University of Kentucky

UKnowledge

University of Kentucky Master's Theses | Graduate School

2004

GRAPHICAL MODELING AND SIMULATION OF A HYBRID HETEROGENEOUS AND DYNAMIC SINGLE-CHIP MULTIPROCESSOR ARCHITECTURE

Chunfang Zheng
University of Kentucky, czhen2@uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation
Zheng, Chunfang, "GRAPHICAL MODELING AND SIMULATION OF A HYBRID HETEROGENEOUS AND DYNAMIC SINGLE-CHIP MULTIPROCESSOR ARCHITECTURE" (2004). *University of Kentucky Master's Theses*. 249.
https://uknowledge.uky.edu/gradschool_theses/249

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

GRAPHICAL MODELING AND SIMULATION OF A HYBRID HETEROGENEOUS AND DYNAMIC SINGLE-CHIP MULTIPROCESSOR ARCHITECTURE

A single-chip, hybrid, heterogeneous, and dynamic shared memory multiprocessor architecture is being developed which may be used for real-time and non-real-time applications. This architecture can execute any application described by a dataflow (process flow) graph of any topology; it can also dynamically reconfigure its structure at the node and processor architecture levels and reallocate its resources to maximize performance and to increase reliability and fault tolerance. Dynamic change in the architecture is triggered by changes in parameters such as application input data rates, process execution times, and process request rates. The architecture is a Hybrid Data/Command Driven Architecture (HDCA). It operates as a dataflow architecture, but at the process level rather than the instruction level. This thesis focuses on the development, testing and evaluation of a new graphic software (hdca) developed to first do a static resource allocation for the architecture to meet timing requirements of an application and then hdca simulates the architecture executing the application using statically assigned resources and parameters. While simulating the architecture executing an application, the software graphically and dynamically displays parameters and mechanisms important to the architectures operation and performance. The new graphical software is able to show system and node level dynamic capability of the HDCA. The newly developed software can model a fixed or varying input data rate. The model also allows fault tolerance analysis of the architecture.

KEYWORDS: Parallel Processing, Dataflow Graph, Static Load-Balancing Algorithm, Dynamic Load-Balancing Algorithm, Graphical Simulation.

Chunfang Zheng

December 14, 2004

Copyright © Chunfang Zheng 2004

GRAPHICAL MODELING AND SIMULATION OF A HYBRID HETEROGENEOUS AND DYNAMIC SINGLE-CHIP MULTIPROCESSOR ARCHITECTURE

By

Chunfang Zheng

Dr. J. Robert Heath

(Director of Thesis)

Dr. Yuming Zhang

(Director of Graduate Studies)

December 14, 2004

(Date)

RULES FOR THE USE OF THESIS

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

THESES

Chunfang Zheng

The Graduate School

University of Kentucky

2004

GRAPHICAL MODELING AND SIMULATION OF A HYBRID HETEROGENEOUS AND DYNAMIC SINGLE-CHIP MULTIPROCESSOR ARCHITECTURE

THESES

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering in the College of Engineering
at the University of Kentucky

By

Chunfang Zheng
Lexington, Kentucky

Director: Dr. J. Robert Heath, Associate Professor of Electrical and Computer Engineering

Lexington, Kentucky

2004

Copyright © Chunfang Zheng 2004

ACKNOWLEDGEMENT

The following thesis, while primarily an individual work, benefited from the insights and direction of several people. First, I would like to express my great gratitude to my Thesis Chair, Dr. J. Robert Heath, for his continuous guidance, encouragement, patience and support. This includes not only each stage of this thesis process, but also throughout my master's study at the University of Kentucky. His faith in me and his timely instructions have made this work possible. Next, I would like to thank Dr. William (Bill) R. Dieter and Dr. Zongming Fei (from the Computer Science Department) for serving in my advisory committee. The classes that I have taken with Dr. Dieter and Dr. Fei have benefited me greatly in both this thesis and my career path. I would also like to thank my current DGS, Dr. Yuming Zhang, and former DGS, Dr. William T. Smith for their help and support during my graduate study at the University of Kentucky.

In addition to the technical and instrumental assistance above, I received equally important assistance from family and friends. My husband, Zheyang Du, has provided moral and emotional support throughout the thesis process, as well as technical assistance critical for the completion of the project in a timely manner. My lovely son, Andy, has been my constant inspiration. Without the hope he instilled in me, I would never have completed this work.

TABLE OF CONTENTS

ACKNOWLEDGEMENT iii
TABLE OF CONTENTS..... iv
LIST OF TABLES..... v
TABLE OF FIGURES..... vi
1 INTRODUCTION 1
1.1 Background..... 1
1.2 Thesis Objectives..... 2
1.3 Related Research..... 3
2 OVERVIEW OF HDCA..... 4
2.1 Dataflow Technique..... 4
2.2 Basic Structure of HDCA 6
2.3 Application Mapping and Load-Balancing Strategy of HDCA..... 8
3 QUEUING THEORY MODELING OF THE HDCA 11
3.1 Review of Queuing Theory Model 11
3.1.1 Linear Pipeline..... 14
3.1.2 Fork..... 16
3.1.3 Join..... 18
3.1.4 Feedback 20
3.2 Static Load Balancing Algorithm Analysis for Program "COPY"..... 22
4 GRAPHICAL SIMULATOR 27
4.1 Programming Environment and Language 27
4.2 Simulation Algorithm 28
4.2.1 Graphic Module (GM) 30
4.2.2 Simulation Module (SM) 34
4.2.3 Update Module..... 36
4.3 Program "hdca"..... 36
4.3.1 Graphic Module (GM) 44
4.3.2 Simulation Module (SM) 55
4.3.3 Update Module..... 61
4.3.4 Shutoff Module 63
4.4 Input and Output File Format 63
4.5 How to run HDCA? 65
5 SIMULATION RESULTS 68
5.1 Application 1..... 68
5.2 Application 2..... 116
5.3 Application 3..... 152
6 CONCLUSIONS AND FUTURE RESEARCH 191
APPENDIX A C Code for Program copy.c..... 193
APPENDIX B C Code for Program hdca.c 203
REFERENCES 224
Vita..... 226

LIST OF TABLES

Table 3-1: Sample Parameter Values for Example Dataflow Graph 24
Table 3-2: Results of the Analysis of the Example Flow Graph 26
Table 4-1: Description of the Global Arrays 38
Table 4-2: Functionalities of the function program in HDCA....................................... 39
Table 4-3: Example Run of Program "hdca" 65
Table 5-1: Parameter Values for Data Flow Graph of Application 1 70
Table 5-2: Input Files for Application 1 70
Table 5-3: Application 1 Results: 20 Tokens 112
Table 5-4: Parameter Values for Data Flow Graph of Application 2 117
Table 5-5: Application2 Results: 20 Tokens at Node11, 20 Tokens at Node 12.......... 148
Table 5-6: Parameter Values for Data Flow Graph of Application 3 153
Table 5-7: Application 3 Results: 20 Tokens at Node11 186

TABLE OF FIGURES

Figure 2.1 Typical Dataflow Graph 5
Figure 2.2 Basic Structures of Dataflow Graphs 6
Figure 2.3 Basic Structure of a DPCA 7
Figure 2.4 Block Diagram of HDCA 9
Figure 3.1 Single-Node 12
Figure 3.2 A Multi-Copy Node 14
Figure 3.3 Linear Pipeline System 15
Figure 3.4 Fork 17
Figure 3.5 Join 19
Figure 3.6 Feedback Node 21
Figure 3.7 Example of A General Dataflow Graph 23
Figure 4.1 Relationship Between COPY and HDCA Modules 29
Figure 4.2 Schematic of the Display Screen 30
Figure 4.3 An Example Dataflow Graph 33
Figure 4.4 High Level Flow Chart for Simulation Module 35
Figure 4.5 Flow Chart for Program HDCA 41
Figure 4.6 Flow Chart for “draw_dataflow” 45
Figure 4.7 Flow Chart for “draw_link” 48
Figure 4.8 Flow Chart for “draw_queue” 51
Figure 4.9 Flow Chart for Program “redraw” 54
Figure 4.10 Flow Chart for Program “simulation” 56
Figure 4.11 Flow Chart for Program “update” 62
Figure 5.1 Dataflow Graph of Application 1 69
Figure 5.2 a Simu. Results of Application 1 (Input Rate=263micro-cycles/token) 72
Figure 5.3 a Simu. Results of Application 1 (Input Rate=200micro-cycles/token) 85
Figure 5.4 a Simu. Results of Application 1 (Input Rate=100micro-cycles/token) 98
Figure 5.5 Queue Depth Plot for Application 1 113
Figure 5.6 Data Flow Graph of Case 2 116
Figure 5.7 a Simu. Results of Application 2 (Input rate = 263 micro-cycles/token) 118
Figure 5.8 a Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) 128
Figure 5.9 a Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token) 138
Figure 5.10 Queue Depth Plot for Application 2 149
Figure 5.11 Dataflow Graph of Application 3 152
Figure 5.12 Simu. Results of Application 3 (Input rate = 263 micro-cycles/token) 154
Figure 5.13 Simu. Results of Application 3 (Input rate = 200 micro-cycles/token 165
Figure 5.14 Simu. Results of Application 3 (Input rate = 100 micro-cycles/token) 176
Figure 5.15 Queue Depth Plot for Application 3 188

1 INTRODUCTION

1.1 Background

Since the beginning of the computer era, much emphasis has been placed on maximizing throughput, performance, and being able to compute a wide range of applications. Realizing that the speed of the conventional von Neumann organization was at the mercy of technology, researchers sought newer and faster architectures. Out of this need was born the distributed and/or parallel data processor, in which many identical or non-identical computing elements work in harmony to solve a single problem.

Initial distributed parallel architectures were vector processors or array processors [3]. Newer applications such as the rapid execution of massive programs encountered in high-energy nuclear physics research required much more sophisticated parallel/distributed architectures. Applications such as the processing of data from phased array radar or phased array sonar demanded still more from distributed/parallel architectures. In addition to the requirement of distribution of resources to process tremendously high data rates, systems must also sometimes cope with real-time environments and events must be triggered by input data rather than by a central scheduler. In order to meet these and other requirements, various research projects have been conducted within the Computer Architecture Laboratory by at the University of Kentucky over past years. An initially proposed distributed/parallel architecture was a Dynamic Pipeline Computer Architecture (DPCA), a very loosely coupled, highly reconfigurable, real-time dataflow machine as described in [3].

Since the completion of the first version of DPCA, many parallel computer architectures have been developed and implemented to meet current and future computer application requirements. The many computer architectures are commonly divided into three classes: multiprocessor (shared and distributed memory) architectures, distributed computer architectures, and dataflow architectures. The DPCA has since been refined and evolved to a new architecture – the Hybrid Data/Command Driven Architecture(HDCA). This architecture is a versatile, medium to coarse grain, dataflow/ Von-Neumann hybrid architecture developed to meet real-time radar, seismic, underwater sonar, and satellite applications. The architecture is a hybrid dataflow architecture since it uses conventional

Von-Neumann processors as Computing Elements (CEs). Rather than data flowing through the system to initiate processes, incoming system data is stored in shared memory and small control tokens that represent each data input flow through the system, initiating processes in correct order. Processes resident in CEs access/write data in a shared main memory through a scalable non-blocking circuit switch.

Compared to the DPCA, the HDCA moved from a loosely coupled and possibly distributed system to tightly coupled single-chip parallel architecture. The HDCA is reconfigurable at the "system" level in that it can execute a dataflow (or process flow) graph of any topology (cyclic or acyclic), with any number of inputs/outputs. It is reconfigurable at the "node" or process level in that if particular requested processes became "overly requested" as indicated by the control token queue of the processes executing, the requested process exceeding a statically determined queue threshold depth, additional process CEs containing the overly requested process(es) of the node can be dynamically activated to aid the over-queued node in processing in order to reduce the queue depth to an acceptable level. Other important features of the HDCA architecture are that it can use homogenous or heterogeneous CEs; it can be dynamic at the processor architecture level; it is scalable; CEs have parallel access to both medium-size data memories and large-file data memories; it has fault tolerant capabilities; it can utilize a distributed operating system; and it will be shown to be "hybrid," that is, it is a cross between a dataflow and a Von-Neumann architecture. The architecture of HDCA will be presented later in Chapter 2.

1.2 Thesis Objectives

The objective of this master's thesis is to develop a graphical software simulator capable of simulating the HDCA executing an application described by a dataflow or process graph. The purpose of the simulator is to enable the user to observe the movement of data or control tokens through the system, to see the dynamic configuration of the CEs, and finally to get a better intuitive and visional understanding of the HDCA system. One important feature of HDCA is that it is very sensitive to input data rate. This simulator should be able to show processor-level dynamic changes and the ebb and flow of processor queue depth (load distribution/balancing) at each node as the input rate changes.

A new simulation algorithm has been developed, and the algorithm program, “hdca,” is written in the “C” programming language. GNU’s “libplot” library was used for graphical plotting under the LINUX operating system. Chapter 4 describes the simulation algorithm and explains the various modules of the algorithm that are used in the program “hdca.” Three applications were tested by the simulator, and the simulation results are presented in Chapter 5. Chapter 6 concludes this thesis with a discussion of ongoing and future research related to the graphical simulation.

1.3 Related Research

Since the mid-sixties, large military equipment manufacturers and others have been involved in research projects in the field of computer graphics. By the 1970’s, this research had begun to bear fruit. Many obstacles had to be overcome in the early work in this exciting field, but soon computer-aided design (CAD) and flight simulators became viable products of the research efforts in computer graphics [10].

Computer graphics have been used extensively in conjunction with simulations. The pioneer software was described in [10]. Some general-purpose software was introduced in [13]. Most of this software used mathematical simulation and plotted graphs using the simulation result. They couldn’t be used to simulate dynamic dataflow computer architectures. Then, many special-purpose graphic simulators were developed, such as the graphic simulator for the CAD of parallel manipulators described in [11] and an animated graphical simulator for multiple switch architectures described in [12]. In the early nineties, Mahyar R. Malekpour developed a simulator for heterogeneous dataflow architectures [8]. This simulator is able to simulate the execution of a graph on a given system, but it needs the dataflow graph as the input, and then subtracts the information from the dataflow and does simulation. The output is a data file, not a graphic.

This thesis will present a new graphical simulator, hdca, which can simulate the execution of any application described by a dataflow graph on the dynamic HDCA graphically. By watching control token flow through the graph and the dynamic changing of the HDCA configuration, one can visually observe the execution of a computer program (application) on the HDCA. The ebb and flow of queues at each node (an indication of load distribution among processors) is especially evident as the simulation unfolds.

2 OVERVIEW OF HDCA

The Hybrid Data/Command Driven Architecture (HDCA) is an innovation in computer architecture which incorporates many highly desirable features. Among these are the ability to function in a real-time environment as a data driven machine at the process level, a high degree of fault tolerance, and dynamic reconfigurability. Many applications require that a computer analyze data as soon as it is generated by some other device. Examples of this type of application may be found anywhere that automatic monitoring devices are employed. A computer that is to be used in such an application must operate in a real-time mode. In order to prevent delays in the analysis of the data, the data itself should initiate computation. A computer system possessing this ability is said to be data driven. [2]

Parallel processing and pipelining are two major architectural techniques for improving the performance of computers. In parallel processing two or more parts of a given job are executed simultaneously in order to reduce the total time required to process the job. Pipelining is employed where large numbers of jobs that require the same sequence of processes are encountered. The HDCA is a parallel pipeline architecture, which is able to execute algorithms of any structure. The structure of most computer algorithms may be represented in the form of a dataflow graph.

2.1 Dataflow Technique

Dataflow architectures are the smallest class of the three classes of parallel architectures described in chapter one. They may exist at fine, medium, or coarse grain levels. Dataflow architectures at the fine grain level generally operate by having basic single arithmetic/logic operations executed upon the availability of required single element data variables and they do not contain program counters at the instruction level. Dataflow architectures at the medium and coarse grain level may contain normal Von-Neumann processors using program counters. At the medium and coarse grain level, individual processes resident within the Von-Neumann processors are triggered by the arrival of data (normally data sets consisting of multiple data elements) on an input queue and are executed concurrently. Because of this data-driven nature, directed graphs,

specifically dataflow graphs, are used to describe the actions of an application program that executes on a dataflow architecture system. [4]

A typical application dataflow graph is illustrated in Figure 2.1 where the nodes in the graph represent medium to coarse grained processes and the directed arcs represent the flow of data from one process to another.

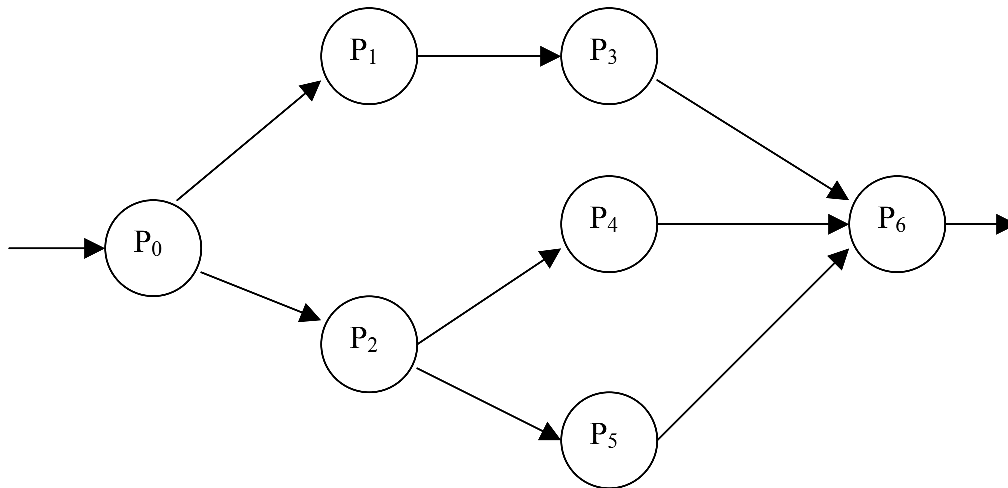


Figure 2.1 Typical Dataflow Graph

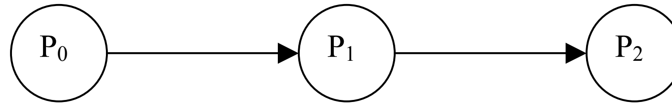
Any dataflow graph consists of three basic structures:

1) The linear pipeline (Figure 2.2a)
2) The fork (Figure 2.2b)
3) The join (Figure 2.2c)

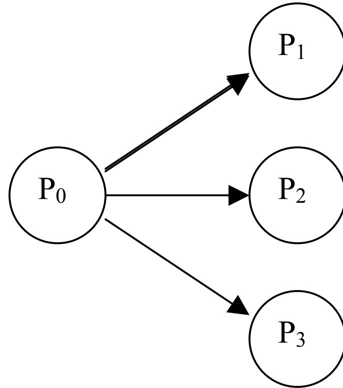
The linear pipeline accepts data/commands (a command is a control token rather than data which initiates a process) from one node while producing data/commands for only one node.

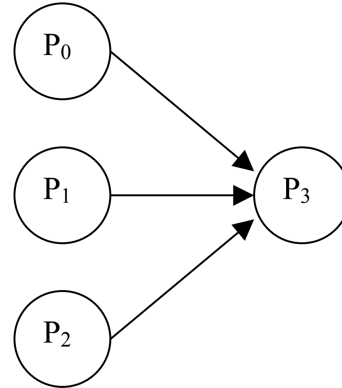
Two types of forks exist: selective and nonselective. The selective fork produces data/commands along a single output path, at any given time, while the nonselective fork produces data/commands along all output paths.

Likewise, two types of joins exist: selective and nonselective. The selective join accepts data/commands from only one input path at a time while the nonselective join accepts data/commands only when all paths contain data/commands for input.



a. Linear Pipeline





b.) Fork

c.) Join

Figure 2.2 Basic Structures of Dataflow Graphs

2.2 Basic Structure of HDCA

The HDCA evolved over time from the distributed computer architecture of DPCA. At a high level, the DPCA consists of a number of identical, general purpose computing elements (CEs), which are connected to memory buffers through a system of circuit switches as shown in Figure 2.3. The CEs are the fundamental building blocks from which the various configurations that the system may assume are formed. Depending upon the specific application, a CE may range from a small microcomputer type processor, which can store a short set of instructions, to a powerful superscalar type processor with many megabytes of program and data memory.

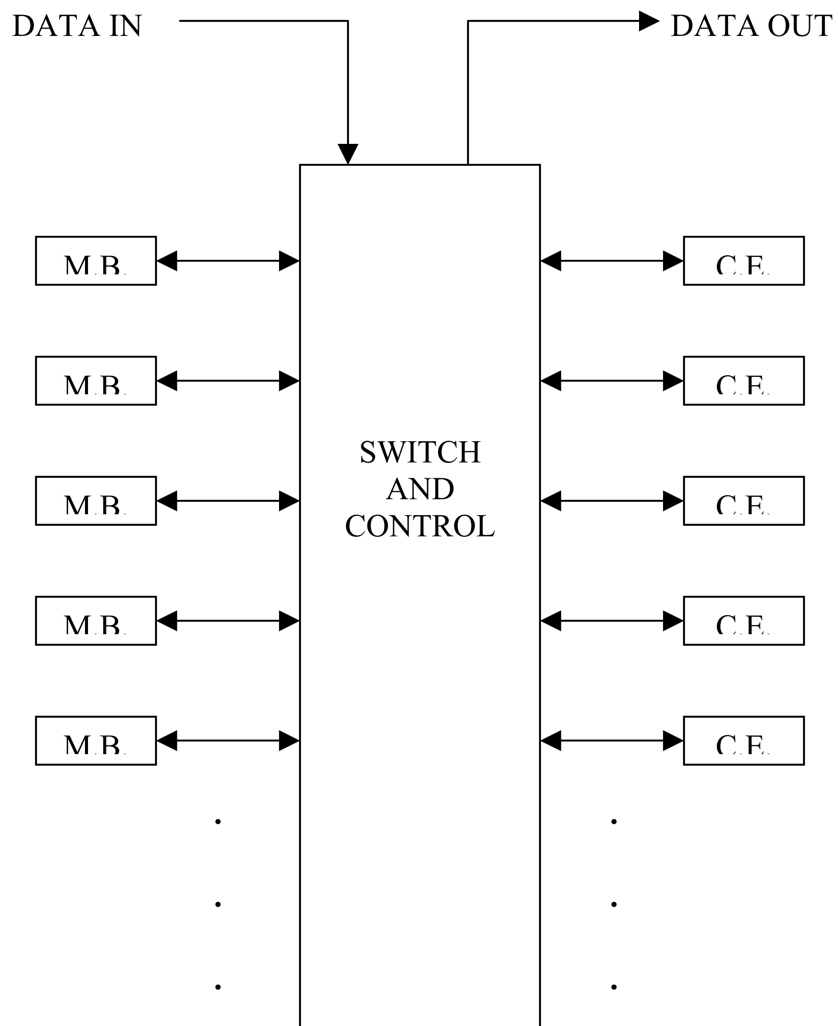


Figure 2.3 Basic Structure of a DPCA

With current day VLSI technology, it is now possible to put all the CEs and other circuits in just one IC chip. Thus HDCA has evolved from a loosely coupled and possibly distributed system to a more tightly coupled single-chip architecture. Figure 2.4 (This figure is from [6]) is the block diagram of the HDCA. The input data is facilitated by high speed FIFOs, which may be loaded externally and unloaded by the CE, which is designated to handle the input process. The CE moves the input data from the FIFOs into the Data Memory and creates Process Request Control Tokens (PRTs) that are mapped by the process mapper (Control Token Mapper) to the initial process of an executing application.

The input process of a dataflow graph, which is being executed by the system, is treated as the beginning process. The beginning process links to the data block pointed to by the PRT and executes its algorithm on the data. Upon completion, this process will deposit its results back into the data memory, then generate a control token and send the token to the token mapper. The mapper will route this token to the queue of the next node to process the data. Data is moved through the system by the continuous routing of control tokens from one CE to another. The final output is memory-mapped through an exit port in the CE-Data Memory Circuit Switch. Since both the input and the output are accessed through this switch, one or several CEs may be designated to handle the input or output processes. Further information on the Token Controlled HDCA can be found in references [3,4.5].

