CHAPTER 2

LITERATURE REVIEW

This chapter briefly reviews various categories of existing task scheduling algorithms such as list-scheduling, clustering, duplication-based and genetic-based scheduling algorithms developed for homogeneous and heterogeneous computing environments. The limitations of the existing task scheduling algorithms and the need for developing new task scheduling algorithms in each category is pointed out. This chapter also presents the necessity for developing a new task scheduling algorithm to minimize the energy consumption or schedule length or both for the recently popular computing paradigm called mobile computing system.

2.1 Preamble

A DHCS is a collection of autonomous dissimilar computing machines that are linked by a network and are coordinated by software to function as a single powerful computing platform. DHCS have the potential to provide low cost and high performance computing in meeting the computational requirements of many applications in science, engineering and commerce [3, 4]. While DHCS offers the promise of vastly increased performance, it introduces additional complexities which are not encountered with sequential processing such as partitioning the program into tasks and scheduling the tasks onto the available processors. Efficient task scheduling algorithm plays a critical role in order to truly exploit the potentials of DHCS. A well known strategy behind the efficient execution of a larger program is to partition it into multiple independent tasks with some precedence relationship and to schedule such tasks over a set of available processors. A task-partitioning algorithm takes care of efficiently dividing an application into tasks of appropriate grain size and an abstract model of such a partitioned application can be represented by a DAG. Each task of a DAG corresponds to a sequence of operations and a directed arc represents the precedence constraints between the tasks. Each task can be executed on a processor and the directed arc shows transfer of relevant data from
one processor to another. The task scheduling problem deals with assigning (matching) each task to a processor and ordering (scheduling) the execution of the tasks on each processor in order to minimize schedule length, or makespan. Since a task scheduling algorithm plays a key role in distributed computing system for achieving high performance, the problem in general, has been widely explored by the research community and consequently various categories of task scheduling algorithms have been developed. The classification of the task scheduling algorithms is presented in the next section.

2.2 Classification of Task Scheduling Algorithms

The task scheduling problem is broadly classified into two major categories namely, deterministic scheduling problem also known as static scheduling or compile-time scheduling problem and nondeterministic scheduling problem also known as dynamic scheduling or run-time scheduling problem [14] based on the characteristics of the program and tasks to be scheduled, system architecture and the availability of the information. Static scheduling is one in which all information about the tasks and their relations to each other such as, execution time and precedence relations are known to the scheduling algorithm in advance. Dynamic scheduling is one in which some information about tasks and their relations may be undeterminable until run-time. The major advantage of static scheduling is that the overhead of the scheduling process is incurred at compile-time, resulting in a more efficient execution time environment when compared to dynamic scheduling. The problem of task scheduling has been proven to be NP-complete [8, 9] in the general case as well as some restricted cases [10]. Approaches based on graph-theoretic techniques [11] and integer programming techniques [12, 13] to determine the optimal solution also exist. However, these methods are computationally very expensive. Hence, attention has been focused in the development of heuristic-based scheduling algorithms [5, 12, 17].

Yu-Kwong Kwok et al. [18] presented a general survey of various static task scheduling algorithms developed for the homogeneous processors. The algorithms are classified into different categories based on the assumptions used in the
tasks, algorithms such as the task graph structure (arbitrary DAG or restricted structure), computation costs (arbitrary costs or unit costs), communication (communication cost considered or not), duplication (task duplication allowed or not), number of processors (limited or unlimited) and connection type among the processors (fully connected or arbitrarily connected). The scheduling algorithms developed for the homogeneous processors form the foundation for the development of the scheduling algorithms for the heterogeneous processors \cite{32} i.e., scheduling algorithms for heterogeneous processors have been developed by using the concepts which were already experimented for the homogeneous processors. In this thesis, a survey of the algorithms developed for the homogenous and the heterogeneous processors have been made and these algorithms are universally classified into two major groups namely, heuristic-based and guided random search-based algorithms. Heuristics-based scheduling algorithms are further classified into three core groups such as list-scheduling, clustering and task duplication-based algorithms. The classification of static task scheduling algorithms that are common to both the homogeneous and the heterogeneous processors is given in Figure 2.1.

![Figure 2.1 Classification of task scheduling algorithms](image-url)
2.3 Heuristic-based Algorithms

To provide solutions to the task scheduling problem, heuristic-based algorithms were developed which give near-optimal solution but acceptable performance, with the benefit of polynomial rather than exponential time complexity. These algorithms produce solutions by making the most realistic assumptions about a priori knowledge concerning the tasks of application and the target computing system. However, when making scheduling decisions by considering the communication delays introducing a big challenge i.e., the trade-off between taking advantage of maximum parallelism and minimum communication delay. High parallelism means dispatching more tasks simultaneously to different processors, thus increasing the communication cost, especially when the communication delay is very high. However, scheduling the tasks only on a few resources means low processor utilization. Hence, to deal with this problem in distributed computing systems, three categories of heuristic-based task scheduling algorithms namely, list-scheduling, clustering and task duplication-based scheduling algorithms were developed for homogeneous and heterogeneous processors. The algorithms developed in each of the categories are introduced in detail in the following sections.

2.3.1 List-Scheduling Algorithms

One of the earliest proposed solutions to the task scheduling problem is the list-scheduling algorithms. Interestingly, a majority of task scheduling algorithms developed for the homogeneous and the heterogeneous processors belong to the class of list-scheduling algorithms [5, 18, 20, 21]. This is probably due to their relative simplicity and low complexity in comparison with other approaches. The basic idea in the list-scheduling is to assign priorities to the tasks of the DAG and place the tasks in a list arranged in decreasing order of priorities. At each scheduling step the ready task (a ready task is one whose predecessor tasks have finished the execution and the communications data are available at the processor to which the task is to be assigned) with the highest priority is selected and scheduled to a suitable processor. If two or more tasks have equal priority, then the tie is resolved by selecting the task randomly or selecting the task that takes more execution time.
The process is repeated until all the tasks are scheduled to the suitable processors. The pseudo code of the general list-scheduling algorithm is given below:

/* List-Scheduling Algorithm */
1. begin
2. read the DAG and its associated parameters, number of processors \( p \);
3. calculate the priority of each task according to some predefined formula;
4. priorityList = \([v_1, v_2, ..., v_n]\) is sorted by descending order of task priorities;
5. while ( priorityList is not empty ) do
6. begin
   a. remove the first task from the priorityList;
   b. select an appropriate processor from the available processors, which gives the minimum completion time for the task;
   c. assign the task to the selected processor;
7. end;
8. end.

There are two important issues in the list-scheduling algorithm: (1) How to prioritize the tasks of the DAG? and (2) How to select the best processor for executing the selected task? The first question is related to the way the algorithm views the task’s urgency of being scheduled and the second question is related to the selection of one particular processor among the available \( p \) processors in the system. A large number of list-scheduling algorithms have been developed by the researchers in the past. However, majority of the list-scheduling algorithms were developed for the homogeneous processors [50 - 60] and few have been developed for the heterogeneous processors [21, 61 - 63].

The existing list-scheduling algorithms use numerous variations in the methods of assigning priorities, maintaining the ready list and the criteria for selecting a processor to accommodate a task. In general the existing list-scheduling algorithms use the attributes such as \( t \)-level (top level), \( b \)-level (bottom level) or combination of these two attributes for assigning priority to the tasks [18, 64]. The \( t \)-level of a task \( v \), is the length of the longest path from an entry task to \( v \), in the
DAG (excluding $v_i$) where the length of a path is the sum of all the task and edge weights along the path. The $t$-level of task $v_i$ highly correlates with $v_i$'s earliest start time, which is determined after $v_i$ is scheduled to a processor. The $t$-level of a task is a dynamic attribute because the weight of an edge may be zeroed when the two incident tasks are scheduled to the same processor. Thus, the path reaching a task, whose length determines the $t$-level of the task, may cease to be the longest one. The $b$-level of a task $v_i$ is the length of the longest path from task $v_i$ to an exit task and is bounded by the length of the critical path. A Critical Path (CP) of a DAG is a path from an entry task to an exit task, whose length is the maximum. Variations in the computation of $b$-level of a task are possible.

Most task scheduling algorithms examine a task for scheduling only after all the parents of the task have been scheduled. In this case, the $b$-level of a task is a constant until it is scheduled to a processor. However, some algorithms allow the scheduling of a child before its parents. In that case, the $b$-level of a task becomes a dynamic attribute. In these algorithms, the earliest start time of any task $v_i$ is not fixed until all the tasks are scheduled, allowing the insertion of a task to a time slot created by pushing some earlier tasks downward. Some algorithms do not take into account the communication time while calculating $b$-level and this is referred to as static $b$-level. Different task scheduling algorithms have used the $t$-level and $b$-level attributes in a variety of ways. Some algorithms assign a higher priority to a task with a smaller $t$-level while some algorithms assign a higher priority to a task with a larger $b$-level. Some algorithms assign a higher priority to a task with a larger ($b$-level - $t$-level). In general, scheduling in descending order of $b$-level tends to schedule critical path tasks first while scheduling in ascending order of $t$-level tends to schedule tasks in a topological order. The composite attribute ($b$-level - $t$-level) is a compromise between the previous two.

The second question deals with the selection of best processor for executing a task. Earliest start time and earliest finish time of a task on a processor $p$ are generally used to select a processor. When determining the start time of a task on a processor, some algorithms only consider scheduling a task after the last task on a
processor. Some algorithms also consider other idle time slots on a processor and may insert a task between already scheduled tasks. In the homogeneous systems, a commonly used mechanism to select a best processor is EST. For example, the *Earliest Time First (ETF)* [65] algorithm computes the earliest start times, at each scheduling step, for all the ready tasks and then selects a task with the smallest earliest start time. When two or more tasks have the same value of their earliest start times, the ETF algorithm breaks the tie by scheduling the task with the highest *static level*. While selecting the best processor in heterogeneous system a commonly used mechanism is *EFT*. The HCPT algorithm [22] tries to assign each task \( v_i \in L \) (list of ready tasks) to a processor \( p_m \in p \) (set of available processors) that allows the task to finish its execution as early as possible. The performance of the HCPT is better than the CPOP by 71 percent, FLB by 95 percent and DLS by 95 percent in terms of various evaluation parameters.

### 2.3.1.1 List-Scheduling Algorithms for Homogeneous Processors

The existing list-scheduling algorithms, by and large, focuses on homogeneous processors which were developed by researchers over a period of time. The list-scheduling algorithms developed for homogeneous processors forms the foundation for the development of list-scheduling algorithms for heterogeneous processors. Therefore, some well known and widely referred list-scheduling algorithms developed for the homogeneous processors are briefly described below. The time complexities of the algorithms are given in terms of \( p \), \( e \) and \( v \) where \( p \) is the number of processors, \( e \) is the number of edges and \( v \) is the number of tasks.

