H2GS;
A Hybrid Heuristic-Genetic Scheduling Algorithm for Static Scheduling of Tasks on Heterogeneous Processor Networks

Mohammad Daoud

A Thesis in the
Department of Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Applied Science at
Concordia University,
Montreal, Quebec, Canada

December 2004

Your file Votre référence
ISBN: 0-494-04364-4
Our file Notre référence
ISBN: 0-494-04364-4

Abstract

H2GS; A Hybrid Heuristic-Genetic Scheduling Algorithm for Static Scheduling of Tasks on Heterogeneous Processor Networks

Mohammad Daoud

The majority of published static scheduling algorithms are only suited to homogeneous processor networks. Little effort has been put into developing scheduling algorithms specifically for heterogeneous processors networks. It is easy to prove, using counterexamples, that the best existing heterogeneous scheduling algorithms [1, 12] generate sub-optimal schedules. Hence, there is much room for the development of better scheduling algorithms for heterogeneous processor networks.

This report presents and tests a novel hybrid scheduling algorithm (H2GS) that utilizes both deterministic and stochastic approaches to the problem of scheduling. H2GS is a two-phase algorithm. The first phase implements a heuristic algorithm (LDCP) that identifies one near-optimal schedule. This schedule is used, together with a small number of other schedules as the initial population of the second customized genetic algorithm (called GATS). The GATS algorithm proceeds to evolve even better schedules. The most important contributions of our research are: (i) the development of a new *hybrid* algorithm, which primes a customized genetic algorithm with a near-optimal schedule produced by a heuristic (LDCP); (ii) The hybrid algorithm succeeds in generating task schedules with completion times that are, on average, 6.2% *shorter* than those produced by the best existing scheduling algorithm, on the same set of test data.

Acknowledgment

First of all, I would like to thank my supervisor Dr. Nawwaf Kharma. It has been a privilege to work under his guidance. I am grateful for his support, encouragement and invaluable suggestions during this work.

I am also infinitely grateful for the support and encouragement offered to me by the HQSF.

Finally, I would like to thank my family for their endless love and support.

Table of Contents

Table of Tables

Table of Figures

Chapter 1: Introduction

A Distributed Heterogeneous Computing System (DHECS) consists of a group of diversely capable machines connected *via* a high speed network, which supports the execution of a set of tasks that have various computation requirements. The performance of DHECS depends on a variety of factors such as machine architecture, network topology and workload characteristics [18].

Scheduling is a key issue in the DHECS operation. It is also an important problem in other domains such as economics, transportation, manufacturing, software development, project management and operational research. In its most general form, scheduling is the process of allocating a set of tasks to a set of available resources, and arranging the execution of these tasks on each resource to obtain optimal performance. Often, the scheduling process must not violate a set of predefined constrains [1].

In task scheduling algorithms for DHECS, the application is represented by a Directed Acyclic Graph (DAG), in which nodes represent tasks and edges represent data dependencies between tasks. The label of a node reflects the computation cost of the associated task, while the label of an edge expresses the time needed for data to pass from one end of the edge to the other. Tasks must be scheduled and assigned to machines in a way that minimizes the total execution time, or the *schedule length*, of the application, and without violating the data dependencies between tasks [1].

Scheduling algorithms are broadly classified into two classes: *static* and *dynamic*. When all information needed for scheduling, such as execution times of tasks, data dependencies and sizes of data transfers are known *in advance*, the scheduling algorithm is described as static. This scheduling process takes place during compile time. On the other hand, in the dynamic model tasks are allocated to processors upon their arrival, and scheduling decisions must be made at run time [3, 5, 21].

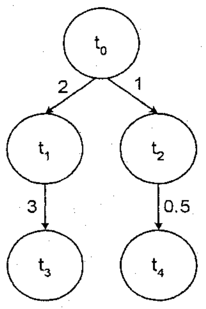
In general, the scheduling problem is NP-complete, meaning that there is no known algorithm that finds the optimal solution in polynomial time. The existing algorithms can deal with various special cases of the problem. Typically, these algorithms take a long time to execute in cases where the size of the problem is large or/and when there are many constrains [3, 4, 5, 11, 14, 16, 22, 23, 25].

Chapter 2: Problem Description

Static task scheduling for a DHECS is the problem of assigning the tasks of a distributed application to a set of diversely capable machines, and specifying the start execution time of the tasks assigned to each processor. This must be done in a way that respects the precedence constrains among tasks. An efficient schedule is one that minimizes the total execution time of the program, or the schedule length of the application. All the information needed to perform scheduling is assumed to be known in advance.

A distributed application is represented by a DAG (T,E,D), where T is a set of *n* tasks, E is a set of *e* edges and D is a *n* x *n* *communication cost* matrix . Each t_i Є T represents a task in the DAG, and each edge $(_{i,j})$ Є E represents a precedence constraint, such that the execution of t_i Є T cannot be started before t_j Є T finishes its execution. Each edge $(_{i,j})$ Є E is associated with a communication cost $d_{i,j}$ Є D that specifies the amount of data that must be transferred from t_i to t_j. However, since the intra-processor bus speed is much higher than the inter-processor network speed, the communication cost between two tasks scheduled on the same processor is taken as zero. If $(_{i,j})$ Є E, then t_i is a *parent* of t_j and t_j is a *child* of t_i, where $\{t_i, t_j\}$ Є T. A task with no parents is called an *entry task*, and a task with no successor is called an *exit task.*

The target DHECS architecture is represented by a set P of *m* heterogeneous processors. The *m* x *n* *computation cost* matrix C stores the execution costs of each task. Each element $c_{i,j}$ Є C represents the estimated execution time of t_j on p_i (Є P). In a homogeneous distributed system, the execution time of a task is the same for all the processors. Further, all processors are assumed to be fully connected. Hence, each processor can communicate directly with any other processor in the network. Moreover, communications between tasks are assumed to be contention free, which allows for deterministic scheduling. Communication between processors occurs *via* independent communication units: this allows for concurrent computation and communication.



(a)

	t_0	T_1	t_2	t_3	t_4
p_0	1.5	4	1.2	6	7
p_1	2	3	2.5	10	8

(b)

Figure 1. (a) A DAG that represents a distributed application (b) a computation cost matrix of a DHECS that is composed of two processors

Figure 1 represents an example of a distributed application and a DHECS that is composed of two processors p_0 and p_1. The distributed application is visualized by the DAG shown on figure 1(a), and the computation cost matrix of the two processors is displayed in figure 1(b). In the table displayed in figure 1(b), a value in the i_{th} column and j_{th} row represents the computation cost of t_i on p_j.

A task can start execution on a processor only when all data from its parents become available to that processor; at that time the task is marked as *ready*. The schedule length is defined as the longest finish time of the DHECS processors, when all the tasks of the application are scheduled.

Chapter 3: Related Work

3.1. Task Scheduling for Distributed Computing System

The problem of task scheduling is known to be NP-complete for arbitrary DAGs [5, 10, 11, 14]. However, because of its key importance, several algorithms were developed to solve it, with various degrees of performance and complexity. Many of these algorithms place restrictions on the structure of DAGs, size of tasks, communication costs and number of processors [5].

The goal of static scheduling is to allocate a set of tasks to a group of processors such that the schedule length is minimized, and the precedence constrains among the tasks are preserved [1, 7]. Static scheduling algorithms can be broadly classified into three main categories: *heuristic algorithms*, *guided random algorithms* and *hybrid algorithms.*

3.1.1. Heuristic scheduling algorithms

Heuristic scheduling algorithms find near-optimal solutions in less than polynomial time. A heuristic algorithm moves from one point in the search space to another, following a particular rule. Such algorithms though efficient, search some paths in the search space, and ignore others [3]. Heuristic scheduling algorithms can be subdivided into three subgroups: *list-based heuristics, clustering heuristics, and duplication heuristics* [1, 3, 7].

In list-based scheduling heuristics, each task is assigned a given priority. The tasks are inserted in a list of waiting tasks, such that tasks with higher priority are placed before those with lower priorities. Three steps are then repeated until all the tasks in the list are scheduled: task selection, processor selection and status update. The highest-priority ready task is removed from the list and selected for scheduling during the first step. Next, the selected task is assigned to the processor that minimizes a predefined cost criterion. In homogeneous computing environment, the *earliest start time* criterion, in which the task is assigned to the processor that minimizes its start execution time, can be employed. Finally, the status of the system is updated. At the end of this process, a valid schedule is obtained [1, 3, 7, 16].

List-based scheduling heuristics differ mainly in the way used to assign priorities to tasks. These heuristics try to find critical tasks and assign them higher priorities. The most-critical ready task is first selected and assigned to the most suitable processor. Many list-based scheduling heuristics work only if a predefined simplifying assumption is maintained. Some assumptions can be justified in particular contexts, but others cannot be satisfied in real-time applications. Homogeneous processors, an unlimited number of processors and no precedence constraints among tasks are examples of these simplifying assumptions [8]. Examples of list-based algorithms are: the Modified Critical Path (MCP) algorithm [20], the Mapping Heuristic (MH) algorithm [13] and the Dynamic Critical Path (DCP) algorithm [7].

Clustering heuristics trade off inter-processor communications cost with parallelization, by allocating heavily communicating tasks to the same processor. In this type of heuristics, task clustering is performed prior to the actual scheduling process. During the clustering phase, a task graph is clustered under the assumption of an unbound number of processors. Tasks that are assigned to the same cluster are executed on the same processor. Scheduling starts by verifying the number of resulting clusters. If the number of clusters is greater than the number of available processors, clusters are merged so that the number of the remaining clusters equals the number of processors. Next, clusters are mapped to the available processors, and local execution of tasks within each processor is determined [1, 4, 15].

During the clustering phase, clustering algorithms are not restricted by the constraint of a limited number of processors, which generally holds in the operation of list-based algorithms. Clustering algorithms can employ low-complexity load-balancing to map clusters to processors. Hence, the resulting complexity of clustering algorithms tends to be lower than that of list-based algorithms [4]. Examples of clustering algorithms are Dominant Sequence Clustering (DSC) [18], Edge-Zeroing (EZ) [19] and Mobility Directed (MD) [20].

Finally, duplication scheduling algorithms try to execute key tasks redundantly to reduce inter-processors communications, and hence reduce the waiting time of dependent nodes. Various algorithms try to blend duplication heuristics with both list-based and clustering heuristics, to improve their performance. However, this improvement in performance comes at the cost of increase of complexity [1, 3, 4, 6, 16]. Some duplication algorithms have a problem of significant growth in required processors as DAG size increases. Other algorithms overcome this problem by duplicating takes selectively instead of duplicating all possible ancestors of a given node. Examples of this type of heuristics are: Selective Duplication (SD) algorithm [4], Critical Path Fast Duplication (CPFD) algorithm [16], Button up Top down Duplication Heuristic (BTDH) algorithm [17] and Duplication Scheduling Heuristic (DSH) algorithm [24].

3.1.2. Guided random scheduling algorithms

Guided Random search techniques use random walks combined with guiding information to investigate the search space of the problem. Such techniques exploit the knowledge gained from previous search results to guide the search process. Among the various Guided Random techniques, Genetic Algorithms (GAs) are the most widely used for the task scheduling problem [1].

A GA is a stochastic search technique that simulates the mechanisms of natural evolution. It operates on a population of solutions, or individuals, and employs a set of genetic operators. These operators operate on the individuals in the population to evolve new solutions. The genetic search process begins by initializing a population of individuals. Each individual corresponds to a particular candidate solution for the problem. During the evolution process, a new population of individuals is created through mating and mutation. However, according to evolutionary principles, the fittest individuals are more able to survive and generate offspring. At each generation, the evolutionary process goes through a simple set of stages: evaluating each individual, selecting individuals for the mating pool and applying genetic reproduction operators to create a new population [3, 26, 27].

The typical heuristic scheduling techniques move from one point in the search space to another using a particular transition rule. In multimodal scheduling problems, this point-to-point transition may mislead the search process. GA-based scheduling techniques overcome this problem by working on a population of points in parallel. Hence this minimizes the probability of converging to a local optimum [3]. On the other hand, a poor representation of the scheduling problem may lead to difficulties in finding good solutions within a reasonable period of time.

In contrast to the heuristic scheduling techniques which require direct information about the DAG and processors to decide the next scheduling step, GA-based scheduling techniques operate on individuals that encode possible candidate schedules. They care about the structure of each individual not about the actual encoded schedule. However, to guide the search process, a GA needs to know how "good" the encoded schedule is. The technique developed by A. Zomaya *et al.* [3] is an example of this class of scheduling techniques.

3.1.3. Hybrid scheduling algorithms

Hybrid scheduling algorithms combine both heuristic scheduling algorithms and GAs. The Genetic List Scheduling (GLS) Algorithm [28] is an example of this class of algorithms. In GLS, tasks are scheduled on the available *modules* and communications are mapped to the available buses. The *resource* set is composed of the available modules and buses, while the *user* set is composed of the tasks and communications of an application. A GA is used first to evolve a set of priorities. These evolved priorities are used by a list-based scheduling algorithm to generate a schedule. Each individual in the GA population encodes two sets of genes: user priorities and user-resource priorities. The user priorities genes encode priorities of all users, and these are used to select users for scheduling. Once the selected user is assigned to a resource, the user-resource priorities are considered.

3.2. Task Scheduling for DHECS

Most static task scheduling algorithms described in the literature assume a homogeneous target system; hence static task scheduling for DHECS is relatively unexplored. In addition to the tradeoff between the gained parallelism and the resulting inter-processors communications, a DHECS scheduling algorithm has to consider the various execution times of the same task on different processors. In this section two scheduling algorithms that support DHECS, the Dynamic Level Scheduling (DLS) algorithm [12] and the Heterogeneous Earliest Finish Time (HEFT) algorithm [1], are presented.

3.2.1. Dynamic level scheduling (DLS) algorithm

The DLS algorithm uses a quantity called *Dynamic Level* DL(n_i,p_j), which is the difference between the maximum sum of computational costs from task n_i to an exit task, and the earliest start execution time of n_i on processor p_j. In this algorithm, the earliest start execution time of n_i on p_j is defined as the maximum of the *ready time* of n_i on p_j or, in other words, the time when all input data of n_i become available to p_j, and the time when p_j finishes the execution of its already scheduled tasks. The DLS algorithm does not schedule tasks between two previously scheduled tasks.

The DL values change during the scheduling process. At each scheduling step, the algorithm evaluates the DL values for all combinations of ready nodes and available processors. The ready node and available processor pair that has the highest DL value is chosen for scheduling.

To accommodate DHECS, the computational cost of a task is taken as the median of the execution times of that task over all processors. Moreover, a new quantity is added to the equation of DL(n_i,p_j) to account for the various execution times of the same task on different processors. This quantity is the difference between the median execution time of n_i over all processors, and its execution time on p_j. The time complexity of the general DLS is $O(m \times n^3)$, where m is the number of processors and n is the number of tasks.

3.2.2. Heterogeneous earliest finish time (HEFT) algorithm

The HEFT algorithm starts by setting the computational costs of tasks and communicational costs of edges to their mean values. Each task is assigned a value called *upward rank*. In this algorithm, the upward rank of a task n_i is the largest sum of mean computational costs and mean communicational costs along any directed path from n_i to an exit task. A task list is then generated by sorting all tasks by decreasing order of their upward rank; ties are decided 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 n_i, the insertion-based scheduling policy considers all possible *idle time slots* on p_j to find a time slot that is capable of the execution time of n_i. This must be done without violating the precedence constrains 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 were successively scheduled on p_j. The search starts from a time equal to the ready time of n_i on p_j, and proceeds until it finds the first idle time slot with a sufficient large time space to accommodate the computational cost of of n_i on p_j. If no such sufficient 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(m x e) where m is the number of processors, and e is the number of edges. The time complexity for dense DAGs, in which the number of edges is proportional to n^2 (where n is the number of tasks), is O(n^2 x m).

Chapter 4: The Proposed Algorithm

The *Hybrid Heuristic-Genetic Scheduling* (H2GS) algorithm is a hybrid scheduling algorithm that combines two algorithms to produce a near optimal schedule. First, the H2GS algorithm runs a heuristic scheduling algorithm, called the *Longest Dynamic Critical Path* (LDCP) algorithm, to generate a task schedule. The schedule generated by the LDCP algorithm is located at an approximate area near the optimal solution, in the search space. Next, a Genetic Algorithm, called Genetic Algorithm for Task Scheduling (GATS), improves the LDCP-generated schedule by searching around it, to discover better optimal (or near-optimal) schedules.

As in the algorithms mentioned earlier, our approach has a set of assumptions. The processor network is assumed to be fully connected, and each processor has dedicated communication hardware, so that communication and computation can take place simultaneously. Moreover, computation costs of tasks are assumed to be linear. In other words, if the computation cost of task t_i on processor p_j is higher than that on processor p_k, then the computation costs of any task on p_i is higher than or equal to that on processor p_k.

4.1. The Longest Dynamic Critical Path (LDCP) Algorithm

This section is subdivided into two subsections. In the first subsection, the problem of identifying priorities of tasks in DHECS is introduced. Next an effective attribute, called

LDCP, which effectively identifies the most important tasks in a DHECS, is explained. In the second subsection, the Longest Dynamic Critical Path (LDCP) algorithm is described.

