASP can be effective at finding good schedules, but it may not always find the optimal solution due to the greedy nature of the construction heuristic [17].

There are several advantages to using the GRASP (Greedy Randomized Adaptive Search Procedure) algorithm for solving scheduling problems in a heterogeneous computing platform. One advantage is that it can find good solutions in a short amount of time due to its greedy construction heuristic, which allows it to focus on the most promising tasks at each constructive phase. Another advantage is that it can adapt to changes in the problem by updating the candidate list and the selection function at each constructive phase, which allows it to explore different parts of the search space. A disadvantage of GRASP is that it may not always find the optimal solution due to its local search nature, which can cause it to get stuck in local optima. It may also be sensitive to the choice of the construction heuristic and the selection function, which can require fine-tuning to achieve good results. In addition, GRASP may be slower than other optimization techniques due to the need to evaluate the objective function at each iteration. The pseudo code for GRASP algorithm is mentioned in **Alg.5** below.

Tabu Search

Tabu search is a heuristic optimization algorithm that can be used to solve scheduling problems in a heterogeneous computing platform [21] [22] [23]. It is a metaheuristic algorithm that combines elements of local search and memory-based search. In tabu search, a schedule S_0 is modified through a series of iterations, or "tabu moves," in which a task is moved from its current position in the schedule to a new position. The new schedule is evaluated using an objective function, and the move is accepted if it results in an improvement in the objective function. If the move does not improve the objective function, it may still be accepted with a certain probability, known as the "aspiration level." The acceptance of non-improving moves allows the algorithm to escape from local optima and explore the search space more extensively. Tabu search also uses a "tabu list T_{List}" to prevent the algorithm from revisiting previously visited solutions and getting stuck in a loop. Tabu search can be effective at finding good schedules, but it may be slower than other optimization techniques due to the need to evaluate the objective function at each iteration.

Alg. 6: Adaptive Tabu Search:

Begin:

1. $S_0 \leftarrow intial\ solution$.
2. $T_{Size} \leftarrow Tabu_{Size}$.
3. $T_{List} \leftarrow \emptyset$
4. **While** *termination conditions not satisfied* **do**
5. $S_1 \leftarrow generation\ solution\ S_1$.
6. **If** $(F(S_1) - F(S_0)\ != 0\ )$&& $(!\,T_{List}.contains\ F(S_1))$ **then**:
7. .....$T_{List} \leftarrow S_1 \cup T_{List}$
8. **If** $length(T_{List}) > T_{Size}$) **then**:
9. T_{list}. Remove (Sol)
10. **Update** (T_{list})
11. .. **EndIf**
12. . **EndIf**
13. **END While**

Return *the best solution*

There are several advantages to using the tabu search algorithm for solving scheduling problems in a heterogeneous computing platform. One advantage is that it can escape from local optima and explore the search space more extensively, which allows it to find good solutions in a short amount of time. Another advantage is

that it uses a memory-based search, which allows it to remember previously visited solutions and avoid revisiting them, which can reduce the risk of getting stuck in a loop. A disadvantage of tabu search is that it may be slower than other optimization techniques due to the need to evaluate the objective function at each iteration. It may also require a large amount of memory to store the tabu list, which can be a limiting factor for large-scale scheduling problems. In addition, the performance of tabu search may be sensitive to the choice of parameters, such as the tabu list size and the aspiration level, which can require fine-tuning to achieve good results. The pseudo code for GRASP algorithm is mentioned in Alg.6 above.

V. Experimental Results and Analysis

The performance of the FCFS, HEFT, GA, SA, GRASP and Tabu Search algorithms was evaluated based on three metrics: schedule length, speedup, and efficiency used in [24].

