T.C.
MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES IN
PURE AND APPLIED SCIENCES

TASK SCHEDULING TECHNIQUES FOR ARBITRARY NETWORK TOPOLOGIES

Ali Fuat ALKAYA

(141027199800105)

136745

THESIS

FOR THE DEGREE OF MASTER OF SCIENCE

IN

COMPUTER ENGINEERING PROGRAMME

SUPERVISOR
Asst. Prof. Haluk TOPÇUOĞLU

İSTANBUL 2003

MARMARA UNIVERSITY

THE INSTITUTE FOR

GRADUATE STUDIES IN PURE AND APPLIED SCIENCES

ACCEPTANCE AND APPROVAL DOCUMENT

TASK SCHEDULING TECHNIQUES FOR ARBITRARY NETWORK TOPOLOGIES

Established committee listed below, on 27.01.2003 and 2003/02-31. by the *INSTITUTE FOR GRADUATE STUDIES IN PURE AND APPLIED SCIENCES*' Executive Committee, have accepted Mr. Ali Fuat ALKAYA's Master of Science thesis, titled as "Task Scheduling Techniques For Arbitrary Network Topologies" in Computer Engineering.

COMMITTEE

Advisor :Asst. Prof. Dr. Haluk TOPÇUOĞLU

Member : Prof. Dr. M. Akif EYLER

Member : Asst. Prof. Dr. Mert DEMİR

Date of thesis' before the committee : 06.02.2003

APPROVAL

Mr. Ali Fuat ALKAYA has satisfactorily completed the requirements for the degree of Master of Science in Computer Engineering at Marmara University.

Mr. Ali Fuat ALKAYA is eligible to have the degree awarded at our convocation on 27.2.2003, 2003/4-50.... Diploma and transcripts so noted will be available after that date.

Istanbul

DIRECTOR

Prof. Dr. Adnan AYDIN
MÜDÜR

ACKNOWLEDGEMENT

I would like to thank my advisor Asst. Prof. Dr. Haluk TOPÇUOĞLU for his support with an endless patience, to my colleagues in the Computer Engineering Department for their motivation in my thesis.

Ali Fuat ALKAYA

January 2003

TABLE OF CONTENTS

PAGE NO

ACKNOWLEDGEMENTS..... I

TABLE OF CONTENTS..... II

ÖZET..... V

ABSTRACT..... VI

LIST OF SYMBOLS..... VII

ABBREVIATIONS..... IX

LIST OF FIGURES..... X

LIST OF TABLES..... XIII

PART I. INTRODUCTION AND OBJECTIVES 1

PART II. GENERAL BACKROUND 3

II.1 MODEL DEFINITIONS..... 3

II.1.1 Application Model..... 3
II.1.2 System Model..... 4

II.2. THEORETICAL ASPECTS 5

II.2.1. Problem Definition And NP-Completeness..... 5
II.2.2. Categorization of Scheduling Algorithms 7
II.2.3. Communication Scheduling 11
II.2.4. Message Routing and Transfer Methods 12

II.3. RELATED WORK FOR COMPARISON STUDY..... 16

II.3.1. Dynamic Level Scheduling Algorithm..... 16
II.3.2. Bubble Scheduling and Allocation Algorithm..... 19
II.3.2.1 Algorithm Definition..... 19
II.3.2.2 Algorithm Modifications and Improvements..... 23

PART III. THESIS STUDY..... 27

III.1. MODEL GENERATORS..... 27

III.1.1. Random Graph Generator 27
III.1.2. Random Topology Generator..... 31

III.2 PROPOSED ALGORITHM..... 36

PART IV. EXPERIMENTAL WORK AND RESULTS..... 40

IV.1 COMPARISON CRITERIA..... 40

IV.2 COMPARISON TESTS..... 41

IV.2.1. Performance with respect to task heterogeneity..... 41
IV.2.2. Performance with respect to processor heterogeneity..... 45
IV.2.3. Performance with respect to link heterogeneity..... 48
IV.2.4. Performance with respect to Number of Nodes..... 50
IV.2.5. Performance with respect to CCR..... 51
IV.2.6. Performance with respect to out degree..... 54
IV.2.7. Performance with respect to Shape..... 56
IV.2.8. Performance with respect to Processor Connectivity..... 57
IV.2.9. Performance with respect to Number of Processors..... 58

PART V. CONCLUSION..... 60

REFERENCES **62**

RESUME **64**

ÖZET

DÜZENSİZ AĞ YAPILARI İÇİN GÖREV PLANLAMA TEKNİKLERİ

Bu tez, görev planlaması problemine yönelik yeni bir algoritma geliştirmeyi amaçlamıştır. Görev planlaması problemi paralel uygulamaları işlemciler üzerine planlayarak toplam hesaplama zamanını minimize etmeyi amaçlamaktadır. Bu çalışmada uygulamadaki görevlerin heterojen olması yanında işlemci bağlantılarının da heterojen olması düşünülmüştür. İşlemcilerin düzensiz olarak bağlanmış olarak düşünülmesi de önemli bir özelliktir ki, çok az programlama sisteminde her işlemci arasında bağlantı bulunur. Bununla beraber, bağlantılar üzerinde aynı anda tek mesajın iletilmesi olan zaman çakışması da göz önüne alınmıştır. Bu özelliklerle beraber, bu model gerçek hayatta karşılaşılabilecek bütün durumları simüle etmektedir. Ayrıca, rastlantısal uygulama ve işlemci sistemi üreten parçalar da sunulmaktadır. Uygulama görevlerini işlemcilere çoğu durumda iyi bir çalışma süresi ve çok daha iyi performanslarla planlayan bir algoritma geliştirilmiştir. Geliştirilen algoritmanın adı Çakışma Gözetimli Heterojen Görev Planlaması'dır. Algoritma heterojen görevleri heterojen işlemciler üzerine planlayabildiği gibi mesajları heterojen karakterli bağlantılar üzerine çakışma gözeterek planlamaktadır. Algoritma daha önceden öne sürülen BSA ve DLS algoritmaları ile karşılaştırılmıştır. Görülmüştürki çoğu test çalışmasında daha iyi sonuçlar vermiştir. Yol bulma algoritması olarakta Dijkstra algoritması kullanılmıştır. Bununla beraber, mesaj gonderme tekniklerini de karşılatıran daha geniş bir çalışma yapılmıştır. Deneylerimiz gerçek hayatta kullanılan doğrudan ve paketler halinde mesaj gönderme tekniklerini de içermektedir. Bütün çalışmalar C programlama dilinde yazılıp Linux/Unix ortamında gcc ile derlenip simüle edilmiştir. Çeşitli özelliklerdeki uygulama ve işlemci sistemleri simüle edilip algoritmalar karşılaştırılmıştır. Karşılaştırma için kullanılan iki kriter algoritmaların çalışma zamanı ve normalize edilmiş performanslarıdır.

Ali Fuat ALKAYA

Ocak 2003

ABSTRACT

TASK SCHEDULING TECHNIQUES FOR ARBITRARY NETWORK TOPOLOGIES

In this study, it is targeted to develop a new algorithm for the problem of task scheduling. Task scheduling problem is defined as scheduling parallel application tasks onto processors with the aim of minimizing the makespan. In this study, heterogeneity of application graphs as well as the heterogeneity of processor connection links are introduced. Considering the arbitrary connection of processors is worth to notice since very few scheduling systems consider arbitrary connected topologies. We also consider contention-aware scheduling on links where they transfer a unique message at a time. With these properties introduced and defined, our model simulates all of the attributes that a real-world system might have. We also propose random graph generators for application graphs and processor topology graphs. We introduced a new algorithm for scheduling application graphs to processor systems in a considerable amount of time complexity with much better performance in most situations. The algorithm we propose is called Contention Aware Heterogeneous Task Scheduling. The algorithm has the capability of scheduling heterogeneous tasks onto heterogeneous processors as well as the scheduling messages onto links with heterogeneous characteristics by considering the contention. The algorithm is compared with formerly proposed BSA and DLS algorithms. It was observed that our algorithm outperformed the related work in most of the test cases. The routing algorithm inside our proposed algorithm uses Dijkstra's algorithm for finding a path to route the messages. We also propose an extensive study for comparing these algorithms using various message transfer methods. Our experiments include cut-through and store-and-forward message transfer methods, which are widely used in real-world environments. We simulated application graphs and processor systems with several characteristics and compared the algorithms. Two of the criteria we compared the algorithms are the time complexity and their normalized performance.

January 2003 Ali Fuat ALKAYA

LIST OF SYMBOLS

$\mathbf{G_1}$	:Represents the Application Graph.
$\mathbf{T_i}$	:Represents the ith task in the application graph.
V	:Represents the set of v tasks(nodes).
E	:Represents the set of edges (messages) between tasks.
$e_{i,j} \in E$	:Represents the precedence constraint such that task T_i should complete its execution before task T_j starts.
w_i^B	:Represents the execution time required by T_i on a dedicated reference processor in a heterogeneous computing system.
$c_{i,j}^B$	:Represents the base communication cost associated with the edge *(i,j)*, which is for transferring data from task T_i to task T_j.
$\mathbf{G_2}$	:Represents the computing system as an undirected graph.
$\mathbf{P_j}$	:Represents the jth processor in the computing system.
P	:Represents the set of p processors which are connected through an arbitrary network.
L	:Represents the set of links (connections) between processors.
$l_{m,n} \in L$	:Represents the full duplex link between processors P_m and P_n.
$H_{i,k}^P$	:Represents the processor heterogeneity factor of processor P_k for task T_i,
$w_i^B \times H_{i,k}^B$	:Is the computation cost of task T_i on processor P_k (given in seconds).
$H_{x,y}^L$	:Is the link heterogeneity factor,
***tlevel*(T_i)**	The *tlevel* of a task T_i is the length of a longest path from the entry task to task T_i (excluding T_i).
***blevel*(T_i)**	:The *blevel* of a task T_i is the length of a longest path from T_i to an exit task.
num_node	:Stands for the number of nodes (tasks) that the application graph will be formed of.
shape	:Determines the shape, i.e. width and height of the application graph.
depth	:Height or number of levels of an application graph
width	:Number of nodes on each level
out_degree	:Symbolizes the maximum number of children that one task may have.
numpe	:Tells the number of processors that the processor network will contain.
top_type	:Represents the topology type of the system.
consistent	:Consistency among of the processor heterogeneity values
$DL(T_i, P_j, \sum(t))$	: Represents Dynamic Level which reflects how well task T_i and processor P_j are matched at state $\sum(t)$,
$DA(T_i, P_j, \sum(t))$	:Is the earliest time that all data required by task T_i is available at processor P_j at state $\sum(t)$.
M_{ij}	:Denotes the message (communication edge) from task T_i to T_j.
$MST(M_{ij}, L_{xy})$	:Denotes the start time of a message from T_i to T_j on a communication link L_{xy}.
$MFT(M_{ij}, L_{xy})$	:Denotes the finish time of a message from T_i to T_j on a communication link L_{xy}.

$ST(T_i,P_x)$:Is the start time of a task T_i on a processor P_x.
$FT(T_i,P_x)$:Is the finish time of a task T_i on a processor P_x.
$DRT(T_j,P_y)$:Represents the *data ready time* of task T_j on processor P_y.
$VIP(T_i)$:Denotes the corresponding predecessor *(very important predecessor)* which sends the latest message to task T_i.

ABBREVIATIONS

DAG : Directed Acyclic Graph

DLS :Dynamic Level Scheduling algorithm

HEFT :Heterogeneous Earliest Finish Time

BSA :Bubble Scheduling Algorithm

BNP :Bounded Number of Processors

UNC :Unbounded Number of Clusters

TDB :Task Duplication Based

APN :Arbitrary Processor Network

CP : Critical Path

HLFET: Highest Level First with Estimated Times

ISH : Insertion Scheduling Algorithm

MCP : Modified Critical Path

ETF : Earliest Time First

MH : Mapping Heuristic

RGG : Random Graph Generator

CCR : Communication to Computation Ratio

RTG : Random Topology Generator

IB : In-Branch nodes

OB : Out-branch nodes.

CAHTS: Contention Aware Heterogeneous Task Scheduling

EFT : Earliest Finish Time

SLR : Schedule Length Ratio

LIST OF FIGURES

PAGE NO

Figure II.1: An Application Graph with ten Nodes 3

Figure II.2: A system model with 5 processors forming a ring..... 4

Figure II.3: Pseudo-code for *blevel* calculation..... 9

Figure II.4: Pseudo code for *tlevel* calculation..... 10

Figure II.5 : An example execution for Dijkstra..... 16

Figure II.6: Pseudo-code for algorithm DLS [1]..... 18

Figure II.7: Pseudo-code for algorithm BSA [14]..... 20

Figure II.8: Pseudo-code for procedure *serialization* process [14]..... 21

Figure II.9 : Pseudo-code for procedure *computeMFT()*[14]..... 22

Figure II.10: Pseudo-code for procedure *computeDRT()*[14]..... 23

Figure II.11: Pseudo-code for procedure *computeFT()*[14]..... 23

Figure II.12: An example to the temporary cancellation of messages..... 24

Figure II.13: An example graph and its schedule for rearrangement phase .. 25

Figure III.1.: Pseudo-code for random graph generator..... 27

Figure III.2: Procedure *set_depth()*..... 28

Figure III.3: Procedure *generate_nodes()*..... 29

Figure III.4: Procedure *generate_edges()*..... 30

Figure III.5: Procedure *initialize_graph()*..... 30

Figure III.6: Procedure *save_graph_in_a_file()*..... 31

Figure III.7: Pseudo-code for random topology generator..... 32

Figure III.8: Procedure *define_topology_and_links()*..... 33

Figure III.9: Procedure *build_topology()*..... 34

Figure III.10: Procedure *save_topology_in_a_file()*..... 34

Figure III.11: Procedures *check_connectivity() and mark(processor p)*..... 35

Figure III.12: Examples to five different topology types..... 36

Figure III.13: Pseudo–code for algorithm CAHTS 38

Figure III.14: The procedure of *schedule_communication*(P_x,P_y,T_i,T_j)..... 38

Figure IV.1: The manager program..... 41

Figure IV.2.a,b: Comparison results with respect to task heterogeneity. (cut-through implementation, fully connected topology)..... 43

Figure IV.3.a,b: Comparison results with respect to task heterogeneity. (store-and-forward, implementation, fully connected topology) 43
Figure IV.4.a,b: Comparison results with respect to task heterogeneity. (cut-through implementation, arbitrarily connected topology)..... 43
Figure IV.5.a,b: Comparison results with respect to task heterogeneity. (store-and-forward, implementation, arbitrarily connected topology) .. 44
Figure IV.6.a,b: Comparison results with respect to task heterogeneity. (hypercube topologies) 44
Figure IV.7.a,b: Comparison results with respect to task heterogeneity. (ring topologies) 45
Figure IV.8.a,b: Comparison results respect to task heterogeneity. (linear topologies) 45
Figure IV.9.a,b: Comparison results with respect to processor heterogeneity. (cut-through implementation, fully connected topologies) 46
Figure IV.10.a,b: Comparison results with respect to processor heterogeneity. (store-and-forward imp., fully connected topologies) 47
Figure IV.11.a,b: Comparison results with respect to processor heterogeneity. (arbitrarily connected topologies) 47
Figure IV.12.a,b: Comparison results with respect to processor heterogeneity. (hypercube topologies) 48
Figure IV.13.a,b: Comparison results with respect to processor heterogeneity. (ring topologies) 48
Figure IV.14.a,b: Comparison results with respect to processor heterogeneity. (linear topologies) 48
Figure IV.15.a,b: Comparison results with respect to link heterogeneity. (cut-through implementation, fully connected topologies)..... 50
Figure IV.16.a,b: Comparison results with respect to link heterogeneity. (store-and-forward implementation, fully connected topologies) 50
Figure IV.17.a,b: Comparison results with respect to number of nodes. (store-and-forward implementation, fully connected topologies) 52
Figure IV.18.a,b: Comparison results with respect to number of nodes. (cut-through implementation, fully connected topologies) 52
Figure IV.19.a,b: Comparison results with respect to CCR. (fully connected topologies) 53

Figure IV.20.a,b: Comparison results with respect to CCR. (on arbitrarily connected topologies) 53
Figure IV.21.a,b: Comparison results with respect to CCR. (hypercube topologies with degree 3) 54
Figure IV.22.a,b: Comparison results with respect to CCR. (ring topologies) 54
Figure IV.23.a,b: Comparison results with respect to CCR. (linear topologies) 54
Figure IV.24.a,b: Comparison results with respect to out degree. (fully connected topologies) 55
Figure IV.25.a,b: Comparison results with respect to out degree. (arbitrarily connected topologies) 56
Figure IV.26.a,b: Comparison results with respect to shape value. (cut-through implementation, fully connected topologies)..... 57
Figure IV.27.a,b: Comparison results with respect to shape value. (store-and-forward implementation, arbitrarily connected topologies) 57
Figure IV.28.a,b: Comparison results with respect to degree value. (arbitrarily connected topologies) 58
Figure IV.29: Comparison result with respect to number of processors. (arbitrarily connected topologies) 58

LIST OF TABLES

PAGE NO

Table IV.1: The values of the input parameters in study of suite 1..... 42

Table IV.2: The values of the input parameters in study of suite 2..... 46

Table IV.3: The values of the input parameters in study with respect to link heterogeneity..... 49

Table IV.4: The values of the input parameters in study with respect to number of nodes. 51

Table IV.5: The values of the input parameters in study with respect to CCR.. 52

Table IV.6: The values of the input parameters in study with respect to out degree. 55

PART I: INTRODUCTION AND OBJECTIVES

Set of resources interconnected with a high-speed network provides a new computing platform called heterogeneous computing system, which can support executing computationally intensive applications. In such a system different types of workstations and personal computers (possibly with different processor speeds) can be interconnected which brings the 'heterogeneous' term into the definition.

Scheduling tasks of an application on this heterogeneous computing system is an important aspect of distributed computing where the scale of the system and resource heterogeneity yields large number of scheduling choices. Such an application can be composed of relatively coarse-grained computations interconnected by data dependencies.

The task-scheduling problem can be defined as assigning the tasks on appropriate resources of processing elements and sequencing the execution of the tasks. The goal of the problem is to minimize the schedule length or *makespan* of the application by preserving the task precedence constraints created by data dependencies.

Commonly there are two forms of the scheduling problem: *static* and *dynamic*. By *static*, it is meant that the characteristics of an application are known a priori, including a fairly accurate estimate of task execution times on processors, data dependencies of the application tasks. An application, therefore, can be represented by a node- and edge-weighted *directed acyclic graph* (DAG), in which the node weights represents the task execution times and edge weights represents the data dependencies as well as communication time between the tasks. On the other hand, in *dynamic* scheduling we have very little assumptions about the application and so the scheduling decisions must be made *on the fly*. Our study covers only *static* task scheduling.

Since the problem is proven to be NP-Complete except for a few highly simplified cases, there are many studies that propose various heuristics addressing to solve the problem [1-16]. However, most of these solutions are built on simplifying assumptions both about the application and target heterogeneous computing system and so not useful in practical solutions. Some simplifying assumptions in these studies are uniform task execution times, zero inter-task communication time, no data dependency among the tasks, contention-free communication, and full connectivity of parallel processors.

To be applicable in a practical environment, a scheduling algorithm needs to address a number of challenging issues. It should exploit parallelism by identifying the task graph structure, and then take into consideration task granularity, load balancing arbitrary task weights and communication costs, the number of target processors available, and the scheduling of inter-processor communication. For example, to model link contention, messages have to be properly scheduled to the communication links based on appropriate message scheduling techniques. To properly handle message scheduling, the processor network topology should also be considered and a possible routing algorithm should be exploited. To give good performance while taking into account all of these aspects can produce a scheduling algorithm with very high time complexity. High complexity in turn limits scalability. For instance, many heuristics have been evaluated with small task graphs and thus their scalability is not known. It is not clear whether they can yield good solutions within reasonable time for large task graphs found in many real world problems. On the other hand a scheduling algorithm should be fast to be of practical use. These conflicting requirements make the design of an efficient scheduling algorithm more challenging. To effectively harness the computing power of parallel architectures, optimized scheduling of the application tasks to processors is crucial. Indeed, if the tasks of an application are not scheduled properly, the application is likely to produce a poor speedup that causes the parallelization to be useless.