4.1.1. Task Priorities in DHECS

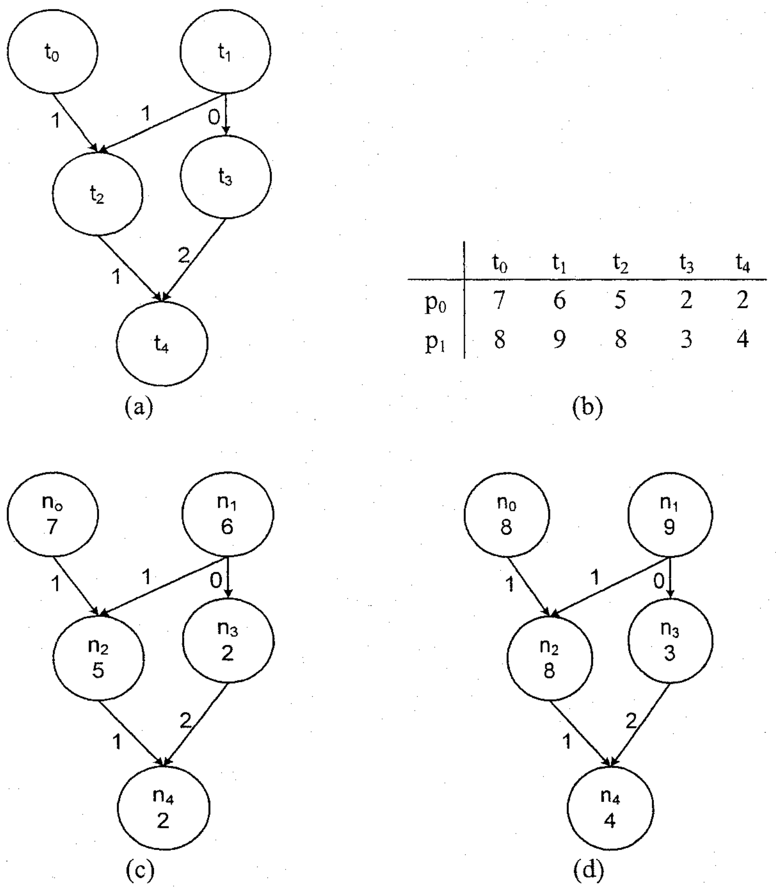
The performance of a list scheduling algorithm depends highly on the method used to assign priorities to tasks. At each scheduling step, a task must be assigned a high priority if the selection of this task for scheduling during the current scheduling step leads ultimately to a shorter schedule length.

For a homogeneous computing environment, the *critical path* (CP) attribute of a DAG provides an effective method for assigning priorities to tasks. For a given DAG, the CP is defined as the path from an entry task to an exit task that has the greatest sum of computation and communication costs. The sum of computation costs of the tasks located on the CP determines the lower bound of the final schedule length. Hence, an efficient scheduling algorithm requires proper scheduling of the tasks located on the CP. On the other hand, when two tasks are scheduled on the same processor, the communication cost between them is zero. Consequently, the CP changes dynamically during the scheduling process. A task located on the CP at a particular scheduling step may not be located on the CP in other steps. To overcome the dynamic behavior of CP, Kwok *et al.* [7] used an efficient way to select tasks for scheduling. In their algorithm, the *Dynamic Critical Path* (DCP) is used. The DCP is simply a CP that considers the cancellation of communication costs among tasks scheduled in the same processor. The DCP identifies the most important task for scheduling at each scheduling step.

In DHECS, the various computation costs of the same task on different processors present us with a problem: the DCP computed using the computation costs of tasks on a particular processor may differ from the DCP computed using the computation costs of tasks on another processor.

For example, consider the application DAG and the computation costs array in figures 2.a and 2.b. Here, the task graph in figure 2.c is constructed using the computation costs of tasks on processor p_0, while the task graph in figure 2.d is constructed using the computation costs of tasks on processor p_1. In other words, processors p_0 and p_1 are used as reference processors to construct the task graphs in figures 2.c and 2.d respectively. At the beginning of the scheduling process, the DCP extracted form the task graph in figure 2.c is composed of tasks t_0, t_2, t_4, while the DCP extracted form the task graph in figure 2.d is composed of tasks t_1, t_2, t_4. Both DCPs share two tasks, t_2 and t_4, and differ by one task. Therefore, a complete scheduling algorithm has to consider the various DCPs that are computed using the computation costs of tasks on all processors.

The problem of identifying a single DCP can be addressed by employing one reference DCP. The reference DCP can be computed using the computation costs of tasks on a reference processor. At each scheduling step, the reference DCP is used to select a task for scheduling. However, an inconsistency problem will result in the processor selection phase. The selected task, which is selected using the computation costs of tasks on a reference processor, is assigned to a processor using its actual computation costs over all processors. Another way to address the problem of variable DCPs is to use the mean value of computation costs of a task over all processors to compute a reference DCP. However, the resulting reference DCP will not accurately identify the most important tasks.



	t_0	t_1	t_2	t_3	t_4
p_0	7	6	5	2	2
p_1	8	9	8	3	4

Figure 2. (a) DAG of a distributed application (b) computation costs array (c) task graph constructed based on the computation costs of tasks on processor p_0 (d) task graph constructed based on the computation costs of tasks on processor p_1

One important attribute that can be used to compute priorities of tasks in DHECS precisely is the *Longest Dynamic Critical Path* (LDCP). The LDCP is explained in Definition 1.

Definition 1. Given a DAG with n tasks and e edges, and a DHECS with *m* heterogeneous processors, the Longest Dynamic Critical Path (LDCP) during a particular scheduling step is a path of tasks and edges from an entry task to an exit task that has the largest sum of communication costs of edges and computation costs of tasks over all processors. In addition, the communication costs between tasks scheduled on the same processor are assumed zero and the resulting execution constrains are preserved.

For example, Consider again the application DAG and the computation costs array in figures 2.a and 2.b. At the beginning of the scheduling process, the DCP computed using the computation costs of tasks on processor p_0 is composed from the tasks t_0, t_2, t_4 and has a length of 16. On the other hand, the DCP computed using the computation costs of tasks on processor p_1 is composed from the tasks t_1, t_2, t_4 and has a length of 23. Hence, at the beginning of scheduling process the LDCP is composed from the tasks t_1, t_2, t_4 and has a length of 23.

The LDCP identifies a set of tasks and edges that play an important role in determining a *provisional* schedule length. Hence, to generate an efficient schedule, tasks on the LDCP must be assigned relatively high priorities.

4.1.2. The LDCP Algorithm

The LDCP algorithm is a generic list-based scheduling algorithm for a finite number of heterogeneous processors. In the LDCP algorithm, each scheduling step consists of three phases: task selection, processor selection and status update. In the task selection phase, a task is selected for scheduling, using the LDCP attribute. Then, during processor selection, the selected task is assigned to the processor that minimizes its finish time. Finally, the system parameters are updated, concluding the 3-phase algorithm.

4.1.2.1. Task selection phase

As described before, the LDCP identifies a set of tasks that play an important role in determining the *provisional* schedule length. To compute the LDCP, a *Directed Acyclic Graph that Corresponds to a Processor* (DAGP), which is explained in definition 2, is constructed for each processor in the system. These DAGPs are constructed at the beginning of the scheduling process.

> **Definition 2.** Given a DAG with n tasks and e edges and a DHECS with *m* heterogeneous processors $\{p_0, p_1, \ldots, p_{m-1}\}$, the Directed Acyclic Graph that Corresponds to processor p_i, called $DAGP_i$, is the task graph constructed using the structure of the DAG, with sizes of tasks set to their computation costs at processor p_i.

Through the course of this report, the task t_i is used to refer to the i_{th} task in the application DAG. The node n_i in $DAGP_j$ corresponds to task t_i in the application DAG with its size set to the computation cost of t_i on processor p_j. Hence, node n_i on $DAGP_j$ identifies the task t_i on the application DAG.

For example, $DAGP_0$ and $DAGP_1$ for the application DAG and the heterogeneous system described in figures 2.a and 2.b, are shown in figure 2.c and 2.d respectively. Node n_2 in $DAGP_0$ corresponds to task t_2 in the application DAG with its size set to the computation cost of task t_2 on processor p_0. Node n_2 in DAGP$_0$ identifies task t_2 in the application DAG.

For each DAGP, all nodes are assigned *upward rank* (URank) values to reflect their priority within that DAGP. The upward rank is explained in definition 3.

Definition 3. The upward rank of a node n_i on a task graph DAGP$_j$, denoted as URank$_j$(n_i), is recursively defined as:

$$URank_j(n_i) = w_j(n_i) + \max_{n_k \in succ(n_i)} \{c_j(n_i, n_k) + URank_j(n_k)\} \qquad (1)$$

where succ$_j$(n_i) is the set of immediate successors of n_i on DAGP$_j$; $w_j(n_i)$ is the size of n_i in DAGP$_j$; $c_j(n_i,n_k)$ is the communication cost between n_i and n_k in DAGP$_j$.

The immediate successor of n_i in DAGP$_j$, which satisfies the maximization term in equation 1, is called the upward rank associated successor (URAS) of node n_i.

The *URank* values of the nodes in a given DAGP are computed recursively by traversing that DAGP upward starting from exit nodes to entry nodes. The *URank* value of an exit node is equal to its size.

For example, consider again DAGP$_0$ shown in figure 2.c. The *URank* value of node n_4, which is an exit node, is equal to its size, which is 2. The *URank* value of node n_1 is equal to 15, which can be computed by adding the *URank* value of node n_2 (8) to the communication cost between node n_1 and node n_2 (1) and the size of node n_1 (6). Node n_2 is the URAS of node n_1.

Since we recursively compute the URank values of the nodes in a given DAGP upward starting for the exit nodes, the node or nodes, with the highest URank value will always be entry node or nodes.

Theorem 1. *The node that has the highest URank value over all DAGPs identifies the entry task of the LDCP.*

Proof. *If node n_i, which is located on DAGP$_j$ has the highest URank over all DAGPs, then the length of the LDCP is equal to the URank value of node n_i. Hence, node n_i identifies the first task on the LDCP.*

Theorem 2. *If the tasks on an LDCP are being identified recursively downward starting from the entry node, and node n_i in DAGP$_i$ is used to identify the last identified task on the LDCP, then the URAS of node n_i on DAGP$_i$ identifies the next task on the LDCP.*

Proof. *If node n_i on DAGP$_i$ identifies task t_a on the LDCP and its URAS is node n_k, then the URank value of node n_k is equal to the length of the portion of the LDCP that extends between the exit task of the LDCP and the task located immediately after task t_a on the LDCP, Hence, n_j identifies the next task after t_a on the LDCP.*

The entry task of a LDCP is determined by locating node n_i on the corresponding DAGP with the highest *URank* value over all DAGPs. The remaining tasks on the LDCP can be identified by recursively traversing the DAGP that contains node n_i. The process of transversal starts from node n_i and moves downward. During the process of transversal, the nodes that identify the tasks on the LDCP are located using theorem 2.

For example, refer again to the application DAG and the computation costs array in figures 2.a and 2.b. At the beginning of the scheduling process, node n_1 in $DAGP_1$ has an *URank* value of 23, which is the highest over all nodes in $DAGP_0$ and $DAGP_1$. Hence, t_1 is the entry task of the LDCP. In $DAGP_1$, node n_2 is the *URAS* of node n_1 so task t_2 is the second node in the LDCP. In the same way, task t_4 can be identified as the last task in the LDCP.

Definition 4. During a particular scheduling step, let the set of nodes N be used to identify the tasks on the LDCP. The unscheduled node in N with the highest URank value is defined as the *key node*. The DAGP in which the nodes in N are located is called the *key DAGP*.

Definition 5. During a particular scheduling step if the key node has unscheduled parents, then the unscheduled predecessor of the key node with the highest URank is defined as the *parent key node*.

At each scheduling step, the key node and the parent key node are used to select a task for scheduling. Ties are broken by selecting the task with the highest number of output edges first; if more than one task exists, the tie is broken on a random basis.

Theorem 3. *Let a task t_i be assigned to a DHECS with m processors in which the q fastest processors have assigned tasks, while the m-q slowest processors have* *no assigned tasks. If $q<m$, then the finish execution time of t_i is minimized when t_i is assigned to one of the $(q+1)$ fastest processors.*

Proof. *Let EFT1 be the earliest finish time of t_i on the $(q+2)_{th}$ fastest processor, and Let EFT2 be minimum earliest finish time of t_i on the $q+1$ fastest processors. Then, EFT1 is greater or equal to EFT2.*

Based on theorem 3, a set of processors and its corresponding set of DAGPs, called *active- processors* and *active-DAGPs*, are defined and considered at each scheduling step. At the beginning of the scheduling process, the active-processors set is composed of the fastest processor in the system, and is considered to select and schedule the first task for scheduling. In the consequent steps, each time a task is assigned to an empty processor, the fastest empty processor is added to the active-processors set. When all the available processors in the system are included in the active-processors set, no more processors are added to the active-processors set. The active-DAGPs set is composed of the DAGPs that correspond to the processors in the active-processor set. Each time a processor is added to the active-processors set, its corresponding DAGP is added to the active-DAGPs set.

For example, refer again to the application DAG and the computation costs array in figures 2.a and 2.b. At the beginning of scheduling process, the active-processors set and active-DAGPs set are composed from the P_0 and $DAGP_0$ respectively. Hence, $DAGP_1$ will not be considered to select the first task for scheduling. Nodes n_0, n_2 and n_4 in $DAGP_0$ are used to identify the LDCP. Hence, the set N is composed from nodes n_0, n_2 and n_4. Since node n_0 is unscheduled yet and it has the highest *URank* value over all nodes in *N*, it is marked as the *key node*. Node n_0 has no unscheduled parents, hence task t_0 will be selected for scheduling.

The procedure for selecting a task for scheduling is outlined in figure 3.

```
Select_Task()
start:

 1. if active-processors set has no empty processors then
 2.     if more empty processors are available then
 3.         Add the fastest empty processor to the active-processors set
 4.         Add DAGP that corresponds to the fastest empty processor to the
            active_DAGPs set
 5.     end if
 6. end if
 7. Find the key DAGP in the active-DAGPs set
 8. Find the key node in the key DAGP
 9. if the key node has no unscheduled parents then
10.     Identify the selected task using the key node
11. else
12.     Find the parent key node
13.     Identify selected task using the parent key node
14. end if
15. return selected task

end:
```

Figure 3. The Select_Task procedure

4.1.2.2. Processor selection phase

In this phase, the selected task is assigned to a processor in the active-processors set that minimizes its finish execution time. The insertion-based scheduling policy, which is described before, is used to select an appropriate processor to execute the selected task at the most suitable time. The procedure for selecting a processor to execute the selected task is presented in figure 4.

```
Assign_Task_To_Processor($t_i$)
start:

    1. for each processor $p_j$ in the active-processors set do
    2.     Compute the FET of $t_i$ on $p_j$ using the insertion based policy
    3. Assign $t_i$ to the processor $p_j$ that minimizes its FET
    4. return $p_j$

end:
```

Figure 4. The Assign_Task_T0_Processor procedure

4.1.2.3 Status update phase

When a task is scheduled on a processor, the state of the system must be updated to reflect the new changes. The scheduling of a task t_i on a processor p_j means that the computation cost of t_i is no more unknown. Hence the sizes of the nodes that identify t_i must be set to the computation cost of t_i on p_j on all DAGPs. Moreover, the communication costs between tasks scheduled on the same processor become zero. Thus, the communication costs of all edges, which extend between a node that identifies t_i and a node that identifies one of its parents that is scheduled on p_j, are set to zero.

For example, refer again to the application DAG and the computation costs array in figures 2.a and 2.b. At the beginning of scheduling process, task t_0 is scheduled on processor p_0, both $DAGP_0$ and $DAGP_1$ must be updated to reflect the actual size of task

t_0. On the other hand, since no tasks are previously scheduled on processor p_0, there is no need to update the communication costs of the edges.

The insertion of a task t_i into a processor p_j will result in new execution constrains. The execution constrains and the order in which the previously scheduled tasks had been selected for scheduling, must be considered during the computation of priorities of tasks. These constrains are shown on all DAGPs by adding a zero-cost edge from the node that identifies the last scheduled task on p_j to the node that identifies t_i. Moreover, when a task is assigned to processor p_j, temporary zero-cost edges are added to $DAGP_j$ from the node that identifies the last task scheduled on p_j (task t_k) to the ready nodes on $DAGP_j$ that do not communicate with t_k. This must be done after removing the pervious temporary zero-cost edges from $DAGP_j$. The temporary zero-cost edges are considered when calculating the *URank* value of the nodes.

> **Definition 5.** A dummy node on $DAGP_j$ is a node that does not have a non-dummy successor node, and defines a task that is scheduled on a processor other than p_j.

For a particular processor p_j, a dummy node on $DAGP_j$ is already scheduled at another processor; moreover, any/all of its successors are already scheduled at other processors. Since the URank values on $DAGP_j$ are used to identify the LDCP based on the computation costs of tasks at p_j. The dummy nodes must not be considered during the computation of URank values on $DAGP_j$. Hence, by the end of each scheduling process, the dummy nodes on all DAGPs are identified and their computation costs are set to zero.