- Schedule length, also known as makespan in equation (7), refers to the total execution time of an application or DAG.
- Speedup in equation (13) is calculated by dividing the $schedule\ length$ by the time it takes for the fastest processor ($Socket_{fastest}$) to complete the task.

$$Speed^{up} = \frac{schedule\ length}{Socket_{fastest}} \tag{13}$$

- Efficiency in equation (14) is calculated by dividing the speedup ($Speed^{up}$) by the number of processors ($sockets_{num}$) used in each run.

$$Efficiency = Speed^{up} \times \frac{100}{sockets_{num}} \tag{14}$$

This work evaluated the performance of the algorithms on acyclic molecular graph presented in [7] and mentioned above in figure 1, using 3,5,7,8 sockets we observed an overall improvement in performance. The comparison between the HEFT, FCFS, GA, SA, GRASP and Tabu Search algorithms was based on the metrics schedule length, speedup, and efficiency presented in tables 2,3,4 respectively. All algorithms were implemented using Java programming language on the NetBeans platform, and the simulation was conducted on a computer with 1.80 GHz CPU and 3.89 GB RAM. To guarantee unbiased results, the experiment utilized the maximum number of function evaluations as the stopping criterion. A table summarizing the other parameter settings used in the experiment is provided below.

Table 1.parameter setting.

GRASP parameter:α	0.4
Initial Temperature T	200
Crossover probability (pc):	0.٧
Mutation probability (pm):	0.3
Tabu Search _tabu-Size	50
Maximum evaluations	٢٠٠٠٠

Table 2. Average Schedule length with (ms) of Algorithms

No. of Sockets	GRASP	HEFT Rank up	Tabu Search	SA	GA	HEFT-Rank down	FCFS
3	**158**	172	166	166	172	168	171
5	**119**	131	129	129	135	132	132
7	**109**	120	118	118	124	123	124
8	**106**	116	116	116	118	122	124

Table 3. Speed-Up of Algorithms

No. of Sockets	GRASP	HEFT Rank up	Tabu Search	SA	GA	HEFT Rank down	FCFS
3	**2.6519**	2.436	2.5241	2.5241	2.436	2.494	2.4503
5	**3.521**	3.1985	3.2481	3.2481	3.1037	3.1742	3.1742
7	**3.844**	3.4917	3.5508	3.5508	3.379	3.4065	3.379
8	**3.9528**	3.6121	3.6121	3.6121	3.5508	3.4344	3.379

Table 4. Efficiency (%) of Algorithms

No. of Sockets	GRASP	HEFT Rank up	Tabu Search	SA	GA	HEFT Rank down	FCFS
3	**0.3315**	0.3045	0.3155	0.3155	0.3045	0.3118	0.3063
5	**0.4401**	0.3998	0.406	0.406	0.388	0.3968	0.3968
7	**0.4805**	0.4365	0.4439	0.4439	0.4224	0.4258	0.4224
8	**0.4941**	0.4515	0.4515	0.4515	0.4439	0.4293	0.4224

This paper presents a set of algorithms recently published that demonstrate promising results in research. The objective was to explore the distinctions among these algorithms, ultimately enabling efficient management of heterogeneous computing environments and saving considerable time. Within the algorithms, HEFT Ranked up, HEFT Ranked down, and FCFS are classified as Static algorithms, while GRASP, Tabu Search, SA, and GA are categorized as heuristic algorithms. It is worth noting that the heuristic algorithms outperform the static algorithms in terms of performance, making them adaptable to changes in the problem domain. Conversely, the static algorithms handle the heterogeneous computing environment in a uniform manner, as they inherently maintain a constant approach regardless of variable conditions. However, HEFT Ranked up, HEFT Ranked down, and FCFS, as depicted in table 2, 3, and 4 respectively, exhibit subpar performance in generating high-quality schedules within this heterogeneous computing environment. This can be attributed to their reliance on specific task assignment orders, which restricts their effectiveness in this context.

Table 2 illustrates the superiority of the GRASP algorithm over various other optimization algorithms, HEFT Ranked up, Tabu Search, SA, GA, HEFT Ranked down, and FCFS, in terms of the average schedule length. The results demonstrate that the GRASP algorithm excels in generating schedules with shorter completion times on average, which is a crucial performance metric for scheduling algorithms. The remarkable performance of GRASP can be attributed to its efficient exploration of the solution space and its utilization of a combination of greedy constructive heuristics and randomized local search techniques to obtain high-quality solutions.

