UNIVERSITY OF CALIFORNIA,
IRVINE

The Effects of System Characteristics on the Performance of Resource Allocation Algorithms in a Heterogeneous Environment

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Nazanin Ghasemian Moghaddam

Dissertation Committee:
Professor Jean-Luc Gaudiot, Chair
Professor Nader Bagherzadeh
Professor Phillip C-Y Sheu

2021

DEDICATION

To

My parents, who raised me and taught me right.

My brother, who always believed in me.

My friends, who were there for me.

My dear husband, for his love and support.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor and committee chair, Professor Jean-Luc Gaudiot, for his guidance and continual support, for his patience and understanding, for his genuine commitment to his students. Without his generous assistance and involvement, this work would not have been possible.

I would like to thank my committee members, Professor Nader Bagherzadeh and Professor Phillip C-Y Sheu, for their valuable comments on my research work.

In addition, I wish to thank Dr. Tongsheng Geng, for the invaluable discussions and for providing much appreciated, critical feedback to this work.

I would also like to thank all of my colleagues in the PArallel Systems & Computer Architecture LAB (PASCAL), for their support over the past six years.

Special thanks to Dr. Miroslav Penchev, for proofreading this manuscript.

VITA

Nazanin Ghasemian Moghaddam

2010	B.S. in Computer engineering, IRAN University of science and Technology
2013	M.S. in Computer Architecture, Shahid Beheshti University, IRAN
2015-21	Teaching Assistant, Graduate School of, University of California, Irvine
2015-21	Graduate Student Researcher, University of California, Irvine
2021	Ph.D. in Electrical and Computer Engineering, University of California, Irvine

FIELD OF STUDY

Computer Architecture, Heterogeneous Scheduling Algorithms

ABSTRACT OF THE DISSERTATION

The Effects of System Characteristics on the Performance of Resource Allocation Algorithms in a Heterogeneous Environment

by

Nazanin Ghasemian Moghaddam

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2021

Professor Jean-Luc Gaudiot, Chair

Resource allocation in heterogeneous environment where machines provide different computational capabilities to different task types is a classic problem, which still attracts significant attention. The performance of a system is highly related to the approach used to assign its resources to the tasks. Current development in the fields of hardware and software encourages using parallelism to fully exploit the computational potential of the system. With current applications, consisting of hundreds of tasks to be executed on large-scale interconnected computing units, the heuristic approach is the only practical solution. Selecting an effective scheduling algorithm that is compatible with the system characteristics is crucial for maximizing the performance. There are numerous resource allocation algorithms reported in works in literature, each designed for a problem space with specific properties. Randomly generated Directed Acyclic Graphs (DAGs) are widely used for comparing scheduling algorithm performance. For a fair comparison, the DAG set should be uniformly distributed over all possible DAG types. Moreover, to select a suitable heuristic for

a problem one must be familiar with the system properties. Therefore, knowing and being able to quantify the characteristics of a system will be helpful in several ways: to select fitting scheduling, to generate a DAG set that contains all task types, and to examine the effect of different properties on the efficiency of the algorithm.

This dissertation work introduces two scheduling algorithms, Parallel Heterogeneous Earliest Finish Time (PHEFT), whose objective is to share resources between two tasks with the same priority, and Optimize All Path (OAP), which is designed based on a concept of improving the critical path execution time. Experimental results show PHEFT performs significantly better than other popular greedy algorithms in a consistent environment. OAP test results show the same performance as some leading heuristics, while having high time complexity.

A major part if this research work focuses on developing and implementing a Random DAG Generator with a wider range of input parameters, compared to other available generating tools. This DAG generator offers advantages in characterizing the workload and ETC matrix, as well as generating a non-biased almost uniformly distributed DAG set, and easy tuning of the parameters to generate any workload/ETC matrix with desired characteristics. The generated DAGs by this tool, are used to study the correlation between each of the environment properties, or task types and the operation of heuristics. The number of levels in a DAG, variation on the number of nodes in each level, machine heterogeneity, and computation to communication cost ratio are some of the properties, which show a higher correlation to the performance of a scheduling algorithm. This information can be used by the system manager to assess the efficiency of a scheduling algorithm for a specific workload/environment pair.

CHAPTER 1

INTRODUCTION

With the increasing need for smaller, faster, more reliable, energy efficient computing systems, groundbreaking research work has pushed the frontiers of both hardware and software to meet these growing demands. At present the use of multicore processors in multiprocessing systems, which are connected through high-speed links to create large-scale grid infrastructures, servers, or data centers is the norm. The computational units, making up these systems, could consist of different processors on an individual computer, different interconnected computer machines in a server room, or many machines distributed over a network. Such type of system can be used to run applications in parallel, whose execution time on a single processor, in a sequential manner, would be significantly longer.

While multiprocessing systems are undergoing continuous improvements in both hardware and software, the complexity of the applications running on these systems is also increasing. There is a high demand for data and computationally intensive applications in diverse areas such as Bioinformatics, High Energy Physics, Climate prediction, etc. [1]. Many of these applications are composed of several tasks, some of which may require meeting deadlines due to either real-time nature, or dependencies between the tasks that need them

to be finished in a timely manner. To satisfy variety of quality of service, applications are more frequently designed to run in parallel. Using parallel processing to handle separate tasks of an application on multiple processors, helps reduce the overall execution time of the program and meet the quality of service required by the system manager.

The quality of service targets a great range of evaluation metrics from meeting the deadline to energy consumption budget, and other factors like robustness, resiliency to errors, and utilization. One of the most important factors to evaluate the efficiency of a system is the execution time of the applications on that system. With shorter execution time it is more likely to meet the deadline for real time system, there is a higher confidence radius in case an error occurs, and shorter execution time leads to lower total energy consumption.

With applications designed to run in parallel and distributed systems made up of various machines, the problem of assigning each task to a processor falls in resource allocation and scheduling categories. Scheduling algorithms take a group of tasks, which could be dependent or independent, and the specification of available resources, and determine assignment of each task to a processor while meeting all dependencies between tasks and aiming to improve the optimization goal. The final execution time of a scheduling scenario is called makespan [2].

A task graph is a model representing an application (a set of tasks with possible dependencies, which require an order of execution). Directed Acyclic Graph (DAG), $G = (V, E)$ is a type of task graph where each node ($v \in V$) is a task or group of sequential instructions, which must be executed on one processor, and edges between nodes ($e \in E$) correspond to the dependencies between tasks [3]. Task scheduling can be defined as the process of allocating different tasks of an application, to different computational units in the

system. For a group of tasks $\{t_1, t_2, \ldots, t_n\}$ and a set of machines $\{m_1, m_2, \ldots, m_m\}$ a possible scheduling is defined as:

$$\{t_1 : [m_{j\epsilon[1,m]}, s_{1,j}], \ldots, t_i : [m_{j\epsilon[1,m]}, s_{i,j}], \ldots, t_n : [m_{j\epsilon[1,m]}, s_{n,j}]\}$$

For each task there is an assignment to a machine and the start time of the task on that machine. Usually alongside the set of tasks and machines there is another piece of information provided to the scheduling algorithm: the *Estimated Time to Compute* (ETC) matrix. ETC is a $n \times m$ matrix, where the value of $ETC(i, j)$ is the estimated execution time of task i on machine j. Based on the assignment each task will have an estimated finish time $\{f_1, f_2, \ldots, f_n\}$, where f_i is:

$$f_i = s_{i,j} + ECT(i, j)$$

Makespan, the final execution time of a set of tasks, is $Max(f_1, f_2, \ldots, f_n)$, and the assignment which results in lower makespan is more desirable.

$$Goal \rightarrow minimize\ (\ Max(f_1, f_2, \ldots, f_n))$$

This problem in general is a NP (nondeterministic polynomial time)-complete [4], in which heterogeneity of the computing units and communication cost between the tasks adds to its complexity. Therefore, the solutions to the scheduling problem are heuristics, whose objective is to find near optimal solutions in an acceptable time.

There are great number of proposed heuristics for the scheduling problem which may have different approaches based on different (1) task type, (2) processing units, (3) provided data, and (4) optimization goal. With numerous different algorithms to choose from, it is important to have means of measuring and comparing their performance. There are some important aspects to focus on with regards to scheduling algorithms, when comparing them:

- **Definition of performance**. There are several different metrics used to define the

performance of a scheduling algorithm on a system. The most popular one is the makespan. Thus to say that algorithm A is better than algorithm B, the former should outperform the later on average over several test cases. Therefore, it is important to have a performance measurement metric, which is fair and independent of the inputs (task, and machine sets).

- **Characterizing tasks and computing environment**. To select an algorithm, which addresses the specific requirements of a problem, one approach is to categorize systems based on the task/machine characteristics and use the algorithms that work better for the selected special category. Properties of a system are important to know and quantify, because they can be used to design tests that are unbiased and consider all possible types of task/machines.

Multiple published studies [5]–[8] show that different task/machine types affect the performance of the scheduling algorithm. Therefore, quantifying the characteristics of tasks and machines, and knowing the strength and limitation of scheduling algorithms can help the system manager to select the best algorithm for each circumstance.

The rest of this dissertation is organized as follows. Chapter 2 presents a literature review of different leading scheduling algorithm in the field. This work aims to categorize them from different aspects and summarize their performance analysis, which are reported in other research works. Chapter 3 introduces two new scheduling algorithms, Parallel Heterogeneous Earliest Finish Time (PHEFT), and Optimize All Path (OAP), and compares them with some of the classical algorithms. Chapter 3 presents a detailed discussion on characterizing and quantifying the inputs of the scheduling algorithm (tasks and computing

units), and describes strategies of using them to establish a fair framework to compare the performance of different heuristics. Different approaches to generating an ETC matrix and a random task graph, and how these different approaches affect the performance of the scheduling algorithm are also discussed in Chapter 4. In addition, a new random DAG generator is introduced in this work, which makes it easier to tune the DAG properties and generate a wider range of task graphs. Then using the randomly generated DAGs, the effects of system characteristics on the performance of the scheduling algorithms were explored. Finally, Chapter 5 presents conclusions and future work.

CHAPTER 2

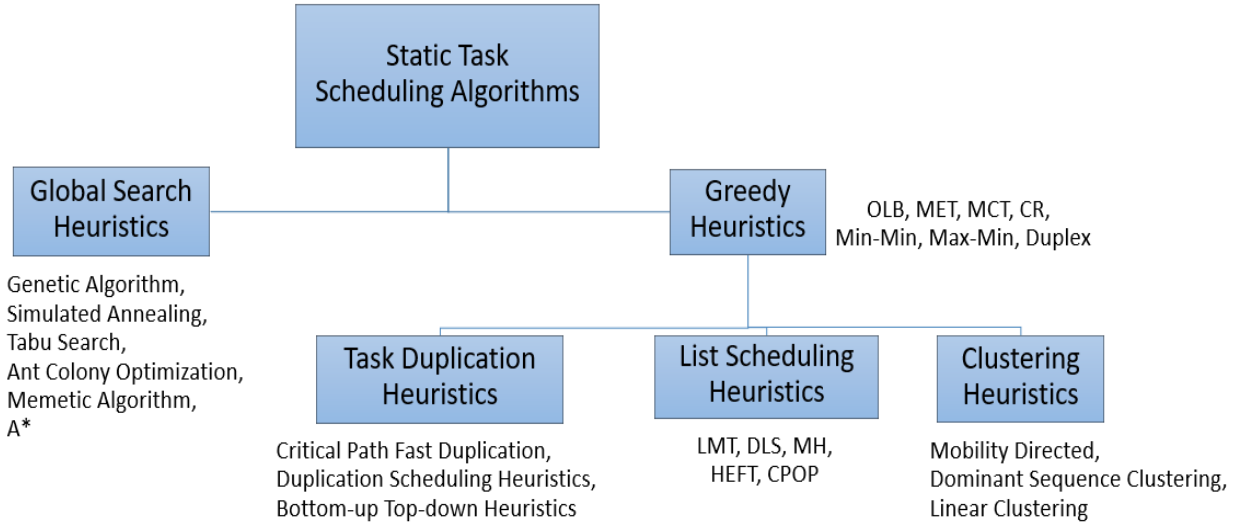
Scheduling a group of tasks in a heterogeneous environment in a manner resulting in acceptable total execution time is a crucial task, due the constant demand to increase the system performance, decrease the energy consumption, reduce errors, and design robust systems. With different optimization goals, the main question is always how to assign tasks to different resources to improve the final goal. This is a NP-complete problem, in which increasing the number of tasks and number of processors makes the possible outcomes too large count. To this end, numerous published research studies propose a solution through heuristics. Exploring all scheduling algorithms proposed in the literature is beyond the purpose of this dissertation. Therefore, this work will focus of some of the more popular ones, which have been studied more extensively because of their novelty or their performance.

Scheduling algorithms can be static or dynamic. In static scheduling there is no change in the task/processor assignment during the execution of the application. While in dynamic, the scheduler keeps receiving feedback from the system and uses this new information to make better decision, thus the scheduling can change during the execution. Since there are many cases in which the computation network has historical data about the execution cost of the applications, it is safe to assume that in majority of cases the static

scheduling would be enough. A concise review of the large body of published works related to static scheduling, resource management, and resource allocation algorithms, will follow in this chapter.

Static heuristics can be classified based on their approach in two main groups of Greedy heuristics and Global Search heuristics. Some published works use Metaheuristics as another group in their classification [9], but since that specifically refers to some modifications on Global Search heuristic algorithms, it will not be considered as a separate group in this dissertation work. Greedy Heuristics can further be classified as List-scheduling heuristics, Clustering heuristics, and Task duplication heuristics. Figure 2.1 shows the classification of static task scheduling algorithms.

Figure 2.1: Static Task Scheduling Algorithms Classification



GREEDY HEURISTICS

Greedy heuristics performs well and find solutions relatively fast. They sacrifice finding the optimal solution to get faster results. Opportunistic Load Balancing (OLB),

Minimum Execution Time (MET), Minimum Completion Time (MCT), Contention Resolution (CR), Min-min, Max-min, and Duplex are some of the more general examples of Greedy Heuristics.

1. Opportunistic Load Balancing Algorithm

The Opportunistic Load Balancing (OLB) algorithm assigns each task, in arbitrary order, to the first available processor regardless of the task's execution time. The idea behind OLB is to keep all processors as busy as possible. This algorithm is very easy to implement and has a low time complexity but since it is not considering the execution cost of the tasks on the processors the result scheduling could be very poor [10].

2. Minimum Execution Time Algorithm

The Minimum Execution Time (MET) algorithm assigns each task in arbitrary order to the machine which gives the best execution time. This algorithm does not take into consideration the availability of the processors. Its goal is to assign each task to its first choice. Therefore, in heterogeneous consistent systems, in which one machine is the best choice for all tasks, the MET would result in load imbalance across resources [10].

3. Minimum Completion Time Algorithm

The Minimum Completion Time (MCT) algorithm takes tasks in arbitrary order and assigns them to a processor with the minimum expected completion time. The expected completion time of a task on a machine is the execution time plus the start time, $max(task\_ready\_time, machine\_available\_time)$ of the task on that machine. MCT combines OLB and MET and tries to prevent the situations in which each of them performs poorly [10].

4. Contention Resolution Algorithm

The Contention Resolution (CR) algorithm [11], first determines the best assignment for each unscheduled task at each iteration and assigns those tasks with no rival for their best options. Then among the remaining tasks, which compete for the same resource, the one with the largest difference between the best and the second-best execution time gets scheduled. In other words, the task which suffer more from not getting its best choice is the most critical task and needs to get assigned first. The concept of suffrage was introduced by Kim et al. [12] and Maheswaran et al. [13].

5. Min-Min Algorithm

Min-Min algorithm [14]–[16] is another member from the two-phase heuristics class. This algorithm keeps track of all unmapped tasks which are ready (i.e. all their predecessors are done and the data from them are received). Min-Min algorithm finds the minimum completion time for each task in the ready list. Then the task with the minimum completion time among all tasks is selected an assigned to the corresponding machine (the one which results in minimum completion time). This process repeats until there is no unmapped task [17]. Min-Min maps the tasks in the order that makes least change in the availability of the processors. Therefore, the percentage of tasks which get assigned to their first choice would be high [16]. This is exactly the opposite of the Max-Min algorithm, which is discussed next. The pseudocode for Min-Min algorithm is listed below.

Algorithm 1: Pseudocode for Min-Min Algorithms

Generate the ready list

While there is an unmapped task in the ready list

- **For** each task in the list find the machine which minimizes the execution cost
- Among all task/machine pairs select the one with the least execution cost
- Schedule the task, and update the ready list

6. Max-Min Algorithm

The first phase in the Max-Min algorithm [14]–[16], like the Min-Min, is to make a list of ready tasks and find the machine which gives the lowest completion time for each task in the list. Then, the Max-Min algorithm selects the task/machine pair that have the maximum completion time among all tasks in the list. This process repeats until all tasks are mapped. The key concept of the Max-Min algorithm is mapping the task with the longer execution time to its first choice, results in reduced overall penalty of assigning next tasks to their second or other choices. Max-Min and Min-Min algorithms may each result in better quality solutions under different circumstances [8].

Algorithm 2: Pseudocode for Max-Min Algorithm

Generate the ready list

While there is an unmapped task in the ready list

- **For** each task in the list find the machine which minimizes the execution cost
- Among all task/machine pairs select the one with the highest execution cost
- Schedule the task, and update the ready list

7. Duplex Algorithm

Duplex algorithm [10], [14], [15] performs both Min-Min and Max-Min, and uses the

better solution. Since Min-Min and Max-Min both have low time complexity, this algorithm can find the best of these two with a negligible overhead.

LIST SCHEDULING ALGORITHMS

List scheduling algorithms or sorting algorithms are a group of Greedy heuristics, which follow two steps to assign resources to tasks. The first step is to find an order for the tasks based on their priorities and the second step is to take a task from the list and assign each selected task to an available resource, thus resulting in minimizing the predefined cost function. The main problem with this group of scheduling algorithms is that the static priority assignment, which takes place in the first step, may not always be a good indicator of the relative importance of the tasks. Some of the examples of this group are Heterogeneous Earliest Finish Time (HEFT), Critical Path on a Processor (CPOP), Levelized Min-Time (LMT), Dynamic Level Scheduling (DLS), and Mapping Heuristic (MH) algorithms.

1. Heterogeneous Earliest Finish Time Algorithm

The Heterogeneous Earliest Finish Time (HEFT) algorithm [18] is classified under list scheduling group. This algorithm first determines an order for the tasks in the DAG using the upward rank algorithm. The upward rank of each task is the average execution time of the task over all processors plus the maximum rank of its children added by the average communication time between the two tasks. Eq. 1 gives the formula for upward rank of task i, where $\overline{w_i}$ is the average execution cost of task i, and $\overline{c_{i,j}}$ is the average communication cost between task i and j (one of the successors of task i).

$$rank_u(i) = \overline{w_i} + \max_{j \in \text{succ(i)}} (\overline{c_{i,j}} + rank_u(j)) \qquad \text{Eq. 1}$$

Upward rank for the end task which has no children is given in Eq. 2 by the average execution cost of it on all available machines.

$$rank_u(n_{exit}) = \overline{w_{exit}} \quad \text{Eq. 2}$$

Then one can start from the end tasks and move upwards to find the rank of every task in the graph. In the second phase of the algorithm tasks are taken from the list in decreasing order of their rank and assigned to a processor which results in minimum execution time. If two tasks have same upward rank, one gets selected randomly, or another strategy is used to select the task that its immediate successor has the higher rank. For most of the task scheduling algorithms the earliest available time for a processor is the time when the last assigned task on it is completed. However, HEFT algorithm considers the idle periods on the processor for the assignment. To find the best time slot for a task the algorithm needs to consider the ready time of the task, which is the time when all its predecessors are done and if they were assigned to another processor the time needed for sending data to current processor has passed. The ready time of the task ($task_{ready\_time}$) must be greater or equal to the start of the time slot ($time\_slot_{start}$) and the difference between end of the time slot and max($task_{ready\_time}, time\_slot_{start}$) must be greater or equal to the task execution time on that processor. The time complexity of the HEFT algorithm is $O(e \times q)$ where e is the number of edges in the DAG, and q is the number of processors. Results published by Topcuoglu et al. [18], show that on average HEFT performance is better compared to some older scheduling algorithms. The pseudocode for HEFT algorithm is listed below.

Algorithm 3: Pseudocode for HEFT

Task prioritizing phase

- Set the computation cost of tasks and communication cost of edges with mean values
- Compute $rank_u$ for all tasks by traversing graph upwards, starting from the exit task

$$rank_u(i) = \overline{w_i} + \max_{j \in \text{succ(i)}} (\overline{c_{i,j}} + rank_u(j))$$

$$rank_u(n_{exit}) = \overline{w_{exit}}$$

- Sort the tasks in a scheduling list by non-increasing order of $rank_u$ values.

Processor selection phase

while there are unscheduled tasks in the list **do**

- Select the first task in the list and find the best machine for it that minimizes the execution cost

2. Critical Path on a Processor Algorithm

The Critical Path on a Processor (CPOP) algorithm proposed by Topcuoglu et al. [18], has the same two phases; task prioritizing and processor selection phase like HEFT. However, CPOP uses different strategies for both raking tasks and assigning processors to each task. Here the algorithm calculates both upward rank and downward rank for the tasks. Upward rank is same as the one defined by HEFT. Downward ranking of each task is the maximum downward rank of its predecessors plus the average computation cost of the predecessors plus the average communication cost between the current task and its predecessors. The formula for downward rank of task i is given by Eq. 3, where the $\overline{w_j}$ is the

average execution cost of task j, and $\overline{c_{i,j}}$ is the average communication cost between task i and j (one of the predecessors of task i).

$$rank_d(i) = \max_{j \in \text{pred(i)}} (rank_d(j) + \overline{w_j} + \overline{c_{i,j}}) \quad \text{Eq. 3}$$

Downward rank for the entry task, which has no parent is set to zero. Then the priority of each task is the summation of its upward rank and downward rank. The CPOP algorithm aims to dedicate the fastest machine to the tasks in the critical path and then use other resources to schedule other tasks. The length of the critical path $|CP|$ is the sum of the execution cost of the tasks on the path and the communication cost between them. This is the *lower bound* on the makespan of the scheduling algorithm. The critical path length is equal to the priority of the entry task. Next task on the critical path is the immediate successor of the entry task, which has the highest priority value. This process continues until all tasks on the critical path are found. In the processor selection phase, the algorithm first decides to assign the fastest processor to the critical path. Then at each iteration if the highest priority task is on the critical path, it will be scheduled on the critical path processor, otherwise it will be scheduled on a processor which gives the earliest finish time. The time complexity of the CPOP algorithm is $O(e \times q)$, where e is the number of edges in the DAG and q is the number of processors. However, results show that CPOP performance is not as good as HEFT algorithm. The pseudocode for the CPOP algorithm is listed below.

Algorithm 4: Pseudocode for CPOP Algorithm

Task prioritizing phase

- Set the computation cost of tasks and communication cost of edges with mean values
- Compute $rank_u$ and $rank_d$ for all tasks

$$Priority(i) = rank_u(i) + rank_d(i)$$

- Sort the tasks in a scheduling list by non-increasing order of $priority$ values.

Processor selection phase

- Find the *critical path* and *critical path processor*

while there are unscheduled tasks in the list **do**

if the task is in the *critical path*

schedule it on the *critical path processor*

else

schedule it on the best machine which minimizes *EFT*

3. Levelized Min-Time Algorithm

Levelized Min-Time (LMT) algorithm [19] consists of two phases. In the first phase, the algorithm groups those task that can be executed in parallel on the same level. Thus, tasks in the lower levels would be ready for execution before higher levels and have higher priorities. At each level the task with higher execution cost gets assigned first. Each task is assigned to a processor, which results in minimum completion time (communication costs are considered). The time complexity is $O(v^2 \times p^2)$, where v is the number of tasks and p is the number of processors [11].

4. Dynamic Level Scheduling Algorithm

Dynamic Level Scheduling (DLS) algorithm [20] selects the available task and processor pair at each level, which maximize the dynamic level value. The dynamic level value for task i on processor p is the upward rank value for task i minus the earliest start time of the task i on processor p ($DL(i,p) = rank_u(i) - EST(i,p)$). In order to find the upward rank with DLS algorithm, researches have demonstrated using the median execution cost of a task on the processors, without considering the communication costs. The general form of this algorithm has an $O(v^3 \times p)$ time complexity, where v is the number of tasks in the DAG and p is the number of processors [11].

5. Mapping Heuristic Algorithm

The mapping heuristic (MH) algorithm uses the static upward rank to set the priorities of the tasks. This upward rank does not contain the communication cost, while the computation cost of a task on a processor is calculated by the number of instructions in the task divided by the speed of the processor. The MH algorithm does not assign tasks to an idle time slot between two tasks already scheduled. The time complexity is equal to $O(v^2 \times p^3)$, where v is the number of tasks and p is the number of processors [11].

CLUSTERING HEURISTICS

The objective of Clustering Heuristics algorithms is to map the tasks in each graph to some clusters. Tasks in each cluster are then assigned to a processor. The common goal is to minimize the communication cost and hence increase the performance. Clustering algorithms have high time complexity and require additional steps before generating the

final scheduling. They usually start with unlimited number of clusters and then try to merge some of the clusters by moving tasks around to fit the limited number of processors. The final step consists of scheduling the tasks in a cluster and finding an order for their execution. Some examples of this group of algorithms are Mobility Directed (MD) [21], Dominant Sequence Clustering (DSC) [22] and Linear Clustering (LC) algorithm [23].

1. Mobility Directed Algorithm

Mobility directed (MD) algorithm [21] has two phases. In the first phase the algorithm assigns some mobility values to each task and schedules them on virtual processing elements (PEs) based on their mobility values. Then in the second phase it maps these virtual PEs to actual processors. This algorithm, which was defined on a homogenous system, first finds the earliest start time ($T_s(i)$) and the latest start time ($T_l(i)$) for each task, based on ASAP (as-soon-as-possible) and ALAP (as-late-as-possible) bindings respectively. The relative mobility of each task i is defined by $relative\ mobility = (T_l(i) - T_s(i))/w(i)$, where $w(i)$ is the execution time of the task. Then among tasks with the least relative mobility, the algorithm selects the one all of whose predecessors are already scheduled (if there is more than one, selection occurs in arbitrary order) and assigns it to the first available PE. The number of PEs are unlimited, therefore if the first one is not available, the task is assigned to the second one, and so on. The communication cost between this task and all other tasks, which are scheduled on the same PE is changed to zero. Followed by updating relative mobility and repeating this until there is no task left in the list.