The insertion of a task t_i into a processor p_j will affect the *URank* values of the nodes that identify t_i and the nodes that identify the previously scheduled tasks. Hence, by the end of each scheduling step the *URank* values of the nodes that identify the currently scheduled task and the previously scheduled tasks are updated over all DAGPs.

The procedure for updating the status of the system after scheduling the task t_i on processor p_j, is formalized in figure 5.

```
Update_Status(t_i, p_j)
Start:

    1. Update the size of the node that identifies task t_i on all DAGPs.
    2. Update the communication costs on all DAGPs.
    3. Add the execution constrains, which results from assigning t_i to p_j, to all
       DAGPs.
    4. Remove the temporary zero-cost edges that were inserted in the previous
       scheduling step, from all DAGPs.
    5. Add the temporary zero-cost edges, which results from scheduling t_i in p_j, to
       all DAGPs.
    6. Identify and remove the dummy nodes from all DAGPs.
    7. Update the URank of the nodes that identify the currently scheduled task and
       the previously scheduled tasks on all DAGPs.

end:
```

Figure 5. The Update_Status procedure

In our reference example, when task t_0 is scheduled on processor p_0, task t_0 is the first scheduled task in the system. Hence, there is no need to insert any execution constrains to $DAGP_0$ nor $DAGP_1$. However, a temporary zero-cost edge must be added to $DAGP_0$ from node n_0 to node n_1. In both $DAGP_0$ and $DAGP_1$, no nodes are marked as dummy.

DAGP$_0$ and DAGP$_1$ after scheduling task t_0 on processor p_0 are shown in figure 6.a and 6.b respectively. The temporary zero-cost edges are represented by dotted lines.

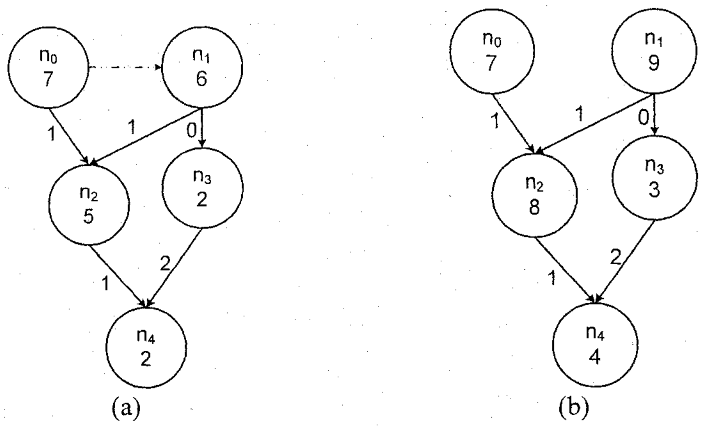


Figure 6. (a) DAGP$_0$ (b) DAGP$_1$

4.1.2.4. The Proposed LDCP Algorithm

Using the procedures discussed above, the proposed LDCP algorithm is formalized in figure 7. The LDCP algorithm has a general time complexity of O(m x n^3) where m is the number of processors, and n is the number of tasks.

```
LDCP_Algorithm()
start:

  1. Using the application DAG and the computation costs array, construct DAGPs
     for all processor in the system.
  2. while not all nodes are scheduled do
  3.     Call Select_Task() and record the selected task in selected_task.
  4.     Call Assign_Task_To_Processor(selected_task) and record the selected
         processor in selected_processor.
  5.     Call Update_Status(selected_task, selected_processor).
  6. end while

end:
```

Figure 7. The LDCP algorithm

4.2. Genetic Algorithm for Task Scheduling (GATS)

In this section, the Genetic Algorithm for Task Scheduling (GATS) is introduced. The GATS accepts the output of the LDCP algorithm as its own input. The schedule that is produced by the LDCP algorithm is located at an approximate area in the search space around the optimal schedule. The GATS searches around that approximate area to improve the schedule. GATS can work on top of any other heuristic scheduling algorithm to optimize below. The operation of the GATS algorithm is formalized in figure 8.

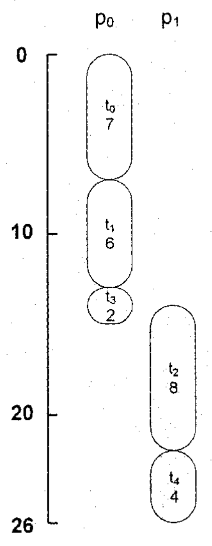
```
GATS()
start:
  1. Create the initial population using the schedule generated by the LDCP
     algorithm.
  1. while termination criteria is not met do
  2.     Evaluate the fitness of the chromosomes in the population.
  3.     Copy the best 10% of the population chromosomes to the elitism set.
  4.     Select chromosomes from the population to the mating pool.
  5.     Apply the swap crossover operator on the chromosomes in the mating pool.
  6.     Apply the swap mutation operator on the chromosomes in the mating pool.
  7.     Combine the chromosomes in the mating pool and elitism set to generate the
         new population.
  8. end while
end:
```

Figure 8. The GATS algorithm

4.2.1. Schedule Encoding

The structure of task scheduling problem, which is composed from the application DAG that must be allocated to a set of processors, makes the using of binary encoding inefficient. In GATS, a 2-dimensional string chromosome is used to encode a schedule. The 2-dimensional chromosome is composed of a number of substrings; each substring represents a processor in the system. Each substring represents the tasks assigned to an individual processor. For example, consider the application DAG and computation cost matrix shown in figure 2, the schedule shown in figure 9.a is encoded in the chromosome shown figure 9.b.

Each time a chromosome is decoded, the application DAG is used to preserve the precedence constrains among the tasks of the encoded schedule. To decode a chromosome, the application DAG is traversed downward, starting from the entry task. At each decoding step, all ready tasks on the application DAG are defined and assigned to processors according to their location among the substrings of the chromosomes. The insertion-based scheduling policy is used to compute the start execution time of the assigned tasks. If two, or more, ready tasks are located on the same substring, only the first ready task on that substring is selected and assigned to a processor.



(a)

p0	t_0	t_1	t_3
p1	t_2	t_4	

(b)

p0	t_3	t_1	t_0
p1	t_4	t_2	

(c)

Figure 9. (a) A scheduling example (b) + (c) chromosome examples

Using this decoding technique, the information encoded within the 2-dimensional chromosome will always lead to a feasible schedule. Hence, this representation provides for a dense search space, and hence an efficient search process.

The same schedule can be encoded using more than one chromosome. For example, both chromosomes shown in figures 9.b and 9.c decode the same schedule shown in figure 9.a. The directly encoded chromosome, in which the order of tasks in the encoding chromosome is identical to the order of tasks in the encoded schedule, is used to create the initial population.

4.2.2. Initialization

The first step in GATS is the creation of the initial population. GATS uses the LDCP algorithm to create the first chromosome in the initial population. The schedule generated by the LDCP is encoded, and the resulting chromosome is inserted into the initial population. This guarantees a more directed search for GATS.

In addition, for each individual processor in the system, exactly one chromosome is created and inserted into the initial population, such that all tasks in the encoded schedule are randomly allocated to that processor. The rest of the chromosomes in the initial population are created randomly.

The population size is determined before running GATS, and does not change during the run. The minimum population size is equal to the total number of processors in the system plus 1. The GATS used in the results section has a population size of 50.

4.2.3. Fitness Evaluation

The calculation of fitness of a chromosome is quite simple. First, each chromosome in the population is decoded. Next, the fitness value of a chromosome equals to the inverse of the length of its decoded schedule. The fact that fitness evaluation is simple significantly enhances the performance of the GATS algorithm. This leads as we shall see in the results section, to fast computation of near optimal schedules.

4.2.4. Selection and Elitism

Copies of the best 10% of the chromosomes in the population are copied without change to the elitism set. This mechanism guarantees that the best chromosomes are never destroyed by either the crossover or the mutation operators.

A rank-based selection mechanism with replacement is used to select chromosomes to the mating pool. This rank-based selection assists in preventing premature convergence of the population towards a sub-optimal point on the fitness surface. The size of the mating pool equals to 90% of the population size.

4.2.5. Swap Crossover

Swap crossover works on the chromosomes in the mating pool; it works on two chromosomes to produce two offspring chromosomes, each with genetic material from both parents. For each parent, swap crossover randomly chooses one substring to apply the crossover on it. Next on each chosen substring, swap crossover locates two crossover points.

For each parent, swap crossover creates an intermediate modified chromosome copy called the *mask chromosome*. In the mask chromosome, the task-slice located between the two crossover points is deleted and replaced by a task-slice parent, which is delineated by the two crossover points, from the other parent. If the inserted task-slice contains a task that does not exist on the deleted task-slice, this task is marked as *don't move* (DM).

For example, consider the two chromosomes and two crossover points CP1, CP2, shown in figures 10.a and 10.b. The mask chromosomes that correspond to the parent chromosomes in figure 10.a and 10.b are shown in figures 10.c and 10.d respectively. DM tasks in the mask chromosomes are indicated by the (*) sign.

To create the new chromosome, both the mask chromosome and its parent are traversed string by string, starting form the first task at each substring. If the current task on the parent is identical to the current task on the mask chromosome, then it is moved to the offspring chromosome. However, if this task is marked as DM on the mask chromosome, it is deleted from both the parent and mask chromosomes. If the current task on the parent is different from the current task on the mask chromosome, only the current task on the parent is moved to the offspring chromosome. Finally, before traversing to the next substring, all the tasks left on the current substring of the mask chromosomes are moved, as is, to the offspring chromosome. Figure 10.e shows the offspring chromosome created by the parent chromosome in figure 10.a and its mask chromosome in figure 10.c, while the offspring chromosome shown in figure 10.f is created by the parent chromosome in figure 10.b and its mask chromosome in figure 10.d.

In GATS, the swap crossover is applied with crossover probability of 0.7.

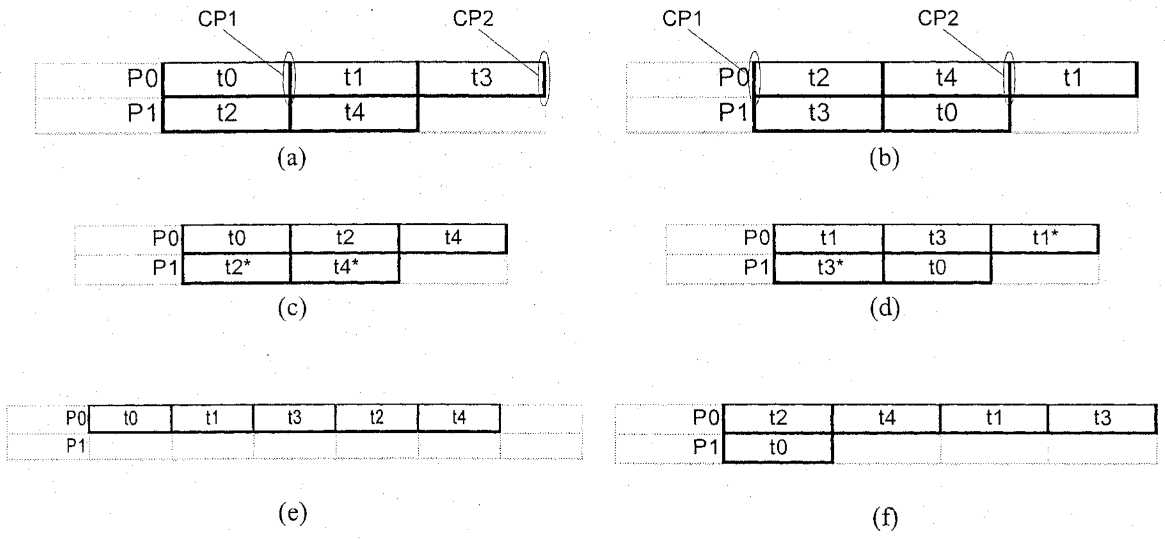


Figure 10. The Swap crossover operator

4.2.6. Swap Mutation

Swap mutation works on the chromosomes in the mating pool to preserve the diversity of the population. It randomly selects two tasks from a chromosome and swaps them. Swap mutation is applied with probability of 0.5.

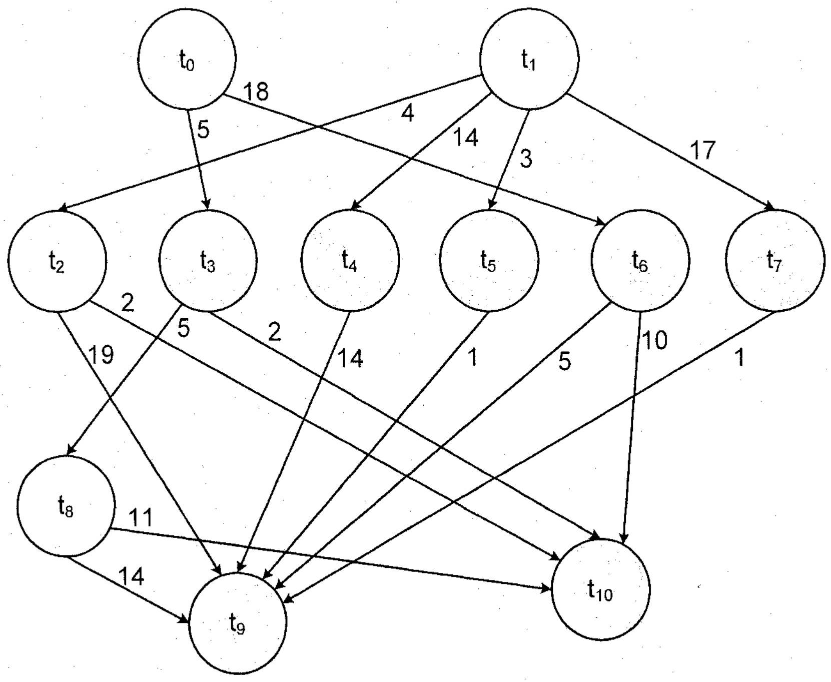
After applying the swap crossover and swap mutation operators, the chromosomes in the mating pool are combined with the chromosomes in the elitism set to create the new population.

4.2.7 Termination Criterion

GATS runs for a predetermined number of generations. The GATS used in the results section runs for 20 generations.

Chapter 5: Scheduling Example

In this section, a DHECS, shown in figures 11.a and 11.b, is used to illustrate the effectiveness of the proposed scheduling algorithm. In this scheduling example, the schedule generated by H2GS algorithm is compared to two other algorithms.



(a)

Task	p_0	p_1
t_0	4	6
t_1	15	22.5
t_2	4	6
t_3	13	19.5
t_4	10	15
t_5	7	10.5
t_6	8	12
t_7	4	6
t_8	12	18
t_9	6	9
t_{10}	9	13.5

(b)

Figure 11 (a) A DAG of a distributed application (b) a computation cost matrix of a DHECS

The schedule generated by the DLS algorithm is shown in figure 12.a. For simplicity, communication edges are not shown in the schedule. The task-processor pair that is selected for scheduling at each scheduling step along with the associated DL value are presented in the scheduling trace shown in figure 12.b. The generated schedule has a length of 65.5.

The schedule generated by the HEFT algorithm along with its schedule trace, are shown in figure 13.a and 13.b respectively. The generated schedule length is equal to the length of the schedule generated using the DLS algorithm. Obviously, the task selection procedure employed by HEFT algorithm for this DHECS example is not efficient. This is because the algorithm uses the mean value of computation costs to compute the upward rank of tasks, and hence to select tasks for scheduling.

The schedule generated by the LDCP algorithm is shown in figure 14.a. The schedule length is 64 which is shorter than that of the DLS and HEFT algorithms. This is because the LDCP selects a task for scheduling based on its importance in determining the provisional schedule length.

A summary of the scheduling trace is shown in figure 14.b. The tasks selected at each step along with the selected processors are shown in the second and third columns. The DAGPs used to select the scheduled tasks are shown in the last column.

The detailed schedule traces at each scheduling step are shown in figures 15 to 25. At the first step, $DAGP_0$, shown in figure 15.a, is added to the active-DAGPs set. As shown in figure 15.b, nodes n_0 and n_1 in $DAGP_0$ have the highest URank values. However, n_1 has a higher number of output edges so it becomes the *key node*. Node n_1 is marked by KN in the fourth column. Since n_1 has no unscheduled parents, it identifies t_1 to be selected for scheduling. Since n_1 in $DAGP_0$ is used to identify the selected task, it is marked by an asterisk in the first column. The set *N* is composed from the nodes n_1, n_4 and n_9 as shown in third column in figure 14.b. Hence, the LDCP is composed from the tasks t_1, t_4 and n_9. Processor p_0, shown in the last column, is selected to execute task t_1. By the end of this step, the size of n_1 in $DAGP_0$ and $DAGP_1$ is set to the computation cost of task t_1 at processor p_0. The URank Value of node n_1 is updated in both DAGPs. A temporary zero-cost edge, indicated by a doted edge, is inserted from n_1 to n_0 on $DAGP_0$ as shown in figure 16.a.