#### Highest Level First with Estimated Times (HLFET) Algorithm

Adam et al., proposed the HLFET algorithm [66] which is one of the earliest and simplest scheduling algorithms developed for the fully connected homogeneous processors. In the HLFET the priority is assigned to the task based on the sum of computation costs of all the tasks along the longest path from the task to an exit task. The algorithm schedules a task to a suitable processor that allows the earliest start time. The main drawback of the HLFET algorithm is that it ignores the communication costs on the edges. The time complexity of HLFET is \( O(v^3) \).
**Insertion Scheduling Heuristic (ISH) Algorithm**

Kruatrachue et al., proposed the ISH algorithm [67] which effectively utilizes the idle time created by the partial schedules. The algorithm initially assigns priority to the tasks based on static b-level. Then it picks an unscheduled task with highest priority and schedules it to a processor that allows the earliest start time. While scheduling a task to the suitable processor, the ISH algorithm considers the possibility of inserting a task between already scheduled tasks in the same processor if sufficient idle time slot exists in between the already scheduled tasks. It is proved that the ISH algorithm provides better results than the non-insertion based scheduling algorithms. The ISH algorithm is developed for the fully connected homogeneous processors and it has the time complexity of $O(v^2)$.

**Earliest Time First (ETF) Algorithm**

Hwang et al., proposed the ETF algorithm [65]. The ETF algorithm uses static task priorities and assumes only a bounded number of processors. However, a task with a higher priority may not necessarily get scheduled before the tasks with lower priorities. This is because at each scheduling step, the ETF algorithm first computes the earliest start times for all the ready tasks and then selects the one with the minimum earliest start time. A task is ready if all its parent tasks have been scheduled. The earliest start time of a task is computed by examining the start time of the task on all processors exhaustively. When two tasks have the same value of the earliest start time, the ETF algorithm breaks the tie by scheduling the one with a higher static priority. The major deficiency of the ETF algorithm is that it may not be able to reduce the partial schedule length at every scheduling step. The time complexity of the ETF algorithm is $O(pv^4)$.

**Modified Critical Path (MCP) Algorithm**

Wu et al., devised the MCP algorithm [68] based on an attribute called the *Latest Possible Start Time (LPST)* of a task. A task’s latest possible start time is determined through the *As Late As Possible (ALAP)* binding by traversing the task graph upward from the exit tasks to the entry tasks and by pulling the tasks
downwards as much as possible constrained by the length of CP. i.e., the ALAP time of a task is computed by first computing the length of CP and then subtracting the b-level of the task from it. The MCP algorithm first computes all the LPSTs for all tasks. Then, each task is associated with a list of latest possible start time which consists of the LPST of the task itself, followed by a decreasing order of the LPSTs of its children tasks. The MCP algorithm then constructs a list of tasks in an increasing lexicographical order of the latest possible start time list. At each scheduling step, the first task is removed from the list and scheduled to a processor that allows for the EST. The MCP algorithm assigns higher priorities to tasks which have minimum latest possible start time. However, the MCP algorithm does not necessarily schedule tasks on the CP first. It is in this respect that the MCP algorithm works in a similar way as that of the ETF algorithm. The MCP algorithm does not assign task priorities accurately even though it takes communication among tasks into account for computing the priorities.

Fast Critical Path (FCP) Algorithm

Radulescu et al., developed a low complexity list-scheduling heuristic called FCP [51, 69]. The FCP algorithm is derived from the MCP algorithm and it uses the simplified version of the MCP algorithm in which ties between task priorities are broken randomly instead of considering all the descendants of the tasks. The motivation of the FCP algorithm is based on the observation regarding complexity of the list-scheduling heuristics. Basically, a list-scheduling heuristic has $O(e+v)$ time ranking phase, $O(v \log v)$ time ordering phase, and finally $O((e+v)xp)$ time processor selection phase. Generally the third term is larger than the second term. The FCP algorithm does not sort all the tasks at the beginning but maintains only a limited number of tasks sorted at any given time. Instead of considering all processors as possible targets for a given task, the choice is restricted to either the processor from which the last messages to the given task arrives or the processor which becomes idle at the earliest. This implies that restricting the selection to these two processors indeed does not affect the performance of the algorithm, while drastically reducing its complexity to $O(v \log(p)+e)$, which shows significant improvement over the typical time complexity $O(v \log(v)+(e+v)p)$ of the current
list-scheduling approach. The FCP algorithm gives better results with less time complexities than the DLS, MCP and the Critical Path Method (CPM) algorithms.

**Critical Nodes Parent Trees (CNPT) Algorithm**

Hagras et al., proposed the CNPT algorithm [52] which aims to achieve high performance and low complexity. The algorithm divides the task graph into a set of unlisted-parent trees. The root of each parent-tree is a Critical Node (CN). A CN node (task) is defined as the node that has zero difference between its EST and Latest Start Time (LST). The algorithm consists of two phases, namely, **listing phase** and **processor assigning phase**. In the listing phase, the algorithm starts with an empty queue $L$ and an auxiliary stack $S$ that contains the critical nodes pushed in a decreasing order of their $LST$s, i.e., the entry node is on the top of $S$ ($top(S)$). Consequently, $top(S)$ is examined. If $top(S)$ has unlisted-parents (i.e. has parents not in $L$), then the parent with the smallest $LST$ is pushed on the stack. Otherwise, $top(S)$ is popped and enqueued into $L$. In the processor assigning phase, the algorithm tries to assign each ready task $v_i \in L$ on a processor $p_m \in P$, that allows $v_i$ to be executed as early as possible. The CNPT algorithm provides better performance than the MCP, DLS and the ETF algorithms.

The list-scheduling algorithm developed for homogeneous processors is not limited only to the above algorithms. Development of list-scheduling algorithms has been extensively carried out for the homogeneous processors in literature and consequently many algorithms have been developed over a period of time. These algorithms include Earliest Ready Time (ERT) [53], Heavy Node First (HNF) [17], Bottom-Up (BU) [54], Generalized List-Scheduling (GLS) [56], Fast Assignment using Search Technique (FAST) [57], Decisive Path Scheduling (DPS) [58], Parallel Bubble Scheduling Algorithm (PBSA) [59], Bubble Scheduling and Allocation (BSA) [60] and Fast Dynamic Level Scheduling (FDLS) [51]. The taxonomy of the existing list-scheduling algorithms developed for the homogeneous processors with their time complexity, network topology along with the observations and the limitations of the algorithms are tabulated and given in Table 2.1.
Table 2.1 Taxonomy of list-scheduling algorithms developed for homogeneous processors

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Network Topology</th>
<th>Complexity</th>
<th>Observations/Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLFET</td>
<td>fully connected</td>
<td>$O(v^2)$</td>
<td>Ignores communication cost on the edges</td>
</tr>
<tr>
<td>ISH</td>
<td>fully connected</td>
<td>$O(v^2)$</td>
<td>Introduces new idea of inserting a task in the idle time slot, but ignores the communication cost on the edges</td>
</tr>
<tr>
<td>ERT</td>
<td>fully connected</td>
<td>$O(pv^2)$</td>
<td>Communication cost is considered and the task that can be processed earliest is selected</td>
</tr>
<tr>
<td>ETF</td>
<td>fully connected</td>
<td>$O(pv)$</td>
<td>Communication delays are not considered. Higher time complexity</td>
</tr>
<tr>
<td>CPM</td>
<td>fully connected</td>
<td>$O(v^2)$</td>
<td>High priority is given to tasks that lie in the CP</td>
</tr>
<tr>
<td>HNF</td>
<td>fully connected</td>
<td>$O(v \log v)$</td>
<td>High priority is given to the task with large computation. Proved better than CPM</td>
</tr>
<tr>
<td>MCP</td>
<td>fully connected</td>
<td>$O(v^2(\log(v)+p))$</td>
<td>Does not effectively determine the number of processors required for executing the program</td>
</tr>
<tr>
<td>BU</td>
<td>arbitrarily connected</td>
<td>$O(v^2 \log v)$</td>
<td>One of the earliest algorithms developed for the arbitrarily connected processors</td>
</tr>
<tr>
<td>GLS</td>
<td>arbitrarily connected</td>
<td>$O(pv^2)$</td>
<td>Provides better results than ETF</td>
</tr>
<tr>
<td>FAST</td>
<td>fully connected</td>
<td>$O(e)$</td>
<td>Outperforms ETF, DSC and DLS with significantly less time complexity</td>
</tr>
<tr>
<td>DPS</td>
<td>fully connected</td>
<td>$O(v \log v+(e+v)p)$</td>
<td>Better than HNF, HLFET, DSC and CPND</td>
</tr>
<tr>
<td>PBSA</td>
<td>fully connected</td>
<td>$O(p^2e^2v)$</td>
<td>Inaccuracy in estimating start time of a task. Takes high time complexity</td>
</tr>
<tr>
<td>BSA</td>
<td>fully connected</td>
<td>$O(p^2e^2v)$</td>
<td>Provides better results, but higher time complexity</td>
</tr>
<tr>
<td>FCP</td>
<td>fully connected</td>
<td>$O(v \log(p)+e)$</td>
<td>Reduces time complexity by sorting few tasks and provides better results than the DLS, MCP and the CPM</td>
</tr>
<tr>
<td>FDLS</td>
<td>fully connected</td>
<td>$O(v \log(p)+e)$</td>
<td>Reduces the time complexity by sorting a few tasks and proves better than the MCP and DLS</td>
</tr>
<tr>
<td>CNPT</td>
<td>fully connected</td>
<td>$O(v^2)$</td>
<td>Provides better results than DLS, ETF and MCP</td>
</tr>
</tbody>
</table>
2.3.1.2 List-Scheduling Algorithms for Heterogeneous Processors

The algorithms developed for homogeneous systems cannot be applied directly to DHCS because of various heterogeneity factors such as processors, network bandwidth, network topology, etc. Hence, research work has been carried out for the development of scheduling algorithms for heterogeneous processors taking into account the heterogeneity of the target system. The list-scheduling algorithms developed for the heterogeneous processors over a period of time are briefly discussed in the following sections.

Mapping Heuristic (MH) Algorithm

El-Rewini et al., proposed the MH algorithm [62] is one of the earliest list-scheduling techniques that considers several real world parameters like, interconnection topology, link transfer rates and contention delay which are neglected in the other list-scheduling algorithms. The MH algorithm is an event driven scheduling heuristic. There are four possible types of events in the event list, they are task ready, task done, message sent and message received. The event list is always sorted according to the event time. A task ready event is generated when all the immediate predecessors of a task finish their executions. A task done event is generated when the task is scheduled onto a processor. These two types of events exist whether contention delay is taken into consideration or not. The task done event initiates a message sent event to pass data to the successor of the task. The processing of a message sent event puts a message received event into the event list.

In the MH algorithm, a routing table is maintained by each processing element. Each entry of the table is indexed by the destination processing element, and it contains three fields: the number of hops, the preferred outgoing edge and the communication delay due to contention. The routing tables will direct messages from one machine to another along a path with minimum communication time. Initially, the shortest paths between processing elements are stored in the routing tables. When a message is sent, the route from source to destination machine becomes busy, carrying the message for a certain amount of time. When a message
is received, its route becomes free and this route can be used by other processors for transmission of messages again. So every time a message sent or a message received event is processed, the routing tables affected will be updated to provide the directions for the fastest communication routes at any time.