The aim of this study is to propose new task scheduling algorithms that are designed to cope with all of the challenges above and consider link contention, message routing, support for arbitrarily connected networks, heterogeneous task execution and heterogeneous link capacities. Our scheduling algorithm is compared with formerly proposed algorithms and their performance results are shown. We also studied on the two message transfer techniques and compared and contrasted these techniques by applying them on the algorithms.

PART II : GENERAL BACKROUND

II.1 MODEL DEFINITIONS

Multiprocessors scheduling has been an active research area since the notion of parallel processing was introduced. In the following sections, we first define the DAG model, which is followed by the topology model. The models defined below are completely realistic and it is stated how each concept can be realized in real environments.

II.1.1 Application Model

An application is represented by a directed acyclic graph, $G_1 = (V, E)$, where V is the set $\{T_1, T_2, ..., T_v\}$ consisting of v tasks, and E is the set of e edges between the tasks. Each edge $e_{i,j} \in E$ represents the precedence constraint such that task T_i should complete its execution before task T_j starts. An example DAG is given below in Figure II.1.

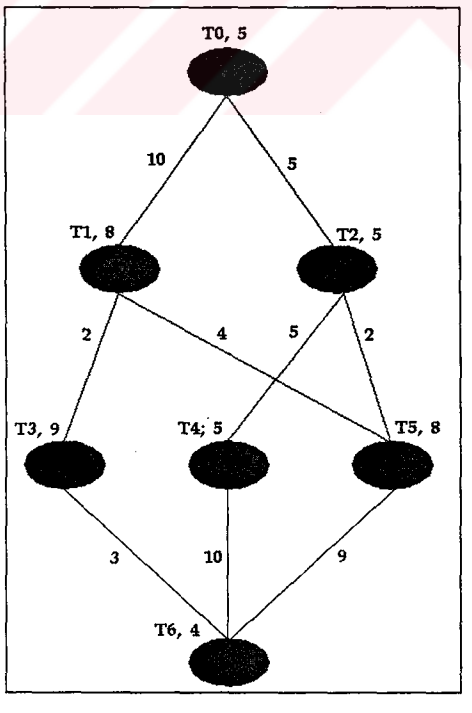


Figure II.1: An Application Graph with 7 nodes

In this model, we assume that none of the tasks is divisible. Each task T_i is associated with a base computation cost w_i^B which is the execution time required by T_i on a dedicated reference processor in a heterogeneous computing system. This base computation cost can be realized as the total number data in bytes to be processed in order to consider task T_i as finished. Similarly a base communication cost $c_{i,j}^B$ is associated with the edge *(i,j)*, which is for transferring data from task T_i to task T_j. This base communication cost can be realized as the transfer size in bytes.

In a given application graph, a task without any parent is called an *entry* task and a task without any child is called an *exit* task. Some of the task scheduling algorithms may require single-entry and single-exit task graphs. If there is more than one exit (entry) task, they are connected to a zero-cost *pseudo exit (entry) task* with zero-cost edges, which do not affect the schedule.

II.1.2 System Model

A heterogeneous computing system is represented by an undirected graph $G_2 = (P, L)$ where P is the set of p processors $\{P_1, P_2, ..., P_p\}$, which are connected through an arbitrary network. Each edge L_{mn} in L represents the full duplex link between processors P_m and P_n. Processors in this system are assumed to contain dedicated hardware so that computation can be overlapped with communication. An example system model which is a ring is shown in Figure II.2 below.

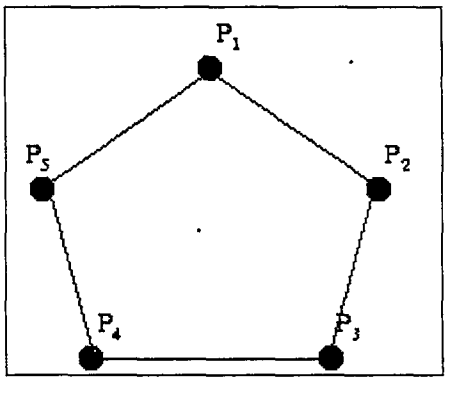


Figure II.2: A system model with 5 processors forming a ring.

The heterogeneity factors (the processor heterogeneity and link heterogeneity factor) are used to model the heterogeneity of the system. Specifically, the processor heterogeneity factor of processor P_k for task T_i, $H_{i,k}^P$, is determined by measuring the computing power

difference between P_k and the reference processor with respect to T_i. Two given processors P_x and P_y may show different relative processing capabilities for any two tasks T_i and T_j. This fact yields for the heterogeneity factor $H^P_{i,k}$ to have two indices. Then the computation cost of task T_i on processor P_k is given in seconds by $w^B_i \times H^B_{i,k}$.

Similarly, if the communication edge between the tasks T_i and T_j is scheduled to the communication link between two neighbor processors P_x and P_y, the communication cost of the edge is given by $c^B_{i,j} \times H^L_{x,y}$, where $H^L_{x,y}$ is the link heterogeneity factor. This link heterogeneity factor can be thought as the media speed between two processors P_x and P_y in *seconds/byte*. When both T_i and T_j are scheduled on the same processor, the communication cost of the edge is zero since we assume that the intra-processor communication cost is negligible when it is compared with the inter-processor communication cost. On the other hand, if a communication edge is scheduled on a route of several communication links, one of the two methods, either *cut-through* or *store-and-forward*, is used to schedule the message. These methods are mentioned in a separate part in the following sections.

II.2 THEORETICAL ASPECTS

II.2.1 Problem Definition And NP-Completeness

Scheduling problem consists of scheduling tasks of an application whose characteristics are as defined above on a group of connected processors. The goal is to minimize the finish time of the exit task such that the precedence constraints are satisfied. The problem is proven to be as NP-complete even in its simplest forms. Kwok states that there are only two special cases for which optimal polynomial time algorithms exist: scheduling tree-structured applications with identical computation costs on arbitrary number of processors, and scheduling arbitrary applications with identical costs on two processors [2-5],[12].

Here we will give two problems; namely MULTIPROCESSOR SCHEDULING and SCHEDULING TO MINIMIZE WEIGHTED COMPLETION TIME, which are the very basic versions of the task-scheduling problem. Then we will present a proof that shows us that the problem is NP-Complete. We first define PARTITION problem, one of the very basic NP-Complete problems.

PARTITION problem is proved to be NP-Complete in [3] by Garey. The problem definition is as follows.

INSTANCE: A finite set A and a "size" $s(a) \in Z^+$ for each $a \in A$.

QUESTION: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)?$$

The problem above is proved to be NP-Complete in pp. 60 in [3].

Our problem can also be defined as follows.

MULTIPROCESSOR SCHEDULING:

INSTANCE: A finite set of A "tasks", a "length" $l(a) \in Z^+$ for each $a \in A$, a number $m \in Z^+$ of "processors", and a "deadline" $D \in Z^+$.

QUESTION: Is there a partition $A = A_1 \cup A_2 \cup ... \cup A_m$ of A into m disjoint sets such that $\max\left\{\sum_{a \in A_i} l(a) : 1 \le i \le m\right\} \le D$?

PROOF:

Let us prove that MULTIPROCESSOR SCHEDULING problem is also NP-Complete by restricting it to PARTITION problem. The restriction may be performed as follows,

INSTANCE: A finite set of A "tasks", a "length" $l(a) \in Z^+$ for each $a \in A$, 2 "processors", and a "deadline" $D = \frac{1}{2}\sum_{a \in A} l(a)$.

QUESTION: Is there a partition $A = A_1 \cup A_2$ of A in 2 disjoint sets such that

$$\max\left\{\sum_{a \in A_i} l(a) : i = 1,2\right\} \le D?$$

$$\text{i.e. } \sum_{a \in A_1} l(a) = \sum_{a \in A_2} l(a) = D.$$

□.

One should note that in the proof given above the problem MULTIPROCESSOR SCHEDULING is the simplest form of the general scheduling problem. That is, it does not take into account processor heterogeneity, message scheduling and processor connectivity. Adding these to the problem makes the problem much harder to solve.

The other problem referred above, SCHEDULING TO MINIMIZE WEIGHTED COMPLETION TIME, is defined as follows:

SCHEDULING TO MINIMIZE WEIGHTED COMPLETION TIME:

INSTANCE: Set T of tasks, number $m \in Z^+$ of processors, for each task $t \in T$ a length $l(t) \in Z^+$ and a weight $w(t) \in Z^+$ and a positive integer K.

QUESTION: Is there an m-processor schedule σ for T such that the sum, over all $t \in T$, of $(\sigma(t)+l(t)).w(t)$ is no more than K?

Restricting this problem to PARTITION problem also shows that the task-scheduling problem is NP-Complete.

Since the problem requires a heuristic approach to solve, there are many heuristics proposed to solve the problem. But each heuristic has different assumptions about the problem and most of these are not as realistic as ours. A heuristic to be applicable in real situations, it must be relied and tested on realistic assumptions; otherwise it would be a specific solution that is only applicable to specific problems. We claim that our application model and system model is the most general model that takes into account all details that one can face in real world scheduling problems. Hence any algorithm tested on our model will give closest results in real world applications.

II.2.2 Categorization of Scheduling Algorithms

It is stated that there are many heuristics proposed in order to solve the task-scheduling problem. Naturally all of these heuristics address to solutions that has specific assumptions. For instance, some of them aim to solve the problem in fully connected networks, some has basic assumption of unbounded number of processors and some of them ignore the communication scheduling part of the problem.

Ahmad and Kwok categorized the task scheduling algorithms in four groups [14]. This groping can be summarized as follows:

- **Bounded Number of Processors (BNP) Scheduling**: These algorithms schedule the application graph to a bounded number of processors. The processors are assumed to be fully-connected.
- **Unbounded Number of Clusters (UNC) Scheduling**: These algorithms schedule the application graph to an unbounded number of clusters. The processors are assumed to be fully connected. The technique is also called clustering.
- **Task Duplication Based (TDB) Scheduling**: These algorithms also schedule the application graph to an unbounded number of clusters but employ task duplication technique to further reduce the completion time.

- **Arbitrary Processor Network (APN) Scheduling**: These algorithms perform the scheduling and mapping on the target architectures in which the processors are connected via a network of arbitrary topology. This is the target of our study and our new algorithms may be mentioned in this category.

We will discuss only BNP and APN groupings of these, because these algorithms have close structure to our system and share common properties of our algorithm in order to define basic attributes used.

BNP Scheduling Algorithms:

Most BNP scheduling algorithms are based on the list scheduling technique. List scheduling technique is a class of scheduling heuristics in which the nodes are assigned priorities and placed in a list arranged in a descending order of priority. The node with a higher priority will be examined for scheduling before a node with a lower priority. Two frequently used elements for assigning priority are the *tlevel* (top level) and *blevel* (bottom level. The *tlevel* of a task T_i is the length of a longest path from the entry task to task T_i (excluding T_i). Length of a path can be defined as the sum of all computation and communication costs along the path. The *blevel*(T_i) of a task T_i is the length of a longest path from T_i to an exit task. The *blevel* of a task is bounded from above by the length of a critical path. Critical path (CP) of an application graph is a longest path in the application graph. Pseudo-codes for *blevel* and *tlevel* calculation are given in Figures II.3 and II.4, respectively.

```
1.  for each task T_i
2.     T_i.visit=0
3.     T_i.tlevel=0
4.  endfor
5.  Let S be a stack set of ready tasks for blevel calculation
6.  Insert the exit node to the set S.
7.  while S is not empty do
8.     T_i=S.pop()
9.     for each parent P_j of T_i
10.          t=T_i.blevel+ w_i^B
11.          if t+c_{j,i}^B > P_j.blevel then
12.             P_j.blevel=t+ c_{j,i}^B
13.          if ++P_j.visit = P_j.num_child then
14.             S.push(P_j)
15.    Endfor
16.    T_i.blevel=T_i.blevel+  w_i^B
17. endwhile
```

Figure II.3: Pseudo-code for *blevel* calculation.

The procedure given in Figure II.3 calculates the *blevel* value for each task starting from the exit node. In the algorithm, the term $P_j.visit$ carries a flag when to insert P_j to the stack, and $P_j.num_child$ represents the number of children that P_j has.

```
1.  for each task T_i
2.     T_i.visit=0
3.     T_i.tlevel=0
4.  endfor
5.  Let S be a stack set of ready tasks for tlevel calculation
6.  Insert the entry node to the set S.
7.  while S is not empty do
8.     T_i=S.pop()
9.     t=T_i.tlevel+  w_i^B
10.    for each child P_j of T_i
11.       if t+c_{i,j}^B > P_j.tlevel then
12.          P_j.tlevel=t+c_{i,j}^B
13.       if ++P_j.visit = P_j.num_parent then
14.       S.push(P_j)
15.    endfor
16. endwhile
```

Figure II.4: Pseudo code for *tlevel* calculation.

The procedure given in Figure II.3 calculates the *tlevel* value for each task starting from the entry node. $P_j.visit$ carries a flag when to insert P_j to the stack, and $P_j.num_parent$ represents the number of parent that P_j has.

List scheduling algorithms use these *blevel* and *tlevel* definitions and/or their variations as their basis. Another definition that needs some attention is the *ready* tasks. By *ready* tasks we mean the tasks whose all predecessors are scheduled and they are ready to be scheduled.

Below is a list and short explanations of some list scheduling heuristics [14].

- MCP Algorithm: The MCP (Modified Critical Path) algorithm uses an attributed called ALAP time of a node as task priority [18].
- DLS Algorithm: The DLS (Dynamic Level Scheduling) algorithm uses as task priority an attribute called dynamic level which is the difference of a task and its earliest start time on a processor [1].
- HEFT Algorithm: The HEFT (Heterogeneous Earliest Finish Time) algorithm is a compile-time scheduling algorithm for heterogeneous processors that uses *blevel* for task priority phase. The processor selection is based on EFT (Earliest Finish Time) using insertion-based heuristic [11].

APN Scheduling Algorithms:

The algorithms in this class take into account specific architectural features such as the number of processors as well as their interconnection topology. These algorithms can schedule tasks on the processors and messages on the network communication links. There are two algorithms that are worth to mention.

- MH Algorithm: The MH (Mapping Heuristic) Algorithm first assigns priorities by computing the static *blevels* of all tasks. A ready task list is then initialized to contain all entry tasks ordered in decreasing priorities. Each task is scheduled to a processor that gives the smallest start time [8].
- BSA Algorithm: The BSA (Bubble Scheduling and Allocation) algorithm constructs a schedule incrementally by first injecting all the nodes to the pivot processor, defined as the processor with the highest degree. Then, the algorithm tries to improve the start time of each task by migrating it to one of the adjacent processor of the pivot processor if the migration can improve the start time of the task [12].

II.2.3 Communication Scheduling

As stated in previous sections except very few studies, many research activities commonly focused on the problem of task scheduling in homogeneous/heterogeneous computing systems from the computational point of view [11][1][16][7][8][9][10][13]. Nevertheless, these proposals do not consider communication cost and so assuming that the communication media has enough resource for any case. However, as the computational power of new processors raise, the interconnection network media may become the bottleneck, particularly when executing applications with huge network bandwidth requirements, like multimedia applications, video-on-demand applications, etc. Therefore, the mapping problem must be studied not only from the computation point of view but also from the communication point of view. Given a network topology system and given an application consisting of tasks defined earlier, an ideal scheduling strategy would map the tasks to processors taking into account both computational and communication requirements of the tasks.

With many scheduling algorithms communication is further assumed to be contention free meaning that multiple messages can be using the same link at the same time without affecting each other. So we can define contention as carrying only one message at one time. One can easily see that contention free scheduling is an unrealistic issue unless very high technological media is used.

In this context, Zomaya stated and proved that results quoted for schedulers which assume zero contention (the majority of research to date) are completely unrealistic [15]. Just as the relaxation of the assumption of zero communication changed the face of scheduling, so must the inclusion of link contentions in the communication model. Zomaya, in his work, assumes the target architecture to be a fully connected message passing system comprising an arbitrary number of processing elements (each with a separate I/O processor performing store-and forward routing) connected via a network of links. In the study, it is assumed that a task receives all inputs before commencing execution and then runs without preemption until completion, at which point it sends any outputs to its successors. This defined application and system model is very close to ours, so we conclude a very important result for our study and its contribution.

Zomaya defined communication delay in two components: the time required to transmit the message over an empty route, and the time sent waiting for links on the route to

become empty. The first component is a linear function of both message size and the number of individual links on comprising the route. This delay is easily calculated on a static architecture. Even though the first component is considered in some algorithms, the second component, the contention delay, is neglected in almost all algorithms. Zomaya artificially included contention delays in the final schedule such that a second pass is used to create a communication schedule and then simulate the combined schedules. This second pass uses routing tables and a time-ordered event list. Zomaya studied three basic scheduling algorithms, HLFET [16], DSH and ISH [17]. The study shows us that any schedule obtained by these heuristics that includes contention is about two or three times bigger than the schedule assuming no contention, and this difference gets bigger with more number of processors.

II.2.4 Message Routing and Transfer Methods

As we described in previous sections, scheduling of messages to communication links is as important as scheduling tasks to processors. A message from task T_i to task T_j where T_i is scheduled to P_x and T_j is scheduled to P_y needs to be routed on a path from source (P_x) to destination (P_y). In fact there are two issues to be handled when scheduling a message on communication links. The first one is to find an efficient route and the second one is how to transfer the message, i.e. handle the contention problem on the route. We examine these issues in separate sections below.

Message Transfer Methods:

The ideal situation in passing a message from a source processor to a destination processor occurs when there is a direct link between the source processor and the destination processor. In most systems and computations and especially in our study, it is often necessary to route a message through intermediate processors from source to the destination. There are two basic ways that messages can be transferred from a source to a destination: *circuit switching* and *packet switching*. We can define these concepts as follows:

Circuit switching involves establishing the path and maintaining all the links in the path for the message to pass, uninterrupted, from the source to the destination. All the links are reserved for the transfer until the message transfer is complete. A simple telephone system (not using advanced digital techniques) is an example of a circuit-switched system. None of the links can be used for other messages until the transfer is complete. We will refer this circuit switching as *cut-through* message transfer method.

In packet switching, the message is divided into "packets" of information, each of which includes the source and destination addresses for routing the packet through the interconnection network. A packet remains in a buffer if blocked from moving forward to the next processor. The mail system is an example of a packet-switched system. This form of packet switching is called *store-and-forward* packet switching. Store-and-forward packet switching enables links to be used by other packets once the current packet has been forwarded.

From now on, we will refer these two transfer method definitions as *cut-through* and *store-and-forward* message transfer. Most parallel and distributed systems utilize either store-and-forward or cut-through message transfer.

To our knowledge, we found the unique study that used and compared both store-and-forward and cut-through message transfer methods done by Adler and Rosenberg [6]. They showed the problem of transferring optimally many messages is also NP-Complete, both when messages may be buffered in transit and when they may not be.

Since the media speed between processors are heterogeneous and sending the same amount of data differs on the different links we must somehow find a method to send messages for cut-through and store-and-forward transfer methods. While implementing cut-through transfer method in our study, we reserved all links on the path according to the maximum transfer cost of these links.

Message Routing:

Message transfer is one aspect of the communication-scheduling problem. The other aspect that needs much more attention is routing the messages over a path of processors. If a network topology is not fully connected, then a logical path for a message transfer across machines might have multiple physical paths from the source machine to the destination machine. That is, if a message emerges from T_i scheduled on P_x and targets T_j scheduled on P_y, where P_x and P_y are not neighbors, obviously needs a route from source to destination. Routing these messages must be done in the most effective and efficient way that the time complexity and performance of the algorithm must give the best. In this case, the polynomial all-pairs shortest-path algorithm could be used to choose a shortest physical path that takes the least time for each message transfer.

The single-source shortest-path problem is described as follows: Given a graph (topology in our case), $G_2 = (P, L)$, we want to find a shortest path from a given source vertex (processor) $x \in P$ to every vertex (processor) $y \in P$. The algorithm for the single-source problem, including the following variants, can solve many other problems.