At the second step, $DAGP_1$, shown in figure 16.c, is added to the active-DAGPs set. Since node n_1 is already scheduled, it is marked by 'S' in the fourth column in the tables shown in figures 16.b and 16.d. The size of node n_1 in both DAGPs, shown in figures 16.a and 16.c, is equal to the computation cost of t_1 at processor p_0. Node n_0 in $DAGP_1$ has the highest URank value over both DAGPs. Hence, n_0 becomes the *key node* and hence it is marked by KN in the fourth column of the table shown in figure 16.d. Because n_0 in $DAGP_1$ is used to identify the selected task, it is marked by an asterisk in the first column of the table in figure 16.d. Processor p_1 is selected to execute t_0 as shown in the last column of the table in figure 16.d. By the end of this step, the size of n_0 in $DAGP_0$ and $DAGP_1$ is set to the computation cost of task t_0 at processor p_1. The URank Values of nodes n_0 and n_1 are updated in both DAGPs. In $DAGP_0$, temporary zero-cost edges are inserted from n_1 to n_3 and n_6. In addition, in $DAGP_1$ temporary zero-cost edges are inserted from n_0 to n_2, n_4, n_5 and n_7.

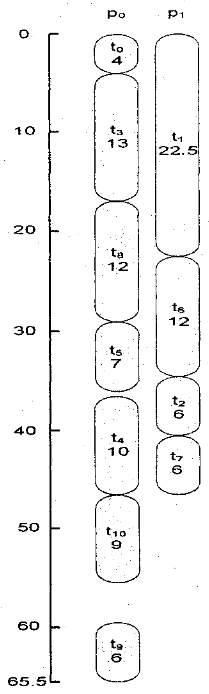
At the third step, node n_0 in $DAGP_1$ has the highest URank value over both DAGPs. The set *N* is composed from the nodes n_0, n_3, n_8 and n_{10} in $DAGP_1$, as shown in figure 17.d. Node n_3 in $DAGP_1$ is the first unscheduled node in *N*, so it comes to be the *Key node*. Since n_3 in $DAGP_1$ has no unscheduled parents, it identifies task t_3 to be selected for scheduling. By the end of this step, the size of n_3 in $DAGP_0$ and $DAGP_1$ is set to the computation cost of task t_3 at processor p_1. The URank Values of nodes n_0, n_1 and n_3 are updated in both DAGPs. In $DAGP_0$, temporary zero-cost edges are inserted from n_1 to n_6 and n_8. In addition, in $DAGP_1$ temporary zero-cost edges are added from n_3 to n_5, n_6 and n_7. The communication cost between node n_0 and n_3 is set to zero on both DAGPs.

At the fourth step, task t_8 is scheduled at processor p_0. Hence, by the end of this step, a zero-cost edge is inserted from node n_1 to n_8 on both DAGPs. This edge is indicated by a solid thick edge in figures 19.a and 19.c.

At the seventh step, the set *N* is composed from the nodes n_0, n_3, n_8, n_4, n_2 and n_9. Node n_9 is defined to be the *key node*. Since n_9 has unscheduled parents, node n_6 becomes the *parent key node* and is marked by PKN in the fourth column in figure 21.d. Node n_6 defines task t_6 to be selected for scheduling, hence it is marked by an asterisk in the first column in figure 21.d.

At the tenth scheduling step, task t_9 is scheduled on processor p_0. Hence, nodes n_4, n_5, n_7 and n_9 in $DAGP_1$ become dummy tasks. The dummy tasks are marked by a cross in $DAGP_1$ as shown in figure 25.c.

The schedule generated by the LDCP algorithm is then encoded in the chromosome shown in figure 26.a. The GATS inserts the chromosome that encodes the LDCP generated schedule into its initial population. After running the GATS for 15 generations, the chromosome that has the highest fitness value is shown in figure 26.b. This chromosome is then decoded to generate the final schedule of the H2GS algorithm. The generated schedule has a length of 61.5 and is shown in figure 26.c.

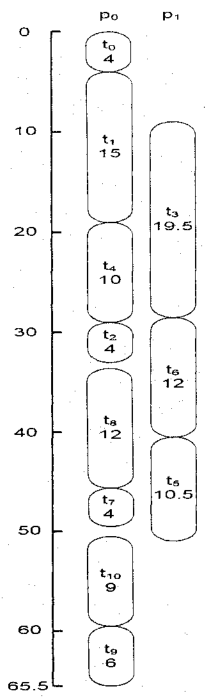


(a)

step	task	processor	DL
1	t_0	p_0	48.5
2	t_3	p_0	41.75
3	t_1	p_1	35
4	t_8	p_0	12.25
5	t_6	p_1	-3.25
6	t_5	p_0	-11
7	t_4	p_0	-14
8	t_2	p_1	-19.25
9	t_7	p_1	-29
10	t_{10}	p_0	-33
11	t_9	p_0	-50.5

(b)

Figure 12. (a) The schedule generated by the DLS algorithm (schedule length = 65.5), (b) the schedule trace of the DLS algorithm

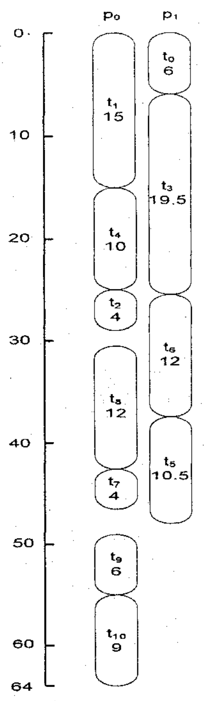


(a)

node	upward rank	priority in the task list
t_0	68.5	1
t_1	66.75	2
t_2	31.5	6
t_3	58.5	3
t_4	34	5
t_5	17.25	8
t_6	31.25	7
t_7	13.5	9
t_8	37.25	4
t_9	7.5	11
t_{10}	11.25	10

(b)

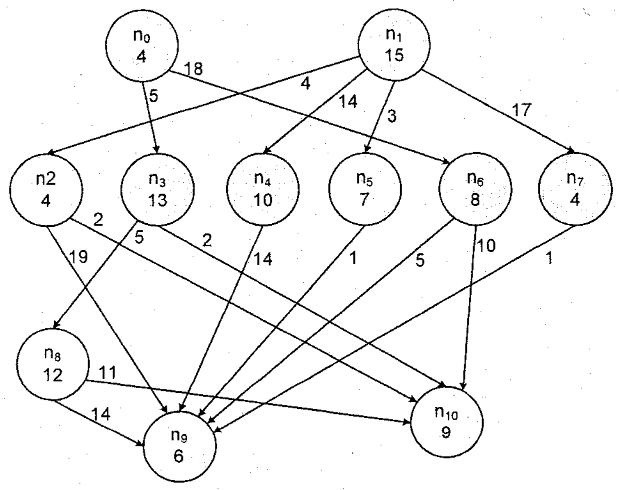
Figure 13. (a) The schedule generated by the HEFT algorithm (schedule length = 65.5), (b) the schedule trace of the HEFT algorithm



Step	Task	Processor	DAGP
1	t_1	p_0	$DAGP_0$
2	t_0	p_1	$DAGP_1$
3	t_3	p_1	$DAGP_1$
4	t_8	p_0	$DAGP_1$
5	t_4	p_0	$DAGP_0$
6	t_2	p_0	$DAGP_0$
7	t_6	p_1	$DAGP_1$
8	t_5	p_1	$DAGP_1$
9	t_7	p_0	$DAGP_1$
10	t_9	p_0	$DAGP_1$
11	t_{10}	p_0	$DAGP_0$

(a) (b)

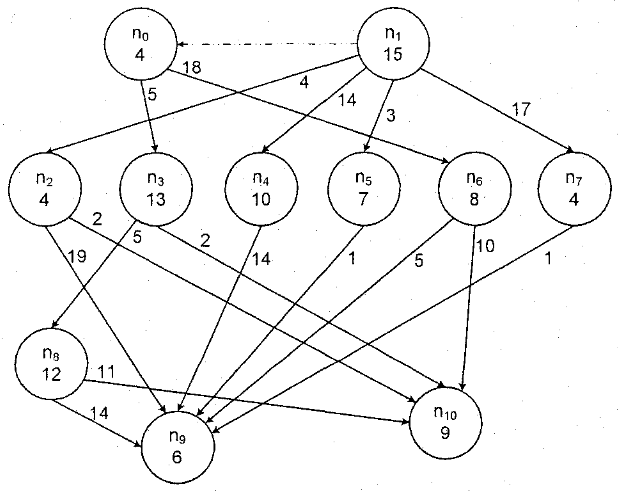
Figure 14. (a) The schedule generated by the LDCP algorithm (schedule length = 64), (b) the summary of schedule trace of the LDCP algorithm



Node	$URank_0$	N	Flag	Processor
n_0	59			
$*n_1$	59	1	KN	p_0
n_2	29			
n_3	50			
n_4	30	2		
n_5	14			
n_6	27			
n_7	11			
n_8	32			
n_9	6	3		
n_{10}	9			

(a) (b)

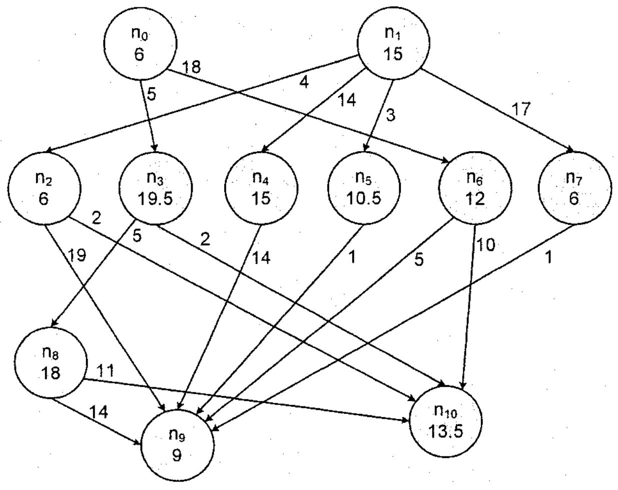
Figure 15 . (a) $DAGP_0$ at step 1 of the LDCP algorithm (b) the nodes attributes of $DAGP_0$ at step 1 of the LDCP algorithm (c) $DAGP_1$ at step 1 of the LDCP algorithm (d) the nodes attributes of $DAGP_1$ at step 1 of the LDCP algorithm



(a)

Node	$URank_0$	N	Flag	Processor
n_0	59			
n_1	74		S	p_0
n_2	29			
n_3	50			
n_4	30			
n_5	14			
n_6	27			
n_7	11			
n_8	32			
n_9	6			
n_{10}	9			

(b)

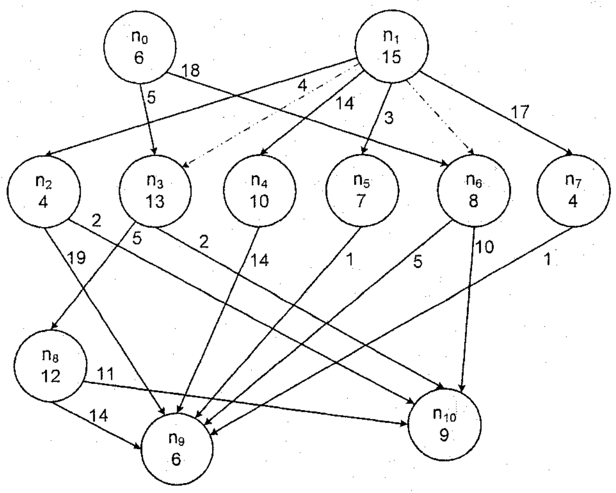


(c)

Node	$URank_0$	N	Flag	Processor
*n_0	78	1	KN	p_1
n_1	67		S	p_0
n_2	34			
n_3	67	2		
n_4	38			
n_5	20.5			
n_6	35.5			
n_7	16			
n_8	42.5	3		
n_9	9			
n_{10}	13.5	4		

(d)

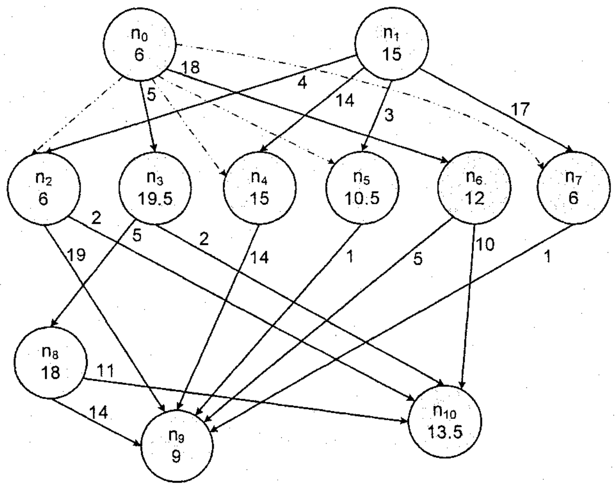
Figure 16. (a) $DAGP_0$ at step 2 of the LDCP algorithm (b) the nodes attributes of $DAGP_0$ at step 2 of the LDCP algorithm (c) $DAGP_1$ at step 2 of the LDCP algorithm (d) the nodes attributes of $DAGP_1$ at step 2 of the LDCP algorithm



(a)

Node	$URank_0$	N	Flag	Processor
n_0	61		S	p_1
n_1	65		S	p_0
n_2	29			
n_3	50			
n_4	30			
n_5	14			
n_6	27			
n_7	11			
n_8	32			
n_9	6			
n_{10}	9			

(b)

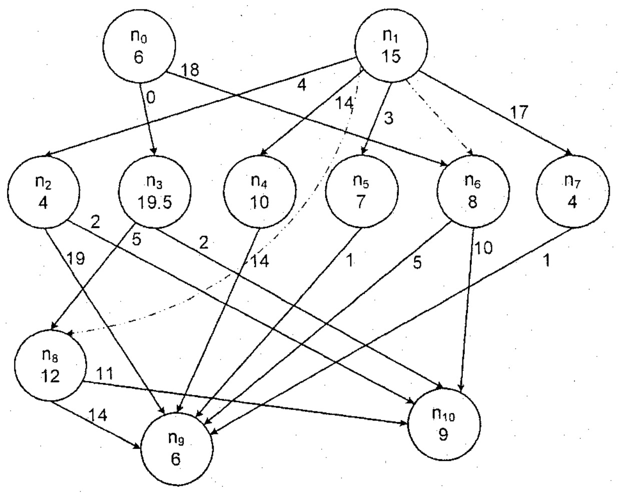


(c)

Node	$URank_0$	N	Flag	Processor
n_0	78	1	S	p_1
n_1	67		S	p_0
n_2	34			
$*n_3$	67	2	KN	p_1
n_4	38			
n_5	20.5			
n_6	35.5			
n_7	16			
n_8	42.5	3		
n_9	9			
n_{10}	13.5	4		

(d)

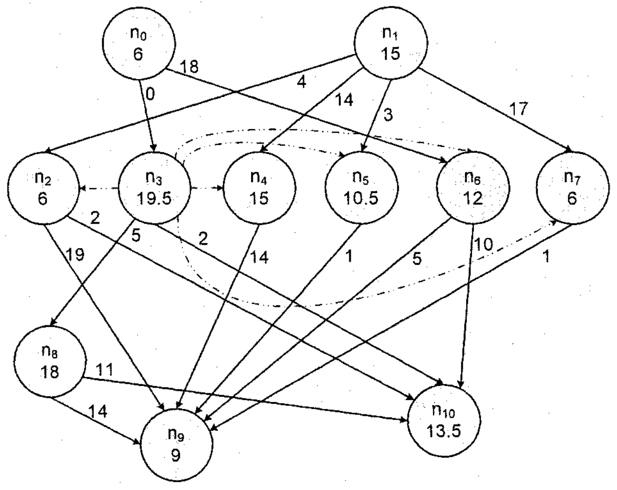
Figure 17. (a) $DAGP_0$ at step 3 of the LDCP algorithm (b) the nodes attributes of $DAGP_0$ at step 3 of the LDCP algorithm (c) $DAGP_1$ at step 3 of the LDCP algorithm (d) the nodes attributes of $DAGP_1$ at step 3 of the LDCP algorithm



(a)

Node	$URank_0$	N	Flag	Processor
n_0	62.5		S	p_1
n_1	59		S	p_0
n_2	29			
n_3	56.5		S	p_1
n_4	30			
n_5	14			
n_6	27			
n_7	11			
n_8	32			
n_9	6			
n_{10}	9			

(b)

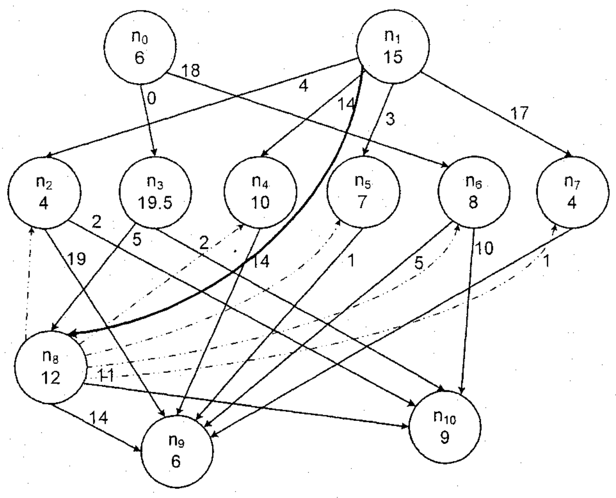


(c)