2.3 Application Mapping and Load-Balancing Strategy of HDCA

Since applications are represented by dataflow graphs, a real-time dataflow architecture must be able to map the dataflow graph representation of any problem space to its system hardware in order to function properly. Two load balancing strategies are used for this mapping purpose in the HDCA system – static (prior to execution of the application) load balancing and dynamic (while the application is in execution) load balancing.

There are usually a limited number of Computing Element (CE) processors in any architecture. So an efficient static load balancing algorithm is needed to analyze algorithms in order to allocate the system's resources in the optimum configuration, map

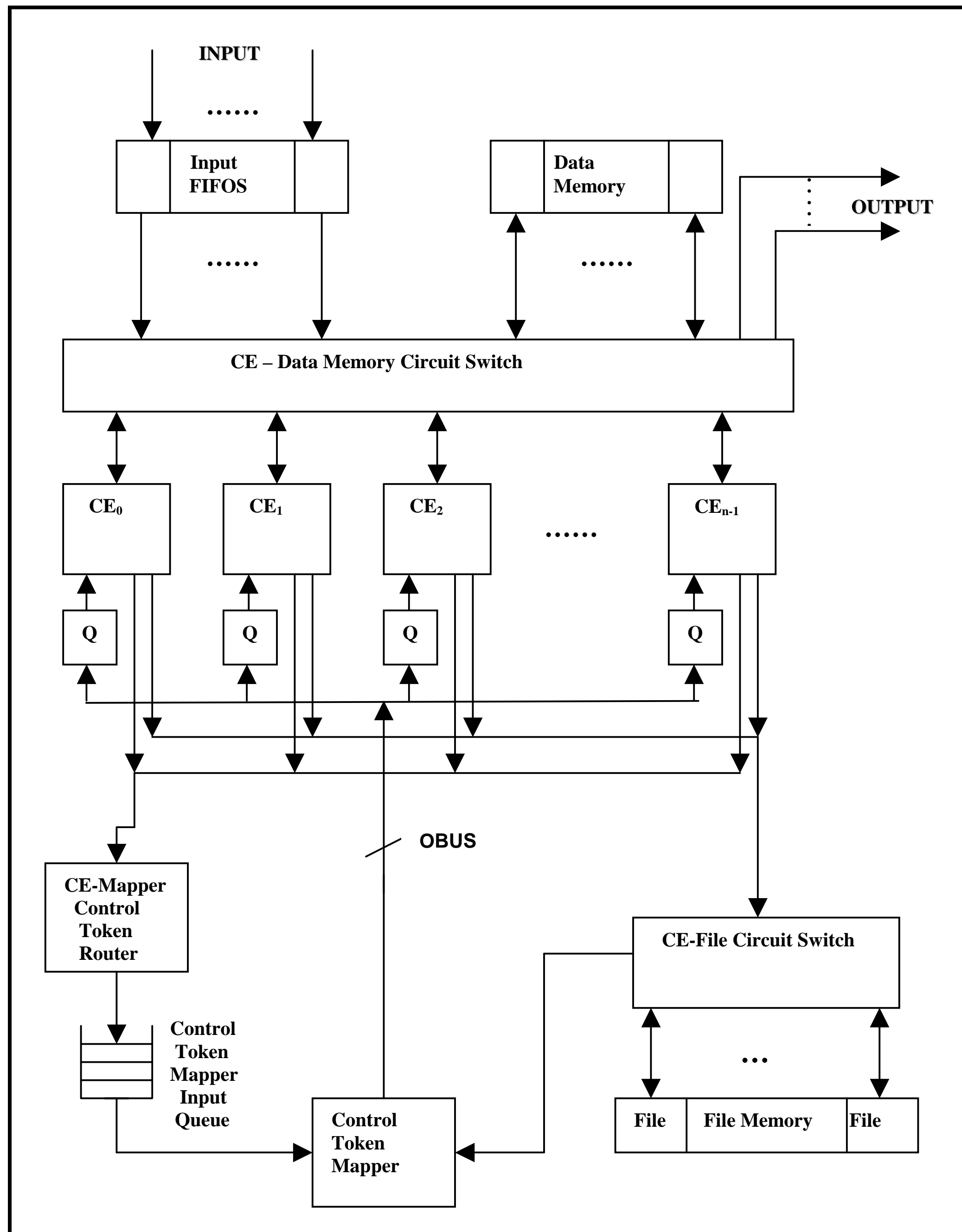


Figure 2.4 Block Diagram of HDCA

processes to a minimum number of CEs and meantime, meet the real-time timing requirements. J. Cochran has developed a static load balancing algorithm, applicable to the HDCA, in [2]. The program "COPY" analyzes a dataflow algorithm to be mapped to the HDCA and calculates the number of necessary copies of process needed to execute a given algorithm with maximum efficiency based on input data rates, process execution times, and queuing levels in the input buffers of the system processor. Static load balancing is effective as long as these parameters remain at expected levels. However, when these system parameters experience unexpected fluctuations, such as the input data rate exceeding the maximum limits, the static load balancing algorithm will become ineffective and in such cases dynamic load balancing is used to schedule the processes dynamically during run time.

Once a system starts running, dynamic load balancing is required to handle both "expected" and "unexpected" situations. The main expected situation is the case when a requested process has been mapped to and resides in several CEs. Dynamic load balancing is required to select the most appropriate CE from several containing a copy of the requested process based on a certain criteria. Unexpected situations include omission of parameters from the static load balancing algorithm, other unplanned interruptions and delays, possible impreciseness of the static load balancing algorithm at certain specific times, and when a CE fails. In short, the dynamic load balancing mechanism prevents excessive queuing of data and commands at a node during run-time and in doing such it balances the load over the entire system. The goal of the load balancing circuit within the parallel architecture is to dynamically maintain a queue level for each processor at or below a statically set queue threshold level at each processor which will allow soft system real-time constraints to be met. The dynamic load balancing function for the HDCA is performed by the "Mapper Control Token Queue" and "Control Token Mapper", which are shown within Figure 2.4.

Detailed description of the dynamic load balancing algorithm and dynamic load balancing circuits can be found in [4] and [6]. The static load balancing algorithm developed by J. Cochran can be found in [2]. Since this thesis will use the result of the program "COPY," there will be an introduction of the algorithm used by "COPY" in Chapter 3.

3 QUEUING THEORY MODELING OF THE HDCA

3.1 Review of Queuing Theory Model

The problem of providing to the operating system the static load balancing algorithm involves detailed mathematical modeling of the architecture. These models are used to determine the effect of changes in system parameters on the demand for resources and they are heavily based on queuing theory. In a dataflow graph, each node can be modeled as a buffer and a Computing Element (CE). The buffer is a storage place where the data is entered and stored for the CE to process while the CE is busy processing other data. Thus, each node is an individual queuing system and the entire system composes a queuing network.

The input data rates, the process execution times, and the queuing levels in the input buffers of the system nodes are the most important system parameters to incorporate into the model of the HDCA. The inclusion of these parameters allows for the analysis of the throughput time for each node in the system and ultimately, the determination of the number of copies of each node (process) necessary to execute a given algorithm with maximum efficiency.

The following symbols will be used throughout the discussion of modeling the various nodes, which may compose a general dataflow graph.

R_i = the number of jobs (process request tokens in the case of the HDCA) input to the buffer of a node per unit time.

R_o = the number of jobs output from the computing element per unit time.

$n(t)$ = the number of jobs in the buffer at time t.

t_q = the length of time that a job spends in the buffer awaiting execution.

t_s = the length of time that a job spends in the CE in execution.

t_t = the throughput time (data input-to-output time) for a node.

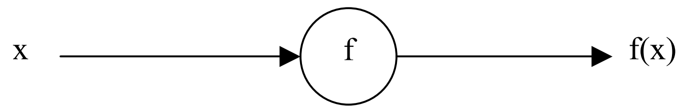
t_a = acceptable delay or throughput time for a node.

t_{sc} = service time of a clogging node.

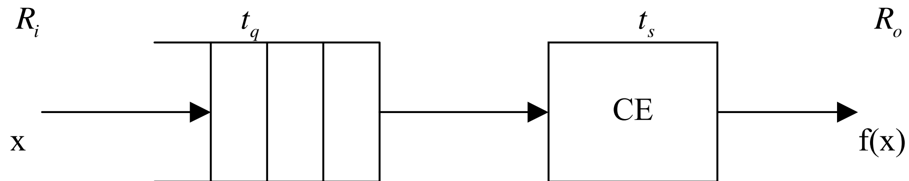
t_T = the throughput time for the system.

N = number of copies of a node required to maintain t_t within acceptable limits.

The simplest dataflow graph consists merely of an input, a functional node, and an output as illustrated in Figure 3.1 (a) and its hardware representation is shown in Figure 3.1 (b). A complex dataflow graph is composed of many nodes. In such cases, a second subscript is added to the basic symbol for the parameters as listed above. For example, R_{i2} is used to denote the input data rate to node number two, and t_{q7} represents the time that a job spent waiting in the buffer of node seven.



(a) A Single-Node Dataflow Graph



(b) Hardware Oriented Representation with Pertinent Parameters

Figure 3.1 Single-Node

From the above definitions and Figure 3.1 (b), the following relationships can be derived algebraically. Equation (1) assumes that the input buffer is never empty.

$$R_o = 1/t_s \qquad (1)$$

$$n(t) = R_i t - R_o t = (R_i - R_o)t = (R_i - 1/t_s)t \qquad (2)$$

$$t_q = n(t)t_s \qquad (3)$$

$$t_t = t_q + t_s = n(t)t_s + t_s = (n(t)+1)t_s \qquad (4)$$

If the calculated throughput time t_t for a given node is found to be unacceptable, the node is said to be "clogged" and additional copies of CE need to be initiated in order to remove the "clog." The additional copies of the node would be placed in parallel with the clogging node as in Figure 3.2, and would perform the same function as the original node. The service time of the new multi-copy node is t_{sc}/N, where t_{sc} is the service time of the original clogging node and N is the number of copies present in the multi-copy node.

$$N = \lceil t_{sc} / t_a \rceil \qquad (5)$$

where t_a is the acceptable delay time for the node and must be greater than or at least equal to the inverse of R_i because it is impossible for data to be output faster than they are input.

The above equations form the basis of a mathematical queuing model for a simple dataflow graph.

A complex dataflow graph can be resolved into basic component configurations such as the Linear Pipeline, the Fork, the Join, and the Feedback Node. The queuing theory models of each of these configurations are developed below.

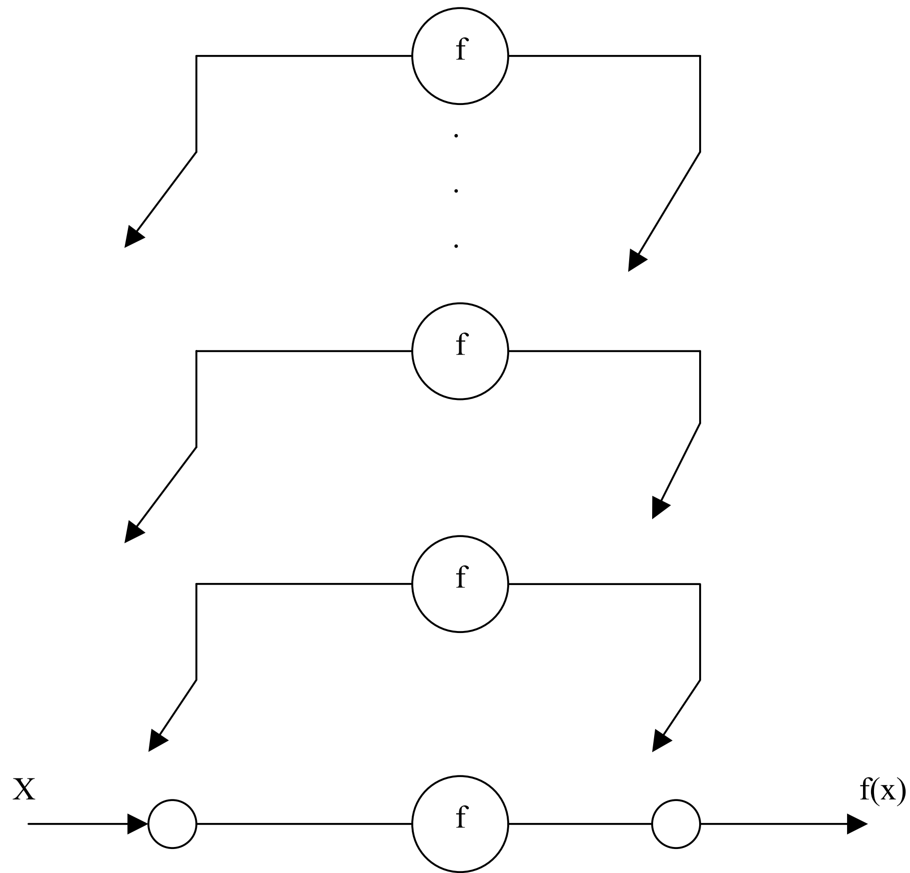


Figure 3.2 A Multi-Copy Node

3.1.1 LINEAR PIPELINE

Figure 3.3 illustrates a linear pipeline configuration dataflow graph and its hardware oriented representation. The system throughput time t_T is merely the sum of the delays contributed by the individual nodes. Equation (6) and (7) give the formula for individual node delay time and system throughput time.

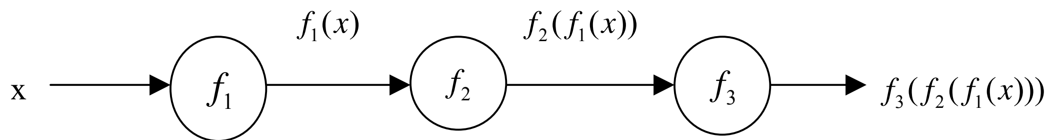
$$t_t = (n(t) + 1)t_s \tag{6}$$

$$t_T = \sum_{i=1}^{N} t_{ti} \tag{7}$$

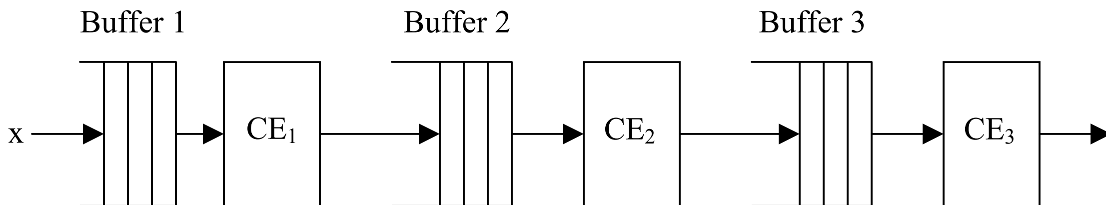
By substituting equation (6) into equation (7), we have

$$t_T = \sum_{i=1}^{N} (n_i(t) + 1) t_{si}$$

If the value of t_T is not acceptable, the node with the longest throughput time (clogging node) needs to be determined and duplicated by the operating system. The throughput time of the new system thus formed can then be calculated. The above procedure is repeated until the value of t_T is acceptable.



(a) Dataflow Graph of a Linear Pipeline System



(b) Hardware Oriented Representationof a Linear Pipeline System

Figure 3.3 Linear Pipeline System

3.1.2 FORK

Figure 3.4 shows a dataflow graph and hardware oriented representation of a generalized fork. The letter S denotes the source node and D denotes a destination node.

Chapter 2 has mentioned two types of fork – selective fork and nonselective fork. For selective fork, the data vector leaves the source node and then chooses one path to proceed at a certain probability. Let's define P(x) as the probability that a given data vector will follow path x upon reaching the fork. Then we can have:

$$R_{path1} = R_{os} P(1) \tag{8}$$

but

$$R_{path1} = R_{iD_1} ,$$

therefore

$$R_{iD_1} = R_{os} P(1) \tag{9}$$

In general,

$$R_{ix} = R_{os} P(x) . \tag{10}$$

Note: If P(x) is a normalized probability distribution, then

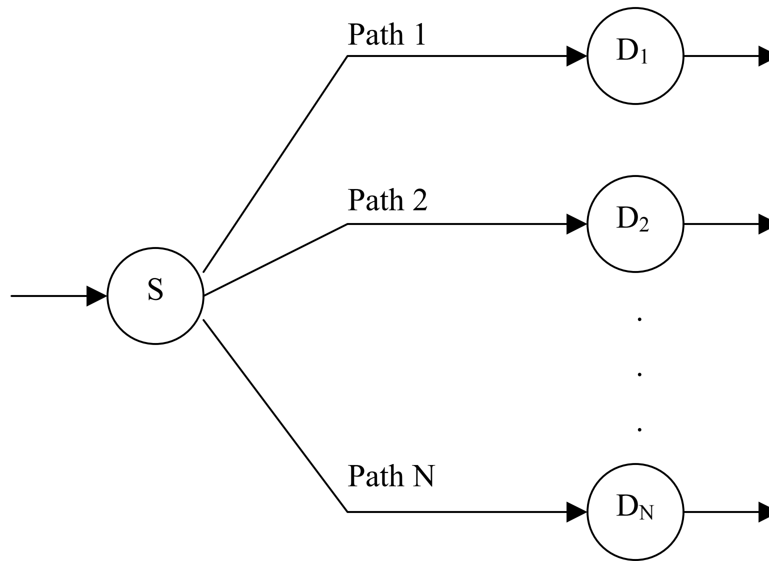
$$\sum_{x=1}^{N} P(x) = 1 . \tag{11}$$

For nonselective fork, the data vector leaves the source node and then proceeds to all the following paths, so

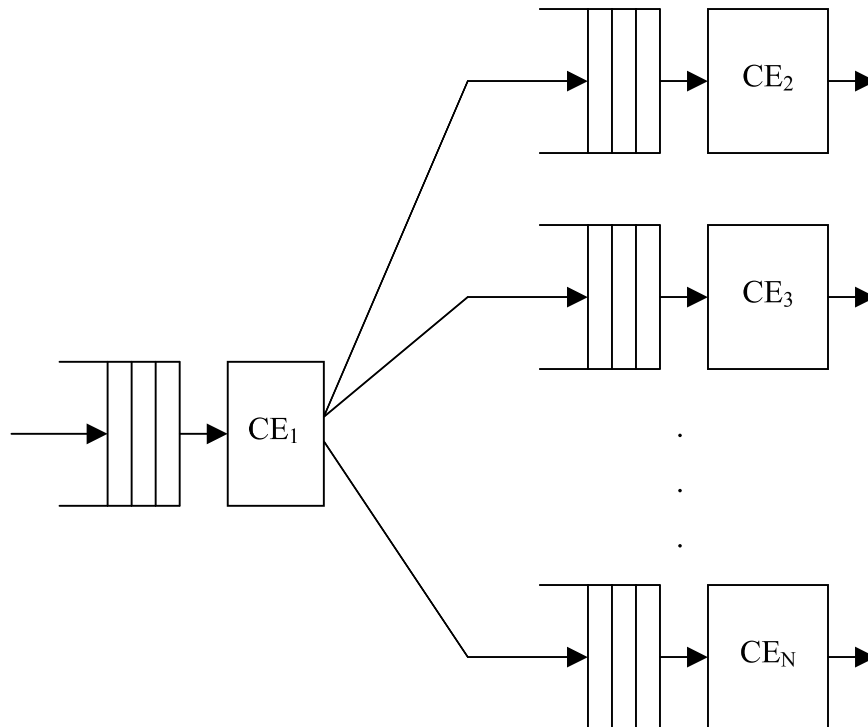
$P(x) = 1$ for all x, so

$$R_{ix} = R_{os}$$

The throughput time for any branch of the fork system can be considered as the source node and the particular destination node to compose a linear pipeline. Then the techniques we introduced in the last section can be used to determine the t_T .



(a) Dataflow Graph of a Fork



(b) Hardware Representation of a Fork

Figure 3.4 Fork

3.1.3 JOIN

A join is a node at which two or more branches merge to enter a single node as shown in Figure 3.5. There are also two types of join – selective join and nonselective join. A selective join processes the data on a "first come, first served" basis. In this case, the input data rate to node D, denoted R_{iD}, is simply the sum of the output data rates from all the source nodes whose data flows are joined. That is,

$$R_{iD} = \sum_{j=1}^{N} R_{os_j} \ . \tag{12}$$

The throughput time for a given data vector entering the system of Figure 3.5 can be determined as follows. The data vector will enter one of the source nodes S_n and will first be delayed by an amount of time equal to the throughput time of that node, t_{tn}. The data vector will then join with data from the other source nodes at the input of the destination node to await execution. The time needed for the data vector to transit the destination node can be found by application of a combined form of equations (2) and (4):

$$t_t = \left[\left(R_i - 1/t_s\right)t + 1\right]t_s \tag{13}$$

Upon substituting equation (12) in equation (13), we have

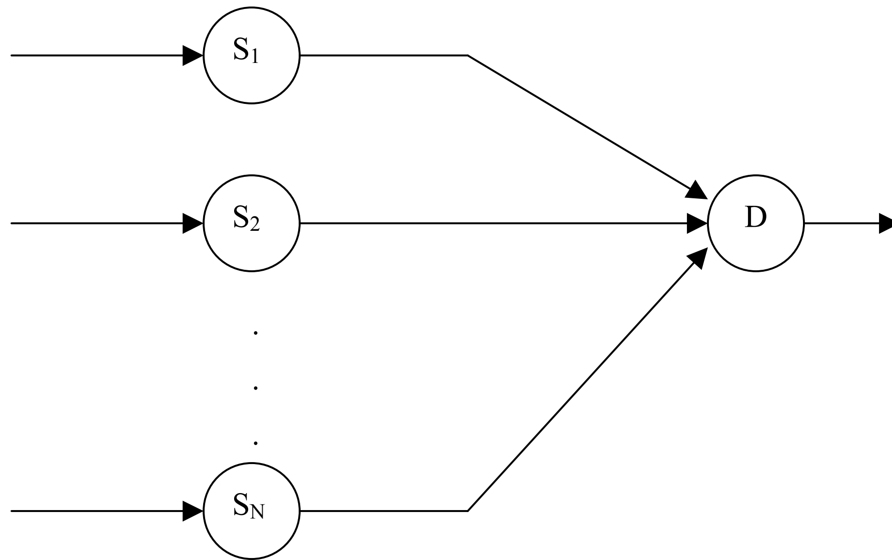
$$t_{tD} = \left[\left(\sum_{j=1}^{N} R_{os_j} - 1/t_{sD}\right)t + 1\right]t_{sD}. \tag{14}$$

We can now find the system throughput time for the data vector entering the join through any source node n.

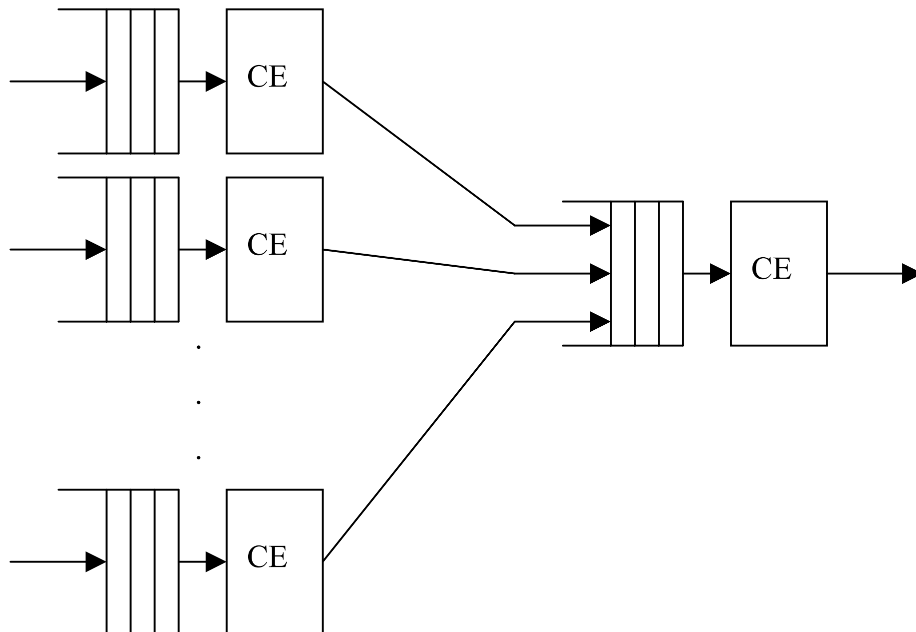
$$t_{Tn} = t_{sn} + \left[\left(\sum_{j=1}^{N} R_{os_j} - 1/t_{sD}\right)t + 1\right]t_{sD} \tag{15}$$

A nonselective join processes the data in a certain order (other than first come, first served). In this thesis, only the selective join was simulated. So the detail of nonselective

join will not be discussed. A detailed description of an example of nonselective join can be found in [2,3].