**Dynamic Level Scheduling (DLS) Algorithm**

Sih et al., proposed the DLS algorithm [20]. The DLS algorithm is also one of the earliest list-scheduling algorithms designed for heterogeneous processors. However, in contrast to traditional list-scheduling, DLS does not maintain a scheduling list during the scheduling process. It determines task priorities dynamically by assigning an attribute called the *Dynamic Level (DL)* to all unscheduled tasks at each scheduling step. The DL of task \( v_i \) on processor \( p_j \), denoted as \( DL(v_i, p_j) \) and is computed by using *static level of task \( v_i \)* and *Start Time ST\((v_i, p_j)\)*. The DL is defined as the difference between static level of task \( v_i \) and \( ST(v_i, p_j) \). At each scheduling step, the DLS algorithm computes the DL for each ready task on every processor. Then the task-processor pair that constitutes the largest DL among all other pairs is selected and the selected task is scheduled to the selected processor. This process is repeated until all the tasks are scheduled.

The algorithm is adapted for scheduling in a heterogeneous environment by modifying the definition of DL. A term \( (v_i, p_j) = E_x(v_i) - E(v_i, p_j) \) is added to the expression of DL to account for the varying processing speeds, where \( E_x(v_i) \) is the median of execution times of task \( v_i \) over all processors \( p_j \). In order to consider how the descendants of \( v_i \) matches \( p_j \), another term called *Descendant Consideration (DC)* is added to the DL expression. The DC term is the difference between the median execution times of the most significant descendant \( D(v_i) \) to which \( v_i \) passes the most data and a lower bound on the time required to finish execution of \( D(v_i) \) after \( v_i \) finishes execution on \( p_j \). This reveals how well the most expensive descendant of \( v_i \) match \( p_j \) if \( v_i \) is scheduled on \( p_j \). In addition to the descendant consideration effect, the algorithm takes into account the situation where certain processors are not capable of executing some tasks. The DLS algorithm has been proved better than the HLFET algorithm.
Levelized Min Time (LMT) Algorithm

The LMT algorithm [69] proposed by Iverson et al., is a two phase scheduling algorithm. The first phase groups the tasks that can be executed in parallel using the level attribute. The second phase is a greedy method that assigns each task to the fastest available processor. A task at a lower level has higher priority for scheduling than a task at a higher level. Within the same level, the task with the highest average computation cost has the highest priority. If the number of tasks in a level is greater than the number of available processors, the fine-grain tasks are merged into a coarse-grain task until the number of tasks is equal to the number of processors. Then the tasks are sorted in reverse order (largest task first) based on average computation time. Beginning from the largest task, each task will be assigned to the processor that minimizes the sum of computation cost of the task and the communication costs with tasks in the previous layers and does not have any scheduled task at the same level. For a fully connected graph, the time complexity is \(O(v^2p)\) when there are \(v\) tasks and \(p\) processors.

Modified Mapping Heuristic (MMH) Algorithm

Gan et al., proposed the MMH algorithm [70]. The MMH algorithm improves the MH algorithm in three aspects. The goals of the MMH algorithm are to get a shorter schedule, to minimize contention delay and to minimize the number of processors used for execution without lengthening execution time. The first improvement by MMH is to try to achieve a shorter schedule by considering the task priority as well. The MH will schedule a task with the earliest ready time instead of one with the highest priority. A postponed event list is also maintained which stores the tasks that have lower priority but reaches the top of the original event list before higher priority tasks. An event in this list will only be scheduled when its priority level is the highest among all the unscheduled tasks. The next improvement is in the processor selection. The MH algorithm allocates a task on the earliest available processor and a tie is broken arbitrarily whereas, the MMH breaks the tie by selecting the processor which results in a minimum total communication cost for the task. The third improvement by MMH is brought about by noticing that the MH does not try to minimize the number of processors used. As a result, the processors
used may spread over a wider area of the mesh than necessary. MMH tries to locate processors in a fixed sequence and tries to localize the processors used every time a task needs to be scheduled. This means the processors chosen will be closer to each other and thus less communication costs will be incurred.

**Bubble Scheduling and Allocation (BSA) Algorithm**

Kwok and Ishfaq Ahmad proposed the BSA algorithm [71]. The algorithm initially schedules all the tasks to a single processor called *pivot processor* and it is selected as follows: The first processor in the heterogeneous system is considered and the corresponding heterogeneity factor is multiplied to the nominal execution cost of each task. Based on the set of actual execution costs the CP is computed. This process is repeated for other processors and eventually the processor that gives the shortest CP length based on actual execution costs is selected as the first *pivot processor*. After the parallel program is serialized to the first pivot processor, tasks have to be considered for possible migration to the neighbour processors in order to improve their finish times (bubble up). To determine whether a migration is beneficial or not, the algorithm computes the finish time of the task on a neighbour processor. To compute the start time, first the *Data Ready Time (DRT)* of a task is determined. The DRT of a task is defined as the latest arrival time of messages from its predecessors.

The algorithm uses a procedure called *ComputeMFT* to determine the finish time of every incoming message of the task on a neighbour processor. The predecessor which sends this latest message is called the *Very Important Predecessor (VIP)* of the task. Then another procedure called *ComputeFT* is called to determine whether a task can improve its finish time through migrating to a neighbour processor of the pivot processor. If the finish time is improved, then the task is rescheduled to the neighbour processor and its incoming and outgoing messages are also rearranged. Otherwise, if the finish time does not improve, then a task will also migrate if its VIP is scheduled to that neighbour processor. The rationale behind this heuristic decision is that if a task and its VIP are scheduled to the same processor, the successors of the task may subsequently improve their finish
times also. This process is repeated for all the remaining tasks on the pivot. Then a neighbour processor is chosen to be a new pivot. Thus, each processor in the heterogeneous system in turn will be assigned as the pivot in a breadth-first manner. Throughout the entire bubbling up process, messages are automatically routed in the migration process of tasks from the pivot processor to other processors. This process is repeated for all the processors in the processor list. The BSA algorithm provides better performance compared to the DLS algorithms for different cases.

**Fast Load Balancing (FLB) Algorithm**

The FLB algorithm [72] was proposed by Radulescu and Van Gemund. The FLB algorithm computes the *Earliest Execution Start Time (EEST)* for all ready tasks and selects the one with the lowest value at each scheduling step. The ready task is defined as the task having all its parents scheduled. In order to obtain the *EEST* of a given ready task on a partial schedule, the given task must be scheduled either to the machine from which the last message is sent or to the machine that becomes idle earliest.

**Critical Path On a Processor (CPOP) Algorithm**

Topcuoglu et al., proposed the CPOP algorithm [21] and it was developed around the critical path reduction scheme. The algorithm begins by setting the computation costs of the tasks and communication costs of the edges to their mean values. Then it assigns priority to the tasks using upward rank and downward rank. The upward rank of a task \( v \), is the largest sum of mean computation costs and mean communication costs along any directed path from task \( v \) to an exit task. The downward ranks are computed recursively by traversing the task graph downward starting from the entry task of the graph and it is the longest distance from the entry task to the task \( v \), excluding the computation cost of the task itself. The priority of each task is assigned with the summation of upward and downward ranks. A priority queue is constructed with the key of upward rank and downward rank. The critical path length is equal to the entry task’s priority. Initially, the entry task is the selected task and marked as a critical path task. An immediate successor of the selected task that has the highest priority value is selected and it is marked as a
critical path task. This process is repeated until the exit task is reached. For tie breaking, the first immediate successor which has the highest priority is selected. While scheduling the unscheduled tasks, the task with the highest priority is selected from the priority queue and if the selected task is on the critical path, then it is scheduled on the critical path processor, otherwise, it is assigned to a processor which minimizes the earliest execution finish time of the task. A critical path processor is the one that minimizes the cumulative computation costs of the tasks on the critical path. The CPOP algorithm is proved better than the DLS algorithm, the MH algorithm and the LMT algorithm in terms of average schedule length ratio.

**Heterogeneous Earliest Finish Time (HEFT) Algorithm**

Topcuoglu et al., proposed the HEFT algorithm [21] which has it root from the MCP algorithm [63] and is similar to the CPOP algorithm except in the ranking of tasks. The CPOP and the HEFT algorithms are the most frequently referred algorithms by the researchers. Like the CPOP algorithm it begins by setting the computation costs of tasks and communication costs of edges to their mean values. Each task is assigned a value called *upward rank*. The calculation of upward rank is similar to the CPOP algorithm. A task list is then generated by sorting all the tasks by decreasing order of their upward rank and the ties are broken on a random basis. At each scheduling step, the unscheduled task with the highest upward rank value is selected and assigned to the processor that minimizes its finish execution time using the *insertion-based scheduling policy*. When a processor $p_j$ is assigned a task $v_i$, the insertion-based scheduling policy considers all possible *idle time slots* on $p_j$ to find a time slot that is capable of accommodating the execution time of $v_i$. This must be done without violating the precedence constraints among tasks. An idle time slot on processor $p_j$ is defined as the idle time space between the start execution time and finish execution time of two tasks that are successively scheduled on $p_j$. The search starts from a time equal to the ready time of $v_i$ on $p_j$ and proceeds until it finds the first idle time slot with a sufficient large time space to accommodate the computation cost of $v_i$ on $p_j$. If no such idle time slot is found, the insertion-based scheduling policy inserts the selected task after the last scheduled task on $p_j$. The HEFT algorithm has a general time complexity of $O(n^2)$. The time complexity for
dense DAGs in which the number of edges is proportional to \( v^2 \) is \( O(v^2p) \). The HEFT algorithm is better than the CPOP algorithm by 7 percent, the DLS by 8 percent, the MH by 16 percent and the LMT algorithm by 52 percent in terms of average schedule length ratio. The HEFT algorithm is faster than the CPOP algorithm by 10 percent, the DLS algorithm by 84 percent, the MH algorithm by 32 percent and the LMT algorithm by 48 percent in terms of average speedup. The HEFT algorithm is the fastest one and the DLS is the slowest one.

**Heterogeneous Critical Parent Trees (HCPT) Algorithm**

Hagras et al., proposed the HCPT algorithm [22] for a bounded number of heterogeneous machines. The HCPT algorithm aims to achieve high performance and low complexity. The algorithm heuristic uses a new mechanism to construct the scheduling list \( L \) instead of assigning priorities to the application tasks. The heuristic divides the task graph into a set of unlisted-parent trees. The root of each unlisted-parent tree is a \( CN \). The algorithm consists of two phases, *listing tasks* and *machine assignment*. In the *listing phase*, the algorithm starts with the empty queue \( L \) and an auxiliary stack \( S \) that contains the \( CNs \) pushed in a decreasing order of their *Average Latest Start Time* (ALSTs), i.e., the entry node is on the top of \( S \) (\( top(S) \)). Consequently, \( top(S) \) is examined. If \( top(S) \) has an unlisted parent (i.e., has a parent not in \( L \)), then this parent is pushed on the stack. Otherwise, \( top(S) \) is popped and enqueued into \( L \). In the *machine assignment phase*, the algorithm tries to assign each task \( v_i \in L \) (list of ready task) a machine \( p_m \in P \) (set of available processors), that allows the task to finish its execution as early as possible. The complexity of the algorithm is \( O(pvn^2) \). The HCPT algorithm provides better results than the DLS, FCB and CPOP algorithms.