It is noteworthy that the GA performs comparatively poorer than other heuristic algorithms due to its reliance on crossover and mutation, which have already been demonstrated in the constructed solution. Moreover, both tabu search and simulated annealing exhibit better performance than GA but fall short of matching the effectiveness of GRASP. This can be attributed to the fact that GRASP applies local search techniques after constructing the initial solution, unlike tabu search and simulated annealing. The same conclusions can be drawn from figures 3, 4, 5, and 6, which further support the results. Additionally, this study indicates that as the number of sockets increases, the scheduling algorithm becomes more efficient in task assignment and significantly reduces the required completion time, as depicted in figure 7.

This study provides valuable insights into the effectiveness of different optimization algorithms for scheduling problems and highlights the importance of selecting appropriate algorithms based on the specific characteristics

of the problem at hand. Further research can investigate the applicability of the GRASP algorithm to other scheduling problems and explore ways to further enhance its performance. then, GRASP is more efficient than Tabu Search, Simulated Annealing and GA. Table 4 represents GRASP is 33% better then HEFT Ranked up, Tabu Search, SA, GA, HEFT Ranked down and FCFS in case of efficiency.

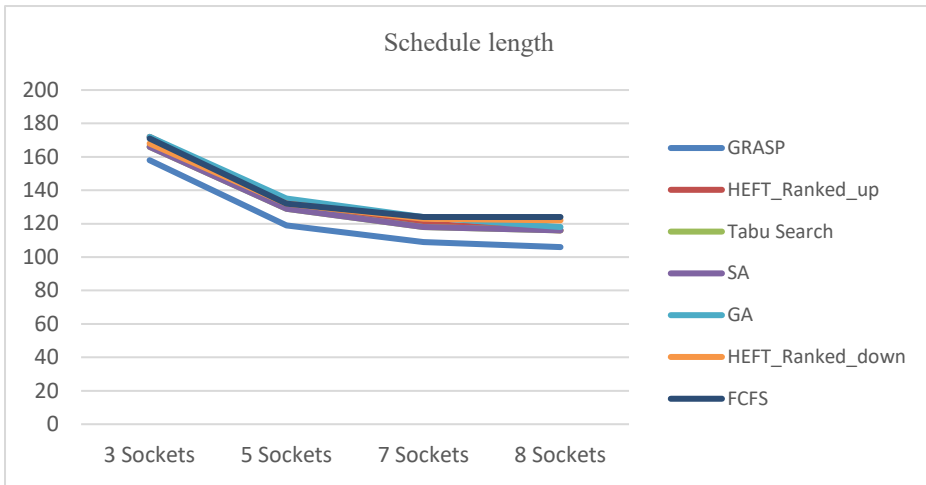

Fig. 3. Relation between numbers of sockets and schedule length

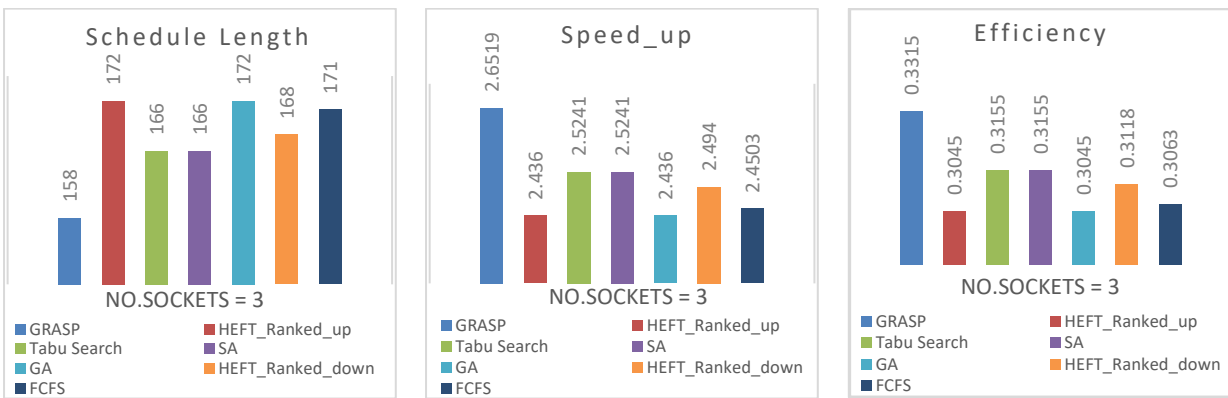

Fig. 4. Analysis of comparative algorithms with 3 sockets.

