

KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

COLLEGE OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY

GABRIEL OSEI-TUTU

**OPTIMIZING THE PROCEDURES FOR THE MOVEMENT OF GOODS IN THE
ECOWAS SUB REGION USING MULTITHREADED ALGORITHMS.**

**A THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE, KWAME
NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI. IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE**

OF

MPHIL. COMPUTER SCIENCE

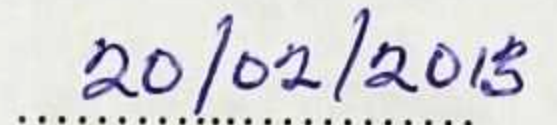
AUGUST , 2012

DECLARATION

I hereby declare that this submission is my own work towards the MPHIL and that to the best of my knowledge, it contains no material previously publish by another person nor material which has been accepted for the award of any other degree of the University, except where due acknowledgement has been in the text.

Osei-Tutu, Gabriel





(PG5091510)

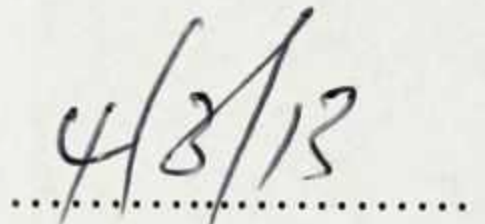
Signature

Date

Certified by :

DR. J.B. Hayfron-Acquah





(Supervisor)

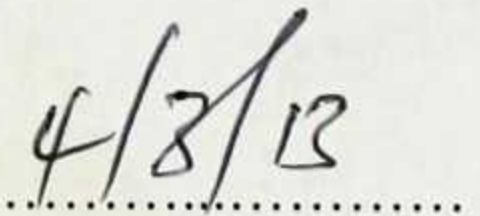
Signature

Date

Certified by:

DR. J.B. Hayfron-Acquah





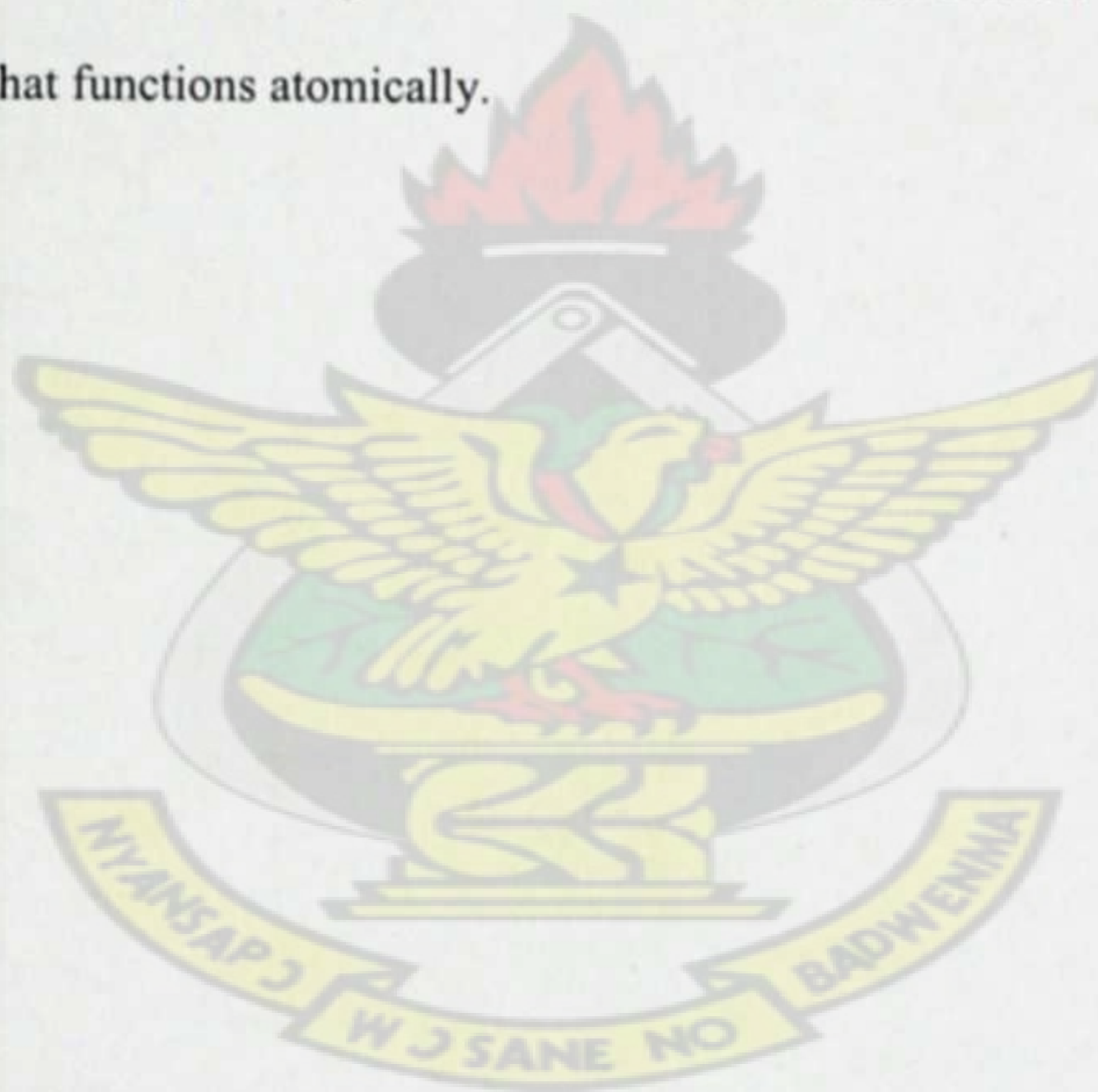
(Head of Department)

Signature

Date

ABSTRACT

In our work we modeled the procedures for the movement of goods using the maximum network flow problem. We used the 'generic' push reliable algorithm as the procedure responsible for the movement of goods. We then used a computational directed acyclic graph with the intuition of vertices as States within the ECOWAS and edges as communications between them. Direct as paths from one State to another. Acyclic as there are no loops in the movement of goods in the transit trade. The computations at the vertices are memoized and are recalled recursively. We implemented the resulting sequential algorithm (Transit Algorithm) by multithreading it using the lock free multithreaded method which results in an interleaving model that functions atomically.



DEDICATION

To my wife Esther and my children Angelo, Jessica, Gabriela, and Michael for the patience and space you gave me. Much of the time I was to be with you I denied you but to my research.

KNUST



TABLE OF CONTENTS

DECLARATION	ii
ABSTRACT	iii
DEDICATION	iv
TABLE OF CONTENTS	v
LIST OF TABLES	vii
LIST OF FIGURES	viii
 CHAPTER ONE	 1
INTRODUCTION	1
1.1 Background	1
1.2 Statement of problems	3
1.3 Objectives of the study	4
1.3.1 The main objective	4
1.3.2 The specific objectives	5
1.4 Hypothesis	5
1.5 Justification of study	5
1.6 Scope of the study	8
The scope of the study is all the transit modules for the Customs Administration Systems in the ECOWAS sub region.	8
1.7 Limitations	8
1.8 Organization Of The Study	9
 CHAPTER TWO	 11
LITERATURE REVIEW	11
2.1 Introduction	11
2.1.1 Facilitating Regional Transport and Transit	13
2.1.2 Progress of Implementation:	16
2.1.3 Trade Flux And Transit Facilitation Challenges In West Africa	16
2.1.4 West Africa Transit trade flows characteristics	18
2.2 Review of relevant literature	25
2.2.1 The Existing System	26
2.3 Theoretical Framework	34
2.3.1 Maximum-flow minimum-cut Theorem	34
2.3.2 The push relabel Method	38
2.3.3 The Blocking flows method	40
2.3.4 Heuristics	40
2.3.5 Parallel algorithms for the maximum flow problem.	41
2.3.6 Summary of Worst case bounds of algorithms in the theoretical framework	42
2.3.7 Multithreading	43
 CHAPTER THREE	 56
METHODOLOGY	56
3.0 Introduction	56
3.1 Study Area	58
3.2 Design of study	61
3.2.1 Initialize Preflow	65
3.2.2 Discharge Pseudocode	68
3.2.3. The Pseudocode of the push operation	70
3.2.4 Recursion of computation dag	74

3.2.5 Parallelizing the Transit Sequential Algorithm	77
3.2.6.Parallelizing the push and the relable methods	81
3.2.7 Parallelizing the Computation Dag.	82
3.3 Variables	83
3.3.1. Parallel Global Updates	83
3.3.2 .Concurrent Global-Update	83
3.3.3 Computational Dag Global Variables (Edges) And Private Variables (Vertices)	84
3.4 Methods of Analysis	84
3.4.1 Informal approaches to correctness	85
3.4.2 Graph framework	86
3.4.3 The Algorithms Framework	87
3.4.4 Sub Domains as packages	89
3.5 Constraints/ Problems	90
CHAPTER FOUR	91
ANALYSIS OF FINDINGS	91
4.0 Introduction	91
4.1 RESULTS	92
4.2 Shared Memory	95
4.3 Distributed Memory	95
4.4 Petri nets model	96
4.5 The Interleaving Model	97
4.5 The dynamic multithreading	98
4.6 Multithreading Computation	99
4.7 Scheduling	100
4.8 The Target Multiprocessor platform	101
4.9 Implementation	102
4.9.1 The Push Relabel Algorithm By Goldberg	102
4.9.2 Implementation of the transit algorithm using Bo Hong's lock-free multithreaded method	104
4.9.3 The Transit Algorithm	106
4.10 Testing Of Hypothesis	110
4.10.1 INTRODUCTION	110
4.10.2 Performance Measure	111
4.10.3 Developing Tools To Analyze Multithreaded Algorithms	115
4.10.4 Greedy Scheduler	119
4.10.5 Analysing The Transit Multithreaded Algorithm	122
4.10.6 The Complexity Bound.	124
CHAPTER FIVE	126
CONCLUSION AND RECOMMENDATION	126
5.1 SUMMARY	126
5.2 CONCLUSION	127
5.3 RECOMMENDATION	127
REFERENCES	128
LIST OF ACRONYMS	134

LIST OF TABLES

Table 1.1 from USAID-WATH	7
Table 2.1 Economic and Demographic Indicators	17
Table 2.2 International trade of Mali	23
Table 2.3 An Implementation of a transit tracking system	33
Table 4.1: 10 worst-case bound algorithms	92
Table 4.2:CPU time taken(in seconds on convex)	92
Table 4.3:. Variable Access Characteristics	105

KNUST



LIST OF FIGURES

Figure 1.1 States making up the Ecowas	3
Figure 1.2 West Africa- First Priority Corridors Check points, Bribes and Delays	6
Figure 2.1 Maritime Transit for Burkina Faso, Mali and Niger	19
Figure 2.2 Total Transit for Burkina Faso, Mali and Niger	20
Figure 2.3 Maritime Transit for Burkina Faso	21
Figure 2.4 Maritime Trade of Burkina Faso (CBC)	22
Figure 2.5 Maritime Transits for Mali	22
Figure 2.6 International Trade of Mali by Corridor	24
Figure 2.7: 3-Tier architecture of the existing system	26
Figure: 2.8 Data layer levels	27
Figure: 2.9 Interfacing for Ecowas	28
Figure 2.10: Cargo tracking data aggregation	29
Figure 2.11: ISRT transit operation	31
Figure 2.12 Flow network G and flow f	37
Figure 2.13: Super Source and Super Sink	38
Figure 2.14 : The parallelization steps	48
Figure 2.15: Six possibilities of interleaving of push and lift operations	54
Figure 2.16: Sequence of saturation pushes on (u,v)	54
Figure 2.17 Single Declaration Document Transit Model	55
Figure 3.1: Transit Model	57
Figure 3.2: Flowchart of Transit algorithm	64
Figure 3.3: Flowchart of initialize preflow (G,s)	68
Figure: 3.4 Discharge Flowchart	70
Figure 3.5: Push (u,v) Operation flowchart	71
Figure 3.6 Flowchart of Reliable Operation	73
Figure 3.7: Flowchart of Computation Dag	76
Figure 3.8: UML Representation of Recursion	77
Figure 3.9: Flowchart Of Parallelizing Transit Algorithm	78
Figure 3.10: Graph Framework	88
Figure 3.11: Object Oriented representation of parallel algorithm	89
Figure 4.1 A computation direct acyclic graph (TRANS N)	94
Figure 4.2: A Directed Computation Dag Representing Trans (4)	112
Figure 4.3: Work and span of composed sub computations	122

CHAPTER ONE

INTRODUCTION

1.1 Background

Before the formation of ECOWAS the movement of goods across countries in the ECOWAS sub-region was the responsibility of that countries Customs department and for that matter the procedures governing customs clearance. For instance in Ghana goods are deemed not to be in Ghana unless the necessary regimes governing the movement of goods to Ghana i.e Direct Import, Re-Importation, Temporary Importation etc are adhered to. The Customs jargon Entered could be used to attest to the fact that movement is synonymous with Customs procedures. For goods to enter into Ghana the Bill of Entry otherwise known as Declaration form is filled and submitted to the Customs department. Another example of movement being synonymous with Customs procedures, that is given a free zones status in a free zone enclave for a company in Shangkai in the People's Republic of China in Dzorwulu a suburb of Accra in Ghana, the company after the production of goods can enjoy the local market unless the goods move into Ghana by going through the Direct Import for Home use regime. The various countries in the sub-region also have these procedures and movement of goods, through their countries were governed by these procedures. Goods for instance in transit from Ghana to Niger would have the Bill of Entries of Ghana then that of Burkina Faso on entering into that country and finally that of Niger the country of destination. In order to have a common policy about the movement of goods the Economic Community of West African States (ECOWAS) was formed, and the policy became one of the major cardinal points of the Union. By the research of Dr Sadok Zerelli et al, [1] West African Road Transport and Transit Facilitation Strategy, we can conveniently say that the Economic Community of West African States (ECOWAS) was established in 1975 through the Lagos treaty with the aim to promote economic development through cooperation among its members. It proposes a staged

approach towards a Customs Union, elimination of Customs duties between partners. Establishment of a common external tariff elimination of obstacles to free movement of capital services joint development of transport infrastructure, harmonization of economic policies. In West Africa, inter-state road transport and transit schemes are governed by two agreements. The Inter-State Road Transport Convention, which handles the technical norms and conditions to fulfill in order to participate in interstate road transport of goods and set the itineraries to use. The Inter-State Road Transport Convention which treats inter-state road transit issues more specifically. In effect Ecowas member states adopted the ISRT scheme under convention A/P4/82 on 29th May 1982 in Cotonou, further completed on 30th May 1990 in Banjul, which contains the ISRT operations guarantee mechanism, the mechanism mainly aim at removing successive customs procedures throughout the different transit countries. Facilitating the movement of vehicles and transported goods among member States. Enabling the collection of regular, and reliable statistics of interstate road transit goods. The implantation of the convention depends on three basic conditions. The issuance of a single, coincide ISRT declaration form, the establishment of a guarantee fund that shall serve as security, the standardization of license vehicles according to defined criteria, indestructible and sealing. This is the context under which Ecowas decided to develop synergies and define a regional programme to facilitate inter-state road transport and transit of goods.

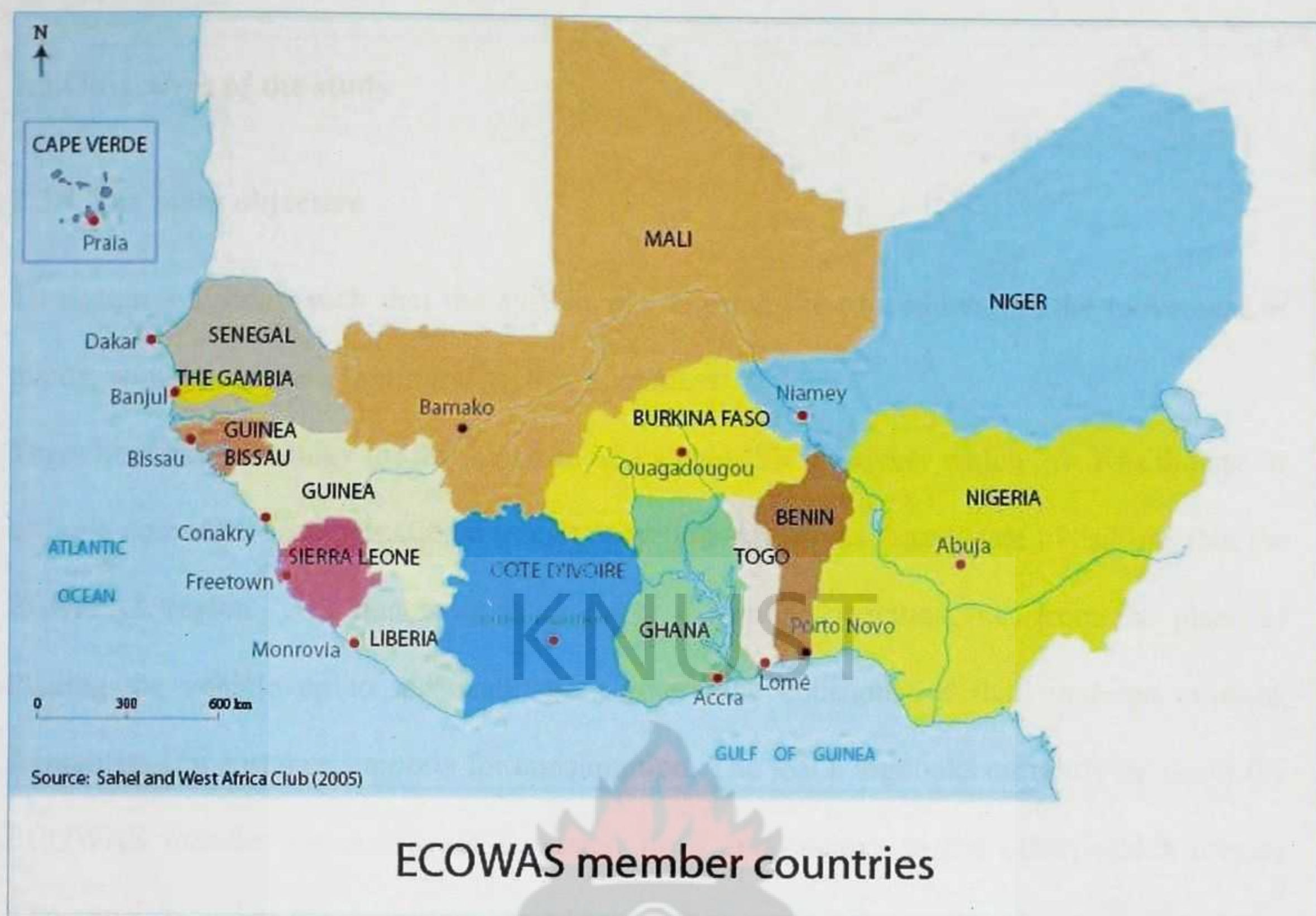


Figure 1.1 States making up the Ecowas

Source: West Africa Road Transport And Transit Facilitation Strategy

1.2 Statement of problems

In order to produce a single declaration document for the movement of goods then data should mean the same thing throughout the system. Data sets and data formats that should be exchanged should be agreeable by all the countries. Data integrity is not compromised. Also the data exchange is permissible by law.

1.3 Objectives of the study

1.3.1 The main objective

To design a system, such that the system will execute the procedures for the movement of goods, would produce a single transaction document.

There has been a strategy for the adoption of a single ISRT voucher which involves the use of a single document for international transit operations known as “inter-state transit”, within the ECOWAS region. This document is to cover the entire operation, i.e. from the place of loading the vehicle up to the destination, where the consignment shall undergo customs formalities (for instance, imports for consumption, The ISRT logbooks currently in use in the ECOWAS member states are not the same from one country to the other, which creates difficulties in using the document. In addition, they are poorly produced, which introduces doubt as to their validity and even authenticity. Thus, most of the ISRT logbooks issued in Mali are not the “real” ones distributed by the Chamber of Commerce (which are identifiable by their “sticker”).

Therefore, there is need to harmonize the logbooks and also homogenize their production conditions. The format and texture of the vouchers are totally obsolete and hamper the proper computerization of the procedure, which is under the supervision of the Customs Service and Guarantee Fund. The harmonization of the logbook (and its leaflets) according to international standards and recommendations is a prelude to reactivating the scheme (placement in the customs warehouse, upgrading or temporary entry). This means that several conditions have to be met. In that regard, from Dr. Zadok et al [1] it is appropriate to ensure that the ISRT logbook matches the TIR model, adopted by the International Road Transport Union (IRU) as much as possible.

1.3.2 The specific objectives

To design a parallel computational model system, using the maximum flow network problem to model the procedures for the movement of goods by way of push relabel and multithreading algorithms.

1.4 Hypothesis

The proposed parallel computational model would provide an abstract method for modeling the transit procedure, using the network maximum flow problem and multithreading, which will enable the realization of a single transaction documentation that will lead to efficient service delivery.

1.5 Justification of study

A survey conducted by Dr Zadock et al, [1] shows the valuable time wasted as a result of this illegal road blocks and fees collected as depicted in figure 1.2 For a round trip between Tema or Lome to Bamako and return, C&F agents are indicating an average of 41 days, out of which only 9 are spent driving (the rest is corresponding to various waiting times). A better knowledge of the other sources of delays would complete the scope of the transport observatories. According to figures quoted from clearing and forwarding agents, border crossing delays and terminal delays during the clearing of the goods are representing a high proportion of the total round.

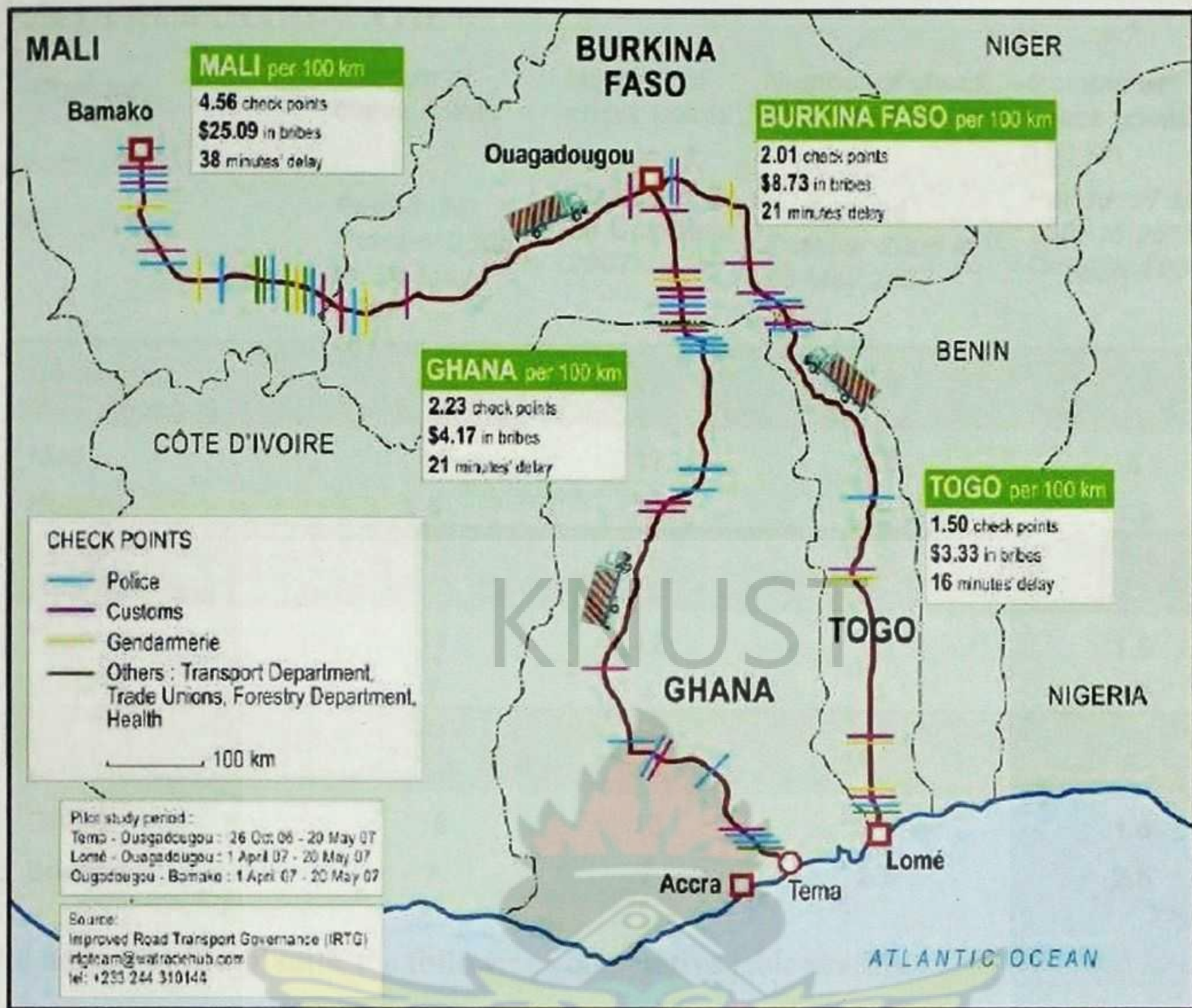


Figure 1.2 West Africa- First Priority Corridors Check points, Bribes and Delays

Source: West Africa Road Transport And Transit Facilitation Strategy

For all corridors, they are greater than the pure driving time. Number of illegal checkpoints.

Table 1.1 from USAID-WATH

Corridor		Number of check points <i>Period :26 October 2006 to 26 May 2007</i>	Number of check points <i>Period : 27 May 2007 to 26 October 2007</i>	Number of check points per 100 km <i>Period : 26 October 2006 to 26 May 2007</i>	Number of check points per 100 km <i>Period :27 May 2007 to 26 October 2007</i>
Bamako Ouagadougou	-	24	25	2.5	2.5
Mali		19	17	4.5	4
Burkina		5	8	1	1.6
Lomé Ouagadougou	-	18	16	1.7	1.5
Togo		11	12	1	1.6
Burkina		7	4	4	1.5
Téma Ouagadougou	-	25	20	2.5	2
Ghana		18	13	2.2	1.6
Burkina		7	7	2.5	2.5

With regard to illegal tolls, the following comparative table reveals:

These individual customs procedure of member states summed up to a prolonged customs procedures affecting trade. At each crossing there is interruption of transit regime reported by World Customs Organization [2]. As the clearance for home consumption becomes mandatory, a bond system which is only valid for the country of origin is executed, additional mechanisms such as escort for monitoring, and each country concerned with transit will have to have his own bond system. This bond is the tax that have to be paid by the trader should the goods fail to exit the country that granted the bond.

The construction of a joint border post by member States of Ecowas. With the proposed system which is this paper, will promote a joint border post which would render an efficient service to traders, this is because the processing of the various procedures would happen at the vertiees and could be coordinated among themselves at the same location. Illegal road blocks would be eliminated and its attendance extortions of illegal fees would be a thing of the past.

Border crossing delays and terminal delays during the clearing of the goods which represent a high proportion of the total round trip time for all corridors, would be less than the pure driving time. Introduction of single document system, harmonization of the guarantee system for inter-state transit operations data exchange, that is how data be requested, how data will be formatted for exchange, where it will be hosted awaiting pickup and be consumed have been addressed by the parallel computational model being proposed as the system abstracts the parallel architectural model that takes care of this concurrency platform and also by means of this joint border post. On the proposed system data definitions would be constant and as a result data would mean the same thing on every vertex thereby automating data exchange. For example since the system is a maximum network flow a field like the declaration number in one vertex carry the same connotation as the field for the declaration number in another vertex. There would not be difficulty in upgrading the system

Ecowas vision of the elimination of Customs duties between partners, the elimination of restrictions, the establishment of a common external tariff, harmonization of economic issues, and the elimination of obstacles to free movement of capital and services would not affect the running of the system should any of these mentioned policies changes.

1.6 Scope of the study

The scope of the study is all the transit modules for the Customs Administration Systems in the ECOWAS sub region.

1.7 Limitations

The parallel random access machine has four levels of abstraction. We are using the parallel computational model for this project. The rest, namely parallel machines model, parallel architecture model and parallel programming model could not be covered in this project due to time constraint . As a result they are out of scope.

1.8 Organization Of The Study

This study is organized into five chapters.

Chapter one: Introduction

The background in this chapter provides the historical background to the targeted period of research thereby relating practices preceding the targeted period to form the background for the main work. The reason for the formation of Ecowas stem from the fact that each state was an island to itself and as such could not facilitate the movement of goods through its territorial borders. Therefore doing business in the sub region was cumbersome, time wasting and not profit worthy. Unnecessary road blocks, illegal collection of fees, the execution of individual procedures, executions of bonds covering the goods, and lack of coordination of information about the transport which carry the goods between the participating countries of the transit transaction hampered trade. The main objective of this paper is to design a system that would process all the procedures of the transit transaction and present it on a single document in an efficient manner. This leads to specifics of using maximum flow problem to model the procedures at each state and also to provide an abstract of an architectural model for implementation. A parallel computational model that would perform efficiently on a real computer given its running time is the hypothesis of this paper. This paper is justified by way of the advantages of providing a joint border post with its attendance advantages, harmonization and simplification of regulations and procedures and the ability of the system by way of its scalability and adaptability. This project will involve the maximum network flow problem, that will model the transit procedures seamlessly and the multithreaded aspect that would take advantage of the multicore technology now available to ensure an efficient system. Limitations, by way of time constraint in treating the rest of the parallel random access machine models.

Chapter two Literature review

Deals with literature review. Review of relevant literature. That is literature connected with the subject we are discussing. We are discussing the single ISRT documentation in the Ecowas sub region. The theoretical framework is concerned with the ideas and principles on which a particular subject is based. Here we are discussing how to use the maximum network flow problem to solve the research problem. We deliberated on the use of the preflow push relable method. Our investigation reveals the generic push relabel algorithm from the push relable method. The computational directed acyclic graph is reviewed as well as multithreading and finally we summarized the chapter and concluded on it.

Chapter three Methodology

Methodology is organized first, by way of introduction to the methodology used in this paper. The study area follows suit and provides the philosophical foundation of inquiry of this research paper. Formative research of the study design comes next having an important and lasting influence on the development of the model which is this paper. The variables which are qualitative assumptions come next while the methodical issues like the informal method of analysis of this paper follows and lastly the constraints and problems.

Chapter four Analysis Of Finding

This chapter starts with introduction of the analysis of findings and it is followed by the implementation of the design of the study. Implementation is used for coding and it is also equally right to use programming as well as development for coding too. The testing of the hypothesis followed. The testing is by way of determining the termination of the Transit algorithm, the runtime or the upper bounds of the algorithm.

Chapter five Conclusion and Recommendations

This chapter deals with the summary of the analysis of findings and the conclusions arrived at. Finally recommendations from the research is given for subsequent follow up.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

The organization responsible for collecting and accounting for duties and taxes on imports exports and local manufactures world-wide especially the developing countries are Customs Administration. Assessment, collection/accounting and protecting the revenue which are under the field of Customs and Excise are the revenue functions of Customs Administration. With the intention to determining the amount of duties and taxes payable a critical examination of the documents and goods are assessed by customs officials. Collection and accounting of monies taken which are paid into the banks and subsequently accounted for are eventually transferred into Government chest with the accountancy connected therewith. Preventing and detecting evasions of the revenue laws referred as protection are the duties mentioned earlier of Customs Administration. Duties and taxes of goods, are only applicable to goods which are in the country for use or consumption. Depending on the particular policy of a country goods may attract various rates of duties. For example in Ghana, the government want to promote export trade, most of the goods that originate from Ghana that enter into international trade are zero rated. Goods moving from one country called the country of origin to another, in this instance called country of destination and intermediate countries have roles to play to ensure the safety and the protection of revenue of those goods. Countries particular in the West Africa community trade, with each other and also serve at times as conduit for goods meant for their neighbors to get to them. It came therefore as no surprise when countries in the West Africa sub region decided to form a union. One of the cardinal points of that union was to promote free movement of goods and services. The Economic Community of West African States was established in 1975 through the Lagos treaty. It proposed a staged approach towards a customs Union accordingly from Dr. Zadok et al [1]:

- . elimination of customs duties between partners
- . elimination to restrictions to trade between partners
- . establishment of a common external tariff
- . joint development of transport infrastructure
- . harmonization of economic policies.

The implementation of the Convention depends on three basic conditions:

- The issuance of a single, concise ISRT declaration form at the beginning of the inter-state road transit operations
- The establishment of a guarantee fund that shall serve as security
- The standardization of licensed vehicles according to defined criteria indestructibility and sealing;

This is the context under which ECOWAS and WAEMU decided to develop synergies and define a regional programme to facilitate inter-state road transport and transit of goods. The West African regional transport and transit facilitation programme is therefore based on the projects developed or implemented in that regard by these two institutions. The UEMOA Council of Ministers, at its meeting held on June 27th 2002, adopted two texts related to transport:

- Recommendation 02/2002/CM/UEMOA relating to the simplification and harmonization of port procedures
- Directive 04/2002/CM/UEMOA relating to Shippers' Councils. The Recommendation and the Directive relate more closely to facilitation, and their content is therefore discussed under the relevant section of this chapter. The Recommendation 02/2002/CM/UEMOA relating to the simplification and harmonization of port procedures requests member states:

- to establish National Facilitation Committees
- to ratify the FAL Convention
- to ratify the Kyoto Revised Convention
- to liberalize port handling services
- for Customs, to adopt the principle of advance declaration, and simplified declaration for transshipment and transit traffic
- to avoid duplication of the capture of the manifest between port authority and Customs through the creation of computerized links between the two institutions

2.1.1 Facilitating Regional Transport and Transit

The texts relating to transport facilitation cover the definition of the instruments such as the guarantee regime for transit, or the motor insurance scheme, and also the institutions created to accelerate and monitor their implementation.

Inter-State Road Transport:

The Protocol A/P.2/5/82 on Inter-State Road Transport (also known as "Convention ISRT") is aiming at regulating the conditions of transport by road between member states.

Application of the TRIE Convention is the responsibility of customs services and the consular offices appointed to protect national interests in administering the TRIE guarantee fund, as laid down in the TRIE Convention. However, ratification and actual implementation proved problematic, and the Authority adopted the additional convention A/SP.1/5/90, which is defining a chain of national bodies responsible for the guarantee, with each national body designated by each member state. With a view to limit road checks for transit trucks, UEMOA adopted Directive 08/2005/CM/UEMOA on December 16th 2005. Containers, reefer trucks, tanker trucks, and all compliant trucks (according to the Inter-State Road Transit Convention)

were to be controlled only at departure, arrival, and at border crossings. Other controls are forbidden. The practical modalities of the controls are defined in the Decision 15/2005/CM/UEMOA adopted the same day.

Brown Card Insurance scheme:

The final main facilitation instrument is the Protocol A/P.1/5/82, establishing the ECOWAS Brown Card, a third party motor vehicle insurance. The Protocol is supplemented by the Decision C/DEC.2/5/83 relating to the implementation of the Brown Card scheme.

Joint border posts

The lead for the establishment of joint border posts has been taken by UEMOA, and later absorbed by the ECOWAS. The creation of joint border posts is contained in the resolution 04/97/CM/UEMOA adopting the Action Plan for transport infrastructure.

The EU through the World Bank [3] support to the ECOWAS Regional facilitation programme includes the creation of Joint Border Posts. However, the EU funding is partly conditioned to the achievement of specific milestones, and the Council adopted Resolution C/RES.2/09/08 to assist in the process.

Integrated Facilitation Programme

The Conference of Heads of States and Governments issued Decision A/DEC.13/01/03 dated January 31, 2003, relating to the establishment of a Regional road transport and transit facilitation programme in support of intra-community trade and cross-border movements.

The main components of the Programme are:

- Harmonization and simplification of regulations and procedures (introduction of single document system, harmonization of the guarantee system for inter-state transit operations)
- Construction of joint border posts

- Improvement of a information system by implementing the ACIS model (Advance Cargo information System with Road Tracker and Port Tracker modules) and by creating observatories of abnormal practices along the inter-States roads; The Trans-Coastal Lagos/Nouakchott and the Trans-Sahelian Dakar/Ndjamena corridors are selected for the implementation of the program.

Facilitation Institutions

The National Committees established by the Decision A/DEC.3/8/94 have a scope which includes facilitation. This scope was also confirmed by UEMOA, in the Recommendation 04/97/CM/UEMOA.

Decision A/DEC.9/01/05 dated January 19th 2005 re organised the institutional framework for the implementation of the facilitation programme, by reformatting or establishing three layers of organs:

- National Facilitation Committees (with a revised composition)
- Cross Border Corridor Management Committees
- A Regional Inter-State Road Transport and Transit Facilitation Committee

The Committee, which partly duplicates for UEMOA the functions of the ECOWAS Regional Facilitation Committee, comprises the following representatives from each member state:

- a representative of Customs
- a representative of Police
- a representative of Gendarmerie
- a representative of Plant Health Services
- the Director for Land Transport
- a representative of the National Facilitation Committee
- two representatives from the private sector

2.1.2 Progress of Implementation:

The multiplication of decisions and recommendations did not improve the transport situation in the region. The adequacy of the instruments developed was not questioned, but the identified issue was their effective implementation. The remedy identified was adopting an appropriate governance structure for facilitation issues, and increased coordination between ECOWAS and UEMOA.