**Longest Dynamic Critical Path (LDCP) Algorithm**

The *Longest Dynamic Critical Path* (LDCP) algorithm [73] consists of three phases namely, *task selection*, *processor selection and status update*. In the *task selection phase*, a set of tasks that play an important role in determining the provisional \( SL \) are identified. To compute the LDCP, a *Directed Acyclic Graph that corresponds to a Processor (DAGP)* is constructed at the beginning of the
scheduling process for each processor. Given a DAG with \( n \) tasks and \( e \) edges and a heterogeneous system with \( m \) processors \( \{p_0, p_1, \ldots, p_{m-1}\} \) the DAGP of \( p_i \), called DAGP\(_{p_i} \), is the task graph constructed using the structure of the DAG, with sizes of tasks set to their computation costs at processor \( p_i \). The entry task of the LDCP is determined by locating a task \( v_i \) that has the highest \( URank \) value over all tasks on all DAGPs. The \( URank \) values of the tasks in a given DAGP are computed recursively by traversing the DAGP upward starting from exit task to entry task. The \( URank \) value of an exit task is equal to its size. The remaining tasks on the LDCP can be identified by recursively traversing the DAGP that contains task \( v_i \). The process of traversing starts from task \( v_i \) and moves downward. At each scheduling step the key task or the parent key task if the key task has unscheduled parents, is used to identify the task that will be selected for scheduling. A set of processors and its corresponding set of DAGPs, called active processors and active-DAGPs, are defined at each scheduling step.

**Processor selection phase:** In this phase, the selected task is assigned to a processor in the active processor set that minimizes its finish execution time using the insertion-based scheduling policy.

**Status update phase:** When a task is scheduled on a processor, the status of the system must be updated to reflect the new changes. The sizes of the tasks that identify \( v_i \) must be set to the computation cost of \( v_i \) on \( p_i \) on all DAGPs. When a task is assigned to processor \( p_i \), temporary zero-cost edges are added to DAGP\(_i \) from the task that identifies the last task scheduled on \( p_i \) (task \( v_k \)) to the ready tasks on DAGP\(_i \) that do not communicate with \( v_k \). This must be done after removing the previous temporary zero-cost edges from DAGP\(_i \). To reflect the new changes, the \( URank \) values of the tasks that identify the currently scheduled task and the previously scheduled tasks are updated on all DAGPs.

Table 2.2 presents the taxonomy of list-scheduling algorithms developed for heterogeneous processors along with their time complexity, network topology, observations made and the limitations.
Table 2.2 Taxonomy of list-scheduling algorithms developed for heterogeneous processors

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Network Topology</th>
<th>Complexity</th>
<th>Observations/Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>arbitrarily connected</td>
<td>O(p^1v^2 + pv^2)</td>
<td>More processors are used to generate schedules. Higher time complexity and does not consider task insertion</td>
</tr>
<tr>
<td>DLS</td>
<td>arbitrarily connected</td>
<td>O(v^1p)</td>
<td>Heterogeneity in links and task insertion are not considered. Generates better schedules than the HLFET</td>
</tr>
<tr>
<td>LMT</td>
<td>fully connected</td>
<td>O(v^2p^2)</td>
<td>Gives high priority to task with larger computation time. Takes high time complexity</td>
</tr>
<tr>
<td>BIL</td>
<td>fully connected</td>
<td>O(v^2p log p)</td>
<td>Provides optimal schedule for linear DAGs and takes higher time complexity</td>
</tr>
<tr>
<td>MMH</td>
<td>arbitrarily connected</td>
<td>O(p^1v^2 + pv^2)</td>
<td>Works badly when the task graphs have multiple entry tasks. Takes high time complexity. Proved better than the MH algorithm</td>
</tr>
<tr>
<td>BSA</td>
<td>fully connected</td>
<td>O(p^2ev)</td>
<td>Higher time complexity, better than the DLS algorithm. Provides optimal solution for the homogeneous systems and links</td>
</tr>
<tr>
<td>FLB</td>
<td>fully connected</td>
<td>O(v(log v+log p)+c)</td>
<td>Generate poor schedules for irregular task graph and large processor speed variance. Takes less time complexity</td>
</tr>
<tr>
<td>CPOP</td>
<td>fully connected</td>
<td>O(v^2p)</td>
<td>High priority is given to the tasks lying in the CP. Processor is selected for CP tasks in advance which may not work well, owing to the involvement of precedence and communication cost. Gives better schedule than LMT and MH algorithms</td>
</tr>
<tr>
<td>HEFT</td>
<td>fully connected</td>
<td>O(v^2p)</td>
<td>Generates inconsistent schedules, since random selection is used when tasks have the same priority. Proved better than DLS, LMT, MH and CPOP</td>
</tr>
<tr>
<td>HCPT</td>
<td>fully connected</td>
<td>O(v^2p)</td>
<td>Does not consider the task insertion. Better than the CPOP, FLB and the DLS</td>
</tr>
<tr>
<td>LDCP</td>
<td>fully connected</td>
<td>O(v^1p)</td>
<td>Provide better results than HEFT and DLS, but takes higher time complexity</td>
</tr>
</tbody>
</table>
2.3.2 Clustering Algorithms

In parallel and distributed systems, clustering is an efficient way to reduce communication delay in DAGs by grouping heavily communicating tasks to the same labelled clusters and then assigning tasks in a cluster to the same processor. The clustering algorithms in general have two phases: the task clustering phase that partitions the original task graph into clusters and a post-clustering phase which can refine the clusters produced in the previous phase to obtain the final task-to-processor map. At the beginning of the clustering process, the tasks selected for clustering can be any task, not necessarily a ready task. Then, the clustering process redefines the previous clusters by merging some clusters. A task cluster could be linear or non-linear. A clustering is called non-linear if two independent tasks are mapped in the same cluster, otherwise it is called linear. Figure 2.2 depicts the two types of clustering for a sample task graph. The algorithm in this class maps the tasks in a task graph to an unlimited number of clusters. In practice, an additional cluster merging (post-clustering) step is needed after clusters are generated, so that the number of clusters generated can be equal to the number of available processors.

![Diagram showing two types of clustering](image)

(a) A Linear clustering  
(b) A non-linear clustering

Figure 2.2 Types of clustering
The problem of obtaining an optimal clustering of a general task graph is NP-complete thus heuristics are designed to deal with this problem [23, 74]. Though the task scheduling problem has been extensively explored by the researchers for homogeneous processors in the past, interestingly only a very few algorithms have been developed using the clustering technique [24, 75-79] including the Clustering and Scheduling (CS) [78] and Dynamic Critical Path Scheduling (DCPS) [79]. Also very few algorithms have been proposed for the heterogeneous processors [25, 80].

2.3.2.1 Clustering Algorithms for Homogeneous Processors

The clustering algorithms have not been explored widely for the distributed systems in general because of the scheduling overhead involved in this technique. Thus only a limited numbers of algorithms in this category have been found in the literature and they are described below.

Linear Clustering Algorithm (LCA)

Kim et al., proposed the LC algorithm [75]. This is one of the earliest algorithms that use the clustering technique for task scheduling. Initially the algorithm merges tasks to form a single cluster, based on the CP. The algorithm first determines the set of tasks constituting the CP then promptly schedules all of the CP tasks to a single processor. The tasks that lie in the CP and the entire edges incident on these tasks are then removed from the DAG. The algorithm immediately zeroes the edges on the entire CP. However, when an edge is zeroed, the CP may change. The edge that should be zeroed next may not be on the original CP.

Edge Zeroing (EZ) Algorithm

Sarkar proposed the EZ algorithm [76] which selects clusters for merging based on edge weights. At each step, the algorithm finds the edge with the largest weight. The two clusters that are incident by the edge are merged if the merging (thereby zeroing the largest weight) does not increase the completion time. After two clusters are merged, the ordering of tasks in the resulting cluster is based on the SLs of the tasks.
**Dominant Sequence Clustering (DSC) Algorithm**

Yang et al., proposed the DSC algorithm [77] and the algorithm is based on an attribute called the *dominant sequence* which is essentially the critical path of the partially scheduled task graph at each step. During scheduling the DSC algorithm checks whether the highest $CP$ task is a ready task by using a composite attribute ($h$-level + $t$-level) and if the highest $CP$ task is a ready task, the DSC algorithm schedules it to a processor that allows the minimum start time. Such minimum start time may be achieved by *rescheduling* some of the task's parent tasks to the same processor. On the other hand, if the highest $CP$ task is not a ready task, the DSC algorithm does not select it for scheduling. Instead, the DSC algorithm selects the highest task that lies on a path reaching the $CP$ for Scheduling. The DSC algorithm schedules it to the processor that allows the minimum start time of the task provided that, such processor selection will not delay the start time of a not yet scheduled $CP$ task. The delayed scheduling of the $CP$ tasks allows the DSC algorithm to incrementally determine the next highest $CP$ task. Although the DSC algorithm can identify the most important task at each scheduling step, it does not schedule a $CP$ task if it is not a ready node. Another drawback of the DSC algorithm is that it uses more processors than necessary because it schedules a task to a new processor if its start time cannot be reduced by scheduling the task to any processor already in use.

**Mobility Directed (MD) Algorithm**

The MD algorithm [68] selects a task at each step for scheduling based on an attribute called the *relative mobility*. Mobility of a task is defined as the difference between a task's *earliest start time* and latest start time. Similar to the ALAP binding mentioned earlier, the earliest possible start time is assigned to each task through the *As Soon As Possible (ASAP)* binding which is done by traversing the task graph downward from the entry tasks to the exit tasks and by pulling the tasks upward as much as possible. Relative mobility is obtained by dividing the mobility with the task's computation cost. Essentially, a task with zero mobility is a task on the CP. At each step, the MD algorithm schedules the task with the smallest mobility to the first processor which has a large time slot to accommodate the task without considering the minimization of the task's start time. After a task is scheduled, all the relative
mobilities are updated. As opposed to the MCP algorithm, the MD algorithm determines task priorities dynamically. Although the MD algorithm can correctly identify the CP tasks for scheduling at each step, the selection of a suitable time slot and a processor are not done properly. The major problem with the MD algorithm is that it pushes scheduled tasks downwards to create a large enough time slot to accommodate a new task without paying any regard to the degradation in the schedule length. It may happen that pushing down the tasks may increase the final schedule length. The second drawback of the MD algorithm is that it looks for a suitable processor by scanning the processors one by one starting with the first processor. This processor selection criterion does not precisely make any effort to minimize the start time of tasks at each step. Another problem with the MD algorithm is that it inserts a task into an idle time slot on a processor without considering whether the descendants of that task can be scheduled in a timely manner.