(a) Dataflow Graph of a Join



(b) Hardware Representation of A Join.

Figure 3.5 Join

3.1.4 FEEDBACK

In certain systems a data vector returns to the input queue of the node after it left the node for further processing. Such a node is considered as a feedback node with a portion of its output coupled back to its input. The dataflow graph for a simple feedback node is shown in Figure 3.6. The output of a feedback node is a fork, so a probability distribution must be determined for a feedback node in order to model it accurately. This probability distribution, denoted $P_f(n)$, is defined as the probability that a data vector leaving node n will “feedback” to node n. Referring to Figure 3.6, the rate at which data returns from the output to the input side of node n is R_f. This feedback rate can be calculated as follows:

$$R_f = P_f(n) R_o' \quad , \tag{16}$$

Where R_o' is the total output rate from the computing element of node n. Similarly, the output rate of node n as seen by a subsequent node is

$$R_o = \left(1 - P_f(n)\right) R_o' . \tag{17}$$

The input of the feedback node is considered as a join. The total input rate to the computing element of the node, R_i', will be the sum of the rates from preceding data sources and the feedback rate. Therefore,

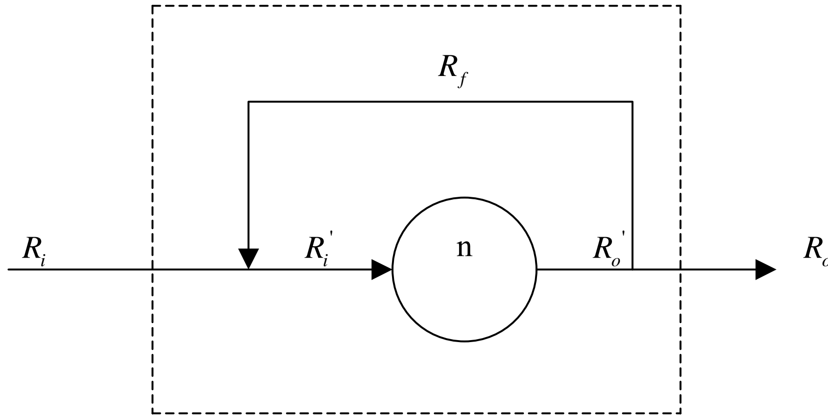
$$R_i' = R_i + R_f \tag{18}$$

By substituting R_i' for R_i in equation (13), we can find the nodal throughput time for a data vector entering the buffer of a feedback node at any time t, i.e.,

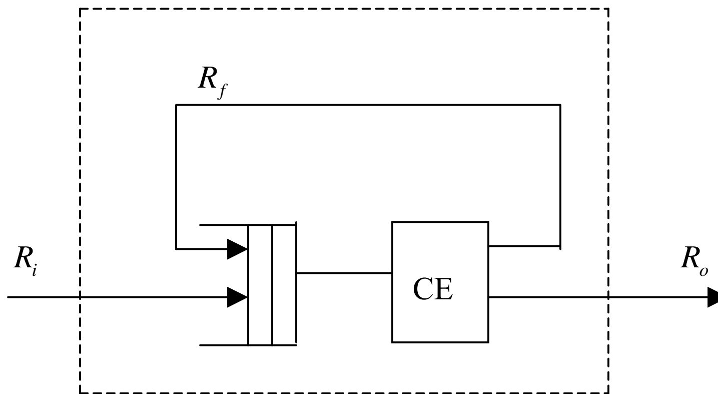
$$t_t = \left[\left(R_i' - 1/t_s\right) t + 1\right] t_s \tag{19}$$

The throughput time of the feedback system depends on the total number of times that a given data vector passes through the buffer and computing element.

$$t_T = \sum_{i=0}^{I} t_{ti} , \tag{20}$$



(a) Dataflow Graph of a Feedback Node



(b) Hardware Oriented Representation

Figure 3.6 Feedback Node

3.2 Static Load Balancing Algorithm Analysis for Program "COPY"

The HDCA is capable of changing its configuration so that it can execute a given algorithm with maximum efficiency. Once the algorithm has been developed in the form of a dataflow graph, we can use the program "COPY" to determine beforehand the optimum number of copies of multi-copy processes, which are required to insure that "clogging" does not occur at any node within a flow graph. A general dataflow graph of an algorithm can be thought of as being composed of a number of pipelines, each executing its own algorithm on the data that is input to it. So, the following five steps are applied in this analysis algorithm:

1. Decompose the dataflow graph into its constituent pipelines.
2. Classify each node of the dataflow graph as to whether it is a fork, a join, a singular node, or a common node.
3. Determine the data arrival rate of each node.
4. Calculate the number of copies of each node that will be required to minimize queuing at the input buffers of the computing elements and thus maximize the system throughput.
5. Determine pairs and/or groups of processes that can be combined to reduce computing element demand.

In order to understand how this flow-graph analysis algorithm functions, let us analyze an example dataflow graph manually according to this algorithm. Figure 3.7 is a general dataflow graph and is adapted from the example radar problem presented in reference [14]. Table 3-1 contains sample parameter values from the graph. The node or process is labeled by P followed by two numbers; P means process; the first number is the level number in the graph, and the second number is the node number in the same level. For example, P_{31} is the first process on the 3rd level in the dataflow graph.

The first step is to decompose the dataflow graph into its constituent pipelines. A pipeline is merely a string of connected nodes through which data may pass in traveling through a system from an input point to an output point. Our example dataflow graph is composed of three pipelines. Pipeline one is made up of the processes labeled P_{11}, P_{21}, P_{31}, P_{41}, P_{51}, P_{61}, and P_{71}. The second pipeline contains P_{11}, P_{21}, P_{32}, P_{41}, P_{51}, P_{61}, and

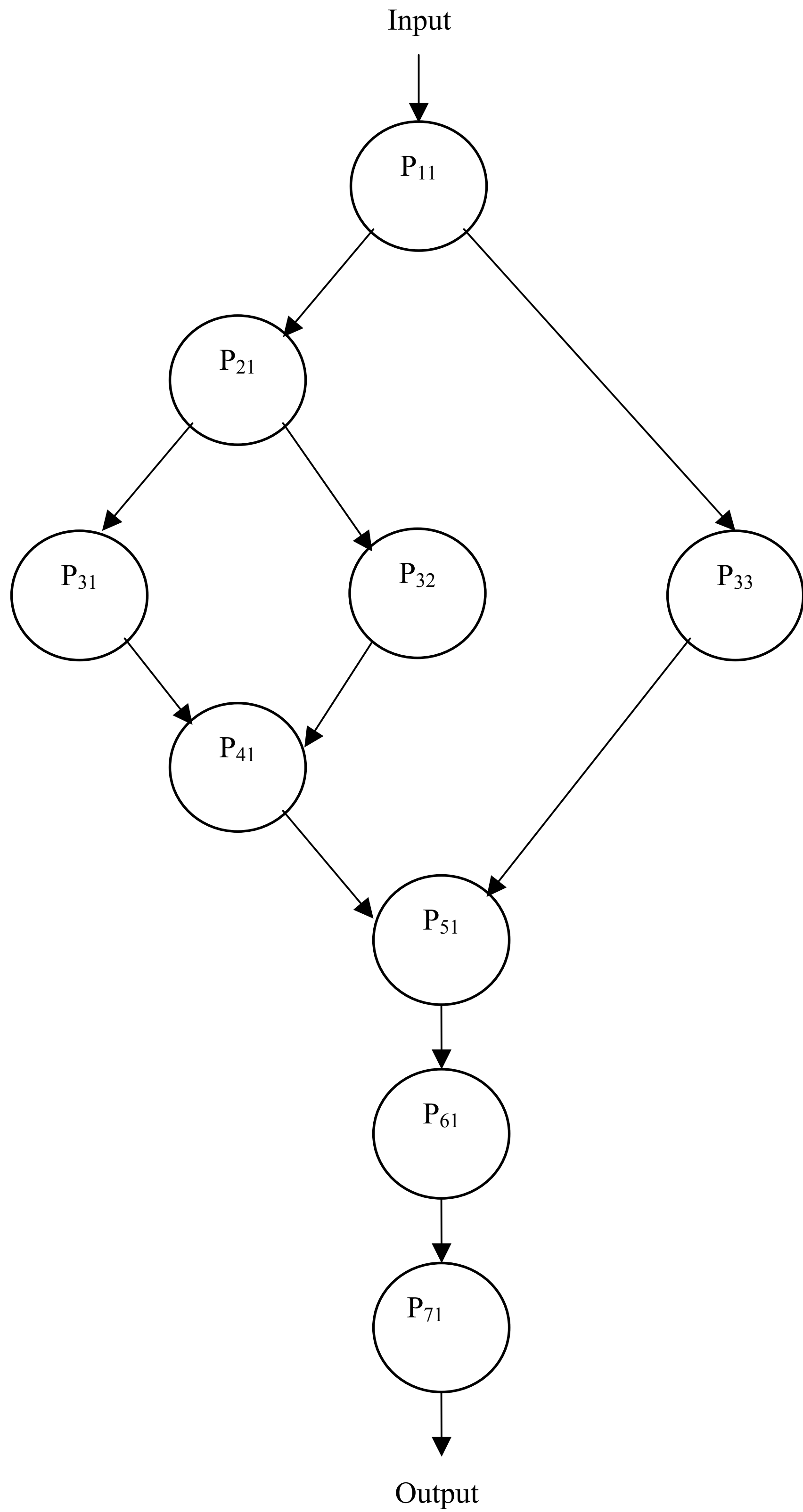


Figure 3.7 Example of A General Dataflow Graph

Table 3-1: Sample Parameter Values for Example Dataflow Graph

Process Designation	Execution Time (Milliseconds)	Process Length (Kilobytes)
11	0.85	0.425
21	1.63	2.5
31	1.3	0.5
32	0.32	0.05
33	2.7	1.5
41	0.96	0.85
51	1.87	0.45
61	0.69	15.0
71	1.12	0.9

Input Data Rates (data items/millisecond)

Peak Load: 3.8

Average Load: 2.5

Probability Distributions for Forks

$P_{11->33}$: 0.65

$P_{11->21}$: 0.35

$P_{21->32}$: 0.2

$P_{21->31}$: 0.8

Program Memory/Computing Element: 16 kilobytes

P_{71}. And the last pipeline has nodes P_{11}, P_{33}, P_{51}, P_{61}, and P_{71}. Next step is to classify each node as to whether it is a fork, a join, a singular node, or a common node. In this graph, P_{11} and P_{21} are forks; P_{41} and P_{51} are joins. P_{31}, P_{32}, and P_{33} are contained in only one pipeline each, so they are referred to as singular nodes. P_{61} and P_{71} are contained in more than one pipeline but they are neither forks nor joins. They are referred to as common

nodes. Once all the nodes have been classified, the next step is to find the data arrival rates at each node based on the queuing models that have been discussed in section 3.1. According to Table 3-1 and Figure 3.1, we know that the data enters the system at P_{11} at a rate of 3.8 Data Items per Millisecond (DI/msec.) under peak load. Here the forks are all selective. So the input rate at P_{21} equals the rate for P_{11} (3.8 DI/msec.) times the probability that a given data item will go from P_{11} to P_{21} ($P_{11->21}$=0.35). The input rate at P_{33} equals the rate for P_{11} times the probability that a given data item will go from P_{11} to P_{31} ($P_{11->33}$=0.65). Similarly, the data rate at P_{31} equals the data rate calculated at P_{21} times $P_{21->31}$, times $P_{21->32}$ for P_{32}. All the data coming out of P_{31} and P_{32} goes to P_{41} as P_{41} is a join. The data rate of P_{41} is the summation of data rate of P_{31} and data rate of P_{32}. The data rate of P_{51} equals data rate of P_{51} plus the data rate of P_{33}. P_{61} and P_{71} form a linear pipeline with source node P_{51}. In this case the input data rates to both P_{61} and P_{71} are the same as that of P_{51}. The results of the above analysis for both the peak rate case and average rate case are summarized in Table 3-2.

Once the data arrival rates are calculated and the process execution times are given in Table 3-1, the maximum number of copies of each node is just the product of the data arrival rate of that node and its process execution time, rounded up to the next integer. Results of these calculations for all the nodes of our example flow graph for both peak and average data rates are also presented in Table 3-2.

The program "COPY" can also combine the processors needed by different nodes in order to reduce the total CE usage. Reference [2] has a detailed description of this combination. The original program "COPY" was written in BASIC in [2], and a detailed programming algorithm was introduced in Chapter 4 of [2]. Then the BASIC version COPY was adapted into a C version "copy" in [1]. Since both references were written some time back and there are not any soft copies of the program, the "C" version program needs to be recompiled. It is considered a part of this thesis. When running the program, the same example data parameters were used as illustrated in [2] in order to validate the computing results. The achieved results are the same as those listed in [2], and this demonstrates that the newest version of "copy" works correctly as desired.

Table 3-2: **Results of the Analysis of the Example Flow Graph**

Node	Peak Rate (DI/msec.)	# of Copies (Peak)	Avg. Rate (DI/msec.)	# of Copies (Average)
P_{11}	3.8	4	2.5	3
P_{21}	1.33	3	0.87	2
P_{31}	1.06	2	0.7	1
P_{32}	0.27	1	0.17	1
P_{33}	2.47	7	1.63	5
P_{41}	1.33	2	0.87	1
P_{51}	3.8	8	2.5	5
P_{61}	3.8	5	2.5	2
P_{71}	3.8	3	2.5	3

4 GRAPHICAL SIMULATOR

4.1 Programming Environment and Language

The simulator was developed using the C programming language and compiled using GNU's "gcc" compiler. It should be run under the LINUX operating system. GNU's 2-D Vector's Graphics Library, libplot 4.1, from the Plotutils Package is used to do the plotting part of the job. GNU libplot 4.1 is a free function library for drawing two-dimensional vector graphics. It can produce smooth, double-buffered animations for the X Window System, and can export graphics files in many file formats. It is "device-independent" in the sense that its API (Application Programming Interface) is to a large extent independent of the display type or output file format.

The graphics programs and GNU libplot can export vector graphics in the following formats.

1. X: If this output option is selected, there is no output file. Output is directed to a popped-up window on an X Window System display.
2. PNG: This is "portable network graphics" format, which is increasingly popular on the Web.
3. PNM: This is "portable anymap" format. There are three types of portable anymap format: PBM (portable bitmap, for monochrome images), PGM (portable graymap), and PPM (portable pixmap, for colored images).
4. GIF: This is pseudo-GIF format rather than true GIF format.
5. SVG: This is a Scalable Vector Graphics format. SVG is a new, XML-based format for vector graphics on the Web.
6. AI: This is the format used by Adobe Illustrator. Files in this format may be edited with Adobe Illustrator (version 5, and more recent versions), or other applications.
7. PS: This is an idraw-editable Postscript format. Files in this format may be sent to a Postscript printer, imported into another document, or edited with the free idraw drawing editor.
8. CGM : This is Computer Graphics Metafile format, which may be imported into an application or displayed in any Web browser with a CGM plug-in.

9. Fig: This is a vector graphics format that may be displayed or edited with the free xfig drawing editor.
10. PCL 5: This is a powerful version of Hewlett--Packard's Printer Control Language. Files in this format may be sent to a LaserJet printer or compatible device.
11. HP-GL: This is Hewlett--Packard's Graphics Language.
12. ReGIS: This is the graphics format understood by several DEC terminals (VT340, VT330, VT241, VT240) and emulators, including the DECwindows terminal emulator, dxterm.
13. Tek: This is the graphics format understood by Tektronix 4014 terminals and emulators, including the emulators built into the xterm terminal emulator program and the MS-DOS version of kermit.
14. Metafile: This is a device-independent GNU graphics metafile format. The plot program can translate it to any of the preceding formats.

In the program hdca.c, the X Plotter is used to plot the graphic and the output is in the X format, that is, the output is directed to a pop-up window on an X Window system display.

There are bindings for C, C++, and other languages. The C binding, which is the most frequently used, is also called libplot, and the C++ binding, when it needs to be distinguished, is called libplotter.

For more information about Plotutils Package, please see the following website. http://www.delorie.com/gnu/docs/plotutils/plotutils_toc.html#SEC_Contents

4.2 Simulation Algorithm

The program “COPY” described in Chapter 3 computes the optimum number of copies of a process required to ensure smooth flow of data. These values will be used in the graphic simulation. The simulation mainly consists of 4 modules: Graphic Module(GM), Simulation Module(SM), Update Module, and Shutoff Module. The relationship between these four modules and the program “COPY” is illustrated in Figure 4.1.

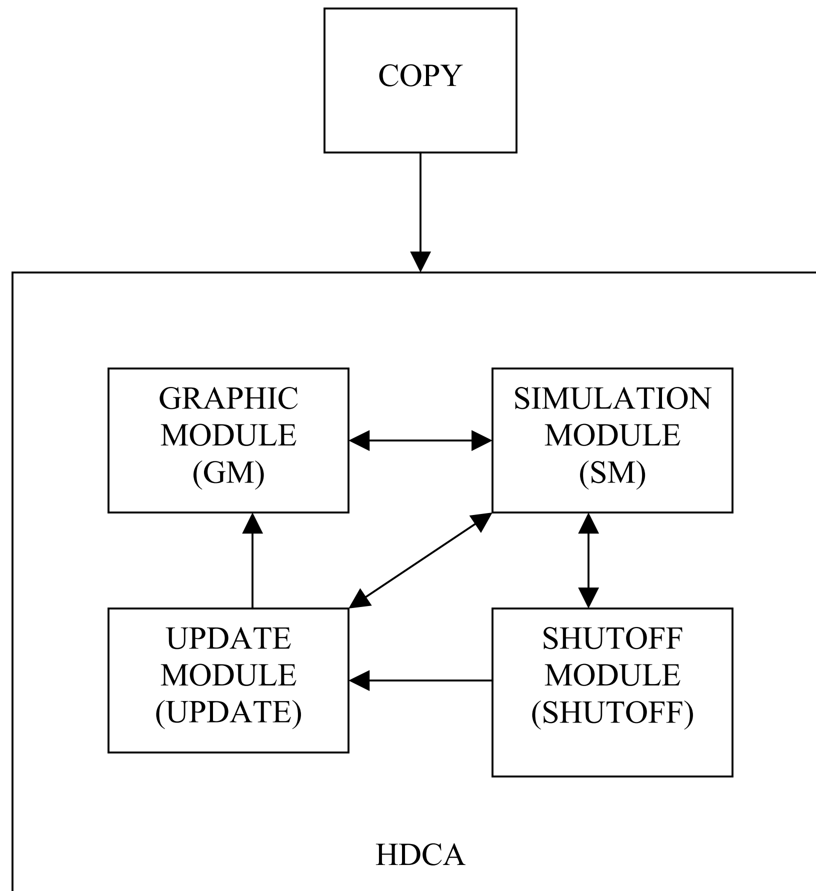


Figure 4.1 Relationship Between COPY and HDCA Modules

Graphic Module (GM) takes the results of "COPY" and other data parameters, and draws a dataflow graph on the screen, with an empty queue beside each node. Then the Simulation Module (SM) takes over the program. As the simulation is proceeding, the status of each copy of the process (busy or idle) and the queue depth change dynamically. Whenever the change occurs, the Update Module works and updates all the parameters that are used in the GM. GM then redraws the dataflow graph while the simulation is going on. The Shutoff Module is used for fault tolerance analysis. The program can enter the Shutoff module at the beginning of the simulation module. There is no clear boundary between the four modules. They work closely to produce a smooth graphic simulation. These modules will be discussed in more detail.

4.2.1 GRAPHIC MODULE (GM)

The basic function of the GM is to collect the required data and draw the basic dataflow graph on a graphics screen. GM operates on a screen area of 4000x4500 Graphic Display Units (GDUs) of a pop up X Window. The program is not terminal dependent, though. It can be adapted to display the vector graph in other formats. GM adjusts all sizes and types of dataflow graphs to fit on to the screens by calculating the spaces between the processes. As shown in Figure 4.2, the upper 4000x500 area is used for the title, and the lower 4000x4000 area is used for the dataflow graph.

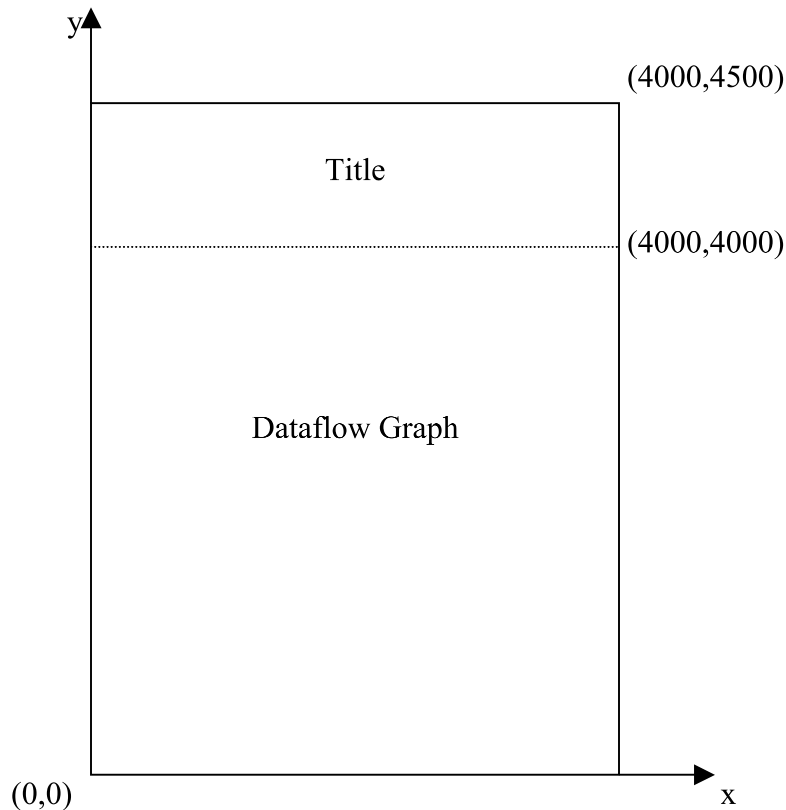


Figure 4.2 Schematic of the Display Screen

For the purpose of drawing, the dataflow graph is treated like a column vector and several row vectors. The rows of the column vector are called “levels” and the columns of each row vector are called “nodes”. Each node represents a process and is represented

by a 200x200 GDUs square box. The user must input the number of levels and the number of nodes of each level of the dataflow graph before the execution of the program. According to the number of the levels of the dataflow graph -- N, GM divides the 4000x4000 area screen into N rectangular boxes of 200 GDUs high along the y-axis. The spacing between these boxes is determined by the number of levels "N" and the total available space in the y direction. In this thesis, the total available space in the y direction for drawing the dataflow graph is 4000 GDUs and the total available space in the x direction is 4000 GDUs too. The decrement or the space between two levels is (4000/N-200). The levels are drawn top down. Each level is now considered independently as a 1xn matrix where n is the total number of nodes in that level. The nodes are drawn from left to right. GM calculates the "increment" or the spacing between the nodes in that level in a similar way. The boxes are spaced equally within the level. The top level is drawn at half the "decrement" from the top and the bottom level is drawn at half the "decrement" above the bottom. Likewise, the first node in a level is drawn at one half the value of "increment" from the left end of the x-axis and the last node in a level is drawn at one half the value of "increment" from the right end of the x-axis. Once a square is drawn, coordinates of its top and bottom midpoints are used to calculate the coordinates of the top midpoint of its queue box and all of these coordinates are stored for later use. The next step is to divide each square into a number of initial copies, as calculated by the program "COPY". GM divides the height of the square into several segments, which equal the number of initial copies, and draws horizontal lines indicating the copies of the process. Then GM draws an empty queue at the right side of the process box and draws a line at the position where the queue depth is 4, indicating the initial threshold. GM finishes the drawing of the dataflow graph by connecting the paths between the processes and queue boxes. The initial dataflow graph is drawn in red.