This approach was developed and formally approved by the organs of the two RECs during the second half of 2003.

2.1.3 Trade Flux And Transit Facilitation Challenges In West Africa

West Africa context

One of the striking features of the region is the contrast:

- contrast between countries, which comprise small island state and landlocked countries, large and small countries
 - contrast in the population, with one country, Nigeria, representing more than half of the Community population
 - contrast in levels of development, although most countries are classified under the category of the least developed countries
 - contrast in density, from highly populated coastal areas to deserts
- The main economic and demographic indicators are summarized in the table below.

Table: Main indicators (World Bank Development Indicators – reference year 2006 Economic and Demographic Indicators [4])

Table 2.1 Economic and Demographic Indicators

	Population, total (millions)	Population growth (annual %)	Surface area (sq. km) (thousands)	GDP (current US\$) (billions)	GDP growth (annual %)	GDP Per Capita
BENIN	8,8	3,1	112,6	4,8	4,1	544,5
BURKINA FASO	14,4	3,0	274,0	6,2	6,4	429,7
CAPE VERDE	0,5	2,3	4,0	1,1	6,1	2 192,3
GAMBIA	1,7	2,8	11,3	0,5	4,5	307,2
GHANA	23,0	2,1	238,5	12,9	6,2	561,1
GUINEA	9,2	2,0	245,9	3,3	2,8	361,7
GUINEA BISSAU	1,7	3,0	36,1	0,3	4,2	181,8
IVORY COAST	18,9	1,8	322,5	17,6	0,9	928,1
LIBERIA	3,6	3,9	111,4	0,6	7,8	176,0
MALI	12,0	3,0	1 240,2	5,9	5,3	490,4
NIGER	13,7	3,5	1 267,0	3,7	4,8	266,4
NIGERIA	144,7	2,4	923,8	115,3	5,2	797,0
SENEGAL	12,1	2,5	196,7	9,2	2,3	761,4
SIERRA LEONE	5,7	2,8	71,7	1,5	7,4	252,6
TOGO	6,4	2,7	56,8	2,2	4,1	344,8

Source: West Africa Road Transport And Transit Facilitation Strategy

As a continent, Africa south of Sahara is marginalized in the world trade, accounting for a share of 1%, despite the fact that it is one of the most dependent regions on world markets, with a share of 40% of its GDP dependent on exports.

Trade patterns have little changed over the last decades, only in terms of trading partners, with Europe losing its dominance, and the share of Asia growing, while in terms of composition, it remained stable. There are two types of corridors in the ECOWAS area: the gateway corridors linking the hinterland to the main seaports, primarily supporting the overseas trade of the region and marginally the intra-regional trade. Intra-regional corridors, key transport infrastructures and services constituting a pipeline for a mix of regional trade flows.

(www.atlas-ouestafrique.org)

Due to the characteristics of the transport infrastructure of the ECOWAS area, a number of Gateway corridors share common parts. The main corridors are linking the main seaports of

Dakar, Abidjan, Tema, Lome and Cotonou to the landlocked countries of Burkina Faso, Mali and Niger. Significantly, in a number of corridors, Burkina Faso is playing a key role as transit country.

2.1.4 West Africa Transit trade flows characteristics

The information on trade flows in West Africa is scarce and fragmented, sometimes inconsistent. The information compiled in this section has been obtained through various channels:

- official websites of the West Africa port authorities
- direct communications with port authorities
- Shipper's Councils
- Extensive literature review

There are basically two options to approach corridor trade flows, from a port perspective, and from the landlocked countries perspective.

Port transit traffic refers to direct trade between the landlocked countries and overseas partners. However, the corridor trade flows also include bilateral trade with the coastal countries, which may or may not be linked to maritime trade. This adds some confusion in the strict comparison of figures. Whenever possible, several sources have been used and included in this section, in order to enable future comparison of trends by referring to the relevant series.

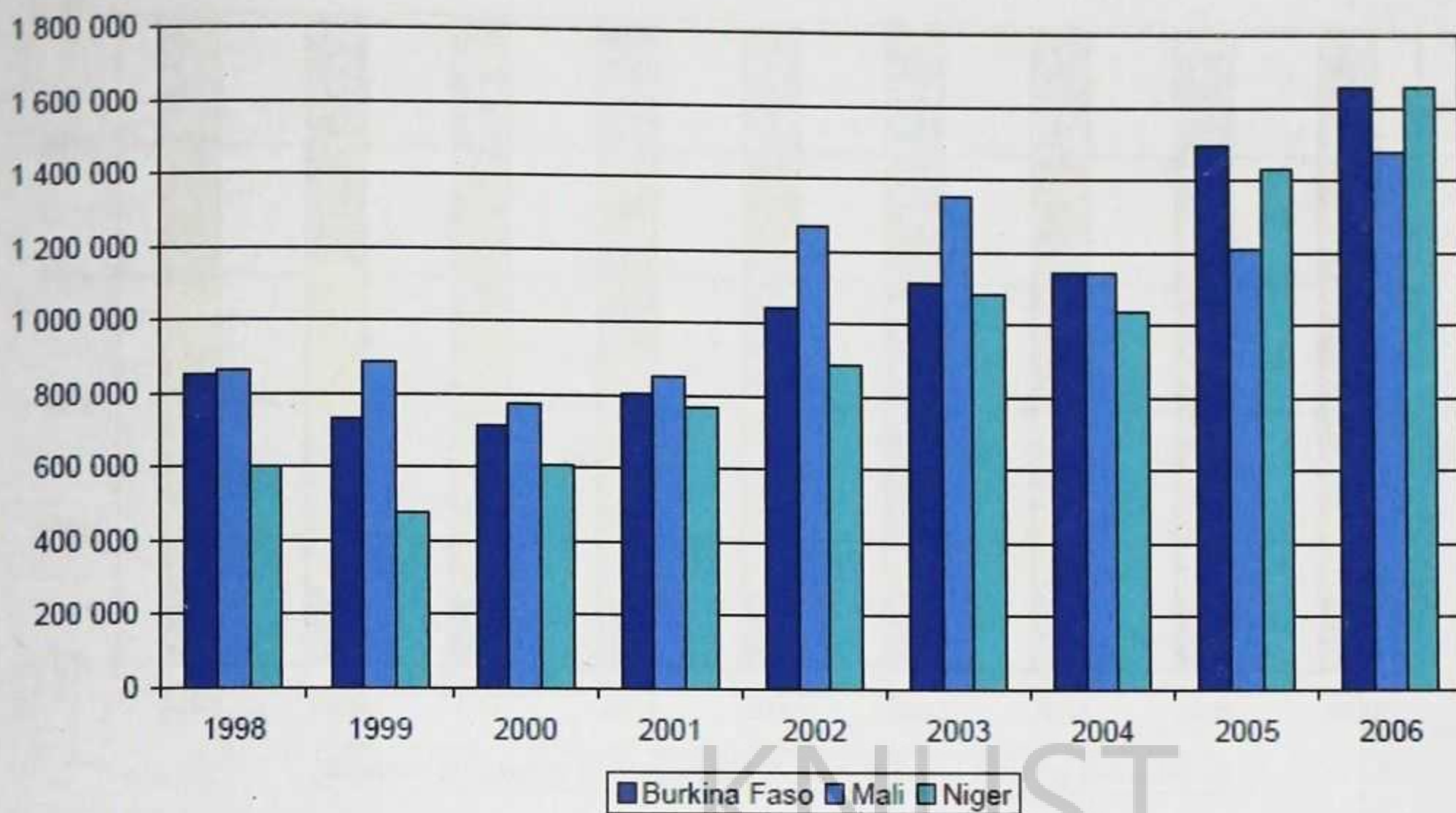


Figure 2.1 Maritime Transit for Burkina Faso, Mali and Niger

Source: World Bank(2007)

The Ivory Coast crisis, starting in September 2002, had a deep impact on the transit patterns, with the diversification of routes for the landlocked countries. However, overall transit volumes for the three landlocked countries of West Africa have not been significantly affected, and the general trend is an increase since the year 2000. On the total maritime trade of the three landlocked countries (Burkina Faso, Mali and Niger) passing through the ports, the respective share of each port is illustrated in the following chart. However, when analyzed from country to country, the evolutions are different.

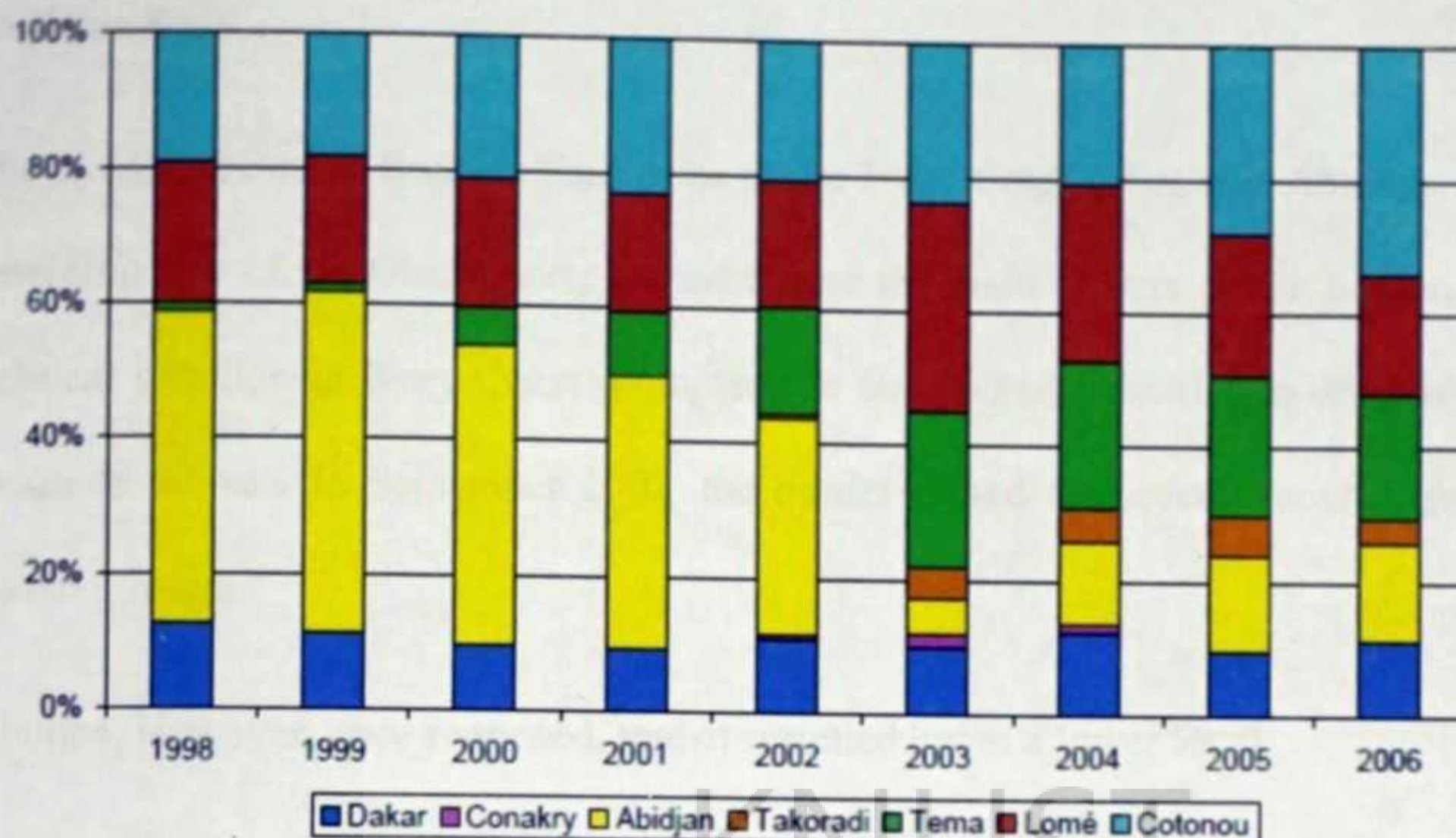


Figure 2.2 Total Transit for Burkina Faso, Mali and Niger

Source: world Bank (2007)

According to Dr. Zadok et al, [1] the effect of the Ivory Coast crisis is clear, and the Ghana ports benefited from the resulting reorganization of the traffic, although the figures reveal that the diversification started before the crisis, with the first signs of political uncertainties. Figures from the Port of Abidjan however indicate that since the signing of the peace agreement in March 2007, transit traffic is sharply picking up again in Abidjan, showing a renewed confidence in the port as a maritime gateway.

Analyzed country per country, the situation is slightly different. Only two countries were relying heavily on Abidjan as maritime gateway, Mali and Burkina Faso. The effect of the Ivory Coast crisis is therefore more apparent than for Niger, for which Abidjan never played a significant role.

The emergence of Takoradi is linked to the saturation of the port of Tema, which had to face almost overnight huge volumes of cargo rerouted from Abidjan. For the port of Conakry, the figures are missing, but it seems that the surge of traffic was short-lived.

Burkina Faso

The main gateway of Burkina Faso prior to the Ivory Coast crisis was Abidjan. However, the emerging role of the Ghana ports started before the main closure of the border, at a time the political situation in Ivory Coast prompted the landlocked countries to diversify their access routes to the sea. In September 2002, the border closed for several months, preventing any transit through.

Abidjan. However, once reopened, transit resumed but at a lower level.

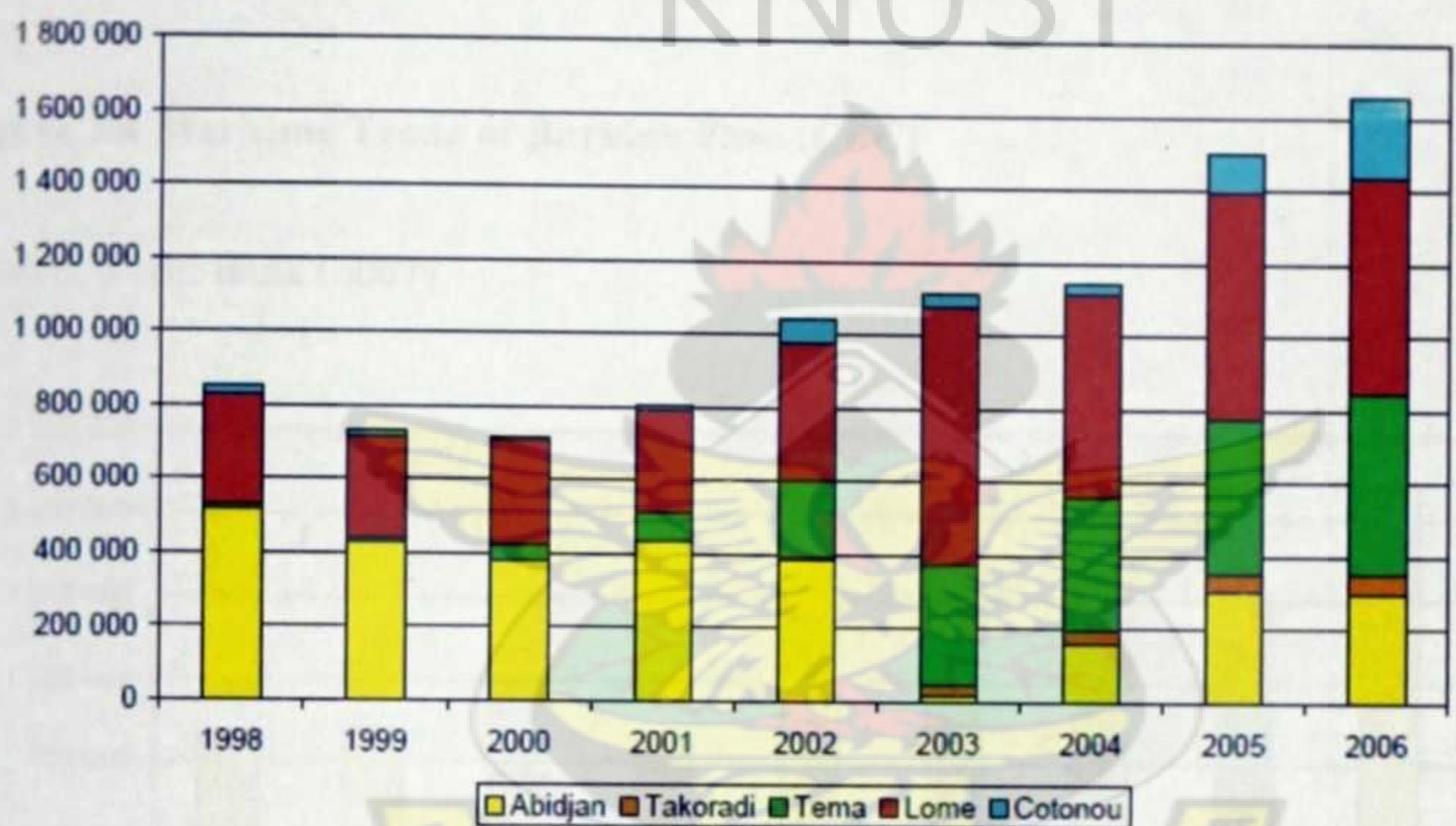


Figure 2.3 Maritime Transit for Burkina Faso

Source: world Bank (2007)

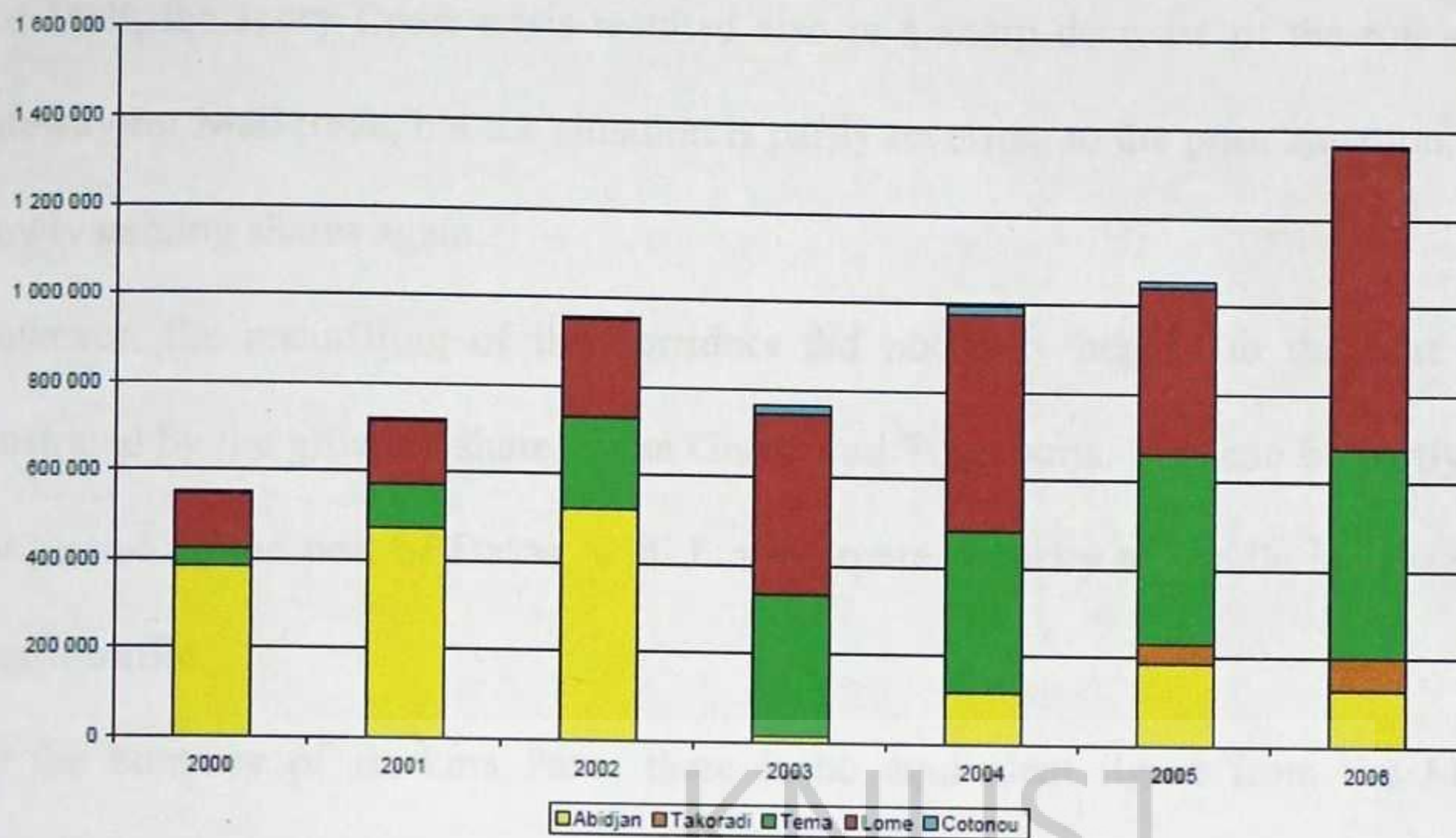


Figure 2.4 Maritime Trade of Burkina Faso (CBC)

Source: world Bank (2007)

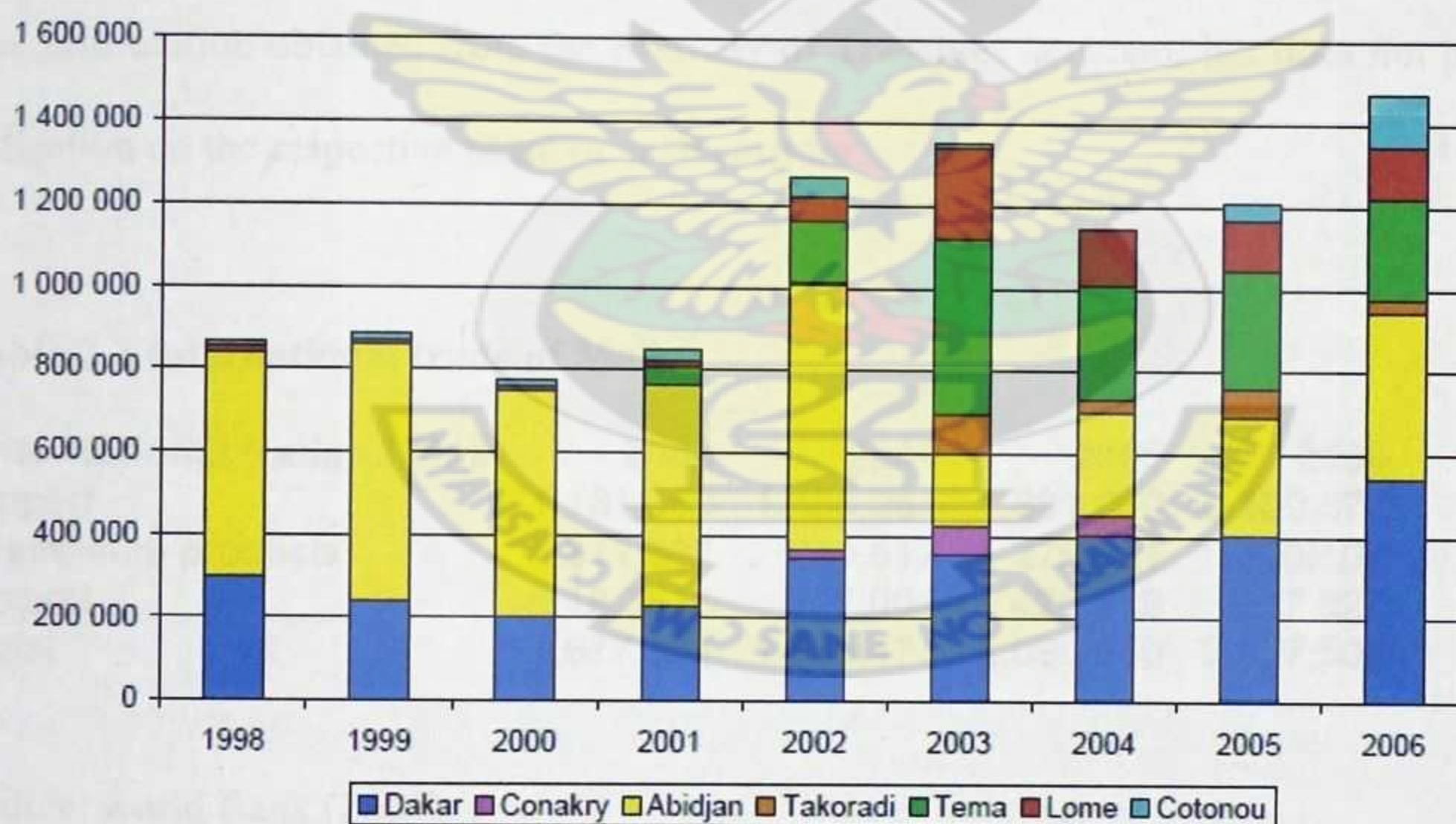


Figure 2.5 Maritime Transits for Mali

Source: world Bank (2007)

For Mali, the Ivory Coast crisis resulted also in a sharp decrease of the role of Abidjan as gateway for Mali trade, but the situation is partly reverting to the prior situation, with Abidjan slowly gaining shares again.

However, the reshuffling of the corridors did not fully benefit to the port of Dakar, as illustrated by the growing share of the Ghana and Togo ports. This can be partly linked to the congestion of the port of Dakar, with limited spare capacity to handle increased volumes of transit traffic.

To the contrary of Burkina Faso, there is no equivalent figure from the Mali Shippers' Council, although two alternative sources can be considered:

- Entrepôts Maliens, a parastatal managing warehouses in the West Africa ports
- The Ministry of Transport, monitoring the international (including regional) trade of Mali.

The information obtained from the Ministry of Transport is recent, but does not provide any indication on the respective share of each corridor.

Table 2.2 International trade of Mali

International trade of Mali	2003	2004	2005	2006
import	1,181,669	1,405,961	1,891,060	2,109,572
Petroleum products	311,837	245,512	279,501	330,107
export	18,3818	227,004	428,289	237,827
total	1,677,324	1,878,477	2,598,850	2,677,506

Source: world Bank (2007)

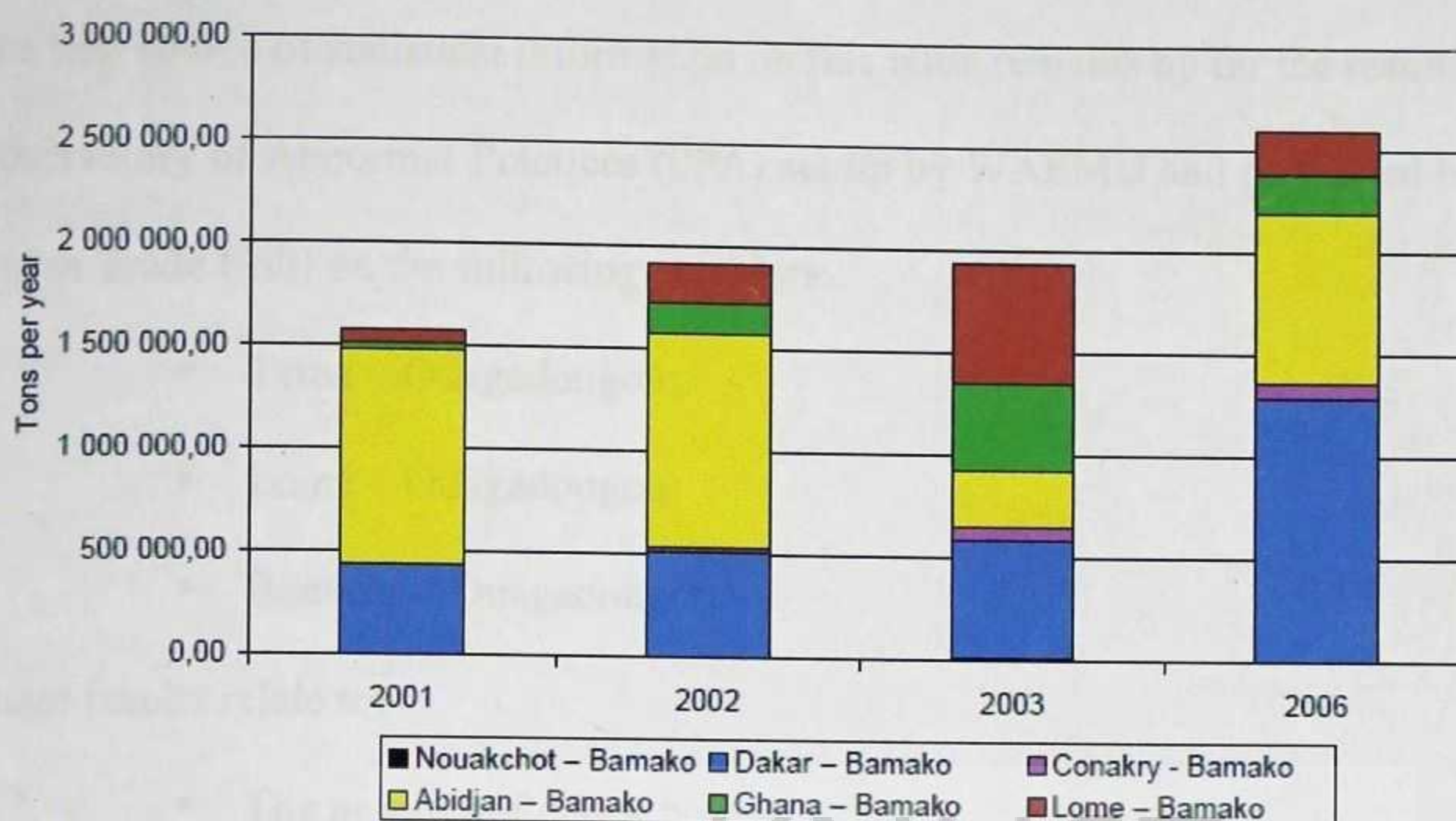


Figure 2.6 International Trade of Mali by Corridor

Source: world Bank (2007)

Abolition of unnecessary road blocks and illegal fees

Too many road blocks and illegal fees have constituted a hindrance to trade in the sub region. Their negative consequences are known to all Governments, yet despite the appeals by the Authorities of sub-regional institutions, despite several meetings, resolutions, recommendations and other instruments the phenomenon is growing from day to day. Thus in its Resolution C/RES/1/12/88 of 6 December 1988 – the ECOWAS Council of Ministers invited member States to reduce the number of road blocks by rationalizing the control services and simplifying road control procedures. These road blocks also compromise road safety, as they are not only traffic hold up but are also not easily obvious especially in the night.

The best source of statistical information on this issue remains by far the results from Observatory of Abnormal Practices (OPA) set up by WAEMU and published by WATH(West Africa Trade Hub) on the following corridors:

- Tema – Ouagadougou;
- Lomé – Ouagadougou;
- Bamako – Ouagadougou.

These results relate to:

- The number of check points;
- The times wasted at check points;
- The illegal collections demanded mainly by the law enforcement officials. The table below compares the number of check points for the period between 27 May 2007 and 26 October 2007 with the number earlier obtained for the period between 21 October 2006 and 26 May 2007. It reveals that:
- Mali recorded a noticeable drop in the number of check points per trip from 4.5 to 4 points per 100 km
- World Bank: “The Cost of Being Landlocked” available at:[4]

2.2 Review of relevant literature

Various Customs Administration in the sub-region have done some form of automation of their procedures in collecting duties and taxes for their various countries. There have been attempts to bring these systems together in order to harmonized these procedures in the sub region. The following shows the various systems in place that the Customs have been using.

2.2.1 The Existing System

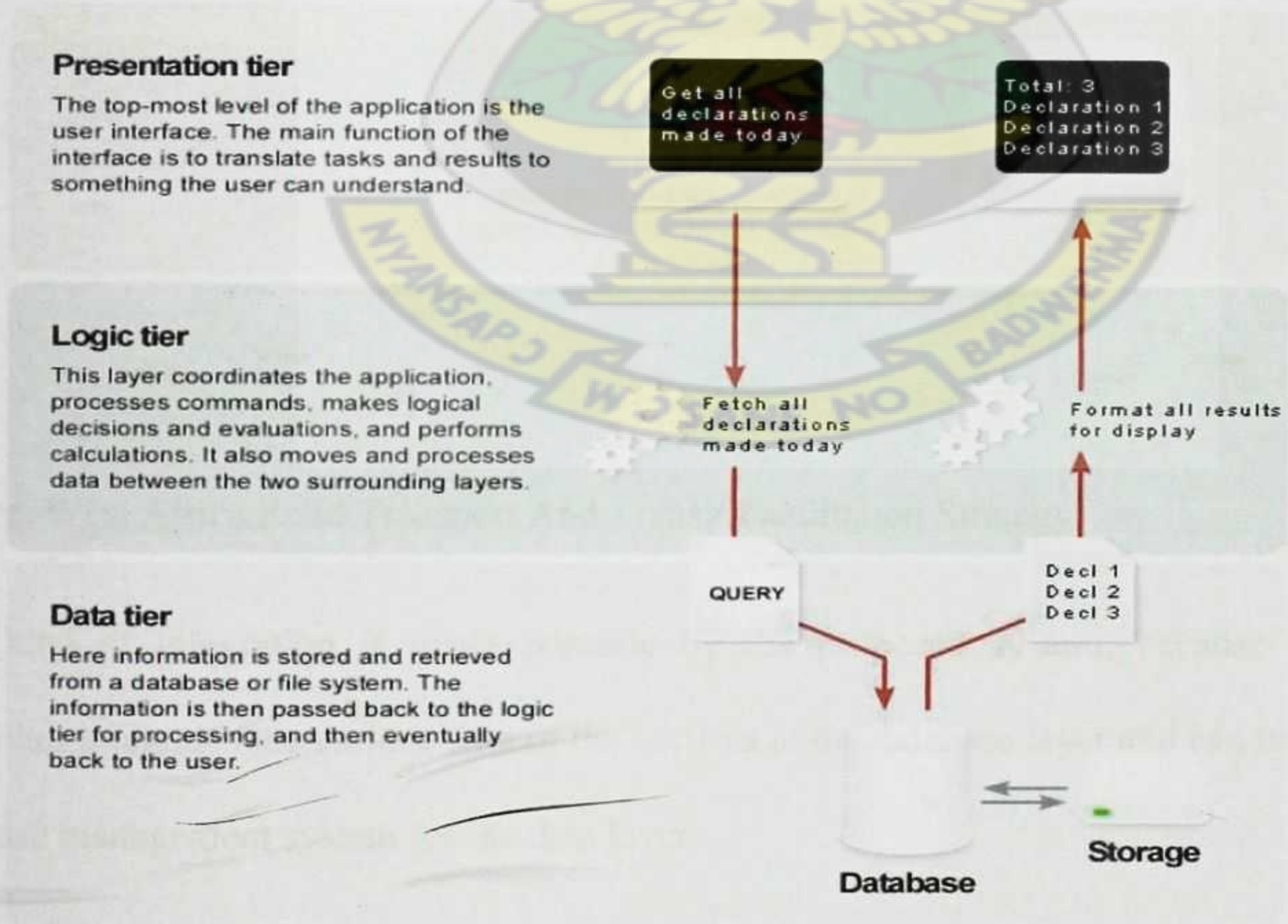
INTERCONNECTION OF CUSTOMS COMPUTERISED SYSTEMS

Architectural Model

There is a general consensus that the ECOWAS/UEMOA countries operate systems that are widely “incompatible”. The analysis by the proponents of this school of thought is that since the systems operate different architectures and are on different platforms at times, they cannot be integrated. Further, the widely divergent formats of standard customs declarations in use make it “impossible” for such kinds of integration.