2.3.2.2 Clustering Algorithms for Heterogeneous Processors

Applying a clustering technique to heterogeneous processors is highly difficult because of various system heterogeneity factors such as rate of data transfer, processor speed, etc. Therefore, the techniques used for the formation of clusters in homogenous processors cannot be directly applied to the heterogeneous processors, and for this reason only very few numbers of algorithms have been proposed using clustering technique for the heterogeneous processors which are described below:

Clustering for Heterogeneous Processors (CHP) Algorithm

Boeres et al., proposed the CHP algorithm [80]. The algorithm consists of two stages. The two stage cluster-based strategy first assumes a virtual homogeneous environment and creates task clusters using the clustering existing clustering technique. In the second stage, a subset of the generated clusters are mapped to the heterogeneous processors considering the execution cost of the tasks on the processors and the transmission characteristics on the communication links. During the second stage, the CHP maps the set of clusters of the virtual schedule to
the heterogeneous target environment. Initially, all clusters in the list are considered unmapped. Two phases such as, cluster choice phase and processor choice phase are used to map the cluster to the processor. In the cluster choice phase, the next cluster is selected to be mapped to a processor of the target architecture in accordance with a given priority. This phase uses a dynamic variant of the bottom level priority to select the next cluster to be mapped. The bottom level of the unmapped clusters is re-calculated and the one with the highest value is selected. In the case of ties, the cluster with the least affinity with those already scheduled will be chosen. In the processor choice phase, another priority is used to select the processor \( p \) to which the chosen cluster will be assigned. This phase of the algorithm will take care of minimizing the overall makespan of all the mapped clusters rather than just minimizing the finish time of the cluster currently being mapped.

**Triplet Algorithm**

Cirou et al., proposed a clustering algorithm for heterogeneous processors called triplet algorithm [25]. The first step is the clustering of tasks. Tasks are grouped into clusters in order to suppress unnecessary communications and some clusters are then merged. For merging clusters the Triplet algorithm considers tasks which belong to a path of length 2 in the task graph. Every path of length 2 is composed of three tasks and is called a triplet. The second step is the workstation clustering. In order to efficiently map clusters to workstations, workstations which are somehow equivalent are grouped together. In the last step, task clusters are mapped to workstation clusters. For mapping, the algorithm allocates task’s cluster, in the order, to workstations having the best completion time as long as the load limit is not exceeded. This mapping ensures that the largest communications are done on the best links and each cluster of workstations has nearly the same time computation load. The triplet algorithm gives better performance than HEFT algorithm.

Taxonomy of cluster-based task scheduling algorithms developed for homogeneous and heterogeneous processors over a period of time is given in Table 2.3 along with their time complexity, processing environment and limitations.
Table 2.3 Taxonomy of cluster-based task scheduling algorithms developed for homogeneous and heterogeneous processors

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Environment</th>
<th>Complexity</th>
<th>Observations/Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCA</td>
<td>Homogeneous</td>
<td>O(e(v+e))</td>
<td>Assume unlimited number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of processors</td>
</tr>
<tr>
<td>EZ</td>
<td>Homogeneous</td>
<td>O(v(v+e))</td>
<td>Assume unlimited number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of processors</td>
</tr>
<tr>
<td>DSC</td>
<td>Homogeneous</td>
<td>O(v^3)</td>
<td>Provide better schedule</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for coarse grain DAGs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The deficiency of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DSC algorithm is that</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>it uses more processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>than necessary</td>
</tr>
<tr>
<td>CS</td>
<td>Homogeneous</td>
<td>O(v^3)</td>
<td>Assume unlimited number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of processors</td>
</tr>
<tr>
<td>DCPS</td>
<td>Homogeneous</td>
<td>O(e+v log v)</td>
<td>Generate better schedules</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>than the DSC with less</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>time complexity</td>
</tr>
<tr>
<td>Triplet</td>
<td>Heterogeneous</td>
<td>O(v(log v +pv))</td>
<td>Works well when there</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>is more number of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>processors with similar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>characteristics</td>
</tr>
<tr>
<td>CHP</td>
<td>Heterogeneous</td>
<td>O(v^2p)</td>
<td>Performs well when the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>number of fast processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>is not severely restricted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>and when the degree of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>heterogeneity in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>system is low</td>
</tr>
</tbody>
</table>

2.3.3 Task Duplication-based Algorithms

An alternate technique to shorten the makespan or schedule length is to duplicate tasks on different processors. The main idea behind the task duplication-based scheduling is utilizing processor idle time to duplicate predecessor tasks [81, 82]. This can avoid the transfer of data from a predecessor to a successor thus reducing the communication cost, network overhead and potentially reduce the start times of waiting tasks. Task duplication-based scheduling is much useful for systems having high communication latencies and low bandwidths. For example, the schedule lengths obtained, with task duplication and without task duplication, for the task graph given in Figure 2.3 along with the estimated task's computation time given in Table 2.4 are 18 and 21 respectively and it is shown in the Figure 2.4. Task duplication-based scheduling algorithms have been developed for homogeneous
processors [28, 29, 83-89] and heterogeneous processors [30-34, 90, 91] in the past. The pseudo code of the general task duplication-based scheduling algorithms is given below:

/* Task Duplication */

1. \textit{begin}
2. \textit{read} the DAG and its associated parameters;
3. assign a priority to each task in the task graph \( G \);
4. add all ready tasks to the ready queue in order of decreasing task priority;
5. \textit{while} (ready queue is not empty) \textit{do}
6. \textit{begin}
   a. take the task from the head of the queue;
   b. schedule the task on an available processor without duplication of ancestors;
   c. then consider the duplication time slot (the idle time period from the finish time of the last scheduled task on the processor and the start time of the task currently under consideration) for the improvement of start time of a task:
   d. if suitable processor is found then duplicate the critical parent of the task into the duplication time slot:
   e. repeat for all the processors;
   f. schedule the task on the processor that gives the earliest start time;
7. \textit{end};
8. \textit{end}.

![Figure 2.3 A random task graph with 5 tasks](image)

Table 2.4 Computation costs of the tasks in Figure 2.3 on two processors

<table>
<thead>
<tr>
<th>Task</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
Figure 2.4 SL generated for the task graph in Figure 2.3 using (a) without and (b) with task duplication

2.3.3.1 Task Duplication-based Algorithms for Homogeneous Processors

Task duplication-based scheduling has been extensively explored for homogeneous processors. Consequently, a variety of task duplication-based scheduling algorithms have been developed over a period of time and this includes the recently proposed algorithms such as Extended Task Duplication-based Scheduling (ETDS), Fast Duplication-based on Earliest Finish Time (FDEFT) and Interpersonal relationships Evolution Algorithm (IREA). A few existing task duplication-based scheduling algorithms developed for the homogeneous processors which form the basis for the development of task duplication-based scheduling algorithms for heterogeneous processors are briefly described below:

Duplication Scheduling Heuristic (DSH) Algorithm

The DSH algorithm [67] proposed by Kruatrachue, as an extension to the ISH algorithm, was the first algorithm to utilize task duplication. Since then, a multitude of algorithms for scheduling with task duplication have been proposed.
The DSH algorithm uses the idea of list-scheduling combined with task duplication to reduce the makespan. Tasks are given priorities using static b-level to indicate the urgency of being scheduled. While selecting a processor for a task, the algorithm first calculates the start time of the task on the processor without duplication of any predecessor. Next, it duplicates the predecessors of the task into the idle time slot of the processor until either such slot is unavailable or the start time of the task does not improve. The algorithm is as follows:

1. Compute the static b-level for each task and set it as its priority. Sort the tasks into a list L in the order of decreasing priority.
2. Pick the first task \( n \) in \( L \).
3. For each processor \( P \), perform the following steps:
   a. Compute the Start Time (ST) of \( n \) on \( P \). Set the candidate as \( n \).
   b. Consider the set of immediate predecessors of the candidate \( n \). Let \( n' \) be the immediate predecessor that is not scheduled on \( P \) and whose message for a candidate has the latest arrival time. Duplicate \( n' \) into the earliest idle time that can accommodate it on \( P \).
   c. If a time slot is unavailable, then ST is recorded and go to step (3). Otherwise, the candidate's start time is replaced by the new start time if the new start time is smaller. Set the candidate as \( n' \). Go to step (3b).
4. Schedule task \( n \) to the processor that gives the smallest ST and perform all necessary duplications on that processor.
5. Repeat steps 2 to 4 until all the tasks are scheduled.

**Bottom-up Top-down Duplication Heuristic (BTDH) Algorithm**

Chung et al., proposed the BTDH algorithm [83] which is an extension of the DSH algorithm described above. The major improvement brought by the BTDH algorithm over the DSH algorithm is that the former keeps on duplicating ancestors of a task even when the duplication time slot is filled up and the start time of the task under consideration temporarily increases. This strategy is based on the intuition that the start time may eventually be reduced by duplicating all the necessary ancestors.
As the BTDH algorithm also uses static level for priority assignment, it may not always accurately capture the relative importance of tasks.

**Linear Clustering with Task Duplication (LCTD) Algorithm**

Shirazi et al., proposed the LCTD algorithm [84]. The LCTD algorithm first iteratively clusters tasks into larger tasks. At each step, tasks on the longest path are clustered and removed from the task graph. This operation is repeated until all tasks in the graph are removed. After performing the clustering step, the LCTD algorithm identifies those edges among clusters that determine the overall completion time. The algorithm then attempts to duplicate the parents corresponding to these edges to reduce the start times of some tasks in the clusters. Linear clustering may not always accurately identify the tasks that should be scheduled to the same processor. In addition to the context of duplication-based scheduling, linear clustering prematurely constrains the number of processors used. This constraint can be detrimental because the start times of some critical tasks may possibly be reduced by using a new processor in which its ancestors are duplicated.

**Duplication First and Reduction Next (DFRN) Algorithm**

Park et al., proposed the DFRN algorithm [85]. The DFRN algorithm first duplicates all parent tasks in a bottom-up fashion to the parent that has been scheduled on the same processor, without estimating the effect of their duplications. Then, each duplicated task is removed if the task does not meet certain conditions. DFRN applies the duplication only for the critical processor with the hope that the critical processor is the best candidate for the join task (task with multiple immediate predecessors). The Critical Immediate Parent (CIP) of a join task $n$ is the immediate parent whose message for the join task has the latest arrival time. The processor on which the CIP of $n$ is scheduled is called the critical processor of $n$.

**Critical Path Fast Duplication (CPFD) Algorithm**

Ahammad and Kwok proposed a duplication-based algorithm called CPFD algorithm [87]. Initially the tasks/nodes in the task graph are classified into three
categories namely, Critical Path Nodes (CPN), In-Branch Nodes (IBN) and Out-Branch Nodes (OBN). CPN nodes are the nodes that lies on the critical path and these nodes are most important nodes because their finish times effectively determine the final makespan. An IBN node is a node that is not a CPN and from which there is a path reaching a CPN. The IBNs are also important because timely scheduling of these nodes can help to reduce the start times of the CPNs. An OBN is a node that is neither a CPN nor an IBN. The OBNs are relatively less important because they usually do not affect the makespan. The algorithm has the following steps:

1. Determine a CP. Ties if any are broken by selecting the one with a larger sum of computation costs. Based on the importance of a node, a priority list called CP-Dominant Sequence is constructed in a way that CPNs can be scheduled as soon as possible. In addition, precedence constraints are also preserved.
2. Select the first unscheduled CPN in the CPN-Dominant Sequence as the candidate node \( nc \).
3. Let \( N \) be a set of processors, including all the processors holding the candidate's parent nodes and an empty processor.
4. For each processor \( p \) in \( N \), find the EST of the candidate on \( p \) and record it.
5. Schedule the candidate to the processor \( p \) that gives the smallest value of EST. All necessary duplications are performed.
6. Repeat the process from steps 2 - 5 for each OBN with \( N \) containing all the processors in use together with an empty processor. The OBNs are considered one by one in topological order.
7. Repeat steps 2 to 6 until all CPNs are scheduled.