Node	$URank_0$	N	Flag	Processor
n_0	73	1	S	p_1
n_1	67		S	p_0
n_2	34			
n_3	67	2	S	p_1
n_4	38			
n_5	20.5			
n_6	35.5			
n_7	16			
*n_8	42.5	3	KN	p_0
n_9	9			
n_{10}	13.5	4		

(d)

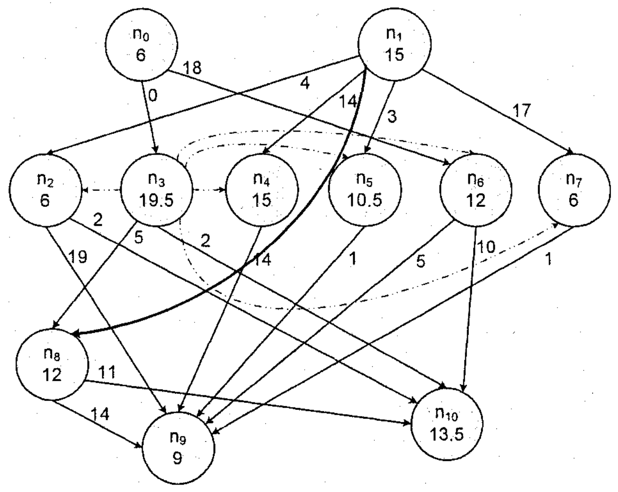
Figure 18. (a) $DAGP_0$ at step 4 of the LDCP algorithm (b) the nodes attributes of $DAGP_0$ at step 4 of the LDCP algorithm (c) $DAGP_1$ at step 4 of the LDCP algorithm (d) the nodes attributes of $DAGP_1$ at step 4 of the LDCP algorithm



(a)

Node	$URank_0$	N	Flag	Processor
n_0	72.5	1	S	p_1
n_1	59		S	p_0
n_2	29			
n_3	66.5	2	S	p_1
$*n_4$	30	4	KN	p_0
n_5	14			
n_6	27			
n_7	11			
n_8	42	3	S	p_0
n_9	6	5		
n_{10}	9			

(b)

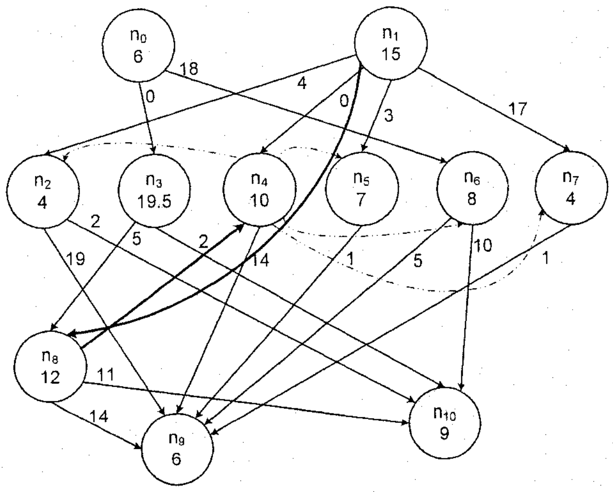


(c)

Node	$URank_0$	N	Flag	Processor
n_0	67		S	p_1
n_1	67		S	p_0
n_2	34			
n_3	61		S	p_1
n_4	38			
n_5	20.5			
n_6	35.5			
n_7	16			
n_8	36.5		S	p_0
n_9	9			
n_{10}	13.5			

(d)

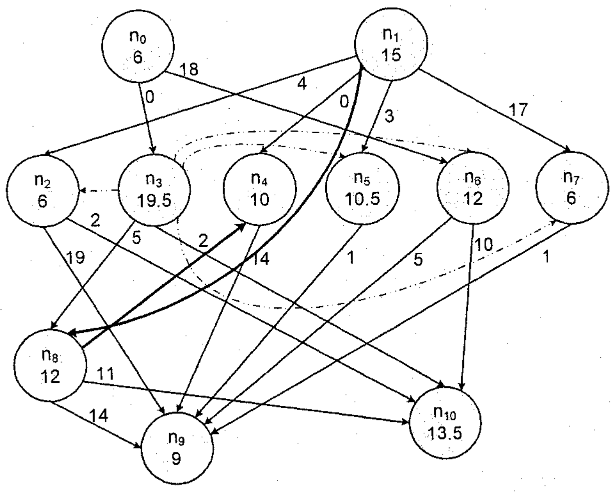
Figure 19. (a) $DAGP_0$ at step 5 of the LDCP algorithm (b) the nodes attributes of $DAGP_0$ at step 5 of the LDCP algorithm (c) $DAGP_1$ at step 5 of the LDCP algorithm (d) the nodes attributes of $DAGP_1$ at step 5 of the LDCP algorithm



(a)

Node	URank$_0$	N	Flag	Processor
n_0	81.5	1	S	p_1
n_1	66		S	p_0
*n_2	29	5	KN	p_0
n_3	75.5	2	S	p_1
n_4	39	4	S	p_0
n_5	14			
n_6	27			
n_7	11			
n_8	51	3	S	p_0
n_9	6	6		
n_{10}	9			

(b)

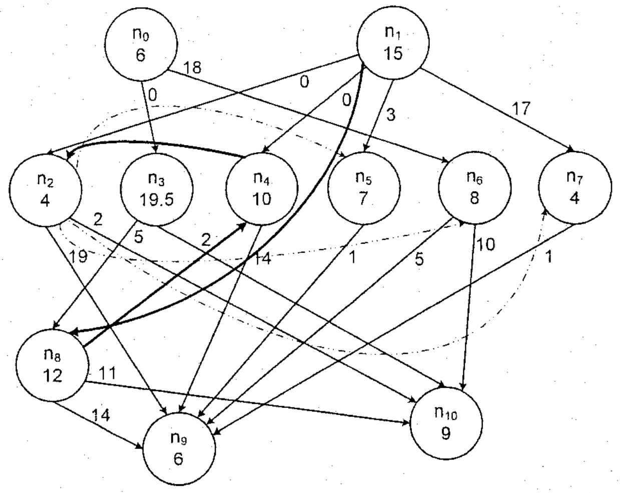


(c)

Node	URank$_0$	N	Flag	Processor
n_0	75.5		S	p_1
n_1	60		S	p_0
n_2	34			
n_3	69.5		S	p_1
n_4	33		S	p_0
n_5	20.5			
n_6	35.5			
n_7	16			
n_8	45		S	p_0
n_9	9			
n_{10}	13.5			

(d)

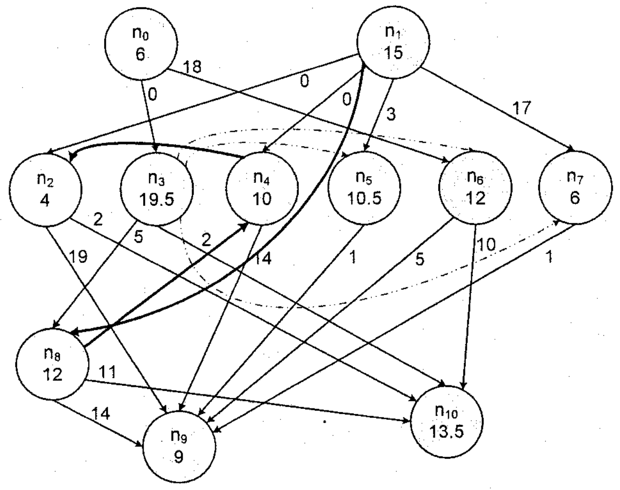
Figure 20. (a) DAGP$_0$ at step 6 of the LDCP algorithm (b) the nodes attributes of DAGP$_0$ at step 6 of the LDCP algorithm (c) DAGP$_1$ at step 6 of the LDCP algorithm (d) the nodes attributes of DAGP$_1$ at step 6 of the LDCP algorithm



(a)

Node	URank$_0$	N	Flag	Processor
n_0	83.5		S	P_1
n_1	68		S	P_0
n_2	31		S	P_0
n_3	77.5		S	P_1
n_4	41		S	P_0
n_5	14			
n_6	27			
n_7	11			
n_8	53		S	P_0
n_9	6			
n_{10}	9			

(b)

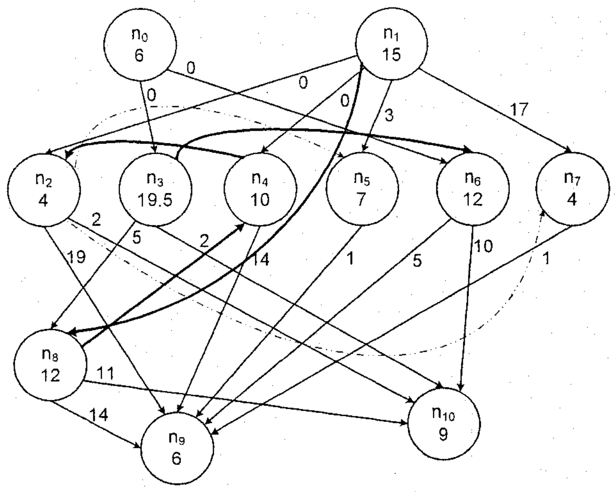


(c)

Node	URank$_0$	N	Flag	Processor
n_0	84.5	1	S	p_1
n_1	69		S	p_0
n_2	32	5	S	p_0
n_3	78.5	2	S	p_1
n_4	42	4	S	p_0
n_5	20.5			
*n_6	35.5		PKN	p_1
n_7	16			
n_8	54	3	S	p_0
n_9	9	6	KN	
n_{10}	13.5			

(d)

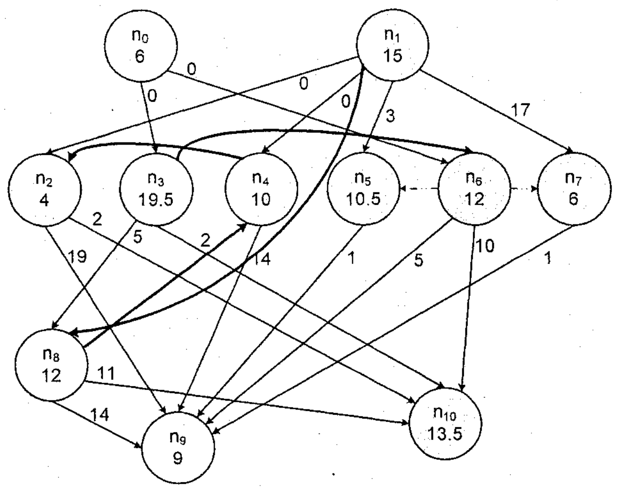
Figure 21. (a) DAGP$_0$ at step 7 of the LDCP algorithm (b) the nodes attributes of DAGP$_0$ at step 7 of the LDCP algorithm (c) DAGP$_1$ at step 7 of the LDCP algorithm (d) the nodes attributes of DAGP$_1$ at step 7 of the LDCP algorithm



(a)

Node	URank$_0$	N	Flag	Processor
n_0	81.5		S	p_1
n_1	66		S	p_0
n_2	29		S	p_0
n_3	75.5		S	p_1
n_4	39		S	p_0
n_5	14			
n_6	31		S	p_1
n_7	11			
n_8	51		S	p_0
n_9	6			
n_{10}	9			

(b)

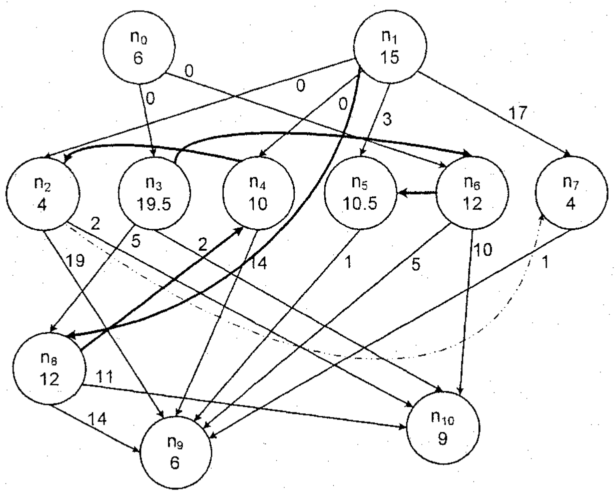


(c)

Node	URank$_0$	N	Flag	Processor
n_0	84.5	1	S	p_1
n_1	69		S	p_0
n_2	32	5	S	p_0
n_3	78.5	2	S	p_1
n_4	42	4	S	p_0
*n_5	20.5		PK N	p_1
n_6	35.5		S	p_1
n_7	16			
n_8	54	3	S	p_0
n_9	9	6	KN	
n_{10}	13.5			

(d)

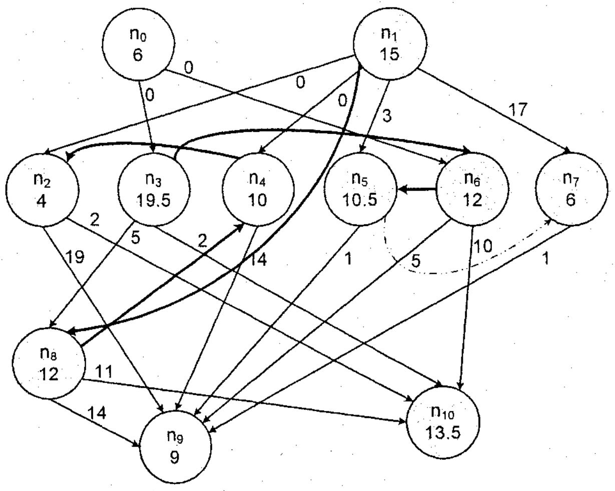
Figure 22. (a) DAGP$_0$ at step 8 of the LDCP algorithm (b) the nodes attributes of DAGP$_0$ at step 8 of the LDCP algorithm (c) DAGP$_1$ at step 8 of the LDCP algorithm (d) the nodes attributes of DAGP$_1$ at step 8 of the LDCP algorithm



(a)

Node	URank$_0$	N	Flag	Processor
n_0	81.5		S	p_1
n_1	66		S	p_0
n_2	29		S	p_0
n_3	75.5		S	p_1
n_4	39		S	p_0
n_5	17.5		S	p_1
n_6	31		S	p_1
n_7	11			
n_8	51		S	p_0
n_9	6			
n_{10}	9			

(b)

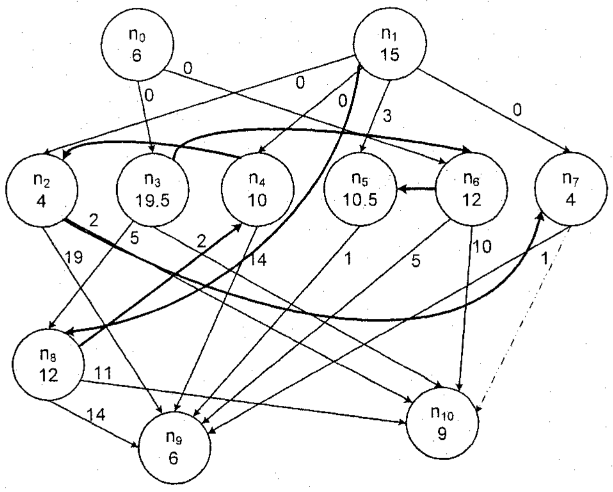


(c)

Node	URank$_0$	N	Flag	Processor
n_0	84.5	1	S	p_1
n_1	69		S	p_0
n_2	32	5	S	p_0
n_3	78.5	2	S	p_1
n_4	42	4	S	p_0
n_5	26.5		S	p_1
n_6	36.5		S	p_1
*n_7	16		PKN	p_0
n_8	54	3	S	p_0
n_9	9	6	KN	
n_{10}	13.5			

(d)

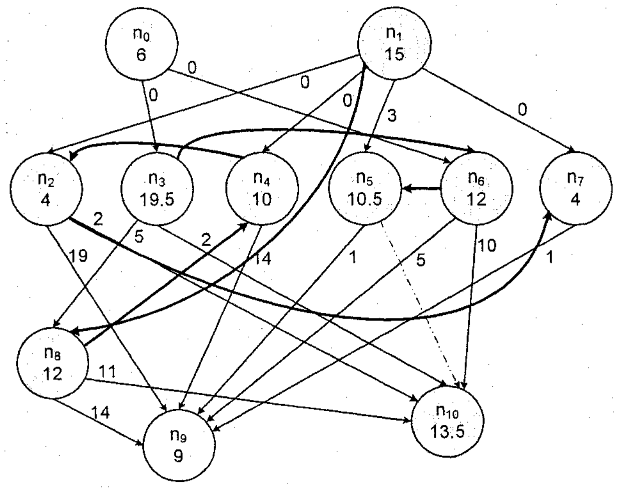
Figure 23. (a) DAGP$_0$ at step 9 of the LDCP algorithm (b) the nodes attributes of DAGP$_0$ at step 9 of the LDCP algorithm (c) DAGP$_1$ at step 9 of the LDCP algorithm (d) the nodes attributes of DAGP$_1$ at step 9 of the LDCP algorithm



(a)

Node	URank$_0$	N	Flag	Processor
n_0	81.5		S	p_1
n_1	66		S	p_0
n_2	29		S	p_0
n_3	75.5		S	p_1
n_4	39		S	p_0
n_5	17.5		S	p_1
n_6	31		S	p_1
n_7	13		S	p_0
n_8	51		S	p_0
n_9	6			
n_{10}	9			