When the simulation is running, GM puts a dot inside the copy of process box indicating the copy is busy, and puts a cross inside the box indicating the copy is shut off. With the input rate changing, the number of copies of a CE that is needed might exceed the initial number calculated by the program "COPY". In this case, GM will draw extra copies at the left side of the process box in green. The total number of copies will also be labeled in green. Whenever one extra copy of processor is initiated, the threshold of the

queue will increase by two automatically. The line indicating the threshold of the queue is also redrawn in green. When a job is finished by one process, it is routed to the next process by the Simulation Module, meanwhile, GM draws an arrow line indicating the direction of the data flow. GM needs to update and reproduce the dataflow graph whenever changes happen.

Figure 4.3 is an example dataflow graph drawn by GM. This dataflow graph has 7 levels altogether. Each node is labeled by its level number followed by its node number. Based on the number of initial copies calculated by the program “COPY”, the process boxes are divided into several smaller boxes. At the lower left corner of each process box is the number of copies that is needed in order to keep the queue level below the threshold. In this graph, all the numbers are the same as the initial numbers, so they are all labeled in red. All queues are empty, and all the thresholds are equal to 4. The arrow lines show the data flow direction.

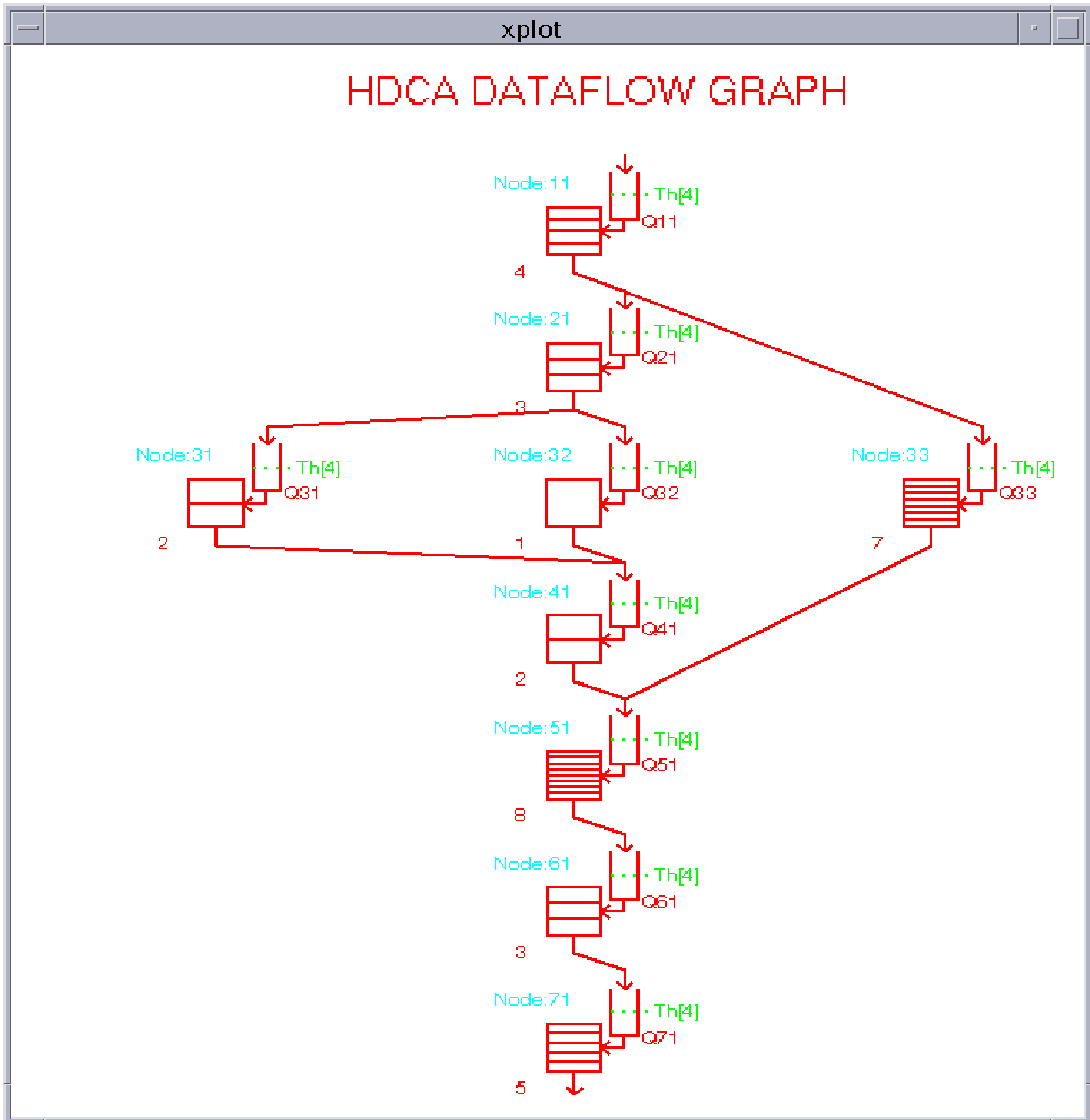


Figure 4.3 An Example Dataflow Graph

4.2.2 SIMULATION MODULE (SM)

The Simulation Module (SM) is the core of the program. It monitors the status of each copy of the processor and queue, allocates resources dynamically, routes the data accordingly, and keeps the program running correctly. Without SM, GM could only draw a static two-dimension vector graph. It is the SM that makes the dynamic simulation possible.

As shown in Figure 4.4, the SM contains three major loop structures that control the logic of the simulation. The outermost loop is the master clock, which controls the timing of the whole system simulation. It is assumed that each loop takes 1 microsecond to execute. So we give this loop a new name, "micro-cycle." The final count of this loop is the number of micro-cycles that the simulation has run. The second loop is the control of level. The third one is the control of the node in each level. SM scans from the first level to the last level, and from the first node to the last node in each level. Inside the third loop, there are many small loops to check the status of each copy of the processor, including whether a copy is shut down, busy, or free, whether a shutdown copy needs to be reopened and whether a busy copy has finished the job. When new data comes, if there are any free copies available, SM will execute the job. If there is no free copy, then SM puts the job in the queue. When there is no new incoming data, SM checks the queue of the node. If the queue is not empty and there is a free copy available, SM will execute one job from the queue and decrease the queue level. Also, the queue depth needs to be checked. If the queue level exceeds the threshold, a new processor needs to be activated to process the data and the threshold of the queue increases by two. When the queue level falls below the new queue threshold, SM will deactivate the extra copy of the processor and set the threshold back to its previous value.

The node at the first level, which receives data for the system, is called "topnode." Since the "topnode" is different from other nodes, it is processed separately at the beginning of the program. When a token/data item arrives at the input, SM scans the copies of that node to see if any free copies are available for process execution on this data. If there are no free copies available, the data item is stored in the queue and the queue counter is incremented. Otherwise the copy that is idle absorbs the incoming data and begins execution of its program on the data item.

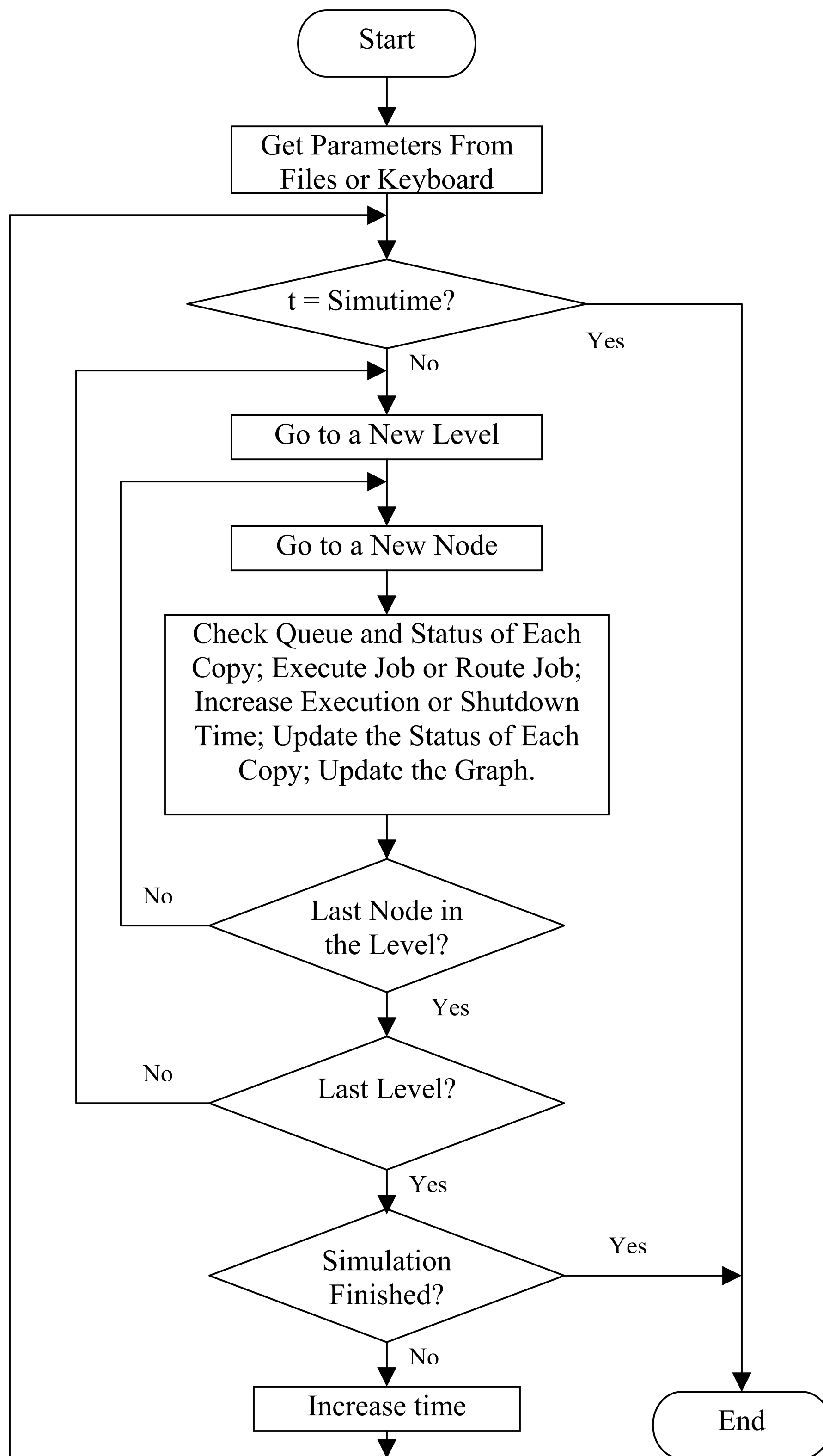


Figure 4.4 High Level Flow Chart for Simulation Module

4.2.3 UPDATE MODULE

When SM detects that a node has finished processing incoming data, it routes the data to the subsequent node. If the current node is a singular node, SM finds the subsequent node and increases the counter "nodejob" of the next node.

If the current node is a fork, the data output after being processed by the fork is directed to one of the branches associated with the fork, depending upon the probability associated with that branch and the random number generated by the function "random." The probabilities associated with that branch of the fork are not absolute probabilities, but are modified as "cumulative probabilities." The following example shows how to calculate the "cumulative probabilities." Suppose a fork has 3 branches a, b, and c respectively, and let the probabilities associated with these branches be 0.1, 0.3, and 0.6. The cumulative probability assigned by SM to the branch a is 0.1. The cumulative probability to branch b is the summation of 0.1 and 0.3, that is, 0.4. The cumulative probability to branch c is the summation of 0.1, 0.3, and 0.6, that is, 1.0. In other words, cumulative probability is just the accumulation of the previous absolute probability. If the random number that is generated by the function "random" is greater than or equal to the associated cumulative probability, SM will route the data to this branch and increase the counter "nodejob" for this subsequent node. If not, SM will check the next branch and perform the above procedure again.

If the current node is a "tailpiece," that means there is no subsequent node. After the data is processed, one job is finished. SM decreases the counter of the left jobs "jobleft."

The Update Module is used to route jobs and to reset the status of the copies of the processor. Although it is explained as a different module, it is actually a subroutine and is embedded in the Simulation Module. After Update Module finishes its job, SM will take over the program again and continue its simulation.

4.3 Program "hdca"

The "C" program "hdca" is shown in Appendix A. It is based on an old "C" program "dpca" which is described in [1]. Some basic types of variables and structures from "dpca" have been kept, and new data structures have been added to the program "hdca." The algorithm for drawing the dataflow graph didn't change very much, but the

simulation algorithm is totally new. In “hdca,” the whole program is divided into several major functional blocks with each block performing a specific function. In this way, the program is very flexible and adding new functionalities becomes very easy. The program consists of five parts: preprocessor commands, global variable definitions, function prototypes, main function, and sub functions. Preprocessor commands define all the standard library header files the program will use. In this program, three header files are used: “stdio.h” is a standard input/output library; “math.h” is the mathematical declaration library; and “plot.h” defines all the functions that are used for drawing the graph. Preprocessor commands tell the compiler to include these three libraries when the program is compiled. The second part defines all the global variables that are accessed by all the functions. There are four structures defined in the program. Structure “link” defines the information for the links in the dataflow graph including the level number and the node number of both the source node and destination node, and the absolute probability for the fork branch. Structure “data” is used to store the coordinates of the upper midpoint and lower midpoint of the node box, and upper queue midpoint. Structure “nodeinfo” is used for the information. Element “fork” indicates whether a node is a fork, a singular node, or a tailpiece. “Processtime” is used to store the processing time of the node. Structure “copyinfo” defines two flags and two counters. Flag “shutflag” indicates whether a copy of the processor is shut down or not. The value of 1 means the copy is already shut down; the value of 0 means it is not. “Stopcount” is the counter used to record the time that this copy has been shut down. Flag “busycopy” indicates whether this copy is busy or not; 1 means it is busy and 0 means it is free. Table 4-1 lists the functionality of some arrays used in the program. There are 12 function prototypes including “draw_dataflow,” “draw_link,” “arrow,” “simulation,” “shutoff,” “redraw,” “update,” “draw_queue,” “draw_shut,” “draw_busy,” “draw_extracopy,” and “varate.” Table 4-2 shows the functionalities of these prototypes.

Table 4-1: Description of the Global Arrays

Variable Name	Function
Level	Number of the total levels of the dataflow graph
Node[i]	Number of the total nodes in i^{th} level
Copy[i][j]	Number of copies needed for the ith level j^{th} node during the simulation
Initialcopy[i][j]	Number of copies needed for the ith level j^{th} node that is calculated by program "COPY"
Qthreshold[i][j]	Threshold of the queue for node ij
Max_extracopy[i][j]	Maximum number of the extra copies that node ij can initiate
X	X coordinate
Y	Y coordinate
Dec	Decrement between levels
Inc	Increment between nodes
Decval[i][j]	Height of each copy for node ij
Linknum	Number of total links
totaljob	Total number of jobs that need to be executed
Jobleft	Number of jobs that still need to be executed
Path[i].lf	Level number of source node for i^{th} link
Path[i].nf	Node number of the source node for i^{th} link
Path[i].lt	Level number of destination node for i^{th} link
Path[i].nt	Node number of destination node for i^{th} link
Path[i].probability	Absolute probability that the data follows this link
Midpoint[i][j].x1	X coordinate of the bottom midpoint for node ij
Midpoint[i][j].y1	Y coordinate of the bottom midpoint for node ij
Midpoint[i][j].x2	X coordinate of the upper midpoint for node ij
Midpoint[i][j].y2	Y coordinate of the upper midpoint for node ij
Midpoint[i][j].qx	X coordinate of the queue's upper midpoint for node ij
Midpoint[i][j].qy	Y coordinate of the queue's upper midpoint for node ij

Table 4-1: Description of the Global Arrays (Continue)

Variable Name	Function
Information[i][j].fork	Flag that indicates whether node ij is a fork, a singular node, or a tailpiece
Information[i][j].processtime	Process time for node ij
Nodecopy[i][j].shutflag[k]	Flag that indicates whether the kth copy of node ij is shut down
Nodecopy[i][j].stopcount[k]	Shut down time for the kth copy of node ij
Nodecopy[i][j].busyflag[k]	Flag that indicates whether the kth copy of node ij is busy
Nodecopy[i][j].exetime[k]	Execution time for the kth copy of node ij

Table 4-2: Functionalities of the function program in HDCA

Function Name	Functionality
Draw_dataflow	Draw all the nodes of the given dataflow graph
Draw_link	Draw all the links between the nodes and empty queues for all nodes
Redraw	Redraw the data flow graph according to the current status
Draw_queue	Draw the queue
Draw_shut	Place a cross in the shut down copy
Draw_busy	Place a dot in the busy copy
Draw_extracopy	Draw extra copies for the node
Arrow	Draw an arrow
Simulation	Simulate the program
Update	Update all the variables and route the data
Clear_one_queue	Erase one token from the queue
Clear_busy	Erase the dot which is used to indicates the copy is busy
Clear_shut	Erase the cross which is used to indicate the copy is shut down
Shutoff	Shut down a copy

Figure 4.5 is the flow chart for the main program of "hdca." The first piece of data that a user should input is whether the program is to be run in a user input environment or in the auto input environment, that is, whether "sel" is 1 or 0. If "sel" equals 1, then the program gets all the data from the keyboard. If "sel" equals 0, the program enters the auto input branch, and gets all the required data from a series of files that are stored in the same directory as the main program. Since the program needs a large amount of data before it can run, it is very easy to get confused when a user is trying to form the data files. The user input environment will be very helpful in such a case. Two environments work in a similar way, except that auto-input mode needs to open and close a set of files. In both cases, the program needs to get the number of levels (N), the number of nodes in each level (node[10]), all links(path[100]) , and the number of copies in each node (nodecopy[10][10]). With these data ready, the program is capable of initiating the plotter and calling function "draw_dataflow" and "draw_link" to draw a complete dataflow graph before simulation begins. Then the program also needs some parameters such as whether a node is a fork, the processing time for each node, and the probability for each path of a fork in order to do simulation. These parameters are called α parameters. They are stored in the structure array "information[10][10]" except that the probability is stored in structure array "path[100].probability". After the α parameters are input and stored, the user needs to input the total number of tokens/data items at the top node, the average speed ratio at which the data is input at the top node, the total simulation time, whether any copy needs to be shutdown, and whether the input rate is variable. After getting all of this information, the program is ready to do simulation. Then the program calls the function "simulation." When the simulation is done, GM draws a complete dataflow graph with the status as it was. Then the "main" program closes the plotter and finishes the simulation. The "main" program looks very easy, because all the detailed jobs have been taken care of by the sub functions like "draw_dataflow," "draw_link," "Simulation," etc. According to their functionalities, the programs are categorized into four modules: graphical module, simulation module, update module, and shutdown module. The following pages will explain all the sub-functions according to their module category.

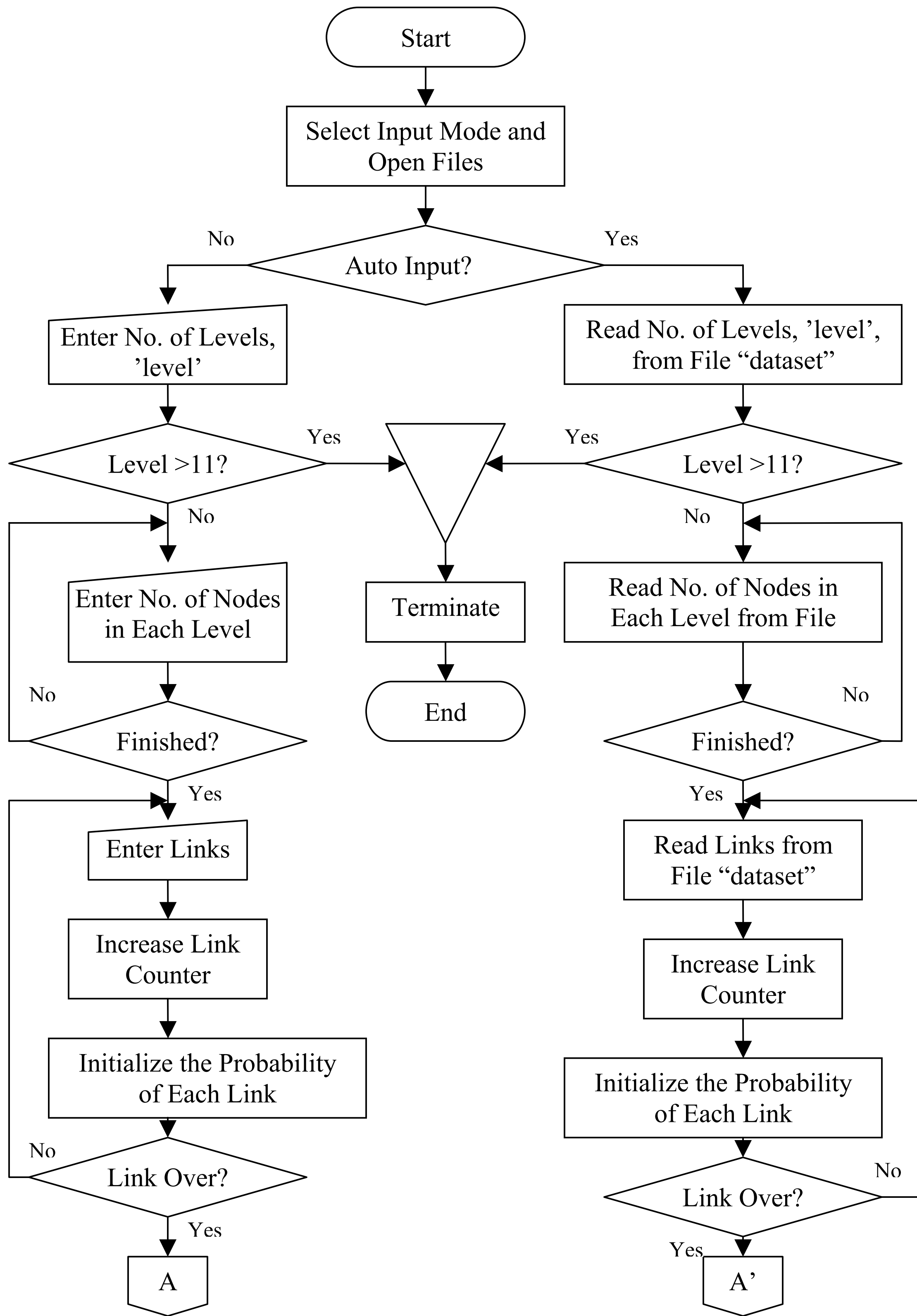


Figure 4.5 Flow Chart for Program HDCA

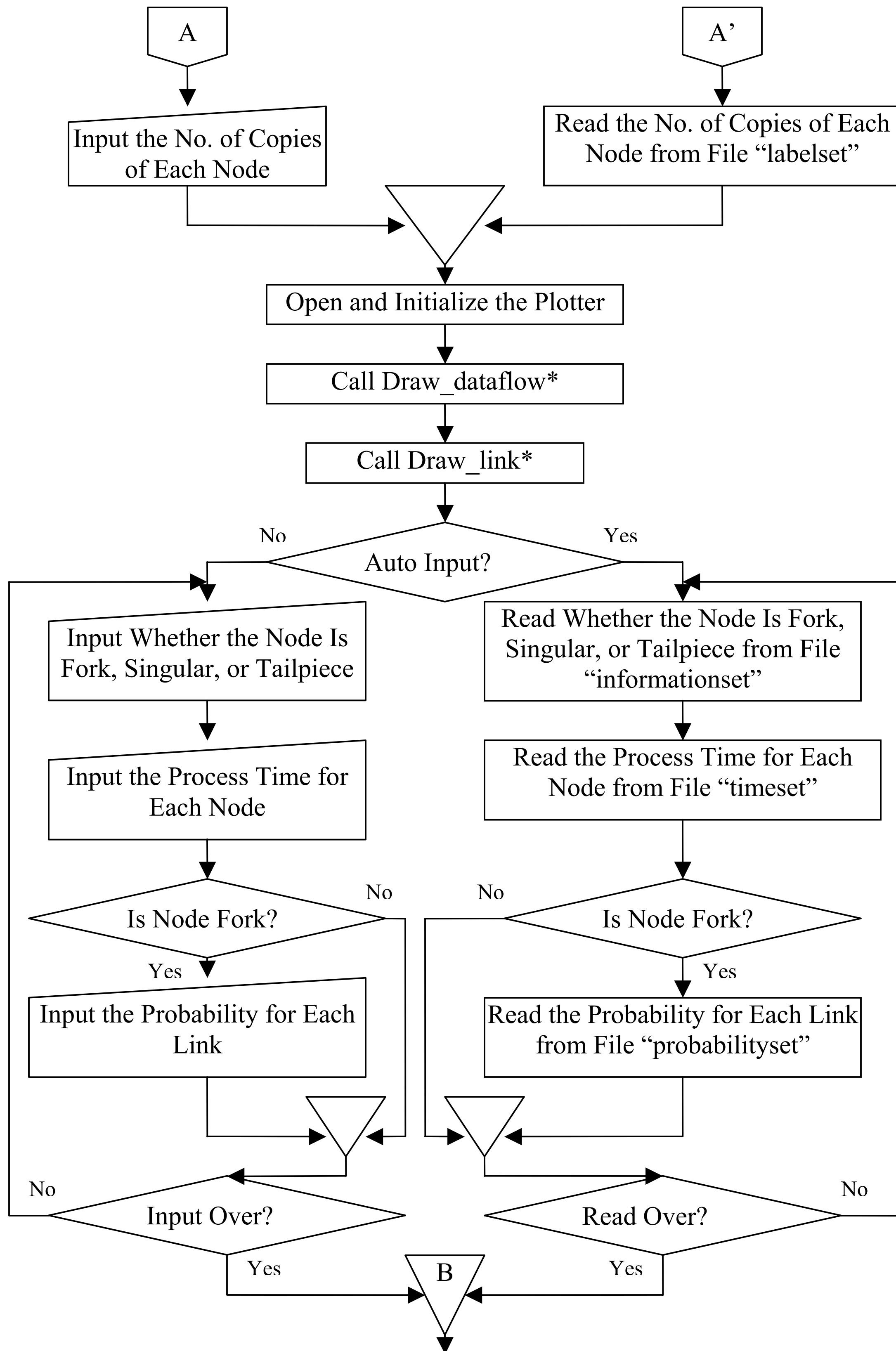


Figure 4.5 Flow Chart for Program HDCA (Continue 2)