The software system in use at the Customs Administration in the ECOWAS region roughly conform to the 3-tier architectural model regardless of whether they are implemented as distributed database systems or centralized database systems as below in Figure: 2.7: 3-tier software architecture

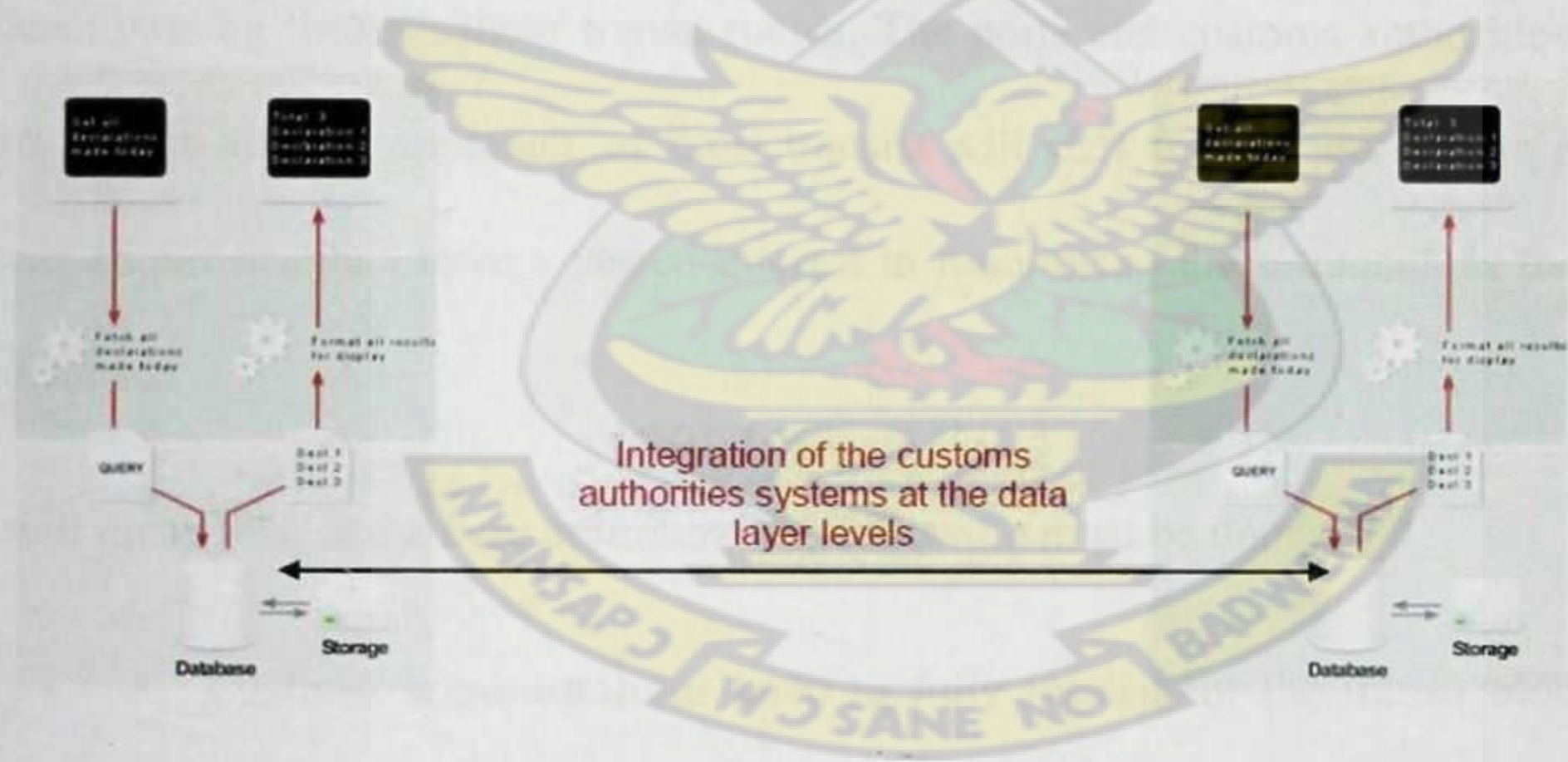
Figure 2.7: 3-Tier architecture of the existing system



Therefore, any attempt at integrating the different customs administration systems will have to take into consideration the suitability at integrating at the presentation layer, the logic layer or the database layer. The customs administration systems in the ECOWAS/UEMOA region fall roughly into six categories, each category having differing characteristics on each tier of its 3-tier software architecture. ASYCUDA 2.7, ASYCUDA++ and ASYCUDA World do not have the same presentation, logic and database implementations even though they are from the same family of software. Neither do the GCMS, the SYDAM and the GAINDE share much in common.

Therefore, it is quite obvious that it is an extremely hard endeavor to try and integrate these systems at the presentation or at the business logic tier. The best attempt at integration can only be made by integrating at the database layer.

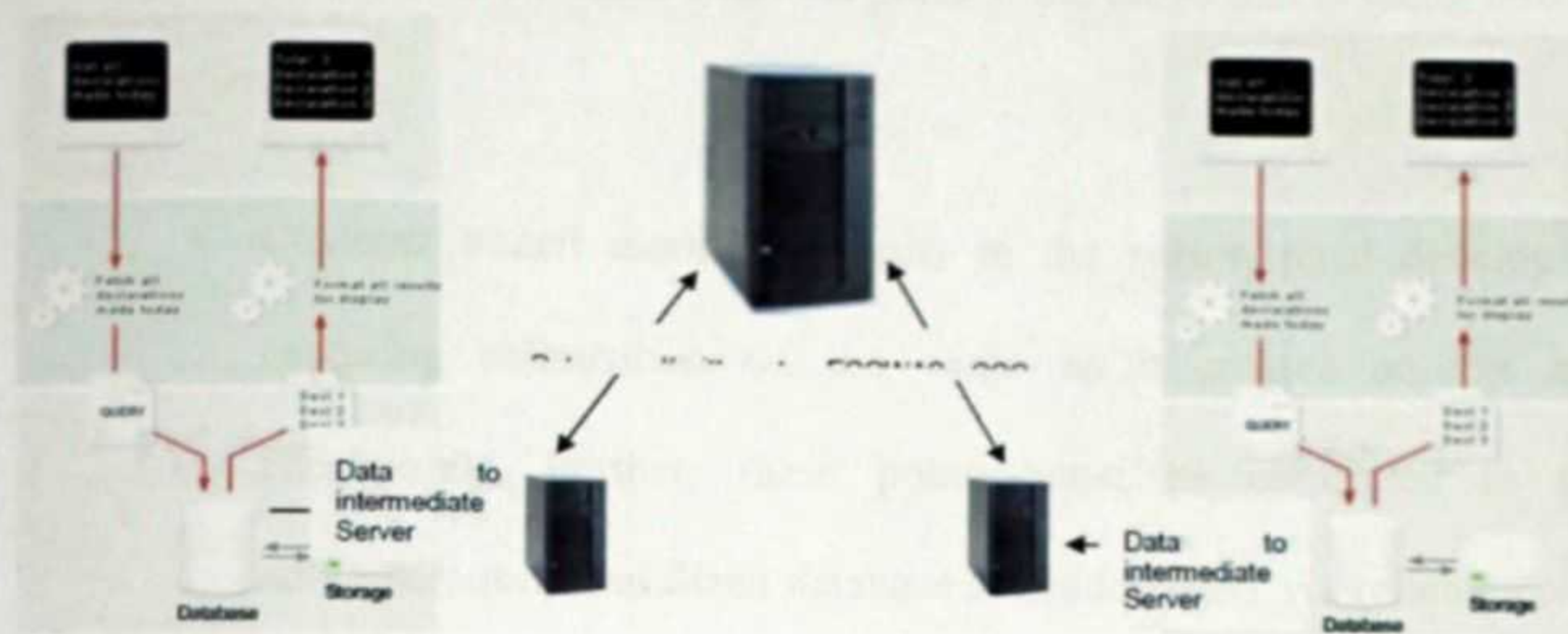
Figure: 2.8 Data layer levels



Source: West Africa Road Transport And Transit Facilitation Strategy

This kind of integration is made possible by the proposed system, because the Transit algorithm integrates the various data of the vertices at the database layer and can use relational database management system for the data layer.

Figure: 2.9 Interfacing for Ecowas



Source: West Africa Road Transport And Transit Facilitation Strategy.

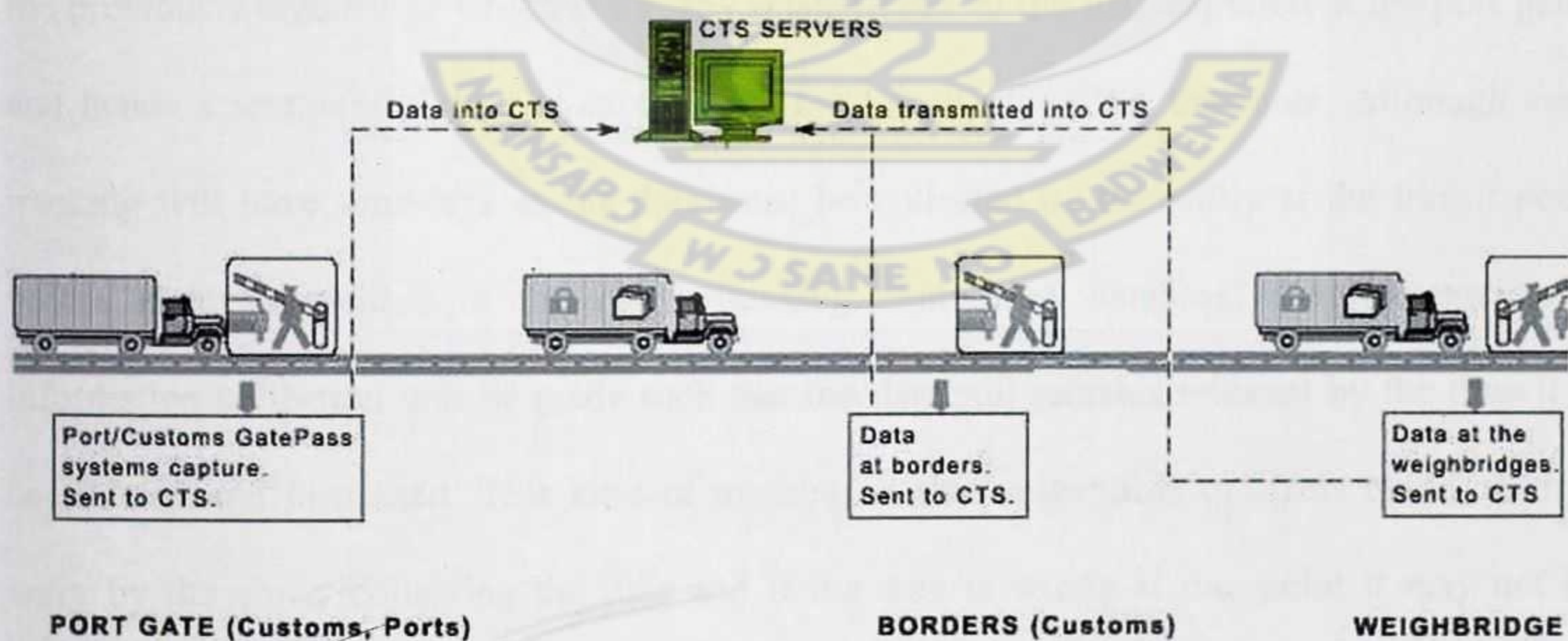
Since the transit routes are designated and documented, and there are checkpoints on these transit routes, then a regional cargo tracking system can be established based on data collected on transit passing through these transit routes. The ports and customs authorities are critical players in this kind of approach, as most transits will start from or end at the port and the customs administrations have a vested interest in monitoring these transits as they travel on the corridor.

This will mean that, at the bare minimum, the following must be done:

- Customs administrations have to fully implement the transit modules of the customs systems and to endeavor to implement the interconnectivity between administrations for information exchange.
- Ports and customs have to implement at least a manifest collection module to gather data from impending consignments to the ports. Ideally, such information should be ready before the vessel docks.

- Ports and customs must develop gate-pass systems to capture outgoing cargo that has been cleared from the ports or the cargo that is going into the ports for export.
- Customs transit monitoring units in the region must develop methods of capturing information on the cargo as it crosses borders and customs checkpoints. Further, these points must be connected to the customs administration centralized database at headquarters via reliable communication links.
- Any permanent weigh-bridges should be automated and modules to capture information on consignments passing through should be captured. All this information will then be aggregated on a prescribed frequency on the regional cargo tracking platform and will be linked to provide a seamless window into the movement of the cargo.

Figure 2.10: Cargo tracking data aggregation



Source: World Customs Organization (2007)

Typically, the linking of the data would happen as follows:

- ❖ As the goods leave the port, the information on the container numbers, seal numbers, customs declaration number, trailer registration number and the truck number are captured.
- ❖ Assuming the next point is a weighbridge, the data that could typically be captured at that point would include truck registration number, trailer number, axle loads.
- ❖ The data could be enhanced to record customs documents numbers, container numbers and seal numbers if the customs transit monitoring units are also based at that weighbridge. At this point, it's easy to use any of the captured data to link it with the previous data at the port.
- ❖ At the border points, customs would typically capture customs declaration numbers, truck and trailer registration numbers, container numbers. If there is a weighbridge, then axle loads and other details are captured.

By using the information captured, it would be possible to relate it back to the data captured at the previous weighbridge which is already related back to the data captured at the port gates and hence a seamless picture is captured as the transit moves on the route. Although such tracking will have time-lags as the data must be collected by the entity at the transit point before it is transmitted to the cargo tracking centralized database, the frequencies of information collection will be made such that the data still remains relevant by the time it is centralized and formatted. This kind of tracking is also susceptible to errors made on data entry by the entity collecting the data and if the data is wrong at one point it may not be possible to correlate it with the data at the other points, hence some portions of that transit might be rendered "invisible". Therefore data quality must be agreed upon and enforced by all participating stakeholders.

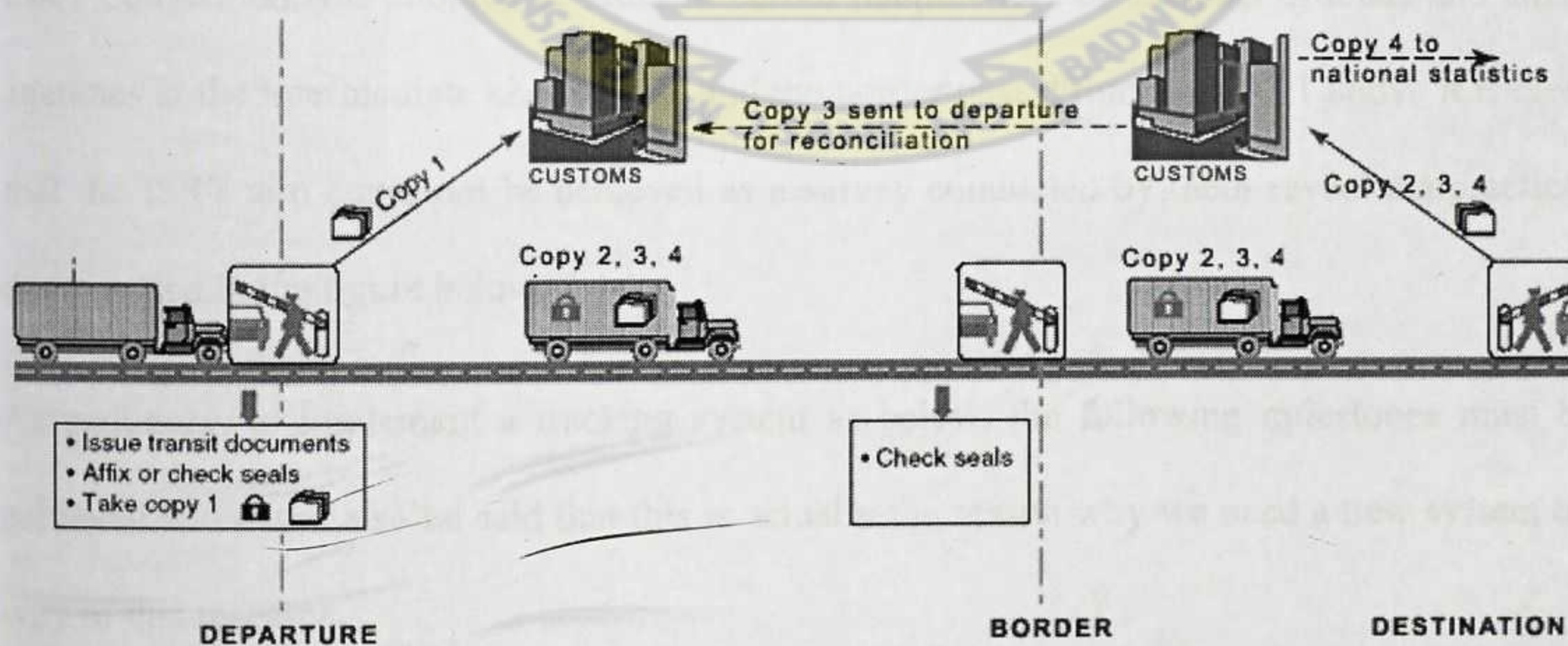
The plus side is that this method is not as expensive as the satellite, GPS and GPRS based methods. However, remote border stations that have no electricity and infrastructure to support computerized operations will necessarily need to be upgraded to at least have the basic internet connection and to meet the basic security requirements.

Also, since distances between the various data collection points are well-known, it is possible to set up alerts on the system to monitor any inordinately long time periods between the last sighting of the transit vehicle at the last checkpoint and the expected sighting at the next. This will serve as an early warning system against diversions.

The tracking of containerized cargo will basically be accomplished the same way as the tracking of the loose cargo under this kind of tracking system. In particular, if weighbridges are included as data collection points, it may be possible to notice any significant weight changes as the truck moves from one check point to the other and use that as an alert system.

The architectural model and the tracking data aggregation gives the ISRT transit operations model which is the model now operating in the sub region and it is shown below.

Figure 2.11: ISRT transit operation



Source: World Customs Organization (2007)

The ISRT works by using a set of leaflets within the ISRT declaration booklet which contain details of the importer and of the consignment as many "Approvals to Transit" leaflets as there are transit countries. Any customs documentation and any other documentation needed for the transit of the cargo is also carried with the cargo. In the transit operation, the first leaflet is retained at the office of departure while the second, third and fourth leaflets travel with the cargo to the destination. At destination (and if the transit was compliant) the third leaflet is sent back to the office of departure for matching and guarantee acquittal while the fourth leaflet is forwarded to the national office of statistics. While en-route, the transporter shows the "Approval to transit" at every border point they cross when demanded. The ISRT convention's intentions are to facilitate traffic by ensuring that transit is faster, customs revenue is protected by the issuing of transit guarantees, and also to address the faster acquittal of the guarantees on successful completion of transit in order to free up traders' capital investments tied up as guarantees. The ISRT transit operation therefore operates by focusing on: - Seals and secure means of transportation: these are attached after inspection at the office of departure so that the goods arrive at destination in the same form, quantity and status.

The issuance of a single ISRT declarations form was one of the three principles on which the ISRT convention was adopted in order to curtail malpractices of customs officials and allied agencies at the intermediate checkpoints and the border post. From figure 2.1 above it is clear that the ISRT aim could not be achieved as a survey conducted by them reveal malpractices documented in the figure below.

As summary, to implement a tracking system as below, the following milestones must be achieved and could also be said that this is actually the reason why we need a new system by way of this research.

Table 2.3 An Implementation of a transit tracking system

Activity	Objective	Responsible
Customs administrations to implement transit modules	Development of business processes, installation of appropriate modules for transit in ASYCUDA, GAINDE, GCMS and SYDAM.	Customs administrations
Ports and Customs to implement an automated manifest system	Development of business processes and software interfaces for capturing ship manifest information prior to ship's arrival.	Ports and Customs administrations
Ports and Customs to implement GatePass systems	Development of business processes and software modules to enable the capturing of all in/out operations at the ports.	Ports and Customs administrations
Customs to link border points to headquarters system via a WAN/LAN	Development of a Wide Area Network to ensure all major border points are networked into the customs systems at headquarters.	Customs administrations
Permanent Weighbridge infrastructure to be upgraded and the operations to be computerized, including linkage to the CTS	Ensure that the infrastructure at weighbridges is secure enough to ensure security if any ICT installations and to ensure that weighbridge data capture is computerized.	Ministries of Transport / Ministries of Public Works and/or Customs Transit Units
Establishment of steering committee for the CTS	To provide the general direction at policy level for the implementation of the CTS.	ECOWAS, Customs and Ports as key stakeholders
Focal points to be established within stakeholders as the technical committee for implementation	The technical committee will act as the technical validation point for the implementation of the CTS system and as the main actors within the	Customs, Ports, Clearing and Forwarding Agents, Transporters, Shipping Agents, relevant Ministries and other trade and transport facilitation stakeholders.

Source: World Customs Organization(2007)

Cont'd An Implementation of a transit tracking system

Training workshops for the pilot phase	Training sessions in each participating member country for all the stakeholders in order to enable the stakeholder community to donate data for tracking and to be able to use the system for tracking.	ECOWAS CCC, Customs Training Centers, Ports Training Centers
Preparation of bidding documents and contracting of a vendor for implementation - Implementation Phase	To enable the procurement of services from a vendor interested in running the CTS for profit.	Steering Committee, ECOWAS
Formation of an oversight body/company by the stakeholders	To monitor the execution of the contract by the selected vendor in as far as the CTS is concerned.	Stakeholders participating in the scheme
CTS Hosting and Sustainability - Implementation Phase	To move the implementation of the CTS for management by an independent outsourced entity and to work out the sustainability measures and the oversight duties of the stakeholders CTS oversight body.	Outsourced company with oversight by the oversight body/company of stakeholders

Source: World Customs Organization (2007)

2.3 Theoretical Framework

2.3.1 Maximum-flow minimum-cut Theorem

If f is a flow network $G = (V, E)$ with source s and sink t , then the following conditions are equivalent:

f is a minimum flow in G

The residual network G_f contains no augmenting paths

$|f| = c(S, T)$ for some cut (s, t) of G .

A classical method for finding maximum flows was described by Ford and Fulkerson [6]. This is a method than an algorithm in the sense that there are several implementations with

LIBRARY
KWAME NKRUMAH
UNIVERSITY OF SCIENCE & TECHNOLOGY
KUMASI

differing running times. The Ford and Fulkerson method depends on three ideas that go beyond the method and are important to many flow algorithms and problems. These are residual networks, augmenting paths, and cuts. An application of this method is finding a maximum matching in an undirected bipartite graph. Intuitively, given a flow network G and a flow f , the residual network G_f consists of edges with capacities that represent how we can change the flow on edges with capacities that represent G . An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. If the value is positive we place that edge into G_f with a residual capacity. The intuition of this is that a flow is increased by forward movement but is decreased by backward movement. By the same intuition given a flow network $G = (V, E)$ and a flow f , an augmenting path p is a simple path from s to t in the residual network G_f . From Ford and Fulkerson we may increase the flow on an edge of an augmenting path by the flow capacity of that edge. This amount can be maximum of the residual capacity. Ford and Fulkerson also show us that a maximum flow can actually be deduced from a cut of the flow network. The running time of Ford-Fulkerson depends on how we find the augmenting path. If we choose it poorly, the algorithm might not even terminate.

Ford-Fulkerson Method

Initialize flow f to 0

While there exist an augmenting path p in the residual network G_f

Augment flow f along p

Return f

Edmonds and Karp [7] improve the bound on Ford-Fulkerson by finding the augmenting path with a breadth-first search. That is to choose the augmenting path as a shortest path from source to the sink in the residual network, where each edge has unit distance (weight). Dinic independently using the breadth-first search [8] proved that this strategy yields a polynomial-

time algorithm. A related idea of using blocking flows was also first developed by Dinic [8]. All the above authors used the capacity constraint and did not relax the flow conservation in arriving at their various solutions. The capacity constraint simply says that the flow from one vertex to another must be non negative and must not exceed the given capacity. The flow conservation says that the total flow into a vertex other than the source or sink must equal the total flow out of that vertex. Informally we can say that flow in equals flow out.

Defining flows formally

Let $G = (V, E)$ be a flow network with capacity c . Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function $f: V \times V \rightarrow \mathbb{R}$ that satisfies the following properties: Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$

Flow conservation: For all $u \in V - \{s, t\}$, we require $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

Karzanov [9] first developed the idea of preflows. That is leaving the traditional way of solving the max-flow min-cut flow problem by relaxing the flow conservation and maintaining the capacity constraint.

Relaxation of flow conservation

Preflow is a function $f: V \times V \rightarrow \mathbb{R}$

Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$

$$e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geq 0$$

We say that the excess at a vertex is the amount by which the flow in exceeds the flow out.

We also say that a vertex $u \in V - \{s, t\}$ is over flowing if $e(u) > 0$.

The flow network G and flow f Augmenting Path method $(G, s, t) -$

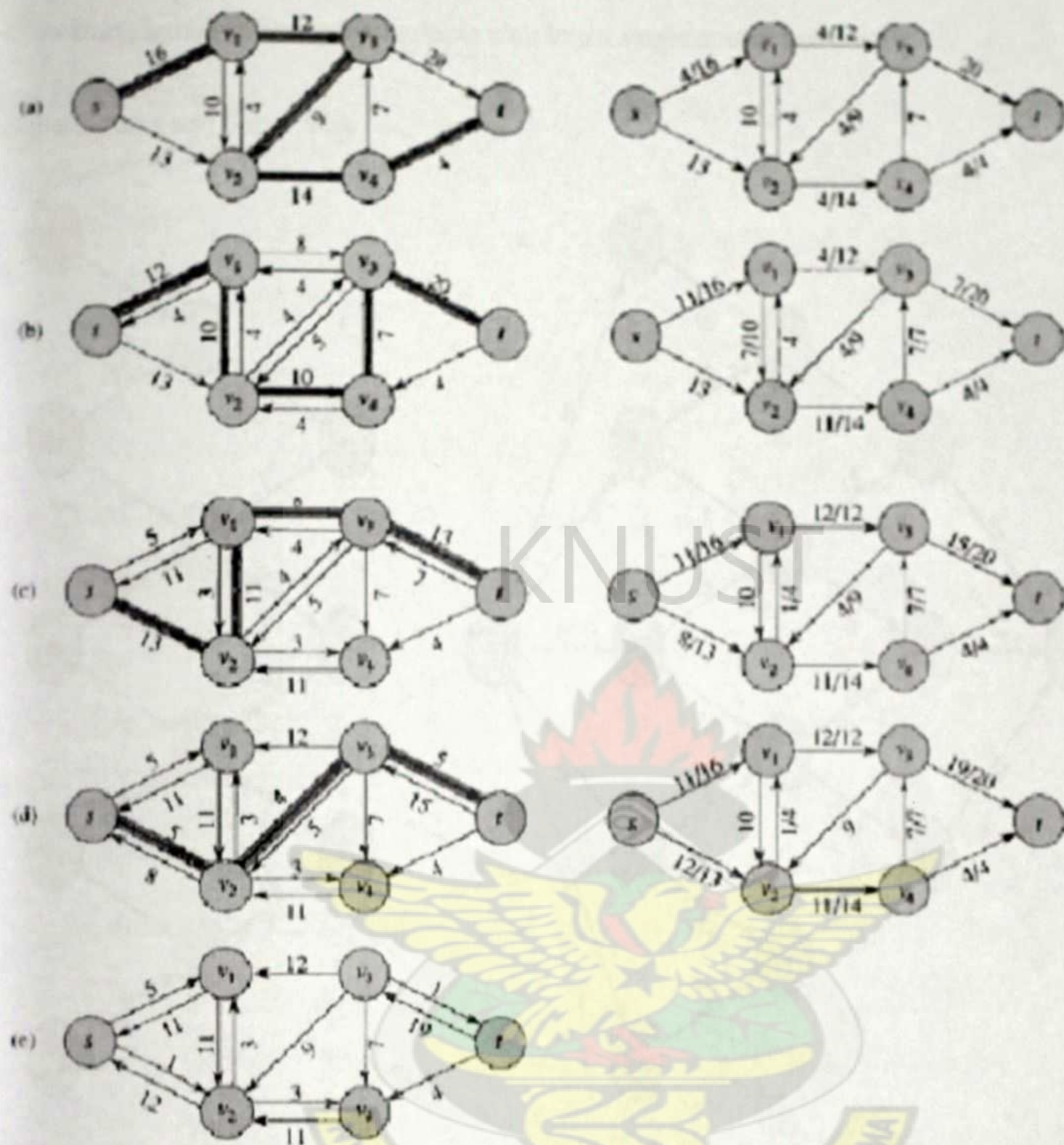


Figure 2.12 Flow network G and flow f

Source: Introduction to Algorithms (2008)

Converting a multiple source a multiple sink into a single source and sink

Super Source and Super Sink

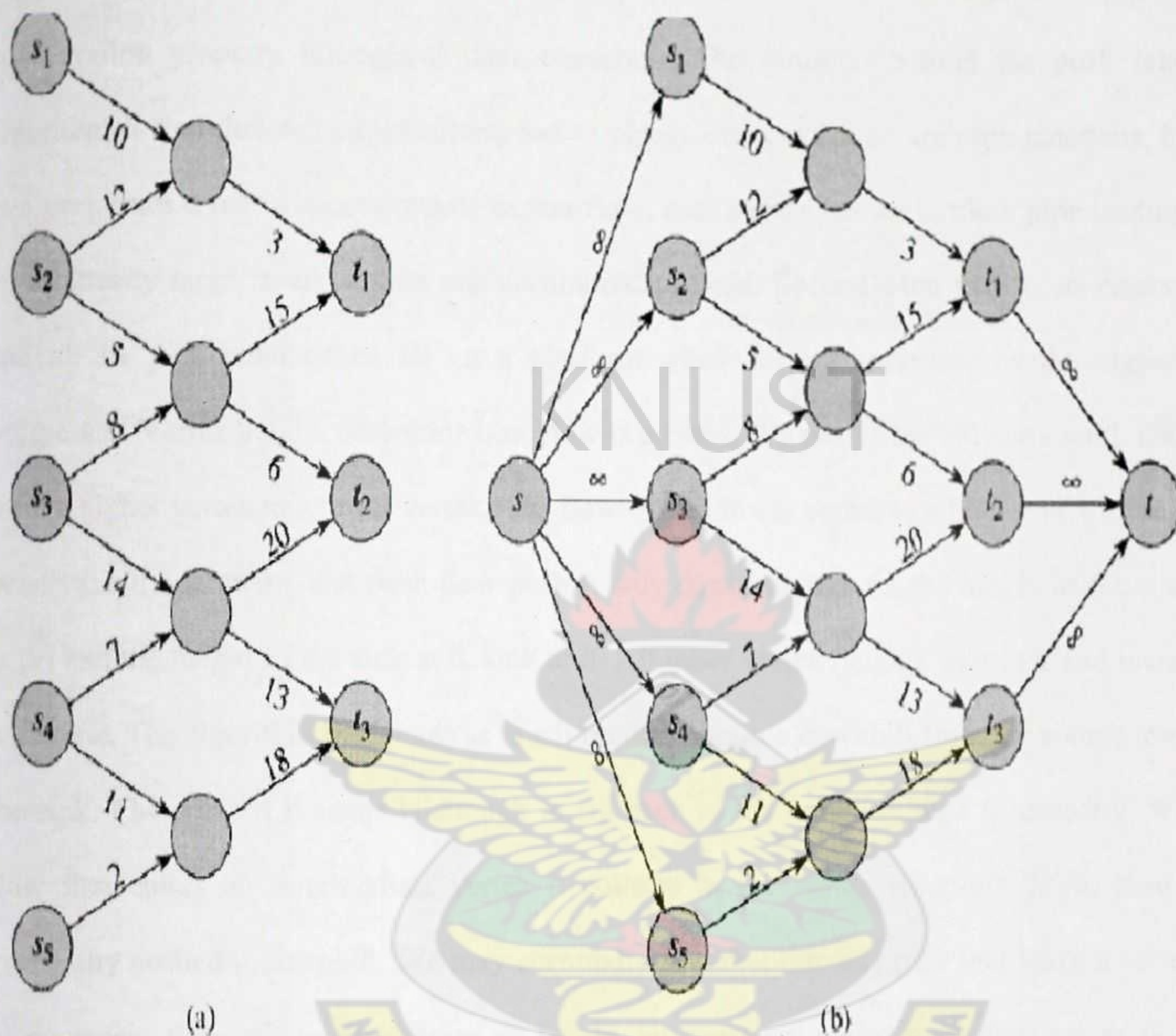


Figure 2.13: Super Source and Super Sink

Source: Introduction to Algorithms (2008)

2.3.2 The push relabel Method

The push relabel method is due to A.V Goldberg [10] and Goldberg and Tarjan [11]. To date many of the asymptotically fastest maximum flow algorithms are push relabel algorithms, and the fastest actual implementations of maximum flow algorithms are based on the push relabel method. Push relabel algorithm work in a more localized manner than the Ford-Fulkerson

method. Rather than examine the entire residual network to find an augmenting path, push relabel algorithms work on one vertex at a time, looking at all the vertex neighbors in the residual network. As mentioned above, push relabel methods do not maintain the flow conservation property throughout their execution. The intuition behind the push relabel algorithm is that directed edges correspond to pipes. Vertices which are pipe junctions, have two properties. First, to accommodate excess flow, each vertex has an outflow pipe leading to an arbitrarily large reservoir that can accommodate fluid. Second, each vertex, its reservoir, and all its pipe connections sit on a platform whose height increases as the algorithm progresses. Vertex height determines how flow is pushed. We push flow only downhill, that is, from a higher vertex to a lower vertex. The flow from a lower vertex to a higher vertex may be positive, but operations that push flow push it only downhill. We fix the height of the source at $|V|$ and the height of the sink at 0. All other vertex heights start at 0 and increase with time. The algorithm first sends as much flow as possible downhill from the source toward the sink. The amount it sends is enough to fill each of the outgoing pipes to capacity. When flow first enters an intermediate vertex, it collects in the vertex reservoir. From here we eventually push it downhill. We may eventually find that the only pipe that leaves a vertex u of its excess flow, we must increase its height, an operation called relabeling vertex u . We increase its height one unit more than the height of its neighbors to which it has an unsaturated pipe. After a vertex is relabeled, it has at least one outgoing pipe through which we can push more flow. Eventually all the flow that can possibly get through to the sink has arrived there. No more can arrive, because the pipes obey the capacity constraints, the amount of flow across any cut is still limited by the capacity of the cut. To make the preflow a legal flow, the algorithm then sends the excess collected in the reservoirs of the vertices back to the source by continuing to relabel vertices to above the fixed height $|V|$ of the source.

Several other researchers have developed push relabel maximum flow algorithms. Ahuja and Orlin [12] and Ahuja, Orlin, and Tarjan [13] gave algorithms that used scaling. Cheriyan and Maheshwari [14] propose pushing flow from the overflowing vertex of maximum height. Cheriyan and Hagerup [15] suggested randomly permuting the neighbor lists.

2.3.3 The Blocking flows method

All algorithms stated above use some notion of distance, for instance the push relabel algorithms use the analogous notion of height, with a length of 1 assigned implicitly to each edge. This new algorithm takes a different approach and assigns a length of 0 to high capacity edges and a length of 1 to low capacity edges. Informally, with respect to these lengths, shortest paths from the source to the sink tend to have high capacity, which means that fewer iterations need be performed. The algorithm of King, Rao, and Tarjan [16] is such an algorithm. Noga Alon [17], Stephen Phillips and Jeffrey Westbrook [18] These researchers developed clever ways that prevented randomly permuting the neighbor list, leading to a sequence of faster algorithms.

2.3.4 Heuristics

Global relabel and Gap relabel heuristics are very essential when implementing a push relabel algorithm. Goldberg and Cherkassky [19] and Goldberg [20] studies reveal that the push relabel algorithm has poor practical performance because relabel is a local operation, the method loses the global picture of the distances. A new maximum flow by Goldberg [21] uses the global relabel heuristic and updates the distance function by computing shortest path distances in the residual graph from all nodes to the sink. This can be done in linear time by a backward breadth first search which is computationally expensive compared to the push and relabel operations. Global relabeling are performed periodically, example after every n relabeling. This heuristic drastically improves the running times. Another useful relabeling

heuristic is gap relabeling discovered independently by Karzanov [22] and Derigs and Meier [23], and based on the following observations. Let g be an integer and $0 < g < n$. Suppose at certain stage of the algorithm there are no nodes with distance label g but there are nodes v with $g < d(v) < n$. Then the sink is not reachable from any of these nodes. Therefore the labels of such nodes may be increased to n . Remember such nodes will never be active. If for every i we maintain linked list of nodes with distance label i the overhead of detecting the gap is very small. Most work done by the gap relabeling is useful. It involves processing the nodes determine to be disconnected from the sink.

2.3.5 Parallel algorithms for the maximum flow problem.

Goldberg's method of parallelizing the push relabel algorithm is almost the same as that of Karzanov [22] and latter by others like R.V Cherkassy [22], Z.Galil [25], Y. Shiloach and Viskin [26], R.E Tarjan [16]. R. Anderson and J. Setubal [27] On the parallel Implementation of Goldberg's "maximum flow algorithm" The algorithm begins with a blocking preflow and moves flow excess through the network while maintaining a blocking preflow, until eventually this flow movement produces a blocking flow. The algorithm maintains a partition of the vertices into two states: blocked and unblocked. We call an arc (v, w) admissible if it is residual and w is unblocked. The algorithm blocks a vertex w when it discovers that none of the arcs leaving v is admissible; once v is blocked, every path from v to t contains a saturated arc.

Excess on blocked vertices is returned from whence it came, by decreasing the flow on appropriate incoming arcs. To keep track of the detailed flow movements, the algorithm maintains a partition of the flow excess into atoms. Consider a time during an execution of the algorithm. An atom is a maximal quantity of excess that has moved in exactly the same way so far. An atom a at a vertex v consists of an amount of excess denoted by $size(a)$; the vertex v is denoted by $position(a)$. An atom located at a vertex other than s or t is called active.