(b)

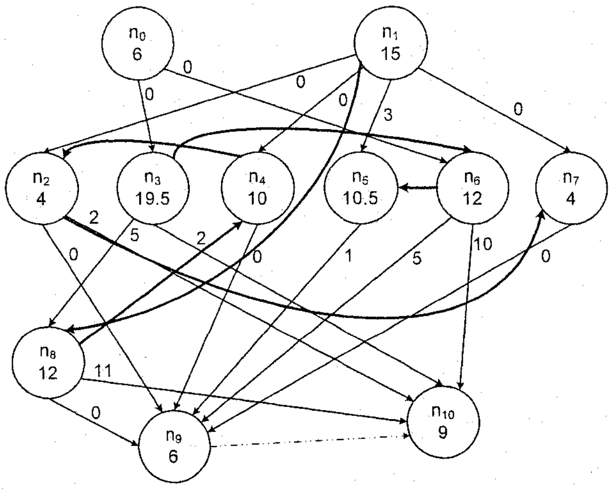


(c)

Node	URank$_0$	N	Flag	Processor
n_0	84.5	1	S	p_1
n_1	69		S	p_0
n_2	32	5	S	p_0
n_3	78.5	2	S	p_1
n_4	42	4	S	p_0
n_5	24		S	p_1
n_6	36		S	p_1
n_7	14		S	p_0
n_8	54	3	S	p_0
*n_9	9	6	KN	p_0
n_{10}	13.5			

(d)

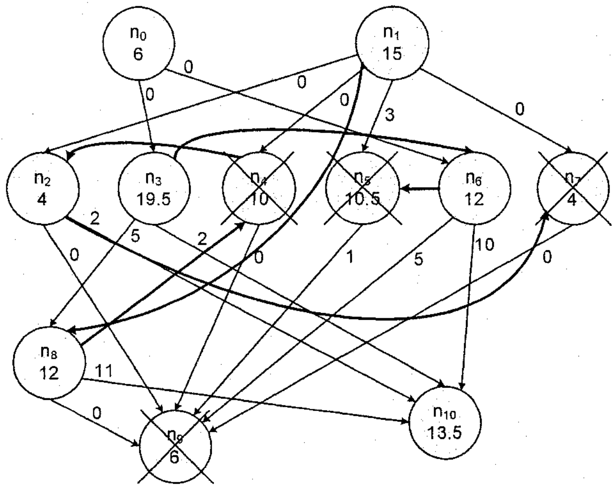
Figure 24. (a) DAGP$_0$ at step 10 of the LDCP algorithm (b) the nodes attributes of DAGP$_0$ at step 10 of the LDCP algorithm (c) DAGP$_1$ at step 10 of the LDCP algorithm (d) the nodes attributes of DAGP$_1$ at step 10 of the LDCP algorithm



(a)

Node	$URank_0$	N	Flag	Processor
n_0	75.5	1	S	p_1
n_1	60		S	p_0
n_2	23	5	S	p_0
n_3	69.5	2	S	p_1
n_4	33	4	S	p_0
n_5	26.5		S	p_1
n_6	38.5		S	p_1
n_7	19	6	S	p_0
n_8	45	3	S	p_0
n_9	15	7	S	p_0
$*n_{10}$	9	8	KN	p_0

(b)



(c)

Node	$URank_0$	N	Flag	Processor
n_0	67		S	p_1
n_1	51.5		S	p_0
n_2	19.5		S	p_0
n_3	61		S	p_1
n_4	-		S	p_0
n_5	-		S	p_1
n_6	35.5		S	p_1
n_7	-		S	p_0
n_8	36.5		S	p_0
n_9	-		S	p_0
n_{10}	13.5			

(d)

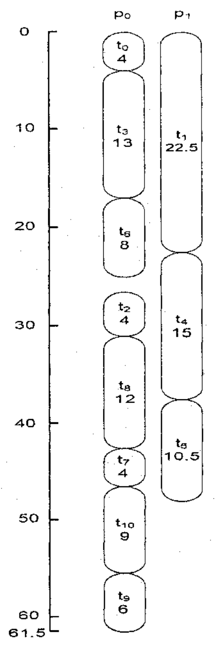
Figure 25. (a) $DAGP_0$ at step 11 of the LDCP algorithm (b) the nodes attributes of $DAGP_0$ at step 11 of the LDCP algorithm (c) $DAGP_1$ at step 11 of the LDCP algorithm (d) the nodes attributes of $DAGP_1$ at step 11 of the LDCP algorithm

p0	t1	t4	t2	t8	t7	t9	t10
p1	t0	t3	t6	t5			

(a)

p0	t0	t3	t10	t6	t2	t8	T7	t9
p1	t1	t4	t5					

(b)



(c)

Figure 26. (a) The chromosome that encodes the schedule generated by the LDCP algorithm (b) the best chromosome evolved by the GATS algorithm (c) the schedule generated by the H2GS algorithm (schedule length =61.5)

Chapter 6: Results and Analysis

In this section, we present a performance comparison of LDCP and H2GS algorithms as well as the two algorithms reported in section 3.2. For this purpose, the LDCP algorithm, the H2GS algorithm, the HEFT algorithm and the DLS algorithm are simulated. A set of randomly generated task graphs are generated and used as the workload for evaluating the algorithms. This section is divided into three subsections. First, the performance metrics used to evaluate the algorithms performance, are described. Second, the generation of the random task graph is explained. Finally, the performance of the four scheduling algorithm is studied.

6.1 Performance Metrics

The performance metrics chosen for the comparison are Normalized Schedule Length (NSL) [4], speedup [1], efficiency [4] and running time of the algorithms. The four metrics are explained below:

- **Normalized Schedule Length (NSL).** For the first comparison, we used the NSL produced by the LDCP algorithm, the H2GS algorithm, the HEFT algorithm and the DLS algorithm. For DHECS, the NSL of a given schedule is defined as the normalized schedule length to the lower bound of the schedule length. The NSL of a schedule is calculated using equation 2.

$$NSL = \frac{\text{Schedule Length}}{\sum_{t_i \in CP_{lower}} \mathrm{c}_{\mathrm{i,a}}} \qquad (2)$$

The CP_{lower} is defined as the CP of the unscheduled application DAG, based on the computation cost of tasks on the fastest processor p_a. The denominator of equation 2 is equal to the sum of computation costs of tasks located on CP_{lower}, when they are executed on p_a. This denominator gives the absolute lower bound of the schedule length. For a given application DAG, the scheduling algorithm that generates a schedule with the lowest NSL value, is the best algorithm from a performance point of view. The average NSL value over a set of application DAGs, is used to study the performance of the algorithms.

- **Speedup.** The speedup of a task schedule is defined as ratio of the schedule length obtained by assigning all tasks to the fastest processor p_a, to the parallel execution time of the task schedule. The speedup value of a task schedule is calculated using equation 3.

$$Speedup = \frac{\sum_{t_i \in T} \mathrm{c}_{\mathrm{i,a}}}{\text{Schedule Length}} \qquad (3)$$

- **Efficiency.** The efficiency of a task schedule is defined as the ratio of the speedup value to the number of processors used. The efficiency calculation is presented in equation 4.

$$Efficiency = \frac{Speedup}{\text{Number of processor used}} \times 100 \qquad (4)$$

- **Running Time.** The running time of a scheduling algorithm is defined as the execution time needed to produce the output schedule. The average running time over a set of application DAGs is studied for the four scheduling algorithms.

6.2 Experimental Set-Up

In our simulated environment, four different DHECSs are used to evaluate the performance of the algorithms. A random DAG generator is used first to generate a set of application DAGs for each one of the four DHECSs. This random DAG generator has a set of input parameters that determine the characteristics of the generated DAGs. Next, the four scheduling algorithms are run to schedule the generated DAGs on each DHECS. Finally, the performance of the scheduling algorithms is evaluated using the performance metrics.

The random DAG generator has the following input parameters:

- DAG size, n, which is the number of tasks in the application DAG.

- Communication to computation cost ratio, *CCR*. It is defined as the average communication cost divided by the average computation cost of the application.

- Parallelism factor, α [1]. The number of levels of the application DAG is calculated by randomly generating a number using a uniform distribution with a mean value of $\sqrt{n}/\alpha$, and then rounding it up to the nearest integer. The width of each level is calculated by randomly generating a number using a uniform distribution with a mean value of $\alpha \times \sqrt{n}$, and then rounding it up to the nearest integer. A low α value leads to a DAG with a low parallelism degree [4].

- The computation cost heterogeneity factor, *h*. A high *h* indicates high variance of the computation costs of a task, with respect to the processors in the system, and *visa versa*. If the heterogeneity factor is set to 0, the computation cost of a task is the same for all processors. The average computation cost of a task t_i ($\overline{w_i}$) is randomly generated using a uniform distribution with a mean value of $\overline{W}$. It is presumed to be an approximation of the average computation cost of the application DAG. If there are *m* processors in the DHECS, the computation cost of a task t_i for each processor is set by randomly selecting *m* computation cost values of t_i from the range below:

$$\left[\left(\overline{w_i} \times \left(1 - \frac{h}{2}\right)\right), \left(\overline{w_i} \times \left(1 + \frac{h}{2}\right)\right)\right]$$

The *m* selected computation cost values of t_i are sorted in an increasing order. The computation cost value of t_i on processor p_0 is set to the first (*i.e* lowest) computation cost. The computation cost of t_i on processor p_1 is set to the second value. This processor allocation continues until all processors are processed [1].

The four DHECSs are composed of 2, 4, 6 and 8 processors respectively. For each DHECS, the generated workload suite consists of 500 random generated DAGs, with 5 different DAG sizes varying form 20 to 100 nodes with an increment of 20 nodes. For each DAG size, there are five different *CCR* values: 0.1, 0.5, 1.0, 2.0, and 5.0, four α values: 0.5, 1.0, 2.0, and 5.0, and five *h* values: 0.1, 0.2, 0.4, 0.6 and 0.8.

6.3 Performance Results

The performance comparison of the LDCP, H2GS, and HEFT and DLS algorithms is done in a number of ways. First, the NSLs produced by these algorithms are compared with each other with respect to various DAG sizes, *CCR* values and numbers of processors. Second, the speedups obtained by these four algorithms are compared against each other by varying the number of nodes, *CCR* value and number of processors. And finally, the efficiency of schedules generated by the four scheduling algorithm is studied with respect to the number of nodes and the *CCR* value.

6.3.1 Comparison of Schedule Lengths

The NSLs produced by each algorithm for each DAG size, *CCR* value and number of processors are shown in tables 1, 2, and 3, and figures 27, 28 and 29, respectively. Every result for a DAG size is an average of 400 application DAGs (5 *CCR* values × 4 α

values × 4 m vales × 5 h values). Every result for a CCR is an average of 400 DAGs (5 n values × 4 α values × 4 m vales × 5 h values). And finally, every result for a number of processors is an average of 500 DAGs (5 n values × 5 CCR values × 4 α values × 5 h values).

As shown in tables 1, 2, and 3, and figures 27, 28 and 29, the performance of the LDCP algorithm outperforms both the HEFT and DLS algorithms. The GATS algorithm takes the schedule generated by the LDCP algorithm and improves it to produce an even shorter schedule than that produced by the LDCP algorithm.

The average NSL value of the LDCP algorithm, on all generated application DAGs, is shorter than that of the HEFT and DLS algorithms by 4.8% and 2.1% respectively. Moreover, the average NSL value produced by the H2GS algorithm is shorter than that of the HEFT and DLS algorithms by 8.8% and 6.2% respectively.

The performance of the GATS algorithm degrades at higher application sizes. At higher numbers of nodes, the size of the search space becomes larger and the GATS algorithm requires more generations to find better schedules.

A large CCR value tests the ability of the algorithm to deal with communication-intensive applications. The differences between the schedule length generated by the H2GS algorithm and those generated by the two other algorithms become more pronounced at larger CCR values.

The LDCP algorithm, using the LDCP attribute, identifies the heavily communicating tasks. Next, the insertion-based scheduling policy, implemented by the LDCP algorithm, determines if allocating those tasks to the same processor leads to a reduction in their finish execution time. The insertion-based scheduling policy allocates those tasks to processors that reduce their execution time. Moreover, the GATS algorithm further eliminates inter-processor communication if such elimination leads to a shorter schedule length.

TABLE 1: Comparison of Average NSL with Respect to *n*

N	DLS	HEFT	LDCP	H2GS
20	2.771	2.716	2.630	2.294
40	3.362	3.265	3.189	3.028
60	3.782	3.662	3.564	3.484
80	4.059	3.959	3.893	3.817
100	4.277	4.148	4.094	4.017

TABLE 2: Comparison of Average NSL with Respect to *CCR*

CCR	DLS	HEFT	LDCP	H2GS
0.1	2.762	2.760	2.734	2.704
0.5	2.866	2.842	2.797	2.756
1.0	3.134	3.056	2.990	2.939
2.0	3.771	3.635	3.561	3.510
5.0	5.719	5.457	5.288	4.730

TABLE 3: Comparison of Average NSL with Respect to *m*

M	DLS	HEFT	LDCP	H2GS
2	5.119	4.977	4.905	4.752
4	3.564	3.435	3.353	3.197
6	3.065	2.987	2.920	2.786
8	2.853	2.799	2.719	2.578