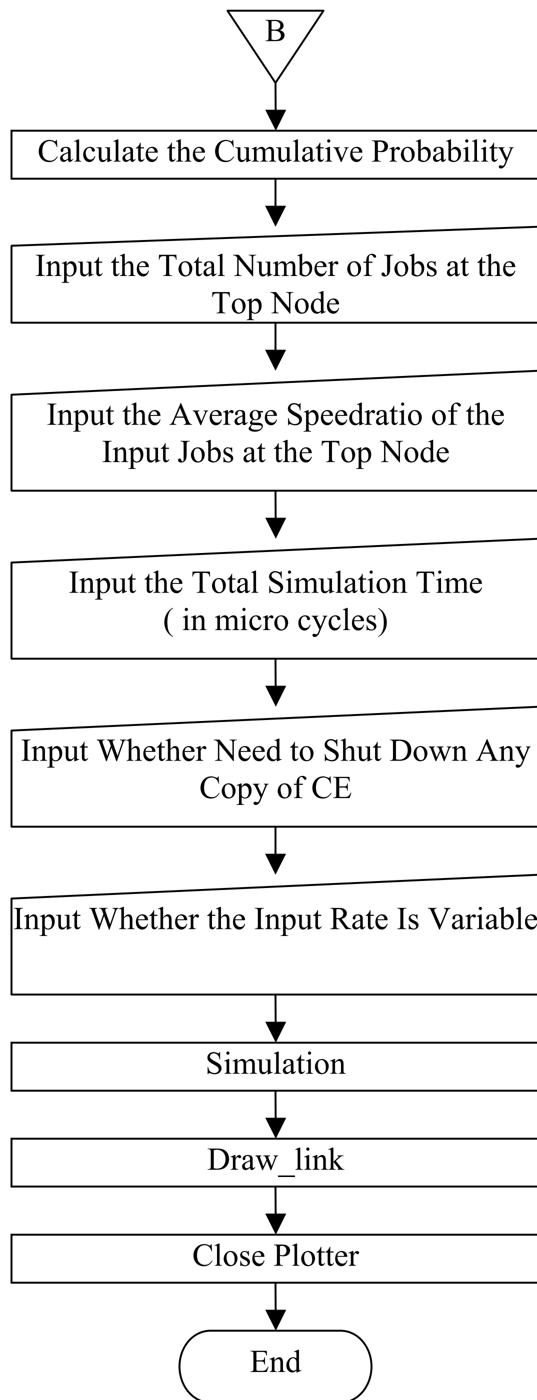


Figure 4.5 Flow Chart for Program HDCA (Continue 3)

4.3.1 GRAPHIC MODULE (GM)

In this module, there are 11 function programs: “draw_dataflow,” “draw_link,” “draw_queue,” “clear_one_queue,” “draw_shut,” “clear_shut,” “draw_busy,” “clear_busy,” “draw_extracopy,” “redraw,” and “arrow.”
These functions will be explained one by one.

a. Draw_dataflow

This function takes only one parameter passed by the calling function, “plotter.” Parameter “plotter” is defined in the main program when initializing the plotter. It is a pointer of type “plPlotter” and is created by calling “pl_newpl_r.” The parameter values of this plotter are specified by calling the “pl_setplparam” function. The “pl_setplparam” function acts on a “plPlotterParams” object, which encapsulates Plotter parameters. When a Plotter is created by calling “pl_newpl_r,” a pointer to a “plPlotterParams” object is passed as the final argument. Then calling the function “pl_openpl_r” will open the plotter. The plotter is alive until it is closed by calling “pl_closepl_r.” During its lifetime, it needs to be passed to the called function as a parameter in order to plot any graph. So “draw_dataflow” takes this parameter, and operates on the data or variables that are defined as global variables. The data type “plPlotter” and all the functions starting with “pl_” and ending with “-r” are defined in the “libplot” library. See reference [15] for more information about “libplot” library.

The flow chart for “draw_dataflow” is shown in Figure 4.5. In the main program, function “pl_space_r(plotter, 0, 0, 4000, 4500)” takes two pairs of arguments, specifying the positions of the lower left and upper right corners of a rectangular window in the user coordinate system that will be mapped to the output device that graphics will be drawn in. That is, the coordinate of the lower left corner is (0,0) and the coordinate of the upper right corner is (4500,5000) in the user coordinate system. “Pl_space_r” sets the affine transformation from user coordinates to device coordinates. At the beginning of the program, function “pl_ffontsize_r(plotter, 150)” sets the font size to 150, then draws a title for the graph at the top of the rectangular box. After that, the font size is set to 75, and is used to label the node name, queue name, and etc. Then the decrement between the levels is calculated according to the number of total levels and this value is stored in the

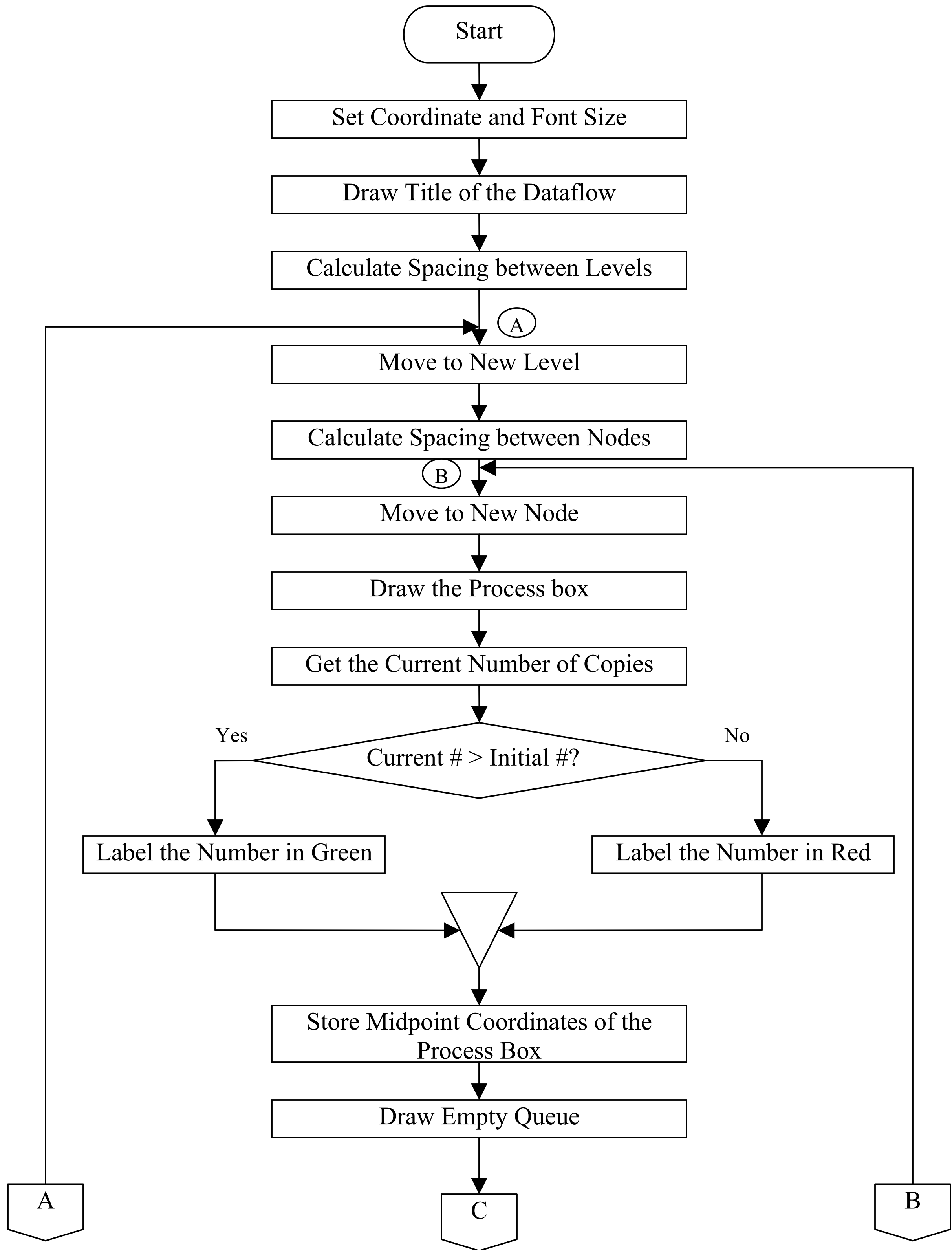


Figure 4.6 Flow Chart for "draw_dataflow"

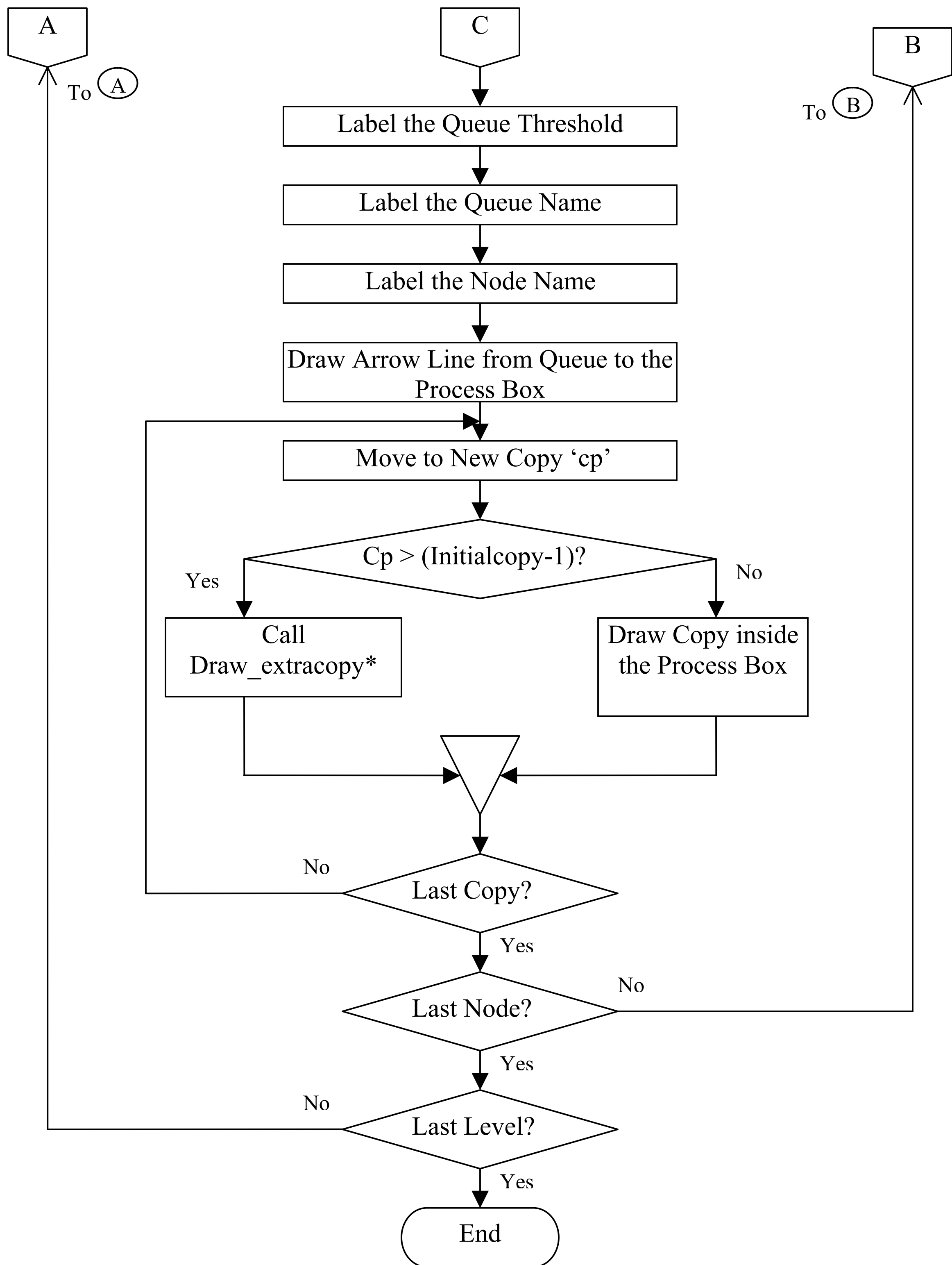


Figure 4.6 Flow Chart for "draw_dataflow" (Continued 2)

variable "dec." Then the program enters three loops. The outer loop calculates the increment between nodes in a certain level and stores the value to the variable "inc." In the second loop, the program draws a 200x200GDUs size box for each node in that level. According to the result of the program "COPY," the program divides the box into several small rectangular boxes with each small box representing a copy of the process. The nodes are drawn from left to right for the same level. The box furthest to the left represents the first node and the box furthest to the right represents the last node. The first level is drawn on the upper side and the last level is on the lower side. After drawing all the copies, the program labels the number of copies each node needs. If the number is the same as the initial assigned number, it is labeled in red. If the number is greater than the initial assigned number, it is labeled in green. That means that initially assigned processors are not enough to produce a smooth simulation that keeps the queue level below the queue threshold, and some extra copies of the process have been initiated. All the midpoint coordinates are then stored for later use. The program then draws an empty queue box, labels the queue threshold, and labels the queue name. An arrow line from the queue box to the process box indicates that the data enters the queue first, and then is executed when there is a free copy of the processor. Then the program enters the inner loop to check whether extra copies need to be drawn. By this time these three loops end, all nodes, queues and labels have been displayed on the screen.

b. Draw_datalink

The only parameter passed by the calling function is still the Plotter. The flow chart for program "draw_link" is shown in Figure 4.7. The function of this program is very pure: draw all the links between the source nodes and the destination nodes. The dataflow graph can be cyclic or acyclic. The feedback line makes the drawing a bit complicated. At the beginning, the program enters a small loop to draw input lines for all topnodes. Then another loop draws the output lines at the last level of the dataflow graph. If there are feedback lines for these tailpiece nodes, then draw the feedback lines. If the destination node is on the left side of the source node, draw the feedback line at the left side of the source node. If the destination node is on the right side of the source node, draw the feedback line at the right side of the node. Last, the program searches all the links and draws the paths from the source nodes to the queues of the destination one by

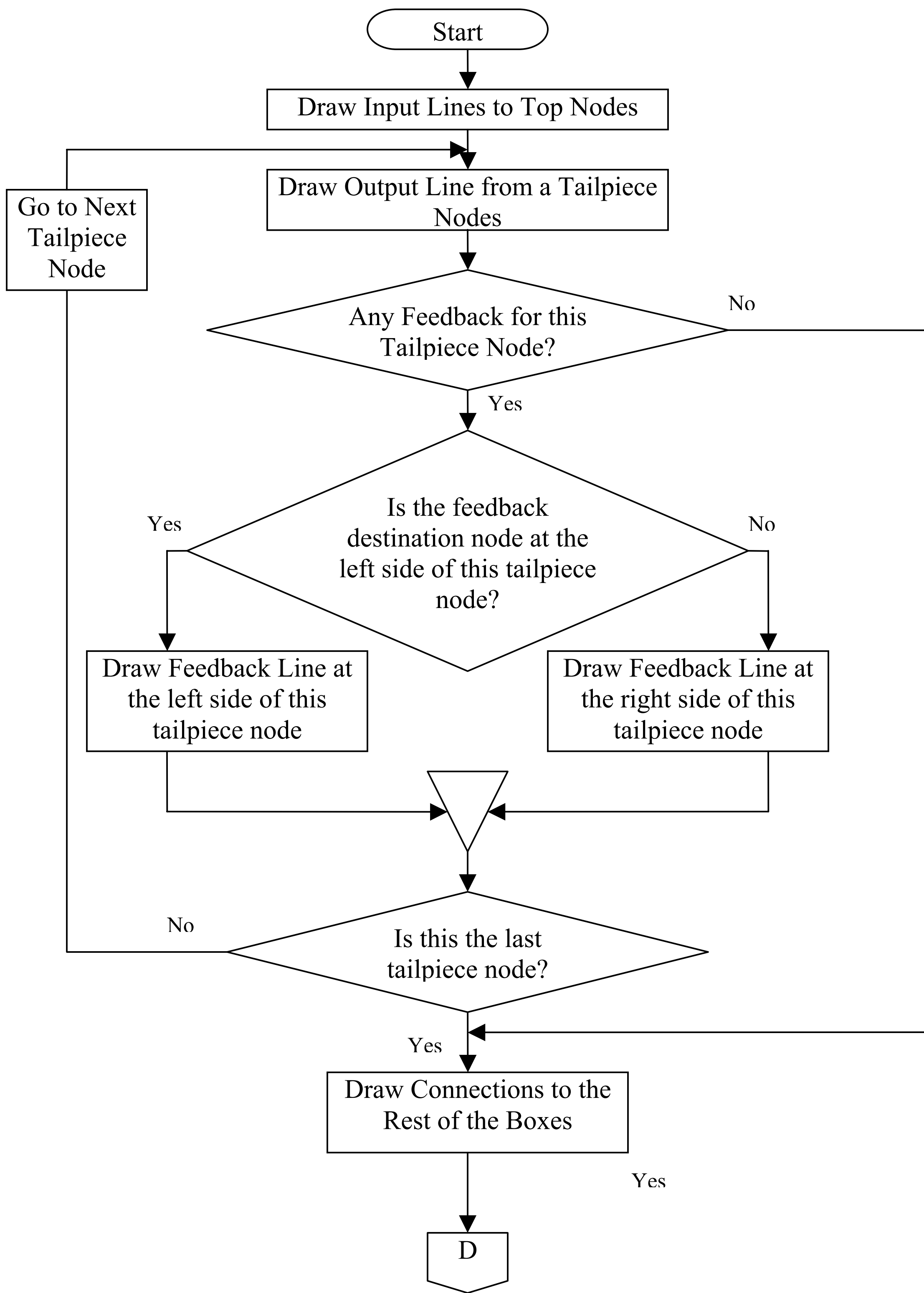


Figure 4.7 Flow Chart for "draw_link"

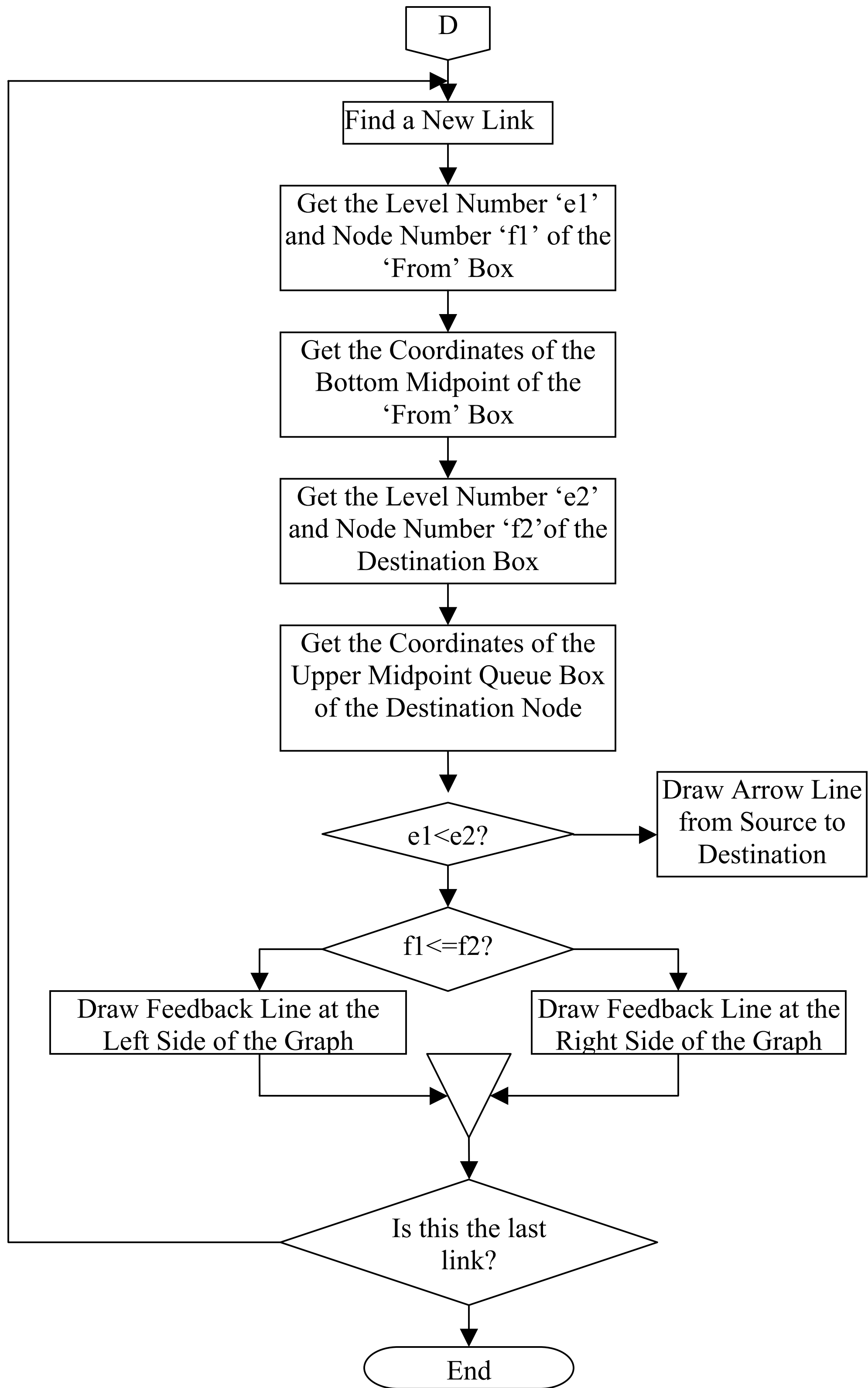


Figure 4.7 Flow Chart for “draw_link” (Continued 2)

one. If the level of source node is greater than the level of destination node, this link is a feedback. In order to draw a nice graph, the position of the destination node has to be considered. If the destination node is on the left side of the source node, draw the feedback line at the left side of the graph; if the destination node is on the right side of the source node, draw the feedback line to the right side of the graph.