Associated with an atom a at a vertex v is a path of arcs in E^+ from s to z .⁷ that the atom followed in arriving at v . This path is denoted by $\text{trace}(a)$. Also associated with v is a simple path from s to v denoted by $\text{path}(a)$, of arcs in E through which the atom moved forward but not backward in the course of reaching w from s . The relationship between $\text{trace}(a)$ and $\text{path}(a)$ is that $\text{path}(a)$ contains each arc (w, a) such that (w, a) but not (w, v) is on $\text{trace}(a)$.

2.3.6 Summary of Worst case bounds of algorithms in the theoretical framework

Ahuja et al, [13] limited their study to the best previous maximum flow algorithms and some recent algorithms that are likely to be efficient in practice. Their study encompasses ten maximum flow algorithms whose discoverers and worst-case time bounds are given in Table 2.17. In the table, n denotes the number of nodes, m denotes the number of arcs, and U the largest arc capacity in the network.

1. Dynamic Programming

Although this paper uses the parallel computational processing as its model some aspects of dynamic programming is used. That is recursion and memoization. R. Bellman[30] began the systematic study of dynamic programming in 1955. The word "programming," both here and in linear programming, refers to using a tabular solution method. Although optimization techniques incorporating elements of dynamic programming were known earlier, Bellman provided the area with a solid mathematical basis [30]. Galil and Park [31] classify dynamic-programming algorithms according to the size of the table and the number of other table entries each entry depends on. They call a dynamic-programming algorithm tD/eD if its table size is $O(n^t)$ and each entry depends on $O(n^e)$ other entries

Greedy Algorithms

Greedy algorithms for scheduling will be used at the latter part of this paper. Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each

step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Much more material on greedy algorithms and matroids can be found in Lawler [32] and Papadimitriou and Steiglitz [33]. The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [34], though the theory of matroids dates back to a 1935 article by Whitney [35]. The proof of the correctness of the greedy algorithm for the activity-selection problem is based on that of Gavril [36]. The task-scheduling problem is studied in Lawler [32]; Horowitz, Sahni, and Rajasekaran [37]; and Brassard and Bratley [38]

2.3.7 Multithreading

The data-parallel model according to J.S. Bulldog [39] and Stone [38] is another popular algorithmic programming model, which features operations on vectors and matrices as primitives. Graham [41] and Brent [42] showed that there exist schedulers achieving the bound of $T(n) = \Theta(\varphi^n)$ where $\varphi = (1 + \sqrt{5})/2$. Eager, Zahorjan, and Lazowska [43] showed that any greedy scheduler achieves this bound and proposed the methodology of using work and span (although not by those names) to analyze parallel algorithms. Blleloch [44] developed an algorithmic programming model based on work and span (which he called the “depth” of the computation) for data-parallel programming. Blumofe and Leiserson [45] gave a distributed scheduling algorithm for dynamic multithreading based on randomized “work-stealing” and showed that it achieves the bound $E[T_P] \leq T_1/P + O(T_\infty)$. Arora, Blumofe, and Plaxton [46] and Blleloch, Gibbons, and Matias [51] also provided provably good algorithms for scheduling dynamic multithreaded computations. The silk [47,48] project at MIT and the silk++ [47] extension to C++ distributed by silk Arts Inc. have an influence on the computation dag pseudocode in this paper.

The coding of a parallel program according to Thomas Rauber et al, [50] for a given algorithm is strongly influenced by the parallel computing system to be used. The term computing system comprises all hardware and software components which are provided to the programmer and which form the programmer's view of the machine. The software aspects include the specific operating system, the programming language and the compiler, or the runtime libraries. The same parallel hardware can result in different views for the programmer, i.e., in different parallel computing systems when used with different software installations. A very efficient coding can usually be achieved when the specific hardware and software installation is taken into account. But in contrast to sequential programming there are many more details and diversities in parallel programming and a machine-dependent programming can result in a large variety of different programs for the same algorithm. In order to study more general principles in parallel programming, parallel computing systems are considered in a more abstract way with respect to some properties, like the organization of memory as shared or private. A systematic way to do this is to consider models which step back from details of single systems and provide an abstract view for the design and analysis of parallel programs.

Models for parallel systems

In the following, the types of models used for parallel processing according to T. Heywood et al [51] are presented. Models for parallel processing can differ in their level of abstraction. The four basic types are machine models, architectural models, computational models, and programming models. The machine model is at the lowest level of abstraction and consists of a description of hardware and operating system, e.g., the registers or the input and output buffers. Assembly languages are based on this level of models. Architectural models are at the next level of abstraction. Properties described at this level include the interconnection network of parallel platforms, memory organization, synchronous or asynchronous processing, and

execution mode of single instructions by SIMD or MIMD. The computational model (or model of computation) is at the next higher level of abstraction and offers an abstract or more formal model of a corresponding architectural model. It provides cost functions reflecting the time needed for the execution of an algorithm on the resources of a computer given by an architectural model. Thus, a computational model provides an analytical method for designing and evaluating algorithms. The complexity of an algorithm should reflect the performance on a real computer. For sequential computing, the RAM (random access machine) model is a computational model for the von Neumann architectural model. The RAM model describes a sequential computer by a memory and one processor accessing the memory. The memory consists of an unbounded number of memory locations each of which can contain an arbitrary value. The processor executes a sequential algorithm consisting of a sequence of instructions step by step. Each instruction comprises the load of data from memory into registers, the execution of an arithmetic or logical operation, and the storing of the result into memory. The RAM model is suitable for theoretical performance prediction although real computers have a much more diverse and complex architecture. A computational model for parallel processing is the PRAM (parallel random access machine) model, which is a generalization of the RAM model. The data parallel model according to Thomas Rauber et al [50] is an aspect of the computational model.

The programming model is at the next higher level of abstraction, and describes a parallel computing system in terms of the semantics of the programming language or programming environment. A parallel programming model specifies the programmer's view on parallel computer by defining how the programmer can code an algorithm. This view is influenced by the architectural design and the language, compiler, or the runtime libraries and, thus, there exist many different parallel programming models even for the same architecture. There are several criteria by which the parallel programming models can differ:

- the level of parallelism which is exploited in the parallel execution (instruction level, statement level, procedural level, or parallel loops)
- the implicit or user-defined explicit specification of parallelism
- the way how parallel program parts are specified
- the execution mode of parallel units (SIMD or SPMD, synchronous or asynchronous)
- the modes and pattern of communication among computing units for the exchange of information (explicit communication or shared variables)
- synchronization mechanisms to organize computation and communication between parallel units.

Parallelization of Programs

The parallelization of a given algorithm or program is typically performed on the basis of the programming model used. Independent of the specific programming model, typical steps can be identified to perform the parallelization. We assume that the computations to be parallelized are given in the form of a sequential program or algorithm. To transform the sequential computations into a parallel program, their control and data dependencies have to be taken into consideration to ensure that the parallel program produces the same results as the sequential program for all possible input values. The main goal is usually to reduce the program execution time as much as possible by using multiple processors or cores. The transformation into a parallel program is also referred to as parallelization. To perform this transformation in a systematic way, it can be partitioned into several steps:

Decomposition of the computations: The computations of the sequential algorithm are decomposed into tasks, and dependencies between the tasks are determined. The tasks are the

smallest units of parallelism. Depending on the target system, they can be identified at different execution levels: instruction level, data parallelism, or functional parallelism. In principle, a task is a sequence of computations executed by a single processor or core. Depending on the memory model, a task may involve accesses to the shared address space or may execute message-passing operations. Depending on the specific application, the decomposition into tasks may be done in an initialization phase at program start (static decomposition), but tasks can also be created dynamically during program execution. In this case, the number of tasks available for execution can vary significantly during the execution of a program. At any point in program execution, the number of executable tasks is an upper bound on the available degree of parallelism and, thus, the number of cores that can be usefully employed. The goal of task decomposition is therefore to generate enough tasks to keep all cores busy at all times during program execution. But on the other hand, the tasks should contain enough computations such that the task execution time is large compared to the scheduling and mapping time required to bring the task to execution. The computation time of a task is also referred to as granularity: Tasks with many computations have a coarse-grained granularity, tasks with only a few computations are fine-grained. If task granularity is too fine-grained, the scheduling and mapping overhead is large and constitutes a significant amount of the total execution time. Thus, the decomposition step must find a good compromise between the number of tasks and their granularity.

Assignment of tasks to processes or threads: A process or a thread represents a flow of control executed by a physical processor or core. A process or thread can execute different tasks one after another. The number of processes or threads does not necessarily need to be the same as the number of physical processors or cores, but often the same number is used. The main goal of the assignment step is to assign the tasks such that a good load balancing results, or thread should have about the same number of computations to perform. But the number of memory

accesses (for shared address space) or communication operations for data exchange (for distributed address space) should also be taken into consideration. For example, when using a shared address space, it is useful to assign two tasks which work on the same data set to the same thread, since this leads to a good cache usage. The assignment of tasks to processes or threads is also called scheduling. For a static decomposition, the assignment can be done in the initialization phase at program start (static scheduling). But scheduling can also be done during program execution (dynamic scheduling).

Mapping of processes or threads to physical processes or cores

In the simplest case, each process or thread is mapped to a separate processor or core, also called execution unit in the following. If less cores than threads are available, multiple threads must be mapped to a single core. This mapping can be done by the operating system, but it could also be supported by program statements. The main goal of the mapping step is to get an equal utilization of the processors or cores while keeping communication between the processors as small as possible.

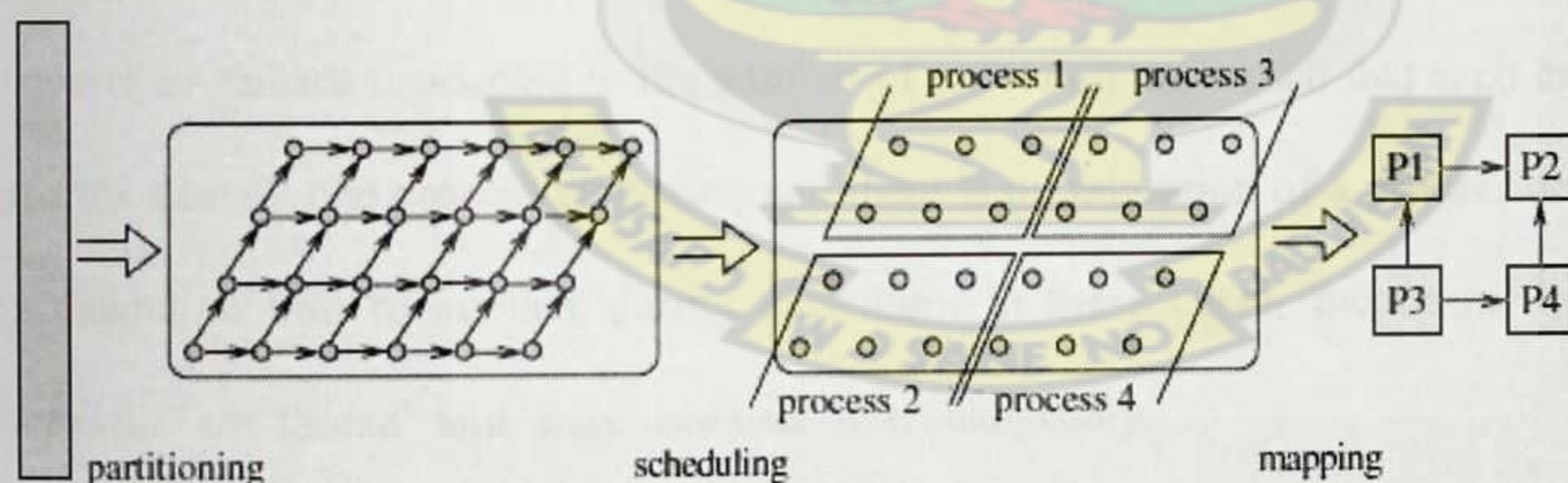


Figure 2.14 : The parallelization steps

Source: Verlag Berlin Heidelberg 2010

In general, a scheduling algorithm is a method to determine an efficient execution order for a set of tasks of a given duration on a given set of execution units. Typically, the number of tasks is much larger than the number of execution units. There may be dependencies between

the tasks, leading to precedence constraints. Since the number of execution units is fixed, there are also capacity constraints. Both types of constraints restrict the schedules that can be used. Usually, the scheduling algorithm considers the situation that each task is executed sequentially by one processor or core (single-processor tasks). But in some models, a more general case is also considered which assumes that several execution units can be employed for a single task (parallel tasks), thus leading to a smaller task execution time. The overall goal of a scheduling algorithm is to find a schedule for the tasks which defines for each task a starting time and an execution unit such that the precedence and capacity constraints are fulfilled and such that a given objective function is optimized. In this project no scheduling algorithm is used as it is out of scope.

Often, Parallel Programming Models the overall completion time (also called make span) should be minimized. This is the time elapsed between the start of the first task and the completion of the last task of the program. For realistic situations, the problem of finding an optimal schedule is NP-complete or NP-hard Ananth Grama [63]. A good overview of scheduling algorithms is given by C.E Leiserson and B.M Maggs [24]. Often, the number of processes or threads is adapted to the number of execution units such that each execution unit performs exactly one process or thread, and there is no migration of a process or thread from one execution unit to another during execution. In these cases, the terms "process" and "processor" or "thread" and "core" are used interchangeably.

The demand of ECOWAS for a single declaration document at the country of origin, that is at the beginning of every transit and the futile attempt by the existing system in solving this problem has led to this paper. In this paper the procedures responsible for the transit of goods through the sub region is modeled on a parallel computational direct acyclic graph which process these procedures at the vertices. To ensure a maximum flow of this process a push

relabel method is employed and a computation acyclic graph to process the various procedures at the vertices.

Shared memory model

The main model of parallel computation is the shared-memory model. In this model all processing units have direct access to a common memory. Such an access is assumed to cost constant time. In addition the processing units have a local memory in which they can store local data. When considering computers with a shared memory, it is important to specify which type of access to the memory are allowed explained by P. Brocker [53]

The EREW, Exclusive Read Exclusive Write, model is the most restrictive. Here it is assumed that in a step each position in the main memory can be accessed by at most one processing unit.

The CREW, Concurrent Read Exclusive Write, model is more convenient. Here it is assumed that in a step arbitrarily many PUs can read a position while at most one PU can write a position.

The CRCW, Concurrent Read Concurrent Write, model is the most powerful. Here it is assumed that in a step a position can be read and written by arbitrarily many processing units..

last case there are several possible ways of handling write conflicts, leading to a further subdivision of the model:

On a Common CRCW machine, concurrent writes are only allowed if all written values are equal. We will see how on such a machine the maximum of n numbers can be computed with $O(n)$ work in $O(\log n)$ time.

On an Arbitrary CRCW machine it is allowed to write different values and one of them will be written. The model does not specify which element is written, so the algorithms designer must be prepared for the worst possible choice.

On a Random CRCW machine it is allowed to write different values and a randomly selected one of them will be written. In this model the algorithms designer may exploit that some kind of average value will succeed.

On a Maximum CRCW machine it is allowed to write different values and the largest of them will succeed.

Depending on the underlying hardware, any of these models may be realistic. They are not far apart anyway: the most powerful of these models, a step of a Maximum CRCW machine with P processing units can be simulated on an EREW machine in $O(\log P)$ steps. The main parallel computer model considered in this text is the PRAM model. A PRAM, Parallel Random Access Machine, is a synchronous parallel computer with a shared memory. When specifying the performance of a PRAM algorithm we will mostly specify the memory access model, so we may say "On a CREW PRAM ...".

It would be most correct to formulate a PRAM algorithm including load and store operations specifying which data have to be copied from the global memory to the local memory and vice versa. However, the existence of a local memory is mostly ignored, specifying algorithms as if all the processing units operate directly on the shared memory itself. This is similar to the practice of writing sequential algorithms without mentioning the memory management. The main reason why the PRAM model is so popular is that it allows us to concentrate on the work to perform. The shared memory model entirely neglects the aspects of data exchange. Thereby it allows us to concentrate on the fundamental aspects of parallelizability.

Work-Time Framework

The PRAM model is simple, but the work-time framework allows to concentrate even more on the underlying concepts. The idea is to not specify the number of involved processing units an algorithm is composed of steps, and in each step there are operations to be executed in parallel. Such an instruction may be of the type "for all i , $0 \leq i < n$, $a[i] = b[i] + c[i]$ ". for example.

An algorithm is evaluated with two cost parameters: the time, that is, the number of parallel steps; and the work, the sum over all steps of the number of performed operations. Specifying the performance of an algorithm in this way leaves the allocation of the work to the processing units unsolved. In principle it should be possible to perform a step in which w operations are executed on a PRAM with P processing units in roundup (w / P) time, but only if each processing units knows which operations to execute.

If the work-allocation problem can be solved, then a WT algorithm gives a PRAM algorithm: if $W(n)$ and $T(n)$ are the work and time for a problem of size n , respectively, then

PRAM algorithm

$$T(P, n) = \sum_{\text{step } i} \text{round_up}(w_i / P)$$

$$\leq T(n) + \sum_{\text{step } i} w_i / P$$

$$\leq T(n) + W(n) / P,$$

$$C(P, n) = P * T(P, n)$$

$$\leq P * T(n) + W(n).$$

These relations immediately suggest a close to optimal choice of P for a given value of n : take $P = P(n) = W(n) / T(n)$: for that value $T(P, n) \leq 2 * T(n)$ and $C(P, n) \leq 2 * W(n)$. So, for this

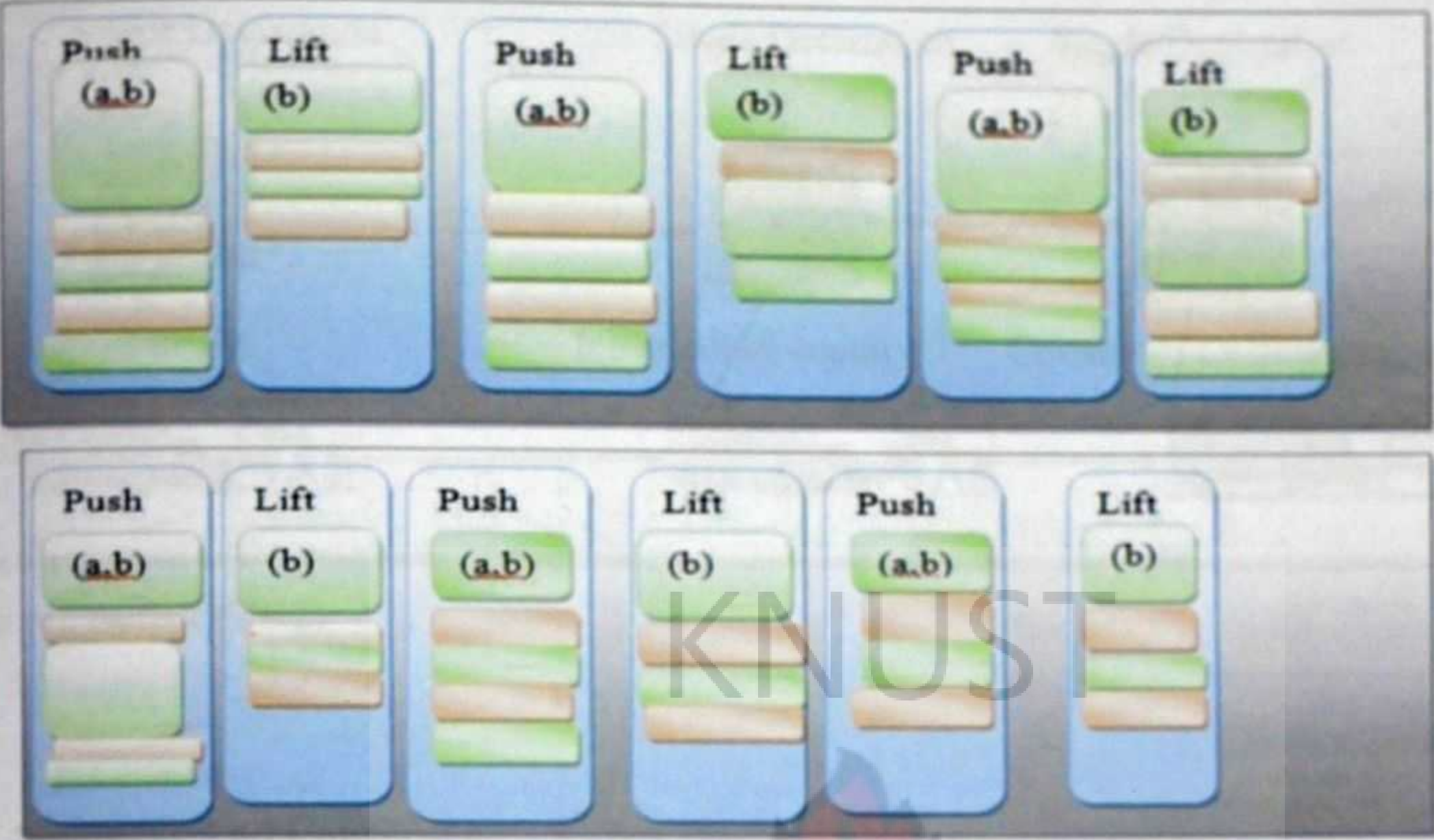
choice of P , the completion time is at most twice the minimum value, and the cost is at most twice as large as the work. This is a good compromise: taking P larger, $C(P, n)$ increases rapidly, without giving a substantial reduction of $T(P, n)$. Taking P smaller gives substantially larger $T(P, n)$, without giving a substantial reduction of $C(P, n)$.

If there is a parallel algorithm in the work-time framework solving a problem of size n with work $W(n)$ and time $T(n)$, then, provided the work allocation can be solved, there is a PRAM algorithm with time $O(T(n))$ and cost $O(W(n))$. Thomas Rauber et al [50]

A directed acyclic graph representing the execution of Trans (4) procedures. Each circle represent one strand, with circles representing either base cases or the part of the procedure (instance) up to the spawn of Trans (n), shaded circles representing the part of the procedure that calls Trans($n-1$) and suspends until the spawn of Trans (n) returns, and white circles representing the part of the procedure after it was suspended and where it saves the vertices in the list. Each group of strands belonging to the same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and heavily shaded for called procedures. Spawn edges and called edges point downward, continuation edges point to the right, and return edges point upward.

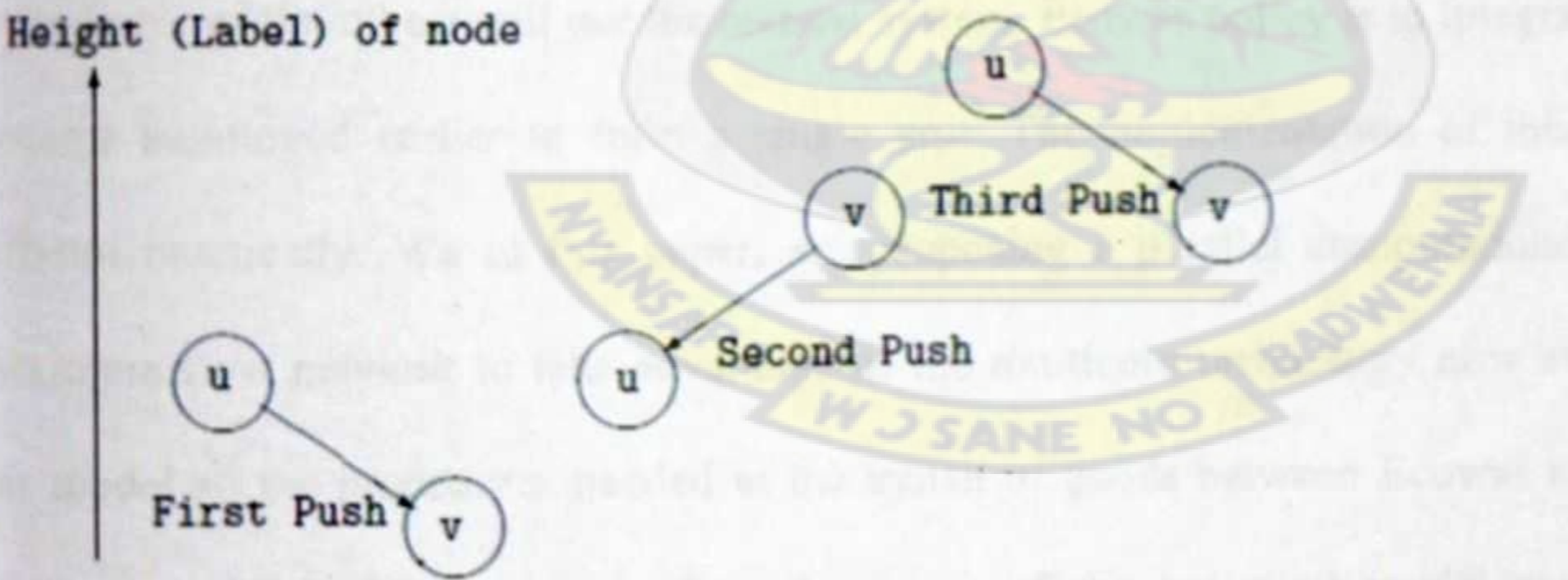
After execution of procedures at the vertices which by the push relabel method was only a temporal storage area would now send the executions to its destination and could then be printed, so that at the beginning of every transit all the necessary information would be available on a single document.

Figure 2.15: Six possibilities of interleaving of push and lift operations



Source: Bo Hong Drexel University, Philadelphia, PA 19104

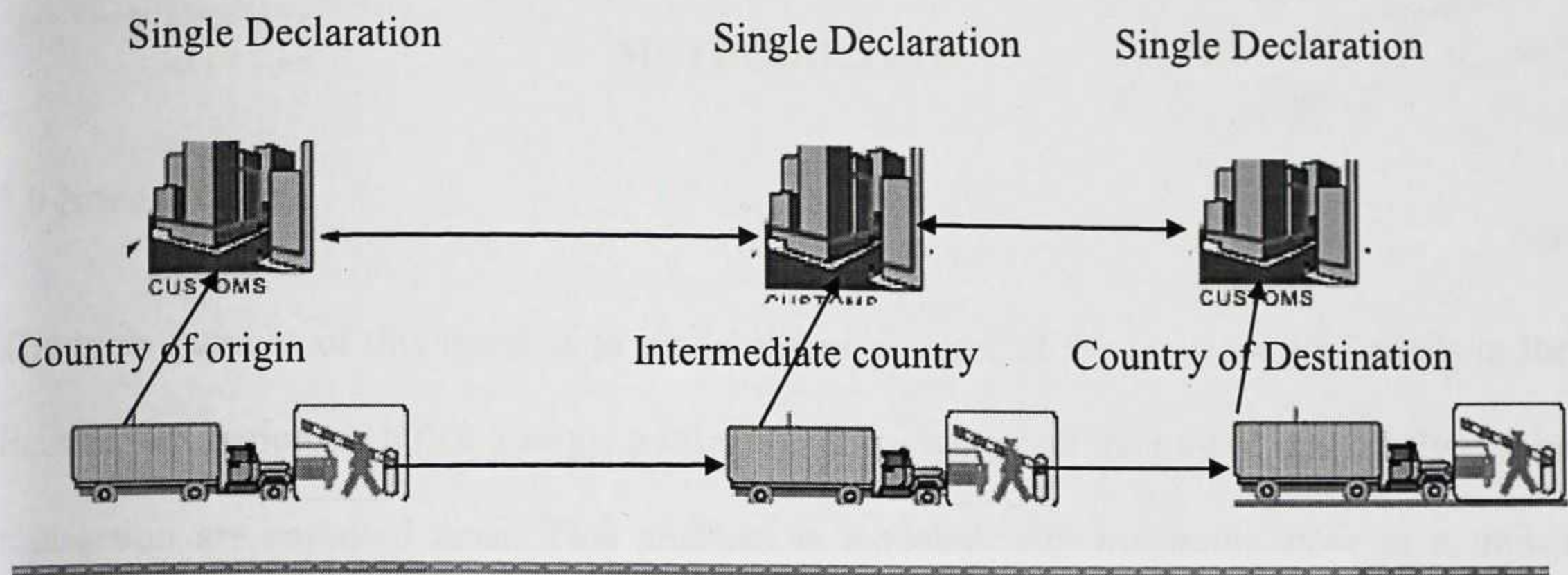
Figure 2.16: Sequence of saturation pushes on (u,v)



Source: Prof. Dorit Hochbaum, IEOR 266,Fall (2003)

Figures 2.16 and 2.17 shows the action taken after the computations at the vertices from the source to the sink , depicting transactions from the country of origin to destination

Figure 2..17 Single Declaration Document Transit Model



Source : Author (2012)

2.4 Summary And Conclusion

The Ecowas in their quest to integrate their procedures in the transit of goods, in order to have a single declaration which will among other things ensure the free movement of goods, have initiated a lot of policies. Many of the States in Ecowas have partially automated their procedures, whilst others still use the manual system. Ecowas policy is to integrate the various systems mentioned earlier to form a single unit. The implementation of this idea is very difficult practically. We in this paper, are proposing a parallel computational model of a maximum flow network to take advantage of the multicore technology now available. With this model all the procedures needed in the transit of goods between Ecowas states could be captured on a single document. The advantages of the proposed model are those which Ecowas want to achieve, i.e the abolition of the quota system, improvement in the conditions of the road transport market (freight centre), harmonization of traffic rules and traffic code, abolition of unnecessary road blocks and illegal fees and rallying towards international standards enumerated earlier.

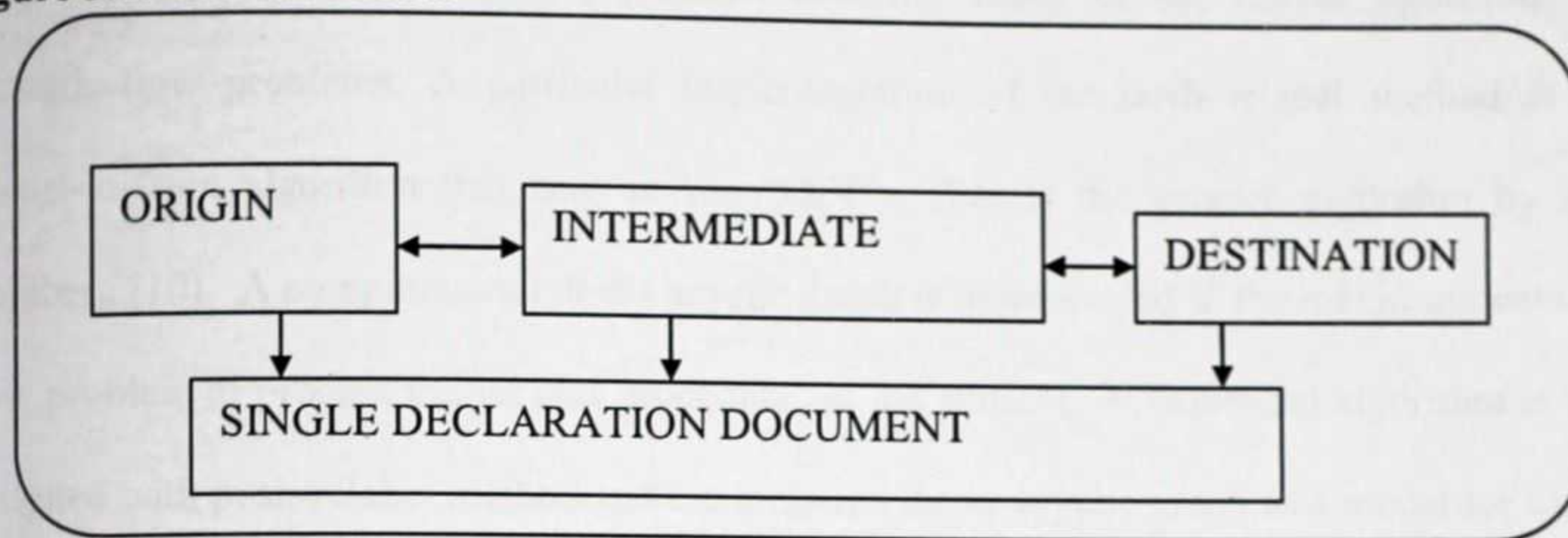
CHAPTER THREE

METHODOLOGY

3.0 Introduction

The main purpose of this thesis is to model the procedures of the movement of goods in the Ecowas sub-region such that a single transaction document of all the procedures involve in the transaction are captured once. This problem is modeled with maximum flow as a transit algorithm is developed, which is a combination of push relabel method and a Computational Dag (Direct Acyclic Graph). In Customs jargon movement of goods is likened to the processing of the customs procedures that is prior to the spatial displacement of the goods. Goods are therefore deemed moved when the procedures responsible for that movement is completed though the goods would have been at the very position first placed. The intuition behind the transit algorithm is as follows, directed graph of the network have a path from one vertex to another and transit of goods should have a predefined route. The network is directed acyclic graph as there should be no loops in the movement of goods. The vertices in the network represent states in the transit corridor. Edges connecting vertices in the network represent communication links between states along the transit corridor. The super source of the flow network is the logical location of the start vertex of all goods coming from various sources to converge on the one source called country of origin which is the starting vertex. The super sink of the flow network located at the end of the flow is the logical location of all the goods converging on the country of destination of the goods. The computation Dag at the vertices represents the processing of procedures of states in the transit corridor. The resultant outcome of the list could be represented as a single declaration document of the transit transaction. The maximum flow problem therefore facilitates an efficient services rendered in the transit transaction.

Figure 3.1: Transit Model



Source: Author (2012)

Modeling problems with maximum flow, is just as we can model a road as a directed graph in order to find the shortest path from one point to another. We can also interpret a directed graph as a “flow network” and use it to answer questions about material flows. Let us imagine a material coursing through a system from a source where the material is produced to a sink, where it is consumed. The source produces the material at some steady rate and the sink consumes at same rate. The “flow” of the material at any point in the system is intuitively the rate at which the material moves. We can think of each directed edge in a flow network as a conduit for the material. Each conduit has a stated capacity given as a maximum rate at which the material can flow through the conduit such as four hundred litres per hour through a pipe. Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them. We can say in other words the rate at which the material enters a vertex must equal the rate at which it leaves the vertex. We call this property flow conservation.

In the maximum problem we wish to compute the greatest rate at which it leaves the vertex. We will ship the material from the source to the sink without violating any capacity constraints. This problem could be solved by efficient algorithms. Basically there are two general methods for solving maximum flow problem. The classical method as demonstrated by Ford-Fulkerson [6], for finding maximum flow. Secondly the push-relabel method as

demonstrated by A.V Karzanov [9] which underlies many of the fastest algorithms for network flow problems. A particular implementation of the push-relabel method is the relabel-to-front algorithm that runs in time $O(V^3)$. That is the generic algorithm by A.V Goldberg [10]. A computational direct acyclic graph is incorporated in the maximum network flow problem to process the various procedures at the vertices. A sequential algorithm is thus designed with push-relabel method and computation direct acyclic graph as a model for transit of goods in the Ecowas. This sequential algorithm is then multi-threaded so as to take advantage of the multicore technology now available. This is done in Chapter 4 where the variables are multithreaded.

3.1 Study Area

The definitions and theory covered in part of this section follows from definitions and theory of maximum flow by Thomas H. Cormen et al [5]. We will have as the study area flow networks and flows, push-relabel method, Computation Dag, parallel processing, multi-threading in particular. We choose the push relable method over A graph theoretic definition of flow networks, their properties and definitions of maximum flow problem precisely. A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$. We further require that if E contains an edge (u, v) then there is no edge (v, u) in the reverse direction. If $(u, v) \notin E$, then for convenience we define $c(u, v) = 0$, and we disallow self-loops. We distinguish two vertices in a flow network, source s and sink t . For convenience, we assume that each vertex lies on some path from the source to the sink. That is for each vertex $v \in V$, the flow network contains a path s to v to t . The graph is therefore connected and since each vertex other than s has at least one entering edge, $|E| \geq |V| - 1$. The definition of a flow formally, let $G = (V, E)$ be a flow network with a capacity function c . Let s

be the source of the network, and let t be the sink. A flow in G is a real valued function $f: V \times V \rightarrow \mathbb{R}$ that satisfies the following properties.

Capacity constraint for all $u, v \in V$ we require $0 \leq f(u, v) \leq c(u, v)$

Flow Conservation : For all $u \in V - \{s, t\}$, we require $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$

When $(u, v) \notin E$, there can be no flow from u to v , and $f(u, v) = 0$. We call the non negative quantity $f(u, v)$ the flow from vertex u to vertex v . The value $|f|$ of a flow is defined as $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$, that is the total flow out of the source minus the flow into the source.

Typically a flow network will not have any edges into the source and the flow into the source given by the summation $\sum_{v \in V} f(v, s)$ will be 0. We include it however because when we introduce residual networks it will become significant. In the maximum flow problem we are given a flow network G with the source s and the sink t , and we wish to find the maximum flow. A maximum flow problem may have several sources and sink, rather than just one of each, we can reduce the problem of determine a maximum flow problem. We convert the network from multiple source and multiple edges to an ordinary flow network with a single source and a single sink by adding a Supersource s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \dots, m$. We also create a Supersink t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \dots, n$.