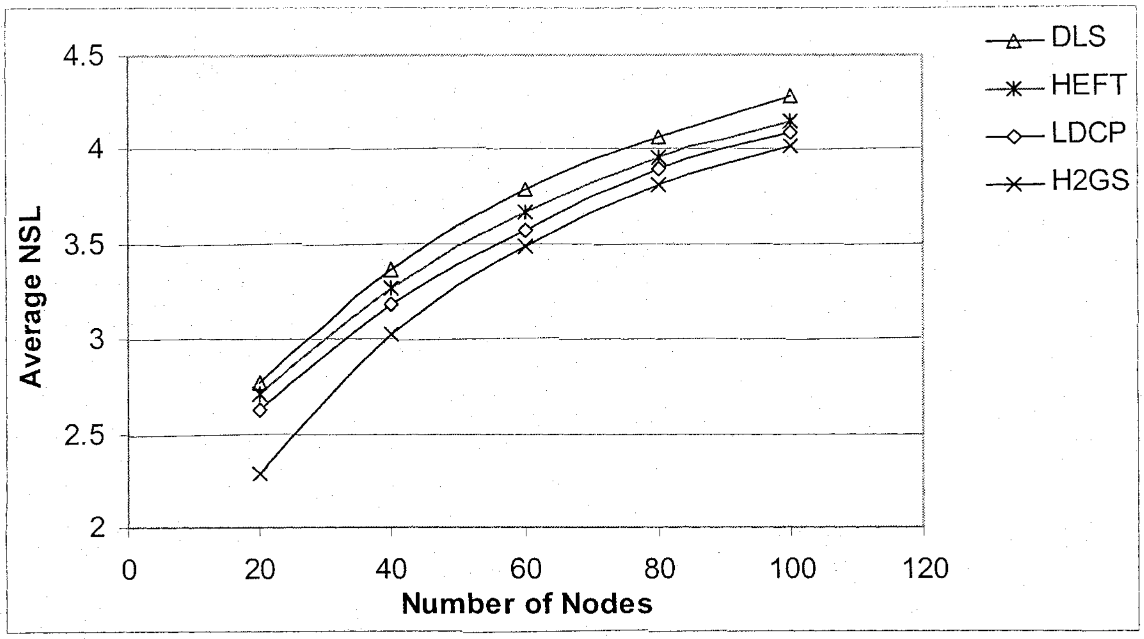


Figure 27. Average NSL with respect to DAG size

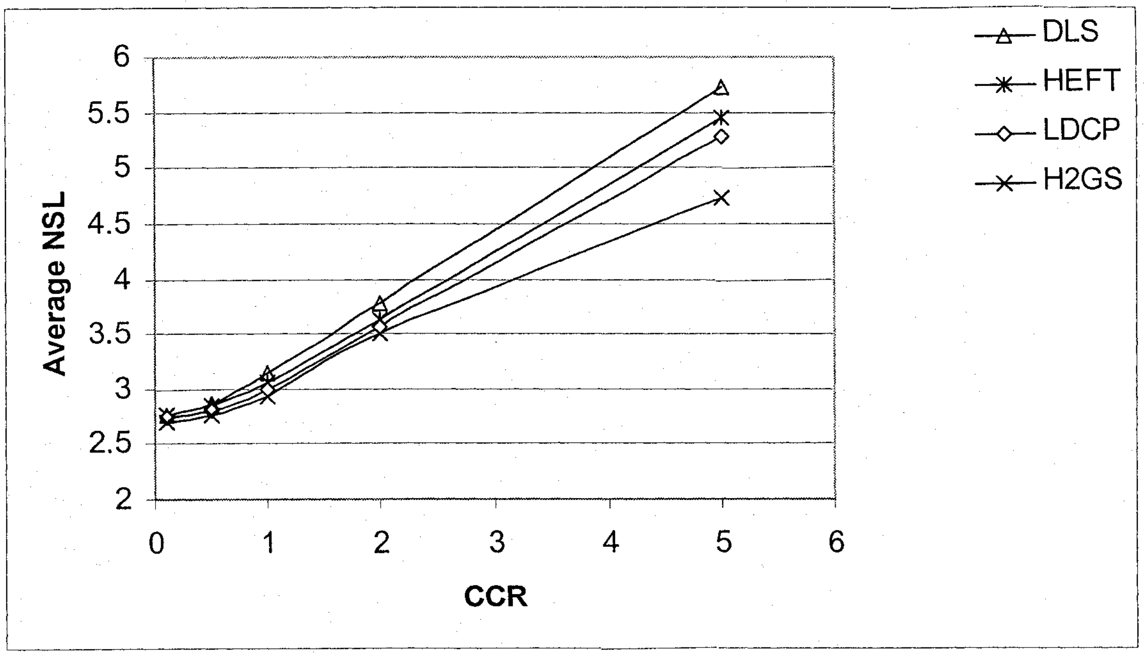


Figure 28. Average NSL with respect to *CCR*

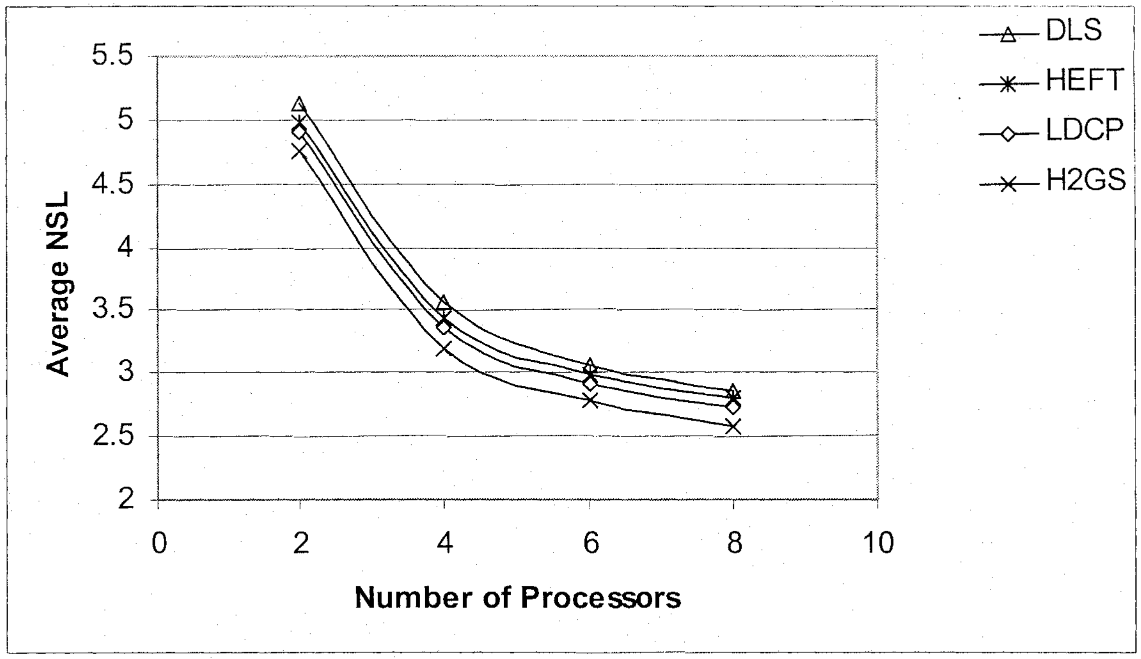


Figure 29. Average NSL with respect to number of processors

As the number of processors in the system increases, the computation cost of the application DAG is distributed between more processors. Hence, the generated schedule length becomes shorter. However, as the workload is distributed between more processors, the impact of the communication cost becomes higher. The differences between the schedule length generated by the H2GS algorithm and those generated by the two other algorithms become higher at larger numbers of processors. Hence, the performance advantage of the H2GS algorithm is even greater for large numbers of processors (up to 8).

As explained before, the H2GS algorithm identifies heavily communicating tasks. Those tasks are allocated to the same processors only if such an allocation leads to a shorter schedule length. The LDCP algorithm allocates each task to a processor that minimizes its finish execution time. Moreover, The GATS algorithm improves the allocation of tasks to generate a shorter schedule. The H2GS algorithm maintains a balance between exploiting the available processors in the system, and reducing inter-processor communications, to generate shorter schedules.

6.3.2 Comparison of Speedup

Tables 4, 5, and 6, and figures 30, 31 and 32, show the speedup gained by the four algorithms with respect to DAG size, *CCR* and number of processors receptively. The average speedup gained by the LDCP algorithm, on all generated application DAGs, is greater than that gained by the HEFT and DLS algorithms by 5.7% and 2.7% respectively. Moreover, the average Speedup gained by the H2GS algorithm is greater than that gained by the HEFT and DLS algorithms by 8.3% and 5.3% respectively.

.

As the application DAG increases, the number of nodes that can be executed, at the same time, in parallel becomes higher, and hence the gained speedup increases. Higher *CCR* values lead to higher communication costs, and hence the gained speedup decreases. As the number of processors in the system increases, the available computation recourses become higher, leading to higher speedup values.

TABLE 4: Comparison of Average Speedup with Respect to *n*

N	DLS	HEFT	LDCP	H2GS
20	1.652	1.709	1.761	1.870
40	2.132	2.202	2.259	2.314
60	2.386	2.460	2.543	2.602
80	2.589	2.654	2.717	2.760
100	2.749	2.820	2.887	2.923

TABLE 5: Comparison of Average Speedup with Respect to *CCR*

CCR	DLS	HEFT	LDCP	H2GS
0.1	3.198	3.215	3.260	3.294
0.5	2.913	2.973	3.051	3.123
1.0	2.457	2.567	2.655	2.708
2.0	1.902	1.978	2.034	2.052
5.0	1.040	1.112	1.168	1.291

TABLE 6: Comparison of Average Speedup with Respect to *m*

m	DLS	HEFT	LDCP	H2GS
2	1.308	1.348	1.379	1.409
4	2.218	2.295	2.353	2.402
6	2.600	2.680	2.765	2.850
8	3.082	3.154	3.236	3.313

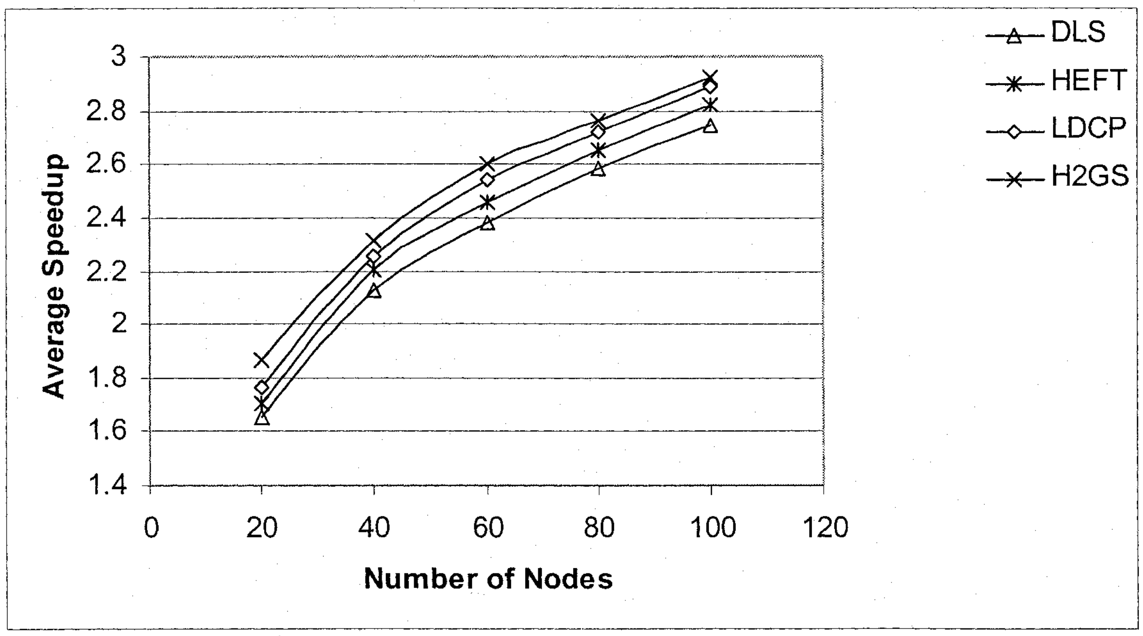


Figure 30. Average speedup with respect to DAG size

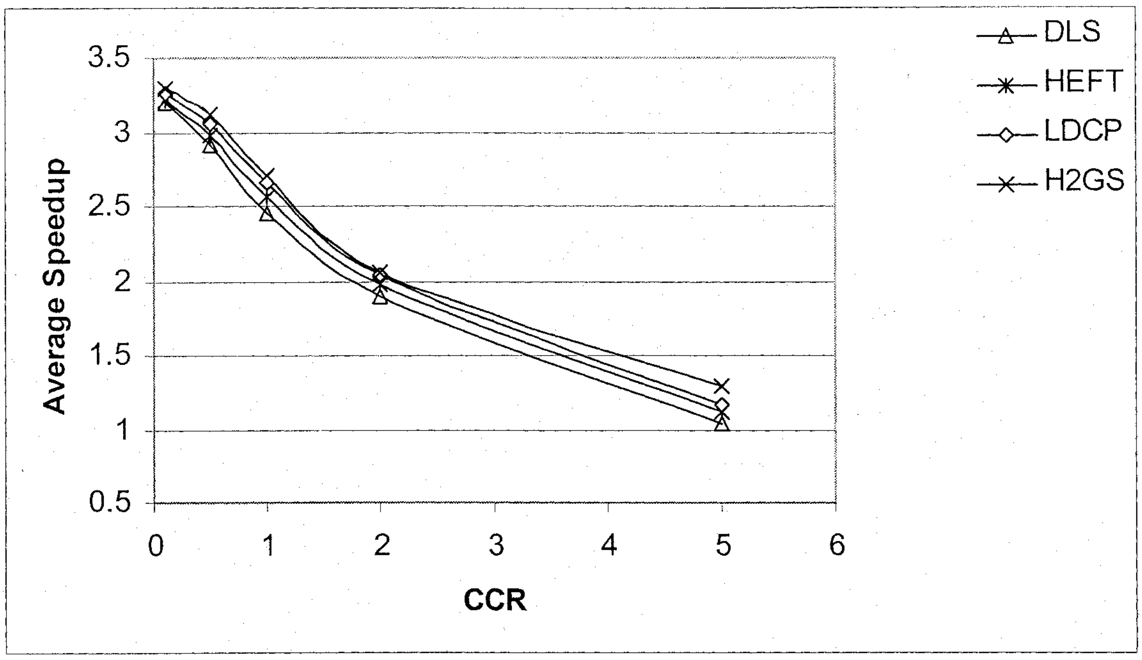


Figure 31. Average speedup with respect to *CCR*

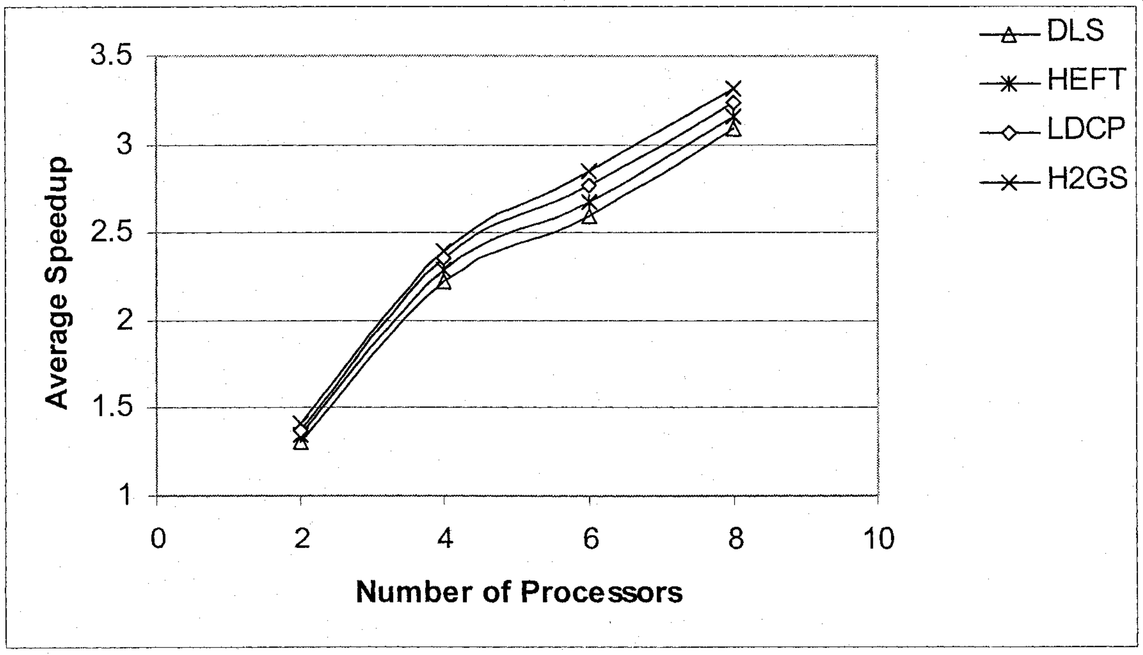


Figure 32. Average speedup with respect to Number of Processors

6.3.2 Comparison of Efficiency

Tables 7 and 8, and figures 33 and 34 show the efficiency gained by the DLS, HEFT, LDCP and H2GS algorithms with respect to the DAG size and *CCR* respectively. The average efficiency gained by the LDCP algorithm is better than that gained by the HEFT and DLS algorithms by 5.7% and 2.5% respectively. Moreover, the average efficiency gained by the H2GS algorithm is better than that gained by the HEFT and DLS algorithms by 8.5% and 5.2% respectively. These efficiency results reflect the superiority of the H2GS algorithm over all others.

As shown figure 33, at the beginning, as the number of tasks in the system increases the gained efficiency increases. However, after a while, increasing the DAG size will not further improve the gained efficiency.

When the DAG size is small, increasing the DAG size leads to increasing the number of tasks that can be executed, at the same time, in parallel. Hence, as the DAG size increases, the tasks can be distributed between the processors efficiently. However, when the number of tasks that can be executed in parallel becomes higher than execution ability of the system, any further increasing of the DAG size will not improve the efficiency of the system.

As shown in figure 34, increasing the *CCR* leads to lower efficiency values. Since, increasing the *CCR* value leads to higher communication costs between tasks, the resulting task schedule is longer and the gained efficiency is lower.

TABLE 7: Comparison of Average Efficiency with Respect to *n*

n	DLS	HEFT	LDCP	H2GS
20	0.444	0.460	0.476	0.506
40	0.526	0.548	0.560	0.574
60	0.555	0.574	0.590	0.602
80	0.563	0.578	0.592	0.603
100	0.574	0.585	0.595	0.604

TABLE 8: Comparison of Average Efficiency with Respect to *CCR*

CCR	DLS	HEFT	LDCP	H2GS
0.1	0.695	0.697	0.707	0.713
0.5	0.649	0.659	0.675	0.686
1.0	0.570	0.591	0.608	0.618
2.0	0.467	0.492	0.505	0.510
5.0	0.281	0.306	0.321	0.357