The second phase of the algorithm consists of assigning the virtual PEs to the real processors in a way to minimize the total communication traffic between them. If the number

of PEs is same as the number of processors the task is trivial, otherwise the algorithm needs to reassign some tasks to other PEs. To do so, the algorithm usually starts with an initial assignment and then improves it iteratively. The time complexity of this algorithm is $O(v^3)$ where v is the number of tasks. The pseudocode for DM is given by Algorithm 5.

Algorithm 5: Pseudocode for DM Algorithm

Phase 1: Scheduling

Calculate *relative mobility* for all tasks

While there is an unassigned task

Make a list of all tasks with the least *relative mobility* values (l)

For all tasks in l which all have no unscheduled parent

- Assign it to the first available PE
- Update communication cost between this task and all other tasks on the same PE
- Update *relative mobility* values

Phase 2: Mapping

Create an initial assignment

While there is a PE to assign

Change the scheduling of a random task

If the new scheduling improves the performance

Replace the old scheduling

---

2. Dominant Sequence Clustering Algorithm

The Dominant Sequence Clustering (DSC) algorithm [22] is based on a dominant sequence of a graph which is the critical path of a partially scheduled DAG. The algorithm starts with calculating b_level and t_level for all tasks. The b_level of a node is the length of the longest path from it to an exit node, and t_level of node i is the length of the longest path from an entry node to i. The priority of each task is the summation of its b_level and t_level. At any given time, there are two groups of tasks; examined group (EG) and unexamined group (UEG). DSC selects a free task (a task whose predecessors are all already examined) and a partially free task with highest priority from the unexamined group. A task is partially free if it is unexamined but at least one of its predecessors (not all of them) has been examined. At each iteration the algorithm tries to minimize the t_level of the free task by zeroing multiple incoming edges to the free task. This process continues until all tasks are examined. A distinctive feature of DSC is that the t_level of a node is computed incrementally and the b_level does not change until the node is scheduled [24]. These decisions are taken in order to lower the time complexity which is $O\big((v+e)\log v\big)$, where v is the number of nodes and e is the number of edges. The pseudocode for DSC is shown by Algorithm 6.

Algorithm 6: Pseudocode for DSC Algorithm

Phase 1: Scheduling

$EG = \emptyset,\ \ UEG = V$

Compute b_level for each node and set $t_{level} = 0$ for all free nodes

While $EG \neq \emptyset$

$n_x = free\ task\ with\ the\ highest\ priority$

$n_y = partially\ free\ task\ with\ the\ highest\ priority$

If $prio(n_x) > prio(n_y)$

- Call minimization procedure

Else

- Call minimization procedure while guaranteeing effective reduction of t_level of n_y in future steps

Update the priorities

Phase 2: Mapping

Create an initial assignment

While there is a cluster to assign

Change the scheduling of a random task

If the new scheduling improves the performance

Replace the old scheduling

---

3. Linear Clustering Algorithm

The Linear Clustering algorithm [25] is a critical path based clustering method. In the scheduling phase, the algorithm identifies the tasks on the critical path, assigns all of them to a cluster and removes them from the DAG. This process repeats until there is no task in the DAG. Subsequently, in the mapping phase each cluster is assigned to a processor [26].

TASK DUPLICATION HEURISTICS

Task Duplication Heuristic algorithms duplicate some of the tasks in several processors to reduce the communication cost between them. The main difference between the algorithms in this group is their strategy to select the tasks to duplicate. Just as clustering

heuristics, most of the algorithm in this group also target unlimited number of processing units and have high time complexity, which makes them impractical under most scenarios. Some of the examples of this group are Critical Path Fast Duplication (CPFD), Duplication Scheduling Heuristic (DSH), and Bottom-up Top-down Duplication Heuristic (BTDH).

1. Critical Path Fast Duplication Algorithm

The Critical Path Fast Duplication (CPFD) algorithm [27] classifies the tasks in the graph to three categories. CPN are the tasks on the critical path, IBN are tasks that have an edge to a task on CPN group, and OBN are tasks that are neither CPN nor IBN. The CPFD algorithm aims to optimize the execution cost of the critical path, because the critical path execution time defines the makespan of the scheduling. CPFD starts tasks on the CPN as early as possible, and keeps duplicating tasks in the IBN and CPN group, which leads to other tasks on the CPN group. This algorithm was designed for a set of homogeneous processors and its time complexity is equal to $O(v^2 \times e)$ for scheduling a DAG with v tasks connected with e edges [11].

2. Duplication Scheduling Heuristic

The Duplication Scheduling Heuristic (DSH) algorithm [28] combines list scheduling with task duplication. This algorithm first assigns each task a priority value based on their b_level (b_level of a node is the length of the longest path from it to an exit node). It then takes in consideration tasks on descending order of their priority and determines if the task needs its predecessors (one or all of them) duplicated or not. The parent tasks selected for duplication are assigned to the idle time slots of available processors. This algorithm also was designed for a homogeneous set of processors, therefore it does not have any processor

selection phase [27]. The time complexity of DSH is $O(v^4)$, where v is the number of tasks to schedule [17].

3. Bottom-up Top-down Duplication Heuristic

The Bottom-up Top-down Duplication Heuristic (BTDH) algorithm [29] is very similar to the DSH algorithm. DSH duplicates ancestors of a task even if there is no free time slot to fill up, resulting in increase of the start time of the task [24], [30]. BTDH, on the other hand, only duplicates tasks when it leads to earlier start time for the child task. The time complexity of BTDH is $O(v^4)$, where v is the number of tasks to schedule [11].

GLOBAL SEARCH HEURISTICS

Another group of scheduling algorithms are global search heuristics. These algorithms usually start with a set of solution (or just one solution in case of simulated annealing), then use some random based operators to modify and evaluate them at each iteration. The search continues until some predefined criterion is reached. The most popular ones in this group are those which are based on Genetic Algorithm. Simulated Annealing, A star, and Tabu Search are some other less common examples of this groups. The global heuristic algorithms in general result in better solutions but they take more time to find the solution.

1. Genetic Algorithm

Genetic Algorithm (GA) is a bio-inspired algorithm which is based on the idea that the fittest survive. This algorithm was first introduced by Goldberg [31] and Whitley [32], however, since then numerous adaptations of it have been used to solve various problems.

This algorithm starts with a set of solutions. Generally, this initial population is generated randomly, however there are some published works, which have added solutions from greedy heuristic to the initial set [11], [33]. Each solution in the population, called chromosome, is a complete scheduling and each element in a scheduling, called gene, specifies the resource allocation for each task. For example, in a vector of length N, the ith element is the processor id that is assigned to the ith task. The quality of the population and each solution is defined by a fitness function, which is based on computation cost, reliability, execution time, robustness, etc. Usually, the two fittest chromosomes at each iteration are selected for the crossover and mutation process to generate new offspring. The new generated solutions are also analyzed and added to the population if their fitness value is higher than a threshold. There are different approaches to keep the population size same. Some just delete the least fit chromosome, and some keep the extra chromosomes separately in a place called graveyard to possibly use them later. This process of generating new offspring continue and at each step the whole population improves until the algorithm reaches the stop point. The stop point could be the number of iterations [34], the improvement percentage between two iterations, or not adding new offspring (because their fitness is low) for several iterations.

The crossover operation also has several variations. In general, there are two scheme, one-point, or two-points crossover. In one-point crossover one gene is randomly selected and the value of that gene (the assigned processor) are exchanged between the two parents to generate the offspring [34]. One form of two-points crossover is so-called reduced surrogate procedure [11], [32]. This procedure first finds all different genes in the two parents, then randomly selects two of them and exchanges the value of the first gene from

first chromosome with the second gene from the second chromosome, and doing the same for the other pair. In the work by Khajemohammadi et al. [35], the genes are sorted based on their level and crossover, and then mutation is performed at each level.

The mutation process can be done on the new offspring or some random chromosome. Then the gene for mutation is chosen randomly or using some mutation rate probability [11]. Once the mutation gene is selected its value will change randomly to another processor, or the one which is not overloaded or has better execution time. Some other published works have employed their own unique design for crossover and mutations [34], [36], [37].

Some also use a form of post-mutation operator [17], [38]. For example, after each mutation the algorithm first finds the machine that has the lowest probability to meet the optimization goal, then an application is selected on this machine which if removed and assigned to another machine improves the performance. The application machine pair that results in greater improvement is selected and the assignment changes.

Nasonov et al. [39] combined HEFT with genetic algorithm, using GA to find the near optimal schedule and HEFT for rescheduling in case of some failure in the system. Several reports have demonstrated using multi-objective genetic algorithm to optimize performance and energy consumption [40]–[43].

Genetic algorithm-based scheduling usually gives better results compared to other forms of scheduling algorithms, but they take relatively long time to generate the result. Furthermore, finding the optimal values for the parameters used in this algorithm is usually based on the experiment and varies for each workload. Some of these parameters are the population size, cross over rates, and mutation rates.

The pseudo code for general form of Genetic Algorithm is listed below.

Algorithm 7: Pseudocode for GA

Generate the initial population and calculate their fitness

While the stopping criteria not met

- Select parents
- Do cross over
- Do mutation
- Add new offspring to the population and remove the less fit ones

2. Simulated Annealing

Simulated Annealing (SA) algorithm, also known as Monte Carlo annealing or probabilistic hill-climbing, is inspired by annealing in metallurgy [44]–[46]. Simulated annealing algorithm starts with a random solution and aims to improve it through several iterations. At each iteration, mutation process is applied to the current solution (S) to generate a new solution (S_{new}). After checking that the new solution is unique and never appeared before in the process, the algorithm determines to discard this new solution or keep it. If S_{new} has higher quality, it replaces the old solution, otherwise the new solution may be accepted with some probability. This probability is based on the system temperature (T) which decrease with each iteration. Higher system temperature means it is more probable to accept a poorer solution. The condition of accepting or denying the new solution are given in Eq. 4.

$$random[0,1) > \frac{1}{1+exp^{(\frac{q_{new}-q_{old}}{T})}} \quad \text{Eq. 4}$$

The algorithm selects a random value from a uniform distribution, if this value is greater than $\frac{1}{1+exp(\frac{q_{new}-q_{old}}{T})}$ the new solution with poorer quality is accepted. q_{old} is the quality of the current solution and q_{new} is the quality of the new solution. At the end of each iteration there is the cooling procedure, in which the system temperature is reduced. The amount of this reduction is defined by the cooling rate which is usually in the range of (0.9,1). The process continues until some stop criteria such as system temperature, or number of iterations are met. The initial system temperature and the cooling rate are determined through experiment and have different optimal values for different problems and workloads. There are different forms of using simulated annealing in the literature. Two studies [10], [47] have used SA in combination with GA and their results show improvements over both GA and SA. The pseudo code for the SA algorithm is shown below.

Algorithm 8: Pseudocode for SA

Generate a random initial solution (S_{old})population and calculate their fitness

While the stopping criteria not met

$S_{new} \leftarrow result\ of\ a\ succesful\ mutation$

If $q_{new} > q_{old}$

$S_{old} \leftarrow S_{new}$

Else if $random[0,1) > \frac{1}{1+exp(\frac{q_{new}-q_{old}}{T})}$

$S_{old} \leftarrow S_{new}$

Update system temperature

3. Tabu Search

Tabu Search algorithm looks the solution space both locally and globally to find a near optimal solution [10], [47]. It keeps track of visited areas in a "Tabu list", preventing the search procedure from exploring those parts again. The algorithm starts with an initial solution, which is sometimes generated randomly and sometimes it is totally or partially constructed from the result of a greedy algorithm. Wong et al. [48] used the result from HEFT as initial solution. There are two main steps in Tabu search; Short Hop, which searches the solution area locally, and Long-Hop, which explores globally and aims to prevent the algorithm from being trapped in local minimum. Short-Hops explore around the current solution by considering assigning every possible pair of processors to every possible pair of tasks and evaluating the quality of new solution. If the quality is higher than the current solution's quality, it is a successful hop and new solution replaces the current one [49], [50]. This process continues until there is no improvement, or the algorithm reaches the limit number of successful hops. When the short-hop ends, the final solution is added to the Tabu list, and the algorithm starts the long-hop process. To have a successful long-hop a random generated solution is needed, with at least half of tits assignments being different from all members of the Tabu list. Each successful long-hop leads to another short-hop. This process continues until the number of successful short-hops plus the number of successful long-hops reach the $limit_{hops}$. Like the GA and SA, some parameters like the length of the Tabu list, and $limit_{hops}$ are set through experiments and could vary from one problem to another. The pseudo code for Tabu Search algorithm is listed below.

Algorithm 9: Pseudocode for Tabu Search

Generate initial schedule

While the stopping criteria are not met

While $limit_{short\_hops} > 0$

Explore locally through short-hops

If S_{new} is better than the current one

Accept S_{new} and decrease the $limit_{short\_hops}$

Add the final result to the *Tabu_list*

Perform long-hop **Until** finding a solution which is different enough from all other results in the *Tabu_list*

4. A star

A* algorithm also explores the solution space to find the near optimal solution [10]. This algorithm uses a μ-ary tree where the root node is an empty solution and μ is the number of processors. As the tree grows, each node represents a partial mapping of the tasks. Each child node has one more task mapped compared to the parent node. The first level assigns the first task, the second level assigns the second task and so on. To keep the time complexity of the algorithm acceptable, the algorithm limits the maximum number of active nodes in the tree at any time. Whenever the number of active nodes is more than the limit, the algorithm uses pruning process to remove nodes with the worst fitness value. The fitness value for each node is the makespan of the partial mapping they present added to the maximum value of the execution time of the rest of the tasks, if all of them were assigned to the best processor of their choice.

Algorithm 10: Pseudocode for A*

While leaf nodes are not reached

Calculate the fitness of each node

Generate μ children for each node

If $active_{nodes} > limit_{active\_nodes}$

Remove the nodes with worse fitness

5. Memetic algorithm

Memes are the ideas or an element of a culture or system of behavior that can be passed by non-genetic means. R. Dawkins in "The Selfish Gene" introduced meme as a unit of imitation in cultural transmission, which is analogous to the gene in biology. Memetic algorithm [51] is very similar to GA and based on the definition of metaheuristics, i.e. finding better solution by modifying and improving local search in heuristics, could be classified as a metaheuristic. Essentially, MA works with an initial population in which the local search is performed by every chromosome. Solution to the problem (chromosomes) are presented by memes based coded strings, where the encoding scheme greatly affects the performance of the algorithm. The initial population is generated randomly. The best chromosomes are selected for the evolutionary process (the crossover and mutation). Then local search on the generated offspring is executed, which can be performed using Tabu search, SA, or hill-climbing algorithm. MA results show improvements over GA. The pseudocode for the MA is shown below.

Algorithm 11: Pseudocode for MA

Generate the initial population

While the stopping criteria are not met

Do local search on each member of the population

Select parents, do cross over, do mutation

If the new offspring are better than other member of the population

Add the offspring to the population and remove the less fit ones

6. Ant Colony

Ant Colony Optimization (ACO) algorithm was introduced by Dorigo et al. [52] based on the behavior of a group of ants. Any kinds of blind insects use a substance called pheromone, which helps others to choose the best way. The path with higher pheromone means it was selected by majority of the group therefore it is a good direction. In ACO algorithms artificial ants act as agents and search the solution space while recording the chosen path in a pheromone table. Initially all cells of the pheromone table are set to 1, then in each iteration each ant selects a task to map to its best machine. At the end of each iteration, the fitness of the solution and the performance of each ant is evaluated, and the pheromone table is updated [11]. Different versions of ACO use different strategy for selecting next task for each ant, fitness function, and implementation of the pheromone table. The pseudocode for ACO algorithm is listed below.

Algorithm 12: Pseudocode for ACO

Initialize pheromone table

While the stopping criteria are not met

For each ant

While there are unmapped tasks

Select a task and map it to its best computing node

Compute fitness of the ants

Update pheromone table

ACO belongs to the class of swarm optimization algorithms, which is inspired from the collective intelligence of animals. Various optimization and search problems have been solved using Swarm Intelligence (SI) techniques. The artificial bee colony (ABC), particle swarm optimization (PSO), cat swarm optimization (CSO) and bat algorithm (BA) are some other members of this class [52].

OTHER CLASSIFICATION METHODOLOGIES

Scheduling algorithms can be classified in different categories based on specific properties or parameters. Some of these categories are listed below.

- Type of task to schedule, which can be categorized as dependent or independent tasks. Independent tasks are referred to as a bag of tasks (there is no order on the execution of them). On the other hand, dependent tasks need to follow a specific order in the execution time. They are usually represented as a Directed Acyclic Graph.
- Type of mapping, which can be static or dynamic.
- Data type, which can be deterministic or stochastic. Scheduling algorithms need

some data about the execution time of the tasks, or the communication cost between two computing units. These data are usually provided through some previous experiments. This estimated information is not very accurate, and some algorithms are using a range of values to produce a more resilient and robust scheduling. Dealing with deterministic data reduces the complexity of the algorithm, while results from using stochastic data are more realistic.

- Target environment, which can consist of homogeneous or heterogeneous computing units. A homogeneous system is an environment in which all the computation units have the same characteristics. On the other hand, machines in a heterogeneous environment have different computational strength.
- Consistency of the computing units, which can be consistent or inconsistent. In a consistent system if machine A is faster than machine B for one task, then machine A is faster than machine B for all other tasks.
- Optimization goal, which can be categorized to goal oriented or constrain oriented. Scheduling algorithm always aim to optimize some cost functions. The cost function could be comprised of performance, or as referred to in literature the makespan, power consumption, robustness, algorithm speed, resiliency, etc. Some scheduling problems must reach a solution while meeting a specific deadline or specific energy budget. In general, they need to maintain some level of quality of service, thus these group of scheduling algorithms are called constrained ones.

CHAPTER 3

In this chapter, two new scheduling algorithms are introduced; the Parallel Heterogeneous Earliest Finish Time (PHEFT) and the Optimize All Paths (OAP), which are designed for a heterogeneous environment and a set of dependent tasks. Two algorithms are explained through some examples, and then compared with some of the heuristics listed in the Chapter 2. For this comparison, a set of DAGs generated with a Random DAG Generator are used. Details about the Random DAG Generator, which can be used cover a wide range of applications and environment characteristics, are provided in Chapter 4. The PHEFT algorithm shows substantial improvement over other greedy heuristics in a consistent computing system, and the same performance in the case of inconsistent environment. Due to OAP algorithm's higher time complexity, another set of random DAGs with a fewer number of nodes is used for the comparison purpose. OAP algorithm's results do not show any improvement. In fact, it shows same performance as some very simple and fast greedy heuristics.

PHEFT ALGORITHM

The Heterogeneous Earliest Finish Time (HEFT) algorithm was discussed in the previous the previous chapter. Here in this section, a new algorithm is introduced, Parallel Heterogeneous Earliest Finish Time (PHEFT). PHEFT is very similar to HEFT. Like HEFT, this new algorithm consists of two phases. In the first phase, priorities are assigned to the tasks. Then in the second phase, the algorithm schedules tasks in order of their priority.

The same upward ranking algorithm to find the priorities is employed here. The rank for each task is the average execution cost of the task over all available machines plus the maximum average communication cost between the task and all its children, plus the rank of the children. The formula for upward rank [HEFT] is given by Eq. 5.

$$rank_u(i) = \overline{w_i} + \max_{j \in \text{succ(i)}} (\overline{c_{i,j}} + rank_u(j)) \quad \text{Eq. 5}$$

Upward rank for the end task, which has no children, is its average execution cost on all available machines, given by Eq. 6. Thus, one can start from the end tasks and move upwards to determine the rank of every task in the graph.

$$rank_u(n_{exit}) = \overline{w_{exit}} \quad \text{Eq. 6}$$

In PHEFT, this dissertation work explored a number of different ways to rank the tasks in a graph, however none of them produced better result than the original upward ranking. For example, using the rank and communication cost of all the children of a task compared to just using the maximum value was attempted. This given by Eq. 7.

$$rank_u(i) = \overline{w_i} + \sum(\overline{c_{i,j}} + rank_u(j)) \quad \text{Eq. 7}$$

The reasoning behind it was that since a task with more children would have a greater effect on the final execution time, it should have higher priority. This did not turn out as expected, because only one of the children (the one with the maximum communication cost plus rank) would be on the critical path of the graph. Therefore, it was decided to use the original upward rank algorithm. Moreover, not only the order of the tasks in the ranking was used, but also their actual ranking values. How similar or different the priorities of the tasks are, was considered. If given two tasks with relatively close numbers for their priority, then sharing the best available machine between these two tasks is preferred. However, if their ranks differ too much, even if sharing resources resulted in faster execution of the two tasks combined, it would slow down the higher priority task. This will result in overall slower execution.

After finding the priorities, the second phase is to assign a machine to each task. This is done using a list of available resources, and a list of tasks ready to be executed. If there is only one task in the ready list, that task is taken and assigned to the best machine available and the ready list is updated with the successors of that task. If there is more than one task in the ready list, then the two tasks with the highest priorities are taken. If the priority level of these two tasks is relatively close, sharing resources between these two tasks is given a consideration. For this reason, first all priority levels, found in the first step, are divided by the highest level (the root). Thus, all priority levels would be some values between 0 and 1. Two tasks are considered to have a close priority level, if the difference between their priority levels is less than 0.2. If the difference between priority levels of two tasks is greater than this, then resources between them are not shared. Different values in the range of 0.09

to 0.5 for this threshold were tested in the experiment, and it was found that this threshold worked better for most of the DAGs.

Before sharing resources, it is confirmed that the two tasks are competing for the same resource at the same time. If the best option for these two tasks are two different machines, or they need to use the same machine, but in two different period with no time overlap, then the first task can be assigned, while decision about the second one can be done in the next round of scheduling. However, if both tasks require the same machine, during the same period, the best of the two options listed below is chosen.

1. Run T1 and then run T2
2. Run T1 and T2 in parallel

Where T1 is the fastest finish time for the first task and T2 is the fastest finish time for the second task. The first option is same as the one used by the HEFT algorithm. First, the algorithm schedules the task with higher priority, updates the available resources and then schedules the second task. The second option finds the finish time, if half of the cores on the fastest machine were assigned to the first task and the other half were assigned to the second task. One of these two options is selected, based on which gives the best finish time. Subsequently tasks are assigned to the machines, and the ready list and available resources are updated. This process is repeated until there are no tasks on the ready list. The pseudo code for the PHEFT algorithm is given below.

Algorithm 13: Pseudocode for PHEFT Algorithm

- Task prioritizing phase
    - Set the computation cost of tasks and communication cost of edges with mean values.
    - Compute $rank_u$ for all tasks by traversing graph upwards, starting from the exit task.

$$rank_u(i) = \overline{w_i} + \max_{j \in \text{succ(i)}} (\overline{c_{i,j}} + rank_u(j))$$

$$rank_u(n_{exit}) = \overline{w_{exit}}$$

    - Sort the tasks in a scheduling list by non-increasing order of $rank_u$ values.
- Processor selection phase

while there are unscheduled tasks in the list **do**

- Select the first two tasks in the list i, j and find the best machine for them
- If the best machine for these two tasks is different, assign the first task to its best machine
- Else if both tasks require same machine (k) at the same period find these 2 values:
    - $EFT(i,k) + EFT(j,k)$
    - $EFT(i, k(half\ core)) + EFT(j, k(half\ core))$
- If the first option is faster, assign task i to machine k, otherwise assign both tasks i, j to machine k (parallel assignment)

End while

As an illustration, Figure 3.2 and Figure 3.3 present the schedules obtained by PHEFT and HEFT algorithms, respectively, for the sample DAG in Figure 3.1. The computation cost is presented in Table 3.1, while Table 3.2 gives the upward rank values for the given graph. The scheduling order of the tasks based on their upward rank is $\{n_1,\ n_3,\ n_2,\ n_5,\ n_4,\ n_8,\ n_6,\ n_7,\ n_{10},\ n_9\}$.