The push-relabel approach to computing maximum flows is employed in this paper over the Ford-Fulkerson method . These are the following advantages, why the former method is adopted:

- Push relabel algorithm runs faster than Ford-Fulkerson method
- Ford-Fulkerson algorithm may not terminate if the path is chosen poorly.
- Push relabel algorithm relaxes the law of conservation and thus facilitates preflows.

- Push relabel works in a more localized manner than Ford-Fulkerson which examine the entire residual network.

Goldberg's generic maximum flow algorithm, which has simple implementation that runs in $O(V^2E)$ time thereby improving on the Edmonds-Karp algorithm [7]. The push relabel to front algorithm refines the generic algorithm to obtain a runtime of $O(v^3)$ preflow, which is a function $f: V \times V \rightarrow R$ that satisfies the capacity constraint and the following relaxation conservation: $\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geq 0$ for all vertices $u \in V - \{s\}$. That is the flow into a vertex may exceed the flow out. We call the quantity $e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v)$ the excess flow into vertex u . The excess at a vertex is the amount by which the flow in exceeds the flow out. A vertex $u \in V - \{s, t\}$ is overflowing if $e(u) > 0$.

The intuition behind the generic push relabel algorithm is that directed edges correspond to pipes. Vertices, which are pipe junctions, have two properties. First to accommodate excess flow, each vertex has an outflow pipe leading to an arbitrarily large reservoir that can accumulate fluid. Second, each vertex its reservoir, and all its pipe connections sit on a platform whose height increase as the algorithm increases as the algorithm progresses. Vertex heights determine how flow is pushed. We push flow only downhill, that is, from a higher vertex to a lower vertex. The flow from a lower vertex to a higher vertex may be may be positive, but operations that push flow push it only downhill. We fix the height of the source at $|V|$ and height of the sink at 0. All other vertex heights start at 0 and increase with time. The algorithm first sends as much flow as possible downhill from the source toward the sink. The amount it sends is exactly enough to fill each outgoing pipe from the source to capacity, that is, it sends the capacity of the cut $(s, V - \{s\})$. When flow enters an intermediate vertex, it collects in the vertex's reservoir. From here we eventually push it downhill. We may find that the only pipes that leave a vertex u and are not already saturated with flow connect to vertices that are on the same level as u of its neighbors to which it has unsaturated pipe. After a vertex

is relabeled, therefore, it has at least one outgoing pipe through which we can push more flow. Eventually all the flow that can possibly get through to the sink has arrived there. No more can arrive, because the pipes obey the capacity constraints, the amount of flow across any cut is still limited by the capacity of the cut. To make the preflow a legal flow the algorithm then sends the excess collected in the reservoirs of overflowing vertices back to the source by continuing to relabel vertices to above the fixed height $|V|$ of the source. Once we have emptied all the reservoirs, the preflow is not only a legal flow, it is also a maximum flow.

The third model is the computational model which is at the next higher level of abstraction and offers an abstraction or more formal model of a corresponding architectural model. Thus a computational model provides an analytical method for designing and evaluating algorithms. The complexity of an algorithm should reflect the performance on a real computer. This view is influenced by the architectural design and language, compiler, or the runtime libraries and thus, there exist many different parallel programming models even for the same architecture. In this paper, computational model is used to abstract the model of the movement of goods. In other words a computational model of a maximum flow problem is abstracted using a network flow which is finally multithreaded.

3.2 Design of study

The design of this study expresses how the transit algorithm is to be constructed. It describes the parts involved and how they are to be assembled. This design consists of a set of documents, these are diagrams, together with explanations of what the diagram mean. A form of design which is flowcharts are used together with pseudocodes

Transit Pseudocode:

Let $G = (V, E)$ be a flow network with source s and sink t and let f be a preflow in G . Let the neighbor list $u.N$ for a vertex $u \in V$ be a singly list of neighbors of u in G . Let $u.N.head$ be an attribute points to the first vertex in $u.N$, $v.next$ neighbor points to the next vertex, $u.current$ points to the vertex under current consideration.

The Transit pseudocode

Initialize preflow

List of vertices in any order with the exception of the source s and the sink t .

For each vertex $u \in G.V - \{s, t\}$

The current pointer points to the first vertex.

Start with first vertex in the List

While there are more vertices Do

Old height = Current height

Else stop

If vertex is the first in list then

Compute transit procedure

Store computation in List

Else send vertex to be discharged

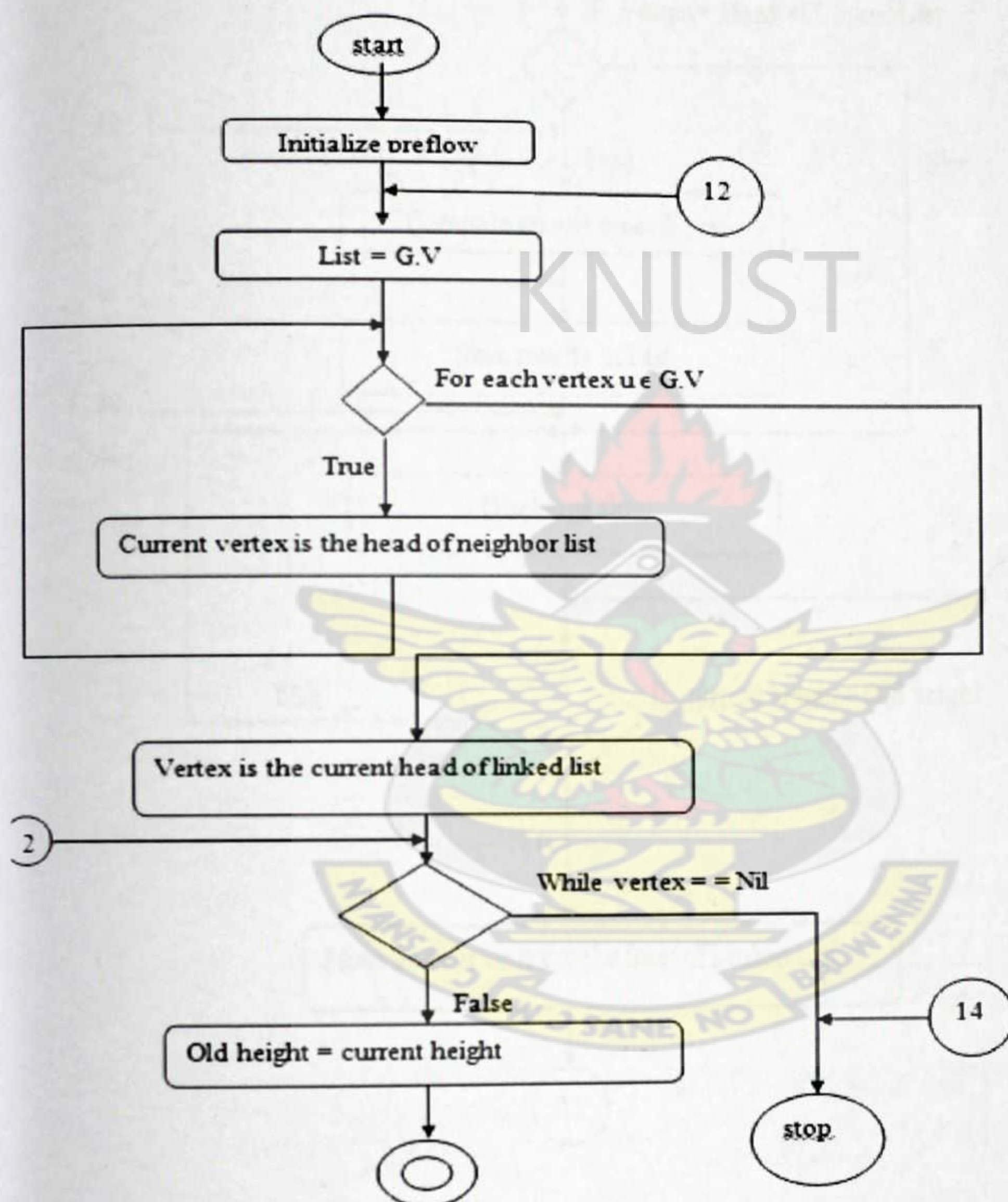
Discharge flow

If current height is greater than old height, true

move vertex to the front of List

Else maintain vertex position in List

Shift pointer to the next vertex in the neighbor list



Source : Author (2012)

Con't Flowchart of Transit algorithm

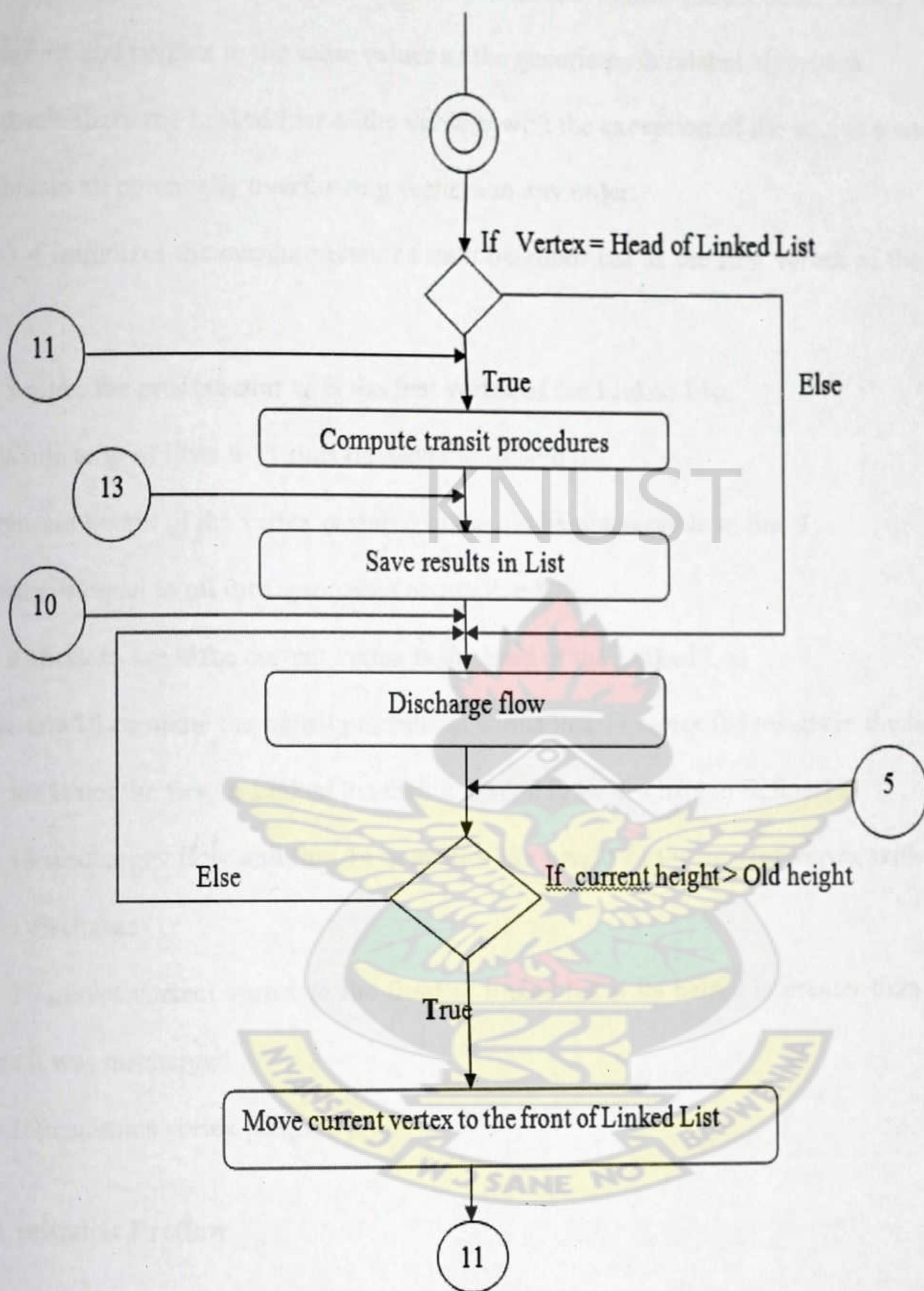


Figure 3.2: Flowchart of Transit algorithm

Source: Author (2012)

In this section we will explain the intuition behind the Transit pseudocode. Line 1 initializes the preflow and heights to the same values as the generic push relabel algorithm.

Line 2 initializes the Linked List of the vertices with the exception of the source s and the sink t to contain all potentially overflowing vertices in any order.

Line 3-4 initializes the current vertex of each neighbor list as the first vertex of the neighbor list

Line 5 make the process start with the first vertex of the Linked List

The While loop of lines 6-11 runs through the Linked list

The current height of the vertex is stored in the old height variable in line 7

If vertex is equal to nil then processes end in line 8

Line 9 check to see if the current vertex is the head of the Linked List

If true line 10 compute the transit procedures whilst line 11 stores the results in the list

If vertex is not the first in Linked list then it is send to be discharged in line 12

Line 13 discharges flow and line 14 compares the height of the current vertex with its height before discharge

Line 15 moves current vertex to the front of linked list if its height is greater than its height before it was discharged

Line 16 maintains vertex position in list

3.2.1 Initialize Preflow

The generic push relabel algorithm uses the following subroutine to create an initial preflow network.

For each vertex v belongs to $G.V$

Initial height of vertex = 0

Initial excess flow at vertex = 0

For each edge $(u,v) \in G.E$

All other vertex carry no flow

Fix the height of source $s = |V|$

For each vertex $v \in s.\text{Adj}$ (adjacent to source s)

Vertex u adjacent to Source s has excess flow to capacity

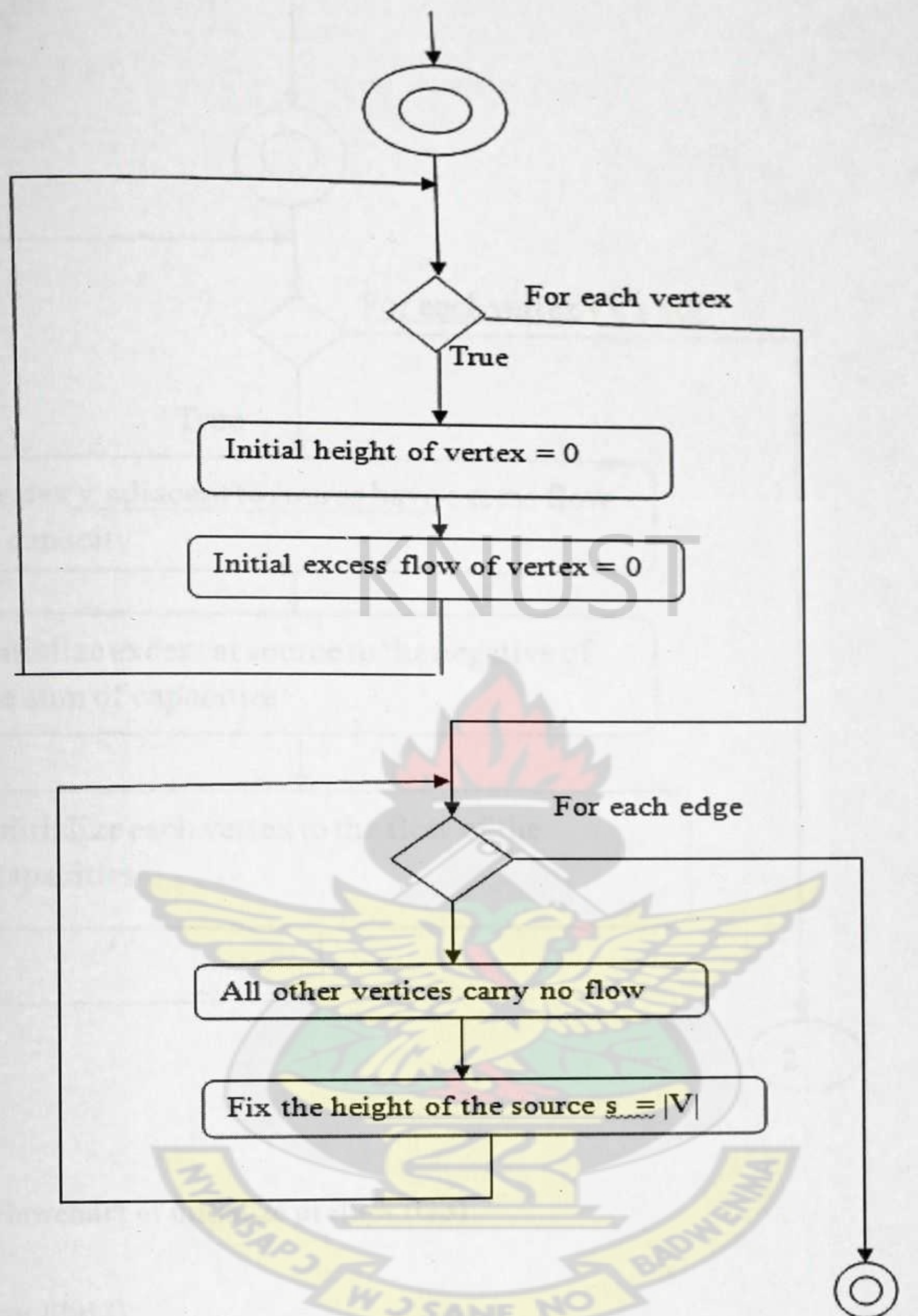
Initializes excess at Source to the negative of the sum of the capacities.

Initialize each vertex to the flow of the capacity

Initialize preflow f creates an initial preflow f defined by

$(u,v).f = c(u,v)$ if $u = s$ or 0 otherwise. That is we fill to capacity each edge leaving the source s , and all other edges carry no flow. For each vertex u adjacent to the source we initially have $v.e = c(s,v)$, and we initialize $s.e$ to the negative of the sum of these capacities. The generic algorithm also begins with an initial height function h , given by $u.h = |V|$ if $u = s$ or 0 otherwise. The above equation defines a height function because the only edges (u,v) for which $u.h > v.h + 1$ are those for which $u = s$ and those edges are saturated, which means that they are not in the residual network. Initialization followed by a sequence of push and relabel operations executed in no particular order yields the generic relabel algorithm.

Flowchart of initialize preflow (G,s)



Source: Author(2012)

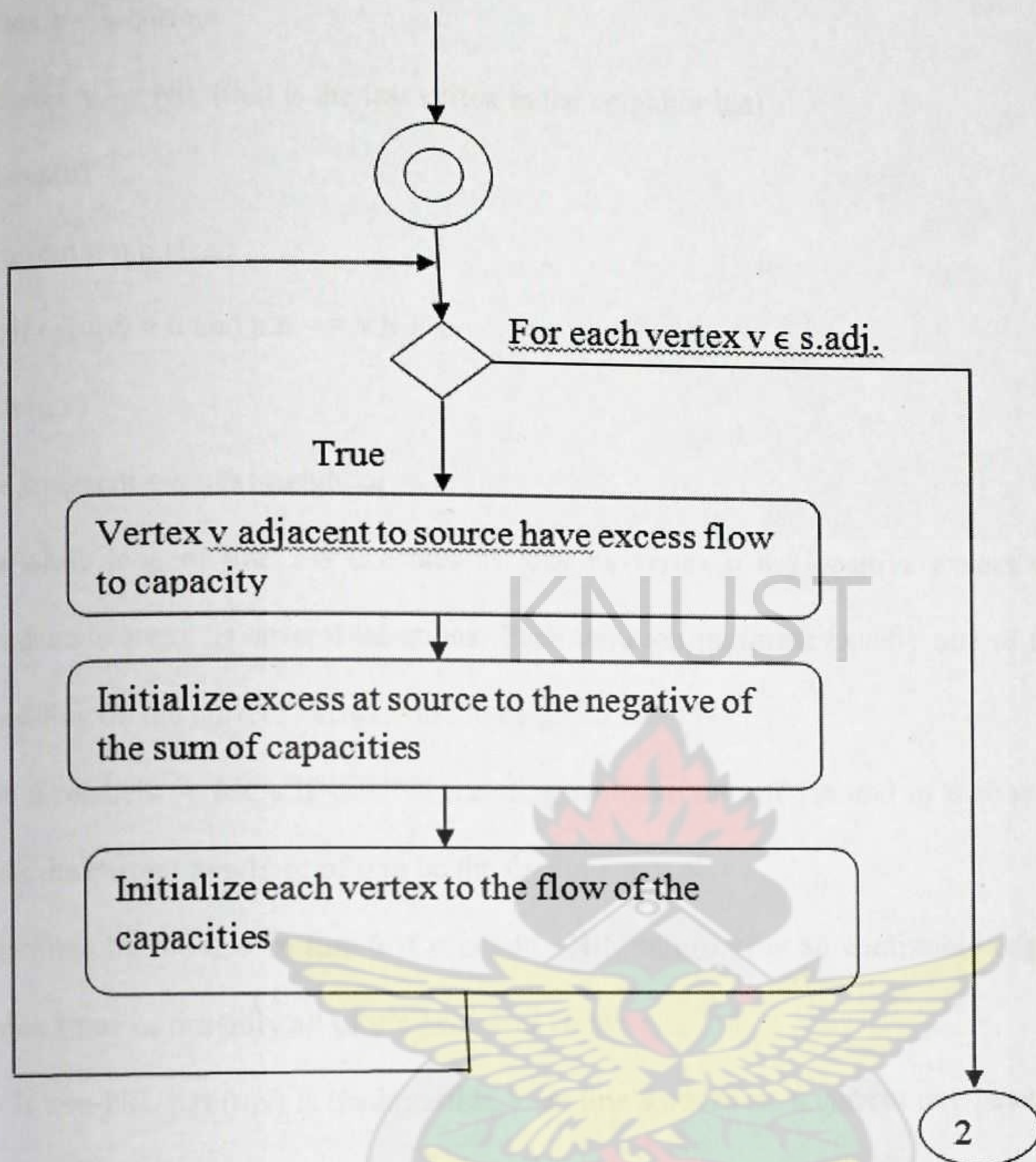


Figure 3.3: Flowchart of initialize preflow (G,s)

Source: Author (2012)

3.2.2 Discharge Pseudocode

An overflowing vertex u is discharged by pushing all of its excess flow through admissible edges to neighboring vertices relabeling vertex u as necessary to cause edges leaving u to become admissible. The pseudocode goes on as follows;

Discharge Pseudocode

While There is excess flow

Vertex $v = u.\text{current}$

If vertex $v == \text{Nil}$, (that is the last vertex in the neighbor list)

Relabel(u)

$U.\text{current} = u.n.\text{Head}$

Elseif $c_f(u,v) > 0$ and $u.h == v.h + 1$

Push(u,v)

Else $u.\text{current} = v.\text{next-neighbor}$

The while loop of line 1-8 executes as long as vertex u has positive excess such that the pseudocode steps is several iterations. Each iteration performs exactly one of three actions, depending on the current vertex v in the neighbor list $u.N$

Line 4 relabels vertex u if v is Nil meaning we have run off the end of $u.N$ and then line 5 resets the current neighbor of u to be the first one in $u.N$

Determine by the text in line 6 if v is non -NIL and (u,v) is an admissible edge then line 7 pushes some or possibly all of u 's excess to vertex v .

If v is non-NIL but (u,v) is inadmissible, then line 8 advances $u.\text{current}$ one position further in the neighbor list $u.N$

If discharge is called on an overflowing vertex u , then the last action performed by Discharge must be a push from u . The procedure terminates only when excess flow $u.e$ becomes zero, and neither the relabel operation or advancing the pointer $u.\text{current}$ affects the value of excess flow $u.e$. We must be sure that when Push or Relabel operation is called by Discharge, the operation applies.

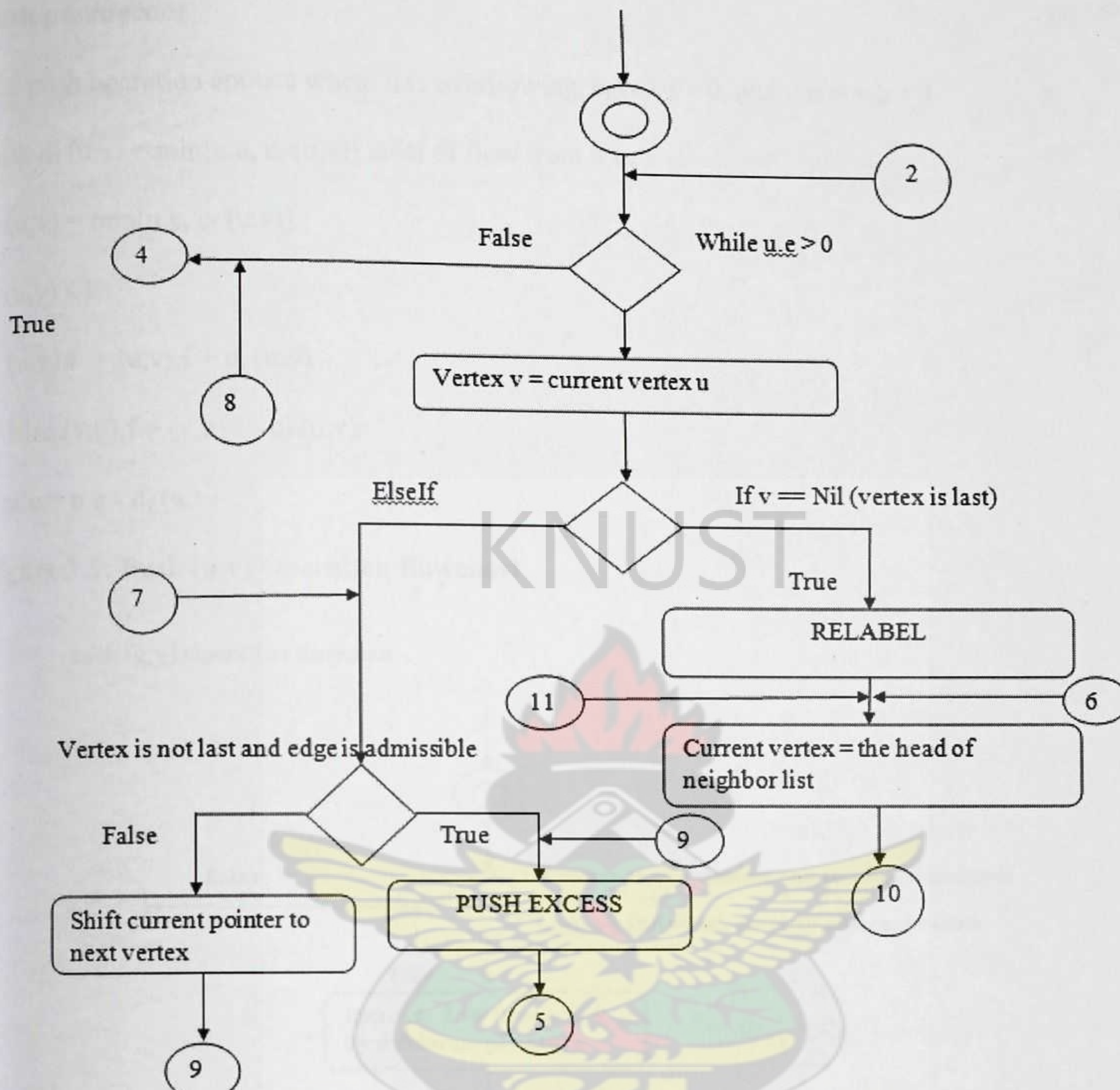


Figure:3.4 Discharge Flowchart

Source : (2012)

3.2.3.The Pseudocode of the push operation

Introduction of some useful notation of the push operation are as follows; $c_f(u,v)$ is a residual capacity in constant time given c and f . The attribute $u.e$, is where the excess flow at the vertex u is stored. The attribute $u.h$ is the height of u . The expression $d_f(u,v)$ is a temporal variable that stores the amount of flow that we can push from u to v .

Push pseudocode

The push operation applies when: u is overflowing, $c_f(u,v) > 0$, and $u.h = v.h + 1$

Push $d_f(u,v) = \min(u.e, c_f(u,v))$ units of flow from u to v

$d_f(u,v) = \min(u.e, c_f(u,v))$

If $(u,v) \in E$

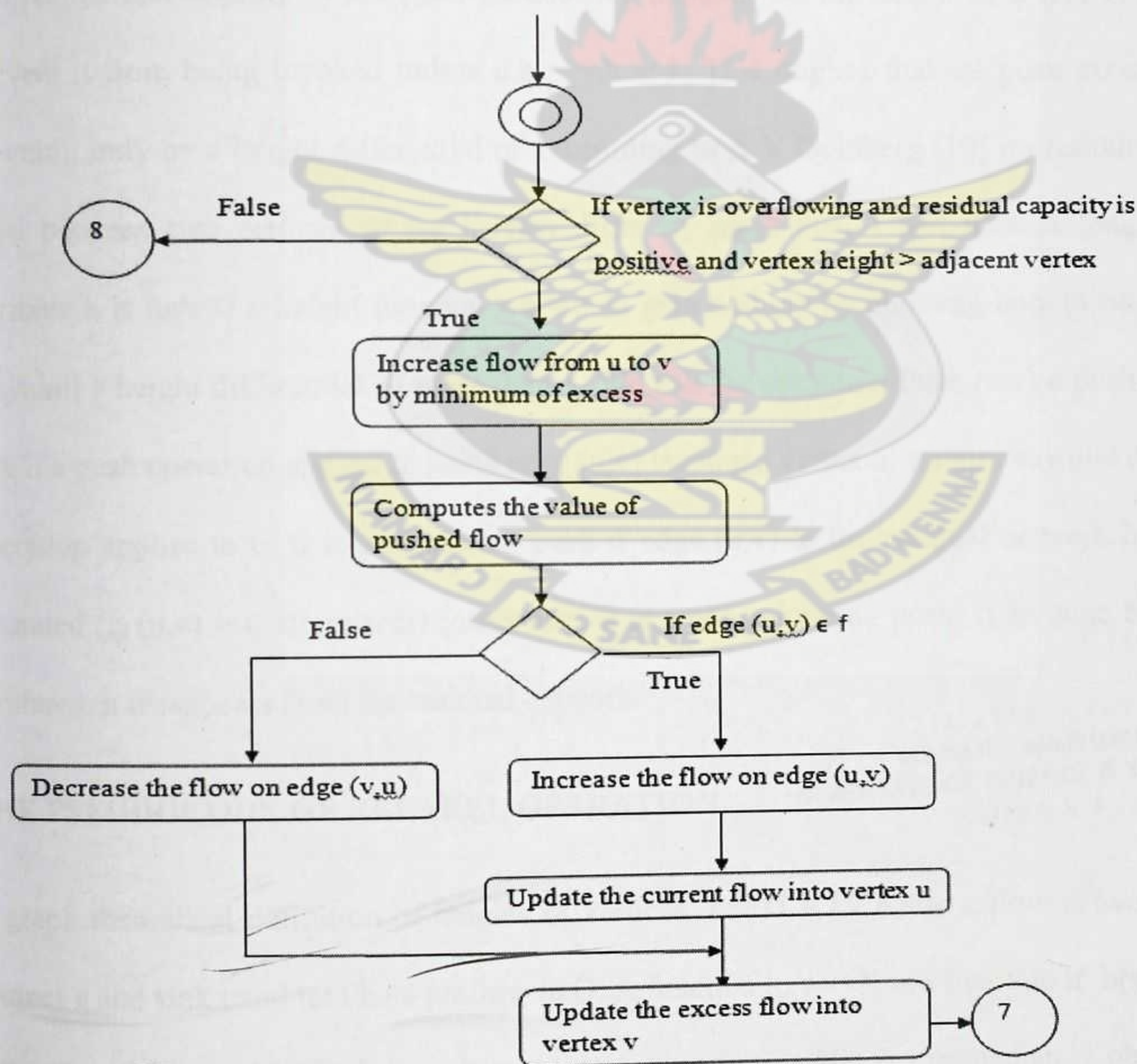
$(u,v).f = (u,v).f + d_f(u,v)$

Else $(v,u).f = (v,u).f - d_f(u,v)$

$u.e = u.e - d_f(u,v)$

Figure 3.5: Push (u,v) Operation flowchart

Push (u,v) Operation flowchart



The Push method operates as follows:

We can increase the flow from u to v by $d_f(u,v) = \min(u.e, c_f(u,v))$ without causing $u.e$ to become negative or the capacity $c(u,v)$ to be exceeded, because vertex u has a positive excess $u.e$ and the residual capacity of (u,v) is positive. Line 3 computes the value of $d_f(u,v)$ and line 4-6 updates f . Line 5 increase the flow on edge (u,v) , because we are pushing flow over a residual edge that is also an original edge. Line 6 decreases the flow on edge (u,v) because the residual edge is actually the reverse of an edge in the original network. Lines 7-8 updates the excess flow into vertices u and v and thus f is a preflow before Push is called, it remains a preflow afterward.

We realize that nothing in the push pseudocode depends on the height of u and v , but we prevent it from being invoked unless $u.h = v.h + 1$. This implies that we push excess flow downhill only by a height differential of 1 according to A.V Goldberg [10] no residual edges exist between two vertices whose heights differ by more than 1 and thus as long as the attribute h is indeed a height function we would gain nothing by allowing how to be pushed downhill a height differential of more than 1. We call the operation Push (u,v) a push from u to v if a push operation applies to some edge (u,v) leaving a vertex u , we also say that the push operation applies to u . It is a saturating push if edge (u,v) in the residual network becomes saturated ($c_f(u,v) = 0$ afterwards) ;otherwise, it is a non saturating push. If an edge becomes saturated, it disappears from the residual network.

THE PSEUDOCODE OF RELABEL OPERATION

LIBRARY
KWAME NKRUMAH
UNIVERSITY OF SCIENCE & TECHNOLOGY
KUMASI

A graph theoretical definition of heights of vertices. Let $G = (V,E)$ be a flow network with sources s and sink t and let f be a preflow in G . A function $h: v \rightarrow \mathbb{N}$ is a function if $h(s) = |V|$, $h(t) = 0$ and $h(u) \leq h(v) + 1$ for every residual edge $(u,v) \in E_f$. We immediately obtain the following lemma: Let $G = (V,E)$ be a flow network, let f be a preflow in G , and let h be a

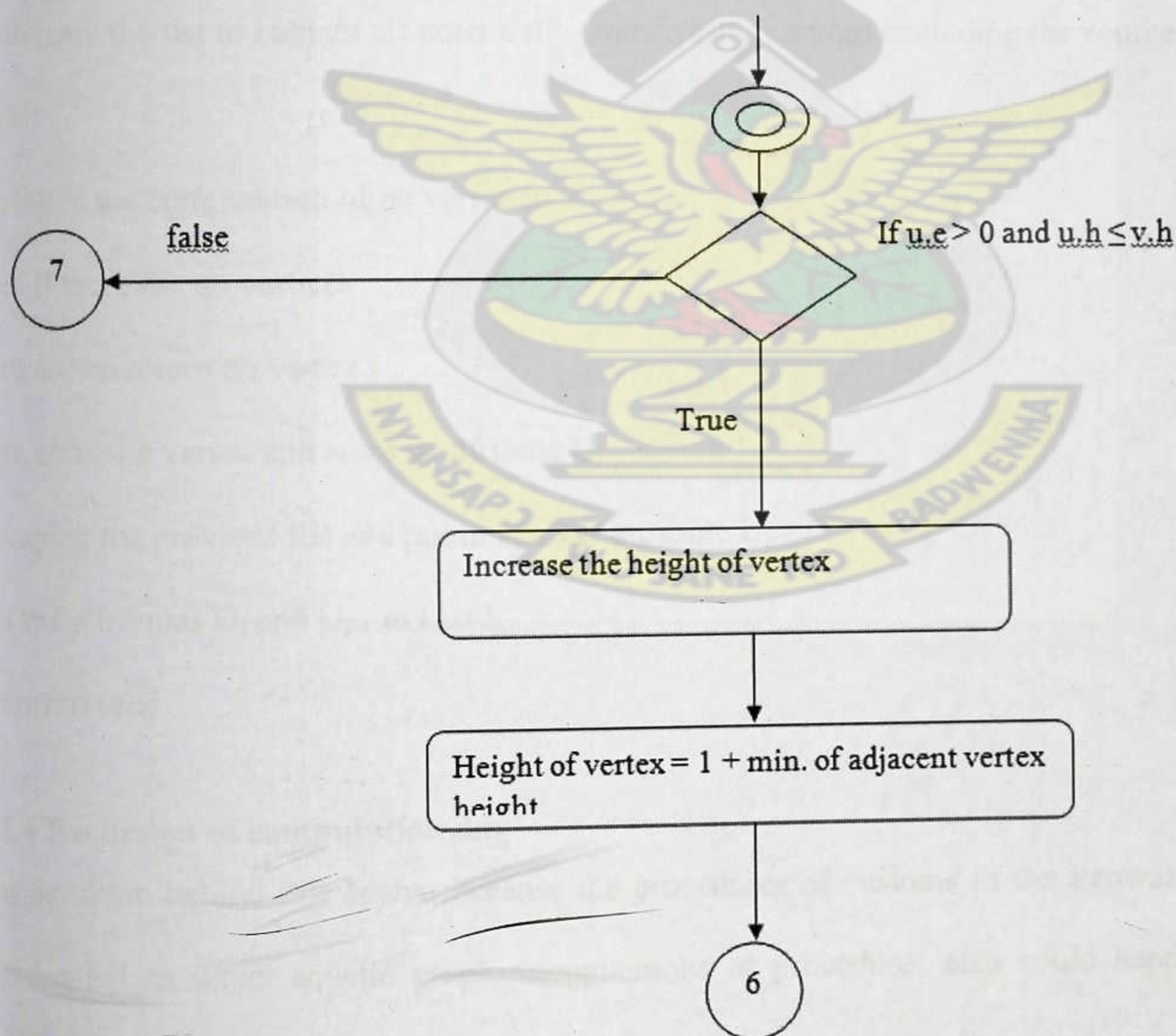
height function on V , for any two vertices $u, v \in V$, if $h(u) > h(v) + 1$ then (u, v) is not an edge in the residual network. The basic operation Relabel (u) applies if u is overflowing and if $u.h \leq v.h$ for all edges $(u, v) \in E_f$. In other words we can relabel an overflowing vertex u if for every vertex v for which there is residual capacity from u to v , flow cannot be pushed from u to v because v is not downhill from u . The source s and the sink t by definition cannot be overflowing and so s and t are negligible for relabeling.