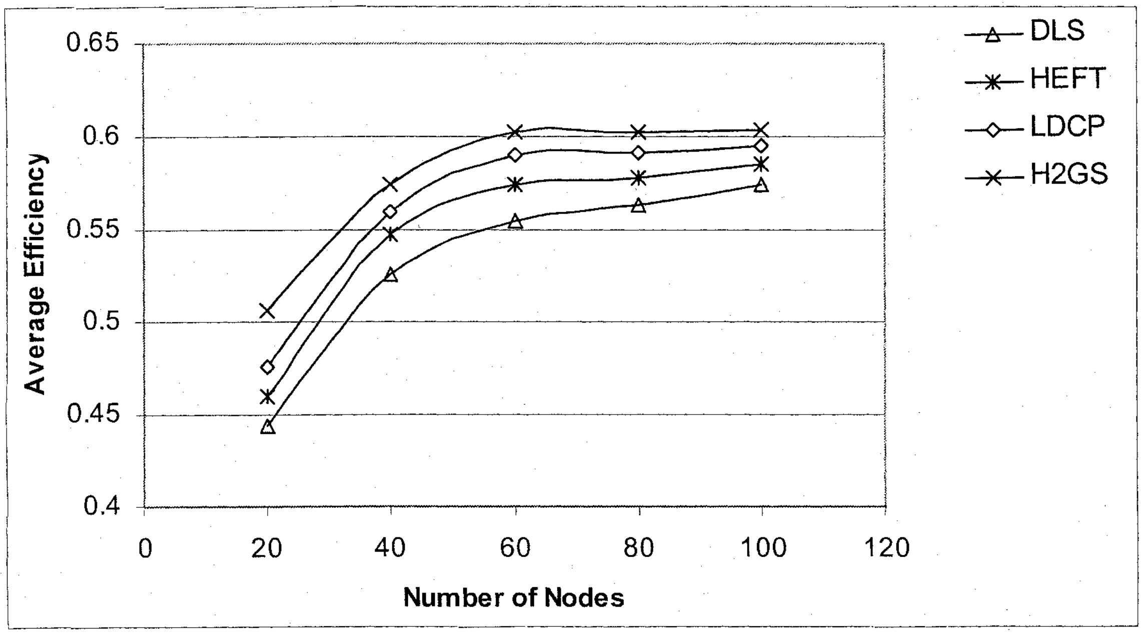


Figure 33. Average efficiency with respect to DAG size

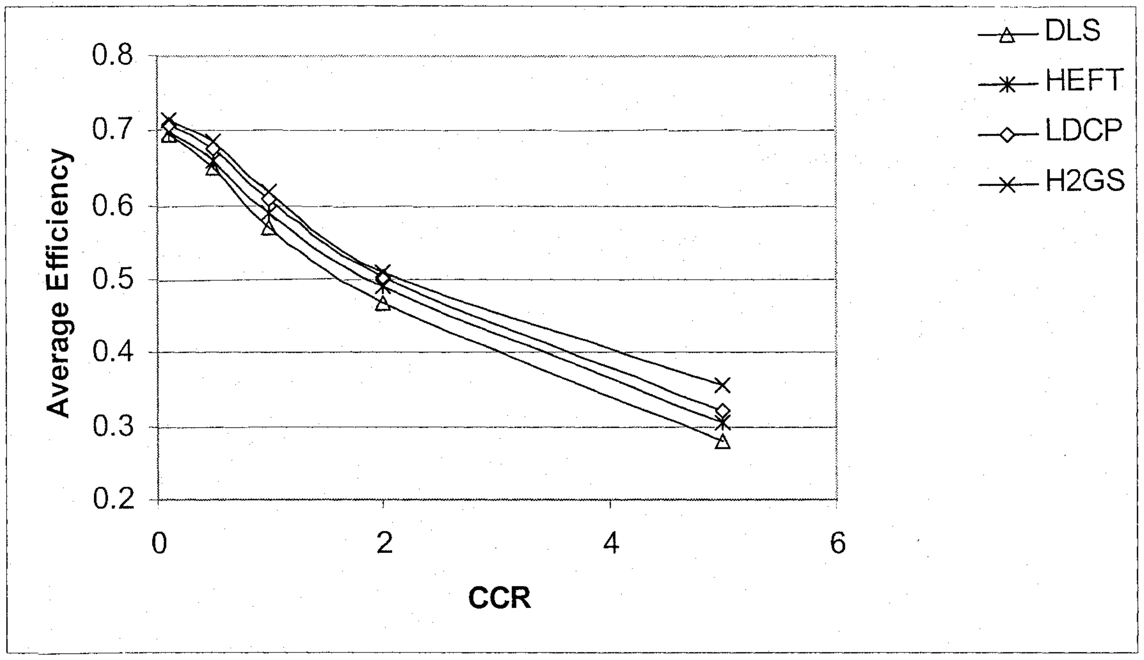


Figure 34. Average efficiency with respect to *CCR*

6.3.2 Comparison of Running Time

Tables 9 and figures 35 show the average running of the DLS, HEFT, LDCP and H2GS algorithms with respect to the DAG size. The average running time of the LDCP algorithm is higher than both the HEFT and DLS algorithms by 230.5% and 72.4% respectively. The average running time of the H2GS algorithm is higher than the HEFT and DLS algorithms by 435.3% and 179.1% respectively.

The LDCP requires higher time to select the tasks for scheduling than both the HEFT and DLS algorithms. The total running time of the H2GS algorithm is the sum of the running times of both the LDCP and GATS algorithms.

TABLE 9: Comparison of Average Efficiency with Respect to

n	DLS	HEFT	LDCP	H2GS
20	41.467	26.822	119.356	163.444
40	109.522	34.278	195.483	307.441
60	179.744	78.967	342.704	537.301
80	269.478	148.711	504.419	822.610
100	496.578	283.111	728.373	1230.256

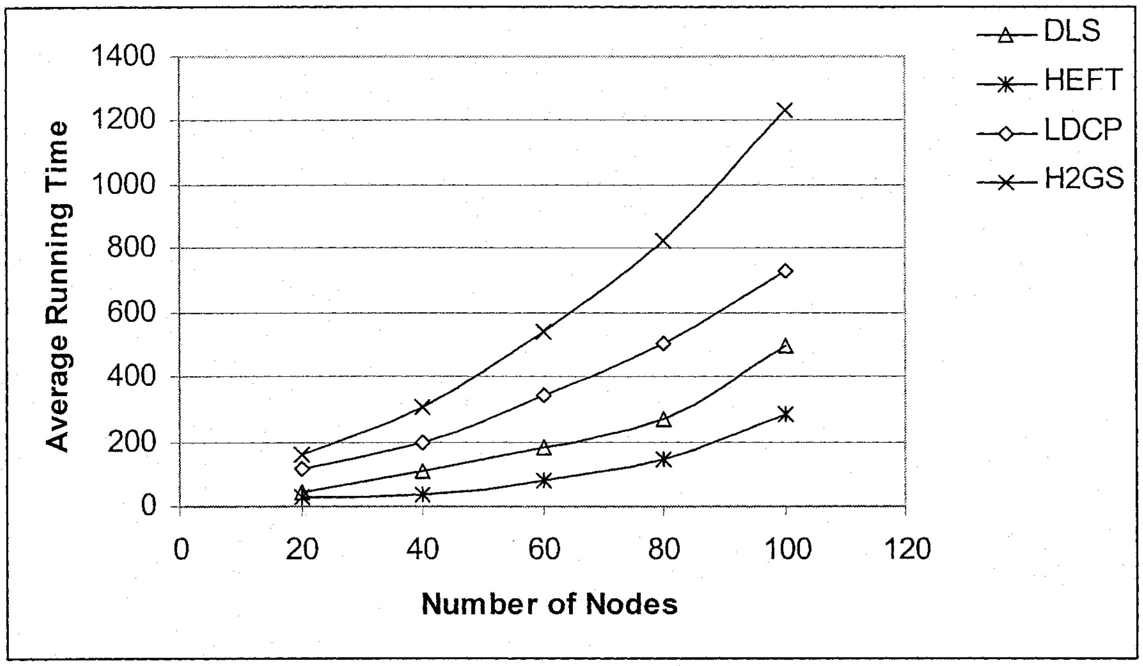


Figure 35. Average running time with respect to DAG size

Chapter 7: Conclusion and Future Work

7.1 Conclusion

Task scheduling algorithms are essential for obtaining high performance in heterogeneous computing systems. In general, static task scheduling is defined as the process of allocating the tasks of an application (as represented by a Directed Acyclic Graph or DAG) to a network of processors, and arranging the execution of these tasks in order to minimize the completion time of the application. Static task scheduling has been shown to be a NP-complete problem, and several algorithms have been proposed to deal with it. However, most of the available algorithms are developed for homogeneous computing systems.

Scheduling algorithms are classified into three main categories: heuristic algorithms, guided random algorithms and hybrid algorithms. Heuristic scheduling algorithms find solutions by moving from one point in the search space to another following a particular rule. Such algorithms find near-optimal solutions in less than polynomial time. However, these algorithms search some paths in the search space, and ignore others.

Guided random search algorithms combine the exploitation of previous results with the exploration of new areas in the search space. Genetic Algorithms (GAs) are the guided random search algorithms most widely used for task scheduling. GA-based scheduling algorithms work on a population of candidate solutions in parallel. Hence, this reduces the probability of prematurely converging to a locally optimum solution. However, the execution time of such algorithms is higher than that of the heuristic algorithms. Hybrid scheduling algorithms combine both heuristic scheduling algorithms and GAs. Such algorithms find high quality schedules.

Most heuristic scheduling algorithms for distributed computing systems belong to the list-based scheduling class. In list-based scheduling heuristics, each task is assigned a priority and then inserted into a list of waiting tasks. Then, the unscheduled task with the highest priority is selected and assigned to the most suitable processor. This process continues until all the tasks in the waiting list are scheduled. The performance of list-based scheduling algorithms is critically dependant on the method used to assign priorities to tasks.

The Critical Path (CP) attribute provides an efficient method for assigning priorities to tasks. However, when two tasks are scheduled on the same processor, the communication cost between them is zero. Hence, the CP changes dynamically during the scheduling process. The DCP, introduced by Kwok *et al.*, overcomes this problem in homogenous computing systems. It considers the cancellation of communication costs between tasks scheduled on the same process.

In Distributed Heterogeneous Computing System (DHECS), the various computational costs of the same task on different processors present a problem when the DCP is applied.

The DCP computed using the computation costs of tasks on a particular processor may differ from those computed using the computation costs on other processors.
In this report, a novel hybrid scheduling algorithm called the Hybrid Heuristic-Genetic Scheduling (H2GS) algorithm is introduced. The H2GS algorithm efficiently optimizes task scheduling in DHECSs. It combines two algorithms to produce a high quality schedule. First, the H2GS algorithm runs a list-based scheduling heuristics, called the Longest Dynamic Critical Path (LDCP) algorithm. The LDCP algorithm generates a near-optimal schedule. The LDCP algorithm employs a new attribute, called LDCP, to calculate task priorities. This attribute identifies a set of tasks and edges that play an important role in determining the provisional schedule length. At each scheduling step, the LDCP algorithm selects the most important task for scheduling. Next, the insertion-based scheduling policy is used to schedule the selected tasks on the processor that minimizes its finish execution time.

A Genetic Algorithm, called Genetic Algorithm for Task Scheduling (GATS), accepts the schedule generated by the LDCP algorithm as its own input. The GATS algorithm uses the LDCP-generated schedule to create its first population. Next, the GATS algorithm evolves this population to find better optimal (or near-optimal) schedules. The insertion of the LDCP-generated schedule to the first population enables the GATS algorithm form searching around an area in the search space that is close to or includes the optimal schedule. Moreover, the GATS algorithm employs a set of genetic operators that are specifically designed for the task scheduling problem. These operators search the search space efficiently. Hence, the time required by the GATS algorithm to find an optimal or near-optimal schedule is minimized.

The performance of the H2GS algorithm is compared to two of the best list-based scheduling algorithms for DHECSs. These two techniques are called the HEFT algorithm and the DLS algorithm. The H2GS, HEFT and DLS algorithms are simulated. A set of 500 randomly generated DAGs with various characteristics, are generated and used as the workload for evaluating the algorithms. This workload is then run on 4 different DHECSs.

The LDCP algorithm outperforms both the HEFT and DLS algorithms. In terms of schedule length, the average NSL of the LDCP algorithm is shorter than that of the HEFT and DLS algorithms by 4.8% and 2.1%, respectively. In terms of efficiency, the average speedup gained by the LDCP algorithm is higher than that gained by the HEFT and DLS algorithms by 5.7% and 2.7%, respectively.

The H2GS algorithm (including GATS) enhances the results gained by the LDCP algorithm, on its own. The performance of the H2GS shows a significant improvement over both the HEFT and DLS algorithms. The average normalized schedule length (NSL) achieved by the H2GS algorithm is better than that of the HEFT and DLS algorithms by 8.8% and 6.2%, respectively. Moreover, the average Speedup gained by the H2GS algorithm is greater than that gained by the HEFT and DLS algorithms by 8.3% and 5.3%, respectively.

7.2 Future Work

As part of our future work, we plan to investigate the scheduling of real-time applications to a DHECS. Such applications include: Gaussian Elimination, Fast Fourier Transformation and Molecular Dynamics Code.

Scheduling algorithms assume usually fully connected networks of processor. However, this assumption cannot be met by all distributed computing environments. To address this problem, we plan to extend the proposed algorithm to partially-connected networks of heterogeneous processors.

The memory and disk access speed improvements are significantly lagged behind advancements of CPU speed. Hence, this increases the penalty of data access operations, such as page faults and I/O operations, relative to normal CPU operation. To improve the performance of distributed computing systems, the scheduling algorithm must consider the available memory and I/O resource on each processing unit. Moreover, the scheduling algorithm should consider the data access requirements of the distributed application.

References

[1] H. Topcuoglu, S. Hariri, and M.Y. Wu, *"Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing"*, IEEE Trans. Parallel and Distributed Systems, Vol. 13, No. 3, pp. 260-274, March 2002.

[2] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke,*" Impact of Workload and System Parameters on Next Generation Cluster Scheduling Mechanisms"*, IEEE Trans. Parallel and Distributed Systems, Vol. 12, No. 9, pp. 967-985, September 2001.

[3] A. Zomaya, C. Ward, and B. Macey, "*Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues*", IEEE Trans. Parallel and Distributed Systems, Vol. 10, No. 8, pp. 795-812, August 1999.

[4] S. Bansal, P. Kumar, and K. Singh, *"An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor systems"*, IEEE Trans. Parallel and Distributed Systems, Vol. 14, No. 6, pp. 533-544, June 2003.

[5] Y.K. Kwok and I. Ahmad, "*Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors"*, ACM Computing Surveys, Vol. 31, No. 4, pp. 406-471, December 1999.

[6] B. Kruatrachue, *"Static Task Scheduling and Grain Packing in Parallel Processing Systems"*, PhD thesis, Department of Computer Science, Oregon State University, 1987.

[7] Y.K. Kwok and I. Ahmad, "*Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors"*, IEEE Trans. Parallel and Distributed Systems, Vol. 7, No. 5, pp. 506-521, May 1996.

[8] M. Grajcar, "*Strengths and Weaknesses of Genetic List Scheduling for Heterogeneous Systems*", Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01), IEEE 2001.

[9] A.Y. Zomaya and Y.H. Teh, *"Observations on Using Genetic Algorithms for Dynamic Load Balancing"*, IEEE Trans. Parallel and Distributed Systems, Vol. 12, No. 9, pp. 899-911, September 2001.

[10] Y.K. Kwok and I. Ahmad, "*Link Contention-Constrained Scheduling and Mapping of Tasks and Messages to a Network of Heterogeneous Processors*", Parallel Processing, 1999 Proceedings, 1999 International Conference on, IEEE.

[11] H. El-Rewini, T.G. Lewis, and H.H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Englewood Cliffs, New Jersey: Prentice Hall, 1994.

[12] G.C. Sih, and E.A. Lee, *"A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures",* IEEE Trans. Parallel and Distributed Systems, Vol. 4, No. 2, pp. 175-187, February 1993.

[13] H. El-Rewini and T.G. Lewis, *"Scheduling Parallel Program Tasks onto Arbitrary Target Machines"*, J. Parallel and Distributed Computing, Vo. 9, pp. 138-153, 1990.

[14] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[15] J. Liou and M.A Palis, *"A Comparison of Genetic Approaches to Multiprocessor Scheduling"*, Proceedings. Int'l Parallel Processing Symp, pp. 152-156, 1997.

[16] I. Ahmad and Y.K. Kwok, *"On Exploiting Task Duplication in Parallel Program Scheduling", IEEE Trans. Parallel and Distributed Systems"*, Vol. 9, No. 9, pp. 872-892, September 1998.

[17] Y.C. Chung and S. Ranka, *"Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors"*, Processing Supercomputing, pp. 512-521, November 1992.

[18] M.Tan, H.J. Siegal. J.K. Antonio, and Y.A. Li, "*Minimizing the Application Execution Time Through Scheduling fs Subtasks and communication Traffic in a Heterogeneous Computing System*", IEEE Trans. Parallel and Distributed Systems, Vol. 8, No. 8, pp. 857-871, August 1997.

[19] V. Sarkar, *"Partitioning and Scheduling Parallel Programs for Multiprocessors"*, MIT Press, Cambridge, MA, 1989.

[20] M. Wu and D. Dajski, *"Hypertool: A Programming Aid for Message Passing Systems"*, IEEE Trans. Parallel and Distributed Systems, Vol. 1, pp. 330-343, July 1990.

[21] B. Hamidzadeh, L.Y. Kit, and D.J. Lilja, *"Dynamic Task Scheduling Using Online Optimization"*, IEEE Trans. Parallel and Distributed Systems, Vol. 11, No. 11, pp. 1151-1163, November 2000.

[22] E.G. Coffman, *Computer and Jop-Shop scheduling Theory*, New York: Wiley, 1976.

[23] M.W. Schaffter, "*Scheduling Jobs with Communication Delays: Complexity Results and Approximation Algorithms*", PhD thesis, Technical University of Berlin, Germany, 1996.

[24] B. Kuatrachue and T.G. Lewis, *"Grain Size Determination for Parallel Processing"* IEEE Software, pp. 23-32, January 1988.

[25] A. Munier and C. Hanen, "*Using Duplication for Scheduling Unitary Tasks on m Processors with Unit Communication Delays*", Theortical Computing Science, 1997.

[26] D.E. Goldberg, *Genetic Algorithm in search, Optimization, and Machine Learning.* Reading Mass.: Addison-Wesley, 1989.

[27] M. Srinivas and L.M. Patnaik, "*Genetic Algorithms: A Survey*", Computer, Vol. 27, pp. 17-26,1994.

[28] M. Grajcar. *"Genetic List Scheduling Algorithm for Scheduling and Allocation on a Loosely Coupled Heterogeneous Multiprocessor System"*, Proceedings of the 36th ACM/IEEE conference on Design automation, pp. 280 – 285, 1999.