Figure 3.1: Sample DAG

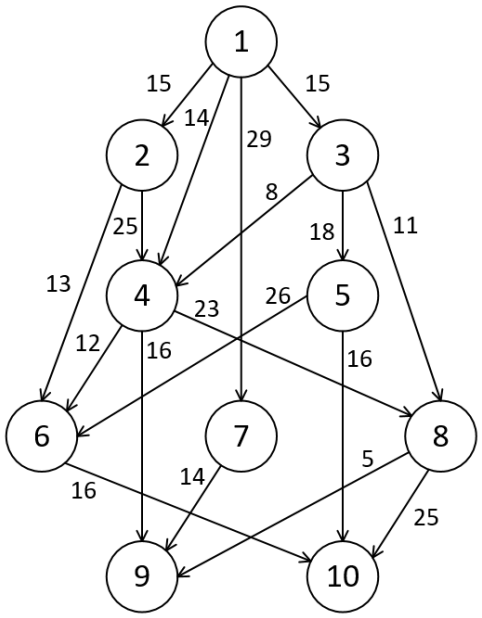


Table 3.1: Computation Cost

	P_1	P_2	P_3
1	24	10	16
2	33	32	22
3	15	26	17
4	19	16	9
5	14	33	7
6	36	40	35
7	11	20	32
8	29	17	39
9	25	27	15
10	34	12	35

Table 3.2: Upward Rank

n_i	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
rank (n_i)	193	160	161.33	118	124	80	57.33	80.33	22.33	27

The schedule length for PHEFT is 131, which is shorter than the schedule length of the HEFT algorithm, which is 151. The only difference is that in PHEFT the algorithm selects to run tasks $n_5,\ n_4$ in parallel. This decision changed the possible options for tasks $n_8,\ n_6$, which results in a faster scheduling for the task graph

Figure 3.2: Time Diagram for PHEFT

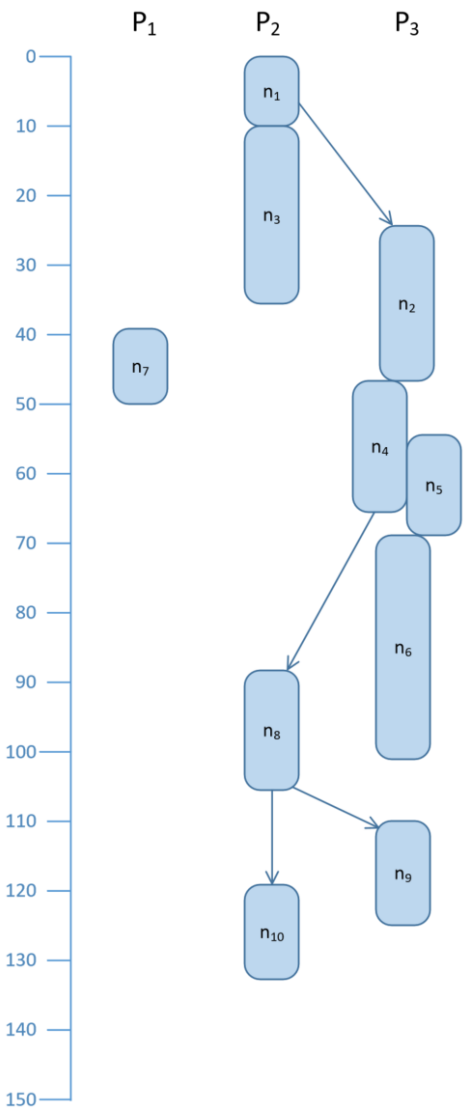
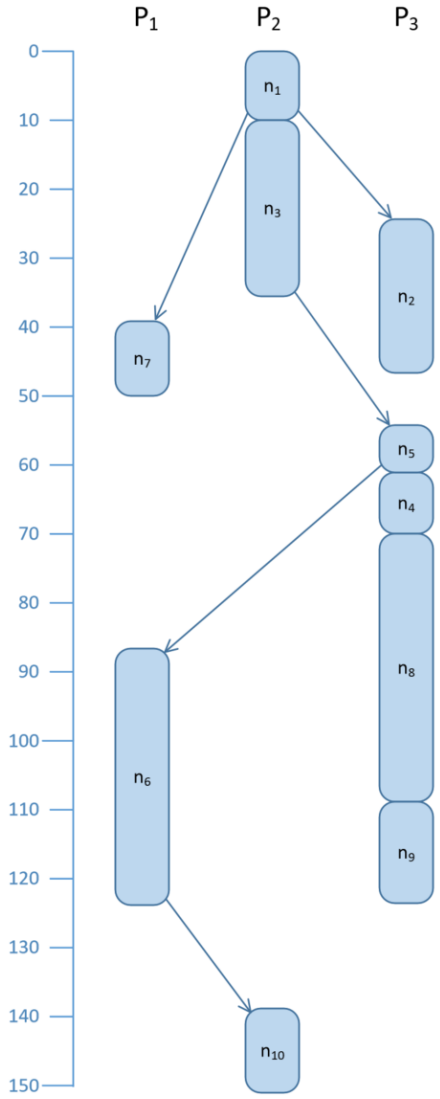


Figure 3.3: Time Diagram for HEFT



The following assumptions for PHEFT algorithm have been employed by this work:

- Machines have separate parts to deal with communication. Thus, computation and communication can take place at the same time.
- Machines can divide their resources and manage this division.

The execution time of the algorithm is an important overhead, which needs to be as low as possible. To keep this algorithm relatively fast some key parameters need to be determined. One of these is the number of tasks considered for parallel assignment. As mentioned earlier, at any point only the first two tasks in the ready list are compared. One can perhaps argue that the number could be three or any arbitrary number. One reason to limit the selection to only two tasks, is it to maintain a relatively lower algorithm complexity. For example, in case of two tasks there would be two options, but with three tasks there would be five options, which are listed below.

1. Run T1, then T2, and then T3
2. Run T1, then run T2 and T3 in parallel
3. Run T1 and T2 in parallel, then run T3
4. Run T1, and T3 in parallel, then run T2
5. Run T1, T2, and T3 in parallel

Increasing the number of tasks considered for parallel assignment increases the complexity of the algorithm. Therefore, only two consecutive tasks in the ready list are considered for the parallel assignment.

The next decision deals with dividing resources between two tasks, when it is necessary to run them in parallel. Here only consideration is given to sharing the resources equally between the two tasks. For example, if a machine has eight cores, then four cores are assigned to task1 and four cores to task2. Other option is to consider their execution cost and share resources between them based on their computational needs. For example, if the execution time of task1 is half the execution time of task2 then assigning 1/3 of cores to task1

and 2/3 to task2 would be appropriate. This also adds to the complexity of the algorithm, therefore only the simple version is used here, which dictates dividing the resources equally between the two tasks.

Time Complexity of PHEFT Algorithm

The PHEFT algorithm has two phases, task prioritizing which assigns a rank to each task, and processor selection, which selects the best available resource for the task. For the first phase the average computation time and communication time are obtained for each task and then sorted in descending order of their rank. This process takes $O(v \log v)$, where v is the number of tasks. The second phase consists of a search for an idle time slot on a processor, with start time earlier than or equal to the time the task is ready to run (all the predecessors finish their execution and sent their data to the current task), while its length is greater or equal to the execution time of the task on the selected processor. For each processor two options are obtained, one is the time slot with the full number of available cores and the other with half of that number. This second option is used for the parallel assignment. Then the algorithm selects the time slot that results in the best finish time for the two consecutive tasks in the ready list. This process is $O(e \times q)$, where e is the number of edges and q is the number of processors. For a dense graph which is closer to a complete graph with $e = v^2$ the time complexity will be $O(v^2 \times q)$. Hence the complexity of PHEFT is $O(v^2 \times q)$.

OAP ALGORITHM

The Optimize All Paths (OAP) algorithm is another scheduling algorithm, which is proposed by this dissertation work, for scheduling DAGs in a heterogeneous environment. The key concept of this algorithm is scheduling a task at each step, which in turn results in making the longest path faster. With other scheduling algorithms the critical path can often change during the scheduling process. Assigning the fastest resources to the critical path leads to assigning other tasks to slower machines and possibly adding to the communication cost of those paths, which may result in generating another critical path. In this OAP algorithm all the paths at every step are examined and an attempt is made to reduce the cost of the longest path at each step.

The first step consists of finding all paths from all start tasks (tasks with no predecessor) to all end tasks. This step is the most time-consuming part. The complexity of this part is $O(2^v)$ where v is the number of nodes. Thus, it is evident that this algorithm can only be considered for small DAGs with small number of nodes.

The second step consists of calculating the critical value for each path and finding the path with the highest critical value. The critical value of each path is the sum of average execution cost of tasks in the path and communication cost between them, as given by Eq. 8.

$$Critical\_Value_p = \sum_{for\ i\ in\ p} \overline{w_i} + \sum_{\substack{for\ j\ following\ i\ in\ p}} c_{i,\,j} \qquad \text{Eq. 8}$$

The algorithm then selects the path with the highest critical value. If the first unscheduled task in this path is in the ready list, it schedules it to its best option. If not, it finds the next path with the highest critical value and tries to schedule a task on that path.

The critical value of paths is then updated, and this process repeats until all tasks are scheduled. The pseudo code for the OAP algorithm is given below.

Algorithm 14: Pseudocode for OAP Algorithm

- Path prioritizing phase
    1. Set the computation costs of tasks and communication costs of edges with mean values.
    2. Find all paths from all start nodes to all end nodes.
    3. Compute $Critical\_Value$ for all paths.

$$Critical\_Value_p = \sum_{for\ i\ in\ p} \overline{w_i} + \sum_{\substack{for\ j\ following \\ i\ in\ p}} c_{i,\ j}$$

- Path selection phase

while there are unscheduled tasks **do**

while a task to assign has not been found **do**

- Select the next path with the highest $Critical\_Value$
- Find the first unscheduled task in the path which is ready to run

End while

1. Assign selected task to its best option
2. Update all paths with the selected task

End while

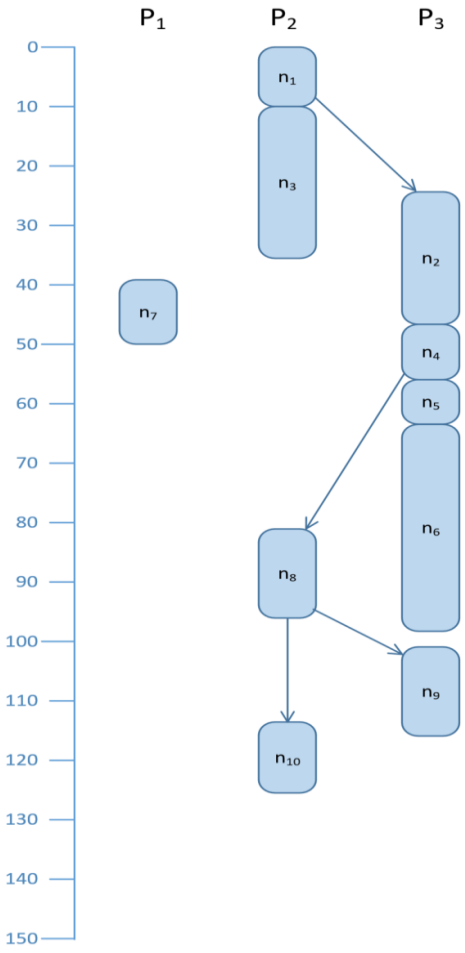
Table 3.3: Updated Weights of Paths with Each Iteration

	Paths	*Step 1*	*Step 2*	*Step 3*	*Step 4*	*Step 5*	*Step 6*	*Step 7*	*Step 8*	*Step 9*	*Step 10*
P_1	n_1, n_2, n_4, n_8, n_9	167.0	160.3	160.3	153.3	134.7	134.7	134.7	123.3	**123.3**	**123.3**
P_2	n_1, n_2, n_4, n_9	126.7	120.0	120.0	113.0	94.3	94.3	94.3	94.3	94.3	94.3
P_3	n_1, n_3, n_4, n_8, n_9	152.3	145.7	137.3	137.3	134.7	134.7	134.7	123.3	**123.3**	123.3
P_4	n_1, n_3, n_4, n_9	112.0	105.3	97.0	97.0	94.3	94.3	94.3	94.3	94.3	94.3
P_5	n_1, n_3, n_8, n_9	117.7	111.0	102.7	120.7	102.7	102.7	102.7	123.3	**123.3**	123.3
P_6	n_1, n_4, n_8, n_9	124.0	117.3	117.3	117.3	134.7	134.7	134.7	123.3	**123.3**	123.3
P_7	n_1, n_4, n_9	83.7	77.0	77.0	77.0	94.3	94.3	94.3	94.3	94.3	94.3
P_8	n_1, n_7, n_9	103.0	96.3	96.3	96.3	96.3	96.3	96.3	96.3	**96.3**	86.3
P_9	n_1, n_2, n_4, n_6, n_{10}	180.0	173.7	173.7	166.7	148.0	148.0	141.0	141.0		
P_{10}	n_1, n_2, n_4, n_8, n_{10}	191.7	185.0	**185.0**	**178.0**	159.3	159.3	**159.3**	**148.0**		
P_{11}	n_1, n_2, n_6, n_{10}	165.7	159.0	159.0	152.0	152.0	152.0	141.0	141.0		
P_{12}	n_1, n_3, n_4, n_6, n_{10}	165.7	159.0	150.7	150.7	148.0	148.0	141.0	141.0		
P_{13}	n_1, n_3, n_4, n_8, n_{10}	177.0	170.3	162.0	162.0	159.3	159.3	159.3	148.0		
P_{14}	n_1, n_3, n_5, n_6, n_{10}	**193.0**	**186.3**	178.0	178.0	**178.0**	**169.0**	141.0	141.0		
P_{15}	n_1, n_3, n_5, n_{10}	139.0	132.3	124.0	124.0	124.0	115.0	115.0	115.0		

P16	n_1, n_3, n_8, n_{10}	142.3	135.7	127.3	127.3	127.3	127.3	127.3	148.0
P17	n_1, n_4, n_6, n_{10}	137.3	130.7	130.7	130.7	148.0	148.0	141.0	141.0
P18	n_1, n_4, n_8, n_{10}	148.7	142.0	142.0	142.0	159.3	159.3	1a.3	148.0

To demonstrate an application of the OAP algorithm in action, the same task graph from Figure 3.1 is employed with OAP. Table 3.3 shows all paths from n_1(start task) to n_9 and n_{10} (end tasks), and how their critical value changes in each step of the algorithm. The time diagram chart in Figure 3.4 shows the final execution time with OAP is 126, which is better than both PHEFT and HEFT, whose execution times are 131 and 151, respectively.

Figure 3.4: Time Diagram for OAP



Time Complexity of OAP Algorithm

The time complexity of the first part of the algorithm is $O(2^v)$, where v is the number of tasks. To find all paths from the source task to all end tasks, each task has two options either be in the path or not. So, the number of all paths are in the order of two to the power of number of tasks. The second part of the algorithm has a time complexity of $O(n \log n \times q)$, where n is the number of paths found in the previous step, and q is the number of the processors. Thus, the overall complexity of this algorithm is $O(2^v)$.

EXPERIMENTAL RRESULTS AND DISCUSSION

To evaluate the performance of PHEFT and OAP, they are compared to some of the algorithms published by others. The workload used to test all these algorithms is based on some randomly generated DAGs. The performance of PHEFT and OAP algorithms introduced by this dissertation work are compared with eight others, which were discussed in detail in Chapter 2, hence a very short description for each of them follows.

The Heterogeneous Earliest Finish Time (HEFT) algorithm [18] determines an order for the tasks in the DAG using the upward rank algorithm. Then takes tasks from the list in order of their ranks and assigns them to a processor resulting in minimum execution time.

The Critical Path on a Processor (CPOP) algorithm [18], uses both upward and downward rank to prioritize tasks. Then the algorithm selects a processor for the critical path, which gives the minimum execution time among all processors. Taking the highest priority task at each step, the algorithm assigns tasks on the critical path to the critical processor and other tasks to their best available processor.

The Contention Resolution (CR) algorithm [11], determines the best assignment for each unscheduled task at each iteration. Those tasks with no rival for their best option get assigned, then among remaining tasks, which compete for the same resource, the one with the largest difference between the two smallest execution times gets scheduled.

The Levelized Min Time (LMT) algorithm [19], first groups those tasks that can be executed in parallel at the same level, and starting from the lowest level it assigns the task with the highest computation time to the best available processor for that task.

The Dynamic Level Scheduling (DLS) algorithm [20], select the available task and processor pair, at each level, which maximize the dynamic level value. The dynamic level value for task i on processor p is the $rank_u(i)$ minus the earliest start time of the task i on processor p, $EST(i, p)$.

The two-phase heuristics, Min-Min and Max-Min algorithms [15], find the best processor for each ready task, which is the one resulting in the minimum execution time, at each step. Then, in the second phase, the Min-Min algorithm selects the pair with the shortest execution time, while the Max-Min algorithm selects the pair with the maximum execution time. This process repeats until there is no task left to schedule.

The Opportunistic Load Balancing (OLB) algorithm [10] assigns each task in arbitrary order, to the first available processor regardless of the task's execution time.

Comparison Metrics

This works adopts the performance metrics presented by Topcuoglu et al. [18], to evaluate the new proposed task scheduling algorithms against the existing algorithms. The paragraphs to follow describe these performance metrics in detail.

The most important factor, which defines the performance of a scheduling algorithm, is its length, which is called **makespan**. The makespan of different scheduling algorithms for a DAG can be compared, however, when dealing with several random DAGs with different properties, it is necessary to employ means of normalizing the results for comparison purpose. The **schedule length ratio (SLR)** of an algorithm on a task graph is defined by dividing the makespan by the sum of the execution cost of the tasks on the critical path, if they all used the fastest machine to run, it is given by Eq. 9.

$$SLR = \frac{makespan}{\sum_{n_i \in CP} min_{p_j \in Q}\{w_{i,j}\}} \quad \text{Eq. 9}$$

The denominator in Eq. 9 consists of finding the critical path cost, if all tasks on that path were executed on the fastest processor. Which could result in different processors for different tasks, if the processors are inconsistent. The communication is not considered here. The minimum value for the critical path is always less than the makespan; therefore, SLR value is always greater than 1. The scheduling algorithm with smaller SLR value is more favorable in terms of performance, compared to one with higher SLR value.

Speedup, also referred to in the literature as parallelism factor of a DAG, is another metric used in this work to compare the scheduling algorithms' performance. It is defined as the ratio of the parallel execution time to the sequential execution time. The parallel execution time is the makespan obtained from the scheduling algorithm. The sequential execution time is the sum of the execution time of all tasks in the graph, if they were to all run on one processor. The processor that gives the minimum final execution time is selected. Speedup depends on the number of processors provided. Increasing the number of processors decreases the makespan, which in turn increases the speedup. Hence, it is

important to consider the number of processors and use the ratio of speedup to the number of processors as an evaluation metric. The formula for the speedup is given in Eq. 10.

$$speedup = \frac{min_{p_j \in Q}\{\sum_{i \in V} w_{i,j}\}}{makespan} \quad \text{Eq. 10}$$

It is possible that, at times, an algorithm can perform better, worse, or equal compared to other algorithm, depending on the DAG parameters. Since this work reports an average value over several results for each DAG it is important to also consider the number of times an algorithm produces better, worse, or equal results to another algorithm.

Another important metric to consider is the **cost of the algorithm**, i.e. how long it takes for the algorithm to produce the desired result. If an algorithm generates near optimal scheduling but take considerably high amount of time to produce the solution, it is not practical to use that algorithm.

Task Graphs to Evaluate Scheduling Algorithms

There are accounts [53] demonstrating scheduling algorithms performance varies based on the random DAG generator algorithm used to generate sample random DAGs for test purpose. The work presented in this dissertation uses two sets of random DAGs. The first group are generated with a random DAG generator, implemented by the author. The second group are generated from another random DAG generator adopted from the work of Topcuoglu et al. [18]. Since, one of the proposed algorithms in this dissertation was inspired by the HEFT algorithm, it is appropriate to employ the random DAG generator used by the authors of the HEFT algorithm, for a fair comparison.

Random DAG Generator Implementation

The algorithm developed by this work, generates random DAGs with various characteristics that are defined by the following input parameters.

- Number of tasks, (v).
- Number of levels, (l).
- Standard deviation of the number of tasks in each level, ($level_{std}$). Each level has at least one task. The remaining tasks ($v - l$) are assigned to different levels using a normal distribution with average value of $(v - l)/(l - 1)$ and standard deviation of $level_{std}$.
- Connection probability (Con_prob), defines the number of connections in the DAG. A DAG with v nodes can have $[(v - 1), (\frac{v(v-1)}{2})]$ edges. Higher connection probability means higher number of edges (Connection probability of 1 means a complete DAG). A task in a level should be connected to at least one task from the previous level. Starting with the minimum number of $v - 1$ edges, the connection ratio would be $(v - 1)/(\frac{v(v-1)}{2}) = \frac{2}{v}$. The algorithm keeps adding edges to the DAG until the connection ratio is close to the connection probability.
- Jump level (jl), indicates the maximum number of levels an edge can jump. The upper bound for the jump level is the level number. A task in level a can have a connection to any tasks in any level from $a + 1$ to l (the last level). The destination level for an edge in the DAG is a random number between $current_{level} + 1$ and the $\min(l, current_{level} + jump\_level)$.
- Average communication cost ($comm\_cost\_mu$), is the average weight of the

connections (edges) in the DAG.

- Standard deviation of the communication cost ($comm\_cost\_std$), defines how different the weights of the connections in the DAG are. Weights of the connections are obtained from a normal distribution with mean value of mm_cost_mu, and $comm\_cost\_std$, as the standard deviation.
- Computation to communication ratio (CCR), shows the relationship between computation cost and communication cost. Using CCR and $comm\_cost\_mu$, the algorithm finds the average value for the computation cost. Then, the computation cost for each task on each machine is obtained from a normal distribution with mean value of $comp\_cost\_mu$, and standard deviation of $comm\_cost\_std$ (same as standard deviation for communication cost). Computation cost of the processors in a consistent system follows the rule of consistency (if a processor is the fastest processor for one task, it is the fastest processor for all other tasks).
- The processor distribution and number of cores ($machine\_core\_dis$), is list of processors with the number of cores for each one. An algorithm shows different behavior on different system. Therefore, to test the algorithms in different scenarios, the algorithm takes the system distribution as an input. Computation cost values and processor distribution data are used to make the execution time matrix.
- Task heterogeneity degree, which shows variation on computational needs of the tasks in a DAG.
- Machine heterogeneity, which shows the variation of computational power of the machines in the system.

In each experiment the values of these input parameters are assigned from the sets below. For different number of nodes, different sets for levels are considered. For example, a DAG with 20 nodes cannot have 30 levels. The $comm\_cost\_mu$ is set to 20, because previous experiments have shown the important factors are the standard deviation of the communication cost, and the computation to communication cost ratio.

- $v = \{20,\ 40,\ 60,\ 80, 100\}$
- $l = \{20: [5,10,15],\ 40: [10, 20, 30],\ 60: [15, 30, 45],\ 80: [15, 40, 65],$
  $100: [20, 50, 80]\}$
- $level_{std} = \{0, 3, 5\}$
- $Con\_prob = \{0.5,\ 0.8,\ 0.9\}$
- $jl = \{5, 10, 20\}$
- $comm\_cost\_mu = 20$
- $comm\_cost\_std = \{0, 4, 8\}$
- $CCR = \{0.5, 1, 2, 5\}$
- $machine\_core\_dis = \{3: [1, 2, 4], 4: [1, 2, 4, 8], 5: [1, 2, 4, 8, 16], 6: [1, 2, 4, 8, 16, 32]\}$
- $Task\_heterogeneity = 0.1\ for\ low\ heterogeneity,$
  $0.6\ \ for\ high\ heterogeneity$
- $Machine\_heterogeneity = 0.1\ for\ low\ heterogeneity,$
  $0.6\ \ for\ high\ heterogeneity$

To add task and machine heterogeneity to the test, the CVB method (described in detail in Chapter 4) is used with gamma distribution. Four different systems are considered: 1) Low task Low machine heterogeneity (LL), 2) Low task High machine heterogeneity (LH),

3) High task Low machine heterogeneity (HL), 4) High task High machine heterogeneity (HH).

All the generated ETC (estimated time to compute) matrices are inconsistent and to obtain the consistent ETC, they are sorted once along rows and then along the columns. The combination of these parameters results in 155,520 different DAGs. Since 15 random DAGs are generated for each DAG type, there are 2,332,800 DAGs available for the performance comparison tests. This large set of DAGs with diverse characteristics prevents biasing toward any specific type of DAG.

Results Using Consistent DAGs

The test platform used in this work generates the random DAG and the ETC matrix, then schedules it using the nine different heuristics listed in Chapter 2, and finally reports the average results. Figures 3.5 to 3.8 represent the results of mean makespan versus number of nodes, for a consistent system on LL, LH, HL, and HH ETC matrices, respectively.

PHEFT outperforms other algorithms with all four heterogeneity degree matrices. Increasing the number of nodes increase the improvement of PHEFT over other algorithms. For LL and HL all other algorithms except PHEFT shows almost same performance. For LH and HH, all algorithms except the two with lower performance (OLB and DLS) show improvements over the same algorithms compared to LL and HL ETCs. Thus, except for OLB and DLS, all other algorithms show better performance when the machine heterogeneity is high.

Figure 3.5: Makespan vs Nodes for LL

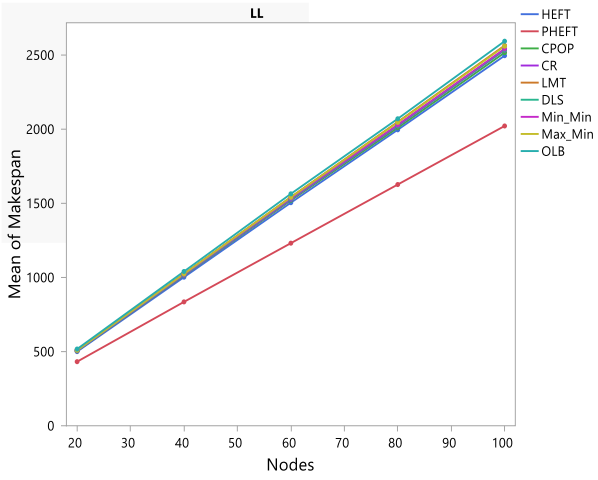


Figure 3.6: Makespan vs Nodes for LH

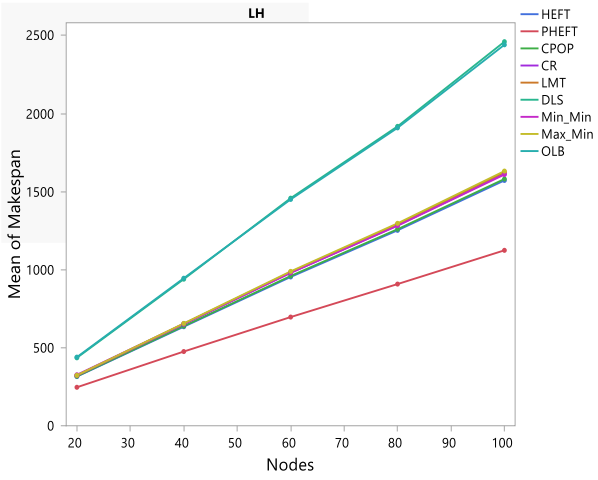


Figure 3.7: Makespan vs Nodes for HL

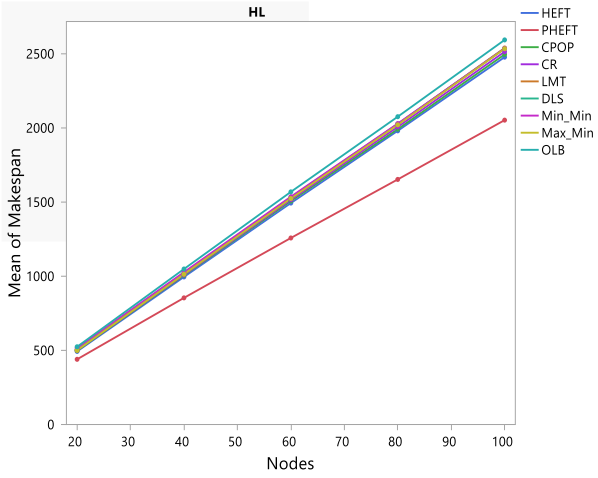
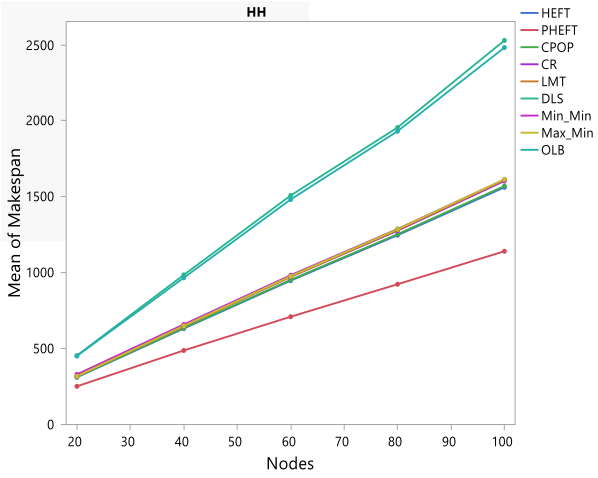


Figure 3.8: Makespan vs Nodes for HH



Figures 3.9 to 3.12 compare the mean SLR for four classes of LL, LH, HL, and HH ETC matrices. SLR is defined by the makespan divided by the minimum critical path cost. The normalization takes away the effect of the size of the DAG to the great extent. An algorithm with lower SLR values is considered to have higher performance. In all four categories PHEFT exhibits better performance compared to the other algorithms. With increasing

number of nodes, the difference between PHEFT and the other algorithms increases. Higher machine heterogeneity results in worse SLR, likewise, higher task heterogeneity also has the same effect. In case of high machine heterogeneity, the SLR values of the algorithms are divided in three distinct groups, the same was observed for the makespan.