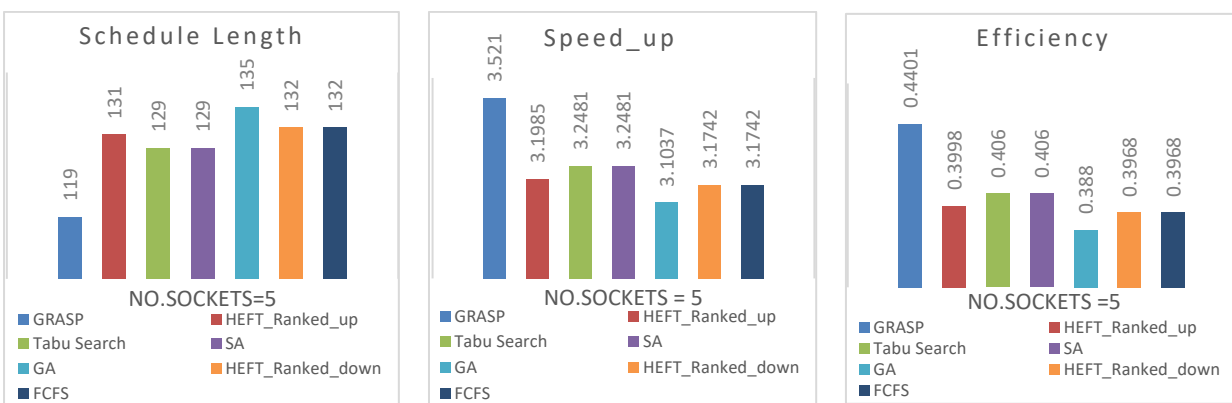

Fig. 5. Analysis of comparative algorithms with 5 sockets

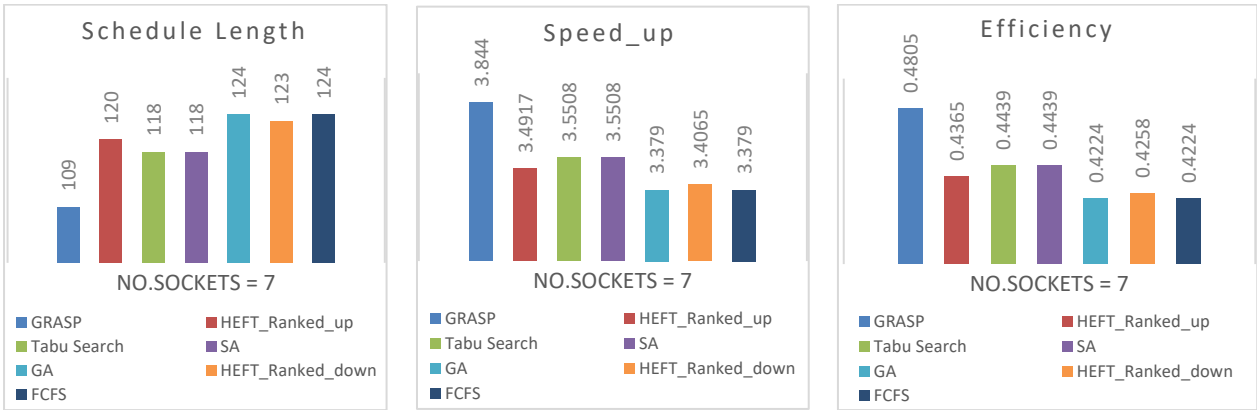

Fig. 6. Analysis of comparative algorithms with 7 sockets

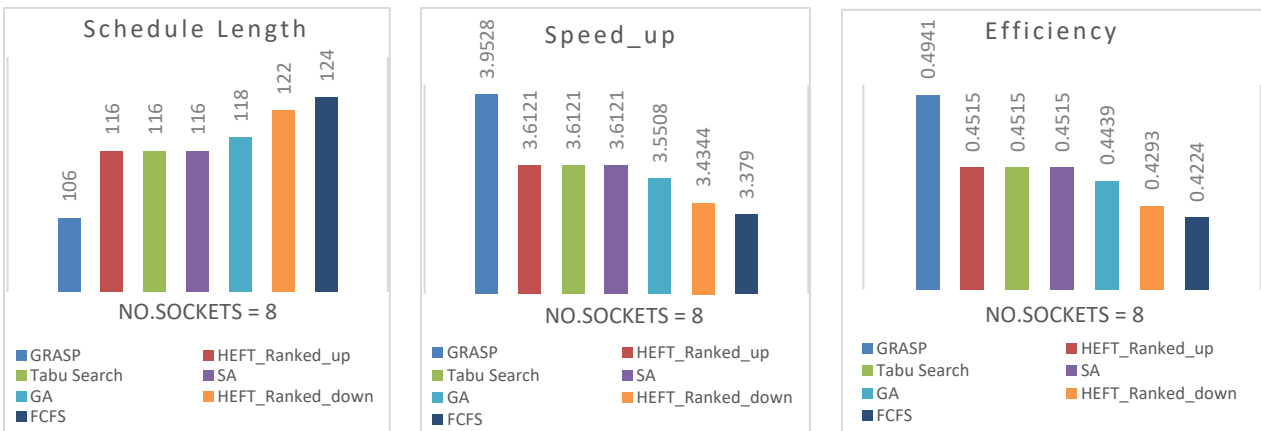

Fig. 7. Analysis of comparative algorithms with 8 sockets

VI. Conclusion and Future work

In this paper, we have explored six different algorithms, namely GRASP, Tabu Search, SA, GA, FCFS, and HEFT, across several parameters such as Schedule Length, Speedup, and Efficiency. These algorithms were applied to Molecular DAG in static task scheduling algorithms within a heterogeneous environment using three, five, seven, and eight sockets. Our results indicate that GRASP outperforms HEFT Ranked up, Tabu Search, SA, GA, HEFT Ranked down, and FCFS for all the parameters we considered. Additionally, increasing the number of sockets leads to improved results across all parameters. Nonetheless, this study shows that there is still considerable scope for improvement in all the algorithms in the existing literature. Although list scheduling is a vast research area, our study highlights the need for developing a technique that can generate an efficient priority list for tasks and an assignment-based algorithm to reduce the overall execution time (makespan). Future studies can examine the suitability of the GRASP algorithm for other scheduling problems and explore methods to further improve its performance.