c. Draw_queue

This program draws the queue for a node as the queue level starts to build up. Figure 4.8 is the flow chart for "draw_queue." This function program takes three arguments from the calling function – the plotter, the number of the level "l," and the node number "n". Each pair of (l,n) decides which node this program works on. The queue depth is stored in "queue[l][n]." Each time this function is called, it draws queue[l][n] small boxes inside the large empty queue box, representing that there are queue[l][n] data, or tokens, in the queue for node "ln." Function "pl_box_r(plotter, x1,y1,x2,y2)" draws a rectangle box starting from the corner (x1,y1) and ending at the point (x1,y1), with the point (x2,y2) as the opposite corner. Then function "pl_marker_r(plotter, x1+50, y1+12,4,4)" puts a dot inside the box. (x1,y1) of the first box is calculated according to the upper midpoint coordinates. Then (x2,y2) is calculated according to (x1,y1). After each loop, increase y1 by 25 and the plotter is ready to plot the next box for the next loop. Before drawing each box, the program checks whether the number of the queue level is greater than the queue threshold. If yes, the pen color is set to green instead of red so that the user can easily notice that the queue level exceeds the threshold and can expect a new processor to be started.

d. Clear_one_queue

This function erases one token from the queue each time one control token is removed from the queue to be executed.

e. Draw_shut

This program works on a specific copy of a node and draws a cross inside the copy box. It takes four arguments: plotter, number of level "l," number of node "n," and the number of copies for this node "cp." "Decval[l][n]" is the height of each copy for node "ln." According to the upper midpoint coordinate of the node--(a2,b2), "decval[l][n],"

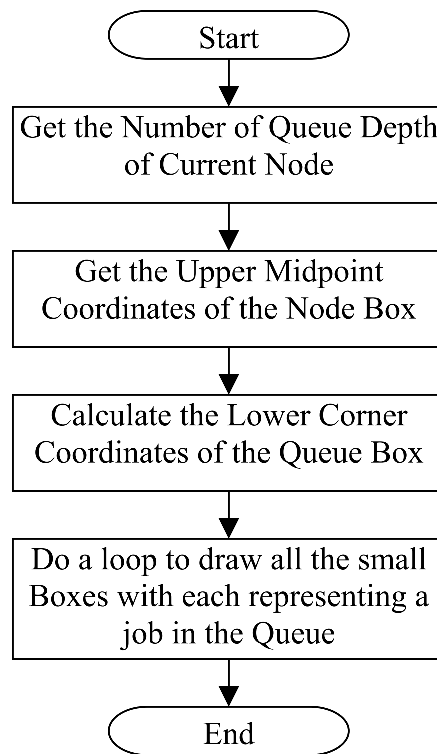


Figure 4.8 Flow Chart for "draw_queue"

and "cp," the lower left corner coordinates (x1,y1) and upper right corner coordinate (x2,y2) are calculated. Then two lines form a cross, with one line from the left lower corner to right upper corner and another line from left upper corner to the right lower corner. This means this copy is shut down.

f. Clear_shut

This function erase the cross that has been drawn by "draw_shut" when the copy is reopened.

g. Draw_busy

This program works in a very similar way except that it draws a dot inside the copy box, not a cross. Another difference is that the copy number needs to be checked to see whether it is the extra copy or not. If it is not an extra copy, then the coordinate of the dot, (x1,y1), are calculated in a similar way as above, and the dot is drawn in red. If it is

an extra copy, then (x1,y1) is calculated by a different formula, and it is drawn in green. The dot indicates that this copy is busy.

h. Clear_budy

This function erase the dot that has been drawn by "draw_busy" when the copy is not busy any more.

i. Draw_extracopy

This program is only called when the real needed number of copies exceeds the original number of copies. The extra copies are drawn on the left side of the node box in green instead of inside the node box in red, so that they can be easily noticed by the user. The program takes three arguments: plotter, the level number "l," and the node number "n." The Plotter is the same plotter that is opened during the initialization time. Pair (l,n) defines which node the program is going to work on. (a1,b1) is the coordinate of the lower midpoint of the node box. According to (a1,b1), the lower left corner coordinate (x1,y1) is calculated. Then a 200x25 GUDs rectangle box is drawn representing a copy. A loop is done for more copies. Local variable "copies" represents the number of extra copies that need to be drawn. In the real system, the resource is limited. So we set the maximum number as 4 for this value. There are only 4 extra copies available in order to reduce the queue level below the threshold.

j. Redraw

This program works together with all of above functions to update the graph each simulation cycle. Before this function is called, the full dataflow graph has been drawn on the screen. When the simulation starts, the status of each copy and the queue changes. At the end of each simulation cycle, this program is called to redraw the graph with queues, busy copies, shut down copies, or extra copies. If only dots and lines need to be added to the graph, the existing graph does not need to be erased. But if the queues, the extra copies, and the dots and the cross inside each box need to be cleared, everything on the screen has to be cleared and the whole dataflow graph needs to be redrawn again. So a flag "updateflag" is defined in the program "simulation" to indicate whether the dataflow graph on the screen needs to be totally cleared before we redraw it. At the beginning of the program, the flag "updateflag" is checked to see whether "pl_erase_r" needs to be called to clear the whole screen and redraw all the node boxes. Then all the input and

output lines are added. Then two loops are executed to check each and every one of the nodes. If a node's queue is not empty, then call "draw_queue" to draw the queue for this node. If any copy of this node is shut down, ie. "shutflag" for this copy is 1, then call "draw_shut" to draw the cross inside this copy box. If this copy is busy, ie. "busyflag" for this copy is 1, then call "draw_busy" to print a dot inside this copy box. In this way, the user can easily see which copy is busy, which copy is shut down, and how many tokens are in the queue. Figure 4.9 shows the flow chart for this program.

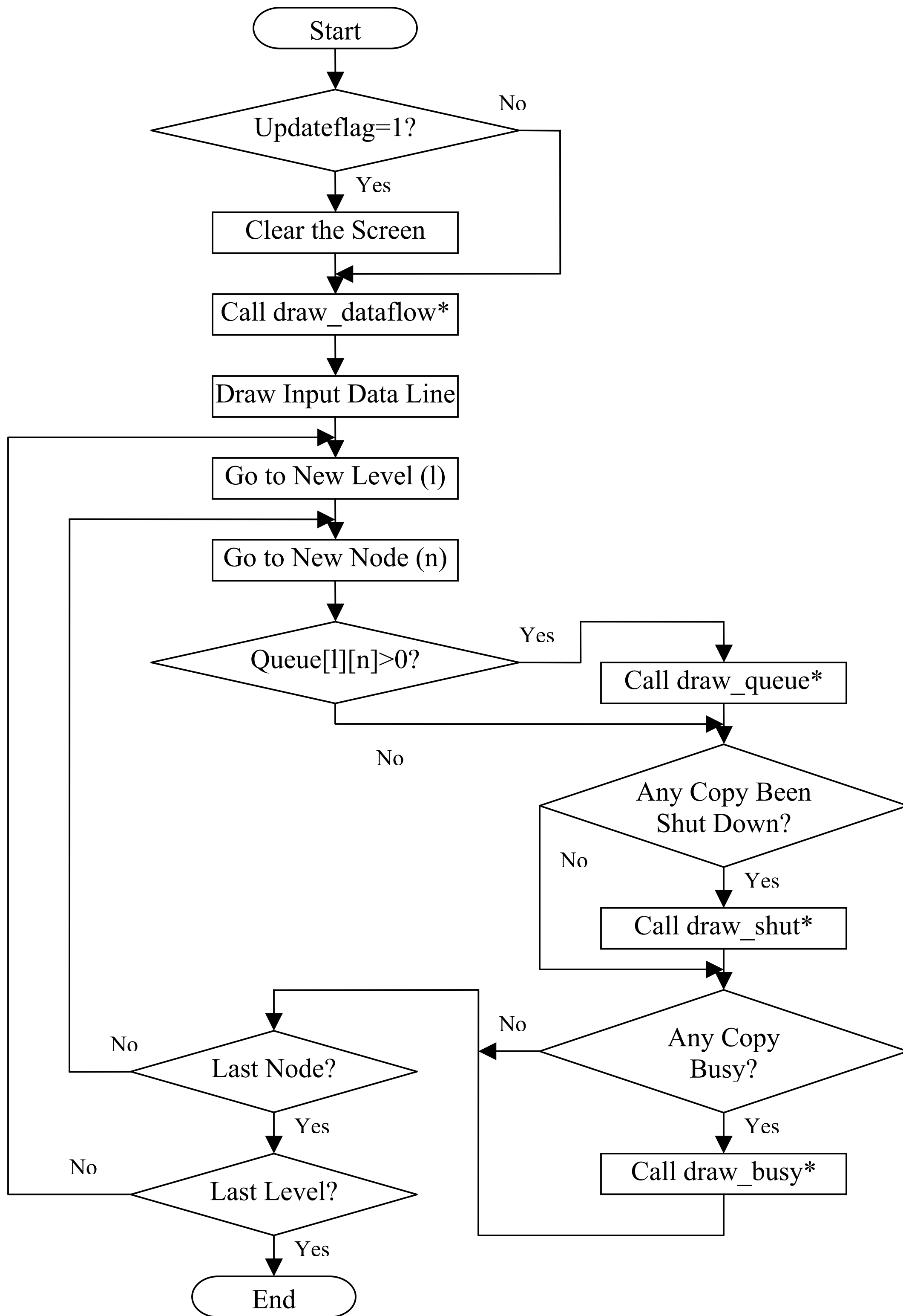


Figure 4.9 Flow Chart for Program "redraw"

4.3.2 SIMULATION MODULE (SM)

The simulation module includes only one program "simulation." The flow chart for program "simulation" is shown in Figure 4.10. At the beginning of the simulation, the program checks to see whether any processor needs to be shut down. If yes, the program will enter the Shutoff Module to shut down the processor. Then it enters the master clock loop to simulate the running of the program. The user need not input any information at this point. All the required data is read from the main program. The information required by the SM is the processing time for each node, information regarding the node, whether the node is a fork or a singular node, and the absolute probabilities associated with all the branches of that fork if the node is a fork, and number of data/tokens available at the input to the system. The rate of input will determine the interval at which the data/tokens are input into the system. For example, if the input rate is 4 data/tokens per millisecond, the SM will input data every 250 microseconds since 1000 microseconds equals 1 millisecond. Since the simulation is not real-time in nature, it is appropriate to treat each "period" of master clock as one microsecond. So for an input rate equal to 4, the SM will input data every 250 master periods. Since the processing times for the nodes are given in milliseconds, all the processing times are scaled to microseconds in the main program for the purpose of simulation. For example, if the processing time for the top node is 0.85 milliseconds and the input rate is 4, the processing time is scaled to 850 microseconds. When the simulation begins, it needs to run 850 loops in order to finish a job in the top node.

In the program, "simutime" represents the total simulation time and "inrate" represents the input rate. The "simutime" needs to be input by the user at the time of simulation. It should be in the unit of microseconds. The input rate can be either fixed or variable. If the input rate is fixed, it equals the "speedratio" that the user enters at the beginning of the simulation. But in the real system, the incoming events usually occur randomly, so the input rate is variable. In this case, a function "varate(speedratio)" will generate some random numbers around the number of "speedratio." Variable "inflag" simulates the time when a new data/token needs to be input. When the time counter equals to "inflag," "nodejob[1][1]" will increase by 1, meaning that the job for node 11 is increased by 1. Remember, the unit of the "simutime" that the user needs to input is in

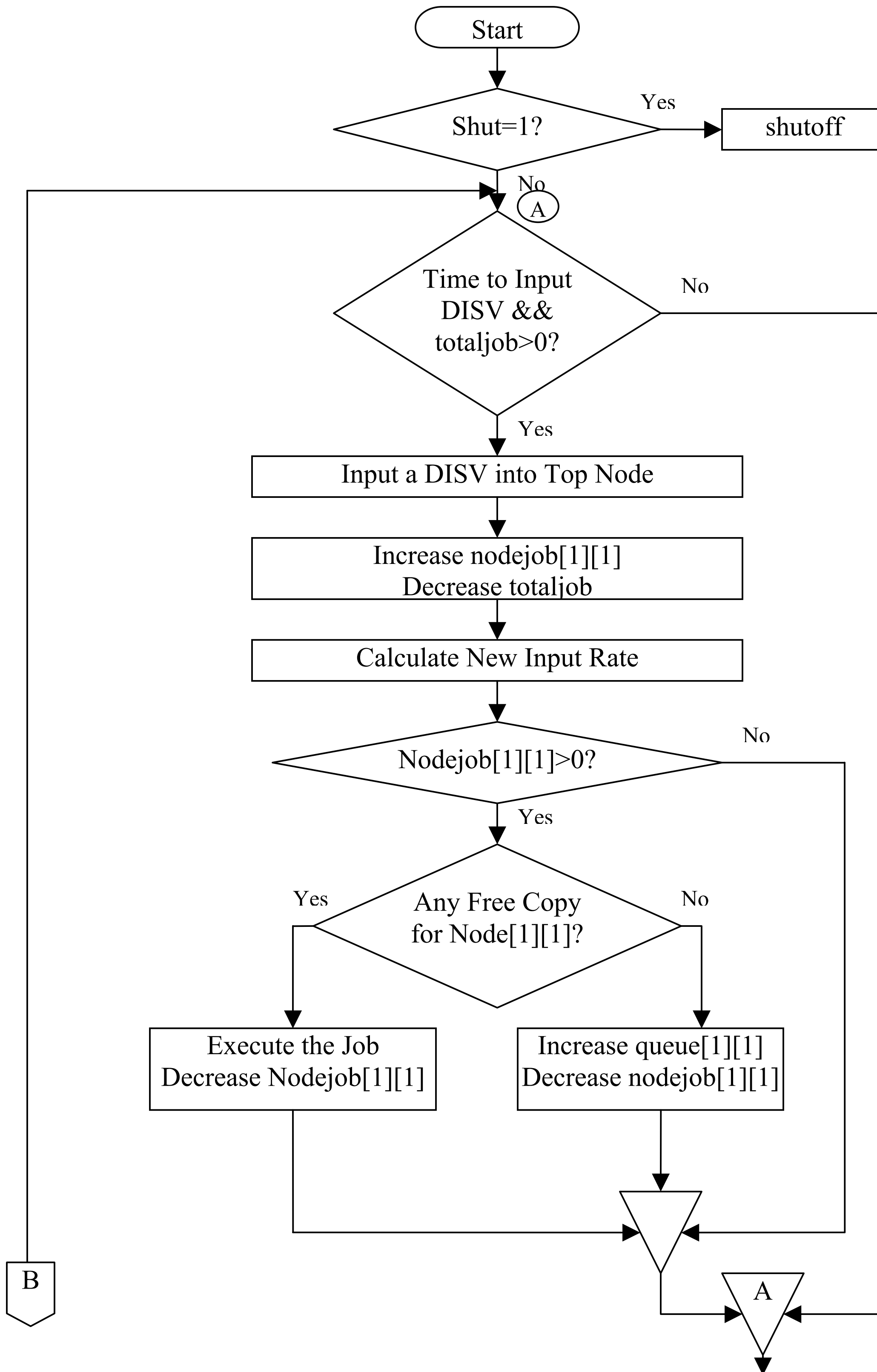


Figure 4.10 Flow Chart for Program "simulation"

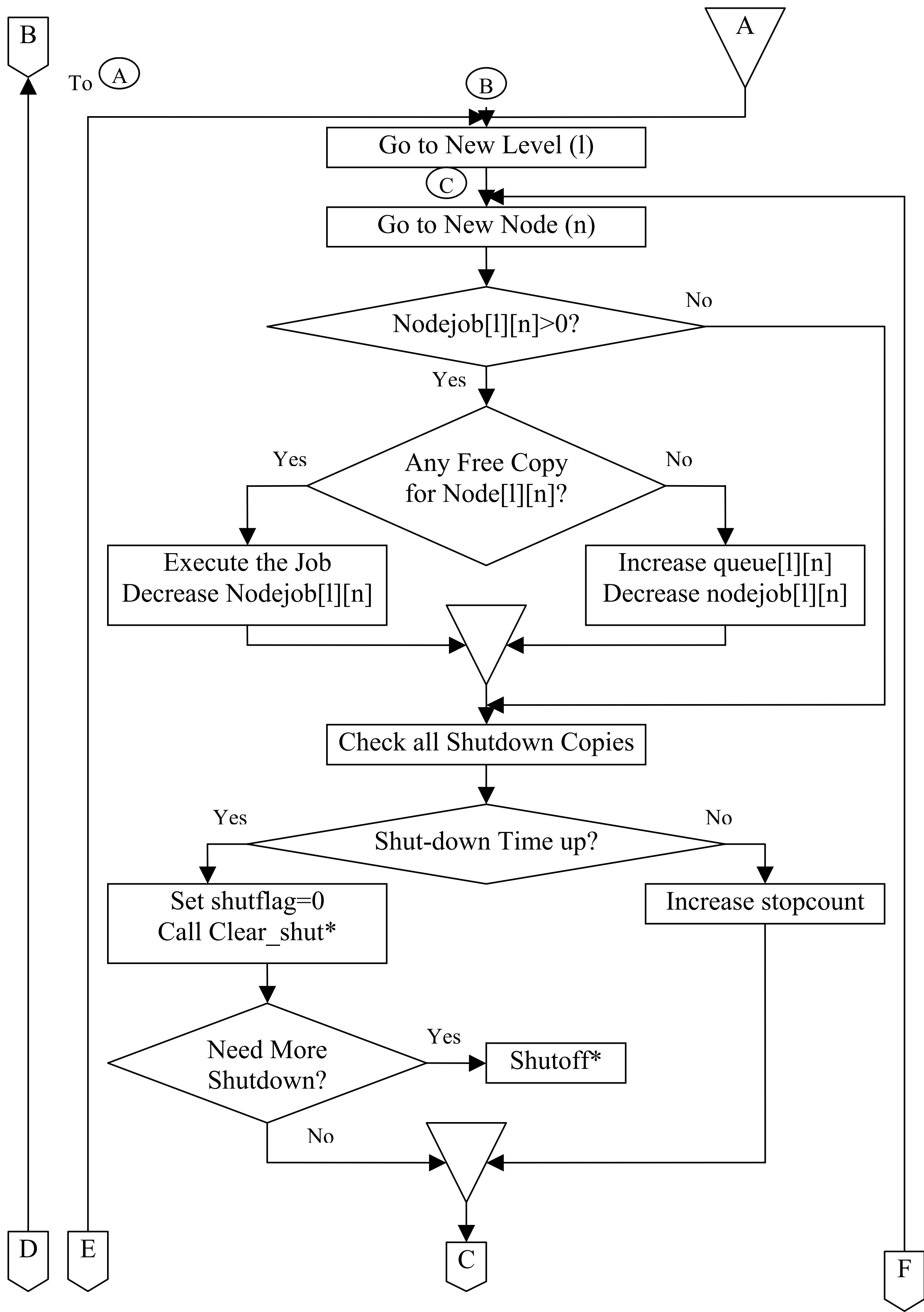


Figure 4.10 Flow Chart for Program "simulation" (Continued 2)

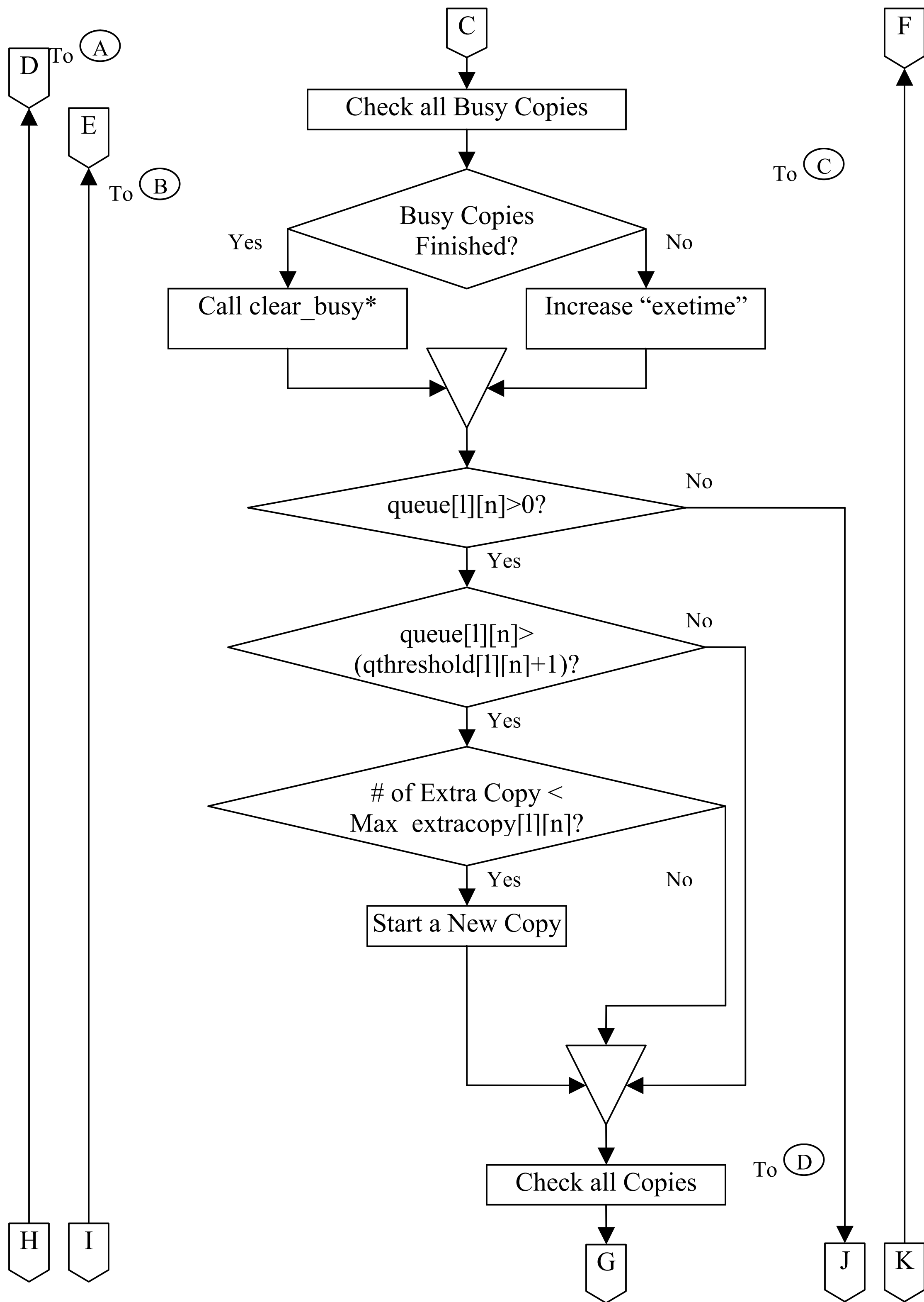


Figure 4.10 Flow Chart for Program "simulation" (Continued 3)