Figure 3.9: SLR vs Nodes for LL

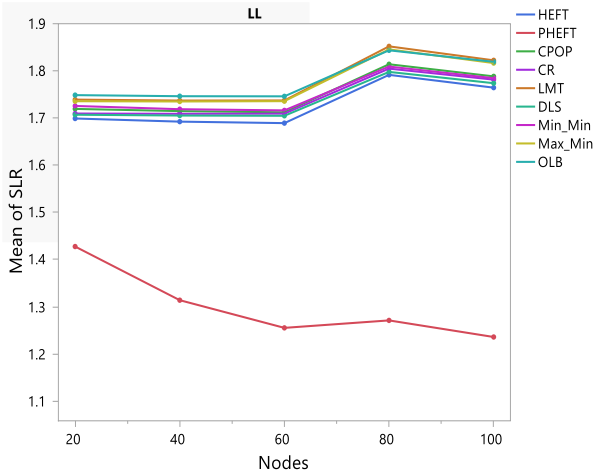


Figure 3.10: SLR vs Nodes for LH

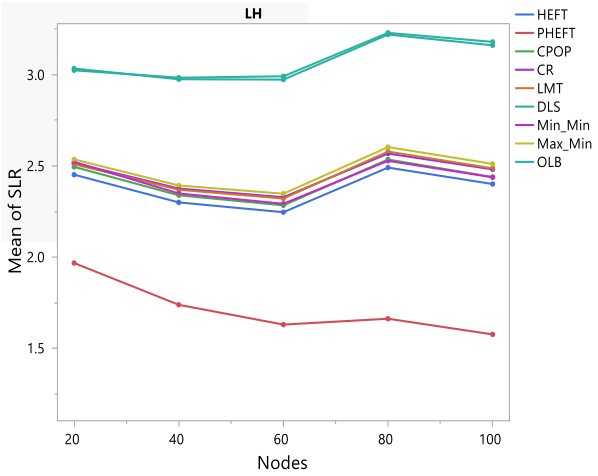


Figure 3.11: SLR vs Nodes for HL

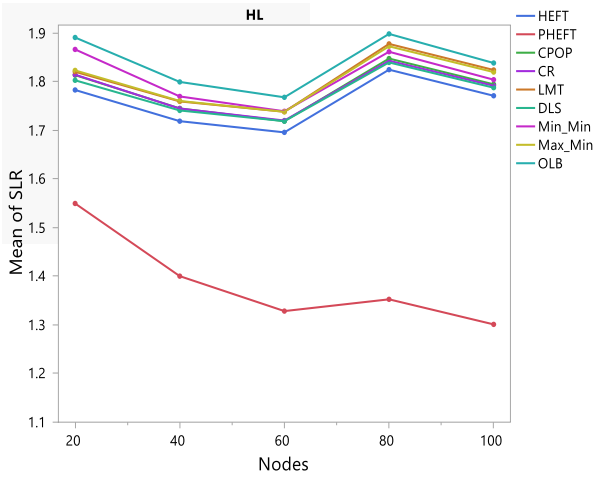
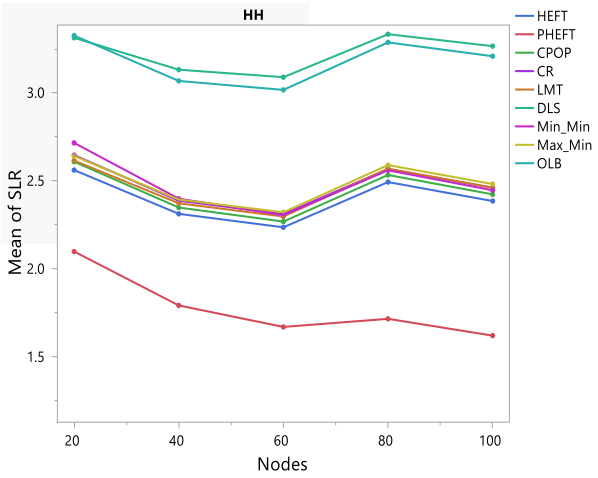


Figure 3.12: SLR vs Nodes for HH



Another metric used to compare the performance of algorithms is the speedup, the results of which are shown in Figures 3.13 to 3.16. Speedup or parallelism shows how much

improvement an algorithm provides over the sequential execution. PHEFT shows higher speedup than all other algorithms. Furthermore, with increasing number of nodes the speedup for PHEFT increases.

Figure 3.13: Speedup vs Nodes for LL

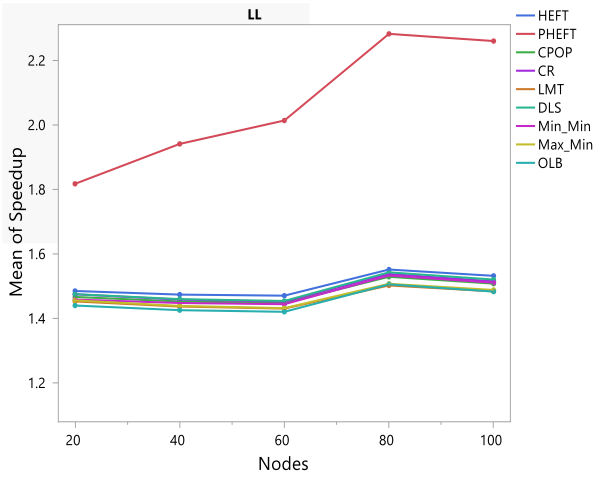


Figure 3.14: Speedup vs Nodes for LH

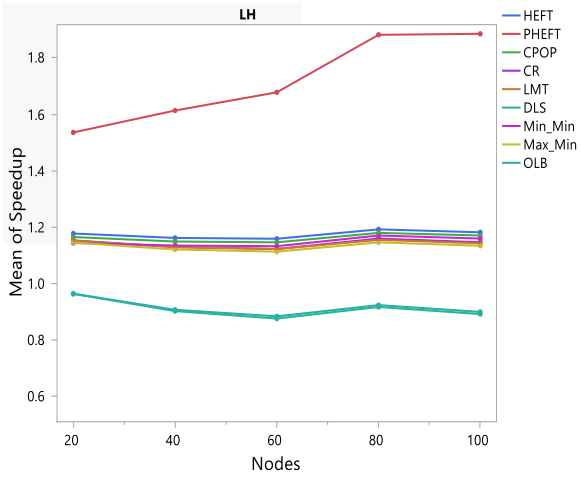


Figure 3.15: Speedup vs Nodes for HL

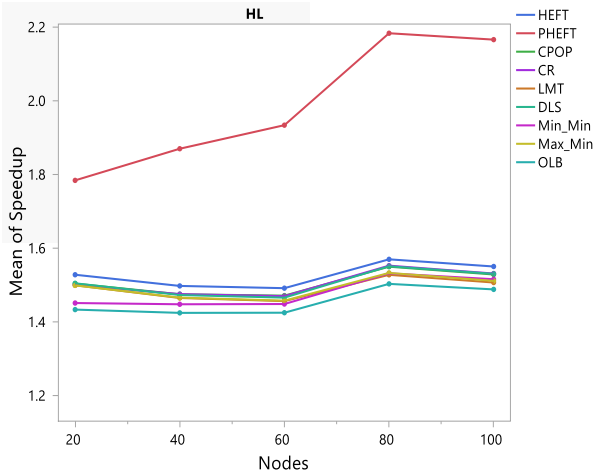
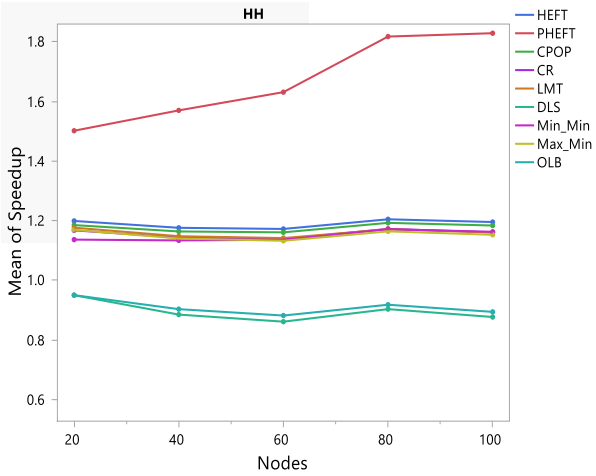


Figure 3.16: Speedup vs Nodes for HH



Results Using Inconsistent DAGs

Figures 3.17 to 3.28 show the results for an inconsistent system. PHEFT performance is same as HEFT and close to the other algorithms. These results show that the consistent system makes better use of parallelism provided by PHEFT algorithm, while the inconsistent does not.

Figure 3.17: Makespan vs Nodes for LL

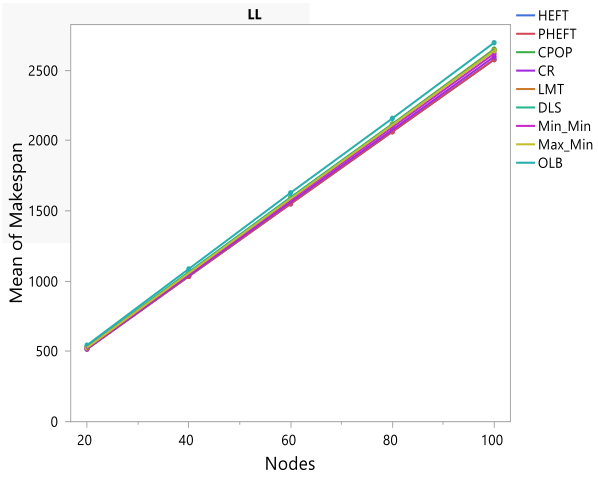


Figure 3.18: Makespan vs Nodes for LH

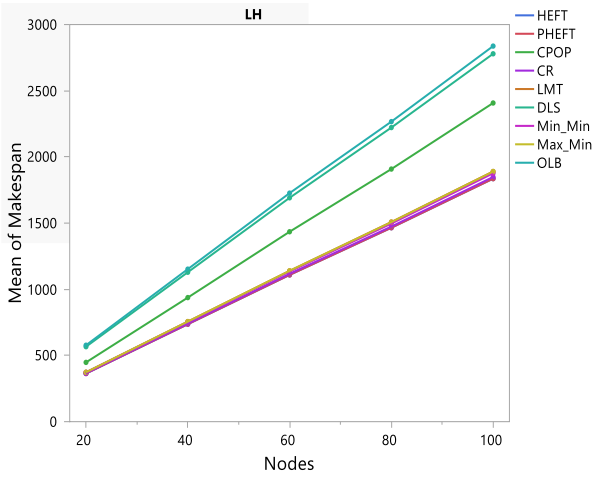


Figure 3.19: Makespan vs Nodes for HL

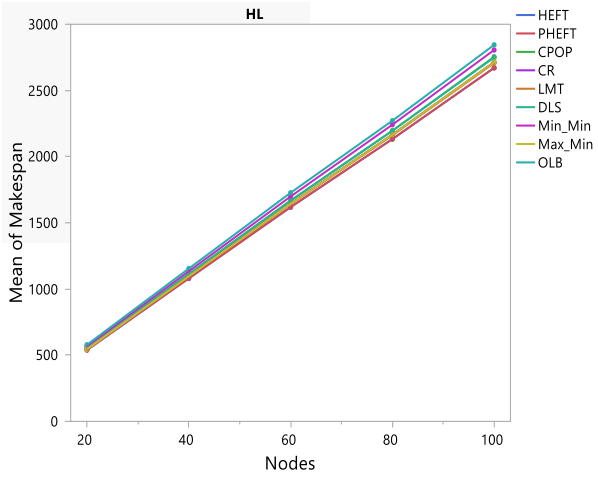


Figure 3.20: Makespan vs Nodes for HH

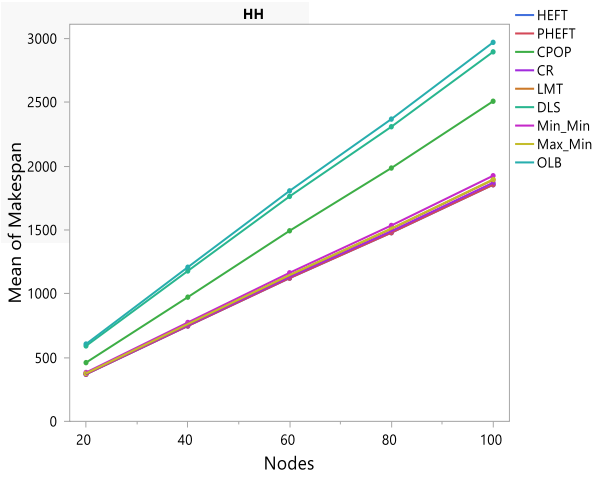


Figure 3.21: SLR vs Nodes for LL

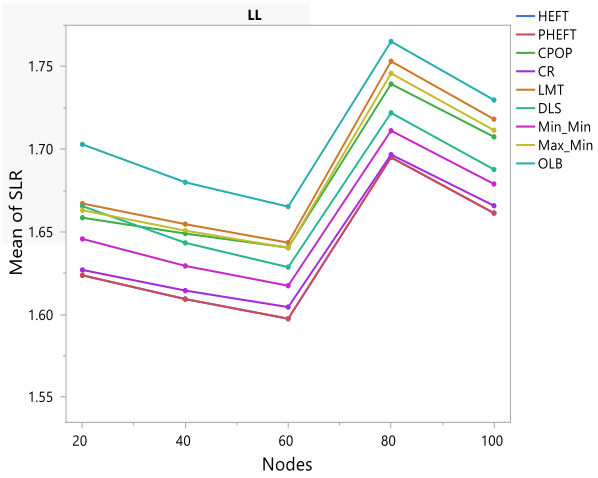


Figure 3.22: SLR vs Nodes for LH

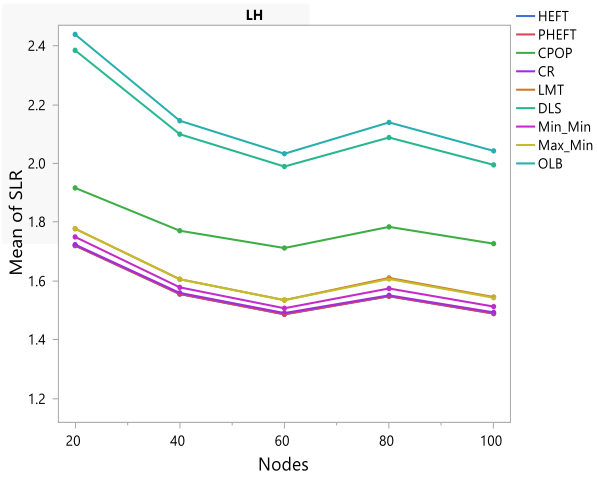


Figure 3.23: SLR vs Nodes for HL

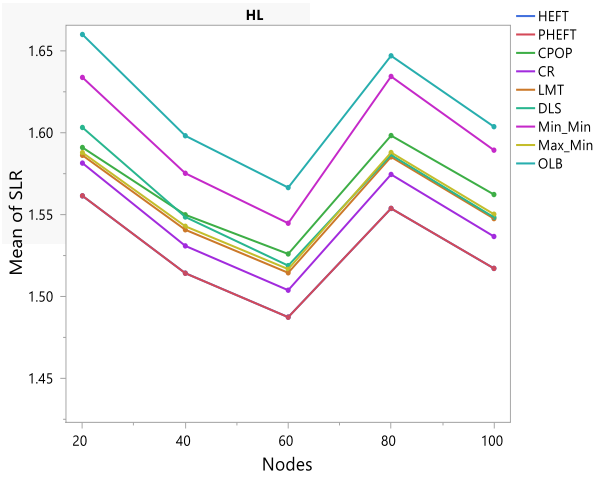


Figure 3.24: SLR vs Nodes for HH

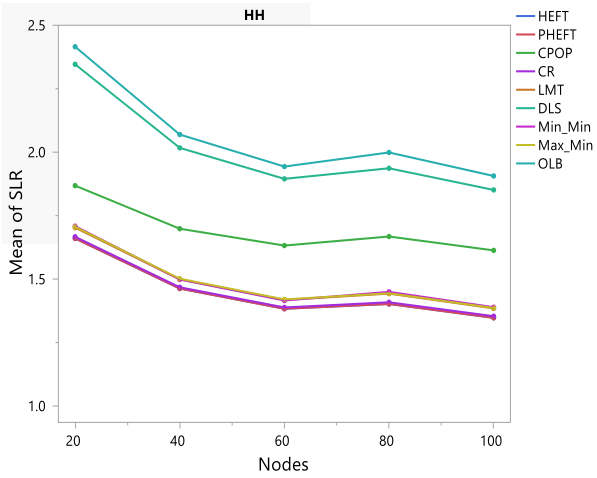


Figure 3.25: Speedup vs Nodes for LL

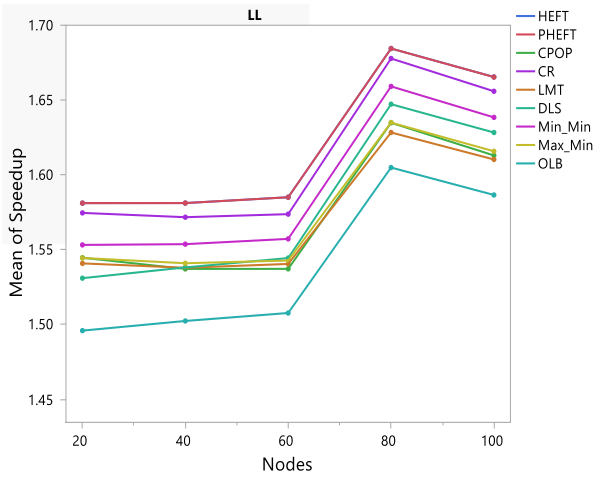


Figure 3.26: Speedup vs Nodes for LH

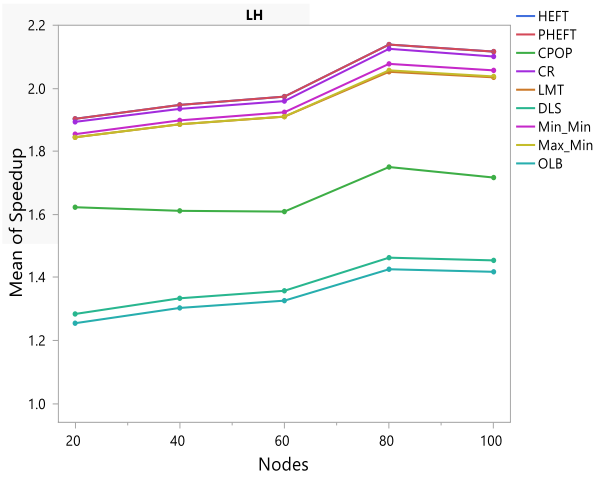


Figure 3.27: Speedup vs Nodes for HL

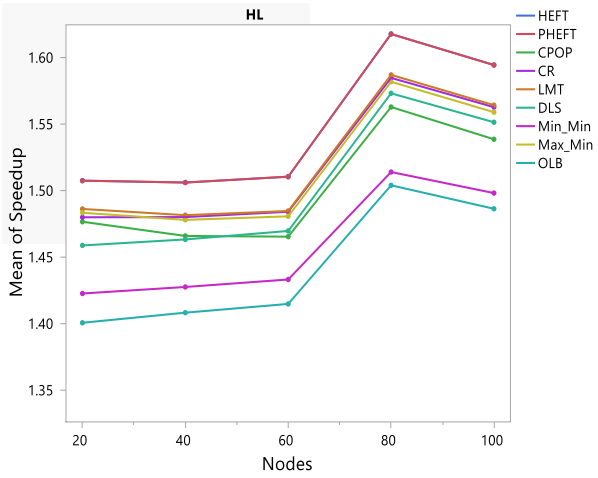
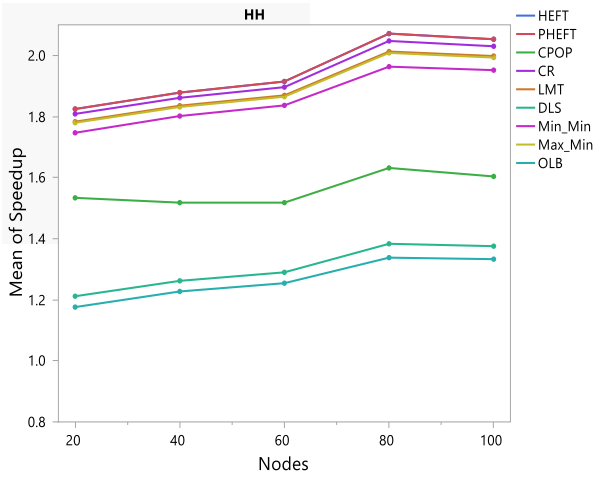


Figure 3.28: Speedup vs Nodes for HH



Comparison of Performance of OAP Algorithm

The time complexity of the OAP algorithm is much higher, which makes it impractical choice for problems with number of nodes higher than 20. Hence, a different workload was designed to compare the performance of the OAP algorithm with other algorithms. The input values to the DAG generator algorithm are listed below:

- $v = \{5,\ 8,\ 10, 12\}$

- $l = \{5: [2, 3],\ 8: [2, 5],\ 10: [3, 6],\ 12: [4, 8]\}$
- $level_{std} = \{0, 1, 3\}$
- $Con\_prob = \{0.5,\ 0.8,\ 0.9\}$
- $jl = \{2, 5, 8\}$
- $comm\_cost\_mu = 20$
- $comm\_cost\_std = \{0, 4, 8\}$
- $CCR = \{0.5, 1, 2, 5\}$
- $machine\_core\_dis = \{3: [1, 2, 4],\ 4: [1, 2, 4, 8]\}$
- $Task\_heterogeneity = 0.1\ for\ low\ heterogeneity,$

  $0.6\ \ for\ high\ heterogeneity$

- $Machine\_heterogeneity = 0.1\ for\ low\ heterogeneity,$

  $0.6\ \ for\ high\ heterogeneity$

Using these values 41,472 random DAGs were generated and used to test the nine different heuristics, to compare the performance of the OAP to the others. Results are shown in Figures 3.29 to 3.30. Figures 3.29 to 3.40 display results for makespan, SLR, and speedup, with LL, LH, HL, and HH ETC matrices, for the consistent system. While Figures 3.41 to 3.52 show the same for an inconsistent system.

Figure 3.29: Makespan vs Nodes for LL

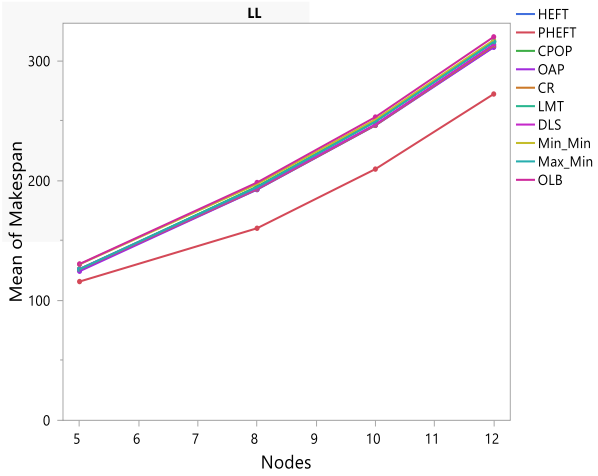


Figure 3.30: Makespan vs Nodes for LH

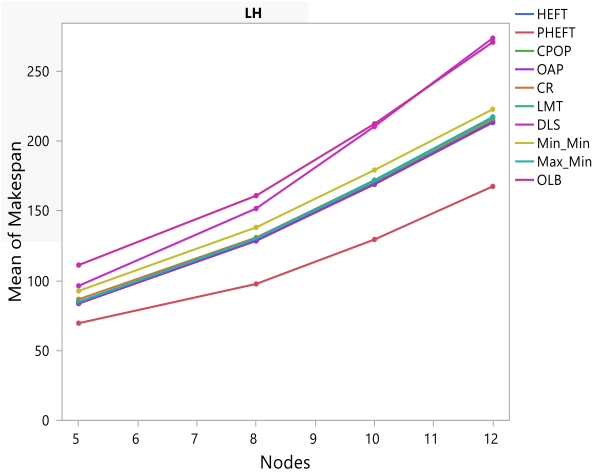


Figure 3.31: Makespan vs Nodes for HL

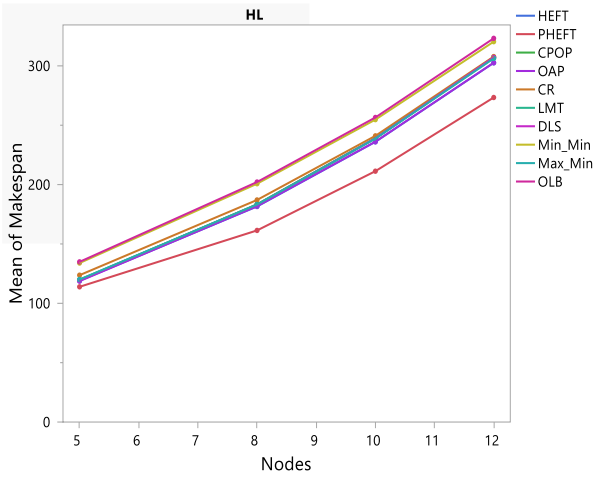
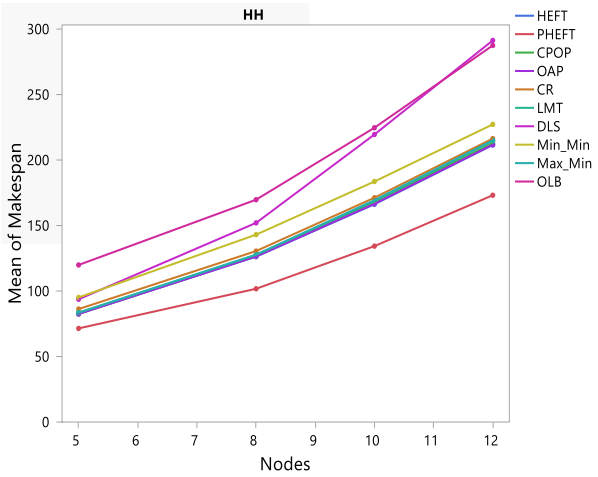


Figure 3.32: Makespan vs Nodes for HH



In LL and HL system, the performance of the algorithms measured by the makespan are very similar. As the machine heterogeneity increases, the difference in performance starts to increase.