Single-destination shortest-paths problem: Find a shortest path to a given destination vertex (processor) t from every vertex (processor) y. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

Single-pair shortest-path problem: Find a shortest path from x to y for given vertices (processors) x and y. If we solve the single-source problem with source vertex (processor) y, we solve this problem also.

All-pairs shortest-path problem: Find a shortest path from x to y for every pair of vertices x and y. Running a single-source algorithm for once from each vertex (processor) can solve this problem.

In task scheduling problem, the last variant of the single-source shortest path problem needs to be solved because in communication scheduling process shortest paths from each processor to each processor must be known.

The concept of a shortest path deserves some explanation. One way of measuring path length is the number of hops. Another metric is the geographic distance in kilometers. However, many other metrics are also possible besides hops and physical distance. For example, each arc could be labeled with the mean queuing and transmission delay for some standard test packet as determined by hourly test runs. With this graph labeling, the shortest path is the fastest path, rather than the path with the fewest arcs or kilometers.

In the most general case, the labels on the arcs links could be computed as a function of the distance, bandwidth, average traffic, communication cost, mean queue length, measured delay, and other factors. By changing the weighting function, the algorithm would then compute the "shortest" path measured according to any one of a number of criteria, or a combination of criteria.

In our model, we have link heterogeneity factor as the link labels between processors. We explained that this heterogeneity factor could be considered as the media speed of the link. For two different message transfer techniques, we use two different functions; we use the max function if the messages are transferred via using cut-through message transfer technique because we are looking for the route that sends messages through the route using cut-through transfer method. On the other hand, if the messages are transferred by using store-and-forward message transfer technique, then the path is determined by using the summation function, because the cost of a message transfer along a path is obtained by summing the costs on each link.

Several algorithms for computing the shortest path between two processors of a topology are known. Dijkstra is the one that is most widely used in these applications. So,

we will shortly describe the Dijkstra's algorithm, with an example in Figure 5, which solves the single-source shortest paths problem on a weighted, directed graph as the same as our network topology.

There are three sets of processors at each step: The black processors show the already visited processors that are on the shortest path. The blues indicate the processors that are reached and candidate processors to be inserted in the shortest path. The red one is the chosen processor (pivot) among the candidates. Each processor is labeled (in parentheses) with its distance from the source along the best-known path (Figure 5 a.). Initially no paths are known, so all processors are labeled with infinity (Figure 5 b.). In the following subfigures (Figure 5 c-h) we select the next processor that gives the shortest path among the candidates. If a processor is selected to be inserted in the shortest path as the pivot, then the link reaching this pivot processor from the previous pivot processor is colored to dark gray in order to figure out the shortest path. Note that the label on a processor indicates the shortest distance from A to itself, and on which processor the path goes. From this figure we can obtain a routing table for A that gives the distance of the shortest path and which neighbor to use to reach A from each processor. Repeating this algorithm for each processor gives us a two-dimensional matrix by which we can easily obtain the shortest paths from each processor to each processor.

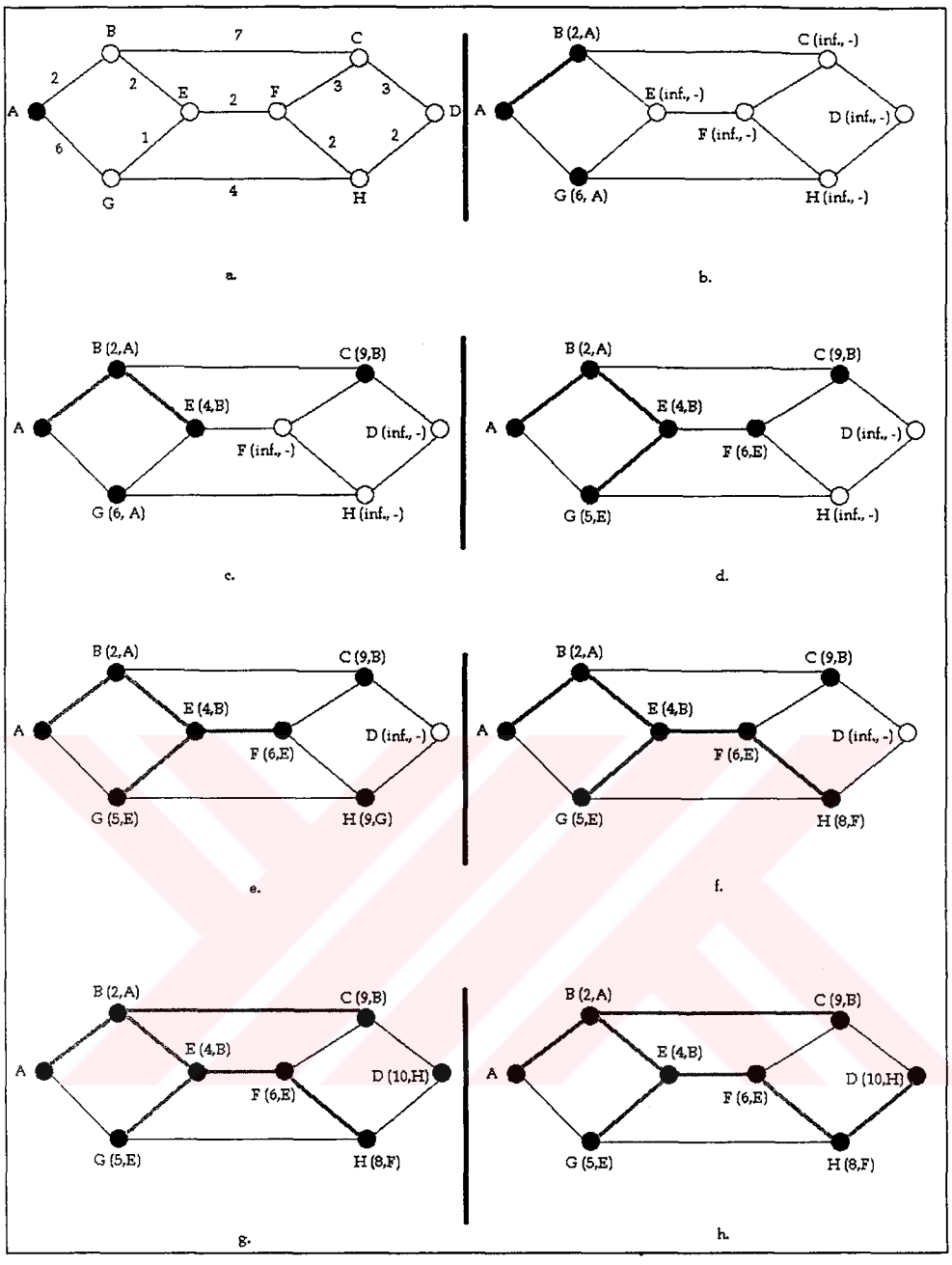


Figure II.5 : An example execution for Dijkstra.

II.3 RELATED WORK FOR COMPARISON STUDY

So far, we examined theoretical aspects of the study. We developed a new algorithm that manages to have success in the application and system model defined in previous sections. For a comparison study, we chose and implemented two algorithms proposed in former studies. These algorithms are the Dynamic Level Scheduling (DLS) and Bubble Scheduling and Allocation (BSA). They are list-scheduling algorithms shortly described in

previous sections. In this section we will briefly explain these algorithms and their superiorities and deficiencies.

It is worth to emphasize that all algorithms that are quoted by several studies to have best performance are based on the list scheduling technique. The success of this scheme originates from the accurate representation of a task's priority by its *blevel* value. Ordering the tasks by their *blevel* values in descending order, automatically establishes an order according to the precedence constraints of the graph.

By selecting the ready node with highest *blevel* for scheduling at each step, the technique successively shortens the longest remaining path to completion.

II.3.1 Dynamic Level Scheduling Algorithm

The Dynamic Level Scheduling algorithm was proposed by Sih and Lee in [1] for interconnection-constrained heterogeneous processor architectures. The DLS algorithm is list-scheduling algorithm with dynamic priorities that accounts for interconnection contention. The *dynamic level*, denoted by $DL\left(T_i, P_j, \sum(t)\right)$, reflects how well task T_i and processor P_j are matched at state $\sum(t)$, where $\sum(t)$ encompasses both the state of processing resources (previously scheduled tasks), and the state of the communication resources (previously scheduled data transfers) at global time t. They first define $DA\left(T_i, P_j, \sum(t)\right)$ to be the earliest time that all data required by task T_i is available at processor P_j at state $\sum(t)$. This quantity represents the earliest time at which all data transfers to task T_i from its predecessors can be guaranteed to be routed on their path and completed with all communication resources reserved in advance. The dynamic level is then expressed as

$$DL\left(T_i, P_j, \sum(t)\right) = blevel(T_i) - \max\left\{t, DA\left(T_i, P_j, \sum(t)\right)\right\}.$$

At each step, the *ready* node and available processor that maximize this expression are chosen for scheduling. (Here it may be necessary to repeat the definition for the term *ready*; by *ready* tasks, we mean the tasks whose all predecessors are scheduled and they are ready to be scheduled.) It is immediately understood that the instantaneous dynamic level for the same task-processor pair will change in successive scheduling steps, based on the resource availability on both processors and links.

The dynamic level has a straightforward interpretation. The maximization term represents the earliest time that task T_i can start execution on processor P_j, because the task

cannot start execution until the current time, and it cannot be invoked until all data from its predecessors has been received. So the dynamic level $DL(T_i, P_j, \sum(t))$ is the difference between the static level of task T_i and its earliest time the task can start execution on processor P_j. When considering prospective tasks for scheduling, a task with large static level is desirable because it indicates a high priority for execution. When comparing candidate processors for assignment, alter starting time is undesirable because it indicates either a busy processor or a long interval required for interprocessor communication. Processors that incur a large amount of communication scheduling are penalized by a later starting time, which lessens the dynamic level. The algorithm evaluates dynamic level over all combinations of ready tasks and available processors to find the best node-processor pair for scheduling [1].

With all these given, DLS algorithm can be summarized in a pseudo-code as in Figure II.6.

```
1.  create a set S that contains the ready tasks (initially the entry task(s) of
    the application graph.)
2.  while not all tasks scheduled do
3.    maxdl=0
4.    for each task T_i in S
5.      for each processor P_j in the topology
6.        if DL(T_i,P_j) >maxdl then
7.        maxdl= DL(T_i,P_j)
8.        select (T_i,P_j) pair as (T_m,P_n)
9.      endfor
10.   endfor
11.   schedule the selected task T_m on the selected processor P_n
      considering the scheduling of incoming messages on
      communication links, according to a selected routing algorithm.
12.   insert the tasks whose all predecessors are scheduled into the set S.
13. endwhile
```

Figure II.6: Pseudo-code for algorithm DLS [1].

As far as we figured out from the proposed paper [1], the algorithm uses cut-through message scheduling technique. That is, is a message is to be scheduled on path of communication links, the message is scheduled on the same time interval on all links through the path.

The DLS algorithm has an $O(v^3 p \times f(p))$ time complexity, where v is the number of tasks and p is the number of processors. The complexity of the function used for data routing

to calculate the earliest start time is $O(f(p))$ [1]. During our implementation of DLS for comparison study, we used Dijkstra's algorithm.

II.3.2 Bubble Scheduling and Allocation Algorithm

The BSA algorithm was first proposed by Kwok and Ahmad for homogeneous network topologies in 1995 [12]. The second version of the proposed algorithm was renewed for heterogeneous processors in 1999 [14]. Therefore we consider the second version of BSA as a comparing algorithm.

II.3.2.1 Algorithm Definition

The algorithm was proposed by claiming to use no routing algorithms for scheduling the messages on communication links in arbitrary processor networks. Kwok and Ahmad see the issue of using a routing table as a problem that is two-fold: (i) the routing table has to be pre-determined, usually using shortest-path algorithm, for the input target topology; (ii) during the scheduling process, the routing table, which has to be frequently updated, may not give optimized routes. For these reasons, BSA does not use a routing algorithm or a routing table.

In the BSA algorithm, initially, the tasks are all scheduled to a single processor that gives the shortest CP length. This processor selection was based on the criteria of highest degree in the original study in 1995 [12]. Then, each task is considered in turn for a possible migration to the neighbor processors. The objective of this process is to improve the finish times of tasks because a task migrates only if it can "bubble up". If a task is selected for a migration, the communication messages from its predecessors are scheduled to the communication link between the new processor and the original processor. After all the tasks in the original processor are considered, the first phase of scheduling completes. In the second phase, the same process is repeated on one of the neighbor processor. Thus, a task migrated from the original processor to a neighbor processor may have an opportunity to migrate again to a processor one more hop away from the original processor. This incremental scheduling by migration process is repeated for all the processor in a breadth-first fashion. The advantage of this incremental approach is that no routing algorithm is needed, because the algorithms adapts its scheduling decisions to each input topology, which may be arbitrary.

The idea of using no routing tables and migrating tasks is considered as an original idea brought into the task scheduling area.

Main algorithm for BSA and all details about the algorithm are given and explained in below figures. Even though the idea seems original, there are some confusing points both in the main algorithm and its details. We will explain and comment on the deficiencies next.

```
1.  initial pivot ← the processor that gives the shortest CP length
2.  serialization(pivot)
3.  build_processor_list(Pivot)
4.  while ProcessorList is not empty do
5.     Pivot ← remove the first processor from ProcessorList
6.     for each T_i on Pivot do
7.       if FT(T_i,Pivot)>DRT(T_i,Pivot) or VIP of T_i is not scheduled to Pivot then
8.         for each neighbor processor P_y of Pivot, compute DRT(T_i,P_y) and FT(T_i,P_y)
9.           if there is a neighbor processor P_y' such that FT(T_i,P_y')<FT(T_i,Pivot) then
10.                make T_i migrate from Pivot to P_y'
11.          else if FT(T_i, P_y')=FT(T_i,Pivot) and VIP of T_i is scheduled to P_y' then
12.               make T_i migrate from Pivot to P_y'
13.        endfor
14.    endfor
15. endwhile
```

Figure II.7: Pseudo-code for algorithm BSA [14].

The terms that take place in the procedures and main algorithm deserves some explanation in order to make the algorithm opaque.

M_{ij} denotes the message (communication edge) from task T_i to T_i.

$MST(M_{ij},L_{xy})$ denotes the start time of a message from T_i to T_i on a communication link L_{xy}.

$MFT(M_{ij},L_{xy})$ denotes the finish time of a message from T_i to T_i on a communication link L_{xy}.

$ST(T_i,P_x)$ is the start time of a task T_i on a processor P_x.

$FT(T_i,P_x)$ is the finish time of a task T_i on a processor P_x.

$DRT(T_j,P_y)$ represents the *data ready time* of task T_j on processor P_y.

$VIP(T_i)$ denotes the corresponding predecessor *(very important predecessor)* which sends the latest message to task T_i.

Pivot processor the processor from which tasks migrate to other processors.

The procedure *computeMFT()* calculates and returns the finish time of a task on a neighbor processor. Using *computeMFT()* Kwok determines the finish times of every incoming messages of the task on a neighbor processor. On the other hand using *computeFT()* one can interrogate whether a task can improve its finish time through migrating to a neighbor processor of the *pivot* processor.

```
1.  compute the t_level and b_level of each task
2.  identify the CP: if there are multiple CPs, select the one with lagest execution cost
3.  put the first CP task (entry task) to the first position of the serial order
4.  i ← 2; T_x ← the next CP task
5.  while not all the CP tasks are included do
6.     if T_x has all its predecessors in the serial order then
7.        put T_x at position i and i++
8.        else let T_y be the predecessor of T_x which is not in the serial order and has the
              largest b_level (ties are broken by choosing the predeccessor with a smaller
              t_level)
9.          if T_y has all its predecessors in the serial order then put T_y at position i and i++
              otherwise recursively include all the ancestors of T_y in the serial order such
              that the tasks with a larger b_level are included first
10.         repeat the above step until all the predecessors of T_x are in the serial order
11.         put T_x at position i and i++
12.    T_x the next CP task
13. endwhile
14. append all OB tasks to the serial order in descending order of b_level
```

Figure II.8: Pseudo-code for procedure *serialization* process [14].

The serialization process, whose pseudo-code is given in Figure 22 and which determines the order of subsequent tasks migration, is a crucial step of the algorithm. In BSA, the serialization process is centered on the term *critical path* (CP) of the application graph. As we previously defined, critical path is the set of tasks forming a path with the largest sum of execution costs and communication costs.

In the serialization process, scheduling of CP tasks and their predecessors carries a special vitality to shorten the schedule length. Kwok defines the predecessors of the CP tasks as the *in-branch* (IB) tasks. The remaining, which are neither CP task nor IB task are called *out-branch* (OB) tasks. This grouping of tasks helps us in serializing them onto the processor. One can easily elect the CP tasks by their *blevel* and *tlevel* attributes. CP tasks of an application graph are the ones whose *blevel* and *tlevel* summation is equal to the CP length.

In the serialization process, the tasks are scheduled according to their CP, IB or OB attributes to a single processor, called the *pivot* processor, which is selected as follows. The CP is constructed based on the actual execution costs of tasks on the first processor. This process is repeated for other processors and eventually the processor that gives the shortest CP length based on actual execution costs is selected as the first *pivot* processor. So it is obviously seen that BSA is also list-scheduling algorithm.

Tasks Migration:

After the application graph is serialized to the first pivot processor, tasks have to be considered for possible migration to the neighbor processor in order to improve their finish times (bubble up). To determine whether a migration is beneficial, the finish time of the task on a neighbor processor must be computed which in turn depends on the data ready time (DRT) of the task. A task's DRT can be defined as the latest arrival time of messages from its predecessors. Computing DRT also requires scheduling of messages to communication links. Kwok defined the pseudo-code in Figure II.9 for computing the finish time of a message on a communication link, which refers to pseudo-codes of procedures computing DRT and FT given in Figures II.10 and II.11 respectively.

1. Check if there exists some s such that $MST(M_{i_{s+1}j_{s+1}}, L_{xy}) - \max\{MFT(M_{i_s j_s}, L_{xy}), FT(T_i, P_z)\} \geq c^B_{i,j} \times H^L_{x,y}$

   where $s=0,1,...,k$, $MST(M_{i_{k+1}j_{k+1}}, L_{xy}) = \infty$,and $MST(M_{i_0 j_0}, L_{xy}) = 0$

2. if such s exists, use the smallest one to compute $\max\{MFT(M_{i_s j_s}, L_{xy}), FT(T_i, P_z)\} + c^B_{i,j} \times H^L_{x,y}$

   which is returned as $MFT(M_{ij}, L_{xy})$, otherwise return ∞

Figure II.9 : Pseudo-code for procedure *computeMFT()* [14].

The procedure computeMFT() has the input of a message Mij, a communication link Lxy on which k messages {Mi1j1,Mi2j2...Mikjk} have been scheduled and T(Ti,Pz) (note that Pz may be Px). With these inputs it returns the MFT(Mij,Lxy) value as an output.

```
1.    DRT(Tj,Py) =0
2.    for each predecessor Ti of Tj do
3.          let Px be the predecessor currently accomodating Ti
4.          if Px =Py then arrival-time=FT(Ti, Px)
5.          else call computeMFT and arrival-time=MFT(Mij,Lxy)
6.          if arrival-time > DRT(Tj,Py) then
7.              DRT(Tj,Py) =arrival-time
8.              VIP= Ti.
9.    endfor
```

Figure II.10: Pseudo-code for procedure *computeDRT()* [14].

The procedure computeDRT() given Figure II.10 has an the input of a neighbor processor Py, a task Tj, its set of predecessors, and their messages to Tj. The procedure returns the DRT(Tj,Py) value as an output.

1. Check if there exists some t such that $ST(T_{P_y^{t+1}}, P_y) - \max\{FT(T_{P_y^t}, P_y), DRT(T_j, P_y)\} \geq w_j^B \times H_{j,y}^B$

   where $t=0,1,...,l$, $ST(T_{P_y^{l+1}}, P_y) = \infty$,and $FT(T_{P_y^0}, P_y) = 0$
2. if such t exists, use the smallest one to compute $\max\{FT(T_{P_x^t}, P_y), DRT(T_j, P_y)\} + w_j^B \times H_{j,y}^B$

   which is returned as $FT(T_j, P_y)$, otherwise return ∞

Figure II.11: Pseudo-code for procedure *computeFT()* [14].