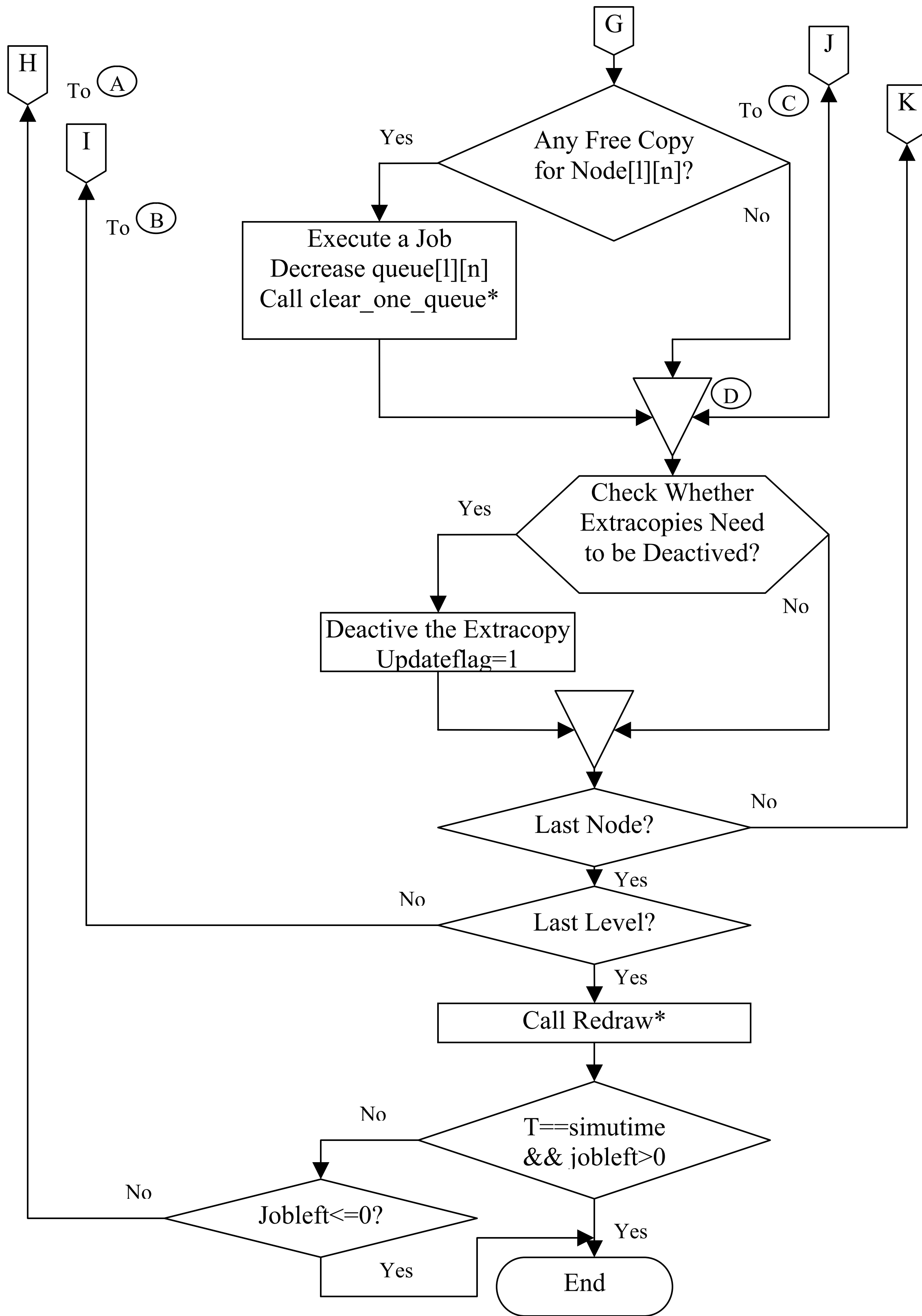


Figure 4.10 Flow Chart for Program "simulation" (Continued 4)

microseconds, and the unit of "inrate" is in microseconds. After checking the input time, the program enters a two level loop to check each and every node. For example, for node "ln," if there is a newly incoming job, that is "nodejob[l][n] >0," the program will do a small loop to check whether any free copy is available. If yes, it will set the "busyflag"of this copy to 1 indicating this copy is occupied by the job and it will set the "exetime" to zero meaning that the execution just started. If not, put the job in the queue of this node. In both cases, "nodejob[l][n]" needs to be decremented by 1 meaning that this new job has been processed, either being executed, or waiting in the queue. After processing the new job, the program does another loop to check the status of all the copies. If any copy is shut down for the time being, then check the "stopcount" to see how long this copy has been shut down. If the time is not enough, then increase the "stopcount" by 10, or reopen the copy by setting the "shutflag" to zero and ask the user whether more copies need to be shut down. If any copy is busy, then check the "exetime" to see whether the executing time equals to the processing time for node "ln." If yes, enter the update module to route the data and update the status of the copy. If not, increase the "exetime" by 1. Then another loop is executed to check the queue for this node. If the queue depth exceeds the queue threshold and the number of extra copies is less than the maximum number of extra copies, then increase the queue threshold by 2 and start a new copy of the processor to help to process all the jobs in the queue starting from the next cycle. If the queue depth is below the threshold, no extra copies are needed, so the program checks all copies to see whether free copies are available to execute the job. Each free copy executes one job from the queue, and the queue decreases by 1. After checking the queue, the program needs to check whether the extra copies that are started earlier need to be deactivated. If the queue level falls back below the queue threshold, then deactivate one extra copy and decrease the queue threshold by 2. This ends the loop of checking all of the nodes of the dataflow graph. At this time, the status of the copies and the queue levels for all the nodes might be different from the last cycle, so "redraw" is called to redraw the dataflow graph on the screen. The last step in one master clock cycle is to check "jobleft," the number of jobs that have not been finished. If "jobleft" is greater than zero but the simulation time has reached the number that the user assigned, then the simulation time is not enough. If

"jobleft" is zero but the simulation time has not reached the number, then finish the simulation immediately.

4.3.3 UPDATE MODULE

The update module has only one program: "update( )." It takes four arguments from the calling function: "plotter," the level number "l," the node number "n," and the copy number "cp." The update module is embedded in the SM and is called only when a job is finished at a node. Figure 4.11 shows the flow chart for "update". The local variable "word" represents whether the node is a fork, a singular node or a tailpiece. In case of a singular node, or "word" equals to zero, the program searches all the links to find the subsequent node where the data should go. Once the subsequent node is found, increase the "nodejob" for that node and sets copy "cp" of the current node to free. Meanwhile, draw an arrow line from the lower midpoint of the current node box to the queue of the subsequent node. If the current node is a fork, things became more complicated. Since the type of fork used for the modeling is a selective fork, the data output after being processed by the fork is directed to only one of the branches associated with the fork. So we need to have a mechanism to find out which branch the data should go to. In this program, function "random" is used to generate a random number. If the accumulative probability associated with a branch is greater than this random number, then route the data to this branch by increasing the "nodejob" of the subsequent node of this branch by one, and draw an arrow line from the current node to the next node indicating the data flow direction. If this branch is a feedback, then the position of this arrow line depends on the position of the destination node. If the destination node is on the left side of the current node, then draw the feedback line at the left side of the screen. If the destination node is on the right side of the current node, then draw the feedback line at the right side of the screen. After directing the data to the subsequent node, set the copy "cp" of the current node free by changing "busyflag" of this copy to 0 and clearing the execution time. If "word" equals to 2, this means the current node is the tailpiece. In this case, just free the copy "cp" and clear the execution time "exetime" for this copy. Then decrease "jobleft" by one, indicating that one job just finished.

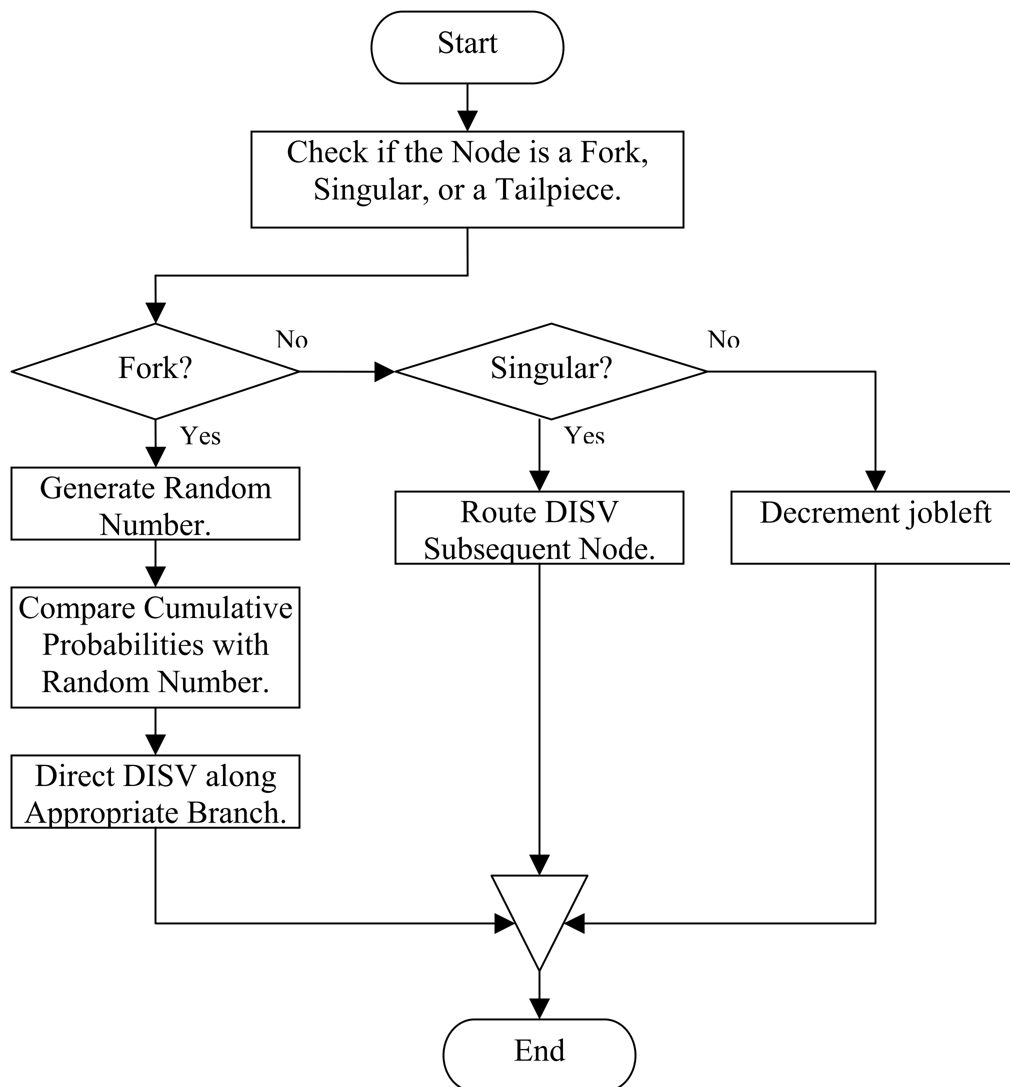


Figure 4.11 Flow Chart for Program "update"

4.3.4 SHUTOFF MODULE

This module is used to shut down any copy of a processor and to do fault tolerance analysis. "Shutoff" is the only program in this module. Each time this program is called, the user needs to input the level number, node number and copy number of the node that the user wants to shut down. Then the copy is "shut down" simply by setting the "shutflag" to 1, and meanwhile set "stopcount" of that copy to 0 meaning the beginning of the "shut down." The shut down time should be the integer number times of the processing time of that node. This integer number should also be inputted by the user at the beginning of this module. After shut down of a specific copy, the user can choose to shut down more copies, or go back to the simulation mode.

4.4 Input and Output File Format

The program "hdca" can get all needed data either from a keyboard or from a series of files. If the program is run in the auto input environment, it needs to read data from 5 files: "dataset," "labelset," "informationset," "timeset," and "probabilityset." "Labelset" and "timeset" are generated by the program "COPY." So we need to run program "COPY" before we start the simulation. In order to run "COPY," file "copyset" is needed if the program "COPY" needs to run in the auto input environment. Let's take the example that we described in 3.2 and Figure 3.7 to see how these files are comprised.

Copyset:

3
7 1 1 2 1 3 1 4 1 5 1 6 1 7 1
7 1 1 2 1 3 2 4 1 5 1 6 1 7 1
5 1 1 3 3 5 1 6 1 7 1
0.85 0.425 1.63 2.5 1.3 0.5 0.96 0.85 1.87 0.45 0.69 15.0 1.12 0.9 0.32 0.05 2.7 1.5
16 3.8 2.5 1 0.35 0.35 0.65 0.8 0.2 1

Labelset:

4 3 2 1 7 2 8 3 5

Dataset:

7 1 1 3 1 1 1 1 1 1 2 1 2 1 3 1 3 1 4 1 4 1 5 1 5 1 6 1 6 1 7 1 2 1 3 2
3 2 4 1 1 1 3 3 3 3 5 1 0 0 0 0

Informationset:

1 1 0 0 0 0 0 0 2

1

Timeset:

0.850000
1.630000
1.300000
0.320000
2.700000
0.960000
1.870000
0.690000
1.120000

Probabilityset:

0.35 0.65 0.8 0.2

For "copyset," the first number represents how many pipelines are in the graph. For Figure 3.7, there are 3 pipelines altogether. Then the following three groups represent three pipelines, with the first number indicating the total number of nodes in the pipeline and the rest of the numbers show the nodes in the pipeline. Each node is represented by two numbers, with the first number representing the level number and the second number representing the node number in the level. For example, node A is represented by 11, and node D is indicated by 32. The following numbers are the execution time in milliseconds and process length in kilobytes for each and every node in the dataflow graph. The beginning of the last line indicates the memory size that is available for the process in kilobytes, maximum input rate, and average input rate. After that, "1" indicates that the fork needs to be considered, so the following 5 numbers are the absolute probabilities for all the fork branches in the graph. The last number indicates whether the combinable processes are desired. "1" means "yes," and "0" means "no."

In "labelset," the numbers represent the number of copies that the node needs for the "clog" free simulation. They are in the order of the nodes P_{11}, P_{21}, P_{31}, P_{32}, P_{33}, P_{41}, P_{51}, P_{61}, and P_{71}.

In "dataset," the first number "7" is the total number of levels in the graph. The following 7 numbers are the number of nodes in these 7 levels, starting from the first level to the 7th level. Starting from the 9th number, every four digits form a link. For example, "1 1 2 1" means a link from P_{11} to P_{21}. The last four zeroes terminates the input of the links.

In "informationset," "1" means the node is a fork, "0" means the node is a singular node, and "2" means the node is a tailpiece. So, in this case, P_{11} and P_{21} are forks, node P_{71} is a tailpiece and all other nodes are singular nodes.

In "timeset", the numbers are the processing time for P_{11}, P_{21}, P_{31}, P_{32}, P_{33}, P_{41}, P_{51}, P_{61}, and P_{71}. They are in milliseconds.

In "probabilityset," they are the probabilities from P_{11} to P_{21}, from P_{11} to P_{33}, from P_{21} to P_{31}, and from P_{21} to P_{32}.

4.5 How to run HDCA?

The program "hdca" is designed to run under the Unix operating system. In order to run this program, the user needs to have the discussed 5 files (dataset, labelset, informationset, timeset, and probabilityset) ready in the directory. And the GNU's "libplot" library should also be installed in the system. The "libplot" library is contained in the GNU Plotutils Package. The package is free software. Its source code is distributed as a gzipped tar file, 3.3 megabytes in size. It can be installed on GNU/Linux, FreeBSD, and Unix systems. Go to this website to find how to get the software.

http://www.gnu.org/software/plotutils/

How to compile?

Libplot was installed in the dirctory "gnulib" in an user account "czhen2", then the following command was used to link the library when the program was compiled:

Gcc hdca.c –L/homes/czhen2/gnulib/lib –lplot –I/homes/czhen2/gnulib/include –o hdca

After successful compilation, an executable file "hdca" was generated. Then "hdca" was typed to run this program. Some parameters need to be inputed interactively as the program is running. Table 4-2 contains the print out of an actual run of program "hdca." The dynamic simulation graph will be introduced in Chapter 5.

Table 4-3: Example Run of Program "hdca"

czhen2@lily:~/thesis> hdca (Enter)

If keyboard entry is desired set KBENTRY to 1,

and for 'read' from file, set KBENTRY to 0 *1* (Enter)

KBENTRY=0

NUMBER OF LEVELS =7

Level 1 has 1 nodes

Level 2 has 1 nodes

Level 3 has 3 nodes

Level 4 has 1 nodes

Level 5 has 1 nodes

Level 6 has 1 nodes

Level 7 has 1 nodes

Link:11-->21

Link:21-->31

Link:31-->41

Link:41-->51

Link:51-->61

Link:61-->71

Link:21-->32

Link:32-->41

Link:11-->33

Link:33-->51

Total 10 links

Node11 needs 4 copies!

Node21 needs 3 copies!

Node31 needs 2 copies!

Node32 needs 1 copies!

Node33 needs 7 copies!

Node41 needs 2 copies!

Node51 needs 8 copies!

Node61 needs 3 copies!

Node71 needs 5 copies!

The cumu probability for link node[1][1]->node[2][1]is 0.350000

The cumu probability for link node[1][1]->node[3][3]is 1.000000

Information[1][1].fork=1 processtime=850

The cumu probability for link node[2][1]->node[3][1]is 0.800000

The cumu probability for link node[2][1]->node[3][2]is 1.000000

Information[2][1].fork=1 processtime=1630

Information[3][1].fork=0 processtime=1300

Information[3][2].fork=0 processtime=320

Information[3][3].fork=0 processtime=2700

Information[4][1].fork=0 processtime=960

Information[5][1].fork=0 processtime=1870

Information[6][1].fork=0 processtime=690

Information[7][1].fork=2 processtime=1120

Input the total number of jobs at the topnode:40

Enter the average speedratio of the input jobs at the topnode:***200*** (Enter)

Enter the total simulation time(in mirco cycles):***30000*** (Enter)

Would you like to shut down any of CE copies?1-yes, 0-no***0*** (Enter)

If variable input rate is desired, set varyinrate to 1, or 0!

1 (Enter)

No DISV's at the input. Execution interrupted.

Change the number of input jobs if desired, and run the program again!

Total simulation time is 17172 micro cycles.

totaljob=0 jobleft=0 inrate=192 simutime=30000 shut=0

5 SIMULATION RESULTS

As mentioned in earlier chapters, the program “hdca” can simulate any given dataflow graph, including both cyclic and acyclic graphs. In this section, three applications are presented as a demonstration of the application capabilities of the program in dynamically simulating the dataflow’s running on HDCA. The first application is the example used in Chapter 4. It is adapted from the example radar problem presented in [14]. The second application is a two input dataflow graph found in [7]. The last one is a cyclic dataflow graph also found in [7].

5.1 Application 1

This simulation is conducted on the dataflow shown in Figure 3.7. The dataflow graph is redrawn in Figure 5.1. The nodes originally represented by circles are now represented by square boxes. The α parameters and input files are shown in Table 5-1 and Table 5-2. The rate of input will determine the interval at which the Data Item or tokens are inputted into the system. For example, the data input rate at peak load is 3.8 data items/millisecond, and the average data input rate is 2.5 data items/millisecond. So the SM inputs a data/token every 263 (1000/3.8) microseconds at peak load, and every 250(1000/4) microseconds at average load. According to these input rates and α parameters, the program “COPY” first calculates the number of copies that each and every node needs for a queue-free simulation and then saves the results in the file “labelset”. The program “hdca” reads the number of copies from file “labelset” and treats those numbers as the initial number of copies that the node needs to do the simulation. When the input rate is less than the peak load input rate of 3.8 tokens/millisecond, or 1000/3.8=263 microseconds /token, the program “hdca” should be able to produce a queue-free simulation graph without having to initiate an extra copy of a processor for each and every node. When the input rate increases, the queue level will also increase. When the input rate is high, the queue level at some nodes will exceed the threshold, and extra copies will be needed. Simulation has been conducted at three different input rates: peak load input rate of 263 microseconds/token, slightly higher than the peak load input rate of 200 microseconds/token, and a very high input rate of 100 microseconds/token to study whether the simulator can simulate the architecture correctly.

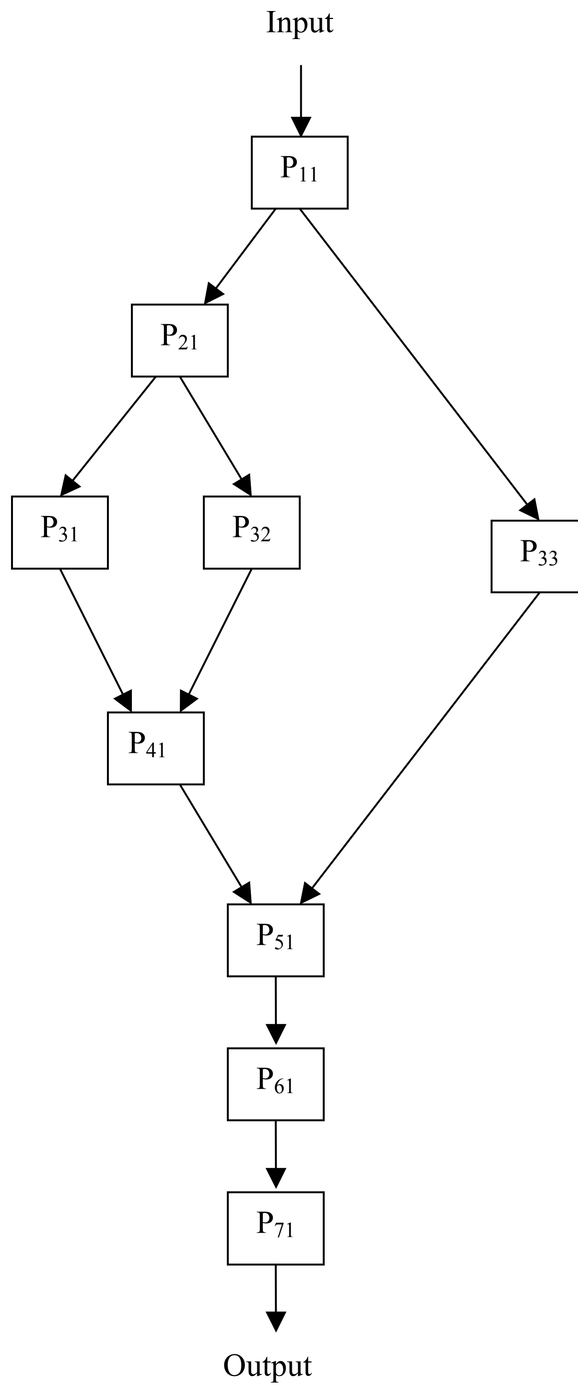


Figure 5.1 Dataflow Graph of Application 1

Table 5-1: Parameter Values for Data Flow Graph of Application 1

Process Designation	Execution Time (milliseconds)	Process Length (Kilobytes)
11	0.85	0.425
21	1.63	2.5
31	1.3	0.5
32	0.32	0.05
33	2.7	1.5
41	0.96	0.85
51	1.87	0.45
61	0.69	15.0
71	1.12	0.9

Input Data Rates (data items/millisecond)

Peak Load: 3.8

Average Load: 2.5

Probability Distributions for Forks

$P_{11\text{->}33}$: 0.65

$P_{11\text{->}21}$: 0.35

$P_{21\text{->}32}$: 0.2

$P_{21\text{->}31}$: 0.8

Program Memory/Computing Element: 16 kilobytes

Table 5-2: Input Files for Application 1

Copyset:

3 7 1 1 2 1 3 1 4 1 5 1 6 1 7 1 7 1 1 2 1 3 2 4 1 5 1 6 1 7 1 5 1 1 3 3 5 1 6 1 7 1 0.85 0.425 1.63 2.5 1.3 0.5 0.96 0.85 1.87 0.45 0.69 15.0 1.12 0.9 0.32 0.05 2.7 1.5 16 3.8 2.5 1 0.35 0.35 0.65 0.8 0.2 1

Dataset:

7 1 1 3 1 1 1 1 1 1 2 1 2 1 3 1 3 1 4 1 4 1 5 1 5 1 6 1 6 1 7 1 2 1 3 2 3 2 4 1 1 1 3 3 3 3 5 1 0 0 0 0

Informationset: 1 1 0 0 0 0 0 0 2

Figure 5.2 shows an actual graphical run of the program “hdca” at the input rate of 263 microseconds/token, and there are 20 Data Items (or control tokens) at the top node for case1. Figure 5.1 a was captured at 1000 Micro-cycles, b was captured at 2000 Micro-cycles, and so on. Figure 5.3 shows an actual graphical run of the program at the input rate of 200 microseconds/token and with the same amount of data items at the input nodes. Figure 5.4 shows the graphical simulation results at a very high data input rate, 100 microseconds/token and with the same amount of tokens at the top node. SM assumes that each cycle of the simulation will take 1 microsecond, so we treat the time in “microcycles.” For example, if the total simulation time is 10000 microseconds, the SM will simulate the program for 10000 cycles. Form now on, we call this cycle “microcycles,” and it is treated as the unit of the processing time. Pictures in Figures 5.2, 5.3, 5.4 and those pictures for application 2 and application 3 are taken every 1000 “microcycles.”