References

[1] M. Homayun, N. T. Reza and H. S. Mirsaeid, "A hybrid meta-heuristic scheduler algorithm for optimization of workflow scheduling in cloud heterogeneous computing environment," *Journal of Engineering, Design and Technology,* 2022.

[2] B. V. J. and and V. Subrahmanyam, "Load and cost-aware min-min workflow scheduling algorithm for heterogeneous resources in fog, cloud, and edge scenarios," *International Journal of Cloud Applications and Computing (IJCAC),* 2022.

[3] R. Aron and A. Abraham, "Resource scheduling methods for cloud computing environment: The role of meta-heuristics and artificial intelligence," *Engineering Applications of Artificial Intelligence Journal,* 2022.

[4] S. Padhy, R. M. and and S. Kumari, "A novel algorithm for priority-based task scheduling on a multiprocessor heterogeneous system," *Microprocessors and Microsystems Journal,* 2022.

[5] W. A. Ahmed, S. Gulzar, N. M. Wasif, M. E. Ullah and N. Ramzan, "Energy Efficient Resource Allocation in Heterogeneous Computing Environments," *IEEE Access,* 2022.

[6] H. Topcuoglu, H. a. Salim, Wu and Min-You, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems,* 2002.

[7] G. Singh, G. Jasbir and a. Singh, "Improved Task Scheduling on Parallel System using Genetic Algorithm," *International Journal of Computer Applications,* vol. 39, pp. 17-22, 2012.