The procedure computeFT() given in Figure II.11 has an input a task Tj, a neighbor processor Py on which 1 tasks $\{T_{P_y^1}, T_{P_y^2}, ..., T_{P_y^l}\}$ have been scheduled and DRT(Tj,Py). The output of the procedure is the value FT(Tj,Py).

II.3.2.2 Algorithm Modifications and Improvements

In the BSA algorithm one of the weaknesses is scheduling the messages (both incoming and outgoing) of a migrated task. It considers scheduling any message on the link

between the old processor and the new processor to any slot that comes later than the finish time of the message owner parent and may be found by insertion based scheduling method.

But this brings some inconsistency since a message following a processor path of more than one processor may be scheduled to non-simultaneous time slots on the links. To overcome this inconsistency we designed a *cut through* message scheduling algorithm as defined in previous sections. So, we schedule a message, from task x to task y over a path of links having more than one processor, such that we look for and obtain a simultaneous time slots transfer the message over these links again using the insertion based approach. We also implemented the *store and forward* version of the algorithm in which data may be stored in processors and sent in consequent time slots.

Some other poor and/or ambiguous aspects are also improved as follows:

Bubbling up process: Before investigating any task for possible migrations to the neighbor processors in order to improve the finish time, its finish time on the current processor must be improved to the possible earliest time. In spite of the fact that this bubbling up process is not explicitly stated in the proposed BSA algorithm, it is intuitively stated in examples. So, we added a procedure that tries to improve the finish time before searching on other processors.

Temporary cancellation of messages from parents on same processor: Consider the following example,

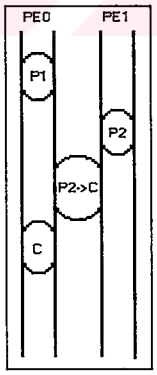


Figure II.12: An example to the temporary cancellation of messages.

Suppose that a task's (task C) finish time is being investigated to migrate from PE0 to PE1, but its parents P0 and P1 are already scheduled as in the figure. In this case computing the finish time of C without canceling the message scheduled on the link 0-1 will cause poor performance. So, we designed a procedure that cancels all messages in similar situations, and relocates cancelled messages after computing finish time.

Rearranging the remaining tasks: The proposed algorithm states that if a task's finish time does improve on a neighbor processor, say Pn, then reschedule the task to Pn and rearrange its incoming and outgoing edges (Section 2.4 pp. 9). On the other hand, we proved that rearranging only the outgoing messages is not adequate, but also we need to rearrange all the tasks that somehow have a parent-child relationship. Actually, even this is not adequate, since we must rearrange all the tasks that has a pseudo link with one of these children, that is, the tasks that is on the same processor and come after one of these children. The following example clearly proves our assertion.

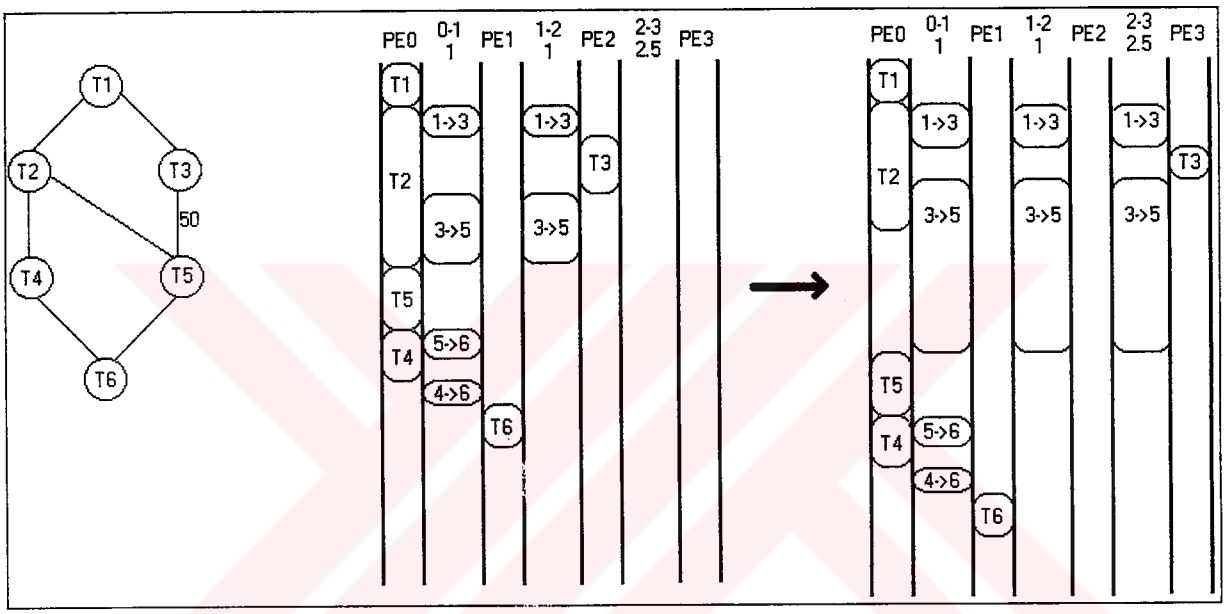


Figure II.13: An example and its output schedule for rearrangement phase.

In the example, link heterogeneity factors (say *media_speed*) of the links between the processors are 1, 1 and 2.5 units for links 0-1, 1-2 and 2-3, respectively. Because task 3 is computed to finish earlier in PE3 than in PE2, it is migrated. But the rearrangement of outgoing edges from T3 cause T4, T5 and T6 to be rearranged with their incoming and outgoing messages. The most important point here is that T4, even though has no child-parent relationship with T3, has to be rearranged in its current processor because it merely comes after one of T3's children. This important phase of the algorithm is mentioned nowhere in related studies of BSA algorithm.

This rearrangement of other remaining tasks brings another ambiguity as figured out below.

Rearranging the tasks in a sorted order: Above we proved by an example that, after a migration some tasks need to be rearranged. But this rearrangement requires an order so that precedence constraints are obeyed. For instance, rearranging T6 as the first task would cause a scheduling error. Thus we developed the idea of sorting these tasks according to their start times. This is a rational method, because in rearrangement we neither bubble up nor migrate the tasks, we only rearrange the tasks. In rearrangement, a task must be rearranged before its children and before the tasks that come after on the same processor. So, starting times of these tasks is the best indicator to list them in a topological order.

PART III : THESIS STUDY

III.1 MODEL GENERATORS

III.1.1 Random Graph Generator

A random graph generator (RGG) is implemented that gets some input and gives as output an application graph with the properties as described above. The pseudo-code for the RGG is given below in Figure III.1.

- Inputs:
  num_node, CCR, shape, out_degree, max_p_data, r_seed
- Output:
  a graph written in a text file

1. set *max_t_data* as *max_p_data*CCR*
2. set the depth value of the graph (the number of levels in the graph) by *set_depth()*
3. generate the nodes that will take place in each level by *generate_nodes()*
4. generate the edges (the messages) between the nodes by *generate_edges()*
5. obtain the weight values and communication costs randomly by *initialize_graph()*
6. save the graph in a file by *save_graph_in_a_file()*

Figure III.1.: Pseudo-code for random graph generator.

Let us figure out the definitions that are mentioned in the algorithm. The value *num_node* apparently stands for the number of nodes (tasks) that the application graph will be formed of.

CCR (communication-to-computation ratio) of the application graph which is defined as its average communication cost divided by its average computation cost on a given system. That is an application created with a high CCR value will be more communication intensive. A CCR value equal to 1 will form an application graph whose base computation and base communication values are from the same range and very close. And consequently, a low

CCR value will form computation intensive application graphs. The first statement of the algorithm *RGG* in Figure III.1 tells obviously about the relation among these three parameters.

Shape parameter is firstly defined in [11] by Topçuoğlu, which determines the shape, i.e. width and height of the application graph. We assume that the *depth* (height or number of levels) of an application graph is randomly generated from a uniform distribution with mean equal to $\sqrt{num_node} / shape$. The *width* for each level (number of nodes on each level) in an application graph is randomly selected from a uniform distribution with mean equal to $\sqrt{num_node} \times shape$. If *shape=1.0* then a balanced application graph will be created. The details can be found in Figure III.2.

Out_degree symbolizes the maximum number of children that one task may have. The number of children for each node is randomly chosen from the range [*1,out_degree*] (Figure III.4).

The base computation cost for a task T_i, w_i^B, is obtained randomly from the range [*1,max_p_data*]. Similarly a base communication cost, $c_{i,j}^B$, associated with the edge *(i,j)*, which is for transferring data from task T_i to task T_j is randomly generated from the range [*1,max_t_data*]. The other details are in Figure III.5.

Finally, *r_seed* stands for any long value that will be used to start the random processes.

Algorithm *RGG* is formed of 6 statements of which last five call procedures, namely *set_depth, generate_nodes, generate_edges, initialize_graph, save_graph_in_a_file*, are discovered in Figures III.2, III.3, III.4, III.5, and III.6 respectively.

The procedure given in Figure III.2 involves the pseudo code for procedure *set_depth()* and determines how many levels will there be in the graph by using *shape* parameter.

1. obtain a random *temp_depth* value from the distribution with mean equal to $\sqrt{num_node} / shape$
2. if *temp_depth*<3 then set *depth* to 3
3. else if *temp_depth*>*num_node* set *depth* to *num_node*
4. else set *depth* to a number obtained from the range [*2,temp_depth*]

Figure III.2: Procedure *set_depth()*

The procedure *set_depth()* determines how many levels will there be in the graph by using the value of *shape*.

```
1.  if depth==num_node then create a unique node for each level
2.  else
3.      cur_nodes=0;
4.      for(i=1;i<depth-1;i++)
5.          assign the number of node that is obtained from the distribution
              with mean equal to shape × √(num_node)
6.          add this number to the cur_nodes
7.      endfor
8.  endif
9.  diff=cur_nodes-num_node
10. if diff>0
11.     while diff>0
12.         delete a node from randomly chosen level having more than 2
        nodes
13.         diff--
14.     endwhile
15. else if diff<0
16.     while diff<0
17.         add a node to a randomly chosen level other than entry and exit
        level
18.     diff++
19.     endwhile
```

Figure III.3: Procedure *generate_nodes()*

The procedure *generate_nodes()* determines how many nodes will there be on each level. If the total number of tasks is not equal to *num_node* then it adds new nodes to or subtracts some from the current nodes.

```
1.  for(i=0;i<depth;i++)
2.      if there is a unique node in that level
3.          connect it to all nodes below that level
4.      else if there is a unique node on the level after that level
5.          connect all nodes to the unique node below that level
6.      else
7.          for each node in that level
8.              obtain temp_outdegree randomly from the range
                [1,out_degree]
9.              connect each node to temp_outdegree randomly chosen
                nodes below that level
10.         endfor
11. endfor
```

Figure III.4: Procedure *generate_edges()*.

The procedure generate_edges() determines the connections between nodes using the out_degree value.

```
1.  for(i=0;i<num_node;i++)
2.      obtain a float value from the range [1,max_p_data] as the base computation
            cost for the current node
3.      for each child of the current node
4.          obtain a float value from the range [1,max_t_data] as the base
                commnunication cost from current node to its child
5.      endfor
6.  endfor
```

Figure III.5: Procedure *initialize_graph()*.

The procedure initialize_graph() in Figure III.5 determines the base computation of the tasks and the base communication costs of the edges (messages) between tasks.

procedure save_graph_in_a_file()
//this procedure saves the graph in a file in the format as shown below:
// only one row is labeled to figure out the values

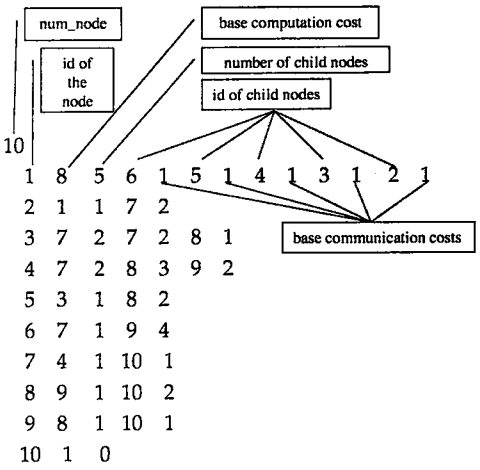


Figure III.6: Procedure *save_graph_in_a_file()*.

Figure III.6 is worth to mention about. The procedure saves the graph in a file in the format as shown (one row is labeled to figure out the values). We gave an example to tell how we represent an application graph as a text-based document. We firstly state the total number of nodes in the graph. Then each node on a separate row, we firstly write the id of the node followed by its base computation cost. After that, comes the total number of children that the node sends data or an edge associated between. Following this integer value, we list each child's id and the base communication cost value. In fact, the example in Figure III.6 is the text format of the application graph demonstrated in Figure II.1.

III.1.2 Random Topology Generator

A random topology generator (RTG) is implemented that gets some input and gives as output a topology system with the properties as described above. The pseudo-code for the RTG is given below in Figure III.7.

- Inputs:
  numpe, top_type, degree, media_speed, max_p_speed, consistent, num_node, r_seed
- Output
  a processor topology written in a text file

1. according to the *top_type* value create a topology by *define_topology_and_links()*
2. create the processor heterogeneity values by *build_topology()*
3. save the topology in a file by *save_topology_in_a_file()*

Figure III.7: Pseudo-code for random topology generator.

Let us comment on the definitions that take place in the algorithm RTG. The value *numpe* obviously tells the number of processors that the processor network will contain.

One of our assertions was the statement that our system model covers most of the common network architectures. In our system we have five different topology types; linear, ring, hypercube, randomly connected and fully connected network topologies. These five types coves most of the real-world architectures except very special ones. Examples to these topologies are in Figure III.12. In the pseudo-code, *top_type* is used for this aim.

Number of neighbors is a significant sign to tell the network connectivity. More number of neighbors for each processor means more connectivity. For a ring network topology, the degree value for each processor is 2. For a linear network topology, this number is 2 for all except the rear and front processors. In a hypercube architecture, degree is the basic factor in deciding the processor number. For example, a hypercube topology with degree 3 has 8 processors each having 3 neighbors. We use *degree* parameter mainly for arbitrarily connected network topologies. For an arbitrarily connected topology, number of neighbors is randomly selected from the range [*1,degree*]. But for a fully connected system, we do not need a degree value because every processor is connected with each other.

Any two connected processors has a link between them whose data passing capability is determined by link heterogeneity factor (*media_speed*). All links' capacities are randomly generated from the range [*1,media_speed*].

Max_p_speed is the maximum value that a processor may process a unit data of any task. All processors' speed are randomly selected from the range [*1,max_p_speed*]. It should be noted here that we modeled two versions of processor heterogeneity factor: consistent and inconsistent processor speeds. In the consistent model we assume that if a processor that processes a task faster than other processors, processes all tasks fastest. In the inconsistent

model, we assumed that a task having same amount of data may be processed fastest on one processor while slowest on another, because of the fact that some data may yield different performances on different architectures.

Num_node has the same meaning as defined in RGG. But one should note that same *num_node* value must be entered to both RGG and RTG.

Finally, *r_seed* stands for any long value that will be used to start the random processes.

Algorithm *RTG* is formed of 3 statements of procedures calls, namely *define_topology_and_links, build_topology, save_topology_in_a_file*, are discovered in Figures III.8, III.9, and III.10 respectively.

```
1.  switch(top_type)
2.      case 1: create a network topology which resembles bus structure
                with rear and front processors unconnected.

3.      case 2: create a ring topology which is similar to linear network
                structure but with rear and front processors connected.

4.      case3: create a hypercube topology according to the degree value.

5.      case4: create a randomly connected processor topology with
                number of neighbours randomy obtained from the range
                [1,degree]

6.      case5: create a fully connected processor topology in which each
                processor is connected to any other with a link.

7.  for each link created above
        assign a link heterogeneity factor from the range [1,media_speed]
8.  endfor
```

Figure III.8: Procedure *define_topology_and_links()*.

The procedure *define_topology_and_links()* determines processor connections according to *top_type* value and assigns the link heterogeneity factors by using the *media_speed* value.

```
1.  for(i=0;i<numpe;i++)
      create the neighbour list for the current processor.

2.  check_connectivity() according to neighbour lists.

3.  if consistent==1 then
      create consistent processor heterogeneity values
    else if consistent==0 then
      create inconsistent processor heterogeneity values.
```

Figure III.9: Procedure *build_topology()*.

The procedure determines the processor heterogeneity values and checks connectivity of processors.

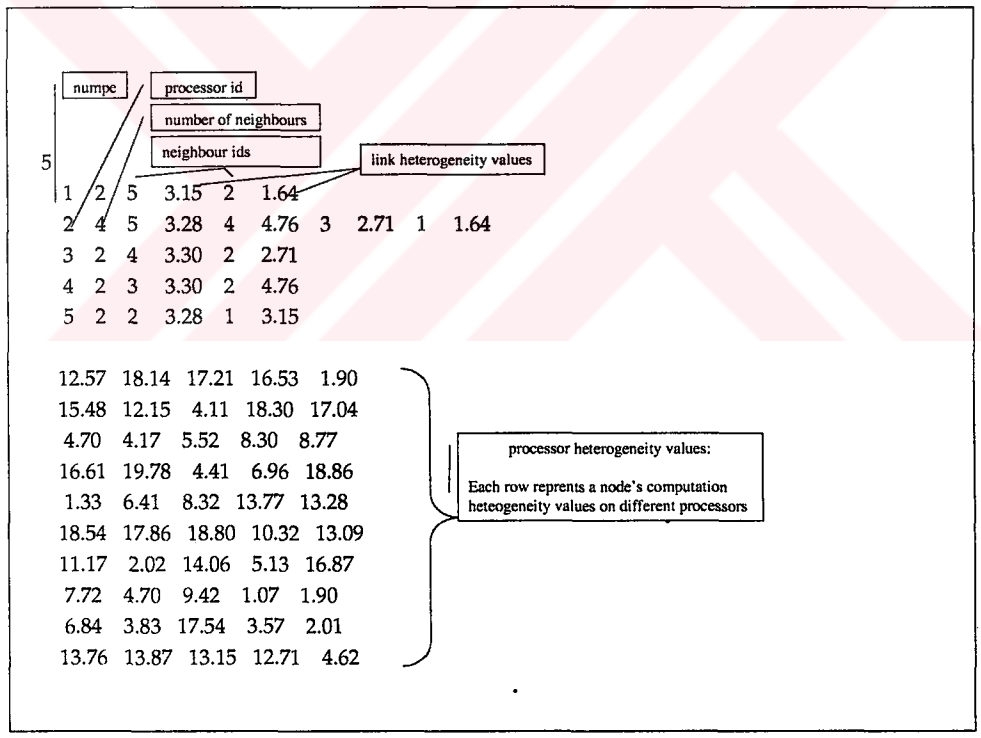


Figure III.10: Procedure *save_topology_in_a_file()*.

In Figure III.10 we gave an example how we state network topology in a text-based document. We start by writing the total number of processors in the system. Then in each row, firstly comes the processor id followed by its total number of neighbors. Rest of the row lists the neighbor ids and the link heterogeneity factor of the link connecting these processors. It is worth to note that the link heterogeneity factor values are symmetric, that is you can see

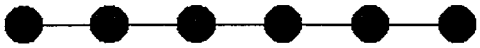
the link heterogeneity factor value of the link connecting P_1 and P_5 both in row 1 and row 5. Figure III.10 contains the text-based representation of Figure II.2.

After the listing of processors, comes the listing of processor heterogeneity values. This matrix consists of *num_node* number of rows and *numpe* number of columns. Each row carries the computation speed of one unit of data of the task on *numpe* number of processors. Consistent and inconsistent models are explained above. The example given in Figure III.10 demonstrates an inconsistent model.