Figures 3.33 to 3.36 show the SLR results for the nine algorithms, including OAP. In an LL system, with the exception of PHEFT, all algorithms perform similarly. With a LH, HL, and HH, distinctive differences in performance are observed.

Figure 3.33: SLR vs Nodes for LL

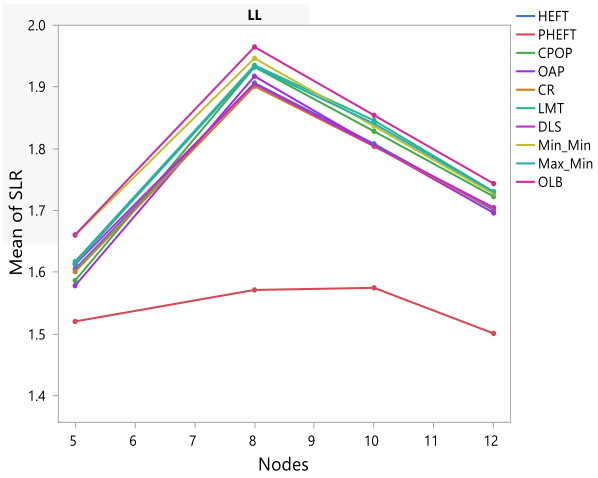


Figure 3.34: SLR vs Nodes for LH

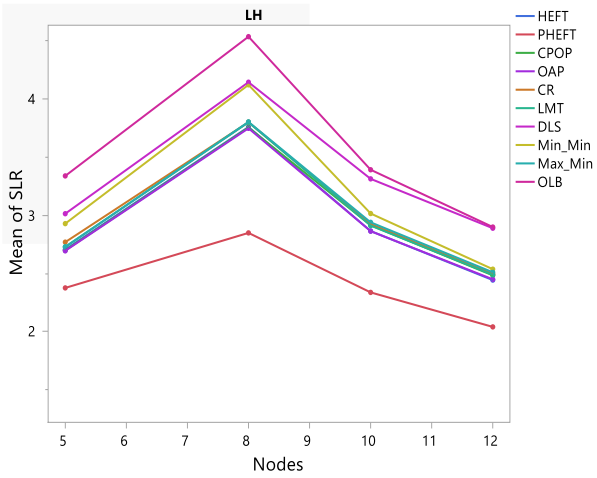


Figure 3.35: SLR vs Nodes for HL

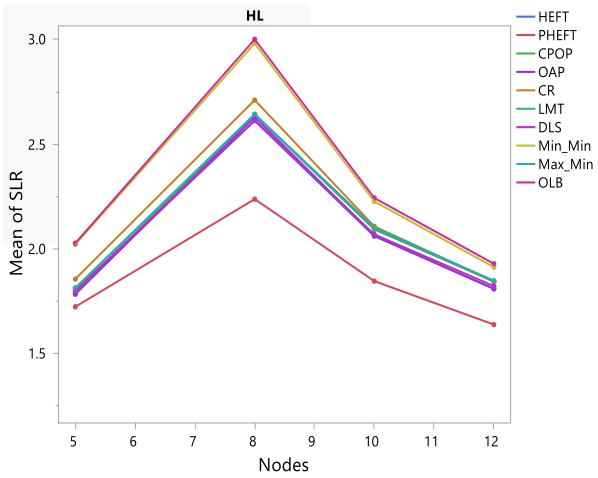
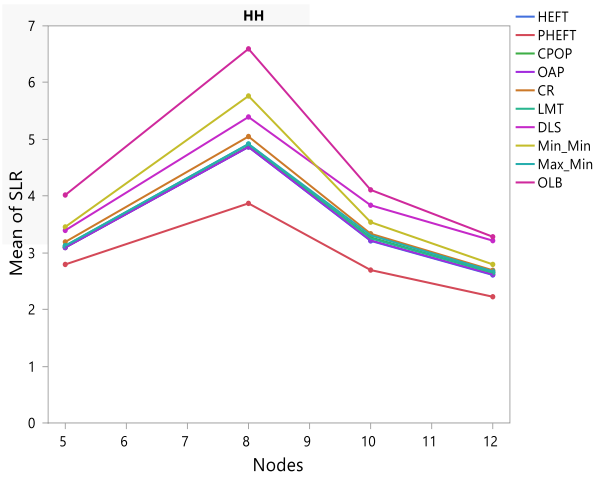


Figure 3.36: SLR vs Nodes for HH



Speedup results for the nine algorithms are shown in Figures 3.37 to 3.40. OAP algorithm shows similar performance to other algorithms.

Figure 3.37: Speedup vs Nodes for LL

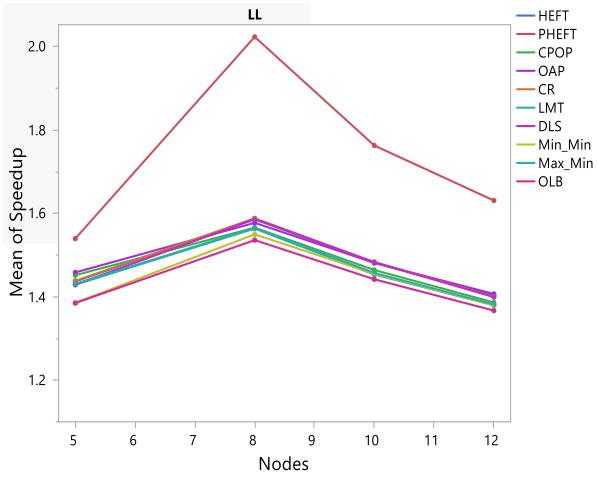


Figure 3.38: Speedup vs Nodes for LH

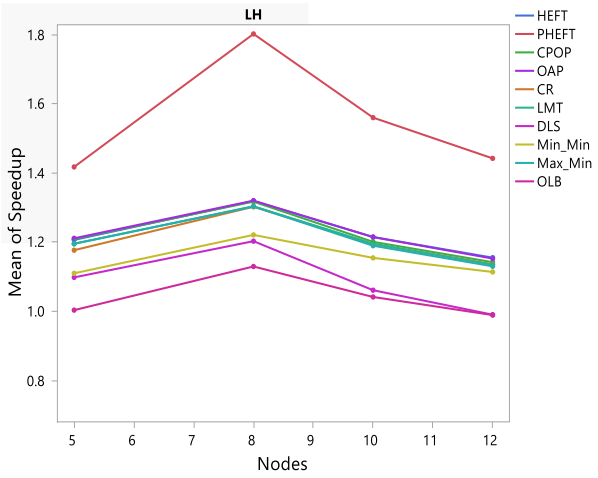


Figure 3.39: Speedup vs Nodes for HL

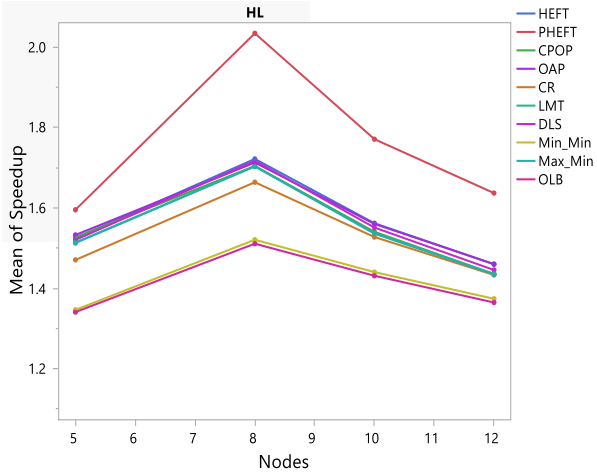
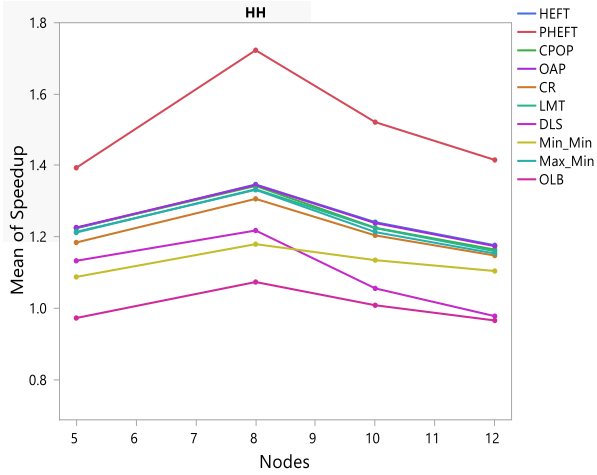


Figure 3.40: Speedup vs Nodes for HH



The results comparing nine algorithms, including OAP, for the inconsistent system are shown in Figures 3.41 to 3.52, which show same trend as the consistent system. Makespan values are shown by the plots in Figures 3.41 to 3.44.

Figure 3.41: Makespan vs Nodes for LL

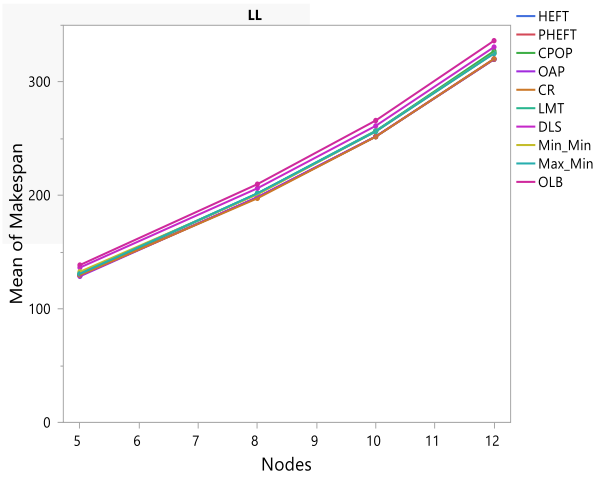


Figure 3.42: Makespan vs Nodes for LH

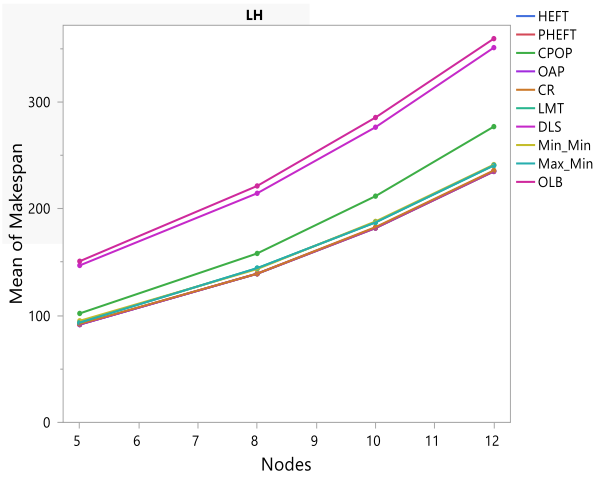


Figure 3.43: Makespan vs Nodes for HL

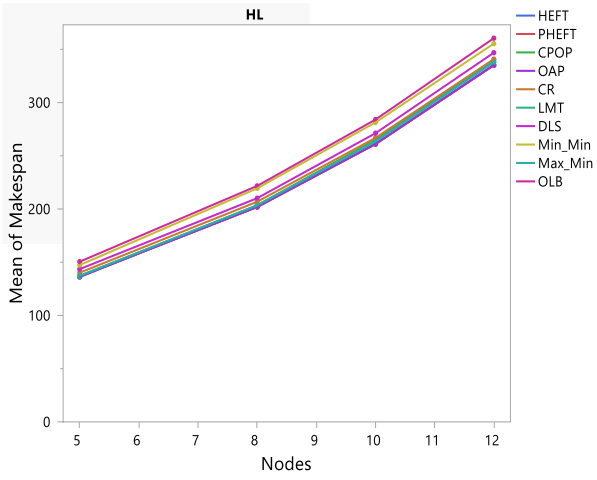
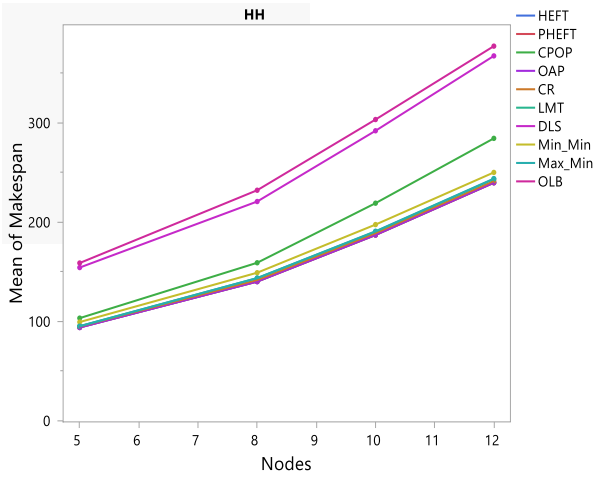


Figure 3.44: Makespan vs Nodes for HH



SLR values for the nine algorithms are shown by Figures 3.45 to 3.48, for LL, LH, HL, and HH ETC matrices, respectively.

Figure 3.45: SLR vs Nodes for LL

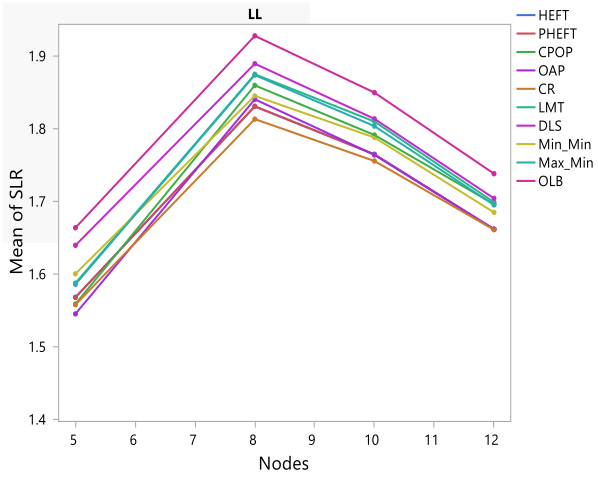


Figure 3.46: SLR vs Nodes for LH

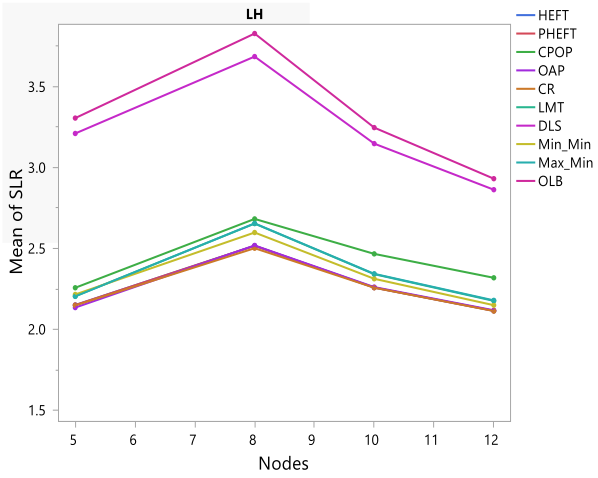


Figure 3.47: SLR vs Nodes for HL

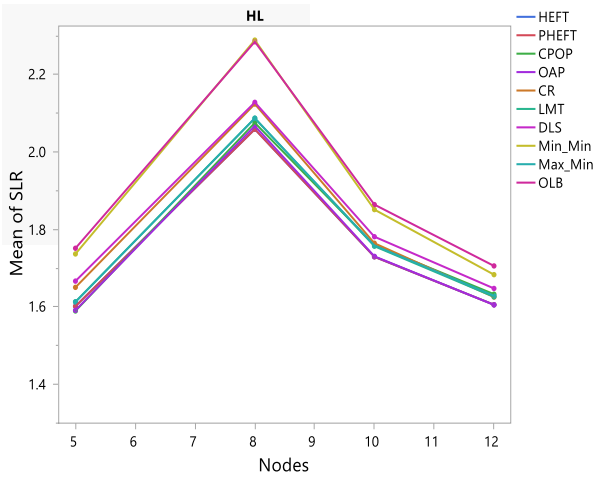
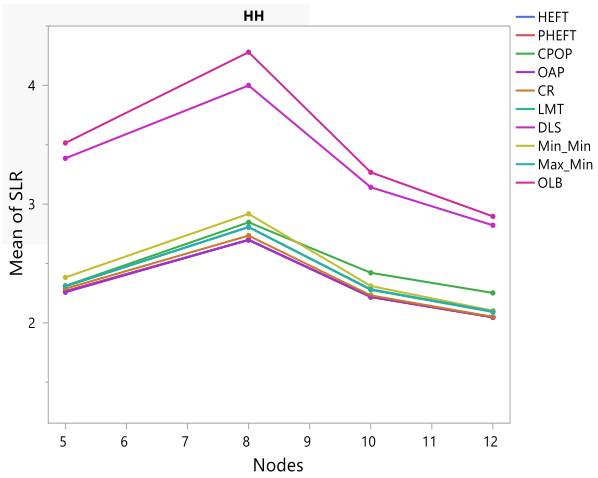


Figure 3.48: SLR vs Nodes for HH



Speedup results for the nine algorithms are shown by Figures 3.49 to 3.52, for LL, LH, HL, and HH ETC matrices, respectively.

Figure 3.49: Speedup vs Nodes for LL

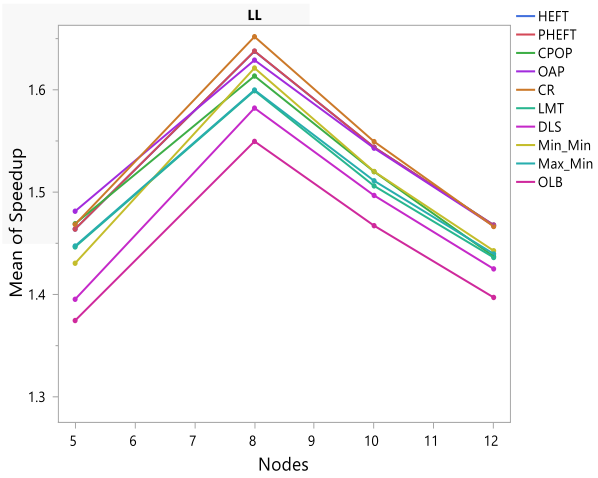


Figure 3.50: Speedup vs Nodes for LH

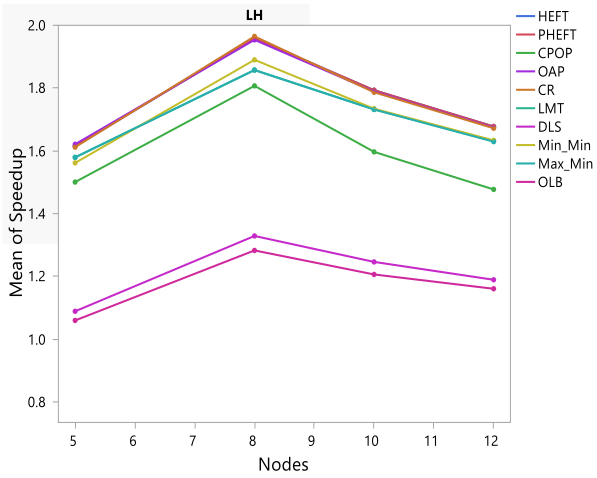


Figure 3.51: Speedup vs Nodes for HL

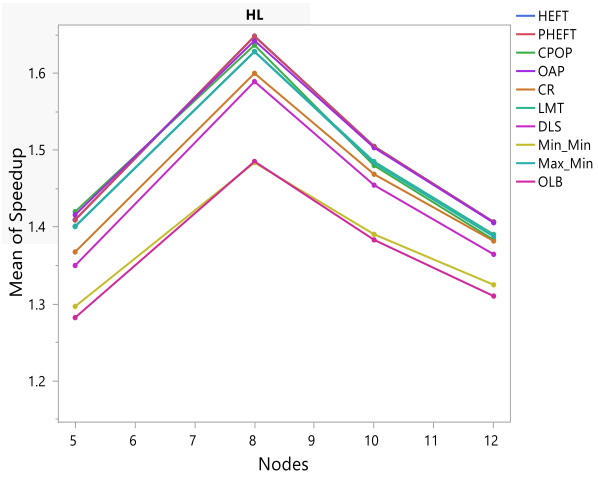
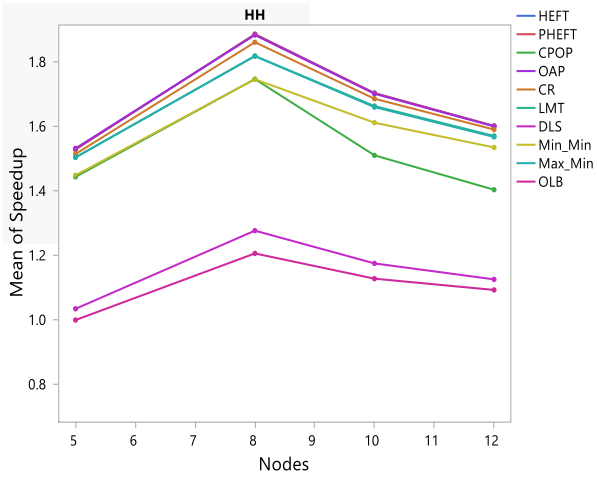


Figure 3.52: Speedup vs Nodes for HH



CHAPTER 4

SCHEDULING ALGORITHM PERFORMANCE DPENDENCIES

Many of the supercomputers today are heterogeneous, and the homogeneous ones may easily become heterogeneous by adding new processors with different architecture. These systems are used to execute a set of heterogeneous workloads, which have different levels of complexity and computational requirements. Practical solutions to the problem of mapping dependent tasks to a heterogeneous computing system are available through heuristics.

The previous chapters examined some new and existing scheduling algorithms proposed to solve this problem. As noted earlier, each of these algorithms was designed for a specific workload/computational environment, with different optimization goals, and under different assumptions to reduce the computational complexity of the problem. Different assumptions in the design phase make the comparison task difficult, therefore most of the previous works have used a set of randomly generated DAGs to ensure they are not biased to a particular type of application/system. However, research by Canon et al. [53] shows the task graphs generated by different random DAG generators are not totally

random. This means the generated population does not represent a normal distribution over all possible types of DAGs.

To be able to design a fair random DAG generator one needs to be able to first quantify characteristics of both the workflow and the system, and then use these characteristics as inputs to the DAG generator. Using these inputs one can tune the DAG generator algorithm and design different types of DAGs, thus generation a set of DAGs, which are less biased toward a specific type.

Knowing the different characteristics of the workflow and the processing environment is also important for another reason. Previous research works [5]–[8] have shown workload and computing environment's characteristics can affect the performance of the scheduling algorithm. For instance, considering the width of a DAG (the maximum number of tasks processed in parallel), and the number of processing units provided, if width of the DAG is less than the number of processors, then this DAG can have maximum parallelism and the complexity of the problem would be reduced to polynomial time. In some cases, a simple greedy algorithm produces the same results as a more complex global search algorithm, which takes much longer time to find the solution. In these cases, the system manager with access to information about the workload and the computing environment would be able to choose the most appropriate heuristic, which meets the user's and system's requirements with minimum cost and maximum performance.

In the first part of this chapter, different properties of a DAG and a heterogeneous computing system are reviewed. Some of these properties are straight forward and others are more complex to define. This work examines and aims to determine which of these properties have greater impact on the performance of the scheduling algorithms.

In the second part of this chapter, some different approaches used for generating random DAGs are reviewed. The random DAG generator developed by this author, which is used for the algorithm performance tests presented in Chapter 3, is discussed in detail. It is demonstrated how the DAG generator algorithm, using a greater number of inputs, can be effortlessly used to design a DAG with specific properties.

TASK AND MACHINE CHARACTERISTICS

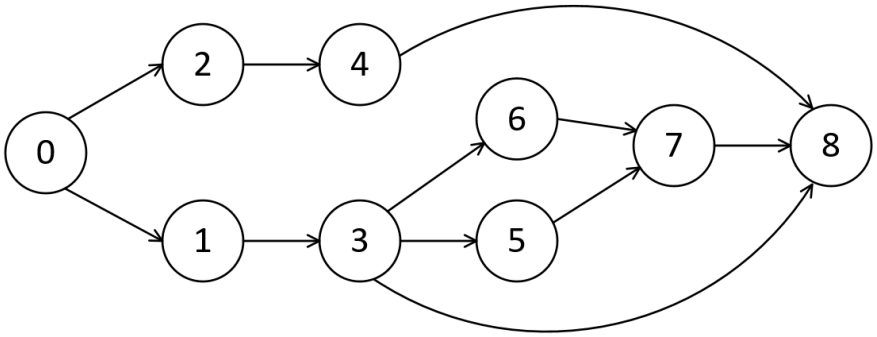
DAG Characteristics

A *directed graph* is a pair (V, E), where V is a set of nodes and $E \in V \times V$ is a set of edges between nodes which represent precedence. A directed graph is *acyclic* or DAG if there is no path with positive length such that $v_{source} = v_{destination}$. The *output degree* of a node v is equal to the number of edges which start at v. Similarly, the *input degree* of v is the number of edges which end at v. The *output* (resp. *input*) degree of a DAG is the maximum output (resp. input) degree of its vertices. *Length* of a DAG (also called level) is the maximum length of a path in it, which in a DAG with n nodes is always less than or equal to $n - 1$.