The process of determining the candidate node \( nc \)'s EST on processor \( p \) is as follows: Let \( ST \) be the start time of candidate node \( nc \) on \( p \) and \( n \) be the immediate predecessor node that is not scheduled on \( p \) and whose message for \( nc \) has the latest arrival time. Try to duplicate the node \( n \) on the earliest idle time slot that can accommodate the node \( n \) on processor \( p \). If the duplication is successful and the new start time of \( nc \) is smaller than \( ST \), then let \( ST \) be the new start time. Now set the
candidate to node \( n \) and repeat the process from the beginning until the duplication is unsuccessful.

**Task Duplication-based Scheduling (TDS) Algorithm**

Darbha et al. proposed the TDS algorithm [28]. The TDS algorithm initially computes \( EST \), *Earliest Completion Time* (ECT), *Latest Allowable Start Time* (LAST), *Latest Allowable Completion Time* (LACT) and *Favourite Predecessor* (FPred) for each task in a DAG. The \( LAST \) is the latest time when a task should be started otherwise, successors of this task will be delayed. The \( FPred \) of a task \( v \), are those which are predecessors of \( v \) and if \( v \) is assigned to the same processors on which these tasks are running it minimizes the \( EST(v) \). The level value of a task which denotes the length of the longest path from that task to an exit task, ignoring the communicating cost along that path is used as the priority to determine the processing order of each task. Based on these values, task clusters are created iteratively. The clustering step is like a depth-first search from an unassigned task having the lowest level value to an entry task. Once an entry task is reached, a cluster is generated and tasks in the same cluster will be assigned to the same resource. In this step, the \( LAST \) and \( LACT \) values are used to determine whether duplication is needed. For example, if task \( v_j \) is a favourite predecessor of task \( v_i \) and \( (LAST(i) - LACT(j)) < C_{ji} \), where \( C_{ji} \) is the communication cost between \( v_j \) and \( v_i \), \( v_i \) will be assigned to the same processor as \( v_j \), and if \( v_j \) has been assigned to other processors, it will be duplicated to \( v_i \)'s processor. In the clustering step, the DAG is traversed similar to DFS from the exiting task, and the complexity of this step would be the same as the complexity of a general search algorithm, which is also \( O(v+e) \). So the overall complexity is \( O(v+e) \). In a dense DAG, the number of edges is proportional to \( O(v^2) \), which is the worst case complexity of duplication algorithm.

Table 2.5 presents the taxonomy of the task duplication-based scheduling algorithms developed for the homogeneous processors including the *Scalable Task Duplication-based Scheduling (STDS)* algorithm [86] along with time complexity, observations and limitations of the algorithms.
Table 2.5 Taxonomy of task duplication-based scheduling algorithms developed for homogeneous processors

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Processors (bounded / unbounded / scalable)</th>
<th>Complexity</th>
<th>Observations / Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSH</td>
<td>unbounded</td>
<td>O(v^4)</td>
<td>Duplicates immediate predecessor tasks when the duplication reduces the EST. Higher time complexity</td>
</tr>
<tr>
<td>BTDH</td>
<td>unbounded</td>
<td>O(v^4)</td>
<td>Duplicates predecessor tasks even though the duplication increases the EST of a task, but in later stage the duplication decreases some other dependant tasks. Higher time complexity and proved better than HLFET and ETF</td>
</tr>
<tr>
<td>LCTD</td>
<td>unbounded</td>
<td>O(v^1 log v)</td>
<td>In the context of duplication based scheduling, linear clustering prematurely constrains the number of processors used. The start times of some critical nodes may possibly be reduced by using a new processor in which its ancestors are duplicated</td>
</tr>
<tr>
<td>DFRN</td>
<td>unbounded</td>
<td>O(v^1)</td>
<td>Performs exhaustive task duplication</td>
</tr>
<tr>
<td>STDS</td>
<td>scalable</td>
<td>O(v^2)</td>
<td>Scales the schedule to the available number of processors in the system</td>
</tr>
<tr>
<td>CPFD</td>
<td>unbounded</td>
<td>O(v^4)</td>
<td>Tasks in the CP are duplicated as early as possible by considering the precedence constraints. Provides better results than DSH, BTDH and LCTD</td>
</tr>
<tr>
<td>TDS</td>
<td>unbounded</td>
<td>O(v^2)</td>
<td>Does not allow any parent task of a join task to be scheduled in the same processor. More restricted optimality conditions</td>
</tr>
<tr>
<td>ETDS</td>
<td>unbounded</td>
<td>O(v^2d), d number of incoming edges of tasks</td>
<td>Allow the two or more parent tasks of a join task to be scheduled in the same processor. Less restricted optimality conditions. Proved better than TDS</td>
</tr>
<tr>
<td>FDEFTI</td>
<td>bounded</td>
<td>O(v^3p)</td>
<td>Higher time complexity and proved to be better than MCP and FLB</td>
</tr>
<tr>
<td>IREA</td>
<td>bounded</td>
<td>O(v^2(v + e))</td>
<td>Provides better result than the DSC, MCP and LAST</td>
</tr>
</tbody>
</table>
2.3.3.2 Task Duplication-based Algorithms for Heterogeneous Processors

A fair amount of research work has been carried out for the development of task duplication-based scheduling algorithms for heterogeneous processors in the past and consequently few algorithms were developed.

Levelized Duplication-Based Scheduling (LDBS) Algorithms

Dogan et al., proposed two task duplication-based scheduling heuristics (version 1 and version 2) [30] for the heterogeneous processors and these heuristics are referred in this thesis as LDBS1 and LDBS2 algorithms. LDBS algorithms use a level sorting technique to arrange the tasks in the application DAG into various precedence levels (referred as level). The tasks belonging to the same level have no data dependencies among them, and hence can be executed concurrently. Tasks are then scheduled level by level starting from the top. The tasks in the topmost level (= 0) are scheduled by using the well known min–min heuristic borrowed from the metatask scheduling techniques. For all other levels algorithm does an exhaustive search for the most suitable task-machine pair that gives earliest finish time. Insertion-based duplication technique is employed to further lower the finish times of tasks, and to achieve a performance much better than the HEFT algorithm. However, the complexity of the two algorithms is quite high \( O(p^3e^v) \) and \( O(p^3e^v^3) \) where ‘e’ corresponds to the number of edges in the DAG. Further, search for the most suitable task-machine pair in LDBS1 algorithm may overlook the priority of a task. A low priority task may get scheduled earlier if it finds a better match in the network. LDBS2 algorithm rectifies this by scheduling the tasks in a given level in the order of their bottom-level priority but, due to the levelized approach as adopted by both of the LDBS algorithms, this priority gets localized to a particular precedence level, which may not reflect the true priority for scheduling a task.

Heterogeneous Critical Node First (HCNF) Algorithm

Baskiyar et al., developed the HCNF algorithm [31]. The HCNF is list-scheduling algorithm combined with task duplication. The algorithm proceeds by identifying the critical path(s) in the DAG. If there is more than one critical path, one critical path is randomly chosen. A list of free tasks (ready tasks) is constructed.
Priorities to the tasks within the list are assigned as follows: tasks falling in the critical path receive the highest priority, followed by those with the highest computation. A task \( v_i \) in the list is scheduled onto a processor \( p \), that gives the lowest \( EFT \) for task \( v_i \). After a task has been scheduled, the new free tasks, if any, are added to the list. Such an approach facilitates local optimization by giving priority to heavier free tasks and it also exploits heterogeneity. Furthermore, to reduce the communication time, the enabling task or the favourite predecessor is considered for duplication. The task duplication step is incorporated in the computation of \( EST \) of task \( v_i \). The complexity of the HCNF algorithm is \( O(v^2 \log v) \). The HCNF algorithm is proved to be better than the HEFT algorithm.

**Task duplication-based scheduling Algorithm for Networks of Heterogeneous systems (TANH) Algorithm**

Bajaj et al., developed the TANH algorithms [32]. The TANH algorithm has its roots from the TDS algorithms [28, 82] developed for homogeneous processors and heterogeneous processors. In the TANH algorithm, the authors have introduced a new parameter called *Favourite Processor (FP)*. A processor is defined as favourite processor for task \( v_i \), if using that processor for execution of results in a minimum completion time for task \( v_i \). Other parameters of a task are computed based on the value of \( Fpred \) (discussed in the TDS algorithm). The TANH algorithm is easily scalable upward or downward and the algorithm has the following steps:

1. The DAG is traversed in a top-down fashion to compute the earliest start and completion times (\( EST \) and \( ECT \)) for a task \( v_i \in V \), indicating when a task could be started or completed at the earliest possible time.

2. The level of a task, \( level(v_i) \) is computed and is defined to be the highest value of the summation of computation costs along different paths from task \( v_i \) to the exit task and indicates the minimum possible time needed from the current task to the end of the program.

3. The favourite processors from \( 1 \) to \( n \), i.e., \( FP_{1}(v_i) \) to \( FP_{n}(v_i) \), where \( n \) is the number of available processors and \( level(v_i) \) for each task are also computed in a top-down traversal.
4. The **LAST** and **LACT** (discussed in the TDS algorithm) of the tasks are calculated in a bottom-up traversal of the DAG.

5. The initial clusters are formed and assigned to its first **FP**, and if the first **FP** has already been assigned, then it is assigned to the second and so on.

6. A fine-tuning of the final schedule is carried out depending upon the available number of processors, in comparison with the required number of processors. If the available number of processors is higher, additional duplication is carried out. On the other hand, if the available processors are fewer, the algorithm can be designed for appropriately scaling down the number of clusters to the available number of processors. Processor reduction is done by using the procedure given in [82].

Compared to the homogeneous version of the algorithm (TDS algorithm), the heterogeneous version has higher complexity, which is $O(t^2p)$. The TANH algorithm is compared with the *Best Imaginary Level scheduling (BIL)* algorithm [61] and proved that the TANH is better than the BIL algorithm in terms of various scheduling metrics.