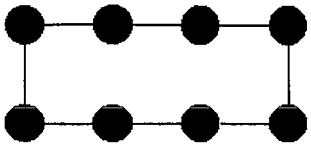
```
procedure check_connectivity()
//this procedure checks to see whether one can walk through all processors when
started from a randomly chosen processor.
    do
        connected=1
        for(i=0;i<numpe;i++)
            p_i.reached=0
        mark(p_0)
        for(i=0;i<numpe;i++)
            if p_i.reached=0 then
                up=p_i
                connected=0
        if connected=0 then
            connect up to a randomly chosen already connected processor
    while connected==0


procedure mark(processor p)
//this procedure recursively marks the processor as reached.

    if p.reached=0
        p.reached=1
        for each neighbour np of p
            mark(np)
```

Figure III.11: Procedures *check_connectivity() and mark(processor p).*

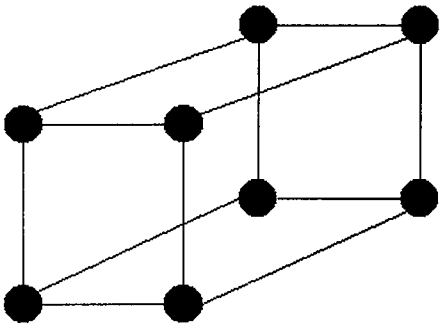
In Figure III.11 we gave the pseudo-code for checking the connectivity of a created processor topology. We needed to write these procedures because connecting the processors randomly may result in unconnected sub-topologies and so we must check whether all the processors are connected or not. In this checking process we make a depth-first-search starting from P_0 to mark as *reached* which processors are connected.



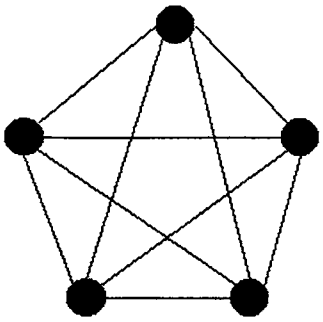
a. A linear network topology consisting of five processors



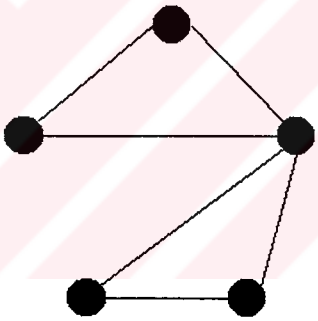
b. A Ring network topology consisting of 5 processors.



c. A Hypercube network topology of degree 3 (8 processors)



d. A Fully connected network topology consisting of 5 processors.



e. A Randomly connected network topology with maximum degree value of 4 with 5 processors.

Figure III.12: Examples to five different topology types.

III.2 PROPOSED ALGORITHM

The algorithm we propose is the Contention Aware Heterogeneous Task Scheduling (CAHTS) Algorithm. The proposed algorithm is a list-scheduling algorithm whose main idea is firstly listing the tasks according to their *blevel* values in a descending order and then scheduling each task onto the processor that gives the earliest finish time (EFT).

One of the former studies that is very close to CAHTS is HEFT and basic assumptions that HEFT was clearly to be applied were as follows:

- Fully connected architecture.
- Scheduling multiple messages on same link without considering overlaps.
- Heterogeneous execution values of tasks.

The HEFT algorithm uses the earliest finish time value, EFT, to select the processor for each task. HEFT has insertion-based approach while computing the EFT.

HEFT and other algorithms were not adequate to cope with the newly introduced constraints, namely arbitrary connection of processor topology, heterogeneous link media speeds between processors and carrying only a unique message at a time i.e. contention awareness. Hence we namely developed Contention Aware Heterogeneous Task Scheduling (CAHTS).

Scheduling tasks onto an arbitrary processor topology naturally brings the problem of message routing and scheduling onto links, because any message from task T_i to T_j needs to be routed to a path of links if T_i and T_j are not scheduled to same processor. Dijkstra is the one that is most widely used in these applications. Hence this route or paths of links between processors are determined by Dijkstra's algorithm.

Another issue is to scheduling the incoming messages of the ready task T_i. Assume that task T_i is the ready task in the ordered list to be scheduled to a processor and it has k number of ancestor i.e. k incoming messages. Scheduling these k messages onto links is a crucial step for the performance of the algorithm, since we take into account the contention factor. We studied three alternatives for ordering these messages before their schedule. These alternatives are

- Scheduling the messages of the parent tasks in an ascending order starting with minimum finish time,
- Scheduling the messages of the parent tasks in an ascending order starting with minimum communication cost,
- Scheduling the messages of the parent tasks in an ascending order starting with minimum communication cost + finish time,

CAHTS algorithm may be summarized in Figure III.13.

```
1.  construct routing table using Dijkstra.
2.  compute b_level for each task by traversing application graph upward, starting from the
    exit node.
3.  sort the tasks in a list by nonincreasing order of b_level values.
4.  while there are unscheduled tasks in the list do
5.      for each processor P_x in the topology
6.          select the first task T_i in the list and remove it.
7.          Case 1: sort the messages of the parent tasks in an ascending order starting with
                minimum finish time in a list H.
            Case 2: sort the messages of the parent tasks in an ascending order starting with
                minimum communication cost in a list H.
            Case 3:sort the messages of the parent tasks in an ascending order starting with
                minimum finish time in a list H.
8.          while the list H is not empty do
9.              select the first message M_ji (coming from T_j) in the list and then remove it
10.             if P(T_j)!= P_x then time_j=schedule_communication(P_x,P(T_j))
11.             else time_j =FT(T_j)
12.             obtain t_max among all time_j
13.         endwhile
14.         calculate FT(T_i, P_x)=schedule time of T_i on P_x considering t_max as the arriving
                time for all messages and using insertion-based approach
15.     endfor
16      select the processor that gives the smallest FT(T_i, P_y) value and schedule T_i on P_y
17. endwhile
```

Figure III.13: Pseudo–code for algorithm CAHTS

```
1.  for each link L_tu on the path from P_y to P_x
2.      search for appropriate neighbor address of P_u among p
    processors
3.      compute the cost of M_ji on link L_tu
4.  endfor
5.  obtain the maximum cost cost_max among these costs
6.  for each link L_tu on the path from P_y to P_x
7.      schedule M_ji among e messages scheduled on L_tu
```

Figure III.14: The procedure of *schedule_communication*(P_x,P_y,T_i,T_j)

The procedure *schedule_communication*(P_x,P_y,T_i,T_j) given in Figure III.14 returns the $MFT(M_{ji},L_{xy})$ where P_x is the candidate processor and M_{ji} is the message from T_j already scheduled on P_y to T_i and $P_x != P_y$.

There are some vital differences between CAHTS and HEFT. Indeed, selecting the best processor in HEFT has only a $O(np)$ of complexity because it must choose the best processor among p processors and for each processor it must temporarily schedule at most n messages onto the links. Hence HEFT has a time complexity of $O(n^2p)$. On the other hand, the vital difference for CAHTS emerges when we analyze its message scheduling time complexity. For both message transfer implementations of the CAHTS algorithm, we see the basic term for order calculation which is $O(n^2p)$. When we use *cut-through* message scheduling method the order for scheduling a unique message is $O(pp+2pe+e+p(e+e))$. In

this term, the first *pp* term refers to finding the cost of message transfer on each link on the predefined route on a maximum *p* processors and its search for appropriate neighbor address among *p* processors. *2pe* stands for merging these *p* links where each merge operation of two links includes merging *2e* number messages. After this operation, we search on the merged link for the best time value to insert the message that is stated by *e*. The last term simply states scheduling the message to the best suitable time slot on *p* links after a search among *e* already scheduled messages. From the above analysis, we can conclude that CAHTS algorithm using cut-through message transfer method has an order of $O(n^2p(p^2+2pe+e+2pe))$, which can be summarized as $O(n^2p^2e)$.

When we repeat the same analysis for CAHTS using *store-and-forward* message transfer technique we obtain $O(n^2p(p(p+e)))$ order of complexity. The $p+e$ term in the inner parentheses stand for the search for appropriate neighbor address among *p* processors and locating the message to the best time slot on the route. This is repeated for a maximum of *p* times for *p* links on the route. When summarized, CAHTS implemented using *store-and-forward* message transfer technique yields a $O(n^2p^2e)$. Although this result of complexity seems equal to *cut-through* implementation, we must take into account additional components in the complexity representation of *cut-through* implementation for a realistic comparison.

When we analyze DLS, we see roughly that both message transfer techniques have the same order of complexity but again with additional terms in cut-through transfer method. Specifically, scheduling a unique message in *cut-through* version has a $O(p(p+e)+p))$ complexity . This can be explained as traversing on *p* number of links on each finding the neighbor address for calculating the message cost and finding the message scheduled with the latest time among *e* messages. The last *p* term is for locating the new message on *p* links. *Store-and-forward* implementation of DLS schedules the messages with a complexity of $O(pp+pe)$. The first *pp* term refers to finding the cost of message transfer on each link on the predefined route on a maximum *p* processors and its search for appropriate neighbor address among *p* processors, while the second term is for finding the message with maximum finish time and scheduling the new message just after that on *p* links. When summarized, we obtain $O(n^3p(p^2+pe+p))$ time complexity for *cut-through* version and $O(n^3p(p^2+p))$ for *store-and-forward* implementation.

BSA algorithm has a complexity of $O(p^3n^2e)$ for both message scheduling implementations. Similar to other algorithms, *cut-through* version of the algorithm has more number of terms than *store-and-forward* version for communication scheduling.

PART IV: EXPERIMENTAL WORK AND RESULTS

IV.1 COMPARISON CRITERIA

In this part we define the experimental work studied to figure out the performances of the algorithms depicted so far. We compared two algorithms, namely BSA and DLS with the proposed algorithm CAHTS.

The criteria used for the comparison are the running times of the algorithms and the Schedule Length Ratio (SLR) values. The SLR is firstly defined in [11] by Topçuoğlu as follows:

$$SLR = \frac{makespan}{\sum_{t_i \in CP_{MIN}} \min_{j \in P} \{w_{i,j}\}}$$

where *makespan* is the schedule length produced by the algorithm. The summation term in the formula denotes the value obtained by adding all minimum computation cost values of tasks on the critical path determined with these minimum computation costs. The SLR value truly indicates the performance of the algorithm since best algorithm found ever cannot find a lower value than the one obtained by the denominator. So the SLR value can not be smaller than 1, hence the algorithm that gives closer schedule length values to SLR is the best one.

In order to analyze the algorithms, we implemented a manager program that repeatedly generates application graphs and processor topologies which are scheduled by the algorithms. The charge of the manager program is as follows. It calls RGG and RTG with predefined inputs to produce a random application graph and a processor topology which store the application graph and topology in separate output text filés. Afterwards, these files are used by the scheduling algorithms as input files and they schedule the graph on the processor topology with their own way and append the schedule length value and other criteria values to their own separate output files. This process is repeated for a predefined number of times. Lastly, manager program processes these output files and gives the average result for each scheduling algorithm. The manager program's work is schematically depicted in Figure IV.1.

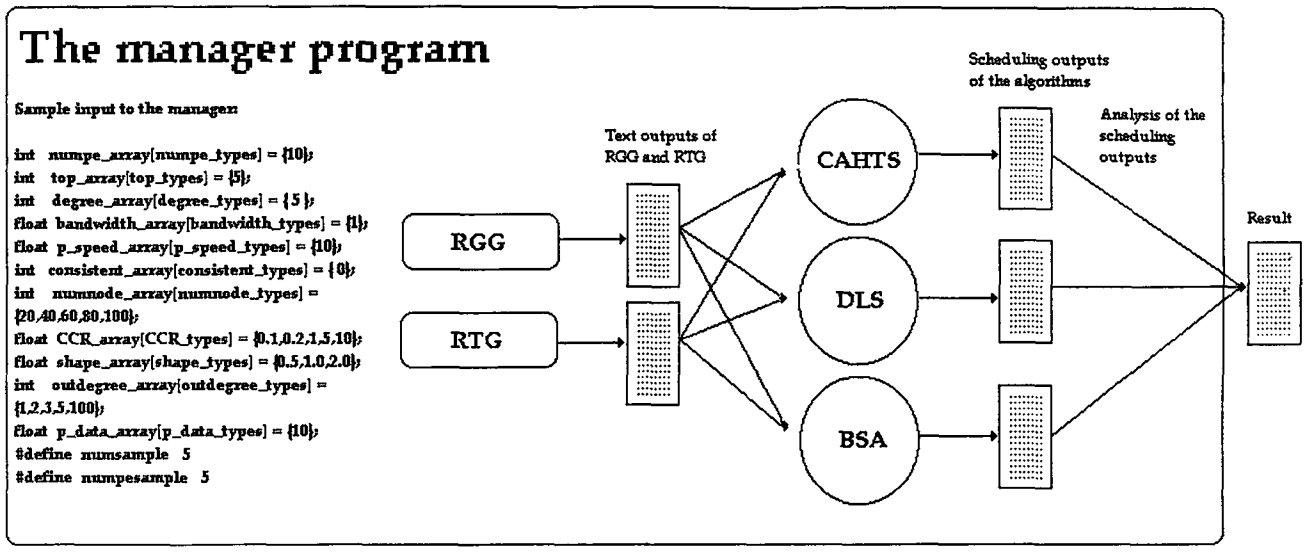


Figure IV.1: The manager program.

We analyzed the algorithms by applying both the *cut-through* and *store-and-forward* message transfer techniques separately. The study is carried on Pentium IV 1.8Ghz machines, which are running the Redhat Linux environment. Naturally the compiler was the gcc compiler of Linux.

The compared algorithms in this comparison are the BSA, DLS, and CAHTS with its 3 different versions as stated in its pseudo-code. Namely, these versions are CAHTS1, CAHTS2 and CAHTS3 representing the CAHTS algorithm using *min(EFT)*, using *min(EFT+comm)* and using *min(comm)*, respectively. The results are shown in figures below.

We defined 10 different suites of tests from which we will extract the performances of the algorithms. These 10 suites are organized such that in each suite we will find out the behaviors of scheduling algorithms for different input parameters with their changing values. Each suite of test is a result of schedules with amount ranging from 30000 to 162000.

IV.2 COMPARISON TESTS

IV.2.1 Performance with respect to task heterogeneity:

We studied the time complexity and SLR performance with respect to task heterogeneity values. In the suite, the inspected task heterogeneity ranges are [1] [1,10] [1,100] [1,1000] and [1,10000]. We created 6000 application graphs with different characteristics and scheduled them on 50 different processor topologies. Number of processors and number of application tasks are set as 8 and 100, respectively, and the study is

performed with non-consistent processor heterogeneity values in the range [1,20]. All input values for the suite are given in Table V.1 below.

1st suite		
constants	variable	value
	numpe	8
	consistent	0
	proc. het.	[1,20]
	link het.	[1]
	CCR	1
	numnode	100
varied values	variable	value
	shape	{0.5,1,2}
	out_degree	{2,5,10,20}
outest loop variable	task heterogeneity.	{[1][1,10][1,100][1,1000][1,10000]}

Table IV.1: The values of the input parameters in study of suite 1.

The results of the suite are shown in Figures IV.2 and IV.3 below. These figures show that the task heterogeneity value does not affect the time complexity and SLR performances of all algorithms. The time complexity for BSA with cut-through implementation is very high that it is not given in Figure IV.1.a. Store-and-forward versions of all algorithms last less than cut-through versions of themselves, but in BSA this decrease is seen sharply because of the fact that using cut-through message transfer technique in BSA brings a heavy burden in terms of time complexity. Specifically, we observe that CAHTS1 gives the best time and SLR performance among all cut-through versions of 5 algorithms. The same is true for store-and-forward versions of the algorithms. Another important point of the results is that each CAHTS algorithm and DLS algorithm shows the same SLR performance when their cut-through and store-and-forward versions are compared. This is a consequence of fully connected structure of the topology and equal link heterogeneity value causing both transfer techniques to yield the same routes.

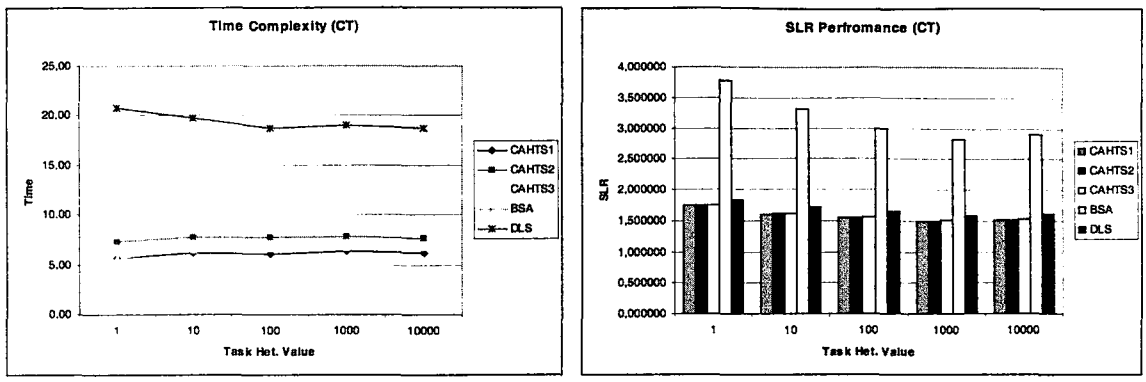


Figure IV.2.a,b: Comparison results of algorithms implemented with *cut-through* technique scheduled on fully connected topologies with respect to task heterogeneity.

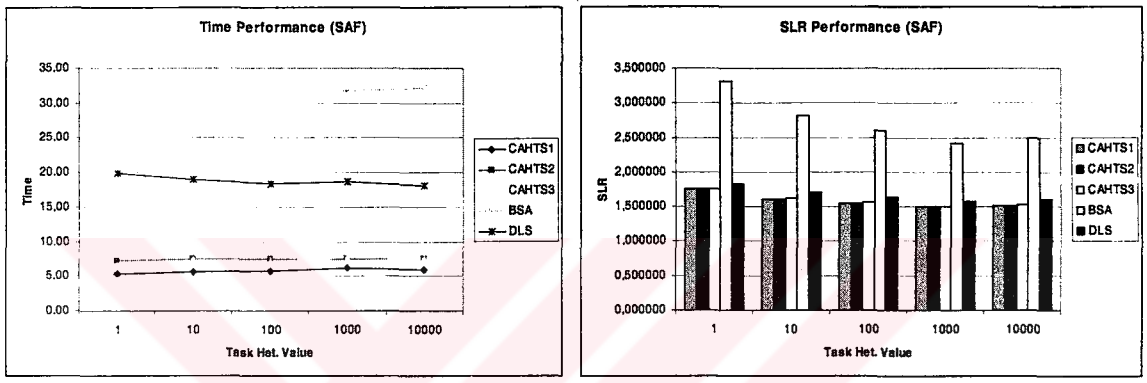


Figure IV.3.a,b: Comparison results of algorithms implemented with *store-and-forward* technique scheduled on fully connected topologies with respect to task heterogeneity.

We repeated the same study on arbitrarily connected processor topologies with degree value selected from the range [1,4]. The results are given in Figure IV.4 and IV.5. Increase in the task heterogeneity value does not severely affect the time complexity and SLR performances of the algorithms as in fully connected structure. All algorithms show similar behaviors. CAHTS1 versions, again, give the best time and SLR performance within their groups.

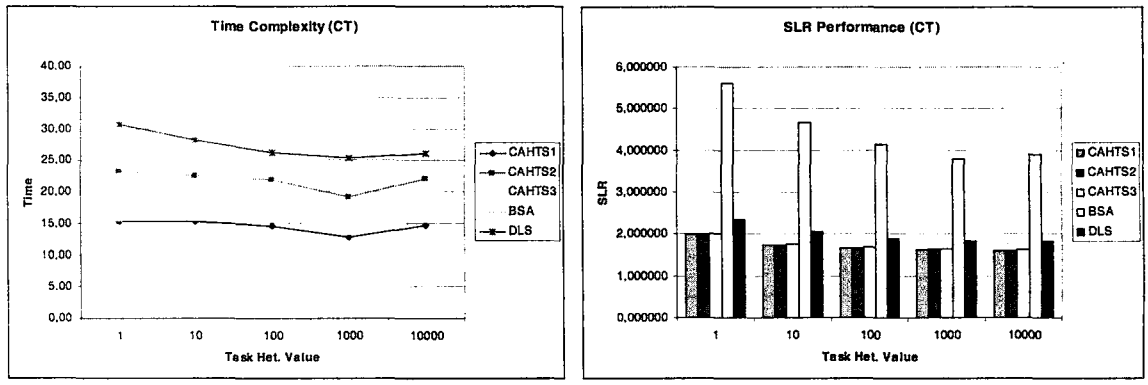


Figure IV.4.a,b: Comparison results of algorithms implemented with *cut-through* technique scheduled on arbitrarily connected topologies with respect to task heterogeneity.