The *shape decomposition* of a DAG is the tuple $(S_1, S_2, \dots, S_k)$ where S_i is the set of nodes in level i, and k is the length of the DAG. The *shape* of a DAG is the tuple $(|S_1|, |S_2|, \dots, |S_k|)$. *Maximum shape* (resp. *minimum shape*) of a DAG is the maximum (resp. minimum) number of nodes in a level ($max_{for\ i\ in\ level}(|S_i|)$). If $|S_i| = 1$ then the only node in level i is called *bottleneck node*. A bottleneck node divides the DAG to two blocks and reduce the complexity of the scheduling procedure. If a DAG with n nodes consist of $m < n$ blocks, then the complexity of any algorithm will be divided by m. The *mass* of a block is the

maximum number of nodes in one level in the block. The *absolute mass* of a DAG is the sum of the mass values of its block ($mass^{abs} = \sum_{blocks} mass(B)$). The *relative mass,* or in general the *mass* of a DAG is $mass = \frac{mass^{abs}}{n}$, where n is the number of nodes in the DAG. For example, the shape decomposition of the DAG shown in Figure 4.1 is ({0}, {1,2}, {3,4}, {5,6}, {7}, {8}), the shape is (1,2,2,2,1,1), and the maximum shape is 2. There are 3 bottlenecks 0, 7, 8, and its absolute mass is $2 + 2 + 2 + 1 = 7$, while the relative mass is $\frac{7}{9}$.

Figure 4.1: Sample DAG



Two nodes are *incomparable* if there is no path between them. In some texts [53] the *width* of a DAG is defined as the maximum number of incomparable nodes. With this definition the width of the DAG is greater or equal to the maximum shape of the DAG. In this work, an alternative definition for the width of the DAG is used, which is same as the maximum shape.

The transitive reduction of a DAG D is a D^T which has the least number of edges and still has an equivalent path for every path in D. The transitive reduction of a DAG is the same DAG without the redundant edges. Table 4.1 summarize all of the DAG properties that may have an impact on the performance of a heuristic. Some of these properties are related, for example $n \times deg_{mean} = \frac{m}{2}$ where m is the number of edges, and $lenght \times shape_{mean} = n$.

Table 4.1: DAG Properties

Symbol	Definition
n	Number of nodes
m	Number of edges
$deg^{max}(deg_{in}^{max}, deg_{out}^{max})$	Maximum (input, output) degree
deg^{min}	Minimum degree
deg^{mean}	Mean (input, output) degree
$deg^{std}(deg_{in}^{std}, deg_{out}^{std})$	Standard deviation of the (input, output) degree
len	Length (also called height, number of levels, longest path, or critical path length)
$width$	width
sh^{max}	Maximum shape
sh^{min}	Minimum shape
sh^{mean}	Mean shape
sh^{sd}	Standard deviation of the shape
sh^{source}	Number of source nodes (nodes with zero input degree)
sh^{end}	Number of last nodes (nodes with zero out degree)
$mass$	Relative mass
Con_{prob}	Connection probability

Machine Characteristics

The number of machines and the number of cores in each processor, the computational power of the processors, and the network topology (connectivity between the processors) are some of the machine characteristics that almost always are known to the system manager. The computation power of a machine for a task defines the execution time of the task on that machine. This information is usually provided through a matrix of data called *Estimated Time to Compute* matrix which is discussed in the next section.

Depending on the workload type and the optimization goal, the scheduling algorithm may need some additional information about the computational resources [6]. Here are some examples of extra data related to the computing environment:

- The suitability or limitations of each machine for each task type (for example for security reasons).
- The energy consumption of each machine, usually provided through the *average power consumption* matrix [54] (obtained from previous executions, machine profiling, or provided by the user).
- The ability to overlap computation and communication.
- Resource failure.
- System control. If there is a central system control or it is distributed over all machines and if the scheduling algorithm is running on one machine or it is divided between different machines.

ETC matrix

A common assumption between all static scheduling algorithms is that they have an estimation of the execution time (or energy cost if the algorithm considers energy optimization) of each task on each processing unit. This data is contained in a $n \times m$ matrix, which is called Estimated Time to Compute (ETC) matrix (n is the number of tasks and m is the number of machines). $ETC(i,j)$ is the estimated execution time of task i on machine j when executed alone. Knowing the exact execution time may not be possible due to unexpected failure of the system or different input types for one task. An ETC matrix can be obtained from previous executions, task profiling, or from the user.

A row in an ETC matrix gives the estimated execution time of a task over different machines. A column in an ETC matrix indicates the estimated execution time of different tasks on one machine. To have complete and not biased test cases for the purpose of studying

scheduling algorithms, the characteristics of the ETC matrices need to be considered. Three properties can be defined for an ETC matrix:

- *Task Heterogeneity*. The variation of execution times on a machine (a column of an ETC matrix) is called the task heterogeneity over that machine and it shows how much the computation length of the tasks in an application on one machine are different from each other. High task heterogeneity occurs when the tasks in a workload are very different in terms of their computational requirements. The degree of heterogeneity for each machine (each column of the ETC matrix) could be different and the task heterogeneity of a system should be defined in a way to consider the combination of all the variations on all machines (all columns).
- *Machine Heterogeneity*. The variation of execution times of a task on different machines (a row of an ETC matrix) in a computing environment is called the machine heterogeneity of a task. It shows how much the computation cost of a task could vary if it executed on different machines. High machine heterogeneity indicates a great discrepancy between the computational power of the machines in the system. The degree of heterogeneity for each task (each row of the ETC matrix) could be different and the machine heterogeneity of a system should be defined in a way to consider the combination of all the variations on all tasks (all rows).
- *Consistency*. Consistency in an ETC matrix could be defined as how much all the tasks have the same preference order for the machines in the system. In a *machine* consistent ETC matrix, if the execution time of a task on machine m_i is lower than machine m_j, then machine m_i is faster than m_j for all other tasks in the ETC matrix. Similarly in a *task consistent* ETC matrix, if task a takes less time than task b to run on

a machine, then a should be faster than b on all other machines. In an *inconsistent* system there is no noticeable pattern on the task/machine pair preference along the ETC matrix. Thus, the order of best machines for each task could be different. A more realistic condition between the two extremes (consistent or inconsistent) is the *partial consistent* ETC which is an inconsistent matrix with some consistent parts. The consistent parts could consist of any subset of tasks and any subset of machines.

It is important to note that since there is no rule to enforce the inconsistency on ETC matrix, the result could be partially consistent. In conclusion, partially consistent matrices are a special case in the bigger group of fully inconsistent matrices.

Based on these three characteristics four different categories were proposed for ETC matrices [10]: (a) high task heterogeneity and high machine heterogeneity, (b) high task heterogeneity and low machine heterogeneity, (c) low task heterogeneity and high machine heterogeneity, and (d) low task heterogeneity and low machine heterogeneity. Considering the consistency, the number of categories increases to twelve (four categories multiplied by three possibilities: consistent, inconsistent, partial consistent).

Different ETC matrices define different parameters for the problem of scheduling in a heterogeneous computing environment and have effect on the performance of the selected algorithm. For example, if the machine heterogeneity is much higher than task heterogeneity (low task heterogeneity, high machine heterogeneity system), then the different computational requirements of a task will not have great impact on determining the order of the tasks selected for execution on the machines.

With the definition of some characteristics of an ETC matrix it is important to quantify these characteristics. This is used both for generating ETC matrices with specific required

properties and to evaluate ETC matrices in an existing workload/environment pair to assign the fittest scheduling algorithm to them. In the next section, different methods to define heterogeneity in a system, and limitations of each of those schemes are discussed.

DEFINING MACHINE HETEROGENEITY

One of the important subjects in the field of heterogeneous computing is to quantify the heterogeneity degree of a system. This is useful in predicting the performance of a heuristic for a given mapping problem (scheduling of a workload/processing units pair).

Machine and task heterogeneity are represented through different statistical metrics to evaluate ETC matrices. At the same time, different approaches are used to add machine and task heterogeneity to the process of generating ETC matrices. Here in this section, some of these different metrics and ETC generating procedures are reviewed.

Range Based Heterogeneity

In this method the task and machine heterogeneities of a system are identifies with two integer values which define the range of execution times on the ETC matrix [7], [10], [13], [55]. R_{task} is the task heterogeneity degree and $R_{machine}$ is the machine heterogeneity degree. Higher values represent higher heterogeneity. These two values are used to generate some coefficient from a uniform distribution with a range $[1, R_{task}]$, and $[1, R_{machine}]$.

To generate an ETC matrix for a system with n tasks and m machine, the execution time of task i on machine j, is generated randomly from a uniform distribution. The pseudocode for this method is presented below, in Algorithm 15.

Algorithm 15: Range Based ETC Generator

For i **from** 0 **to** $(n-1)$

$\tau[i] = U(1, R_{task})$

For j **from** 0 **to** $(m-1)$

$e[i,j] = \tau[i] \times U(1, R_{machine})$

This procedure constructs the ETC matrix one column at a time. At each iteration a sample from a uniform distribution of range $[1, R_{task}]$ is selected to be the coefficient to another sample which is driven from another uniform distribution of range $[1, R_{machine}]$, to generate an entry $e(i,j)$ of the matrix. In real system the variation on computational needs of different tasks is more than the variation on computation costs on different machines ($R_{task} \gg R_{machine}$).

This process is not very accurate, and it is possible to get a high task, low machine heterogeneous ETC with the same values, that targeted low task, high machine heterogeneity (if R_{task} and $R_{machine}$ are close). Another example of low accuracy is the tendency of low task, high machine heterogeneity ETCs to show high task, high machine heterogeneity properties due to the method, which multiplies the task heterogeneity by machine heterogeneity. Furthermore, this technique first produces an inconsistent ETC and to generate the consistent or partially consistent ETCs some extra steps are required.

Coefficient-of-Variation Based Heterogeneity

Some published works [7], [8], [56] have used the coefficient of variation of the values in an ETC matrix as a measure of heterogeneity. These works claim that the coefficient of

variation compared to the standard deviation (which used in the range-based model) is better measure of dispersion in the execution times.

To measure the task or machine heterogeneity on an ETC matrix and not to generate one, Al-qawasmeh et al. [8] defines the Average Task Covariance (ATC) and Average Machine Covariance (AMC). ATC is the average of task covariance on all tasks and AMC is the average of machine covariance on all machines. But these values alone cannot completely define the dispersion of data on an ETC matrix.

Coefficient-of-Variation Based (CVB) ETC Generator

The CVB ETC Generator algorithm takes three inputs; μ_{task} average execution time, V_{task} coefficient of variation on tasks complexity (task heterogeneity), and $V_{machine}$ coefficient of variation on machine computational capability (machine heterogeneity). In a system with n tasks, the algorithm first generates a random vector q of length n using μ_{task} and V_{task}, where $q[i]$ is an estimation of exertion of task i on an average machine. Then the values in each row of the ETC matrix are generated using $q[i]$ (for row i) and $V_{machine}$. Gamma distribution to generate the execution times was used, because of its flexibility to approximate other distributions, which makes the result closer to real life heterogeneous environment. The parameters of the gamma distribution are defined as follows: $\alpha_{task} = 1/{V_{task}}^2$, $\beta_{task} = \mu_{task}/\alpha_{task}$, $\alpha_{machine} = 1/{V_{machine}}^2$, and for task i $\beta_{machine} = q[i]/\alpha_{machine}$. The pseudocode for the CVB ETC generator algorithm is shown below.

Algorithm 16: CVB ETC Generator

$\alpha_{task} = 1/{V_{task}}^2$; $\beta_{task} = \mu_{task}/\alpha_{task}$; $\alpha_{machine} = 1/{V_{machine}}^2$

For i **from** 0 **to** $(n-1)$

$q[i] = Gamma(\alpha_{task}, \beta_{task})$

$\beta_{machine} = q[i]/\alpha_{machine}$

For j **from** 0 **to** $(m-1)$

$e[i,j] = Gamma(\alpha_{machine}, \beta_{machine})$

This algorithm successfully generates all task/machine heterogeneity pair, except the high task low machine heterogeneity. That can be solved by generating the transposed ETC matrix (first generate the columns then the rows).

Heterogeneity Based on other Statistical Properties (Skewness and Kurtosis)

The parameters discussed so far, might not precisely represent the heterogeneity of a system. For example, ETC matrices with same coefficient of variation, or same standard deviation could be very different based on other statistical properties. Because of that, some studies [8], [56] have proposed using skewness (third central moment) and Kurtosis (fourth central moment), to define task and machine heterogeneity. Skewness shows if most of the values are greater (negative skewness) or less (positive skewness) than the average. The formula for the machine skewness for task i, is given by Eq. 11.

$$S_i = \frac{[\frac{1}{m}\sum_{j=1}^{m}(ETC(i,j)-\mu_i)^3]}{(\sigma_i)^3} \quad \text{Eq. 11}$$

Where m is the number of machines, μ_i is the average execution time of task i, and σ_i is the standard deviation of task i. Results from the above mentioned studies [8], [56] show that skewness combined with covariance can predict the performance of some heuristics (Min-Min and Max-Min).

Kurtosis shows if the variation from the mean is the result of many values or a small number of values. Greater kurtosis indicates that there are small number of values which are very distant from the mean. Kurtosis for task i, is given by Eq. 12.

$$K_i = \frac{[\frac{1}{m}\sum_{j=1}^{m}(ETC(i,j)-\mu_i)^4]}{(\sigma_i)^4} - 3 \qquad \text{Eq. 12}$$

It has been demonstrated that kurtosis value also could have effect on the performance of the heuristics [8], [56]. But This effect could be different for different heuristic and happen on different thresholds. Using a regression tree to predict the performance of the scheduling algorithms has been proposed [56].

Machine Performance Homogeneity and Task-Machine Affinity to define Heterogeneity

The heterogeneity measures described till now have two issues. First, the machine heterogeneity is affected by task heterogeneity. For example, machine A, which executes first task twice faster than second task, and machine B, which runs second task twice faster than the first task. The coefficient of variation, the range based, and the skewness or kurtosis based measures indicate a heterogeneous system. But these two machines have same performance in relation to the two tasks. The second issue is related to the definition of consistency, which is a binary parameter. An ETC either is consistent or inconsistent (partially consistent is a subset of inconsistent) there is no way to show the percentage of

inconsistency in an ETC. One published study [5] introduced two other measures to address the two problems mentioned above.

The Machine Performance Homogeneity (MHP) [5] is a measure of how the performance of machines differ over all task types. To calculate MHP, ECS matrix is introduced, in which every entry is the inverse of the entry in the ECT matrix. Higher values in ECS matrix indicate a better choice for task assignment. The columns in the ECS matrix need to be sorted in a way that the sum of the values in the columns are in the descending order. Eq. 13 gives the general formula for MPH.

$$MPH = \frac{\sum_{j=1}^{m-1}(wm_j \sum_{i=1}^{n} wt_i \,.a(i,j)/wm_{j+1} \sum_{i=1}^{n} wt_i \,.a(i,j+1))}{m-1} \qquad \text{Eq. 13}$$

Here n is the number of tasks, m is the number of machines, wm_j is the weight assigned to machine j (some machines may be more preferable, for example due to security reasons), wt_i is the weight of task i, and $a(i,j)$ is the ECS entry on row i and column j.

The Task-Machine-Affinity (TMA) [5] is a degree which shows how much a group of tasks are suitable to be assigned to a group of machines. TMA is define using *singular value decomposition* [57], which can be performed on any matrix to determine its rank. The idea is to use the ratio of the minimum singular value to the maximum one. If the performance of two machines with respect to one task type is similar, then the ratio will be smaller. The first step to find the TMA is to normalize the columns of ECS to have sum of 1 (this is to make TMA as independent as possible from MPH) and compute the singular value decomposition of it. Eq. 14 gives the formula for TMA.

$$TMA = \frac{(\frac{\sum_{i=2}^{m} \sigma_i}{m-1})}{\sigma_1} \qquad \text{Eq. 14}$$

Here m is the number of machines, and σ_i is the i^{th} singular value. The higher σ_1, the maximum singular value, the more the columns are correlated.

TMA and MPH are also losing some information by presenting one value for the heterogeneity to make the comparison process easier in return. Results have demonstrated successful classification of ECS matrices to four classes of low task low machine heterogeneity, low task high machine heterogeneity, high task low machine heterogeneity, high task high machine heterogeneity [5].

DAG GENERATOR ALGORITHMS

To produce a fair comparison of the scheduling heuristics, the Random DAG generator needs to generate test cases that are uniformly distributed over all possible DAG types. There are several research studies, which evaluate the DAG generating tools in terms of their effect on the performance of the scheduling algorithms. One study [58] explored generation methods based on the number of edges, length of the task graph, and output degrees. Martinez et al. [59] did the same thing using different properties, such as number of nodes on the critical path, the width of the DAG, and the number of edges. Kwok et al. [60] compared some scheduling algorithms using random and non-random generated DAGs. Some of the DAG generation tools, which have been proposed in the literature, are discussed in the next section.

Generation Tools and Data Sets

The Task Graphs For Free (TGFF) [61] takes the number of nodes and the maximum input and output degrees and generates a random graph by iteratively adding a node to the

initial single-vertex graph. The process continues until it reaches the number of required nodes.

DAGGEN is another tool which was proposed by Dutot et al. [62]. This algorithm uses the layer-by-layer method and takes five inputs; the number of nodes, the width, the regularity parameter, the density, which determine the number of nodes in each layer, and the jump parameter. The regularity degree defines the average number of nodes in each level and the jump parameter define the number of levels the edges jump to connect nodes.

GGen [58] used both layer-by-layer method and degree-based method to generate random DAGs, as well as some DAGs representing famous applications like the Strassen multiplication algorithm, the Cholesky factorization, and the recursive Fibonacci function [55]. The Pegasus workflow generator [63], the XL-STaGe [64], and the Random Workflow Generator [65] are some of the other generation tools.

In addition to DAG generation tools, there are some data sets available for testing purpose. The Standard Task Graph (STG) set [66] used in some research works to compare scheduling algorithms, has DAG samples with 50 to 5000 nodes. STG uses four different methods to generate DAGs. two of them are based on the Erdős-Rényi algorithm [67] and the other two are layer-by-layer approaches. In the layer-by-layer method the nodes are distributed in layer with the average number of 10, with three possible connectivity values. None of these methods guarantee that the actual values of the parameters are same as what they claimed to be (the value of input parameters like number of layers, or fan-in/fan-out). Therefore, to find the true characteristics of the DAGs there needs to be another step to examine the DAGs separately. The STG set contains some real application DAGs, like robot control, sparse matrix solver and SPEC fpppp program in addition to the random ones [55].

Some of the other random DAG data sets, proposed and used in the literature are designed for the other research fields like project management, and some others are no longer available [55].

PROPOSED DAG GENERATING ALGORITHM

The random DAG generators mentioned in the previous section use a small number of input parameters; hence, the results they produce are not completely distributed over all possible DAGs and ETC characteristics. For this reason, this research work developed and implemented a novel DAG generator tool with more input parameters to define DAG and ETC matrix characteristics. In addition, each DAG after being generated is examined to check if there is any difference between the actual properties and the input parameters (defined characteristics). The algorithm reports the former one, which makes the classification more accurate.

The Random DAG Generator is based on the layer-by-layer technique and takes some parameters as input, generates the random DAG and the ETC matrix, then examines the DAG to find its exact characteristics. Table 4.2 lists the inputs and derivative parameters used by the algorithm.

Table 4.2: List of Inputs and Derivative Parameters

Input parameters	
Symbol	Definition
v	Number of nodes
l	Number of levels
l_{std}	Standard deviation of the number of tasks in each level
jl	Jump level (number of levels an edge jump to connect two nodes

Con_prob	Connection probability
$comm\_cost\_mu$	Average Communication Cost
$comm\_cost\_std$	Standard Deviation of Communication Cost
CCR	Computation to Communication Ratio
$machine_{core_{dis}}$	Processor distribution and number of cores
$Task\_heterogeneity$	Task heterogeneity
$Machine\_heterogeneity$	Machine heterogeneity
$consistency$	If the machines are consistent or not

Derivative parameters	
Symbol	Definition
$level$	Number of levels (if it is different from l)
$width$	Width
$edge\_num$	Number of edges
end_nodes	Number of end nodes (nodes with zero out degree)
$connection\_ratio$	Connection ratio (if it is different from Con_prob)
max_deg_out	Maximum output degree
avg_deg_out	Average output degree
max_deg_in	Maximum input degree
avg_deg_in	Average input degree
Seq_exe_time	Sequential execution time
CP_min	Minimum length of the critical path
P_degree	Parallelism degree of the DAG

This algorithm always generates a DAG with one start node (a node without any predecessors). Those DAGs with more than one start node can easily change to a DAG which starts with one dummy node and some zero weight edges to the initial start nodes. Another assumption is that the number of edges is always greater than or equal $v - 1$, so the generated DAG is always at least a tree (there is no node without connection). Therefore, this algorithm would never generate some types of DAGs, like empty DAG. The next assumption is that the level of a node is always same as the depth of it.

Algorithm start with assigning one node to each level, then using normal distribution it assigns the remaining nodes to other levels (except first level). The mean of the normal

distribution is by $level_{mean} = \frac{remaining_{tasks}}{l-1}$, where $remaining_{tasks} = v - l$, and the standard deviation is $level_{standard\ deviation} = L_{std}$.

The generated result is checked to remove the possible negative values (those levels with negative additional nodes remain with one node). Next, the algorithm forms the connections between the node in different levels. A task in a level i should at least be connected to one task from level $i - 1$ (because level and depth are same). After adding one edge for each node, the remaining number of edges is defined by the connection probability. Another parameter is defined here, the connection ratio, which is the ratio of the number of edges to the maximum possible number of edge (complete graph). A DAG with v nodes can have $[\ (v-1)\ ,\ \frac{v(v-1)}{2}\ ]$ edges. Initially the connection ratio is given by the $least\ number\ of\ edges = v - 1$, where the $number\ of\ edges\ in\ a\ complete\ graph = v(v-1)/2$, and $connection\_ratio = \frac{v-1}{(\ v(v-1)/2\ )} = \frac{2}{v}$.

The process of adding edges continues until the connection ratio is close enough to the connection probability value. Higher connection probability means higher number of edges (connection probability of one indicates a complete DAG).

Jump level indicates the maximum number of levels an edge can jump. The level of the source node for each remaining edge is selected randomly from $[1, l-1]$ (the last level does not have an output edge), while the destination level is selected from $[sourcce_{level} + 1\ ,\ \max(l, source_{level} + jl)]$. After selecting the levels, the nodes in those levels are selected randomly for the source and destination node. Before they are added to it, they checked for the duplicity.

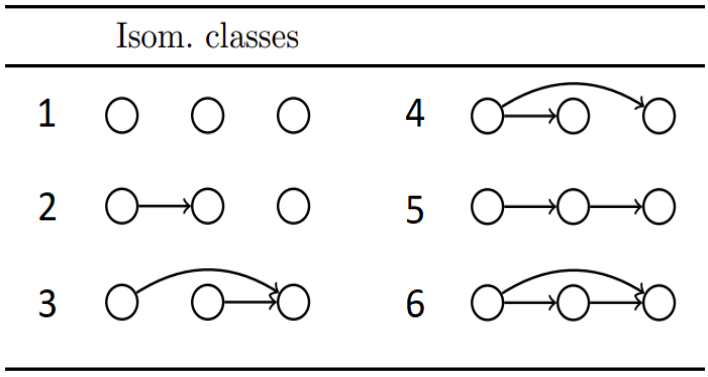
The weights of the connections are derived from a normal distribution, where the mean is $comm\_cost\_mu$ and standard deviation is $comm\_cost\_std$. The negative values are taken out.

The next step, the ETC matrix is generated. Here the coefficient of variation method with gamma distribution is used. At each iteration, first a random value from a gamma distribution with $\alpha_t = 1/(Task\_heterogeneity)^2$ and $\beta_t = \mu_{task}/\alpha_t$ is generated. The μ_{task} is the average execution time for a task, which is calculated using the Computation to Communication Ratio (CCR), $\mu_{task} = CCR \times comm\_cost\_mu$. Then, the execution time of each task on each machine is obtained using a random value from a gamma distribution with $\alpha_m = 1/(Machine\_heterogeneity)^2$, and $\beta_m = (random\ value\ for\ the\ task)/\alpha_m$. This generates an inconsistent ETC matrix. To generate a consistent matrix, the inconsistent one is sorted twice, once over the rows and one over the columns.

After generating the random DAG and the ETC matrix, the DAG is examined to find the derivative parameters from Table 4.2. The level with the maximum assigned number of nodes defines the $width$ of the graph. The Seq_exe_time is the final execution time (makespan) of the DAG if there is only one machine available (considering the fastest machine). The CP_min is the minimum length of the critical path and is a lower bound on the makespan of the DAG. It is calculated by adding the execution time of the task on the critical path, assuming they all use the fastest machine. The parallelism degree of the DAG is the ratio of sequential execution time to the execution time of the DAG, if it is fully parallelized. It should be noted, this time is different from CP_min because it considers not just the tasks on the critical path.

Braun et al. [55] defined the uniformity of a random DAG generator as the ability to generate each isomorphic class of DAGs with the same probability. For example, Figure 4.2 shows all isomorphic DAGs that can be generated with three nodes. If the DAGs generated with a DAG generator tool are from each of these six classes with the probability of $^1/_6$ then that random DAG generator is uniform.

Figure 4.2: Isomorphic DAGs with Three Nodes



The random DAG generator introduced in this dissertation, only generates three classes out of these six classes (classes four to six). Hence, the DAG generator is considered uniform, if it generates each of these classes with the same probability of $^1/_3$ (for the case of three nodes).