[8] A. &. Kumar and S. K. Bharti, "A review of static scheduling techniques for task allocation in heterogeneous distributed computing systems.," *International Journal of Advanced Intelligence Paradigms,* 2019.

[9] G. Li, Y. Li, W. Li and X. & Li, "Survey of dynamic task scheduling strategies in cloud computing.," *Journal of Network and Computer Applications,* 2018.

[10] D. K. Saxena, M. C. Govil and R. K. Gupta, "Performance Evaluation of FCFS and Priority Based Task Scheduling Algorithms for Heterogeneous Computing Systems," in *the International Conference on Computational Intelligence and Communication Networks*, 2015.

[11] Z. Liu, Q. a. and Li, X. a. Chen and Y. Chen, "Performance evaluation of task scheduling algorithms in heterogeneous computing systems: A comparative study," *Journal of Parallel and Distributed Computing,* 2022.

[12] Hafidha, I. H. and W. K. Mahdi, "A Hybrid Genetic Algorithm for Task Scheduling in Heterogeneous Computing Systems," *Journal of SN Computer Science,* 2021.

[13] Zhang and &. L. H. Q., "Scheduling problems in heterogeneous computing platforms: a survey of optimization techniques.," *Journal of Parallel and Distributed Computing,* 2018.

[14] A. Saad, A. and Kafafy, a. A.-E.-R. Osama and N. and El-Hefnawy, "A GRASP-Genetic Metaheuristic Applied on Multi-Processor Task Scheduling Systems," in *2018 13th International Conference on Computer Engineering and Systems (ICCES)*, 2018.

[15] Kirkpatrick, J. S, .. a. G. Vecchi, C. D. and and M. P., "Optimization by Simulated Annealing," *Journal of Science,* 1983.

[16] Z. Liu, K. and Li, X. and Zhang and X. and Yu, "Adaptive Simulated Annealing with Lévy Flights for Multimodal Optimization," *Journal of Applied Soft Computing ,* 2021.

[17] S. Abla, K. Ahmed, A. E. R. Osama and N. El-Hefnawy, "A GRASP-Simulated Annealing approach applied to solve Multi-Processor Task Scheduling problems," in *2019 14th International Conference on Computer Engineering and Systems (ICCES)*, 2019.

[18] F. Herrera and C. and León, "On the Effectiveness of Variable Neighborhood Search and Greedy Randomized Adaptive Search Procedures for Multi-objective Sustainable Supplier Selection," *Journal of Sustainability,* 2021.

[19] G. Dávila and F. and Herrera, "Design of a GRASP metaheuristic for the university course timetabling problem," *Journal of Combinatorial Optimization,* 2020.

[20] S. Nabli, M. and Gharsalli and M. and Ayadi, "A hybrid metaheuristic algorithm based on GRASP and variable neighborhood search for the flexible job-shop scheduling problem," *Journal of Engineering Applications of Artificial Intelligence,* 2019.

[21] A. Rawan, S. and Alotaibi, a. Almazyad, Alqarni and A. Abdulmohsen, "A hybrid Tabu search algorithm with a new promising neighborhood structure for solving the permutation flowshop scheduling problem," *Journal of Ambient Intelligence and Humanized Computing,* 2021.

[22] B. Ghomri, M. a. Moulaï, L. a. Abbas and M. a. Abdelhamid, "Multi-objective task scheduling using a Tabu Search algorithm in heterogeneous computing systems," *Journal of Soft Computing,* 2020.

[23] G. Jose, H. J. Ortega, J. a. Rico and Juan, "A Parallel Tabu Search Algorithm for Task Scheduling in Heterogeneous Computing Systems," *Journal of Grid Computing,* 2020.

[24] M. Tanha, M. H. Shirvani and A. M. Rahmani, "A hybrid meta-heuristic task scheduling algorithm based on genetic and thermodynamic simulated annealing algorithms in cloud computing environments," *Journal of Neural Computing and Applications,* vol. 33, pp. 16951--16984, 2021.

[25] A. W. a. Ahmed, S. G. a. Nisar, M. W. a. Munir, E. U. a. Ramzan and Naeem, "Energy Efficient Resource Allocation in Heterogeneous Computing Environments," 2022.