**Heterogeneous Critical Parents with Fast Duplicator (HCPFD) Algorithm**

Hagras et al., proposed the HCPFD algorithm [33]. The algorithm has its roots from the CPNT algorithm [52] proposed by the same authors earlier for homogeneous computing environments. The HCPFD algorithm aims to achieve high performance and low complexity. The algorithm consists of two phases, a *listing phase* which is a simplified version of the CNPT heuristic for heterogeneous environments and suggested low complexity duplication mechanism as a *machine assignment phase*. In the *listing phase*, the heuristic divides the task graph into a set of unlisted parent-trees. The root of each parent-tree is a **Critical Node (CN)**. A CN is defined as the node that has zero difference between its **Average Earliest Start Time (AEST)** and **Average Latest Start Time (ALST)**. The algorithm starts with an empty queue $L$ and an auxiliary stack $S$ that contains the CNs pushed in decreasing order of their ALSTs, i.e. the entry node is on the top of $S$ ($top(S)$). Consequently.
top(S) is examined. If top(S) has an unlisted parent (i.e. has a parent not in L), then this parent is pushed on the stack. Otherwise, top(S) is popped and enqueued into L. The machine assignment phase simply selects the machine $p_m$ that minimizes the Task Finish Time (TFT) of $v_i$ and duplicates its critical parent at the idle time between $v_i$ and the previous task on $p_m$, if this time slot is enough and this duplication will reduce the Task Start Time (TST) of $v_i$ on $p_m$. The authors compared the performance of the HCPFD algorithm with the non duplication-based scheduling algorithms such as the FLB, HEFT and the CPOP and proved that the HCPFD algorithms outperform these algorithms.

Dynamic Critical Path Duplication (DCPD) Algorithm

Liu et al., proposed the DCPD algorithm [34]. The DCPD initially computes the average execution rates and the average communication rates for all heterogeneous processors. The algorithm initially assumes that each task is assigned to one virtual processor and that the communication overhead between tasks is the average communication rate time the communication volume between tasks. Then the DCPD scheduling algorithm estimates b-levels for all tasks by using the average execution rates for all heterogeneous processors. It determines the critical path of the task graph and selects the next node to be scheduled in a dynamic fashion. In the scheduling process, a task is called examined after it is scheduled to some processor and unexamined before it is scheduled to some processor. The set of the unexamined tasks consists of the ready set, the partially ready set and the unready set. In the ready set, the task’s immediate predecessors are all examined. In the partially ready set, at least one of the task’s immediate predecessors is examined and at least one of its immediate predecessors is unexamined. In the unready set, none of the task’s immediate predecessors of any unready task is examined. Let the tasks without predecessors be in the ready set. When the ready set is not empty, the DCPD scheduling algorithm repeats the following steps:

1. Calculates the priority of tasks in the unexamined ready set. Priority for task $v_l$ is computed as $\text{priority}(v_l) = \tau, \dot{a} (p(v_l)) + b\text{-level}(v_l) - u\text{-level}(v_l)$, where $\tau,$
\((p(v_i))\) be the ACC when task \(v_i\) is allocated to \(p\), and the \(u\)-level\((v_i)\) of an unexamined ready node \(v\), as the shortest average length from entry task to it.

2. Let the dominant task, \(v_{d_1}\), be the task with the maximal priority in the ready set.

3. Find out the processor, \(p(v_{d_1})\), where the \(ect(v_{d_1})\) could be obtained by inserting policy.

4. Find out the dominant predecessors, \(v_{d_2}\), and the related data ready time, \(drt(v_{d_2})\).

5. If there is an enough free time slot from \(drt(v_{d_2})\) to \(est(v_{d_1})\) to schedule \(v_{d_2}\) to \(p(v_{d_1})\), then schedule the duplicated \(v_{d_2}\) to \(p(v_{d_1})\) to advance the \(est(v_{d_1})\) by the inserting policy.

6. Allocate \(v_{d_1}\) to \(p(v_{d_1})\), and make \(v_{d_1}\) examined.

7. Re-check the ready set and re-compute the \(t\)-levels for all tasks in the unexamined ready set.

8. Repeat steps 1 to 7 until all tasks are examined.

The time complexity of the DCPD scheduling algorithm is \(O((v+e)vp))\). The DCPD algorithm is proved to be better than the HCPFD algorithm by 69.96 percent and the TANH algorithm by 75.63 percent based on the average schedule lengths generated.

The taxonomy of the existing duplication-based scheduling algorithms developed for the heterogeneous processors including the recently developed algorithms such as Heterogeneous Limited Duplication (HLD), Duplication-based Bottom-Up Scheduling (DBUS) and the DCPD with their time complexity along with the observations and the limitations is tabulated and given in Table 2.6.
Table 2.6 Taxonomy of task duplication-based scheduling algorithms developed for heterogeneous processors

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Processors (bounded / scalable)</th>
<th>Complexity</th>
<th>Observations/ Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBS1</td>
<td>bounded</td>
<td>$O(v^3 ep^3)$</td>
<td>Higher time complexity and a low priority task may get scheduled earlier if it finds a better match in the network. Proved better than the HEFT.</td>
</tr>
<tr>
<td>LDBS2</td>
<td>bounded</td>
<td>$O(v^3 ep^2)$</td>
<td>Higher time complexity. Tasks more likely on the CP are selected first for scheduling. Proved better than HEFT and LDBS1.</td>
</tr>
<tr>
<td>HCNF</td>
<td>bounded</td>
<td>$O(v^2+v (p+v log v))$</td>
<td>Same as LDBS2 algorithm but lesser time complexity. Proved better than HEFT.</td>
</tr>
<tr>
<td>TANH</td>
<td>scalable</td>
<td>$O(v^2)$</td>
<td>Fork tasks must have equal execution time on all the processors. Proved better than BIL.</td>
</tr>
<tr>
<td>HCPFD</td>
<td>bounded</td>
<td>$O(v^2 p)$</td>
<td>Compared with non duplication based scheduling algorithms HEFT, CPOP and the FLB and proved better than these algorithms.</td>
</tr>
<tr>
<td>HLD</td>
<td>bounded</td>
<td>$O(v^2 max + e)$, where $d_{max}$ is the maximum in/out degree of a task</td>
<td>Higher time complexity. Outperforms HEFT, CPOP and the LDBS.</td>
</tr>
<tr>
<td>DBUS</td>
<td>bounded</td>
<td>$O(v^2 p^2)$</td>
<td>Higher complexity, proved better than the HEFT, HCNF and the HCPFD.</td>
</tr>
<tr>
<td>DCPD</td>
<td>bounded</td>
<td>$O((v+e)vp)$</td>
<td>Outperforms the CPNT, TANH and the HCPFD algorithms. But redundant task duplications.</td>
</tr>
</tbody>
</table>
2.4 Guided Random Search-based Algorithms

Genetic algorithms are the most widely studied guided random search techniques for the task scheduling problems [92-94]. The task scheduling problem with respect to guided random search is a search problem where the search space consists of an exponential number of possible schedules. Guided random search algorithms have been used extensively to solve very complex problems. Evolution computation and stochastic relaxation are the two major categories of guided random search algorithms. Simulated annealing [95] is one of the most important stochastic relaxation algorithms [96]. During the search process, it makes decisions about accepting or rejecting a random generated move based on a random probability related to an annealing temperature. It is able to explore the whole solution space that is independent from the initial starting point. Evolution computation is based on the natural selection principles and the GA is one type of evolution computations [97, 98] that is commonly used. Its search sampling consists of a pool of potential solutions called population that is substantially different from other random search algorithms. It works with an encoding of the solutions and not directly with the solutions. In addition, it uses probabilistic transition rules to evolve from one generation of population to another.

GA is well known for their robustness and are probabilistic techniques that start from an initial population of randomly generated potential solutions to a problem, and gradually evolve towards better solutions through a repetitive application of genetic operators such as selection, crossover and mutation [99]. The evolution process proceed through generations by allowing selected members of the current population, chosen on the basis of some fitness criteria, to combine through a crossover operator to produce offspring thus forming a new population. The evolution process is repeated until certain criteria are met. GA-based scheduling algorithms aim to evolve near-optimal schedules after sufficient number of generations [38]. GAs have been applied successfully to solve scheduling problems in job shop scheduling [40], task scheduling and allocation onto homogeneous multiprocessor systems [41, 92, 100, 101] and task scheduling and matching in heterogeneous computing environments [36, 37, 102, 103].
Genetic Simulated Annealing (GSA)

Shroff et al., used a GSA technique for task scheduling in heterogeneous environments [35]. GSA is a hybrid algorithm that is a combination of GAs and simulated annealing. In this technique, all the standard GA operators such as generation of initial population, evaluation of the fitness, crossover and mutation, etc., were used except the selection operator for selecting the mates for reproduction. For the selection operator, a simulated annealing approach was applied. The fitness function is the reciprocal of the task completion time. If the child's fitness is better than its parent, it is selected. If the fitness is smaller than its parent there is still a chance for its selection as a candidate for reproduction, depending upon a probability value that in turn is a function of current temperature. As current temperature decreases over time according to the schedule, the probability of accepting less fit solutions also decreases, which helps the algorithm to converge. For small-scale problems, the GSA provides optimal solutions.

Task Matching and Scheduling using Genetic Approach

A genetic algorithm for task matching and scheduling when the tasks have interdependencies was presented by Wang et al., [102]. This algorithm will henceforth be referred to as GA. In the GA approach, a chromosome consists of two strings. The first string of a chromosome represents matching, and the second string represents the schedule. The scheduling string of a chromosome is of length \( T \) where each entry is a task index and each string position is in sequential order. The scheduling strings of the initial population are generated as follows. The first scheduling string is a topological sort (a topological sort orders the tasks such that all precedent constraints are satisfied) of the task list. The remaining population is created by a random number of mutations of the first scheduling string. During crossover and mutation, both strings associated with each chromosome are randomly modified but the scheduling strings are constrained by dependencies such that every task \( v_i \) on which task \( v_j \) depends (i.e., the DAG has a directed edge from task \( v_i \) to task \( v_j \)), always precedes task \( v_j \) in every scheduling string. This algorithm provides a robust mechanism for obtaining much better solutions to the dependent scheduling
problem obtained previously. One serious drawback of this algorithm is the very large CPU time needed to obtain solutions for large problems.

**Problem Space Genetic Algorithm (PSGA)**

The Problem Space Genetic Algorithm [37] is intended to increase the efficiency of the GA approach by representing a chromosome in a form that applies more directly to the particular scheduling problem being solved. PSGA defines a chromosome as an array of positive floating point numbers where the array index represents a task and each array element (gene) represents a priority value for the task. The value of each gene in the starting chromosome is the average execution time plus the average communication time taken over all machines and all communication paths. The initial population consists of the starting chromosome plus a set of chromosomes that are random constrained perturbations from the starting chromosome. Once an initial population is created, the PSGA algorithm loops through successive generations until the stopping criteria are satisfied in a manner similar to other GAs by randomized selection, crossover and mutation. The fitness calculation of each randomly generated chromosome for PSGA requires a transformation into a schedule. The EFT heuristic has been used for the transformation in the PSGA algorithm. The EFT heuristic starts with an empty schedule where tasks are then added to the schedule in the order determined by chromosome task priorities and by precedence constraints. That is, tasks are ordered by first creating a task ready-list initially consisting of all the entry nodes. Then the tasks are removed from the ready-list and added to the schedule in order determined by which task in the ready-list has the highest priority. A task is added to the ready-list when all its predecessors have been placed in the schedule. Each time EFT adds a task to the schedule, the task is paired with the machine for which the finish time is minimized. After all tasks have been added to the schedule, the fitness of a chromosome is the *makespan* of the resulting schedule. The PSGA algorithm is more efficient than GA. The PSGA is also better than the GSA because the GSA has used simulated annealing as a selection operator that slows down the convergence of the algorithm [97]. The PSGA embeds a problem-specific heuristic in the genetic algorithm and therefore, it converges in a smaller number of generations with lesser
population size. The major drawbacks of the PSGA algorithm are that it does employ the insertion-based scheduling policy and the task duplication mechanism.