.Relabel (u) pseudocode

Relabel applies when u is overflowing and for all $v \in V$ such that $(u, v) \in E_f$ we have $u.h \leq v.h$

Increase the height of u : $v.h = 1 + \min\{v.h : (u, v) \in E_f\}$

Figure 3.6 Flowchart of Relabel Operation



Source : Author (2012)

When we call the operation Relabel (u) we say that the vertex u is relabeled. Note that when u is relabeled, E_f must contain at least one edge that leaves u , so that the minimization in the code is over a nonempty set. This property follows from the assumption that u is overflowing, which in turn tells us that: $u.e = \sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) > 0$. Since all flows are nonnegative, we must therefore have at least one vertex v such that $(v,u).f > 0$. But then, $c_f(u,v) > 0$, which implies that $(u,v) \in E_f$. The operation Relabel (u) thus gives us the greatest height allowed by the constraints on height functions.

COMPUTATION DIRECT ACYCLIC GRAPH (COMPUTATION DAG)

The computation dag algorithm executes the procedures at the vertex and stores the output in the list.

THE COMPUTATION DAG PSEUDOCODE

Initializes the list to contain all potentially overflowing vertices excluding the source s and the sink t .

Initialize the computation of no vertex to 0

Test if there are no vertices

If true then return no vertex

Else compute vertex and store in attribute U_i

Compute the previous list and put in another attribute U_{i-1}

Put the attributes U_i and U_{i-1} in List L

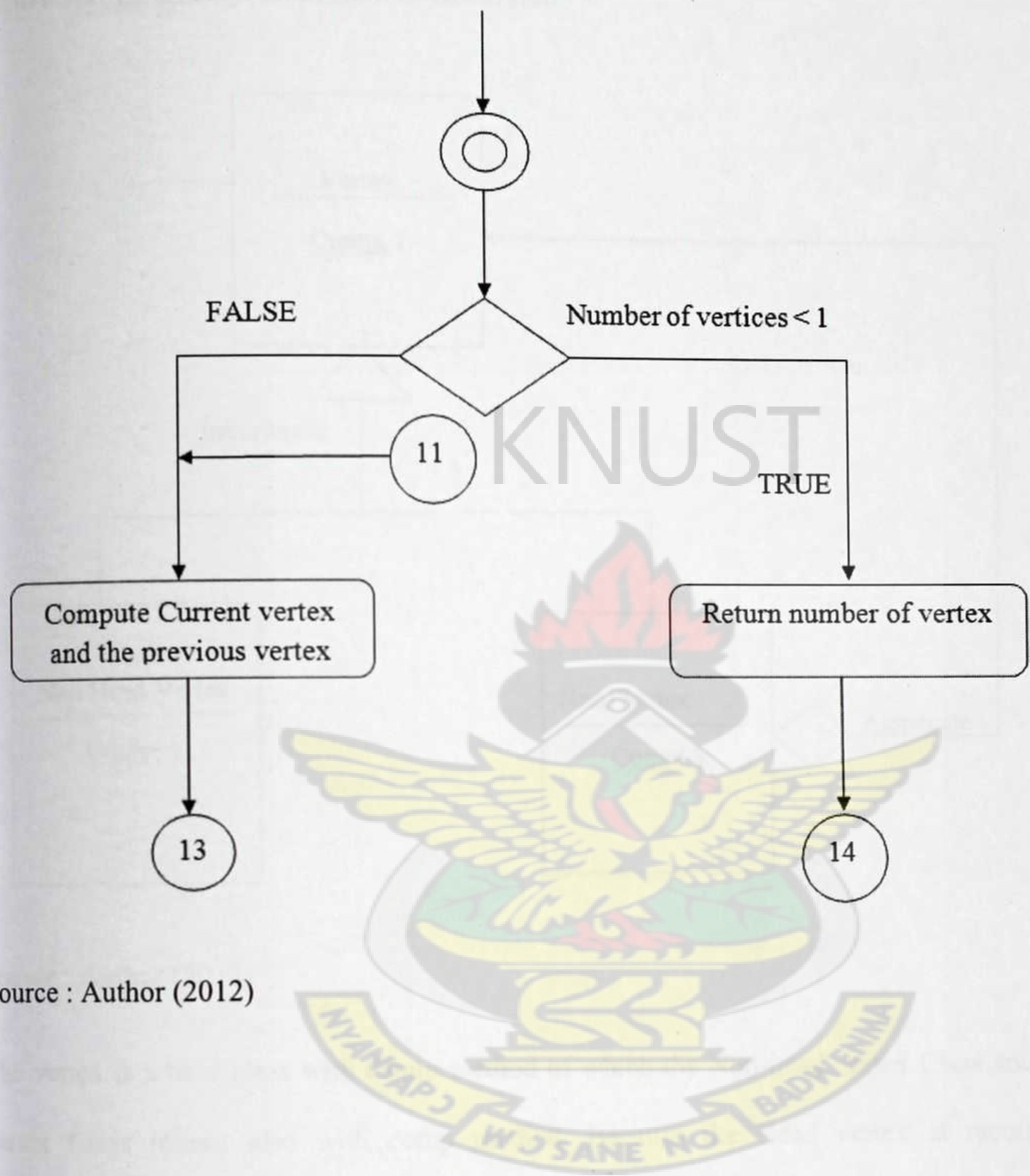
Return results

3.2.4 Recursion of computation dag

The intuition behind this is that because the procedures of customs in the Ecowas could be represented as direct acyclic graph, computations of procedures also could happen at the vertices.

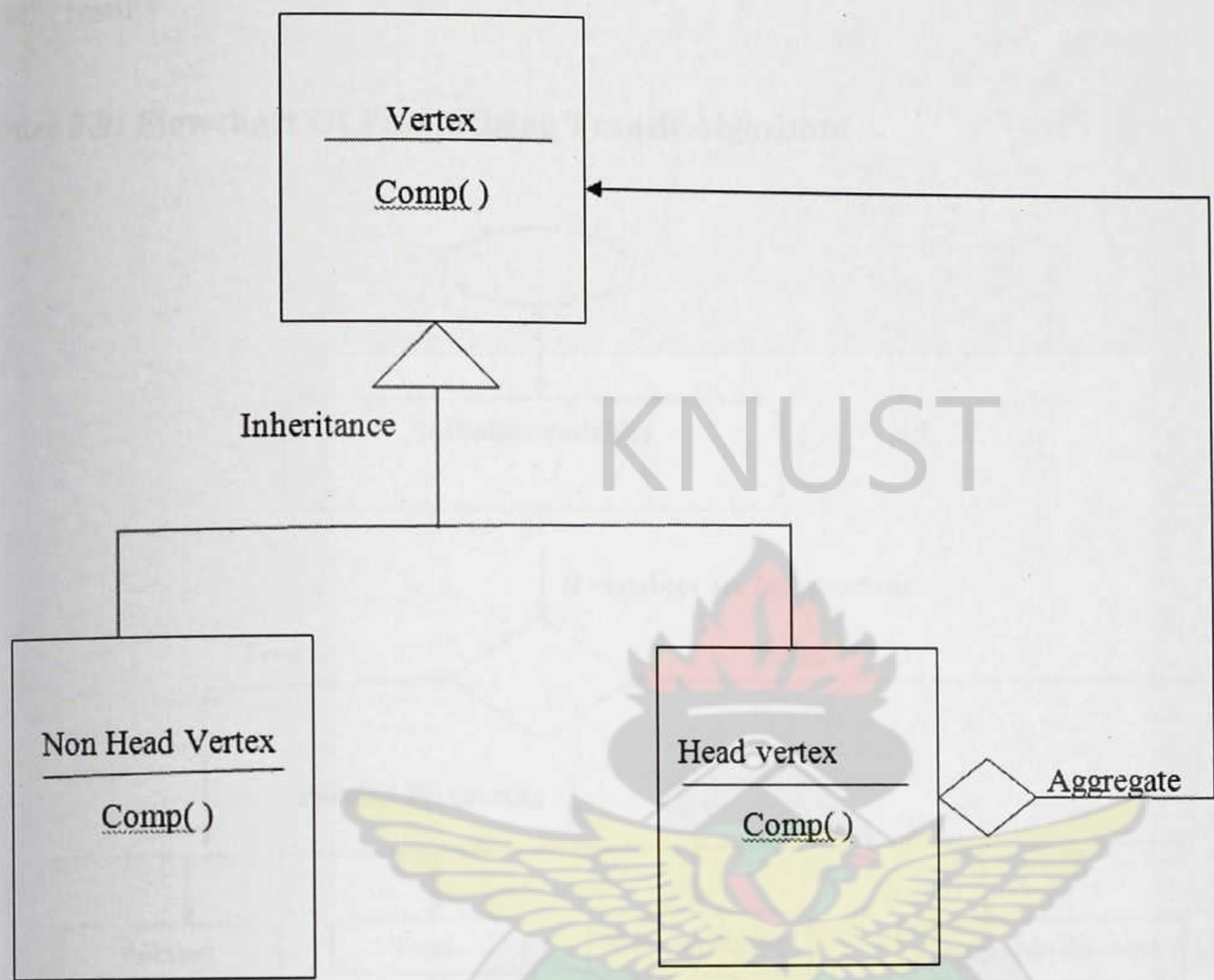
A tree of recursive procedure instances when computing could be formed. Each instance of computation with the same argument does the same work to produce the same results providing an efficient way to compute the direct acyclic graph. By applying dynamic method which works as follows, efficient results are obtained. Having observed that a naïve recursive solution is inefficient because it solves the same sub problems repeatedly, we arrange for each sub problem to be solved only once, saving its results. If we need to refer to this sub problem's solution again later, we can just look it up, rather than re-compute it. Dynamic programming thus uses additional memory to save computation time. It serves an example of time – memory trade off. The savings may be dramatic an exponential time solution may be transformed into a polynomial-time solution. A dynamic programming approach runs in polynomial time when the number of distinct sub problems involved is polynomial in input size and we can solve each sub problem in polynomial time. We will use top down with memoization. In this approach we write the procedure recursively in a natural manner but modified to save the result of each sub problem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this problem. If so it returns the saved value, saving further computation at this level, if not the procedure computes the value in the usual manner. We say that the recursive procedure has been memoized, it remembered what results it has computed previously.

Figure 3.7: Flowchart of Computation Dag



Source : Author (2012)

Figure 3.8: UML Representation of Recursion



Source : Author (2012)

The vertex is a base class with **Comp** method of which the Non-head vertex Class and Head vertex Class inherit also with **comp** method. Because the Head vertex is recursive it aggregates in the Base vertex Class in figure 3.9 above.

3.2.5 Parallelizing the Transit Sequential Algorithm

PSEUDOCODE OF THE PARALLEL TRANSIT ALGORITHM

Initialize the variables

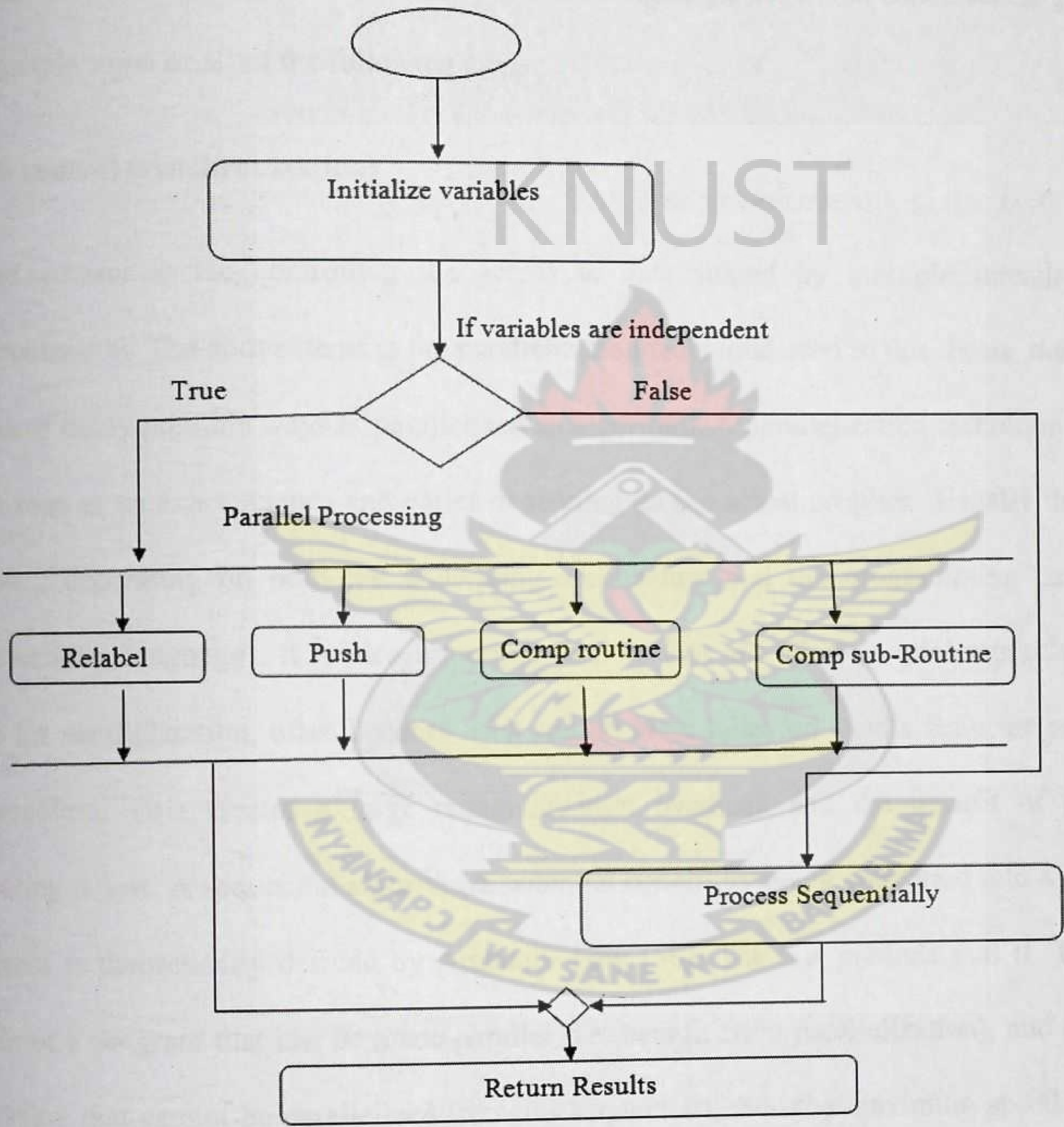
If variables are independent

Process Reliable method, Push method, Compute routine and Compute sub routine together

Else process sequentially

Return results

Figure 3.9: Flowchart Of Parallelizing Transit Algorithm



Source : Author (2012)

The sequential Transit algorithm made up by the combination of a Push Relabel method and a Computation Direct Acyclic Graph is parallelized with its attendant property of concurrency. From Peter Wind et al [57] a parallel algorithm could be described as the steps involved in

solving a given problem on a parallel computer. This however, is an over simplification, as the development of parallel algorithms involves much more than just describing the steps of the computation. At the very least, a parallel algorithm has the added dimension of concurrency and the algorithm designer must specify the sets of steps that can be executed simultaneously. Simultaneously or concurrent execution is essential for obtaining any performance benefits from the use of a multi-core computer. In practice, designing a nontrivial concurrent algorithm may include some or all of the following steps :

Access control (synchronization)

Access control is the controlling the access to data shared by multiple threads using synchronization. The above items is the parallelization technique used in this thesis, but is just one out of many possible ways of parallelized an algorithm. A parallelization technique should not be seen as an exact science and varies depending on the actual problem. Usually the steps also vary depending on both the underlying architecture and the programming paradigm (programming language). It is also important to note that not all sequential algorithms are suited for parallelization, often because each sub problem relies on results from the previous Sub problem. This creates a large communication overhead and the benefit of parallel computing is lost. A sequential algorithm's potential benefit by being converted into a parallel algorithm is theoretically defined by Amdahl's Law [56]. The law predicts that if P is the portion of a program that can be made parallel (i.e. benefit from parallelization), and $(1-P)$ is the portion that cannot be parallelized (remains sequential), then the maximum speed up that can be achieved by using N processors is given by the equation $1/(1-P) + P/N$

By taking the limit as N approaches infinity we are left with the equation $1/(1 - P)$ and clearly as the factor of parallelization increases the speed up increases.

Another important problem to have in mind when decomposing a problem is the granularity of the decomposition. The granularity refers to the size of the subtasks compared to the main problem. If the decomposition consists of a large number of small tasks it is called fine-grained, and if the decomposition consists of fewer larger tasks it is called coarse-grained.

The granularity can have a big impact on the performance of an algorithm in two ways. If the decomposition is too fine-grained, each task is not very computation heavy and much time is used on communication between the different tasks. If the decomposition on the other hand is too coarse-grained, not enough operations can be done concurrently and some processors may be idle for some time. The key is to find the correct task-size so that each processor is always occupied and only minimal time is spent on communication. This optimal task-size can be hard to figure out and is most often achieved through trial-and-error.

Recursive decomposition

Recursive decomposition represents a different and complementary way of thinking about problems. In this approach, the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. This method is useful if the computation can be divided into disjoint tasks which often is the case for algorithm usually solved using the divide-and-conquer strategy. In this technique, a problem is solved by first dividing it into a set of independent sub problems. Each one of these sub problems is solved by recursively applying a similar division into smaller sub problems followed by a combination of their results. The divide-and-conquer strategy results in natural concurrency, as the different sub problems can be solved concurrently. If the sub problems are not completely disjoint (as they are in divide-and-conquer algorithms), the recursive decomposition technique can still be used. However, this requires some communication

between the different tasks, which in many cases isn't trivial. If the communication overhead is too large the data decomposition method should be considered instead.

3.2.6. Parallelizing the push and the relabel methods

In this chapter we are not going to indulge very much on mapping process since it is the preserve of implementation. We will therefore go straight to parallelizing the push relabel method which is an integral part of the Transit algorithm. The idea behind parallelizing push-relabel according to Victoria Popic et al [59] is to find a way to discharge active nodes in parallel. Discharging nodes in parallel (as well as running the concurrent global update) introduces many races. In order to avoid them, locks need to be used in several sections of the code. For example, when flow is pushed between two nodes, both nodes are locked in order to update the flow excess and residual capacity values and prevent a concurrent relabeling of one of the nodes. The node being relabeled needs to be locked throughout the entire operation in order to update its distance label and prevent flow from being pushed to it creating an admissible arc at a lower level. Adding newly activated nodes to the buckets, also requires a locking mechanism or an atomic operation. Finally, we also need to ensure that the updates to the global variables (such as a Max) are performed correctly (a lock is currently used in all the implementations to achieve this).

From the ensuing reason of the push pseudocode and Relabel pseudocode we can parallelize the push and the relabel methods as the excess flow into vertices u and v and thus f is a preflow before Push is called, it remains a preflow afterward. We realize that nothing in the push pseudocode depends on the height of u and v , but we prevent it from being invoked unless $u.h = v.h + 1$. This implies that we push excess flow downhill only by a height differential of 1. A.V. Goldberg [10] which states that given $G = (V, E)$ be a flow network, f be a preflow in G , and h be a height function on V , for any two vertices $u, v \in V$ if $h(u) > h(v) +$

1 then (u,v) is not in the residual network, no residual edges exist between two vertices whose heights differ by more than 1 and thus as long as the attribute h is indeed a height function we would gain nothing by allowing how to be pushed downhill a height differential of more than 1. We call the operation $\text{Push}(u,v)$ a push from u to v if a push operation. From the above analysis parallelizing the push method and the relabel method is feasible. An overflowing vertex can be either pushed or relabeled. That is let $G=(V,E)$ be a flow network with source s and sink t , let f be a preflow, and let h be any height function for f . If u is any overflowing vertex, then either a push or relabel operation applies to it.

Proof: For any residual edge (u,v) , we have $h(u) \leq h(v) + 1$ because h is a height function. If a push operation does not apply to an overflowing vertex u , then for all residual edges (u,v) , we must have $h(u,v) < h(v) + 1$, which implies $h(u) \leq h(v)$. Thus, a relabel operation applies to u .

3.2.7 Parallelizing the Computation Dag.

Many multithreaded algorithms involving nested parallelism follow naturally from the divide and conquer paradigm. Moreover just as serial divide and conquer lend themselves to analysis by solving recurrences, so do multithreaded algorithms. From the section on recursion above, it emerged that a tree of recursive procedure instances when computing could be formed. Each instance of computation with the same argument does the same work to produce the same results providing an efficient way to compute the direct acyclic graph. By applying dynamic method which works as follows, efficient results are obtained. Having observed that a naïve recursive solution is inefficient because it solves the same sub problems repeatedly, we arrange for each sub problem to be solved only once, saving its results. If we need to refer to this sub problem's solution again later, we can just look it up, rather than re-compute it. We can thus allocate the procedure of a sub problem to be computed at the same time as its sub

procedure known as its child and the parent as the procedure itself. By this way we parallelize the computation dag. In parallel programming paradigm it is known as spawning the child procedure.

3.3 Variables

3.3.1. Parallel Global Updates

From [parallelizing] Parallel global update condition, states that given any vertex u , it must be the case that $h(u) \leq h(v)+1$ for all the edges (u,v) in the residual graph G_f and a violation of this condition might result in incorrect results. They associate a wave number with each node, which stores the number of times the node was globally relabeled. It is only allowed to push flow between nodes with the same wave number and the two nodes must be both locked when the push occurs. It is also important to make sure that in any distance label update the node's distance label is never decreased. The relabel operation needs to lock the node being relabeled, similarly the global update also needs to lock each node it reaches. There are two global variables to store the current global update wave number and the current level in the breadth first search tree. Further information and correctness proofs can be found in Thomas H. Cormen et al [5].

3.3.2 .Concurrent Global-Update

The global update needs to run periodically during the push-relabel algorithm. If the push-relabel algorithm is parallelized, then all the processors would need to be suspended in order to run the global update. Anderson et al [27] has presented a correct method for running the global update concurrently with a parallel implementation of the push relabel algorithm. This method ensures that the valid labeling condition of Goldberg's algorithm is met. The valid labeling order to guarantee an approximate highest-level first traversal.

3.3.3 Computational Dag Global Variables (Edges) And Private Variables (Vertices)

From Thomas H. Cormen et al [5] we can think of a multithreaded computation as the set of runtime instructions, executed by a processor on behalf of a multithreaded program, as a directed acyclic graph $G = (V, E)$ called a computation dag. Conceptually, the vertices in V are instructions, and the edges in E represent dependencies between instructions, where $(u, v) \in E$ means that instruction u must execute before instruction v . For convenience, however, if a chain of instructions contains no parallel control (no spawn, sync, or return from a spawn via either an explicit return statement or the return that happens implicitly upon reaching the end of a procedure), we may group them into a single strand, each of which represents one or more instructions. Instructions involving parallel control are not included in strands, but are represented in the structure of the dag. For example, if a strand has two successors, one of them must have been spawned, and a strand with multiple predecessors indicates the predecessors joined because of a sync statement. Thus, in the general case, the set V forms the set of strands, and the set E of directed edges represents dependencies between strands induced by parallel control. From the above it can therefore be deduced that edges represent global variables whereas the vertices or the structure of the dag house the private variables.

3.4 Methods of Analysis

Methods of analysis is to verify the correctness of designs, in other words its sufficiency, from Eric Braude [55] there are informal and formal approaches to these questions. To verify a design informally is to be convinced that it covers the required functionality. Formal methods for establishing correctness involves applying mathematical logic to analyzing the way in which the variables change. Formal methods for correctness are usually applied when design enters the detailed stage. In this paper we will use informal approach to correctness as the method of analysis.

3.4.1 Informal approaches to correctness

Informal approaches to correctness are based on the common sense approaches to correctness, which are based on the idea that before we can proclaim a design to be correct we have to understand it completely. Thus designs and implementation should be readable. Since designs are most often complex, we have to modularize designs, that is break them down into separate understandable parts. Since modularization is a key way to assess the correctness of a design, we first need to understand how client code uses modules. Modules for this paper will either be classes or packages of classes. An interface is a set of function forms (or prototypes). A modules interface defines its uses. This paper is parallel computational processing rather than parallel programming processing nevertheless several object oriented concepts are used. For modularization, all related objects has been divided into the following modules.

- . algorithm: The top package for all algorithm related subdomains.
- . algorithm.parallel: Contains all versions of the parallelized algorithms.
- . algorithm.sequential: Contains the basic sequential push relabel algorithms.
- . edge: Contains all classes related to edges in a graph.
- . vertex: Contains all classes related to vertices in a graph.
- . graph: Contains all classes related to the representation of a graph.

All the above packages define the graph framework and the algorithm. As describes earlier, all classes implements an interface and all relations between classes passes through these interfaces.

3.4.2 Graph framework

Graph framework describes how graph components are represented. There exists several pre-made Graphs Frameworks such as the Java Universal Network Graph Framework (JUNG) Joshua O'Madahain et al [60] which represents graphs edges etc. but a simple framework is made and that allow us to define the exact functionality. From Peter Wind and John Hansen [57] the definitions of the objects are as follows:

Vertex

A Vertex is very simple in our framework. It only consists of an identifier to make it uniquely determinable. A vertex could also simply have been represented by index numbers in an array, but as this thesis is written with the Object Oriented programming paradigm in mind, we have chosen to let a vertex be a class of its own.

Edge

As the name implies, this is a representation of an edge in a graph. It contains information about the two vertices it connect (source and target), and a reference to the complement edge - that is, the edge which has opposite source and target respectively. An edge also contains information about its current flow and capacity which both can be set or retrieved.

Edge Container

The Edge Container class contains a list of Edge Interface's and is in other word a wrapper. The reason for using a wrapper to contain edge Interface's is to have a standard way of collecting edges. If there is a need to change the data structure in which the edges is collected in, only one class should be changed.

Graph

A Graph contains information of all vertices in the graph. The graph has association to several vertices and two vertices are explicitly defined as being the source and the sink. The graph also has references to several Edge Container objects. There exist one Edge Container for each vertex, and Edge Containers are used to represent all outgoing edges for the corresponding vertex (the neighbors of the vertex).

3.4.3 The Algorithms Framework

This section describes how the algorithm framework is modularized and also an UML-like illustration of the complete algorithm framework. It is shown that every algorithm implements the Max-Flow Algorithm Interface. This makes it easy to develop new algorithms and use it in our existing framework. An abstract object Max-Flow Algorithm represents the basic functionality in a Maximum Flow algorithm without the actual algorithm. The packages `algorithm.reference` and `algorithm.sequential` contains algorithms which are extending by the Max-Flow Algorithm and will not be discussed further as the title of the classes should be self explanatory.

The `algorithm.parallel` package contains all the parallel implementations of the algorithms. In this package, some classes are introduced:

PushRelabelParallel Worker: This class acts as the worker that performs the relabel and push operations and because it extends a thread it can have multiple instances running simultaneously.

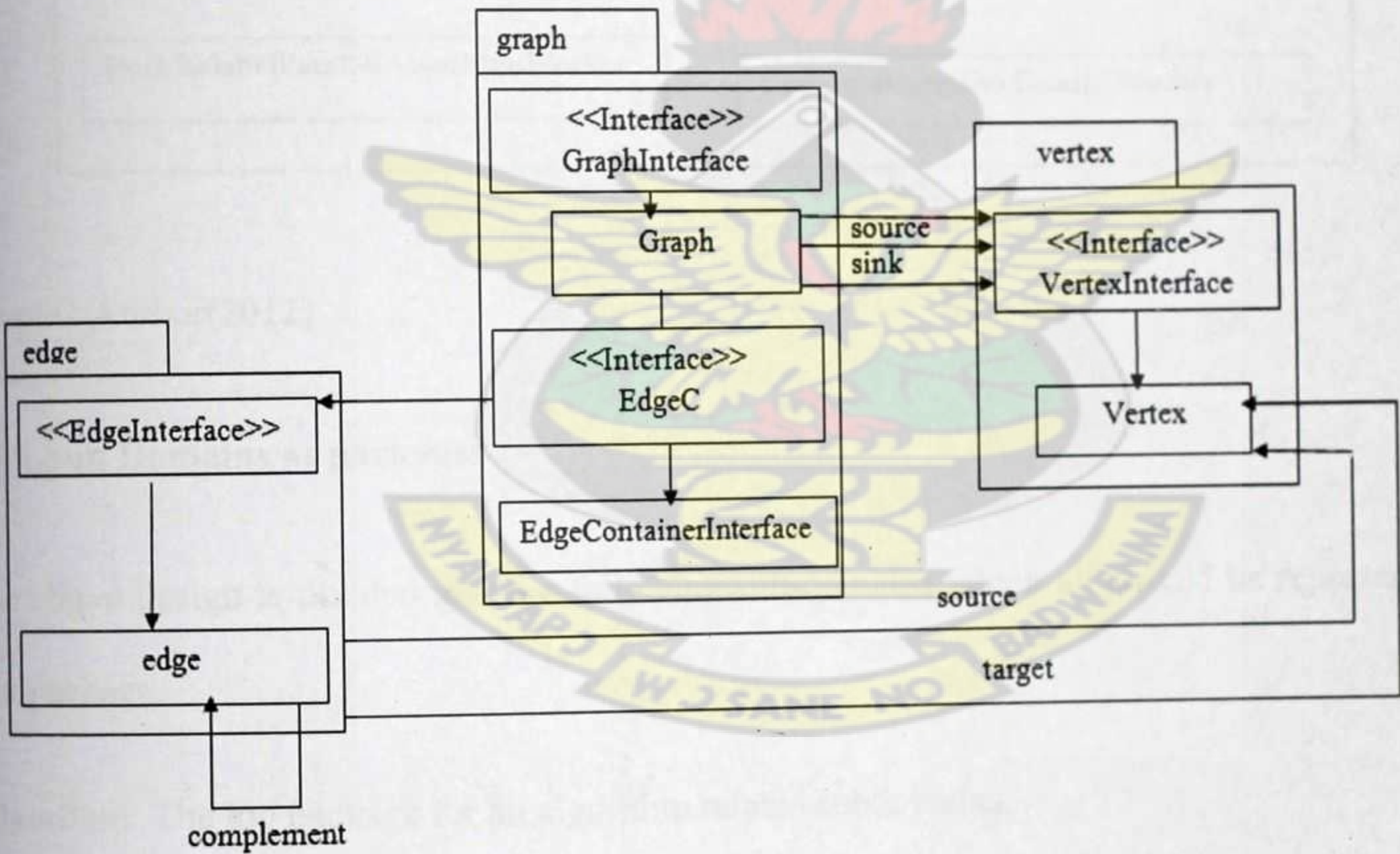
PushRelabelParallel Works as an "organizer" or manager which creates a number of worker threads and "distribute" the subproblems to them. The worker threads are represented by

PushRelabelParallelWorker's and the number of associated workers are defined by the number of processes chosen.

ComputationDagParallelWorker: This class acts as the worker that is responsible for nested parallelism, that is, it allows a subroutine to be spawned allowing the caller to proceed while the spawned subroutine is computing its results, and also the execution of parallel loops whereby the iterations of the loop can execute concurrently.

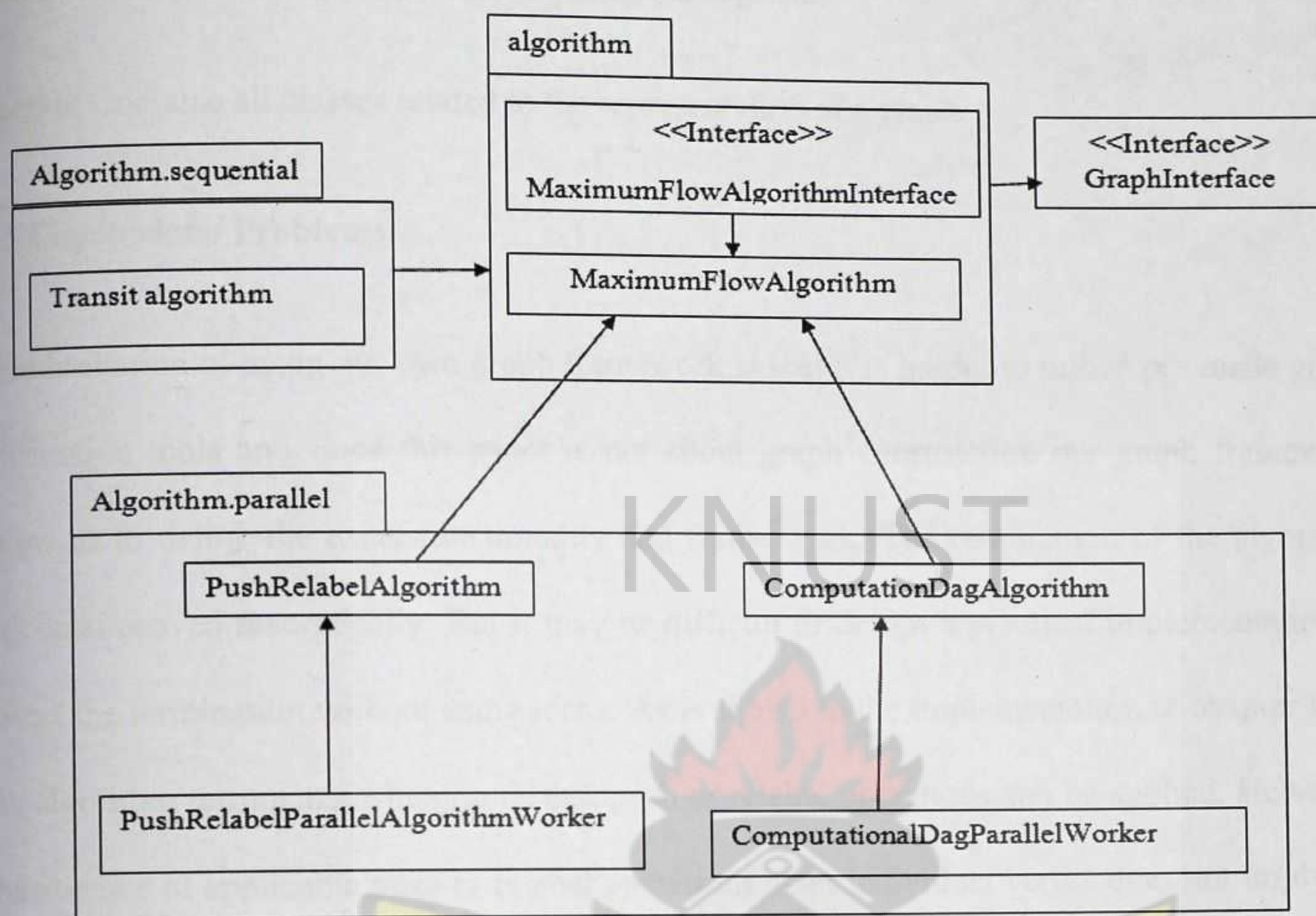
ComputationDagParallel works as an organizer or manager which creates a number of threads and distribute the subroutines and loops to them.

Figure 3.10: Graph Framework



Source : Java Universal Network Graph Framework(JUNG)

Figure 3.11: Object Oriented representation of parallel algorithm



Source : Author(2012)

3.4.4 Sub Domains as packages

The above design is divided into the following sub-domains which all should be represented as a package:

- . algorithm: The top package for all algorithm related subdomains.
- . algorithm.parallel: Contains all versions of the parallelized algorithms.
- . algorithm.reference: Contains the reference-algorithms which are used to determine the correctness of the results gained from the parallelized algorithms.
- . algorithm.sequential: Contains the basic sequential push relabel algorithms.

.edge: Contains all classes related to edges in a graph.

.vertex: Contains all classes related to vertices in a graph.

.graph: Contains all classes related to the representation of a graph.