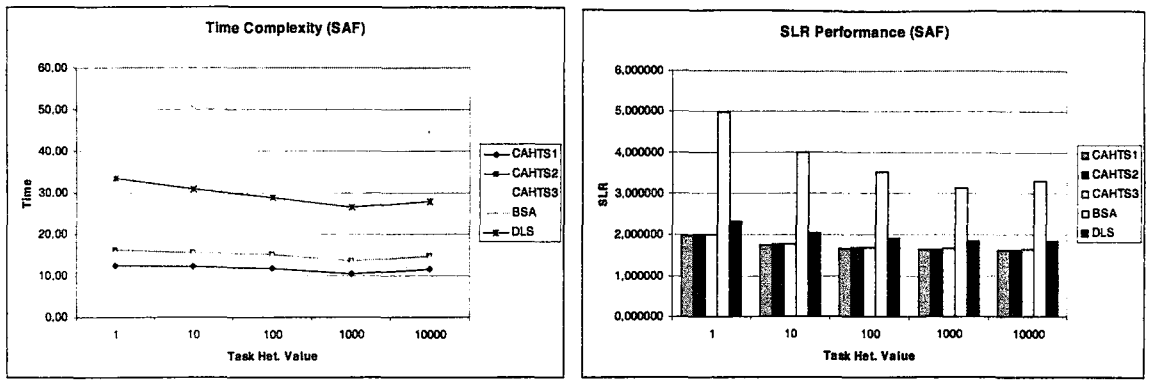


Figure IV.5.a,b: Comparison results of algorithms implemented with *store-and-forward* technique scheduled on arbitrarily connected topologies with respect to task heterogeneity.

The same study is repeated on hypercube topologies with 8 processors (degree 3), ring topologies and linear topologies. The SLR performance results to these tests are given in Figures IV.6, IV.7 and IV.8, respectively. The first thing to be noted from these graphs is the performance order of the algorithms. CAHTS1 outperforms all other algorithms in SLR performance. The algorithms rankings are, in both message transfer implementation versions, listed as CAHTS1, CAHTS2, CAHTS3, DLS and BSA. One can see that the relative SLR performance value increases as the connectivity decreases. That is, CAHTS1 shows much better performance with decreasing connectivity. Specifically, for example, CAHTS1 shows better performance than DLS by %6 in fully connected topologies on average while it shows %18 better performance on linear networks. In these graphs, in fact together with the above ones, we can see that as the connectivity of target architecture decreases, the SLR values of both cut-through and store-and-forward implementations increase. This is because of the fact that as number of links decrease the messages to be transferred get crowded on the few links which in turn increases schedule length.

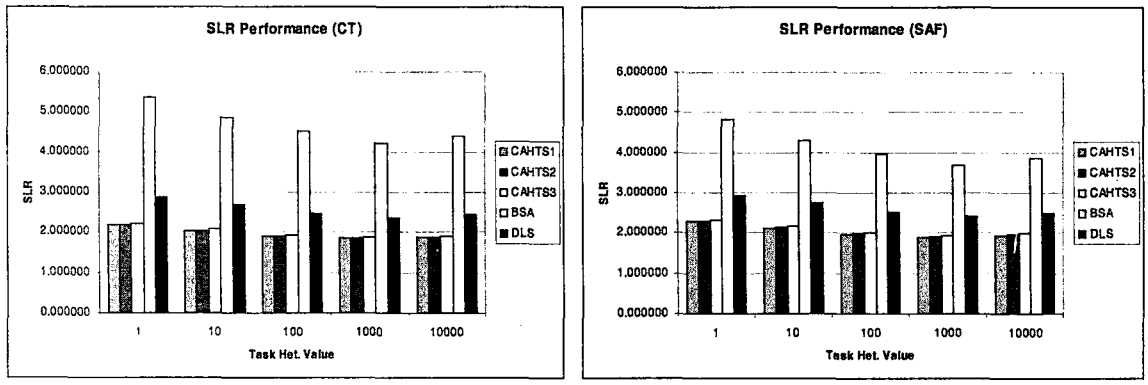


Figure IV.6.a,b: Comparison results of algorithms implemented with both techniques scheduled on hypercube topologies with respect to task heterogeneity.

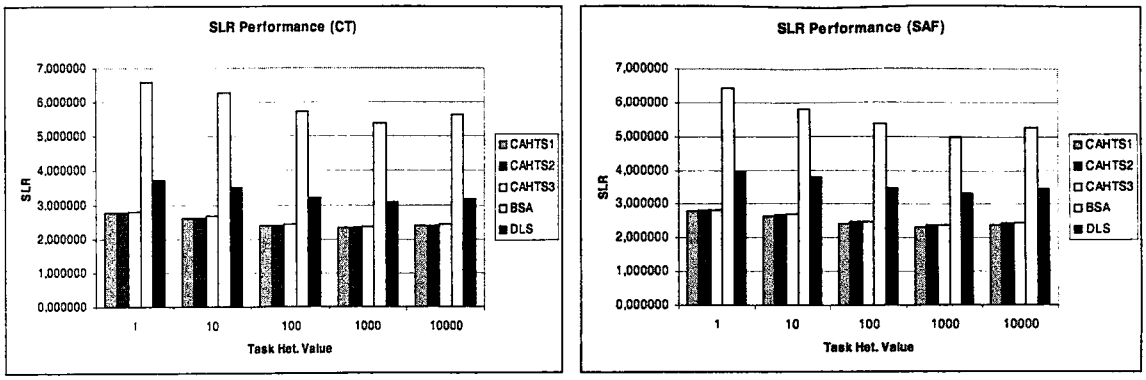


Figure IV.7.a,b: Comparison results of algorithms implemented with both techniques scheduled on ring topologies with respect to task heterogeneity.

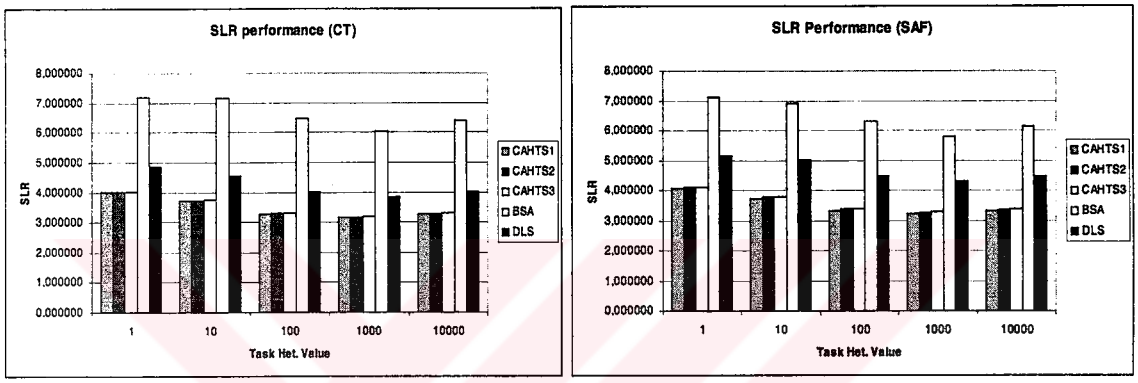


Figure IV.8.a,b: Comparison results of algorithms implemented with both techniques scheduled on linear topologies with respect to task heterogeneity.

The studies in the first suite showed us that task heterogeneity value of application graphs does not severely affect the performances of the algorithms and CAHTS1 gives the best SLR performance results and relatively much better results on topologies with decreasing connection between processors.

IV.2.2 Performance with respect to processor heterogeneity:

The second suite of our study covers the processor heterogeneity affects on the performances. In this context, we selected our processor heterogeneity ranges to be inspected as [1] [1,10] [1,100] [1,1000] and [1,10000] having inconsistent property. The application graphs in the tests consisted of 100 tasks, the topology had 8 processors with connection links having equal link heterogeneity values. The values for the input parameters of the test are given below. As in previous suite of study, we produced 6000 application graphs with different properties and scheduled them on 50 different processors.

2nd suite		
constants	**variable**	**value**
	numpe	8
	consistent	0
	Task het.	[1,100]
	Link het.	[1]
	CCR	1
	Numnode	100
varied values	**Variable**	**value**
	Shape	{0.5,1,2}
	out_degree	{2,5,10,20}
outest loop variable	proc. Het.	{[1][1,10][1,100][1,1000] [1,10000]}

Table IV.2: The values of the input parameters in study of suite 2.

We started to run the defined suite of tests on fully connected topologies. The results are given in Figures IV.9 and IV.10. From the results, we can conclude that CAHTS1 shows the best time and SLR performance among all five both when cut-through or store-and-forward methods are implemented as message transfer technique. The time complexity of all algorithms diminishes when they use store-and-forward message transfer technique. We again observe that the SLR value of each algorithm is identical when used either cut-through or store-and-forward methods. This is because of the fact that both techniques yield the same one-link routes to use. The time complexity of the BSA algorithm using cut-through, is very high so that it is not included. On the other hand, BSA shows a tremendous decrease when processor heterogeneity value increases. This is because it can more easily manage to move tasks with high processor heterogeneity values.

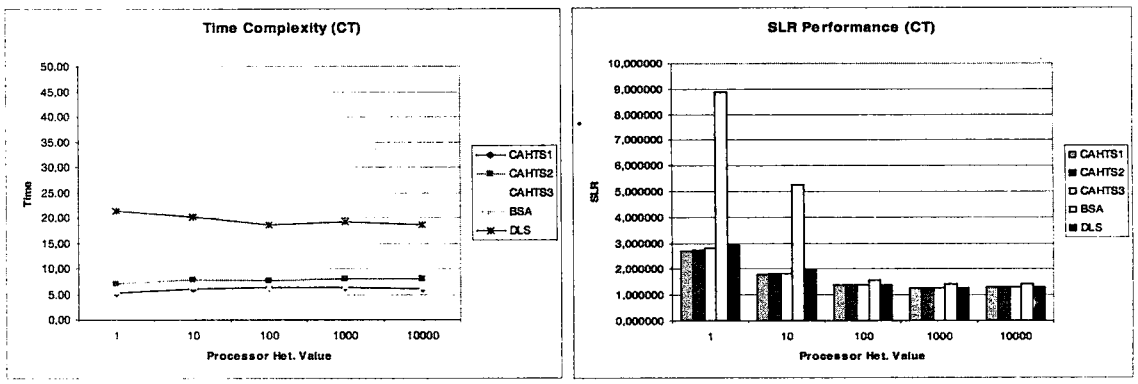


Figure IV.9.a,b: Comparison results of algorithms implemented with *cut-through* technique scheduled on fully connected topologies with respect to processor heterogeneity.

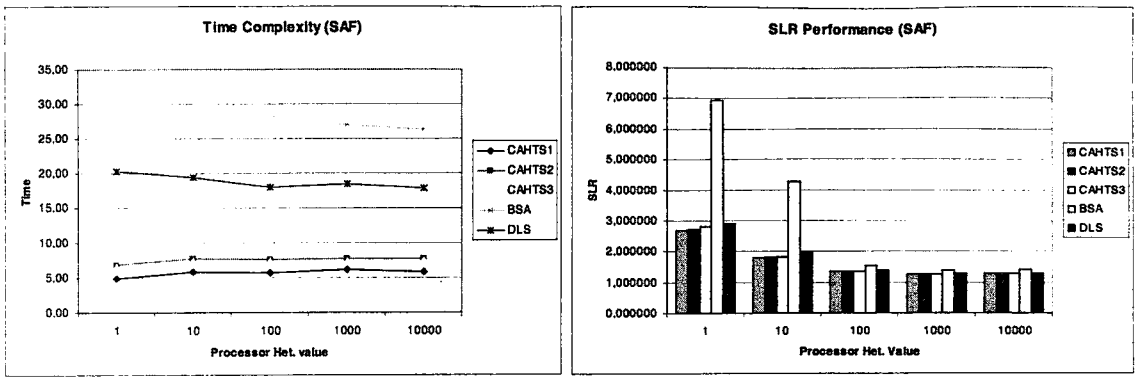


Figure IV.10.a,b: Comparison results of algorithms implemented with *store-and-forward* technique scheduled on fully connected topologies with respect to processor heterogeneity.

We repeated the above test on arbitrarily connected topologies, hypercube topologies with degree 3, ring topologies and linear topologies. The results are shown in Figures IV.11, IV.12, IV.13 and IV.14, respectively. For these tests, we observe the similar results to fully connected test. CAHTS1 outperforms others in terms of SLR performance, and the others' performance order is CAHTS2, CAHTS3, DLS and BSA. CAHTS1 shows relatively better performance while the connectivity between processors decreases. Specifically we can see this from the average SLR values. CAHTS1 outperforms DLS by % 3 on fully connected topologies by %5 on arbitrarily connected topologies, by %7 on hypercube and %10 for ring and linear topologies on the average.

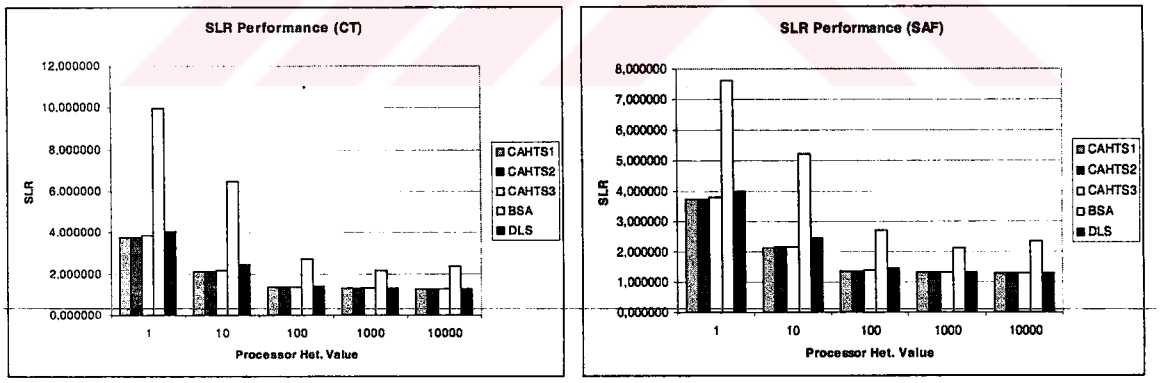


Figure IV.11.a,b: Comparison results of algorithms implemented with both techniques scheduled on arbitrarily connected topologies with respect to processor heterogeneity.

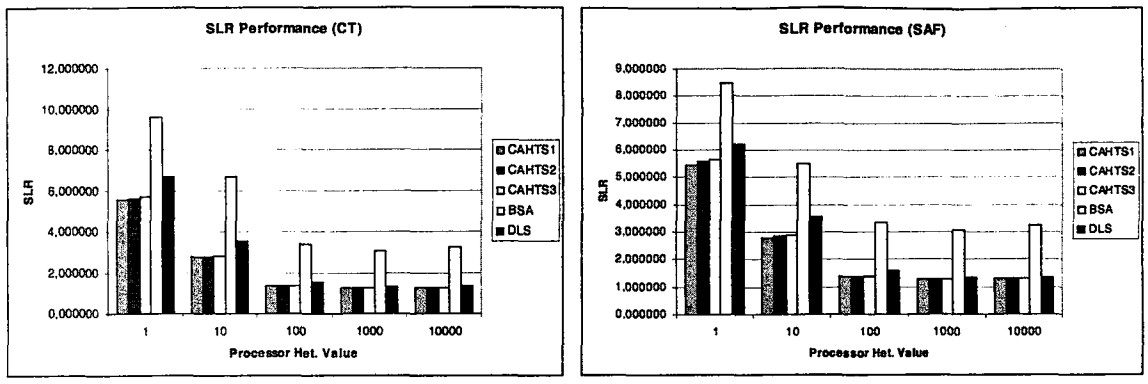


Figure IV.12.a,b: Comparison results of algorithms implemented with both techniques scheduled on hypercube topologies with respect to processor heterogeneity.

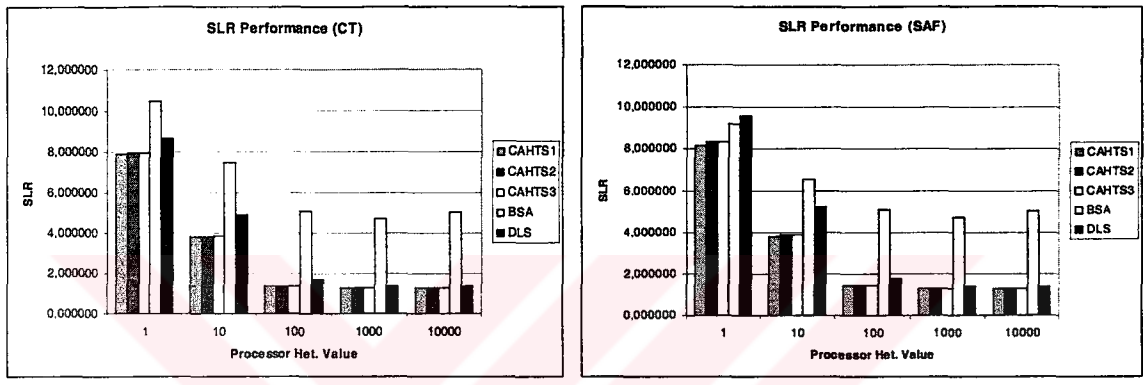


Figure IV.13.a,b: Comparison results of algorithms implemented with both techniques scheduled on ring topologies with respect to processor heterogeneity.

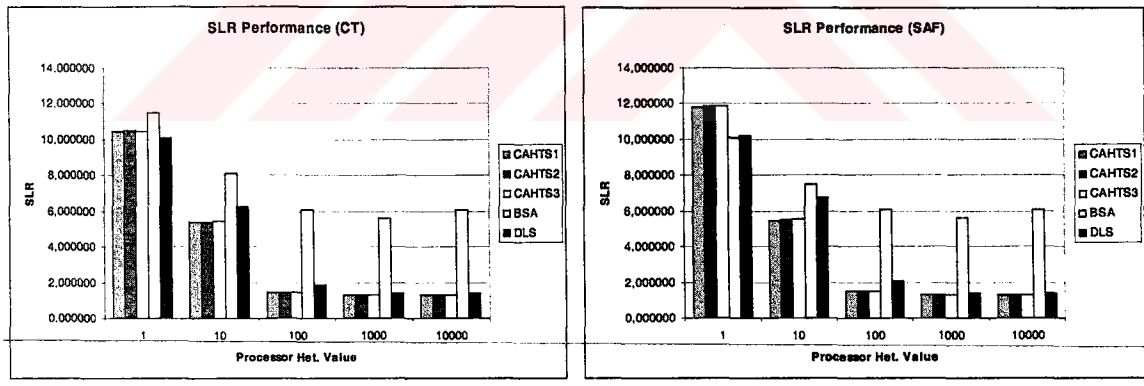


Figure IV.14.a,b: Comparison results of algorithms implemented with both techniques scheduled on linear topologies with respect to processor heterogeneity.

IV.2.3 Performance with respect to link heterogeneity:

Another comparison of algorithms is their performance with respect to link heterogeneity values. In this suite of test we set CCR value as 1 in order to obtain close communication and computation values. In this suite of test the processor heterogeneity value and task heterogeneity values are set to values 20 and 100 respectively, since these values best

represent the performances of the algorithms. The other input parameters' values are summarized in Table IV.3 below.