The algorithm was tested with four and five nodes. Table 4.4 and Table 4.6 show all classes of trees, the DAG generator can generate with four and five nodes, respectively. Table 4.3 and Table 4.5 list the input parameters for the tests. The number of isomorphic classes of DAGs with four nodes and five nodes is listed under each tree type. For a DAG with four nodes, there are 16 isomorphic classes of DAGs, thus for the proposed algorithm to be uniform, it needs to generate each of these classes with the probability of $^1/_{16}$. Since the algorithm has more input parameters, the output percentage of each of these classes is more

manageable. For example, with three levels the probability of generating each of the two tree classes is close to 50%. With fair selections for the connection probability, the probability of generating a DAG from each of the three isomorphic classes in the first tree class is 16.66%, while the probability of generating a DAG for isomorphic classes of second tree type is 12.5%.

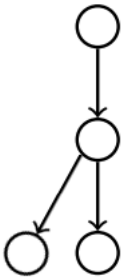
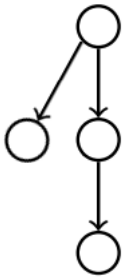
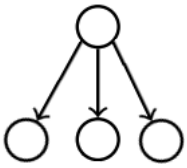
The input values for generating DAGs with four nodes are listed below in Table 4.3. For each level, six different types of random DAGs are generated, and for each of them 25 samples are generated. Thus 150 DAGs, in total, are generated for each level. In this scenario, only number of nodes, levels, connection probability, and jump level are used. Since, other inputs are not influencing the result of this test, they are assigned with some constant values.

Table 4.3: Test Inputs for Four Nodes

Node #	*levels*	*Con_prob*	*Jump level*	Other input parameters
4	2, 3, 4	0.5, 0.7, 0.9	1, 2	NA

The results of the test demonstrate that the proposed algorithm is uniform over tree classes with specified level, but not uniform over isomorphic classes with same level. With three levels instead of the uniform probability of 20%, the observed results are 16.6% and 25%.

Table 4.4: Trees Generated with Four Nodes

4 levels	3 levels		2 levels
			
100%	50.33%	49.66%	100%
8 Isom.	3 Isom.	4 Isom.	1 Isom.
~12%	~16.6%	~12%	100%

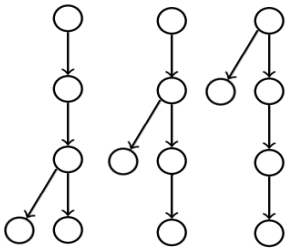
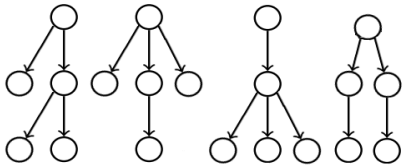
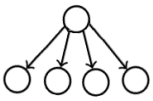
The input values used to generate DAGs with five nodes are listed in Table 4.5. For each level, nine different types of random DAGs are generated, and for each of them 25 samples are generated, producing a total of 225 DAGs.

Table 4.5: Test Inputs for Five Nodes

Node #	*levels*	*Con_prob*	*Jump level*	Other input parameters
5	2, 3, 4, 5	0.5, 0.7, 0.9	1, 2, 3	NA

As the results for four nodes, the algorithm is uniform over tree classes with specified level, but not uniform over isomorphic classes with same level. To be uniform the probability of each isomorphic class of DAG should be $1/175$ or 0.57%.

Table 4.6: Trees Generated with Five Nodes

5 levels	4 levels			3 levels				2 levels
								
100%	32.87%	33.52%	33.6%	25%	26.3%	24.57%	24.1%	100%
64 Isom.	20 Isom.	28 Isom.	32 Isom.	10 Isom.	6 Isom.	4 Isom.	10 Isom	1 Isom.
~1.5%	~1.6%	~1.2%	~1.04%	~2.5%	~4.1%	~6.25%	~2.5%	100%

Those classes, not generated with the proposed algorithm in this work, belong to one of the three groups listed below.

1. Empty DAGs, which are basically a bag of tasks, rather than DAGs.
2. Some nodes with no connections and DAGs with fewer number of nodes. (Smaller DAGs are already covered with smaller values of v as input.)
3. DAGs with more than one start node. (The equivalent version of this type of DAGs has a dummy task with zero execution time, and zero communication cost to the initial start node.)

Therefore, DAGs from other classes are either already covered with different number of nodes, or are not related to the topic of scheduling a group of dependent tasks (represented as a DAG). Consequently, the proposed Random DAG Generator covers a wide range of DAG types and is uniform over tree classes. With different parameters as input, it is

easier to generate the required DAG sets. This feature is helpful in studying the effect of system characteristics on the performance of the scheduling algorithms.

INFLUENCE OF DAG CHARACTERISTICS ON PERFORMANCE

Multiple studies have suggested that heuristics show inconsistent behavior in different environment [5], [8], [10], [53], [56]. Some of these demonstrated changes in the relational performance of Min-Min and Max-Min algorithms, under different values for task and machine heterogeneities [8], [56]. One study showed that different DAG types can lead to the same scheduling algorithm performing differently [53]. Previous studies have been based on small examples, or a group of few special types of DAGs. None of them has explored the influence of workload/computing environment characteristics on the performance of scheduling algorithms, for the whole set of randomly generated DAGs with broad range of characteristics. The following section explores this influence on performance of scheduling algorithms, on some of the greedy algorithms reviewed in Chapter 2. The same results from the experiments presented in Chapter 3 are used here, while the system properties are the inputs to the DAG Generator algorithm from Table 4.2.

The results explore the correlation between each of the DAG input parameters, to the SLR measure of each of the nine scheduling algorithms discussed in Chapter 3. Correlation value of 1 indicates a perfect linear relationship between the two variables, while a correlation of 0 indicates no relationship at all. Correlation values greater than 0.7 indicate strong correlation, value of 0.5 is moderate, while values below 0.3 indicate weak correlation. The correlations for machine heterogeneity and task heterogeneity equal 0, since the is grouped by machine and task heterogeneity.

Figures 4.3 to 4.10 display the results for eight different classes of ETC matrices. Figure 4.3, which shows results for consistent system with low task low machine heterogeneity, indicates that computation to communication cost ratio, and average number of levels have a negative impact on SLR of all algorithms except PHEFT. The higher communication cost ratio and average number of levels, result in lower SLR values for the eight algorithms. In contrast, number of machines has a significant negative effect on the SLR of the PHEFT algorithm alone. Increasing number of machines results in lower SLR of PHEFT.

Figure 4.3: Correlation of DAG Input Parameters for Consistent System with Low Task Low Machine Heterogeneity

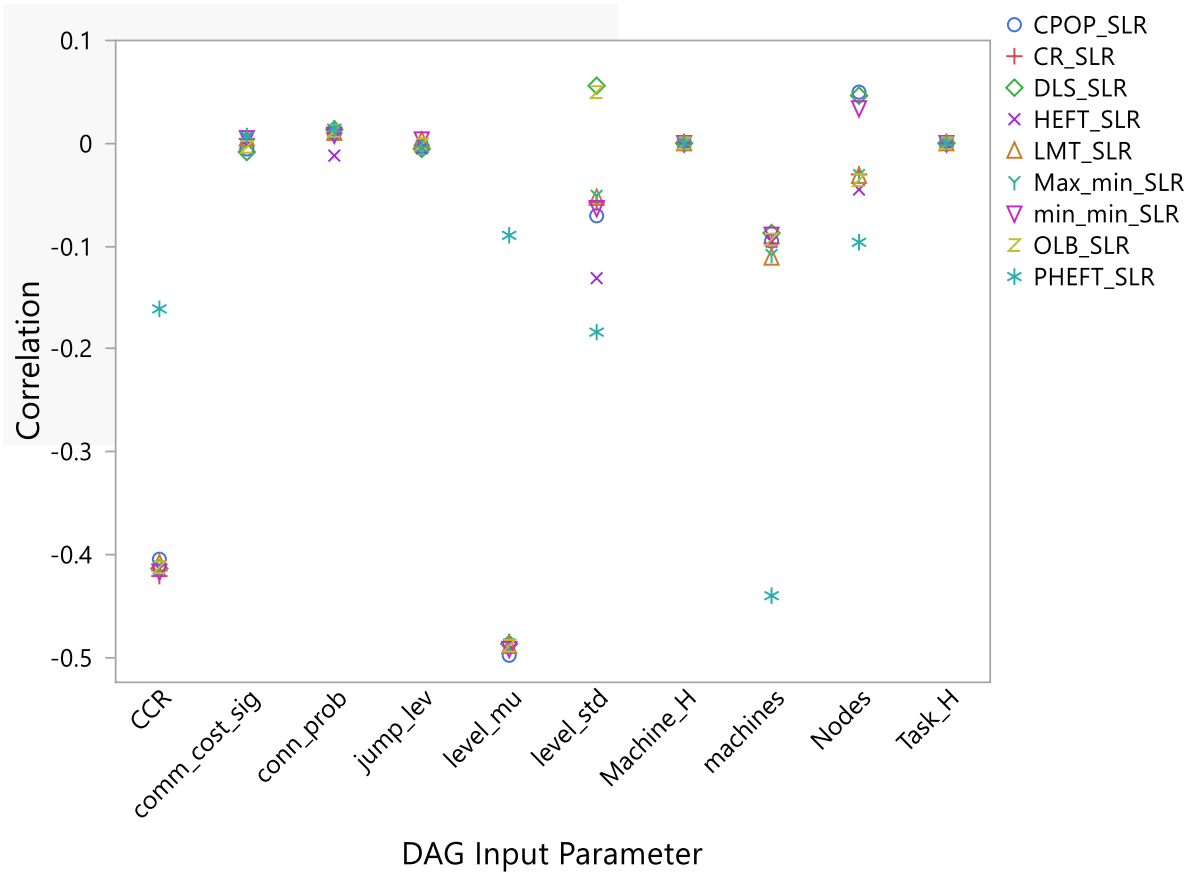


Figure 4.4 shows correlation results for consistent system with low task high machine heterogeneity. Data suggests that increasing the machine heterogeneity increases the

correlation between computation to communication cost ratio, and communication cost and SLR of PHEFT algorithm. While the correlation between these two DAG input parameters and the SLR of the other eight algorithms are reduced. The negative correlation between average number of levels and SLR of all algorithms except PHEFT remains significant.

Figure 4.4: Correlation of DAG Input Parameters for Consistent System with Low Task High Machine Heterogeneity.

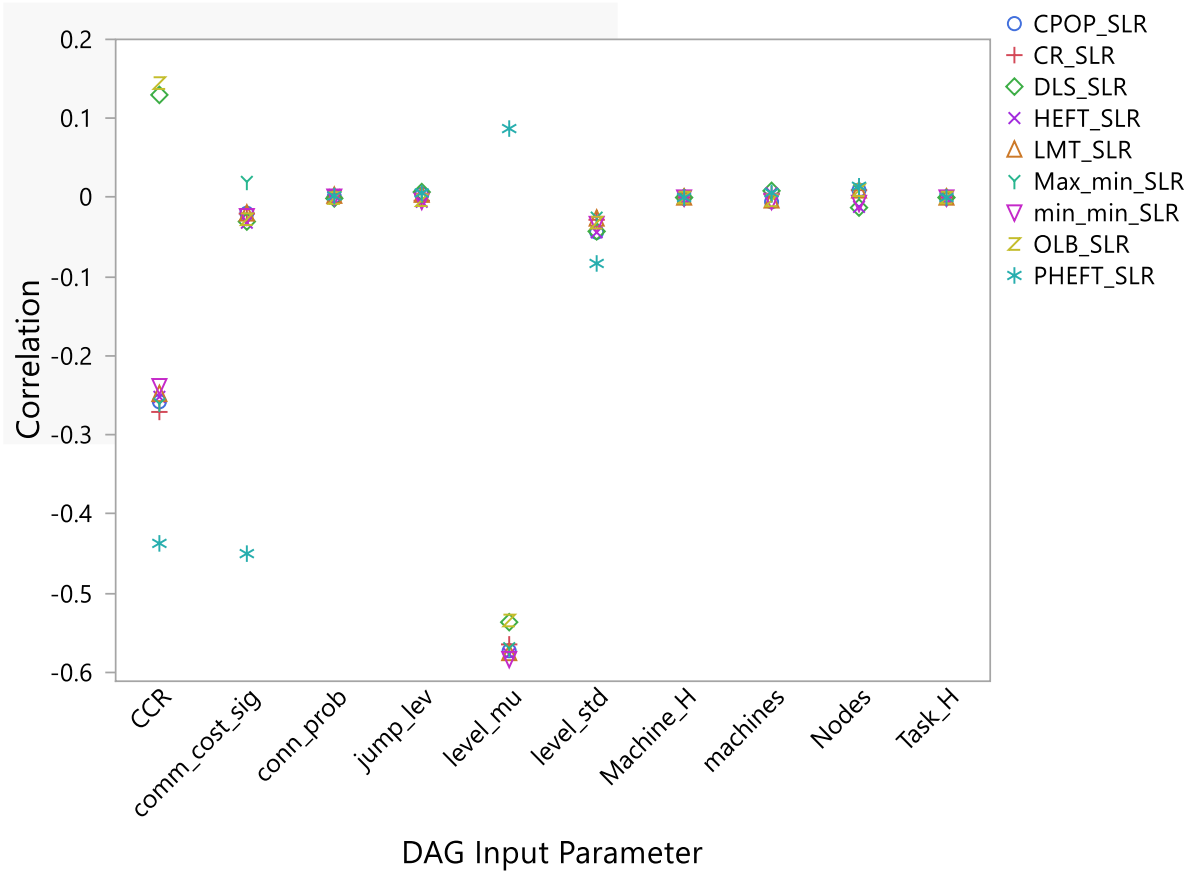


Figure 4.5 shows correlation results for consistent system with high task low machine heterogeneity. Here the number of machines and the variation of the number of tasks in each level (*level_std*) have higher negative impact on the SLR of PHEFT algorithm, compared with the other eight algorithms. In contrast, the other eight algorithms show stronger negative correlation between number of levels and SLR, compared to PHEFT algorithm.

Figure 4.5: Correlation of DAG Input Parameters for Consistent System with High Task Low Machine Heterogeneity.

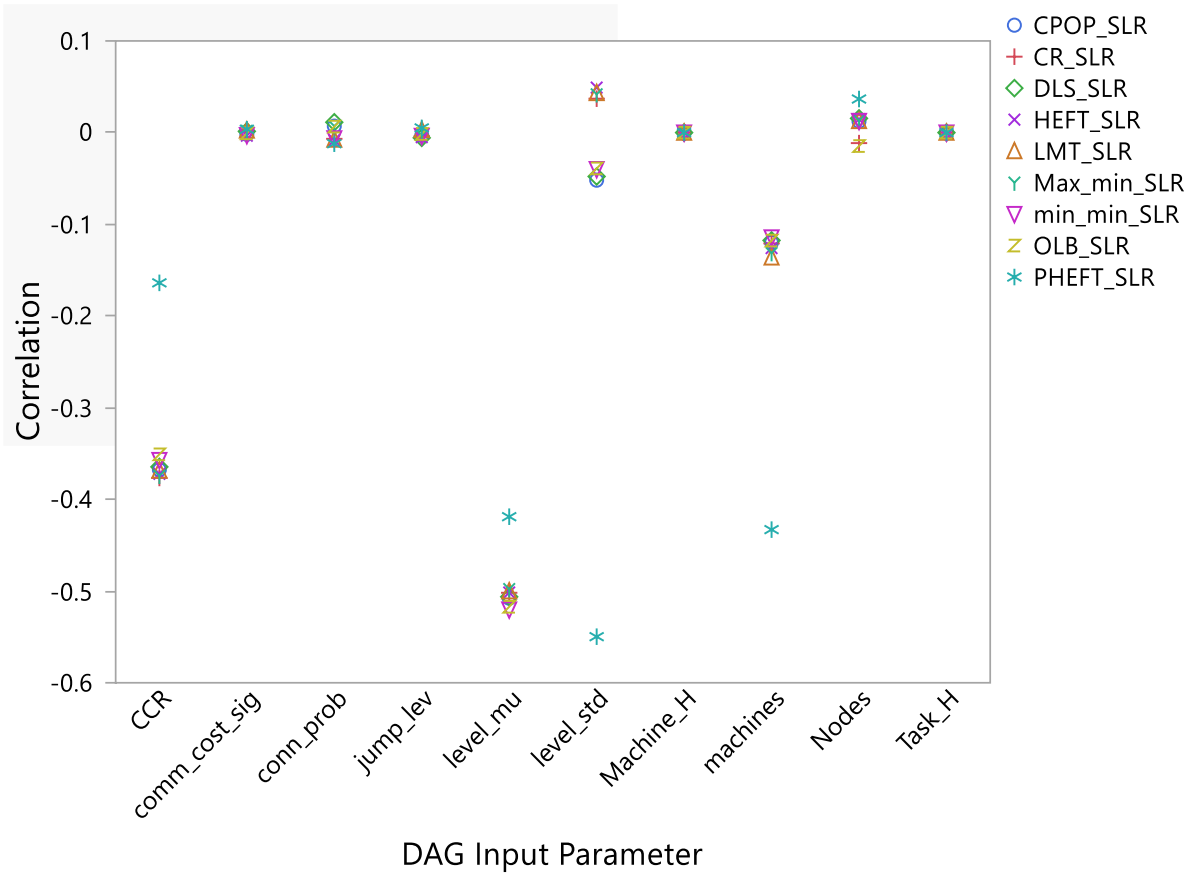
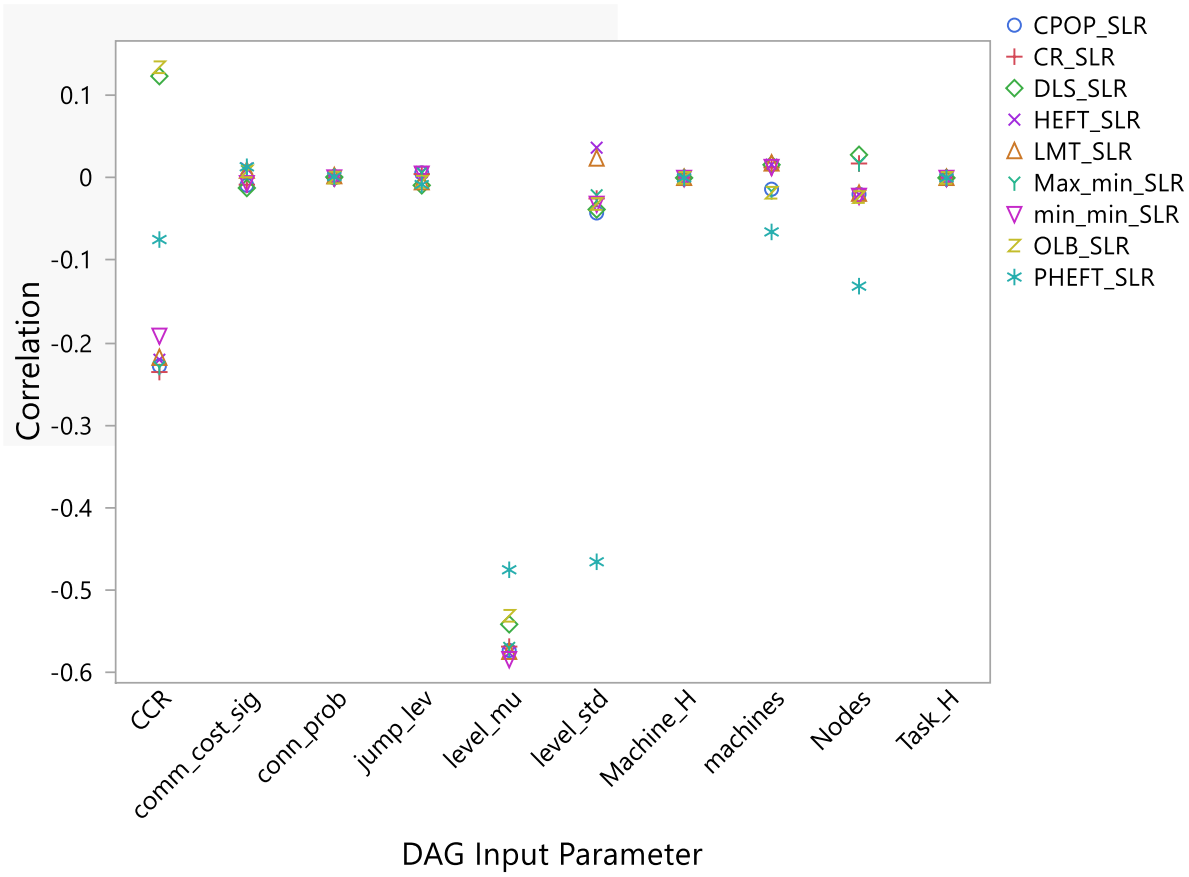


Figure 4.6 shows correlation between DAG input parameters and SLR algorithm values for consistent system with high task high machine heterogeneity. The only DAG parameters showing moderate negative correlation with SLR are the average number of levels, which is similar for all algorithms, and the variation of the number of tasks in each level, which is only significant for the PHEFT algorithm.

Figure 4.6: Correlation of DAG Input Parameters for Consistent System with High Task high Machine Heterogeneity.



Figures 4.7 to 4.10 display the correlation results for an inconsistent system. Figure 4.7 shows correlation results for an inconsistent system with low task low machine heterogeneity. In general, larger number of DAG input parameters show correlation with SLR parameter, compared to the consistent system. The number of levels and the variation of the number of tasks in each level show negative correlation with SLR for some of the algorithms, but not PHEFT. On the other hand, SLR of PHEFT shows significant negative correlation with computation to communication cost ratio and number of machines. SLR of CPOP algorithm also shows negative correlation with computation to communication cost ratio, while SLR for HEFT shows negative correlation with number of machines.

Figure 4.7: Correlation of DAG Input Parameters for Inconsistent System with Low Task Low Machine Heterogeneity

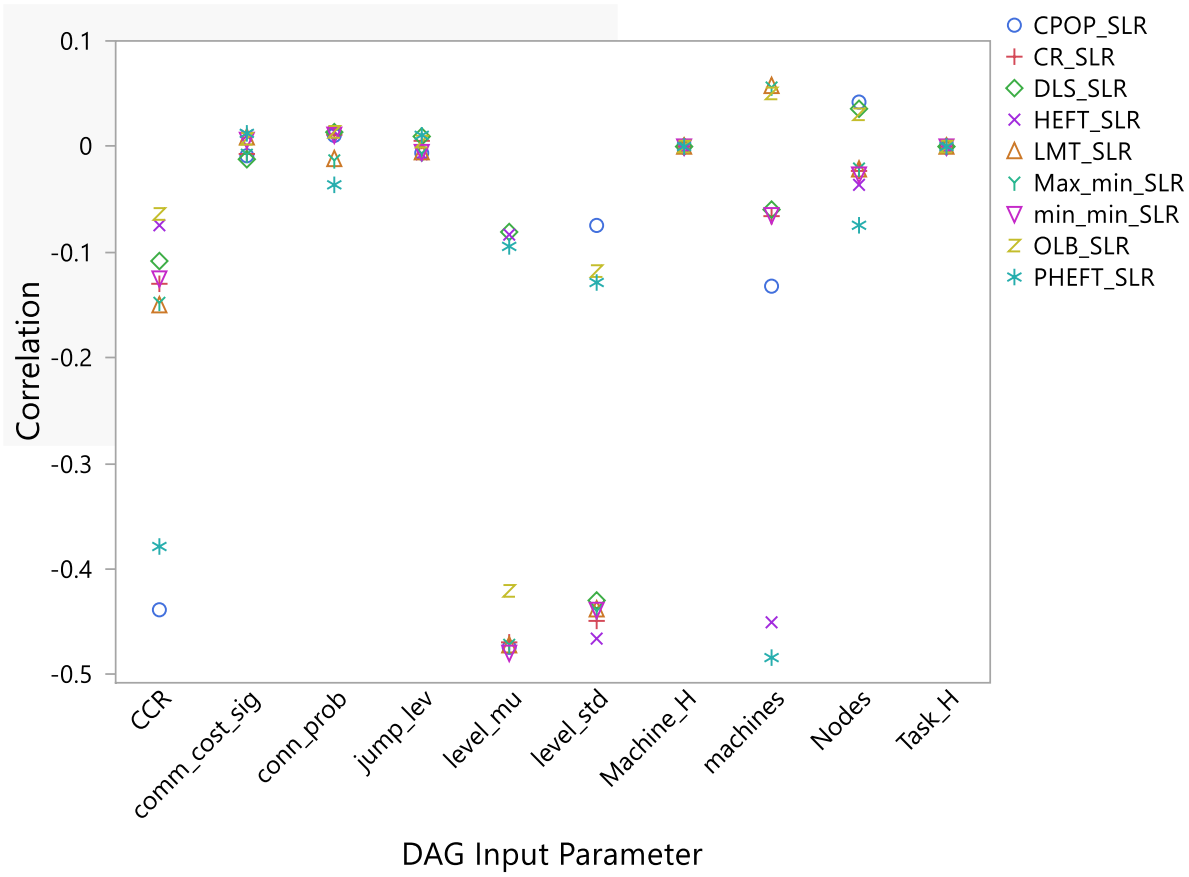


Figure 4.8, shown below, represents correlation results for an inconsistent system with low task high machine heterogeneity. Here the SLR of PHEFT algorithm shows significant negative correlation with number of machines, number of nodes, and computation to communication cost ratio. While the SLR of HEFT algorithm shows significant negative correlation with computation to communication cost ratio and the number of levels.