2.5 Task Scheduling Algorithms for Mobile Computing System

Mobile applications are becoming increasingly popular as they provide users with convenience of accessing information and services anytime and anywhere. This includes mobile commerce, geographical location information, battlefield, disaster management, wildfire prevention and peacekeeping operations. The advancement in microprocessors and communication technology increases the computing power of the mobile node. Collaboration among the group of nodes in the mobile computing environment through distributed processing emerges a promising solution to achieve high processing power in resource restricted mobile computing system. Proper task mapping and scheduling plays an essential role in mobile computing to achieve high performance as in DHCS. Though there exists a large number of task scheduling algorithms for homogeneous processors [17, 28, 29, 65, 77, 83, 85] and few for heterogeneous processors [21, 22, 31, 34, 104] they can not be directly used for scheduling DAG-structured application onto MCS. This is because the developed algorithms for the homogeneous and heterogeneous processors assume that the processors are persistently available and they have continuous power supply, whereas in the mobile computing systems node mobility and the available energy (mobile nodes are operated by batteries and hence it has limited power supply) are the two major critical issues [44].

To alleviate the problem of energy limitation, several hardware based techniques have been proposed and used by today's mobile computing devices. Turning off any idle component is one of the most common techniques [105]. Some processors operate at different voltage levels and it gives a chance for voltage reduction which in turn helps in reducing the energy consumed by the CPU [45, 106]. Computation offloading (remote execution) is a software based approach by which a resource-limited computing device defers computation to a nearby more capable computing device (e.g., a stationary server) [46, 47]. Source code optimization and profiling were exploited to minimize energy consumption in
embedded systems [107]. Zhu et al., devised a mechanism to increase reliability and reduce energy consumption of real-time embedded systems by slack time reclamation [108]. Park et al., tried to make a balance between energy efficiency and fairness in multi-resource for multitasks in embedded system [109].

The problem of scheduling independent tasks with objective to minimize the completion time or the energy consumption has been largely explored by the researchers in the past and consequently, a variety of scheduling algorithms have been proposed for scheduling independent tasks in the energy constrained mobile nodes [110-114]. However, the scheduling of DAG-structured application is not explored fully in mobile computing systems. Shivle et al., [48] developed an algorithm for scheduling sub tasks in a heterogeneous ad hoc grid environment. However this algorithm is also not suited for mobile computing system since it does not addresses the node mobility. Therefore, there is a need for energy-efficient task scheduling algorithm for scheduling DAG structured application onto mobile computing system to minimize the schedule length or energy consumption or both. Additionally, the proposed algorithm should provide fault tolerance computing feature by addressing the node mobility.

2.6 Extract of the Literature Survey

Efficient scheduling of the tasks of a parallel program represented by DAG is a challenging issue in DHCS. The task scheduling problem is proven to be NP-Complete and hence heuristics solutions have been proposed to tackle the scheduling problem. Consequently, a variety of heuristics algorithms such as list-scheduling, clustering, task duplication-based scheduling and the task scheduling using genetic approach have been proposed in the literature.

List-scheduling algorithms are widely accepted as a better solution to task scheduling problem. They provide good quality of schedules compared to other category of algorithms with less scheduling overhead. List-scheduling algorithms have been widely explored for the homogeneous systems [51 - 60] whereas only limited work has been carried out for the heterogeneous systems [21, 61 - 63]. The widely referred list-scheduling algorithms for heterogeneous processors are the DLS
There are two important issues in list-scheduling algorithms: (i) ranking the tasks for selection and (ii) selection of the best processor for executing the selected task. Majority of the existing list-scheduling algorithms uses the combinations of two attributes $t$-level and $b$-level for assigning priorities. Some algorithms use the variations in calculating the $t$-level and $b$-level. The existing list-scheduling algorithms does not take into account the priority of parent task, number of dependant tasks and the size of the data to be communicated among the predecessor and the successor tasks for task prioritization.

Further, a majority of the existing algorithms use random selection of tasks when more than one task has the same priority and they will not consider the insertion of a task between two already scheduled tasks on the same processor, even if sufficient time slot is available to execute a task. This may sometimes increase the schedule length of an application. For example, the HEFT algorithm randomly selects one task when there is a tie and consequently may generate different schedules for the same set of input values for different runs. In order to generate consistent schedules, the algorithms should have a well and fine turned priority assignment scheme. The major drawback of the DLS [20] and the HCPT [22] algorithm is that they do not schedule tasks between two previously scheduled tasks. The CPOP algorithm [21] has been developed around the critical path reduction scheme. It allocates all critical tasks onto a single processor in an attempt to minimize the total execution time. However, the selection of a processor in advance for critical path tasks may not work well owing to the involvement of precedence and communication cost constraints in calculating the start time of tasks on different processors. It is quite possible that a critical path task may start later on the chosen processor due to the late data arrival times from its different predecessors and hence, delaying its completion as well, even if the chosen processor is faster [5].

Though the clustering is an efficient way to reduce communication delay in DAGs, it is well suited only for the homogeneous processors. These algorithms basically schedule the tasks to an unlimited number of processors. In reality the existence of such system is highly impossible. Hence in order to match the number of clusters with the number of processors, clustering algorithms use an additional
cluster merging step. Clustering algorithms have less practical applications in DHCS since adapting clustering techniques to the heterogeneous systems is a difficult job. This is because the formation of cluster is generally based on communication cost and in the homogeneous systems, the duration of a communication depends only on the number of data exchanged and the duration of a task execution depends only on the number of operations to be performed whereas, in the heterogeneous system this is no longer true [25]. Indeed, the duration of communication also depends on the speed of the network link taken and the duration of a task execution depends on the processor that will execute this task. Therefore, techniques used for clustering tasks on homogeneous systems are not suited for heterogeneous systems.

Task duplication is an alternative technique to shorten the schedule length. The task duplication minimizes the transfer of data from a predecessor to a successor task reducing the communication cost and the start times of waiting tasks and consequently improves the overall completion time of the program. The well known task duplication-based scheduling algorithms for heterogeneous processors are LDBS1 [30], LDBS2 [30], HCNF [31], TANH [32], HCPF [33] and DCPD [34] algorithms. The similarity between LDBS1 and LDBS2 algorithms is that both the algorithms initially use level sorting process. The differences between these algorithms is that the LDBS1 algorithm equally treats all the tasks in the same level, whereas, the LDBS2 algorithm on the other hand gives higher priority to the tasks that lies in the critical path. The complexities of LDBS1 and LDBS2 algorithms are $O(p^2 e v)$ and $O(p^2 e v)$ respectively, which is quite higher compared to other algorithms. The HCNF algorithm is similar to LDBS2 algorithm and the difference between these two algorithms is that HCNF algorithm schedule the critical path tasks in a level first, then it selects the task with higher execution time next. The HCNF algorithm selects randomly one critical path node (task) when the task graph has multiple critical paths. Random selection does not guarantee similar output (schedule), when the scheduler (algorithm) runs for the next time with the same set of input. The time complexity of the HCNF algorithm is $O(v^2 \log v)$ and it is comparably lesser than LDBS1 and LDBS2 algorithms. The TANH algorithm [32] introduces some conditions to obtain optimal schedule, which are not practical in
heterogeneous environment. Moreover, a majority of the task duplication-based scheduling algorithms focus on the completely connected heterogeneous processors and only a few works have been carried out for arbitrarily connected heterogeneous processors.

From the limited literature survey made for the genetic algorithms, it is observed that task scheduling using genetic approach has been widely explored by the researchers for the homogenous processors whereas only a limited number of studies have been carried out for the heterogeneous systems. Moreover, majority of the GA start by running a heuristic scheduling (list-scheduling) algorithm to create its initial population and they are not considered a task duplication mechanism to reduce the schedule length further in the heterogeneous environment. For example, the PSGA algorithm [37] which is proved to be better than the GSA algorithm [35] and the GA algorithm [102] uses earliest finish time heuristic to generate the initial population and to produce schedules. The major drawback of the PSGA algorithm is that it does not consider the insertion scheduling policy and task duplication. Since task duplication will greatly reduce the schedule length, development of genetic algorithm with the combination of task duplication mechanism has to be explored.

In the recent days, MCS has been widely used in many areas such as battlefield, disaster management and peacekeeping operations. Though it has the potential to execute larger applications, only a little attention is given so far, for the execution of larger applications represented by DAG. However, a significant amount of work has been carried out by the researchers in the past for executing metatasks. The task scheduling algorithms developed for the DHCS can not be applied directly to MCS. This is because the algorithms developed for the DHCS assume that processors are persistently available and they have continuous power supply, wherein in MCS, the node mobility and the energy consumption are critical issues [44, 105]. Recently six scheduling heuristics have been proposed for scheduling the DAG-structured application onto the heterogeneous ad hoc grid computing system for the minimization of energy as well as for the schedule length of the application, but these heuristics do not consider the node mobility. Hence developing a new task scheduling algorithm to minimize the schedule length or energy consumption or
both is very essential for the execution of a larger program in MCS. Additionally the scheduler employing a scheduling algorithm should provide fault tolerant execution environment at the run-time by means of rescheduling the tasks in the energy exhausted node or node leaving the network to other nodes in the MCS.

Hence, to overcome the limitations of the existing task scheduling algorithms and to provide best schedules, the following task scheduling algorithms are to be devised for the DHCS and MCS.

(i) Devising new task scheduling algorithms using simple scheduling list-heuristics to get rid of the drawbacks of the existing algorithms and provide better results than the existing algorithms reported in the thesis.

(ii) Devising new task duplication-based scheduling algorithms for completely connected DHCS.

(iii) Devising a new task duplication-based scheduling algorithm for arbitrarily connected DHCS.

(iv) Devising new task scheduling algorithms using genetic approach with and without task duplication for DHCS.

(v) Formulation of task scheduling problem for the MCS and provide solution to task scheduling problem which minimizes either the schedule length or energy consumption or both.

2.7 Summary

In this chapter, various categories of task scheduling algorithms such as list-scheduling, clustering, task duplication-based scheduling and guided random search-based algorithms developed for distributed systems in the past are recounted and presented. The limitations of the well known and widely referred algorithms in each category are pointed out. Further, the need for improving the algorithms in each class and the development of new task scheduling algorithms for the mobile computing system which minimizes either the schedule length or the energy consumed or both for completing the execution of the application articulated. Finally, this chapter is concluded with an extract of a literature survey.