3rd suite		
constants	**Variable**	**value**
	Numpe	8
	Consistent	0
	proc. Het.	[1,20]
	task het.	[1,100]
	CCR	1
	Numnode	100
varied values	**Variable**	**value**
	Shape	{0.5,1,2}
	out_degree	{2,5,10,20}
Outset loop variable	link het.	{[1][1,2][1,4][1,6][1,8][1,10][1,20][1,50][1,100]}

Table IV.3: The values of the input parameters in study with respect to link heterogeneity.

These combinations of parameter values yield to 10800 different graphs to be created and scheduled on 90 different processor topologies. The test is firstly run on fully connected topologies and gave the following results in Figure IV.15 and IV.16. The SLR performance of the algorithms shows us that CAHTS1 schedules the graphs best. It outperforms DLS by %10 on the average and BSA by %50 on the average for the cut-through versions. For store-and-forward versions these values are %11 and %63 for DLS and BSA, respectively. All algorithms' SLR value increase as the link heterogeneity value increases since the denominator in SLR term does not include communication with any means.

Another important point of these tests that they reveal the fact of CAHTS carrying a very high running time complexity when we use insertion-based approach for message scheduling in cut-through version. This fact is seen in the Figure IV.15.a. In the figure we can see that CAHTS outperforms DLS by %68 when the link heterogeneity value is set to a homogeneous value. For this homogeneous case, while scheduling a unique message CAHTS does not need to merge the messages on the links which discards the pe term and replaces the $p+e$ term as described at the end of the section III.2. So, for homogeneous link heterogeneity The time complexities of the algorithms are $O(n^2p^2e)$ and $O(n^3p^2e)$ for CAHTS and DLS respectively proving the Figure IV.15.a. As explained in section III.2, the complexity for scheduling a unique message in CAHTS has a $O(p^2+2pe+e+pe)$, while DLS has only a

$O(p^2+pe+p)$. When we analyze the same issue for store-and-forward message scheduling technique, we see that CAHTS has a doubtless leadership for time complexity since this technique does not need an extra term for merging the links and searching a common slot to insert the message. Indeed, CAHTS outperforms DLS by %59 and BSA by %61 on the average.

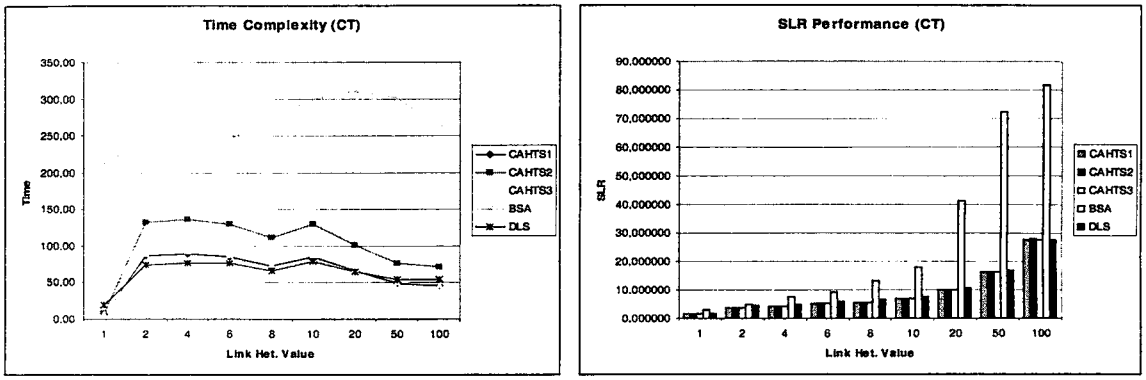


Figure IV.15.a,b: Comparison results of algorithms implemented with cut-through technique scheduled on fully connected topologies with respect to link heterogeneity.

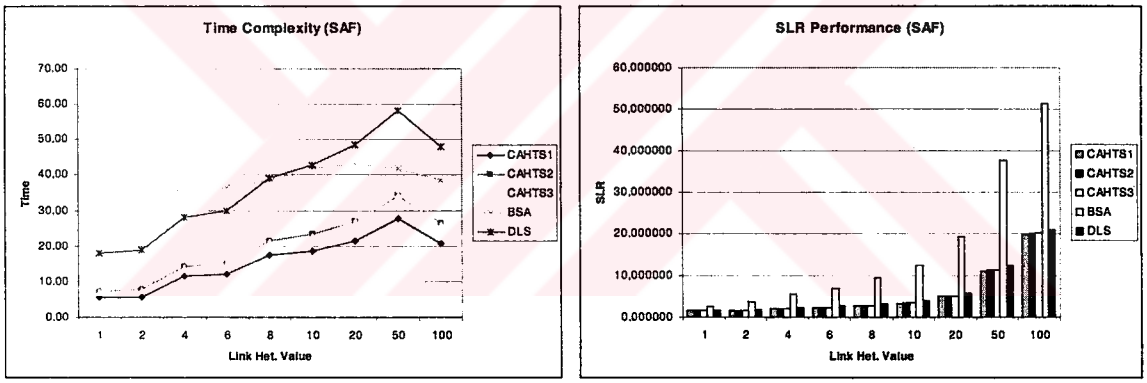


Figure IV.16.a,b: Comparison results of algorithms implemented with *store-and-forward* technique scheduled on fully connected topologies with respect to link heterogeneity.

The tests are repeated for other four topologies and we obtained very similar results to the above ones. CAHTS again gives the best SLR performance among all for referred link heterogeneity values.

IV.2.4 Performance with respect to Number of Nodes:

The performances (SLR values) of the algorithms were compared with respect to number of nodes in Figure IV.17. This suite of test included 90000 application graphs scheduled on 50 different topologies. The application graphs were created by using different

CCR, shape and out degree values and the number of processors was 8. The values for input parameters are given in Table IV.4 below.

4th suite		
constants	**Variable**	**Value**
	Numpe	8
	Consistent	0
	proc. het.	[1,20]
	task het.	[1,100]
	link het.	[1,10]
varied values	**Variable**	**Value**
	Shape	{0.5,1,2}
	Out_degree	{2,5,10,20}
	CCR	{0.1,1.10}
outest loop variable	Numnode	{20,40,60,80,100,150,200,250}

Table IV.4: The values of the input parameters in study with respect to number of nodes.

From these, we can deduce that CAHTS1 applied using store-and-forward message scheduling technique truly outperforms others both in time complexity and SLR performance. This is due to the fact of its low order of complexity when compared with DLS and BSA. Specifically, CAHTS1 outperforms DLS by %56 while it outperforms BSA by %54 when their running times are compared. CAHTS1 has the best SLR value such that it outperforms DLS by %5 and BSA by %59. The ranking for the algorithms is CAHTS1, CAHTS2, CAHTS3, DLS and BSA. For cut-through versions CAHTS1 outperforms DLS by %2 and BSA by %59 when SLR values are compared and we obtain the same ranking. DLS shows the best running time value among all for cut-through version and BSA gives the worst running time complexity. The reason why DLS outperforms CAHTS comes from the fact that CAHTS using a high time complexity for insertion-based message scheduling in heterogeneously linked media. From these, we can conclude that CAHTS1 gives the best SLR performance for both message scheduling techniques and best running time for store-and-forward message scheduling technique.

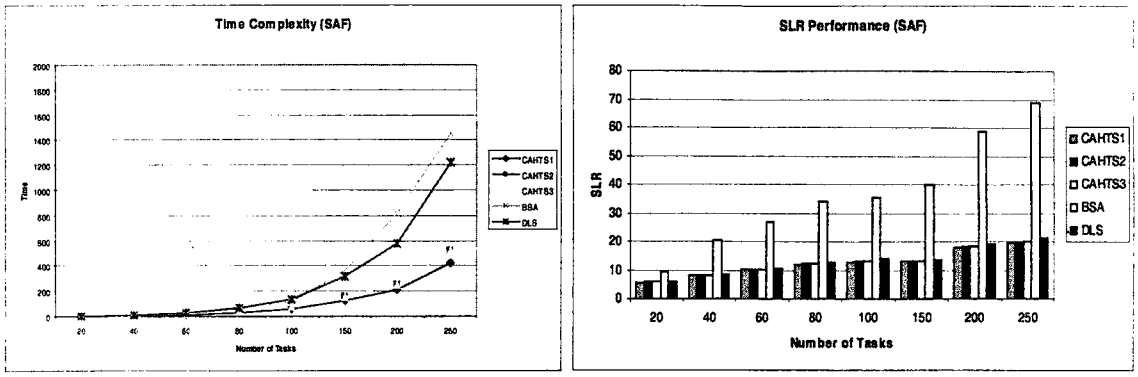


Figure IV.17: Comparison results of algorithms implemented with *store-and-forward* technique scheduled on fully connected topologies with respect to number of nodes.

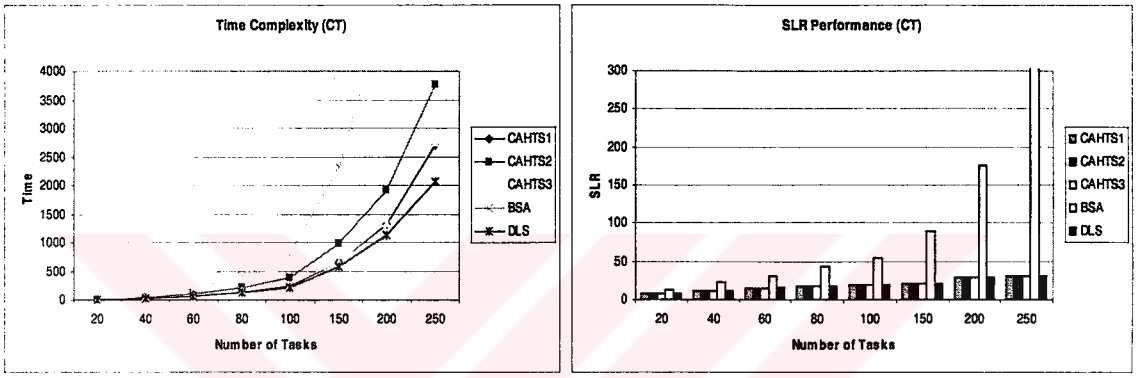


Figure IV.18: Comparison results of algorithms implemented with *cut-through* technique scheduled on fully connected topologies with respect to number of nodes.

IV.2.5 Performance with respect to CCR:

The performance (SLR) with respect to CCR values is also studied. The SLR performance results for the fully connected topologies are given in Figure IV.19. These results are obtained by scheduling 8400 graphs with different characteristics on 70 different fully connected topologies. The values for input parameters are in the Table IV.5.

5th suite		
constants	**variable**	**value**
	numpe	8
	consistent	0
	proc. het.	[1,20]
	task het.	[1,100]
	link het.	[1,10]
	numnode	100
varied values	**variable**	**value**
	shape	{0.5,1,2}
	out_degree	{2,5,10,20}
outest loop variable	CCR	{0.1,0.2,0.5,1,2,5,10}

Table IV.5: The values of the input parameters in study with respect to CCR.

The performance rankings of the algorithms is in the order as CAHTS1, CAHTS2, CAHTS3, DLS and BSA for both cut-through and store-and-forward versions. Specifically, the CAHTS1 algorithm is better than BSA algorithm by %57 and DLS algorithm by %11 for cut-through implementations. The relative performance results for store-and-forward implementations are %60 and %11 for BSA and DLS respectively.

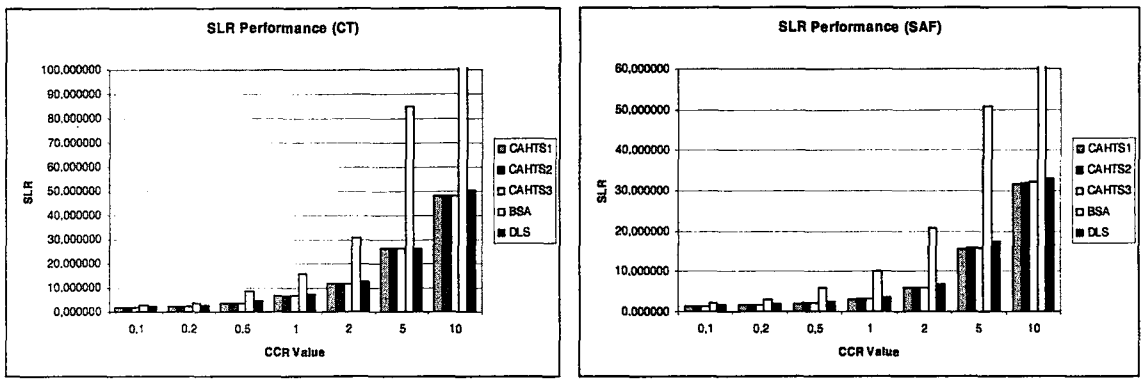


Figure IV.19.a,b: Comparison results of algorithms implemented with both techniques scheduled on fully connected topologies with respect to CCR.

The same tests are repeated for other four topologies in order to exploit the behavior of algorithms on these topologies. The results are given in Figures IV.20, IV.21, IV.22 and IV.23.

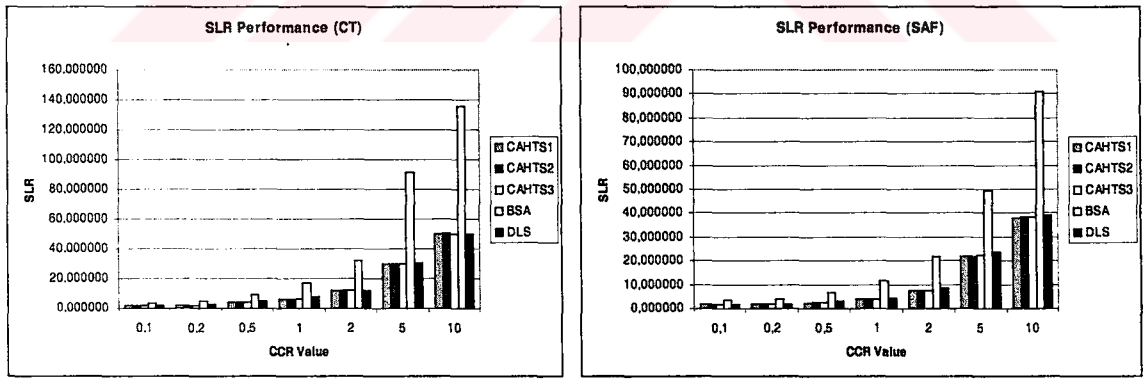


Figure IV.20.a,b: Comparison results of algorithms implemented with both techniques scheduled on arbitrarily connected topologies with respect to CCR.

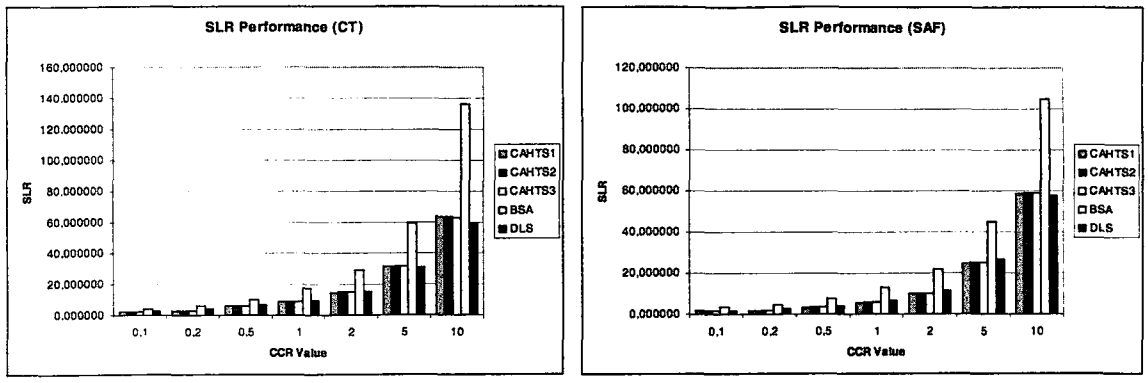


Figure IV.21.a,b: Comparison results of algorithms implemented with both techniques scheduled on hypercube topologies with degree 3 with respect to CCR.

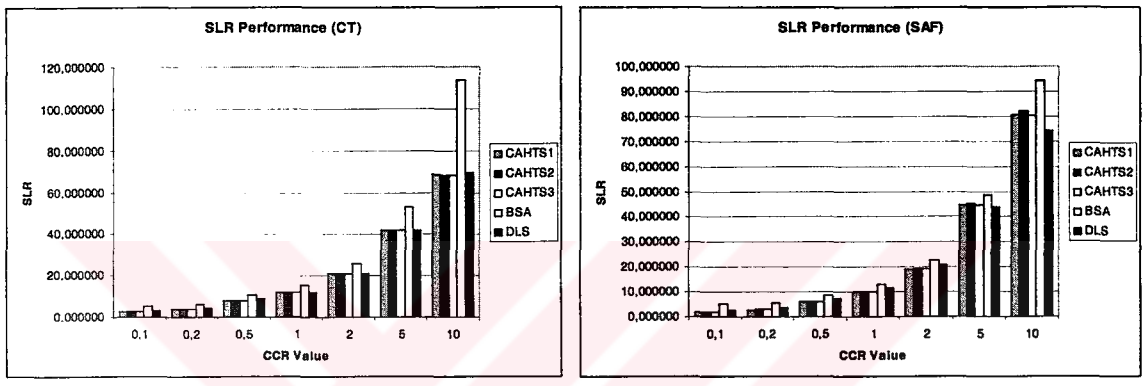


Figure IV.22.a,b: Comparison results of algorithms implemented with both techniques scheduled on ring topologies with respect to CCR.

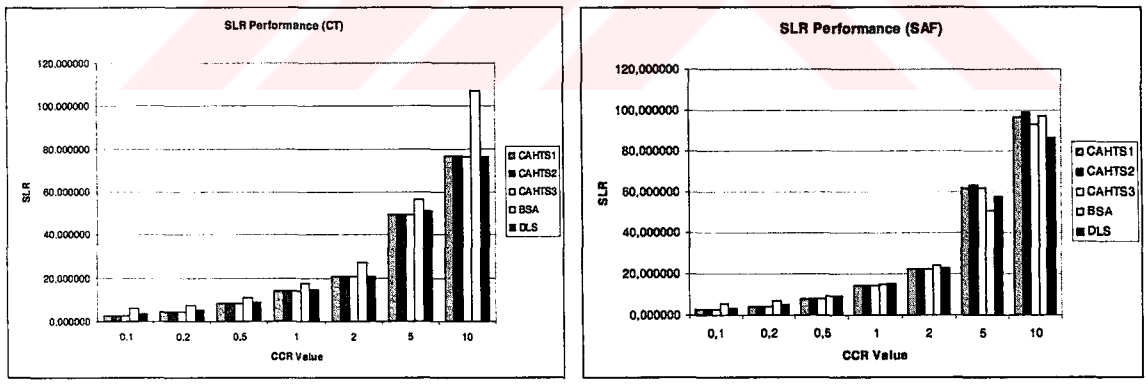


Figure IV.23.a,b: Comparison results of algorithms implemented with both techniques scheduled on linear topologies with respect to CCR.

IV.2.6 Performance with respect to out degree:

Out degree value of a graph indicates how dense that graph will be. A large out degree value will yield a densely connected application graph. This suite of tests will explore the performance of algorithms with respect to out degree value. In this suite, 7200 application graphs are produced by the random graph generator and they are scheduled on 80 different

processor topologies. The explored values for out degree parameter are 1,2,4,6,8,10,20 and 100. With out degree value equal to 1 we obtain a loosely connected application graph such that each task is predecessor of one task on the average. With out degree value 100 we obtain a densely connected application graph such that each task in the graph is most probably connected to each task in the upper and lower level. The values for input parameters are given in Table IV.6 below.

7th suite		
constants	**Variable**	**value**
	Numpe	8
	proc. het.	[1,20]
	task het.	[1,100]
	link het.	[1,10]
	Numnode	100
	Consistent	0
varied values	**Variable**	**value**
	Shape	{0.5,1,2}
	CCR	{0.1,1.10}
outest loop variable	out_degree	{1,2,4,6,8,10,20,100}

Table IV.6: The values of the input parameters in study with respect to out degree.