3.5 Constraints/ Problems

One limitation of using our own graph framework is that it is harder to utilize pre made graph illustration tools and since this paper is not about graph construction our graph framework allow us to define the exact functionality and frame work. The termination of the algorithm has been proved theoretically. But it may be difficult to design a practical implementation to detect the termination without using locks. As is shown in the implementation in chapter four, the algorithm terminates when no further push or relabel operations can be applied. However, the absence of applicable push or relabel operations at an individual vertex does not imply the termination, because other vertices may be active. Furthermore, another vertex may push flow to this idling vertex, making it active again. The termination of the algorithm, which becomes true only when we do not have any applicable push or relabel operations at any vertices, needs to be detected with the help of a global barrier. Barriers are implemented using locks, however. To derive a completely lock-free algorithm, further study is needed for the efficient detection of algorithm termination. If lock free termination detection is infeasible, we need to develop efficient methods that are practically implementable.

CHAPTER FOUR

ANALYSIS OF FINDINGS

4.0 Introduction

The words “implementation” and “programming” are used for “coding”, according to Eric Braude [55]. The word “development” is sometimes used. This section discusses implementation in the context of parallel computation. According to Victoria Popic et al [61] a possible definition of concurrency in computer science is a property of a system in which several computational processes or tasks are executing at the same time, possibly interacting with each other. These tasks may be implemented as separate programs or threads within a single program. In this paper the term task is used instead of the more classical term process to denote the computational process which takes place. This is done to clearly distinguish it from the term processor which denotes a CPU core. On modern computers all tasks may execute in parallel (true parallelism), however it is most often the case that the number of processors is less than the number of tasks. As a result of this, parallelism is often obtained by the method of time-slicing, where the operating system controls the scheduling of tasks between the available processors. A consequence of this is that the actual execution time of a single task is unknown. As stated, tasks in a concurrent system can interact with each other while they are executing. This, combined with the time-slicing method, results in highly unpredictable order of execution between the different tasks, and it is often up to the developer to use different techniques to control this execution order. The interaction between tasks is described more formally later in this section. In general when two or more tasks are interacting with each other, they are said to be “communicating”. Communication is however a very vague term and in concurrency theory two different communication strategies have been defined more formally

4.1 RESULTS

Table 4.1: 10 worst-case bound algorithms

S.No.	Algorithm	Discoverer(s)	Running Time
1.	Dinic's algorithm	Dinic [1970]	$O(n^2m)$
2.	Karzanov's algorithm	Karzanov [1974]	$O(n^3)$
3.	Shortest augmenting path algorithm	Ahuja and Orlin [1991]	$O(n^2m)$
4.	Capacity scaling algorithm	Gabow [1985] and Ahuja and Orlin [1991]	$O(nm \log U)$
Preflow-push algorithms			
5.	Highest-label algorithm	Goldberg and Tarjan [1986]	$O(n^2m^{1/2})$
6.	FIFO algorithm	Goldberg and Tarjan [1986]	$O(n^3)$
7.	Lowest-label algorithm	Goldberg and Tarjan [1986]	$O(n^2m)$
Excess-scaling algorithms			
8.	Original excess-scaling	Ahuja and Orlin [1989]	$O(nm + n^2 \log U)$
9.	Stack-scaling algorithm	Ahuja, Orlin and Tarjan [1989]	$O\left(nm + \frac{n^2 \log U}{\log \log U}\right)$
10.	Wave-scaling algorithm	Ahuja, Orlin and Tarjan [1989]	$O(nm + n^2 \sqrt{\log U})$

Source: Laboratory for Computer Science, MIT, 1995

Table 4.2: CPU time taken(in seconds on convex)

n	d	Shortest Aug. Path	Capacity Scaling	Dinic	PREFLOW-PUSH			EXCESS SCALING			Karzanov
					Highest Label	FIFO	Lowest Label	Excess Scaling	Stack Scaling	Wave Scaling	
500	5	0.41	1.71	0.39	0.11	0.15	0.27	0.21	0.21	0.23	0.33
1000	5	1.25	4.81	1.27	0.28	0.38	0.82	0.54	0.54	0.58	1.02
2000	5	3.84	15.17	3.97	0.76	1.12	2.62	1.54	1.47	1.68	3.18
3000	5	7.80	33.39	7.14	1.32	1.97	5.29	2.60	2.49	2.80	5.54
4000	5	15.89	74.02	13.82	1.98	3.14	11.67	4.50	4.01	4.93	12.37
5000	5	19.74	93.14	18.30	2.89	4.31	13.20	5.69	5.30	6.24	14.33
6000	5	26.80	110.53	24.61	3.65	5.80	21.31	7.86	7.29	8.72	20.05
7000	5	33.09	137.19	31.64	4.25	6.74	26.35	9.52	8.60	10.58	25.99
8000	5	39.07	167.13	40.24	4.88	8.11	30.13	11.36	10.26	12.82	31.61
9000	5	46.81	202.26	42.18	5.55	9.53	36.83	12.91	11.81	14.40	35.85
10000	5	67.48	283.88	57.37	6.94	11.43	52.41	16.40	14.85	18.24	51.58
Mean		23.83	102.11	21.90	2.96	4.79	18.26	6.65	6.07	7.38	18.35

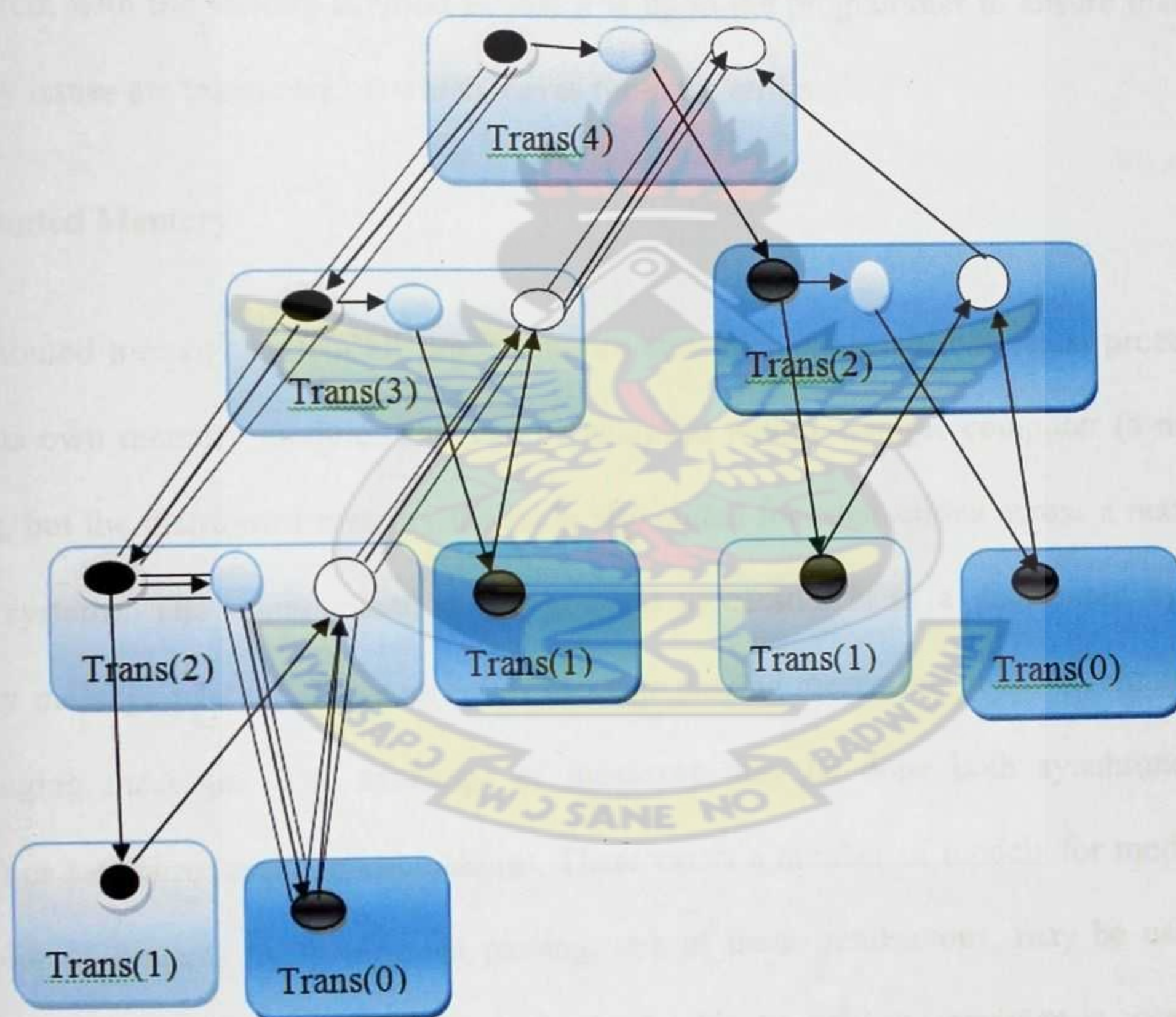
Source : : Laboratory for Computer Science, MIT, 1995

1. The preflow-push algorithms generally outperform the augmenting path algorithms and their relative performance improves as the problem size gets bigger.

2. Among the three implementations of the Goldberg-Tarjan preflow-push algorithms that was tested, the highest-label preflow-push algorithm is the fastest. In other words, among these three algorithms, the highest-label preflow-push algorithm has the best worst-case complexity while simultaneously having the best empirical performance.
3. In the worst-case, the highest-label preflow-push algorithm requires $O(n^2 \sqrt{m})$ but its empirical running time is $O(n^{1.5})$ on four of the five classes of problems that we tested.
4. All the preflow-push algorithms have a set of two "representative operations": (i) performing pushes, and (ii) relabels of the nodes. See also Ahuja and Orlin [12]. Though in the worst-case, performing the pushes is the bottleneck operation, the empirical this time is no greater than the relabel time. This observation suggests that the dynamic tree implementations of the preflow-push algorithms worsen the running time in the practice, though they improve the worst-case running time.
5. It was found out that the number of non saturating pushes is 0.8 to 6 times the number of saturating pushes.
6. The excess-scaling algorithms improve the worst-case complexity of the Goldberg-Tarjan preflow-push algorithms, but this does not lead to an improvement empirically. It was observed that the three excess scaling algorithms tested are somewhat slower than the highest-label preflow-push algorithm. The stack-scaling algorithm was detected to be the fastest of the three excess-scaling algorithms, but it is on the average twice slower than the highest-label preflow-push algorithm.
7. The running times of Dinic's algorithm and the shortest augmenting path algorithm are comparable, which is consistent with the fact that both algorithms perform the same sequence of augmentations (see Ahuja and Orlin [12]).

Though in the worst-case Dinic's algorithm and the successive shortest path algorithm perform $O(nm)$ augmentations and take $O(n^2 m)$ time, empirically we find that they perform no more than $O(n^{1.6})$ augmentations and their running times are bounded by $O(n^2)$. Dinic's and the successive shortest path algorithms have two representative operations: (i) performing augmentations whose worst-case complexity is $O(n^2 m)$; and (ii) relabeling the nodes whose worst-case complexity is $O(nm)$. It was found out that empirically the time to relabel the nodes grows faster than the time for augmentations.

Figure 4.1 A computation direct acyclic graph (TRANS N)



Source: Author (2012)

Consider the computation P-FIB.4/ in Figure 27.2, and assume that each strand takes unit time. Since the work is $T_1 = 17$ and the span is $T_\infty = 8$, the parallelism is $T_1 / T_\infty = 17/8 = 2.125$. Consequently, achieving much more than double the speedup is impossible, no matter how many processors we employ to execute the computation.

4.2 Shared Memory

Which this paper uses is a memory system, whereby the different processors and memory modules are interconnected, and tasks can therefore share the same memory locations. The communication between tasks is carried out by altering some shared variable which then is visible to other tasks. Because different tasks, or threads as they are most often called in this context, are sharing the same memory locations, consistency issues can occur. These issues can be handled by utilizing some kind of locking mechanism (synchronization primitives), which controls the access to the memory locations and preserved the program invariants. The main problem with the locking solution is, that it is up to the programmer to ensure that the consistency issues are taken care of, which leaves room for errors.

4.3 Distributed Memory

In a distributed memory system all processors are still interconnected, but each processor now has its own memory module. This can be achieved within a single computer (a multi-computer), but the distributed memory model is also suited for connections across a network (network system). The communication between the processors is in a distributed system handled by message parsing. The concept of message parsing means that tasks communicate by exchanging messages. The exchange of messages may be done both synchronously (blocking) or asynchronously (non-blocking). There exists a number of models for modeling the behavior of systems using message parsing, one of them ,rendezvous, may be used to model blocking implementations, in which the sender blocks until the message is received. Message parsing systems are often easier to reason about than shared memory systems, and they are often very scalable in size. The downside of distributes memory systems is, that they have a larger communications overhead than shared memory systems and thereby doesn't

utilize the processing power as optimal. By parring safety and liveness properties it can be implied that a program "eventually does something good".

To model and prove properties of concurrent programs, different models have been developed. In this section two useful models for modeling concurrent behavior is introduced, namely Petri Nets and the interleaving model. These models also serve as useful models for introducing several of the most important terms of concurrent programs. Of course several other models exists, but they are out of the scope of this paper.

4.4 Petri nets model

Formally a Petri Net is a bipartite, directed graph that can be used to describe the dynamic behavior of a system, especially the concurrency and synchronization aspects according to Hans [60]. A Petri Net graph consists of two kinds of nodes/vertices. Places which represents states of a system and transitions which represents activities in the system. Places are drawn as circles and transitions are drawn as bars or boxes. Places and transitions are connected via arcs that indicate the dependencies between states and activities. Furthermore, places in a Petri Net may contain zero or more tokens, and a distribution of tokens across a Petri Net is called a marking.

As stated, A Petri Net can describe how a system behaves dynamically over time. This is done by showing how the different states and activities of a system influence each other. The change of state is modeled by letting an initial marking evolve through "firing" of transitions, meaning that a transition consumes a specified number of tokens from its input places (one token per incoming arc), performs some processing task, and produces a specified number of tokens into each of the output places (one token per outgoing arc). A transition is set to be enabled, if each of its input places contains at least one token, and a firing of a transition, which is performed in a single, non pre-emptible step, can only take place if the transition is

enabled. Enabled transitions can fire at any time, and happens in a non-deterministic manner, meaning that multiple transitions may fire simultaneously exactly like the execution of concurrent threads. Mutual exclusion is when certain actions (or sequences of actions) are not allowed to be executed at the same time. Usually this is to prevent the two or more threads access the same resource or data-structure at a time. Mutual exclusion is easily modeled using Petri Nets by using an auxiliary place with a single token that represents the "right" to execute. By letting the transitions, which should not be executed at the same time, claim this single token, the firing rules prevents that more than one of the transitions to be fired at the same time. The auxiliary place with a single token is in most programming languages known as a lock. The lock is acquired before critical code is executed and released when done.

4.5 The Interleaving Model

If one is not interested in the properties related to whether actions are actually performed in parallel, but solely in the properties related to the sequence of states a task will go through, one may use the inter-leaving model of the task behavior. For any given task the execution now can be modeled by a finite or infinite sequence of the form: $s_0 \rightarrow a_0, s_1 \rightarrow a_1, s_2 \rightarrow a_2 \dots$ where s_0 is the initial state of the task and the a_i 's are actions/transitions. The execution of action a_0 will change the state of the task from s_0 to s_1 , etc. If the actions of a program always executes task is said to be sequential. In the interleaving model, a concurrent program is composed of two or more sequential tasks overlapping in time. Because of the overlap in time we need to introduce the concept of atomic actions. An atomic action is an indivisible action, meaning that no other tasks are able to detect. With the introduction of the atomic action, the question is which executions are possible for a program. The execution speed of each action and thereby also each task is unknown, meaning that the execution of a program is given by all possible interleaving of all possible sequences of actions of the tasks.

4.5 The dynamic multithreading

From Thomas H. Cormen et al [5.] One important class of concurrency platform is dynamic multithreading, which is the model we shall adopt in the computation dag. Dynamic multithreading allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, and other vagaries of static-thread programming. The concurrency platform contains a scheduler, which load-balances the computation automatically, thereby greatly simplifying the programmer's chore. Although the functionality of dynamic-multithreading environments is still evolving, almost all support two features: nested parallelism and parallel loops. Nested parallelism allows a subroutine to be "spawned," allowing the caller to proceed while the spawned subroutine is computing its result. A parallel loop is like an ordinary for loop, except that the iterations of the loop can execute concurrently. These two features form the basis of the model for dynamic multithreading that we shall study in this chapter. A key aspect of this model is that the programmer needs to specify only the logical parallelism within a computation, and the threads within the underlying concurrency platform schedule and load-balance the computation among themselves.

This model for dynamic multithreading offers several important advantages:

- ❖ It is a simple extension of our serial programming model. We can describe a multithreaded algorithm by adding to our pseudocode just three "concurrency" keywords: parallel, spawn, and sync. Moreover, if we delete these concurrency keywords from the multithreaded pseudocode, the resulting text is serial pseudocode for the same problem, which we call the "serialization" of the multithreaded algorithm.
- ❖ It provides a theoretically clean way to quantify parallelism based on the notions of "work" and "span."

- ❖ Many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm. Moreover, just as serial divide-and conquer algorithms lend themselves to analysis by solving recurrences, so do multithreaded algorithms.
- ❖ The model is faithful to how parallel-computing practice is evolving. A growing number of concurrency platforms support one variant or another of dynamic multithreading.

4.6 Multithreading Computation

We will consider dynamic multithreading computation by agreeing with Thomas H. Cormen et al [5]. In this paper by explaining our target multiprocessor platform., let us consider the execution of multithreaded algorithms on an ideal parallel computer, which consists of a set of processors and a sequentially consistent shared memory. Sequential consistency means that the shared memory, which may in reality be performing many loads and stores from the processors at the same time, produces the same results as if at each step, exactly one instruction from one of the processors is executed. That is, the memory behaves as if the instructions were executed sequentially according to some global linear order that preserves the individual orders in which each processor issues its own instructions. For dynamic multithreaded computations, which are scheduled onto processors automatically by the concurrency platform, the shared memory behaves as if the multithreaded computation's instructions were interleaved to produce a linear order that preserves the partial order of the computation dag. Depending on scheduling, the ordering could differ from one run of the program to another, but the behavior of any execution can be understood by assuming that the instructions are executed in some linear order consistent with the computation dag. In addition to making assumptions about semantics, the ideal-parallel-computer model makes some

performance assumptions. Specifically, it assumes that each processor in the machine has equal computing power, and it ignores the cost of scheduling.

Although this last assumption may sound optimistic, it turns out that for algorithms with sufficient “parallelism” (a term we shall define precisely in a moment), the overhead of scheduling is generally minimal in practice.

4.7 Scheduling

The actual running time of a multithreaded computation depends not only on its work and its span, but also on how many processors are available and how the scheduler allocates strands to processors. Good performance depends on more than just minimizing the work and span. The strands must also be scheduled efficiently onto the processors of the parallel machine. Our multithreaded programming model provides no way to specify which strands to execute on which processors. Instead, we rely on the concurrency platform’s scheduler to map the dynamically unfolding computation to individual processors. In practice, the scheduler maps the strands to static threads, and the operating system schedules the threads on the processors themselves, but this extra level of indirection is unnecessary for our understanding of scheduling. We can just imagine that the concurrency platform’s scheduler maps strands to processors directly. A multithreaded scheduler must schedule the computation with no advance knowledge of when strands will be spawned or when they will complete it must operate on-line. Moreover, a good scheduler operates in a distributed fashion, where the threads implementing the scheduler cooperate to load-balance the computation. Provably good on-line, distributed schedulers exist, but analyzing them is complicated.

The Transit algorithm which is this paper and is made up of the combination of the push reliable method and the computation dag is implemented using the lock free multithreaded method. The target multiprocessor platform for the Lock Free multithreaded algorithm for the

maximum flow problem which has been explained component by component from the beginning of this chapter is presented as a whole below.

4.8 The Target Multiprocessor platform

The target multiprocessor platform for this paper as espoused by Bo Hong [58] consists of multiple processor that access a shared memory. We assume that the architecture supports sequential consistency and atomic 'read-modify-write' instructions, as most modern parallel architectures do. A system provides sequential consistency if every node (processor cores in a multi-core architecture) of the system sees the memory accesses in the same order, although the order may be different from the order as defined by real time (as observed by hypothetical external observer or global clock) of issuing the operations. Atomic 'read-modify-write' instructions allow the architecture to sequentialize such instructions automatically.

For example, suppose $x \leftarrow x+d_1$ and $x \leftarrow x+d_2$ are executed by two processors simultaneously, the architecture will atomically complete one instruction at a time, thus the final value of x will be the accumulation of d_1 and d_2 . 'Read-modify-write' instructions can be used to implement locks as implemented in many actual architectures. The difference is that a 'read-modify-write' instruction protects individual accesses to a memory location while a lock can be used to protect a sequence of accesses. Locks are much more expensive as it has to implement mechanisms to suspend a processor/thread in case the lock is unavailable. Our algorithm is specially designed to take advantages of the 'read-modify-write' instructions and thus avoiding lock usages. We limit shared variable accesses to 'read-add-write' and read, thus the accesses can be executed atomically by the architecture. More importantly, the specially designed push and relabel operations do not need to be executed atomically, even though each one of them consists of a sequence of accesses to shared variables

4.9 Implementation

4.9.1 The Push Relabel Algorithm By Goldberg

Implementation, programming, and sometimes development are used for coding according to Eric Braude in [55]. This section discusses The implementation of the Transit algorithm using the lock free algorithm method and it is not intended to be a parallel programming per se but in the context of parallel computation.

The algorithm listed below is based on the generic push relabel algorithm by Goldberg []. Before stating the algorithm we briefly restate the notations used in the lock-free algorithm. Given a direct graph $G(V,E)$, function f is called a flow if it satisfies the three constraints above. Given $G(V,E)$ and flow f , the residual capacity $c_f(u, v)$ is given by $c_{uv} - f(u, v)$, and the residual network of G induced by f is $G_f(V, E_f)$, where $E_f = \{(u, v) | u \in V, v \in V, c_f(u, v) > 0\}$. Thus $(u, v) \in E_f, c_f(u, v) > 0$ For each node $u \in G$, $e(u)$ is defined as $e(u) = \sum_{w \in V} f(w, u)$, which is the net flow into node u . Constraint 3 in the problem statement requires $e(u) = 0$ for $u \in V - \{s, t\}$. But the intermediate result before an algorithm terminates may have non-zero $e(u)$'s. We say vertex $u \in V - \{s, t\}$ is overflowing if $e(u) > 0$. An integer valued height function $h(u)$ is also defined for every node $u \in V$. We say u is higher than v if $h(u) > h(v)$. The algorithm, listed below, is based on the push and relabel algorithm by Goldberg [20].

1. Initialize $h(u)$, $e(u)$, and $f(u, v)$
2. While there exist one or more applicable push or lift operations execute the applicable operations asynchronously where the operations of initialize, push, and lift are defined as follows:

- Initialize $h(u)$, $e(u)$, and $f(u, v)$:

$$h(s) \leftarrow |V|$$

for each $u \in V - \{s\}$

$$h(u) \leftarrow 0$$

for each $(u, v) \in E$

$$f(u, v) \leftarrow 0$$

$$f(v, u) \leftarrow 0$$

for each $(s, u) \in E$

$$f(s, u) \leftarrow cs_u$$

$$f(u, s) \leftarrow -f(s, u)$$

$$e(u) \leftarrow cs_u$$

- Push(u, \hat{v}): applies if u is overflowing, and $\exists v \in V$ s.t. $(u, v) \in E_f$ and $h(u) > h(v)$,

$$\hat{v} \leftarrow \operatorname{argmin}_v [h(v) \mid cf(u, v) > 0 \text{ and } h(u) > h(v)]$$

$$d \leftarrow \min(e(u), cf(u, \hat{v}))$$

$$f(u, \hat{v}) \leftarrow f(u, \hat{v}) + d$$

$$f(\hat{v}, u) \leftarrow f(\hat{v}, u) - d$$

$$e(u) \leftarrow e(u) - d$$

$$e(\hat{v}) \leftarrow e(\hat{v}) + d$$

- Lift(u): applies if u is overflowing, and $h(u) \geq h(v)$ for all $(u, v) \in E_f$,

$$h(u) \leftarrow \min \{h(v) \mid cf(u, v) > 0\} + 1$$

The push operation is performed on an active node u , for which there exists an outgoing residual edge $(u, v) \in E_f$ and the node u satisfies the height constraint: $h(u) = h(v) + 1$. If the node u is active and every edge $(u, v) \in E_f$ does not satisfy this constraint then the relabel operation is performed. It can be shown that the generic push-relabel algorithm is correct,

terminates and its running time is $O(V^2E)$. In 2008 Hong [58] presented a lock-free multi-threaded algorithm for the max flow problem based on Goldberg's version [10] of the push-relabel algorithm. Implementation of Hong's algorithm requires a multi-threaded architecture that supports read-modify-write atomic operations. Without loss of generality we assume that the number of running threads is $|V|$ and each of them handles exactly one node of the graph, including all push and relabel operations on it. In several, a few nodes can be handled by one thread. Let u be the running thread representing the node $u \in V$. In Hong's algorithm each of the running threads has the following private attributes. The variable e stores the excess of the node u . The variable h' stores the height of the currently considered neighbour v of u such that $(u,v) \in E_f$. The variable \hat{h} stores the height of the lowest neighbour v of u .

Other variables are shared between all the running threads. Among them there are the arrays with excesses and heights of nodes, and residual capacities of edges. First, the Init operation is performed by the master thread. This init code is the same as its counterpart in the sequential push-relabel version. Next, the master thread starts the threads executing in parallel the lock-free push-relabel algorithm.

4.9.2 Implementation of the transit algorithm using Bo Hong's lock-free multithreaded method

In the trans algorithm we maintain a linked list L consisting of all vertices in $V - \{s,t\}$. The vertices are topologically sorted according to the admissible network. The pseudocode assumes that the neighbor list have already been created for each vertex u . It also assumes that a pointer points to the vertex that follows u and is null if u is the last vertex in the list. In this algorithm as is in Bo Hong's algorithm computation starts from the lowest vertex from u which is the major change from that of Goldberg. Simply because from the transit model the system is to capture once the computations from the country of origin to the country of destination and this is done recursively.

Table 4.3: VARIABLE ACCESS CHARACTERISTICS

Shared variables	Private Variables	Written by threads	Read by threads
$h(u)$		u	u and w where $(w,u) \in E_f$
$e(u)$		u or w where $(u,w) \in E_f$	u
$c_f(u)$		u or v	u and v
$L(u)$		u	u and v
$Trans(u)$		u	u and v
	$\acute{e}, \hat{h}, v^{\wedge}, h', d, \acute{u}, \hat{u}$	per thread	per thread

Source: Author (2012)

The algorithm leads to the following lock-free programming implementation where $\acute{e}, \hat{v}, \hat{h}$, and h', \acute{u}, \hat{u} are per thread private variables and $h(u), e(u)$, and $c_f(u, v)$ ($u \in V, (u, v) \in E_f$), $L(u), Trans(u)$ are shared among all threads. The sharing characteristics of the variables are listed in Table 4.3. For programming convenience, the implementation maintains $c_f(u, v)$ rather than $f(u, v)$. The constraint $f(u, v) \leq c_{uv}$ in the problem statement translates to $c_f(u, v) \geq 0$. Also, $c_f(u, v) > 0, (u, v) \in E_f$. Upon termination of the algorithm, the flow $f(u, v)$ along each edge $(u,v) \in E$ can be derived easily from $c_f(u, v)$ since $c_f(u, v) = c_{uv} - f(u, v)$. Initially, only the master thread is running.

Before stating the algorithm we briefly restate the notations used in the Transit algorithm. Given a direct graph $G(V,E)$, function f is called a flow if it satisfies the three constraints above. Given $G(V,E)$ and flow f , the residual capacity $c_f(u, v)$ is given by $c_{uv} - f(u, v)$, and the residual network of G induced by f is $G_f(V,E_f)$, where $E_f = \{(u, v) | u \in V, v \in V, c_f(u, v) > 0\}$. Thus $(u, v) \in E_f, c_f(u, v) > 0$ For each node $u \in G$, $e(u)$ is defined as $e(u) = \sum_{w \in V} f(w, u)$, which is the net flow into node u . Constraint 3 in the problem statement requires $e(u) = 0$ for $u \in V - \{s, t\}$. But the intermediate result before an algorithm terminates may have non-zero $e(u)$'s. We say vertex $u \in V - \{s, t\}$ is overflowing if $e(u) > 0$. An integer valued height

function $h(u)$ is also defined for every node $u \in V$. We say u is higher than v if $h(u) > h(v)$. The algorithm, listed below, is based on the push and relabel algorithm by Goldberg [10], and the lock-free algorithm of Bo Hong [58]

The destination for pushes is to the neighbor list of each vertex until excess flow is zero, and is essential for the correctness of the Trans algorithm where the push and lift operations are executed asynchronously, as will be presented in the next two sections. The algorithm can be easily multi-threaded by assigning each thread T_i a distinct subset of the vertices V_i (s.t. $V_i \cap V_j = \emptyset$; if $i \neq j$, and $\bigcup_i V_i = V$). The initialization step is performed by the main thread before spawning all the multiple threads. After the initialization step, each thread T_i checks whether any push or lift operations can be applied to any of the vertices in V_i , and then the computations at the vertex is executed before the applicable operations if there exist any. When implementing the algorithm on a real computer, it is reasonable to have the same number of threads as the number of processor cores. Additionally, it is desirable to have balanced load across the threads, letting each thread execute (close-to) the same number of operations. Load balance is determined by the assignment of vertices to the threads (of course, also by the topology of the input graph). Because the focus of this paper is on the lock-free property of the multi-threaded algorithm, we leave the optimal vertex assignment problem for future as it is another open research problem by itself. Without loss of generality, we assume that for each vertex $u \in V$ there is one thread responsible for executing push (u, \hat{v}) and lift (u) . In the following analysis, we will use u to denote both vertex u and the thread responsible for vertex u , which can be easily clarified given the context.

4.9.3 The Transit Algorithm

Initially, only the master thread is running.

1. The master thread initializes $h(u)$, $e(u)$, and $c_f(u, v)$

$h(s) \leftarrow |V|$

$e(s) \leftarrow 0$

for all $u \in V - \{s\}$ do

$h(u) \leftarrow 0$

$e(u) \leftarrow 0$

for each $(u, v) \in E_f$

$c_f(u, v) \leftarrow c_{uv}$

$c_f(v, u) \leftarrow c_{vu}$

for each $(s, u) \in E_f$

$c_f(s, u) \leftarrow 0$

$c_f(u, s) \leftarrow c_{us} + c_{su}$

$e(u) \leftarrow c_{su}$

2. The master thread creates one thread for each vertex $u \in V - \{s, t\}$, and then terminates itself.
3. Each of the newly created thread u executes the following code: (lines 4-33)
4. While $u > 0$
5. do
6. Trans (u)
7. $i \leftarrow u$
8. $\hat{u} \leftarrow \text{spawn comp}(i-1)$ /* The procedure instance, the parent-may continue to execute in parallel with the spawned subroutine –its child */
9. $\hat{u} \leftarrow \text{comp}(i-2)$
10. // Sync /* ~~sync is not~~ explicit as shared variable updates in $L(u)$ and

Trans(u) are all in the form $x \leftarrow x + d$ so due to the

support of atomic read-modify-write instructions they

can be executed correctly by the architecture without any

locks */

11. $L(u) \leftarrow [\hat{u}, \hat{u}]$

12. Return $L(u)$

13. if $e(u) > 0$ and $h(u) > |v|$ then

14. $\acute{e} \leftarrow e(u)$

/* search for u 's lowest neighbor in E_f */

15. $\hat{h} \leftarrow \infty$

16. for each $(u, v) \in E_f$ do /* i.e. for each $c_f(u, v) > 0$ */

17. $h' \leftarrow h(v^\wedge)$

18. if $h' < h(v^\wedge)$ then

19. $v^\wedge \leftarrow v$

20. $\hat{h} \leftarrow h'$

21. endIf

22. end for /* v^\wedge is u 's lowest neighbor in E_f */

23. if $h(u) > h'$ then /* $h(u) > \min\{h(v) | (u, v) \in E_f\}$,

$\exists v \in V$ s.t. $(u, v) \in E_f$ & $h(u) > h(v)$, push is applicable */

24. $d \leftarrow \min(\acute{e}, c_f(u, v^\wedge))$

25. $c_f(u, v^\wedge) \leftarrow (c_f(u, v^\wedge) - d$

26. $c_f(v^\wedge, u) \leftarrow (c_f(v^\wedge, u) + d$

27. $e(u) \leftarrow (e(u) - d$

28. $e(v^\wedge) \leftarrow (e(v^\wedge) + d$

29. else /* $h(u) \leq \min\{h(v) | (u, v) \in E_f\}$, lift (u) is applicable */

30. $h(u) \leftarrow \hat{h} + 1$

31. endIf

32. $u \leftarrow u - 1$

33 end while

The sequential consistency property of the architecture guarantees that each thread executes its own lines(4-33) in the order specified above. Updates to shared variables $c_f(u \wedge v)$, $c_f(\wedge v, u)$, $e(u)$, and $e(\wedge v)$ (lines 24-28), and computation dag shared variable $L(u)$ (lines 6-12,32), due to the support of atomic 'read-modify-write' instructions, are executed atomically by the architecture. Other than the two execution characteristics provided by the architecture, we do not impose any order in which executions from multiple threads can or should be interleaved, as it will be left for the sequential consistency property of the architecture to decide. Shared variable updates in $\text{push}(u, v)$ are all in the form of $x \leftarrow x + d$ so they can be executed correctly by the architecture without any lock protection. Note that $h(u)$ is updated by and only by thread u during a $\text{lift}(u)$ operation and $L(u)$ is also updated by thread u during the spawning of the subroutine (u) operation. Thus even though $h(u)$ is shared (multiple threads may read its value), $h(u)$ does not need lock protection because only the single thread u needs to update it. When another thread reads $h(u)$ while it is being updated by thread u , the reader thread will get the value of $h(u)$ either before or after the update. Our algorithm does not require a strict order as to what value must be obtained by the reader thread. Now we have shown that the algorithm indeed can be implemented without using any locks. Next we will prove that despite the seemingly uncontrolled and unpredictable execution order, the algorithm still solves the maximum flow problem. In fact, letting the threads advance without lock-based synchronization is the essence of our lock-free multi-threaded algorithm.

The basic changes, introduced by Hong, deal with the selection of operation (push or relabel) that should be executed by u , and to which of the adjacent nodes v ; $(u, \wedge v) \in E_f$, a flow must be pushed. In opposite to the push operation of the generic sequence version where any node v connected by a residual edge to u such that $h(u) = h(v) + 1$ could be pushed, it selects the lowest

node among all the nodes connected by a residual edges. Next if the height of \hat{v} is less than the height of u the push operation is performed Otherwise, the relabel operation is performed, that is, the height of u is modified to $h(\hat{v}) + 1$. Note that the relabel operation need not be atomic because only the u thread can change the value of the height of u . Furthermore, all critical lines in the code where more than two threads execute the write instruction are atomic. Hence it is easy to see that the algorithm is correct in respect to read and write instructions.

4.10 Testing Of Hypothesis

4.10.1 INTRODUCTION

The testing phase consists of supplying input to the application and comparing the output with that mandated by the software Requirements Specification provided by Eric Braude in [55] Tests on parts of an application (individual methods, classes, etc.) are called unit tests: Tests of an entire application are system tests.

. Testing and Correctness: Testing is indispensable because it helps to uncover defects. Testing proves the presence of bugs, but never their absence.

. Types of Testing: The principal kinds of testing are;

- Informal developer tests that is performed by individual developers, documented informally in their notebooks.

Unit tests on parts such as methods or classes which may be formally documented Intermediate tests perform on a collections of classes, but not on the whole application System tests perform on whole application and it is thoroughly documented.

For each of these, are two ways to go about designing the tests cases (inputs), white box and black box.

Black box testing compares the output obtained with the output specified by the requirements document. The selection of black box test cases does not take into account the manner in which the application is designed.