Figure 4.8: Correlation of DAG Input Parameters for Inconsistent System with Low Task High Machine Heterogeneity

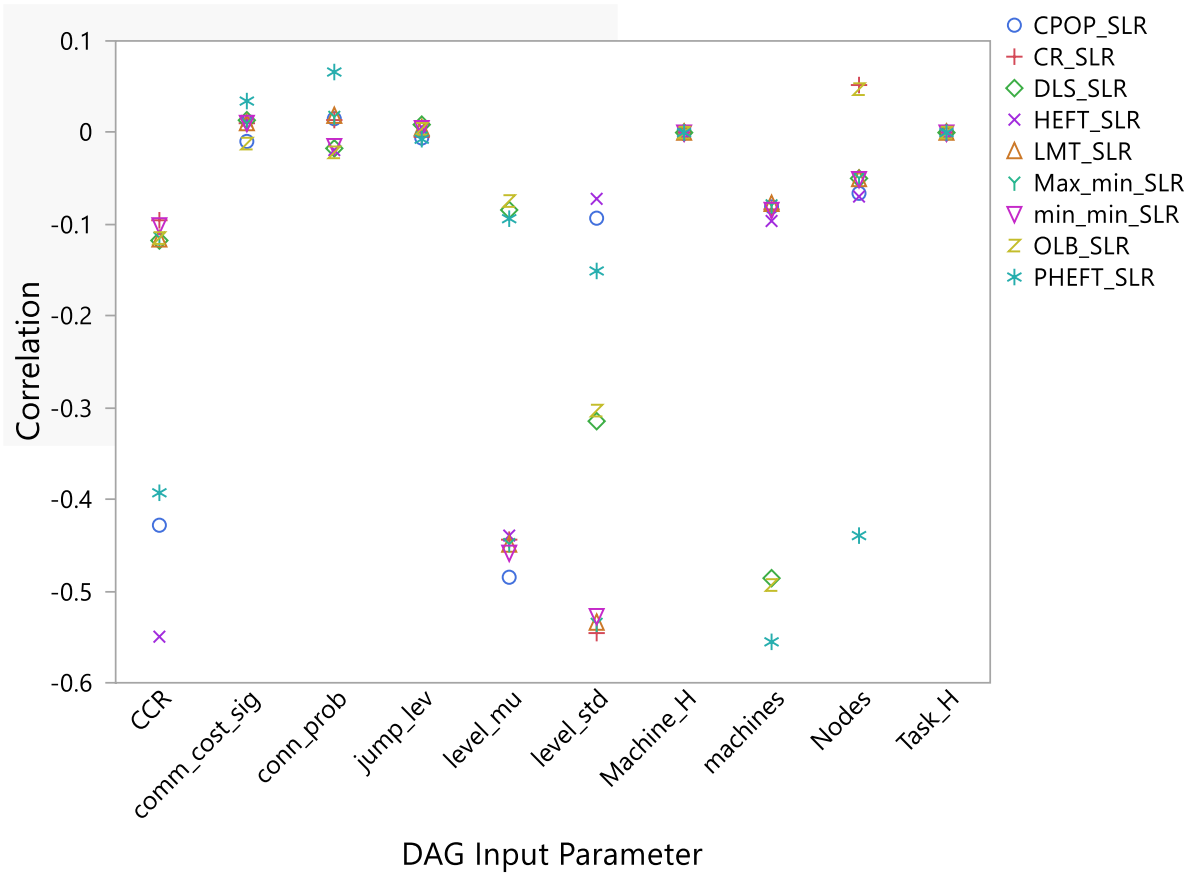


Figure 4.9, shown below, represents correlation results for an inconsistent system with high task low machine heterogeneity. The SLR of PHEFT algorithm has a negative correlation with the number of machines, which is only stronger with the SLR of the DLS and OLB algorithms. The SLR of HEFT algorithm has a negative correlation with the computation to communication cost ratio, however the SLR of PHEFT algorithm has no correlation with this DAG parameter.

Figure 4.9: Correlation of DAG Input Parameters for Inconsistent System with High Task Low Machine Heterogeneity

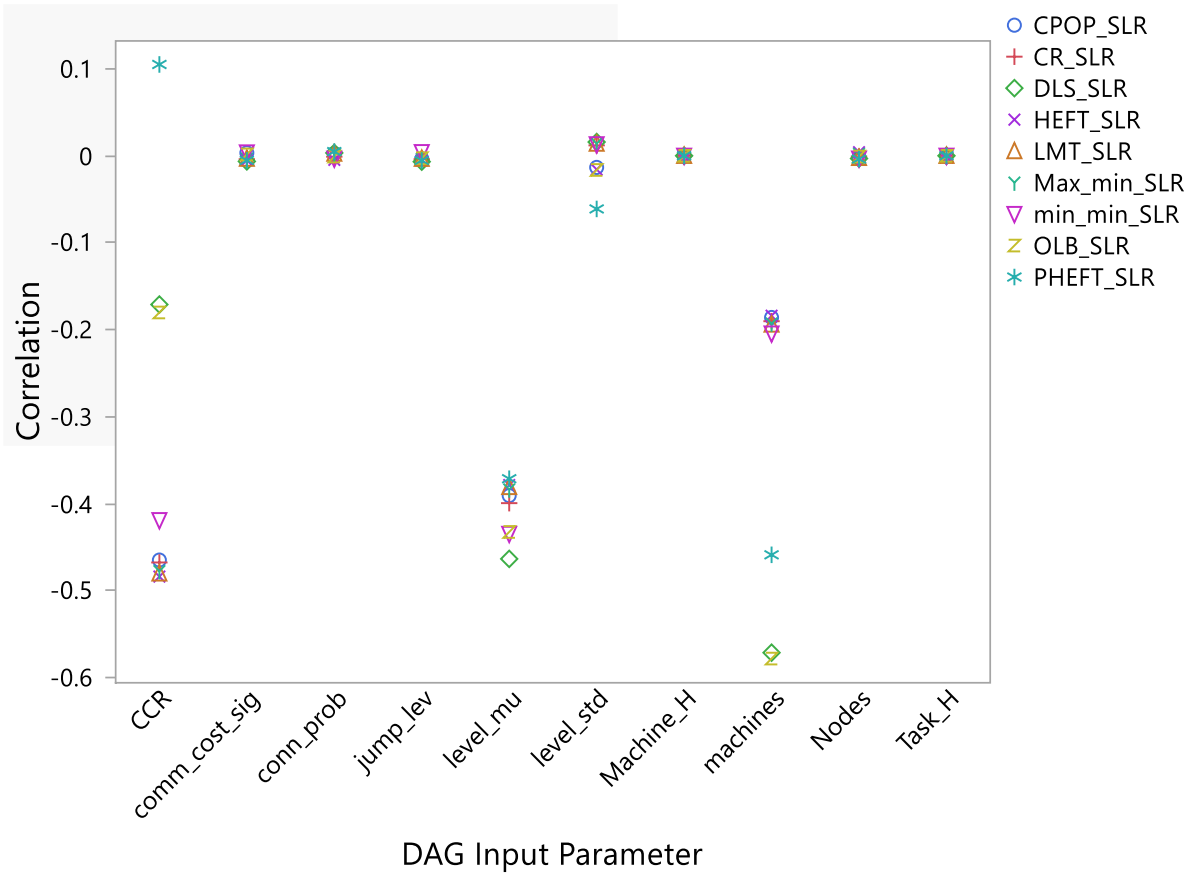
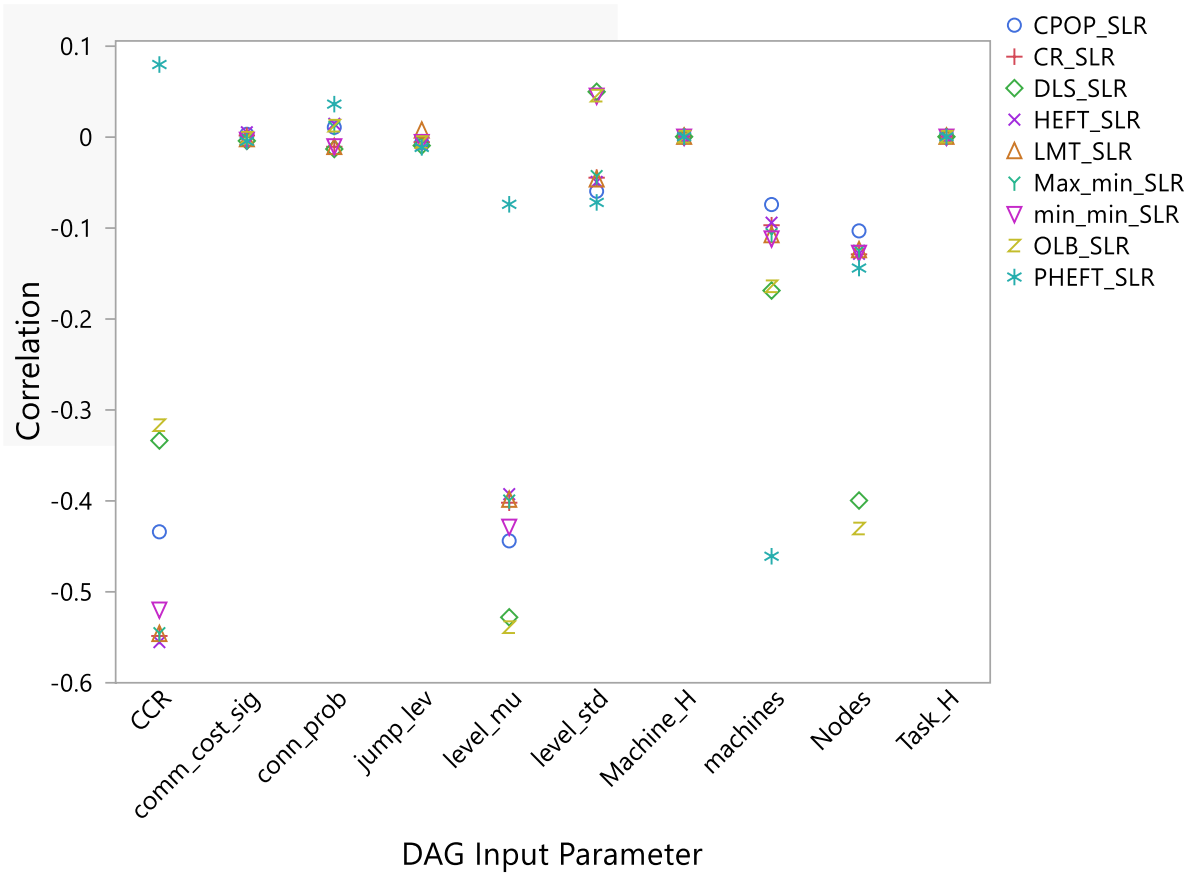


Figure 4.10 represents correlation results for an inconsistent system with high task high machine heterogeneity. Under this scenario, the SLR of PHEFT algorithm only shows correlation with the number of machines. The greater number of machines is correlated with lower SLR values. Several other algorithms show negative correlation between their SLR and the number of level, the number of nodes, and the computation to communication cost ratio.

Figure 4.10: Correlation of DAG Input Parameters for Inconsistent System with High Task High Machine Heterogeneity



In general, the correlation values found between the input parameters of the DAG generator and the SLR values, used to measure performance of the scheduling algorithms, are moderate, but they do show how these parameters could affect algorithm performance under various system environments. Using this information and a decision tree, the system manager would be able to select an algorithm which is more compatible to any special workload/environment pair.

CHAPTER 5

CONCLUSION

Two scheduling algorithms are introduced in this work. The first, Parallel Heterogeneous Earliest Finish Time (PHEFT), utilizes a single machine for more than one task. With high-performance computing infrastructures even when a task is assigned to a machine some of its resources are not fully utilized. The implementation complexity of this concept requires further examination, since it necessitates central control for sharing resources fairly between two tasks. PHEFFT shows substantial improvement of more than 25%, over the classic HEFT algorithm in consistent systems. The second algorithm, Optimize All Path (OAP), is based on the concept of improving the critical path execution time. Different assignments of tasks to machines can generate several potential critical paths. The initial located critical path might not be the final critical path that defines the system makespan. OAP's main objective is to observe all paths in the DAG and change decisions to optimize the critical path at each instant. This algorithm has high time complexity because of the high number of different paths in a graph and has the same performance as some popular greedy heuristics. These results suggest that targeting to optimize all possible paths

has adverse outcomes and may lead to some poor decisions, which are not improving the algorithm performance overall.

A major part of this research work focuses on developing and implementing a Random DAG Generator with a wider range of input parameters, with the aim to gain a better control and understanding of the characteristics of the workload and the computing environment. The proposed DAG generator offers advantages in characterizing the workload and ETC matrix, as well as generating an unbiased, almost uniformly distributed DAG set. It also offers the ability for tuning of the parameters to generate any workload/ETC matrix with desired characteristics. Using the extensive and unbiased workload generated with this DAG Generator, the correlation between each of the environment properties, or task types and the operation of heuristics was examined. Different scheduling algorithms show different behavior under different circumstances. The number of levels in a DAG, variation on the number of nodes in each level, number of machines, number of nodes, and computation to communication cost ratio are some of the properties, which show a higher correlation to the performance of a scheduling algorithm. This study also demonstrates that some other characteristics like connection probability, and the number of levels an edge passes to connect two tasks in a graph, have almost no effect on the response of a heuristic in different systems. This information can be used by the system manager to assess the efficiency of a scheduling algorithm for a specific workload/environment pair.

REFERENCES

[1] V. Sundaram, A. Chandra, and J. Weissman, "Exploring the throughput-fairness tradeoff of deadline scheduling in heterogeneous computing environments," *SIGMETRICS'08 Proc. 2008 ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, vol. 36, no. 1 SPECIAL ISSUE, pp. 463–464, 2008, doi: 10.1145/1375457.1375522.

[2] P. De and T. E. Morton, "Scheduling to Minimize Makespan on Unequal Parallel Processors," *Decis. Sci.*, vol. 11, no. 4, pp. 586–602, 1980, doi: https://doi.org/10.1111/j.1540-5915.1980.tb01163.x.

[3] S. Handley, "On the use of a directed acyclic graph to represent a population of computer programs," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, 1994, pp. 154–159 vol.1, doi: 10.1109/ICEC.1994.350024.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990.

[5] A. M. Al-Qawasmeh, A. A. Maciejewski, and H. J. Siegel, "Characterizing heterogeneous computing environments using singular value decomposition," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–9, doi: 10.1109/IPDPSW.2010.5470875.

[6] S. Ali *et al.*, "Characterizing Resource Allocation Heuristics for Heterogeneous Computing Systems," *Adv. Comput.*, vol. 63, pp. 91–128, 2005, doi: 10.1016/S0065-2458(04)63003-8.

[7] S. Ali, H. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing Task and Machine Heterogeneities for Heterogeneous Computing Systems," *Tamkang J. Sci. Eng.*, vol. 3, 2003.

[8] A. M. Al-Qawasmeh, A. A. Maciejewski, H. J. Siegel, J. Smith, and J. Potter, "Task and Machine Heterogeneities : Higher Moments Matter," no. January, 2009.

[9] J. Thaman and M. Singh, "Current Perspective in Task Scheduling Techniques in Cloud Computing: A Review," *Int. J. Found. Comput. Sci. Technol.*, vol. 6, pp. 65–85, 2016, doi: 10.5121/ijfcst.2016.6106.

[10] T. D. Braun *et al.*, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, 2001, doi: https://doi.org/10.1006/jpdc.2000.1714.

[11] V. Shestak, J. Smith, A. Maciejewski, and H. Siegel, "Stochastic robustness metric and its use for static resource allocations," *J. Parallel Distrib. Comput.*, vol. 68, pp. 1157–1173, 2008, doi: 10.1016/j.jpdc.2008.01.002.

[12] J.-K. Kim *et al.*, "Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment," *J. Parallel Distrib. Comput.*, vol. 67, no. 2, pp. 154–169, 2007, doi: https://doi.org/10.1016/j.jpdc.2006.06.005.

[13] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems," *J. Parallel Distrib. Comput.*, vol. 59, no. 2, pp. 107–131, 1999, doi:

https://doi.org/10.1006/jpdc.1999.1581.

[14] R. F. Freund *et al.*, “Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet,” in *Proceedings Seventh Heterogeneous Computing Workshop (HCW’98)*, 1998, pp. 184–199, doi: 10.1109/HCW.1998.666558.

[15] R. Armstrong, D. Hensgen, and T. Kidd, “The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions,” in *Proceedings Seventh Heterogeneous Computing Workshop (HCW’98)*, 1998, pp. 79–87, doi: 10.1109/HCW.1998.666547.

[16] O. H. Ibarra and C. E. Kim, “Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors,” *J. ACM*, vol. 24, no. 2, pp. 280–289, Apr. 1977, doi: 10.1145/322003.322011.

[17] P. Sugavanam *et al.*, “Robust static allocation of resources for independent tasks under makespan and dollar cost constraints,” *J. Parallel Distrib. Comput.*, vol. 67, no. 4, pp. 400–416, 2007, doi: https://doi.org/10.1016/j.jpdc.2005.12.006.

[18] H. Topcuoglu, S. Hariri, and M.-Y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, 2002, doi: 10.1109/71.993206.

[19] M. A. Iverson, F. Özgüner, and G. J. Follen, “Parallelizing Existing Applications in a Distributed Heterogeneous Environment,” in *4TH HETEROGENEOUS COMPUTING WORKSHOP (HCW ’95*, 1995, pp. 93–100.

[20] G. C. Sih and E. A. Lee, “A Compile-Time Scheduling Heuristic for Interconnection-

Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 175–187, 1993, doi: 10.1109/71.207593.

[21] M. Wu and D. D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 3, pp. 330–343, 1990, doi: 10.1109/71.80160.

[22] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 9, pp. 951–967, 1994, doi: 10.1109/71.308533.

[23] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 2, pp. 113–120, 1994, doi: 10.1109/71.265940.

[24] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors," in *Proceedings Second International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'96)*, 1996, pp. 207–213, doi: 10.1109/ISPAN.1996.508983.

[25] S. Kim and J. Browne, "General approach to mapping of parallel computations upon multiprocessor architectures," in *Proceedings of the International Conference on Parallel Processing*, 1988, pp. 1–8.

[26] G.-L. Park, B. Shirazi, and J. Marquis, "DFRN: a new approach for duplication based scheduling for distributed memory multiprocessor systems," in *Proceedings 11th International Parallel Processing Symposium*, 1997, pp. 157–166, doi:

10.1109/IPPS.1997.580875.

[27] I. Ahmad and Y.-K. K. Yu-Kwong Kwok, “A New Approach to Scheduling Parallel Programs Using Task Duplication,” in *1994 Internatonal Conference on Parallel Processing Vol. 2*, 1994, vol. 2, pp. 47–51, doi: 10.1109/ICPP.1994.37.

[28] B. Kruatrachue and T. Lewis, “Grain Size Determination for Parallel Processing,” *IEEE Softw.*, vol. 5, no. 1, pp. 23–32, 1988, doi: 10.1109/52.1991.

[29] Y.-C. Chung and S. Ranka, “Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors,” in *Supercomputing ’92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992, pp. 512–521, doi: 10.1109/SUPERC.1992.236653.

[30] R. Kaur and R. Kaur, “Multiprocessor Scheduling Using Task Duplication Based Scheduling Algorithms : A Review Paper,” 2013.

[31] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[32] D. Whitley, “The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best,” vol. 89, 2000.

[33] K. Kaur, A. Chharbra, and S. Gurvinder, “Heuristics Based Genetic Algorithm for Scheduling Static Tasks in Homogeneous Parallel System,” no. 4, pp. 183–198.

[34] F. Pop, C. Dobre, and V. Cristea, “Genetic algorithm for DAG scheduling in Grid environments,” in *2009 IEEE 5th International Conference on Intelligent Computer*

Communication and Processing, 2009, pp. 299–305, doi: 10.1109/ICCP.2009.5284747.

[35] H. Khajemohammadi, A. Fanian, and T. A. Gulliver, "Fast workflow scheduling for grid computing based on a multi-objective Genetic Algorithm," in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 2013, pp. 96–101, doi: 10.1109/PACRIM.2013.6625456.

[36] J. Gu, J. Hu, T. Zhao, and G. Sun, "A New Resource Scheduling Strategy Based on Genetic Algorithm in Cloud Computing Environment," *JCP*, vol. 7, pp. 42–52, 2012, doi: 10.4304/jcp.7.1.42-52.

[37] C. Zhao, S. Zhang, Q. Liu, J. Xie, and J. Hu, "Independent Tasks Scheduling Based on Genetic Algorithm in Cloud Computing," in *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing*, 2009, pp. 1–4, doi: 10.1109/WICOM.2009.5301850.

[38] T. Yamada and C. R. Reeves, "Permutation flowshop scheduling by genetic local search," in *Second International Conference On Genetic Algorithms In Engineering Systems: Innovations And Applications*, 1997, pp. 232–238, doi: 10.1049/cp:19971186.

[39] D. Nasonov, N. Butakov, M. Balakhontseva, K. V Knyazkov, and A. Boukhanovsky, "Hybrid Evolutionary Workflow Scheduling Algorithm for Dynamic Heterogeneous Distributed Computational Environment," in *SOCO-CISIS-ICEUTE*, 2014.

[40] J. Kolodziej, S. U. Khan, and F. Xhafa, "Genetic Algorithms for Energy-Aware

Scheduling in Computational Grids," in *2011 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2011, pp. 17–24, doi: 10.1109/3PGCIC.2011.13.

[41] G. Shen and Y.-Q. Zhang, "A Shadow Price Guided Genetic Algorithm for Energy Aware Task Scheduling on Cloud Computers," in *Proceedings of the Second International Conference on Advances in Swarm Intelligence - Volume Part I*, 2011, pp. 522–529.

[42] J. Liu, X.-G. Luo, X.-M. Zhang, F. Zhang, and B.-N. Li, "Job Scheduling Model for Cloud Computing Based on Multi-Objective Genetic Algorithm," *Int. J. Comput. Sci. Issues*, vol. 10, no. 1, pp. 134–139, Jan. 2013, [Online]. Available: https://www.proquest.com/scholarly-journals/job-scheduling-model-cloud-computing-based-on/docview/1442567478/se-2?accountid=14521.

[43] T. Wang, Z. Liu, Y. Chen, Y. Xu, and X. Dai, "Load Balancing Task Scheduling Based on Genetic Algorithm in Cloud Computing," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 2014, pp. 146–152, doi: 10.1109/DASC.2014.35.

[44] A. A. P. Kazem, A. M. Rahmani, and H. H. Aghdam, "A Modified Simulated Annealing Algorithm for Static Task Scheduling in Grid Computing," in *2008 International Conference on Computer Science and Information Technology*, 2008, pp. 623–627, doi: 10.1109/ICCSIT.2008.163.

[45] S. Fidanova, "Simulated Annealing for Grid Scheduling Problem," in *IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06)*, 2006, pp.

41–45, doi: 10.1109/JVA.2006.44.

[46] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*. Berlin, Heidelberg: Springer-Verlag, 2010.

[47] S. Zheng, W. Shu, and L. Gao, "Task Scheduling using Parallel Genetic Simulated Annealing Algorithm," in *2006 IEEE International Conference on Service Operations and Logistics, and Informatics*, 2006, pp. 46–50, doi: 10.1109/SOLI.2006.328980.

[48] Y. W. Wong, R. S. M. Goh, S.-H. Kuo, and M. Y. H. Low, "A Tabu Search for the Heterogeneous DAG Scheduling Problem," in *2009 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 663–670, doi: 10.1109/ICPADS.2009.127.

[49] I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro, "Improving search by incorporating evolution principles in parallel Tabu Search," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, 1994, pp. 823–828 vol.2, doi: 10.1109/ICEC.1994.349949.

[50] F. Glover and M. Laguna, *Tabu Search*. Boston, MA: Springer, 1997.

[51] P. Merz and B. Freisleben, "A Genetic Local Search Approach to the Quadratic Assignment Problem," in *in Proceedings of the 7th International Conference on Genetic Algorithms*, 1997, pp. 465–472.

[52] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 53–66, 1997, doi: 10.1109/4235.585892.

[53] L.-C. Canon, M. Sayah, and P.-C. Héam, "A Comparison of Random Task Graph

Generation Methods for Scheduling Problems," *ArXiv*, vol. abs/1902.0, 2019.

[54] K. M. Tarplee, R. Friese, A. A. Maciejewski, H. J. Siegel, and E. K. P. Chong, "Energy and Makespan Tradeoffs in Heterogeneous Computing Systems using Efficient Linear Programming Techniques," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1633–1646, 2016, doi: 10.1109/TPDS.2015.2456020.

[55] T. D. Braun *et al.*, "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, 1999, pp. 15–29, doi: 10.1109/HCW.1999.765093.

[56] A. M. Al-Qawasmeh, A. A. Maciejewski, H. Wang, J. Smith, H. J. Siegel, and J. Potter, "Statistical measures for quantifying task and machine heterogeneities," *J. Supercomput.*, vol. 57, no. 1, pp. 34–50, 2011, doi: 10.1007/s11227-011-0572-x.

[57] G. H. Golub and C. F. Van Loan, "Matrix Computations, SecondEdition." The Johns Hopkins University Press, Baltimore, MD, 1989.

[58] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*, 2010, p. 10.

[59] A. V. Martinez, "Synthetic loads analysis of directed acyclic graphs for scheduling tasks," *Algorithms*, vol. 9, no. 3, 2018.

[60] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *J. Parallel Distrib. Comput.*, vol. 59, no. 3, pp. 381–422, 1999.

[61] R. P. Dick, D. L. Rhodes, and W. Wolf, “TGFF: task graphs for free,” in *Proceedings of the Sixth International Workshop on Hardware/Software Codesign.(CODES/CASHE’98)*, 1998, pp. 97–101.

[62] P.-F. Dutot, T. N’takpé, F. Suter, and H. Casanova, “Scheduling parallel task graphs on (almost) homogeneous multicluster platforms,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 7, pp. 940–952, 2009.

[63] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, “Characterizing and profiling scientific workflows,” *Futur. Gener. Comput. Syst.*, vol. 29, no. 3, pp. 682–692, 2013.

[64] P. Campos, N. Dahir, C. Bonney, M. Trefzer, A. Tyrrell, and G. Tempesti, “XL-STaGe: A cross-layer scalable tool for graph generation, evaluation and implementation,” in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016, pp. 354–359.

[65] I. Gupta, A. Choudhary, and P. K. Jana, “Generation and proliferation of random directed acyclic graphs for workflow scheduling problem,” in *Proceedings of the 7th International Conference on Computer and Communication Technology*, 2017, pp. 123–127.

[66] T. Tobita and H. Kasahara, “A standard task graph set for fair evaluation of multiprocessor scheduling algorithms,” *J. Sched.*, vol. 5, no. 5, pp. 379–394, 2002.

[67] P. Erdös and A. Rényi, “On random graphs,” *Math. debrecen*, vol. 6, pp. 290–297, 1959.