Figure IV.24 gives the SLR performance results of the algorithms on fully connected topologies. From the graphs one can easily conclude that CAHTS1 shows the best performance and BSA shows the worst. The SLR performance ranking is CAHTS1, CAHTS2, CAHTS3, DLS and BSA. CAHTS1 outperforms DLS by %6 on the average and BSA by %66 for cut-through versions. The same ranking result is obtained when we repeated the test on store-and-forward implementation versions of the algorithms.

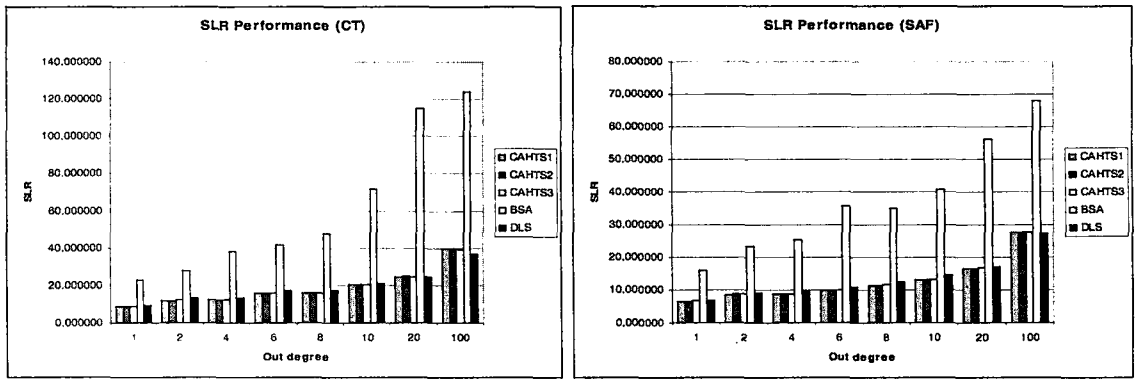


Figure IV.24.a,b: Comparison results of algorithms implemented with both techniques scheduled on fully connected topologies with respect to out degree.

The result of the same tests that are run on arbitrarily connected topologies yield the same rakings but this time CAHTS1 is better than DLS by %4.5 and BSA by %63 for cut-through versions and the same ranking of performance values when store-and-forward is applied.

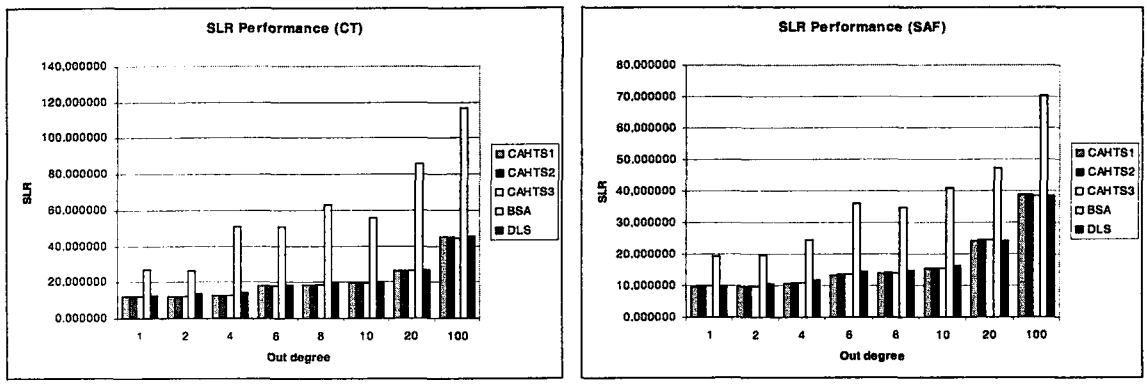


Figure IV.25.a,b: Comparison results of algorithms implemented with both techniques scheduled on arbitrarily connected topologies with respect to out degree.

IV.2.7 Performance with respect to Shape:

Figure IV.26 gives the time complexity and SLR performances of the algorithms' cut-through versions when the shape values were {0.1,0.2,0.5,1,2,5,10} . A low shape value produces a graph with a longer depth and low parallelism. This case even results in graphs that are sequentially ordered with number of levels equal to the number of tasks when shape value is given as 0.1 to a graph to consisting 100 nodes. Conversely a high shape value results in a highly parallelized graphs with shorter depths. From the Figure IV.26 we can conclude that CAHTS1 gives better time complexity for higher shape values and on the contrary, DLS shows better time complexity for low shape values. This just because as shape value increases, the number of ready tasks to be selected for scheduling increases which in turn leads to high number of processor-node pairs to compute dynamic level value and compare for DLS. The arc for all CAHTS are because for equally shaped application graphs there are more number of edges to be scheduled on links than the other extreme cases. The same is also true for DLS and BSA. When their SLR performance are compared, CAHTS1, on the average, outperforms DLS by %1.

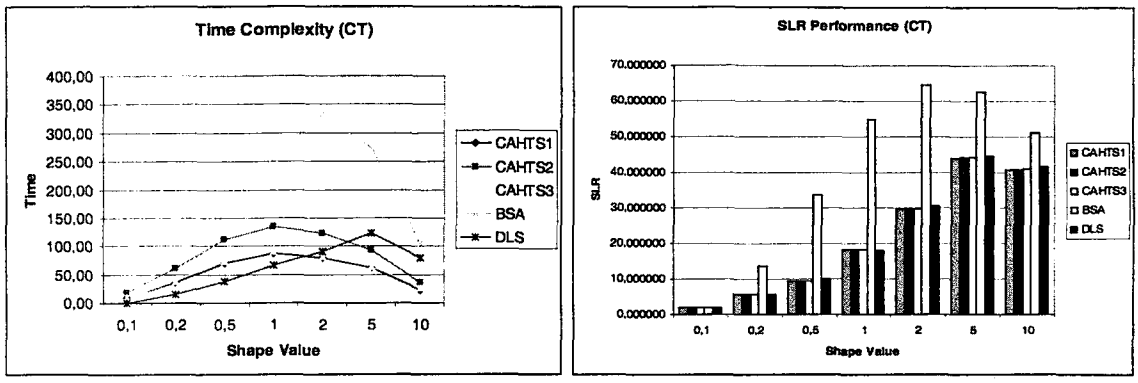


Figure IV.26.a,b: Comparison results of algorithms implemented with cut-through technique scheduled on fully connected topologies with respect to shape value.

For the store-and-forward versions of the algorithms we see the same time complexity results for CAHTS1, CAHTS2 and CAHTS3 and a much higher value for DLS. CAHTS1 outperforms DLS by % 3 on the average and BSA by %52.

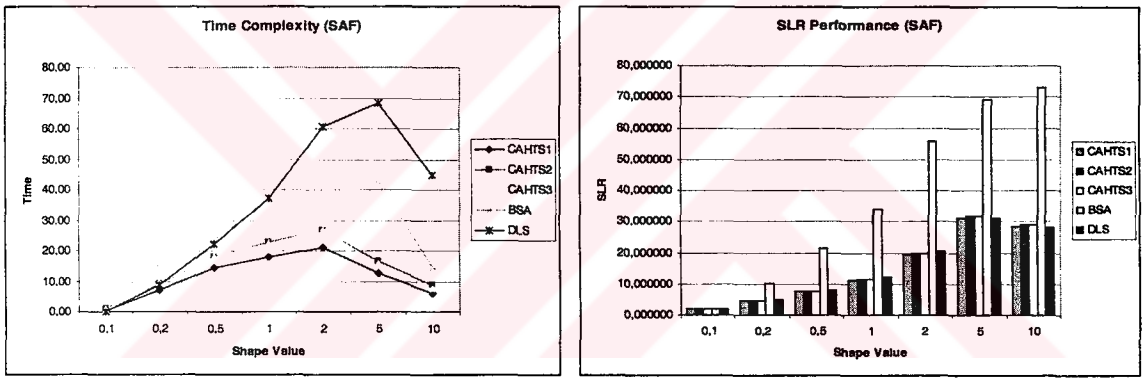


Figure IV.27.a,b: Comparison results of algorithms implemented with store-and-forward technique scheduled on arbitrarily connected topologies with respect to shape value.

The results are very close to the above ones when they are repeated on arbitrarily connected topologies.

IV.2.8 Performance with respect to Processor Connectivity:

In an arbitrarily connected topology degree value represents how many neighbors does a processor has. A low degree value creates a topology that is loosely connected and consequently less number of links between processors. We designed a test that will exploit the algorithms' performances with respect to degree value. In this test we created 12000 application graphs and scheduled them on 100 different topologies with changing degree values. The results of the test are given below in Figure IV.28. On lowly connected

topologies because the communication of messages gets crowded on limited number of links which in turn lengthens the schedule-length. CAHTS1 is the best of all algorithms for both communication implementations. Specifically CAHTS1 outperforms DLS by %13.5 and BSA by %65 for cut-through message transfer technique while these values are %18 and %77.

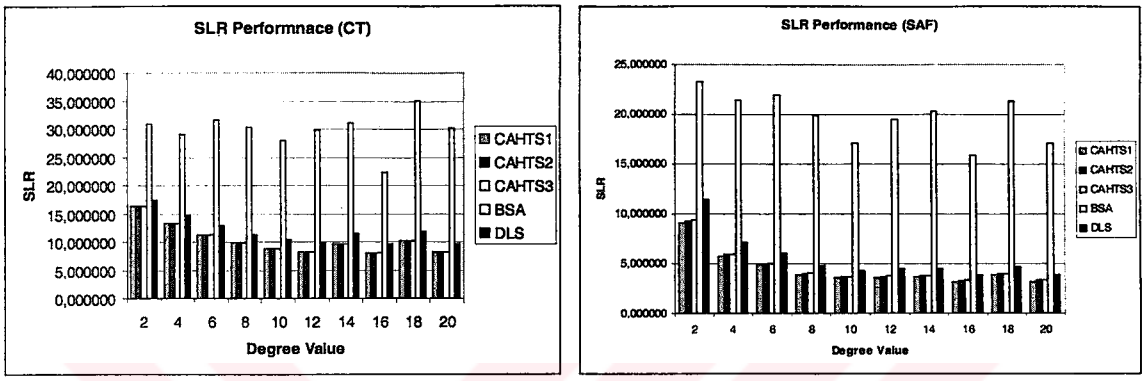


Figure IV.28.a,b: Comparison results of algorithms implemented with both technique scheduled on arbitrarily connected topologies with respect to degree value.

IV.2.9 Performance with respect to Number of Processors:

One of the most aspect of algorithm comparison involves comparing the algorithms with respect to number of processors. We tested the algorithms to schedule 100-noded tasks on topologies with 2,4,6,8,10,12,14,16,32 and 64 processors. The results are given below in Figure IV.29. The results show us that CAHTS1 schedules the application graphs with lowest SLR value for all processor values.

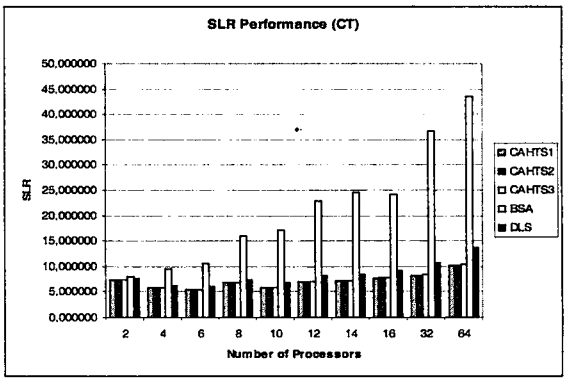


Figure IV.29: Comparison result of algorithms implemented with both technique scheduled on arbitrarily connected topologies with respect to number of processors.

The average value that CAHTS1 outperforms DLS is %13 and BSA is %57. When the algorithms are repeated for other topologies and store-and-forward implementation versions we obtained the same ranking among algorithms and close performance values.

PART V: CONCLUSION

In this thesis study, we firstly developed a new model for the NP-Complete problem of task-scheduling. Scheduling problem consists of scheduling tasks of an application whose characteristics are defined in above parts on a group of connected processors. The goal is to minimize the finish time of the exit task such that the precedence constraints are satisfied. The model includes arbitrarily connection of networks task heterogeneity, processor speed heterogeneity and links' media speed heterogeneity.

After defining the model we introduced message scheduling issue as the most significant element of the task-scheduling problem. In the thesis we studied two methods of message transfer used in the literature. Applying these methods in the study with all realistic details are summarized.

To be applicable in a practical environment, a scheduling algorithm needs to address a number of challenging issues. It should exploit parallelism by identifying the task graph structure, and then take into consideration task granularity, load balancing arbitrary task weights and communication costs, the number of target processors available, and the scheduling of inter-processor communication. The new model defined in the thesis is the most realistic model defined ever in the literature. Comparing the algorithms in such a realistic environment will give the best results for the performances of the algorithms.

The aim of this study is to propose new task scheduling algorithms that are designed to cope with all of the challenges above and consider link contention, message routing, support for arbitrarily connected networks, heterogeneous task execution and heterogeneous link capacities. We proposed the CAHTS algorithm that manages to cope with all the details of the system given above. Our scheduling algorithm is compared with formerly proposed algorithms and their performance results are shown. The compared algorithms are the BSA and DLS algorithms.

The results of the study show us that the new algorithm outperforms BSA and DLS in all graph types and in all network topology environments when compared for SLR performance. The proposed algorithms outperform other algorithms in terms of running time when store-and-forward message scheduling technique is used. For cut-through message scheduling technique our algorithms have very close running time complexity when compared with the DLS and outperforming in most conditions. CAHTS1 version of the algorithm is

seen as the best one among the three versions. The best point of the algorithm is that it shows best SLR performance among all.

The future work of the study is to develop a new algorithm that aims to use a learning routing algorithm which has the ability to learn and change the path according to current conditions of the links. This kind of a routing algorithm embedded in the scheduling algorithm will probably extend CAHTS algorithm's performance since it will help the scheduling algorithm to use idle time slots on routes between processors. This part of the study is also a crucial part of the thesis since there is no study in the literature about using a learning-based routing strategy for communication scheduling.

REFERENCES

[1] Gilbert C. Sih, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures", IEEE Transaction on Parallel and Distributed Systems, Vol 4, No 2, February **1993**.

[2] E.G. Coffman, Computer and Job-Scheduling Theory, Wiley, New York, **1976**.

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

[4] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," Annals of Discrete Mathematics, no. 5, pp. 287-326, **1979**.

[5] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," Operations Research, vol. 19, no.6, pp. 841-848, Nov. **1961.**

[6] Micah Adler, Arnold L. Rosenberg, Ramesh K. Sitaraman, and Walter Unger, "Scheduling Time-Constrained Communication in Linear Networks", Proceedings of 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA) **1998**.

[7] M. Iverson, F. Ozgüner, G. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environments," Proc. Of Heterogeneous Computing Workshop,93-100,**1995**.

[8] H. El-Rewini and T.G. Lewis, "Scheduling parallel Program Tasks onto Arbitrary Target Machines," Journal of Parallel and Distributed Computing, vol. 9, pp. 138-153,**1990**.

[9] H. Singh, A. Youssef, "Mapping and Scheduling Htereogeneous Task Graphs Using Genetic Algorithms," Heterogeneous Computing Workshop,**1996**.

[10] P. Shroff, D. W. Watson, N.S. Flann and R. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments," Proc. Of Heterogeneous Computing Workshop, **1996**.

[11] H. Topcuoglu, S. Hariri, and M.-Y. Wu. "Task scheduling algorithms for heterogeneous processors." Proceedings of the 8th Heterogeneous Computing Workshop, pp. 3-14, San Juan, Puerto Rico, April **1999**.

[12] Yu-Kwong Kwok, Ishfaq Ahmad, "Bubble Scheduling: A Quasi Dynamic Algorithms for Static Allocation of Tasks to Parallel Architectures," Proceedings of 7th Symposium on Parallel and Distributed Processing, pp. 36-43., October **1995**.

[13] L. Wang, H.J Siegel, and V.P. Roychowdhury, " A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing environments," Proc. Of Heterogeneous Computing Workshop, **1996**.

[14] Yu-Kwong Kwok, Ishfaq Ahmad, "Link-Contention-Constrained Scheduling and Mapping of Taks and Messages to a Network of Heterogeneous Processors," **1999**.

[15] Benjamin S. Macey, Albert Y. Zomaya, "A performance Evaluation of CP List Scheduling heuristics for Communication Intensive Task Graphs," in the Proceedings of the Joint 12th International Parallel Processing Symposium/9th

Symposium on Parallel and Distributed Processing, pp. 538-541, March 30 - April 3, Orlando, Florida, **1998**.

[16] T.L. Adam, K.M. Chandy and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," Communication of the ACM, 17:685-9,**1974**.

[17] B. Kruatrachue, "Static Task Scheduling and Grain Packing in Parallel processing Systems," PhD Thesis, Oregon State University, **1987**.

[18] M.Y. Wu and D.D. Gajski, "Hypertool: Aprogramming Aid for Message-passing Systems," IEEE Trans. On Parallel and Distributed Systems, vol. 1 no. 3,pp. 330-343, Jul. **1990**.

Resume

ALI FUAT ALKAYA
Esenkent Mah. Kanipasa Cad.
Saadet Apt. No: 124 D: 13
Maltepe ISTANBUL

Telephone Home:0216 376 40 99, Cellular:0532 502 06 89
e-mail: falkaya@eng.marmara.edu.tr, falkaya@hotmail.com,

CAREER OBJECTIVES

Accomplishing challenging tasks in a competitive environment in accordance with the organizational objectives during a long term career path with the help of my mathematical and programming background. Adopting and applying modern techniques of programming to academics by combining theory, experience and dynamism.

EDUCATION

1998-...	**Marmara University, Faculty of Engineering, Istanbul** Computer Engineering M.S.
1994-1998	**Koç University, Istanbul** Mathematics BS
1987-1994	**Private Fatih Boys High School, Istanbul**
1982-1987	**Sisli Kaptanpasa Primary School, Istanbul**

INDEPENDENT STUDY and THESIS TOPICS

Fall 1997-Spring 1998	Differential Geometry properties of some spaces
2000-2001	Task Scheduling Techniques for Arbitrary Network Topologies

AWARDS AND HONORS

Fall 1997	Dean's Honor Roll
Fall 1996	2nd in the football tournament

WORK EXPERIENCE

Summer 1997	Lab assistant at Koc University CIT Dep.
1998-...	Research Assistant in Marmara University, Faculty of Engineering Computer Engineering Department
1998-...	Instructor in Computer Usage Certificate Programs teaching MS Word 2000, MS Excel MS 2000, MS PowerPoint 2000
2000-...	Instructor and manager of Certificate Program Web Page Design with MS FrontPage 2000
1999-...	NT 4.0, Windows 2000 Advanced Server Administration NT Domain and Active Directory Administration
1999-2001	Lecturing Computer Programming with Pascal courses in the faculty

PROFESSIONAL SKILLS

English: fluent (TOEFL : 623)
German: fair
GRE: 780, 710, 360 (Q, A, V scores respectively)
Experienced in MS Office 2000 (Word, Excel 2000, PowerPoint)
MS Access 2.0/97 MS Visual Basic 5.0, Fortran 77/90
MS Visual C++, C++ Builder 4.0, Turbo C, ASP
Delphi 4.0/5.0, Turbo Pasçal, Matlab 5.0

C Programming in Unix, Linux environment.

RESEARCH INTERESTS

Currently involved in thesis study titled "Task Scheduling Techniques for Arbitrary Network Topologies"
Computational Applications of Math to Business and Information Systems
Software Development & Information Systems
Networking, Task Scheduling and Parallel Processing, Routing Algorithms, Artificial Intelligence, Fuzzy Logic, Computational Theory,

PUBLICATIONS

Haluk Topcuoğlu and Ali Fuat Alkaya, "Heuristics for Scheduling Combined Task Graphs on Heterogeneous Processors", (not published yet)

MS Frontpage 2000 Ders Notları (Prepared for Web Design course)

PERSONAL

Born on 1 April 1976, Hatay
Married
Quick and precise decision maker
Leadership Quality
Good Organizing Skills
Team-player
Good Communication Skills
Can work under pressure or under other difficult circumstances
Can understand and work with all kinds of people
Can learn and adapt quickly and successfully
(At spare times) increase programming experience, go to cinema, concerned in daily technological developments especially in computer.

REFERENCES

Available upon request