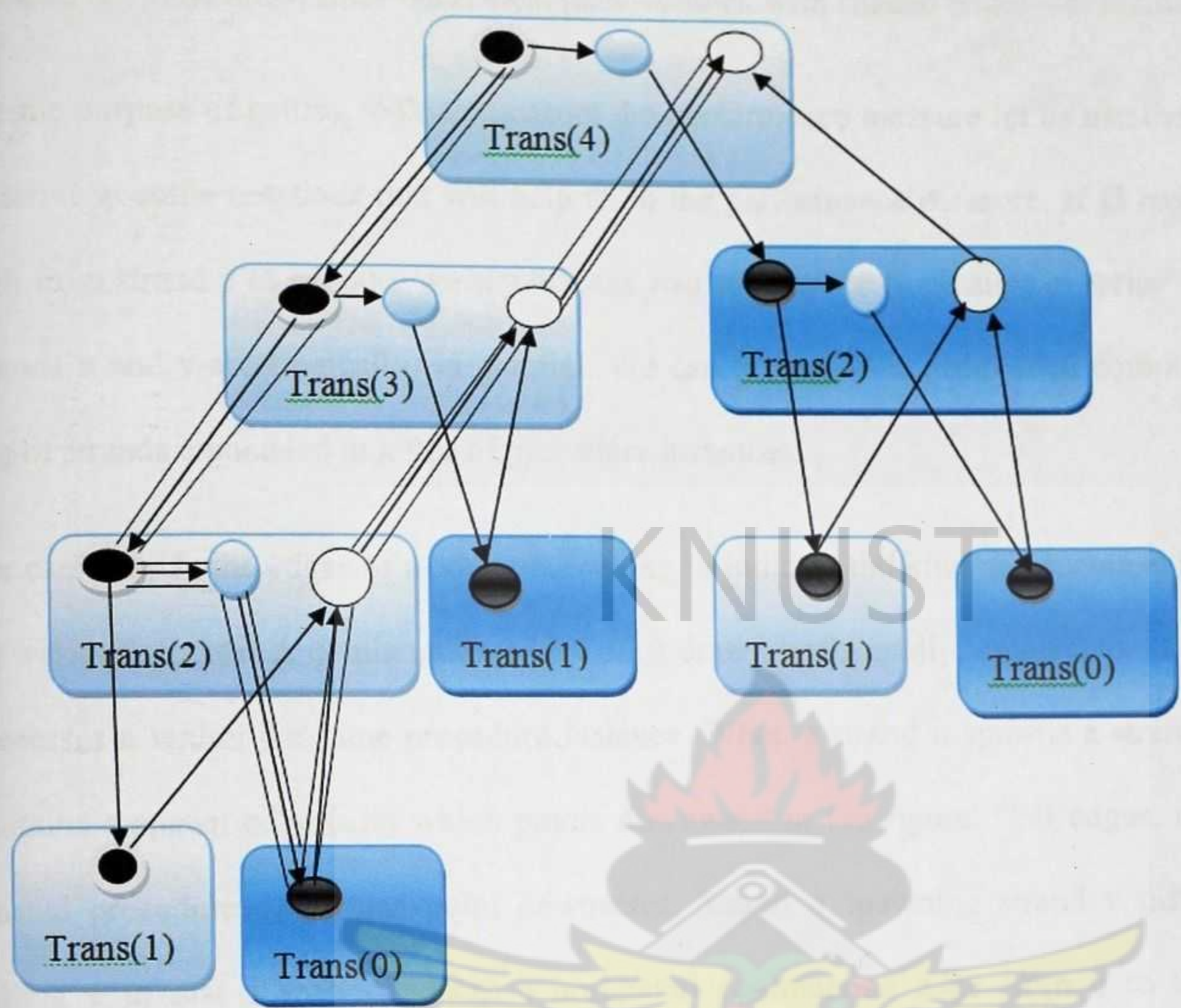
White box testing on the other hand is based on the design. White box test cases are selected to exercise specific features such as branching, loops, interfaces between modules, limits on storage, etc. We will adopt the white box testing in this paper. When developing and implementing an algorithm it can be hard to verify and prove that the algorithm performs as it should. In the case of this paper the focus is on the modeling process and performance and not necessarily on proving correctness. The push relabel algorithm has been proven to be correct in earlier studies by Goldberg [10].

4.10.2 Performance Measure

MODEL FOR MULTITHREADED EXECUTION

It helps to think of a multithreaded computation the set of runtime instructions executed by a processor on behalf of a multithreaded program—as a directed acyclic graph $G = (V, E)$ called a computation dag.

Figure 4.2: A DIRECTED COMPUTATION DAG REPRESENTING TRANS (4)



Source : Author (2012)

A directed acyclic graph representing the computation of four vertices ,Trans(4) Each circle represents one strand, with black circles representing either base cases or the part of the procedure (instance) up to the spawn of Trans(u), shaded circles representing the part of the procedure that calls Trans(u) up to the where it suspends until the spawn of Trans(u-1) returns, and white circles representing the part of the procedure after the sync where it stores comp in L(u) up to the point where it returns the result. Each group of strands belonging to the same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and heavily shaded for called procedures. Spawn edges and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming

that each strand takes unit time, the work equals 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with shaded edges—contains 8 strands.

For the purpose of getting tools to measure the performance measure let us use the figure 4.2 to arrive at some notations that will help us in the performance measure. If G has a directed path from strand u to strand v we say that the two strands are (logically) in series. Otherwise, strands u and v are (logically) in parallel. We can picture a multithreaded computation as a dag of strands embedded in a tree of procedure instances.

We can classify the edges of a computation dag to indicate the kind of dependencies between the various strands. A continuation edge (u, \acute{u}) drawn horizontally, connects a strand u to its successor \acute{u} within the same procedure instance. When a strand u spawns a strand v the dag contains a spawn edge (u, v) which points downward in the figure. Call edges, representing normal procedure calls, also point downward. Strand u spawning strand v differs from u calling v in that a spawn induces a horizontal continuation edge from u to the strand \acute{u} following u in its procedure, indicating that \acute{u} is free to execute at the same time as v , whereas a call induces no such edge. When a strand u returns to its calling procedure and x is the strand immediately following the next sync in the calling procedure, the computation dag contains return edge (u, \acute{u}) , which points upward. A computation starts with a single initial strand, the black vertex in the procedure labeled $\text{Trans}(4)$ and ends with a single final strand, the white vertex in the procedure labeled $\text{Trans}(4)$ is free to execute at the same time as v , whereas a call induces no such edge. When a strand u returns to its calling procedure and \acute{u} is the strand immediately following the next sync in the calling procedure, the computation dag contains return edge (u, \acute{u}) which points upward.

. Notations to gauge the theoretical efficiency of a multithreaded algorithm

-The Work: Of a multithreaded algorithm is the total time to execute the entire computation on one processor. That is the work is the sum of the times taken by each of the strands.

The Span is the longest time to execute the strands along any path in the dag.

The Critical path in the dag is the longest path of vertices. We can find a critical path in a dag

$G = (V, E)$ in $\Theta(V + E)$ time

Let us take as an example the serial pseudocode of the Trans computation in order to develop the performance measure.

Trans(u)

If $u \leq 1$

Return u

else $x = \text{Trans}(u-1)$

$y = \text{Trans}(u-2)$

Return $x+y$

Since the Trans procedure does not memoize, instances of the Trans procedure return the same result, that is replicates the work that the first call performs.

Let $T(n)$ denote the running time of Trans (n) Since Trans(n) contains two recursive calls plus a constant amount of extra work, then according to Brent [42] we obtain the recurrence

The running times of serial Trans(u)

$T(u) = T(u-1) + T(u-2) + \Theta(1)$. This recurrence has solution

$T(u) = \Theta(F_u)$ using substitution method .

For an inductive hypothesis, we assume that

$T(u) \leq a(F_u) - b$ where $a > 1$, and $b > 0$ m are constants, substituting we obtain

$$T(u) \leq (aF_{u-1} - b) + (aF_{u-2} - b) + \Theta(1)$$

$$= a(F_{u-1} + F_{u-2}) - 2b + \Theta(1)$$

$$\leq aF_u - b \text{ If we choose } b \text{ large enough to dominate the constant in the } \Theta(1)$$

We can then choose a large enough to satisfy the initial condition.

The analytical bound

$$T(u) = \Theta(\varphi^u) \text{ where } \varphi = (1 + \sqrt{5})/2 \quad \text{From Brent [42]}$$

4.10.3 Developing Tools To Analyze Multithreaded Algorithms

The two recursive calls $\text{Trans}(u-1)$ and $\text{Trans}(u-2)$ can run in parallel. We augment our pseudocode to indicate parallelism by adding the concurrency keywords, spawn and sync. In our case of lock free multithreading sync is not explicitly declared but implicitly the reason being that every procedure executes a sync implicitly before it returns, thus ensuring that all its children terminate before it does.

The parallel Transit algorithm

$\text{Trans}(u)$

If $u \leq 1$

Return u

Else $x = \text{spawn Trans}(u-1)$

$Y = \text{Trans}(u-2)$

//sync

Return $x+y$

The keyword spawn does not say, however, that a procedure *must* execute concurrently with its spawned children, only that it *may*. The concurrency keywords express the logical parallelism of the computation, indicating which parts of the computation may proceed in parallel. At runtime, it is up to a scheduler to determine which sub computations actually run

concurrently by assigning them to available processors as the computation unfolds. The actual running time of a multithreaded computation depends not only on its work and its span, but also on how many processors are available and how the scheduler allocates strands to process. In the development of tools to analyze multithreaded algorithms we shall investigate the execution of multithreaded algorithms on an ideal parallel computer, which consists of a set of processors and a sequentially consistent shared memory. Sequential consistency means that the shared memory, which may in reality be performing many loads and stores from the processors at the same time, produces the same results as if at each step, exactly one instruction from one of the processors is executed. That is, the memory behaves as if the instructions were executed sequentially according to some global linear order that preserves the individual orders in which each processor issues its own instructions. For dynamic multithreaded computations, which are scheduled onto processors automatically by the concurrency platform, the shared memory behaves as if the multithreaded computation's instructions were interleaved to produce a linear order that preserves the partial order of the computation dag. Depending on scheduling, the ordering could differ from one run of the program to another, but the behavior of any execution can be understood by assuming that the instructions are executed in some linear order consistent with the computation dag. In addition to making assumptions about semantics, the ideal-parallel-computer model makes some performance assumptions.

Specifically, it assumes that each processor in the machine has equal computing power, and it ignores the cost of scheduling. Although this last assumption may sound optimistic, it turns out that for algorithms with sufficient "parallelism" (a term we shall define precisely in a moment), the overhead of scheduling is generally minimal in practice.

The actual running time of a multithreaded computation depends not only on its work and its span, but also on how many processors are available and how the scheduler allocates strands

to processors. To denote the running time of a multithreaded computation on P processors, we shall subscript by P . For example, we might denote the running time of an algorithm on P processors by T_P . The work is the running time on a single processor, or T_1 . The span is the running time if we could run each strand on its own processor—in other words, if we had an unlimited number of processors—and so we denote the span by T_∞ . The work and span provide lower bounds on the running time T_P of a multithreaded computation on P processors:

. In one step, an ideal parallel computer with P processors can do at most P units of work, and

thus in T_P time, it can perform at most

PT_P work.

Total work to do is T_1 ,

we have $PT_P \geq T_1$.

Dividing by P yields the *work law*

The work law = $T_P \geq T_1/P$

.A P -processor ideal parallel computer cannot run any faster than a machine with an unlimited number of processors. Looked at another way, a machine with an unlimited number of processors can emulate a P -processor machine by using just P of its processors.

Thus, the *span law* follows

$$T_P \geq T_\infty$$

We define the *speedup* of a computation on P processors

by the ratio T_1/T_P ,

which says how many times faster the computation is on P processors than on 1 processor.

By the work law,

we have $T_P \geq T_1/P$, which implies that

$T_1/T_P = P$. Thus, the speedup on P processors can be at most P

When the speedup is linear in the number of processors,

that is, when the computation is

$$T_1/T_P = \Theta(P) \quad \text{exhibits linear speedup and}$$

when $T_1/T_P = P$, we have *perfect linear speedup*.

The ratio of the work to the span

$$T_1/T_\infty = \text{parallelism of the multithreaded computation.}$$

We can view the parallelism from three perspectives.

1. As a ratio, the parallelism denotes the average amount of work that can be performed in parallel for each step along the critical path.

2. As an upper bound, the parallelism gives the maximum possible speedup that can be achieved on any number of processors.

3. The parallelism provides a limit on the possibility of attaining perfect linear speedup.

Specifically, once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup.

To see this last point, suppose that

$$P > T_1/T_\infty, \text{ in which case the span law implies}$$

that the speedup satisfies $T_1/T_P \leq T_1/T_\infty < P$. Moreover,

if the number P of processors in the ideal parallel computer greatly exceeds the

parallelism that is, if

$$P \gg T_1/T_\infty$$

then $T_1 \ll T_P/P$, so that the speedup is

much less than the number of processors. In other words, the more processors we use beyond the parallelism, the less perfect the speedup

$T_1/T_\infty/P = T_1/(PT_\infty)$, is the parallel slackness

which is the factor by which the parallelism of the computation exceeds the number of processors in the machine.

Thus, if the slackness is less than 1, we cannot hope to achieve perfect linear speedup, because

$$T_1/(PT_\infty) < 1$$

and the span law imply that the speedup on P processors satisfies

$$T_1/T_P \leq T_1/T_\infty < P.$$

4.10.4 Greedy Scheduler

Good performance depends on more than just minimizing the work and span. The strands must also be scheduled efficiently onto the processors of the parallel machine. Instead, to keep our analysis simple, we shall investigate an on-line centralized scheduler, which knows the global state of the computation at any given time. In particular, we shall analyze greedy schedulers, which assign as many strands to processors as possible in each time step. If at least P strands are ready to execute during a time step, we say that the step is a complete step, and a greedy scheduler assigns any P of the ready strands to processors. Otherwise, fewer than P strands

are ready to execute, in which case we say that the step is an incomplete step, and the scheduler assigns each ready strand to its own processor.

From the work law, the best running time we can hope for on P processors is

$T_P = T_1/P$, and from the span law the best we can hope for is

$$T_P = T_\infty.$$

The following theorem shows that greedy scheduling is provably good in that it achieves the sum of these two lower bounds as an upper bound.

Theorem 4.0

On an ideal parallel computer with P processors, a greedy scheduler executes a multithreaded computation with

work T_1 and span T_∞ in time

$$T_P \leq T_1/P + T_\infty$$

Proof We start by considering the complete steps. In each complete step, the P processors together perform a total of P work. Suppose for the purpose of contradiction that the number of complete steps is strictly greater than T_1/P

Then, the total work of the complete steps is at least

$$\begin{aligned} P \left(\left\lceil T_1/P \right\rceil + 1 \right) &= P \left\lceil T_1/P \right\rceil + P \\ &= T_1 - (T_1 \bmod P) + P \\ &> T_1. \end{aligned}$$

Thus, we obtain the contradiction that the P processors would perform more work than the computation requires, which allows us to conclude that the number of complete steps is at most $\lceil T_1/P \rceil$. Now, consider an incomplete step. Let G be the dag representing the entire computation, and without loss of generality, assume that each strand takes unit time. (We can replace each longer strand by a chain of unit-time strands.) Let G' be the sub graph of G that has yet to be executed at the start of the incomplete step, and let G'' be the sub graph remaining to be executed after the incomplete step. A longest path in a dag must necessarily start at a vertex with in-degree 0. Since an incomplete step of a greedy scheduler executes all strands with in-degree 0 in G' , the length of a longest path in G'' must be 1 less than the length of a longest path in G' . In other words, an incomplete step decreases the span of the unexecuted dag by 1. Hence, the number of incomplete steps is at most T_1 .

Since each step is either complete or incomplete, the theorem follows.

The following corollary to Theorem 4.0 shows that a greedy scheduler always performs well.

Corollary 4.1

The running time T_P of any multithreaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors is within a factor of 2 of optimal.

Proof Let $T_{P'}$ be the running time produced by an optimal scheduler on a machine with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Since the work and span laws give us

$$T_{P'} \geq \max(T_1/P, T_\infty)$$

Theorem 4.0 implies that

$$\begin{aligned} T_P &\leq (T_1/P + T_\infty) \\ &\leq 2 \max(T_1/P, T_\infty) \\ &\leq 2T_{P'} \end{aligned}$$

The next corollary shows that, in fact, a greedy scheduler achieves near-perfect linear speedup on any multithreaded computation as the slackness grows.

Corollary 4.0

Let T_P be the running time of a multithreaded computation produced by a greedy scheduler on an ideal parallel computer with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Then, if $P \ll T_1/T_\infty$, we

have $T_P \approx T_1/P$, or equivalently, a speedup of approximately P .

Proof

If we suppose that $P \ll T_1/T_\infty$, then we also have $T_\infty \ll T_1/P$, and hence Theorem 4.0 gives us $T_P \geq T_1/P + T_\infty \approx T_1/P$. Since the work

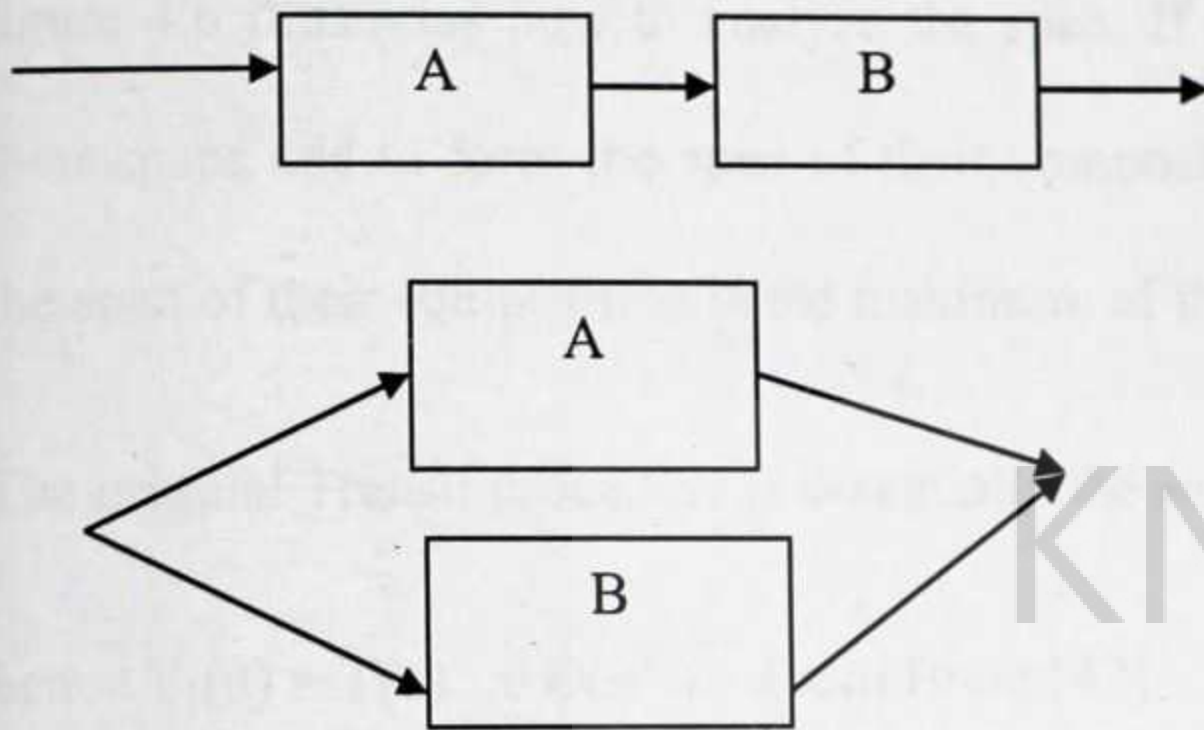
law dictates that $T_P \geq T_1/P$, we conclude that $T_P \approx T_1/P$, or equivalently,

that the speedup is $T_1/T_P \approx P$.

4.10.5 Analysing The Transit Multithreaded Algorithm

The work and span of composed sub computations

Figure 4.3: Work and span of composed sub computations



Source : Introduction to Algorithms (2008)

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a) Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

(b) Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

The work and span of composed sub computations. (a) When two sub computations are joined in series, the work of the composition is the sum of their work, and the span of the composition is the sum of their spans. (b) When two sub computations are joined in parallel, the work of the composition remains the sum of their work, but the span of the composition is only the maximum of their spans.

We now have all the tools we need to analyze multithreaded algorithms and provide good bounds on their running times on various numbers of processors. Analyzing the work is relatively straightforward, since it amounts to nothing more than analyzing the running time of an ordinary serial algorithm—namely, the serialization of the multithreaded algorithm. From figure 4.6 illustrates how to analyze the span. If two sub computations are joined in series, their spans add to form the span of their composition, whereas if they are joined in parallel, the span of their composition is the maximum of the spans of the two sub computations.

The original Transit procedure is essentially the serialization of $\text{Trans}(u)$, and

hence $T_1(u) = T(u) = \Theta(\phi^u)$ From Brent [42]

For $\text{Trans}(u)$, the spawned call to $\text{Trans}(u-1)$ in line 3 runs in parallel with the call to $\text{Trans}(u-2)$ in line 4. Hence, we can express the span of $\text{Trans}(u)$ as the recurrence

$$\begin{aligned} T_\infty(u) &= \max(T_\infty(u-1), T_\infty(u-2)) + \Theta(1) \\ &= T_\infty(u-1) + \Theta(1) \end{aligned}$$

which has solution $T_\infty(u) = \Theta(u)$

The parallelism of $\text{Trans}(u)$ is $T_1(u)/T_\infty(u) = \Theta(\phi^u/u)$, which grows dramatically

As u gets large. Thus, on even the largest parallel computers, a modest value for u suffices to achieve near perfect linear speedup for $\text{Trans}(u)$ because this procedure exhibits considerable parallel slackness. This proves the effectiveness of the computation dag aspect of the whole Transit algorithm.

4.10.6 The Complexity Bound.

In this section, because we are using the lock free method of Bo Hong,[55] the ensuing discussion is what Bo Hong did to show that the algorithm indeed terminates, and indeed it executes at most $O(|V|^2|E|)$ push/lift operations for a given graph $G(V,E)$. Note that the complexity is analyzed in the number of push and lift operations rather than in the execution time. This is because the algorithm is executed by multiple threads simultaneously. The time complexity depends on multiple factors including the number of threads and the assignment of vertices to the threads. The total number operations is therefore a more concrete measure of the complexity of the algorithm. The proof technique is similar to that used by Goldberg in [9]. We first set a bound on the height of the vertices, which is then used to bound the number of lift and push operations. The proofs of the lemmas and the theorems can be found in the work of Bo Hong's Lock free Multithreaded Algorithm for the maximum flow problem.

Theorem 1: Given graph G , if the algorithm terminates, then the calculated function f is maximum flow for G .

Lemma 1: If the algorithm terminates, then there is no path from s to t in the residual graph G_f when the algorithm terminates. Here f is the flow function calculated by the algorithm

Lemma1, says that there is no path from s to t in G_f . According to the maximum-flow minimum-cut theorem, f must be a maximum flow in G .

Lemma 2: During the execution of the algorithm, for any vertex u s.t. $e(u) > 0$, there exists a path from u to s in the residual graph G_f .

Lemma 3: Given graph G , source vertex s , and sink vertex t , then during the execution of the algorithm, if $e(u) > 0$, then there exists a path $u_1 \rightarrow u_2 \dots \rightarrow u_k$ in the residual graph from u to s ($u_1 = u, u_k = s$) and $h(u_i) \leq h(u_{i+1}) + 1$ for $i = 1, \dots, k - 1$.

Lemma 4: Given graph $G(V, E)$, source vertex s and sink vertex t , then during the execution of the algorithm, we always have $h(u) \leq 2|V| - 1$ for $u \in V$.

Lemma 5: Given graph $G(V, E)$ with source vertex s and sink vertex t , then during the execution of the algorithm, the total number of lift operations is less than $2|V|^2 - |V|$.

Lemma 6: Given graph $G(V, E)$ with source vertex s and sink vertex t , then during the execution of the algorithm, the number of saturating pushes is less than $(2|V| - 1)|E|$.

Lemma 7: Given graph $G(V, E)$ with source vertex s and sink vertex t , then during the execution of the algorithm, the number of non-saturating pushes is less than $4|V||E|$.

Theorem 2: Given graph $G(V, E)$ with source vertex s and sink vertex t , the algorithm executes $O(|V|^2|E|)$ push and lift operations

Proof immediately from lemmas 5, 6, 7

In this section, we show that the algorithm indeed terminates: it executes at most $O(|V|^2|E|)$ push/lift operations for a given graph $G(V, E)$.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATION

5.1 SUMMARY

We modeled the procedures for the movement of goods in the Ecowas sub region as a network flow problem. The directed edges in the flow network serve as a conduit for the movement of processed procedures from one vertex to another. The vertices are the conduit junctions and store no flow. As the computation of the procedures in a transit transaction could be represented as a computational directed acyclic graph, we use it to execute the procedures at the vertices, likened to the States in a transit corridor. The results of the computed procedures are recursively call upon whenever needed. We then employed the maximum network flow problem as a solution for the transit transaction for a qualitative service delivery. We then call upon the push relabel method which is one of the maximum network flow methods and specifically the generic push relable algorithm which is very efficient is used. We then combine the Computational Directed Acyclic Graph and the push relable method to get a sequential algorithm which we call the Transit algorithm that depicted the processing of procedures of the model at the vertices which represented the States in the transit corridor. We then multithreaded the resultant sequential algorithm (the Transit algorithm) by atomically performing the relable operation, the push operation and the spawn procedures concurrently, the shared memory behaves as if the multithreaded computation's instructions were interleaved to produce a linear order that preserves the partial order of the computation dag. It therefore means that a single resultant procedure is achieved with all the needed results all at once.

5.2 CONCLUSION

Our aim of achieving a single transit document at the beginning of a transit transaction has been achieved. This is because the parallel computational process we adopted in this paper abstracts the architectural model of a network flow and by the complexity of the ensuing algorithm we can safely predict that the performance of our model on a real computer would reflect the bound of our algorithm which is good. Here the performance measure is not based on theoretical cost models but on measured execution times of our algorithm. The algorithm indeed terminate, it executes at most $O(|V|^2|E|)$ push/lift operations for a given graph $G(V,E)$ and the parallelism of the Transit algorithm $\text{Trans}(u)$, which is $T(u) = \Theta(\phi^u/u)$, which grows dramatically as u gets large. Thus on the largest parallel computers, a modest value of u suffices to achieve near perfect linear speedup for $\text{Trans}(u)$ because this procedure exhibits considerable parallel slackness.

5.3 RECOMMENDATION

The model we adopted for this paper is the computational model. The parallel random access machine (PRAM) has four levels of abstraction. The remaining three of the abstractions are parallel machine, parallel architecture and parallel programming. Our model of this research serves as a blue print for developing for example a parallel programming software for the Transit algorithm, which can perform well on parallel machines as well as on parallel architecture. This research of ours, is therefore an incentive for a vast area of research given all the four levels of abstractions of the parallel random access machine.

REFERENCES

- [1] Dr. Zadok Zerelli, Mr. Olivier Hartmann, Mr. Bernard Stoven, Mr. Kristian Bernauw, Mr. Athman Mohamed Athman ALI, Mr. Azoumana Moutaye, Mr. Gerard Delanne, - West Africa Road Transport And Transit Facilitation Strategy.
- [2] World Customs Organization; TRS Methodology. Available at:
<http://www.wcomd.org/files1%20public%20files/PDFandDocuments/Procedures%20%Facilitation/TimeRelease%20StudyENG.pdf>
- [3] World bank. SSATP Discussion Paper No. 7 "Lessons to be learnt on corridor performance measurements" Available at: <http://siteresources.worldbank.org/EXTAFRSUBSAHTRA/Resources/DP07with-cover.pdf>.
- [4] World bank. : "The Cost of Being Landlocked". Available at:
<http://siteresources.worldbank.org/EXTAFRSUBSAHTRA/Resources/DP07with-cover.pdf>.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. – third edition
- [6] Lester R. Ford Jr. and D. R. Fulkerson. – Flows in networks, Princeton University Press 1962
- [7] Jack Edmonds and Richard M. Karp. Theoretical Improvements in the algorithmic efficiency for network flow problems, Journal of the ACM, 19(2). 248-264, 1972
- [8] E.A. Dinic. Algorithmic for solution of a problem of maximum flow in a network with power estimation. Soviet mathematics. Doklady, 11(5); 1277-1280, 1970
- [9] A.V. Karzanov. Determining the maximal flow in a network by the method of preflows. Soviet mathematics. Doklady , 15(2), 434-437, 1974
- [10] Andrew V. Goldberg, ~~Efficient Graph~~ Algorithms for sequential and parallel computers. Phd thesis, Department of Electrical Engineering and Computer Science MIT, 1987

- [11] Andrew V. Goldberg and Robert E. Tarjan. A New Approach to the maximum flow problem. Journal of the ACM, 35(4):921-940, 1988
- [12] Ravindra k. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem . SIAM Journal on campus, 18(5):939-954, 1989
- [13] Ravindra K. Ahuja, James B, Orlin and Robert E. Tarjan, Improved line bounds for the maximum flow problem SIAM Journal on computing, 18(6): 1057-1086, 1989
- [14] Joseph Cheriyan and S.N Maheshwari. Analysis of preflows push algorithm for maximum network flow. SIAM journal on computing, 18(6):1057-1086, 1989
- [15] Joseph Cheriyan and Torben Hagerup. A randomized maximum flow algorithm. SIAM Journal on computing. 24(2); 203-226, 1995
- [16] King Rao and Robert E, Tarjan. A faster deterministic maximum flow algorithms, 17(3);447-474, 19994
- [17] Noga Alon: Generating pseudo-random permutations and maximum flow algorithms. Information processing Letters, 35204, 1990
- [18] Stephen Phillips and Jeffrey Westbrook. Online load balancing and networkflow. In proceedings of the 25th Annual ACM symposium on Theory of computing pages 402-411, 1993
- [19] Boris V. Cherkassy, Andrew V. Goldberg. On Implementing the push -reliable method for the maximum flow problem. Stanford University, Stanford, CA, USA, Tech.Rep, 1994
- [20] A.V. Goldberg. Processor-Efficient Implementation of a Maximum Flow Algorithm . Information processing letters, pages 179-185, 1991
- [21] A.V Goldberg. A new Max-Flow. Technical MIT/LCS/TM-291, Laboratory for Computer Science, MIT, 1995
- [22] A.V Karzanov. Efficient Graph Algorithms for Sequential and Parallel Computers. PHD thesis, MIT, January

- [23] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithms – Computational Investigation. ZOR, - Methods and Models of Operational Research, 33:383 – 403, 1989
- [24] C.E. Leiserson, and B. M. Maggs, Communication- Efficient Parallel Graph Algorithms. In proc. Of International Conference on Parallel Processing pages 861-868, 1996
- [25] Y. Shiloach and U Vishkin. An $O(n^2 \log n)$. Parallel Max-Flow Algorithm. J.Algorithms 3:128-146, 1982
- [26] Z. Galil, An $O(V^{5/2}E^{2/3})$ algorithm for the maximum flow problem. Acta Informatics, 14:221 – 242. 1980
- [27] R. Anderson and J. Setubal, “On the parallel Implementation of Goldberg's “maximum flow algorithm” in 4th Annual Symposium. Parallel Algorithms and architectures (SPAA-92), San Diego, CA July 1992 pp, 168-177
- [28] D. Bader and V.SACHDEVA. “A Cache- Aware parallel implementation of the push relabel network flow algorithm and experimental evaluation of the gap relabeling heuristics,” in the 18th ISCA International Conference on Parallel and Distributed Computing (PDCS 2005), September 12-14, 2005
- [29]- W. Daniel Hillis and Jr. Guy L. Steele. Data Parallel algorithms Communications of the ACM, 29(12): 1170-1183, 1986
- [30] Richard Bellman, Dynamic Programming, Princeton University Press, 1957
- [31] Zvi Galil and Kumzoo Park. Dynamic Programming with convexity, concavity and sparsity. Theoretical Computer Science, 92(1): 49-76, 1992
- [32] Eugene L. Lawler. Combinatorial Optimization: Networks and Matroids. Holt, Rinehart and Winston, 1976
- [33] Christos H. Papadimitriou and Kenneth Steiglitz. Combinatorial Optimization. Algorithms and Complexity, Princeton Hall. 1982

- [34] Jack Edmonds , Matroids and The Greedy Algorithm. Mathematical Programming, 1(1): 127-136, 1971
- [35] Hassler Whitney: On the abstract paper properties of linear dependence. American Journal of Mathematics 57(3);509-533, 1935
- [36] F. Gavril. Algorithms for minimum coloring ,maximum clique, minimum covering by cliques and maximum independence. Set of a chordal graph. SIAM Journal on computing
- [37] Ellis Horowitz, Sartaj Sahni and Sanputhevar Rajasekaran. Computer Algorithms. Computer Science Press 1998.
- [38] Gilles Brassard and Paul Bratley, Fundamentals of algorithmics, Prentice Hall 1996
- [39] J.R. Ellis Bulldog. A compiler for VLIW Architectures. MIT Press, Cambridge, MA, USA, 1986
- [40] H. S. Stone. An efficient algorithm for the solution of a tridiagonal linear System of equations. Journal of the ACM, 20:27 – 38, 1973.
- [41] Ronald L. Graham. Bounds for certain Multiprocessor anomalies. Bell System Technical Journal, 45(9); 1563-1581, 1966
- [42] Richard P. Brent The Parallel evaluation of general arithmetic expressions. Journal of the ACM, 21(2); 201- 206, 1974.
- [43] [30] Derek L, Edger, John Zahorjan and Edward D. Lazowska, speed up versus efficiency in Parallel Systems. IEEE Transactions on Computers, 38(3); 408-423,1989
- [44] Guy E. Blelloch. Scan Primitives and Parallel Vector Models. Phd thesis, Department of Electrical Engineering and Computer Science MIT, 1989, Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-463
- [45] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM, 46(5), 720- 748, 1999

- [46] Nimar S. Arora, Robert D. Blumofe, C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 119-129 1998
- [47] Robert D. Blumofe, Christopher F. Georg, Bradley C. Kuszmanl, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk; An efficient multithreaded runtime System. Journal of Parallel and Distributed Complexity 37(1); 55 – 69 1996.
- [48] Matteo Frigo, Charles E. Leiserson and Keith H. Randall. The Implementation of the Cilk- 5. Multithreaded language . In Proceedings of the 1998 ACM SIGPLAN Conference on Programmers Language Design and Implementation pages 212-223, 1998
- [49] Cilk Arts, Inc., Burlington, Massachussets, Cilk ++ Programmer's Guide, 2008, Available at <http://www.cilk.ccom/archive/docs/cilk1guide>
- [50] Thomas Rauber, Gudula Runger. Parallel Programming for Multicore and Cluster Systems. Springer-Verlag Berlin Heidelberg 2010
- [51] T Heywood and S. Ranka. A practical hierarchical model of parallel computation. Journal of Parallel and Distributed Computing 16: 212-249, 1992
- [52] M.R. Garey and D.S Johnson, Computers and Intractability. A guide to the theory of NP-Completeness, Freeman, New York. 1979.
- [53] P. Brucker. Scheduling Algorithms. 4th edition , Springer-Verlag,, Berlin , 2004
- [54] A.V. Goldberg. An efficient Parallel Algorithm for the Single Function Coarsest Partition Problem on EREW PRAM available at: <http://>
- [55] Eric Braude. Software Design from Programming to Architecture. Boston University .Johnson Wiley and sons. 2004
- [56] Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. Pages 7-81, 2000.

- [57] Peter Wind and John Hansen Design Implementation and analysis of algorithms on multicore Systems. Technical University of Denmark, Informatic and Mathematics Modelling.
- [58] B. Hong, "A lock free multithreaded algorithm for the maximum flow problem; in IPDS, IEEE, 2008, PP 136-146
- [59] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Mathias. Provably efficient scheduling for languages with fine grained parallelism. In Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 1-12. 1995
- [60] Joshua o' Madahain, Daniel Fischer, Scott White , and Yan – Biao Boey. Java Universal Network/graph. World Wide Web electronic publication, <http://jung.sourceforge.net/>, 2006.
- [61] Victoria Popic and Javier Valez. Parallelizing the Push Relabel Max-Flow Algorithm. Available at:
http://courses.csail.mit.edu/6.884/spring10/projects/suiq_valezj_maxflowpot.pdf
- [62] Hans Henrik L_vengreen. Basic Concurrency Theory. Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Denmark, 1.1 edition, 2005.
- [63] Ananth Grama. Introduction to parallel computing. Addison-Wesley, Harlow, England, 2nd edition, 2003.

LIST OF ACRONYMS

1. ASYCUDA : Automated System For Customs Data
2. BCEOM : The Bureau Central d'Etudes pour les Equipement d'Outre-Mer
3. ECOWAS : Economic Community of West African States
4. EU : European Union
5. FAL : The Fussil Automatic Ledger
6. GAINDE : Customs Computer System used by Senegal for the clearance of goods
7. GCMS : Ghana Customs Management System
8. GPRS: General Packet Radio Service
9. IMO : International Maritime Organization
10. ISRT : Inter-State Road Transit
11. KYOTO : Kyoto protocol was adopted in Kyoto, Japan
12. MARPOL : Marine Pollution
13. RECS : Renewable Energy Certificate
14. RDBMS : Relational Database Management System
15. SSATP : Sub Saharan African Transport Policy Program
16. SDAM : Systeme de Douanement des Machandises
17. UEMOA : Union Economique et Monetaire Ouest African
18. WAEMU : West Africa Economic And Monetary Union
19. UN/EDIFACT : United Nations/ Electronic Data Interchange For Administration,
Commerce and Transport