By looking at Figure 5.2, there are 3 tokens in node 11 and 1 token in node 33 at 1000 Micro-cycles (Shown by Figure 5.2a). At 2000 Micro-cycles(Shown by Figure 5.2b), there are 3 tokens in node 11, 1 token in node 21, and 4 tokens in node 33. By the time of 6000 Micro-cycles, all of 20 token have been scattered to nodes 31, 32, 33, 41, 51, 61, and 71. There is one token stored in the queue of node 33. As time goes on, the data items that have been finished will get out of the graph from the node 71 (“tailpiece”).

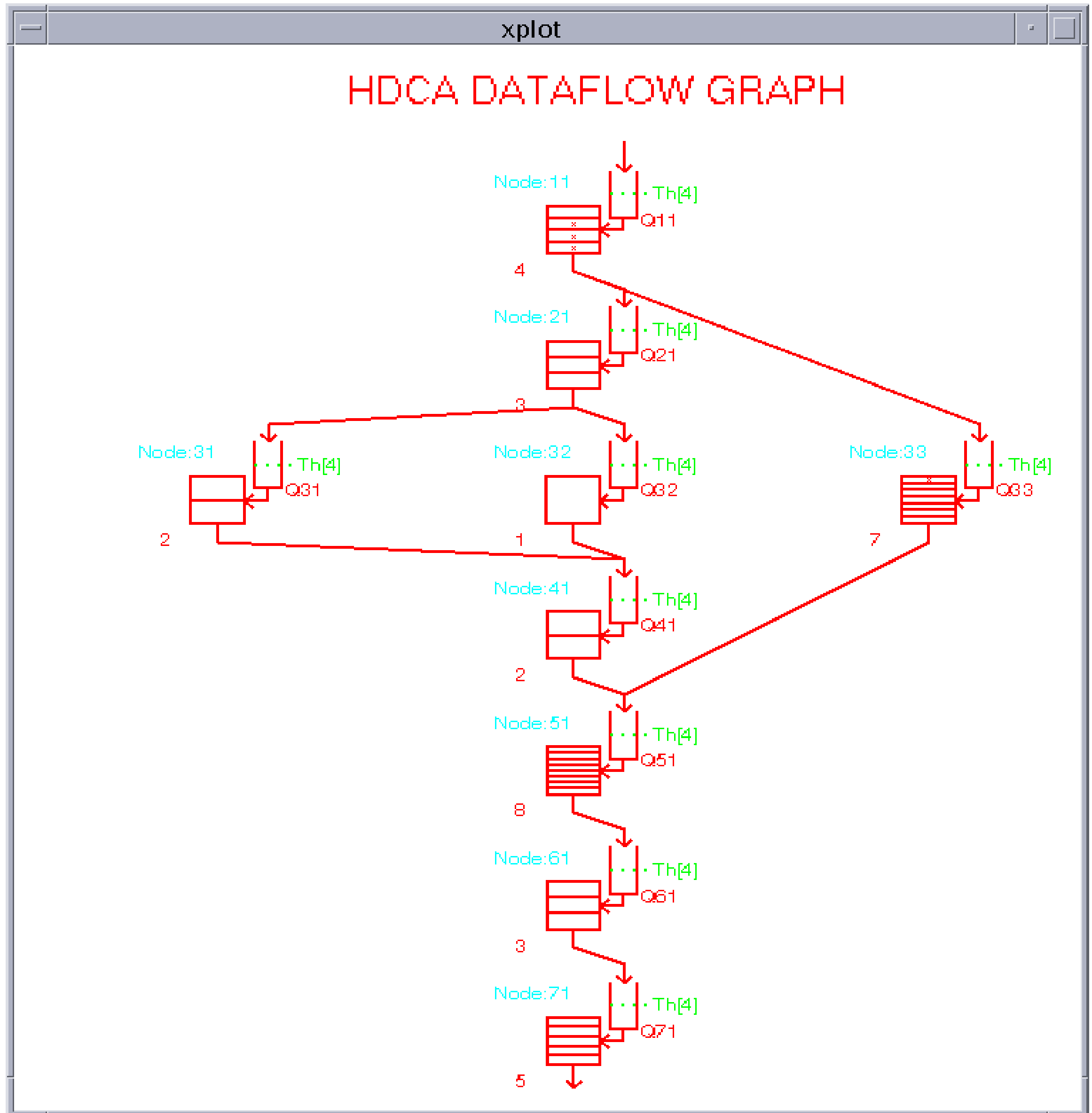


Figure 5.2 a   Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t =1000 micro-cycles)

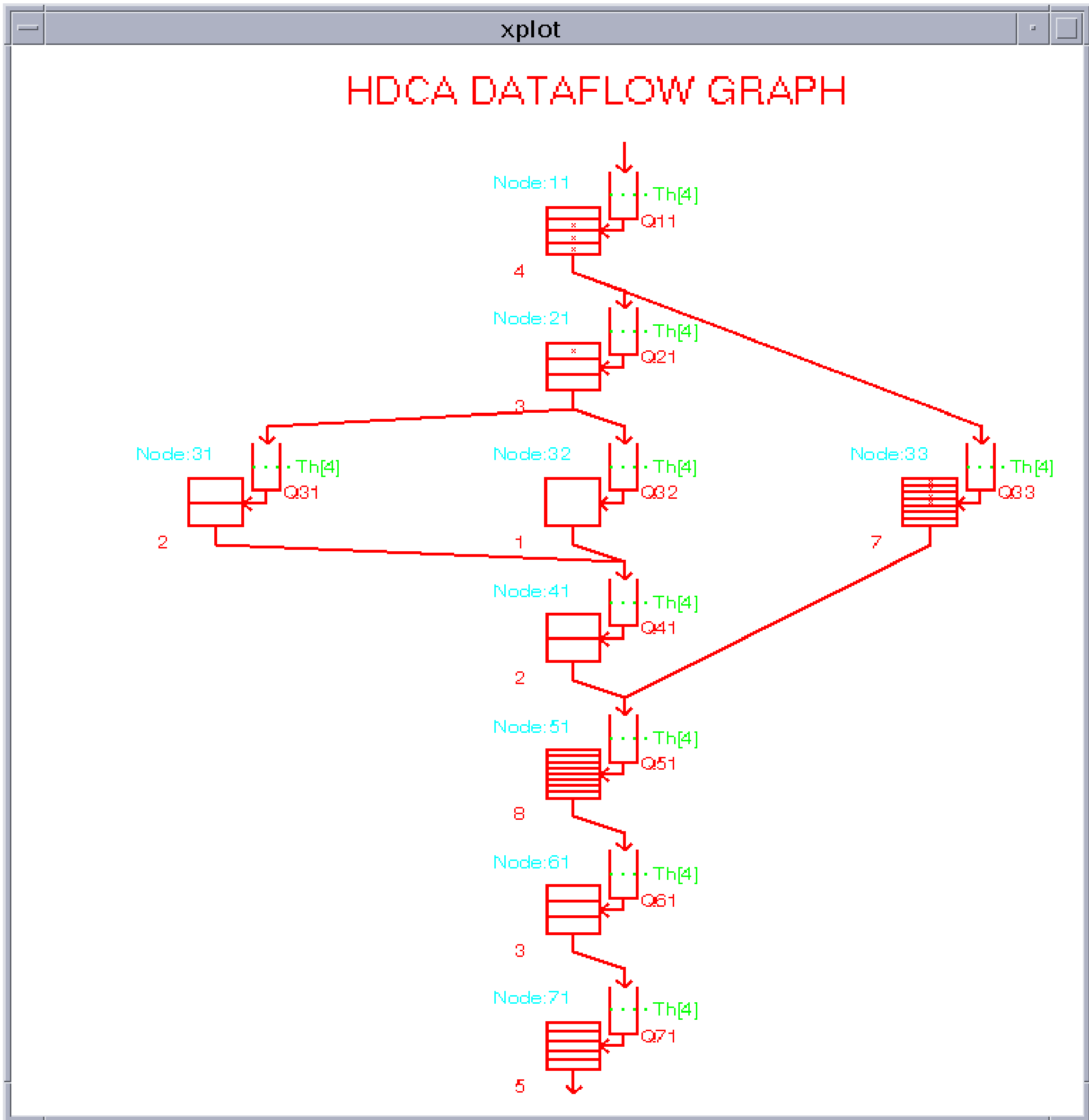


Figure 5.2 b Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=2000 micro-cycles)

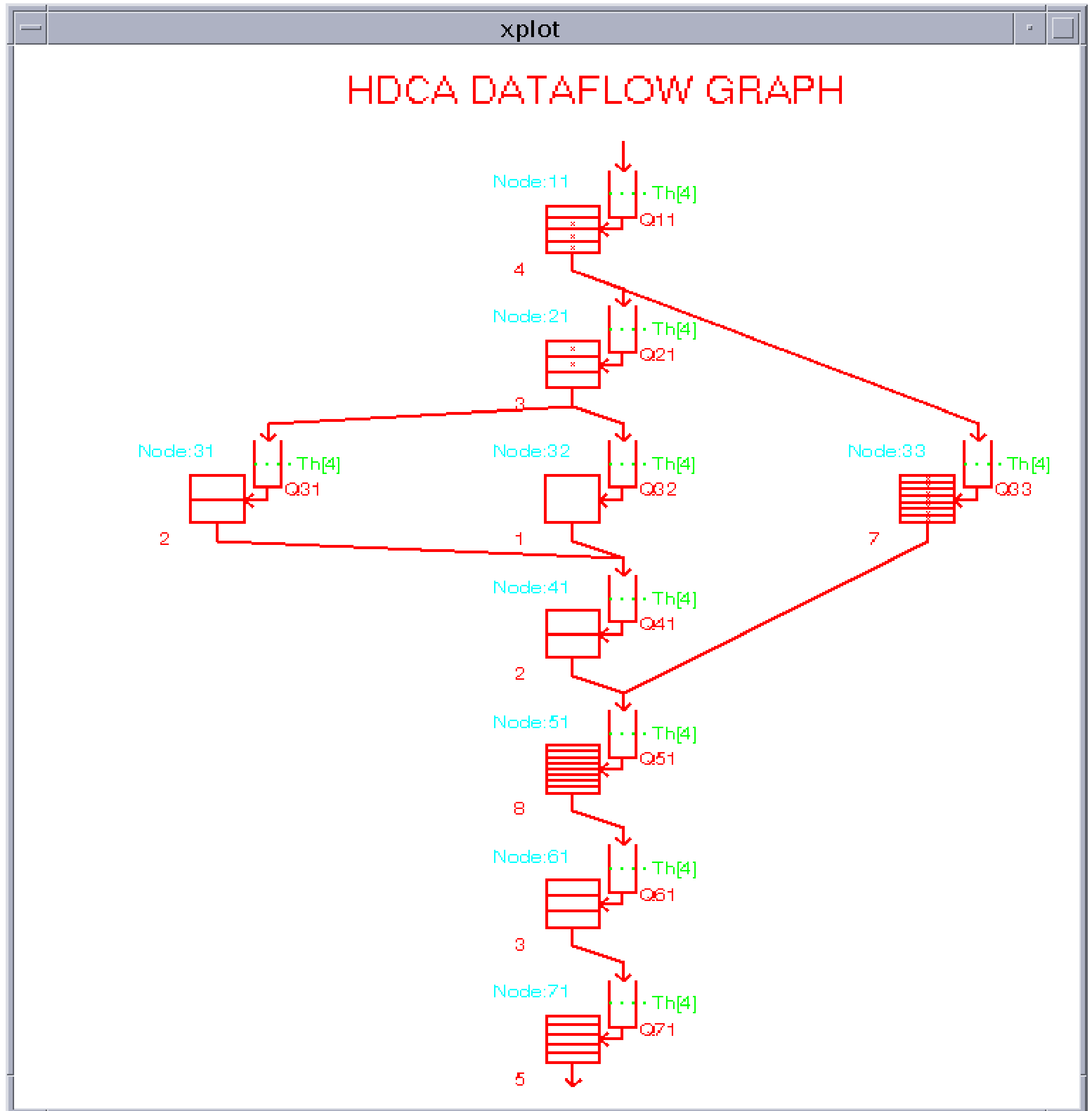


Figure 5.2 c Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=3000 micro-cycles)

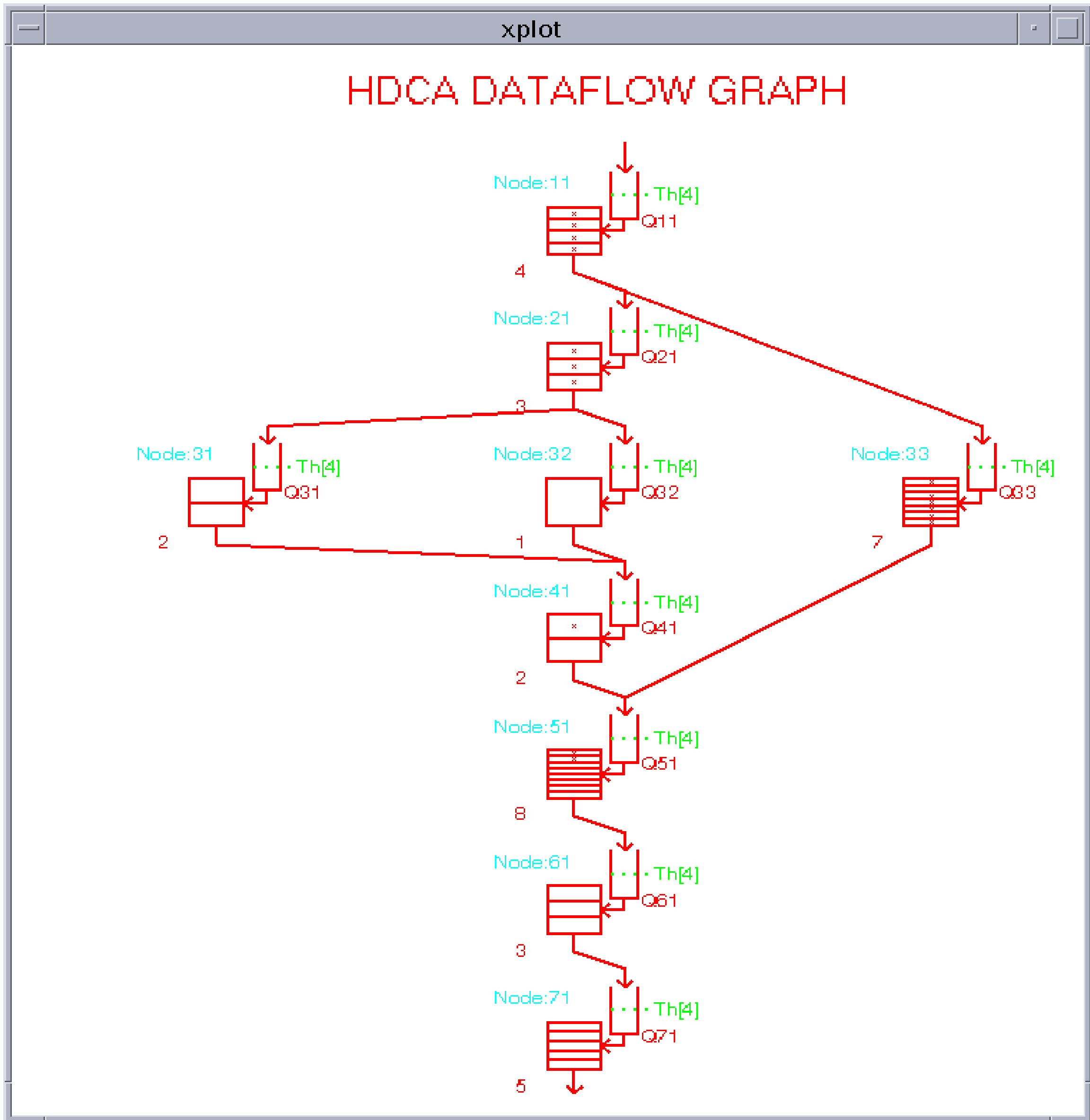


Figure 5.2 d Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=4000 micro-cycles)

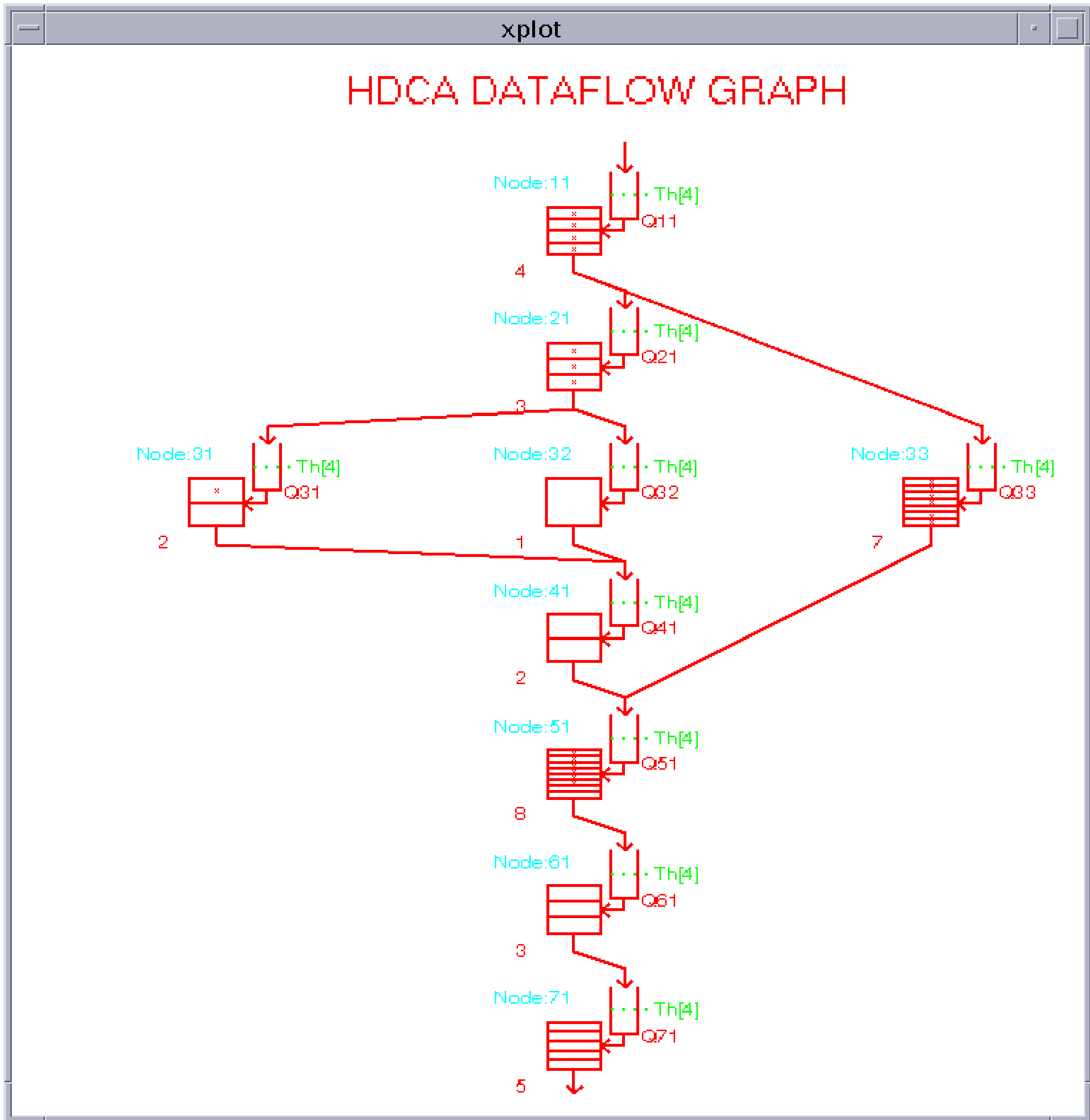


Figure 5.2 e Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=5000 micro-cycles)

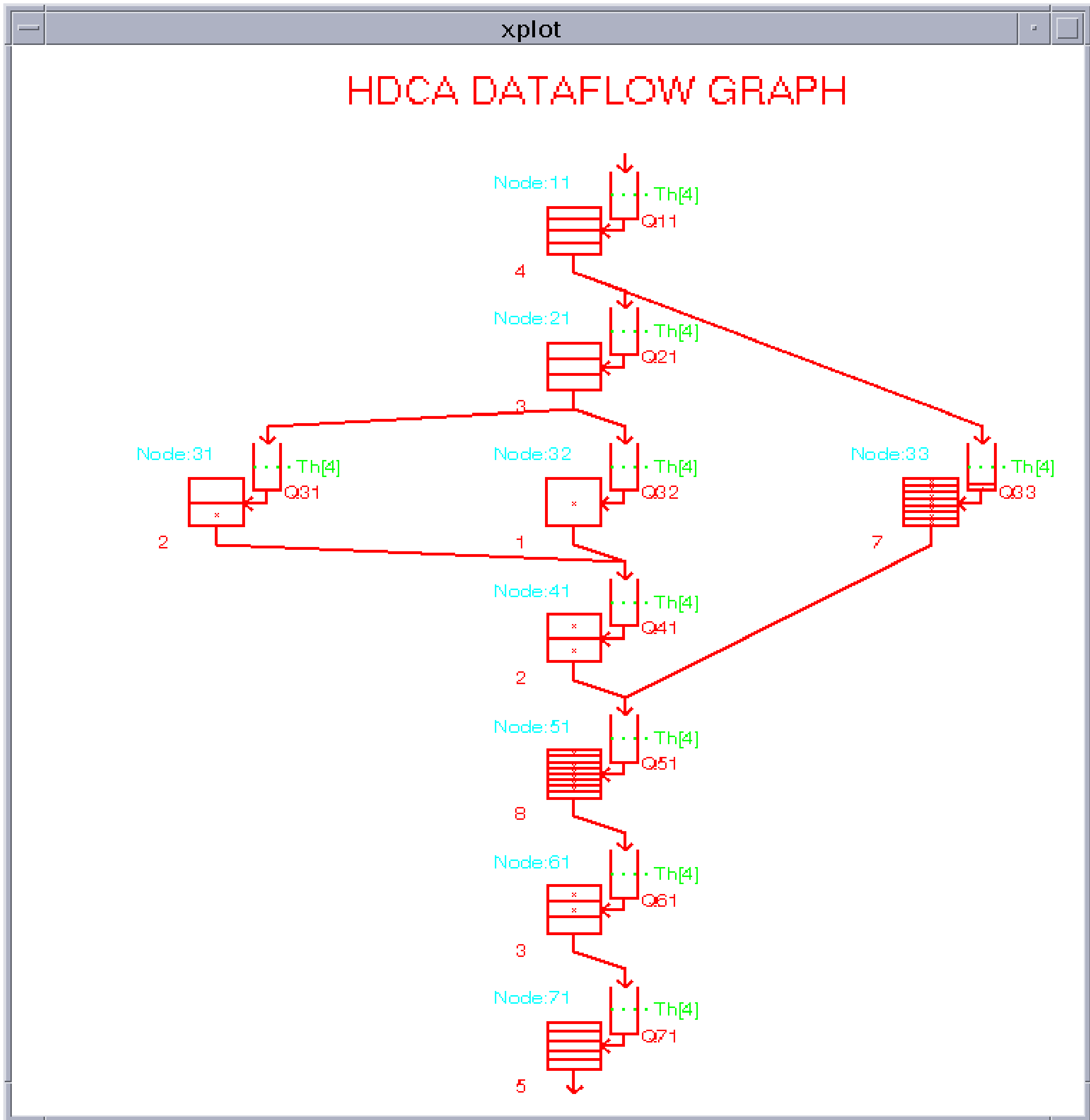


Figure 5.2 f Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=6000 micro-cycles)

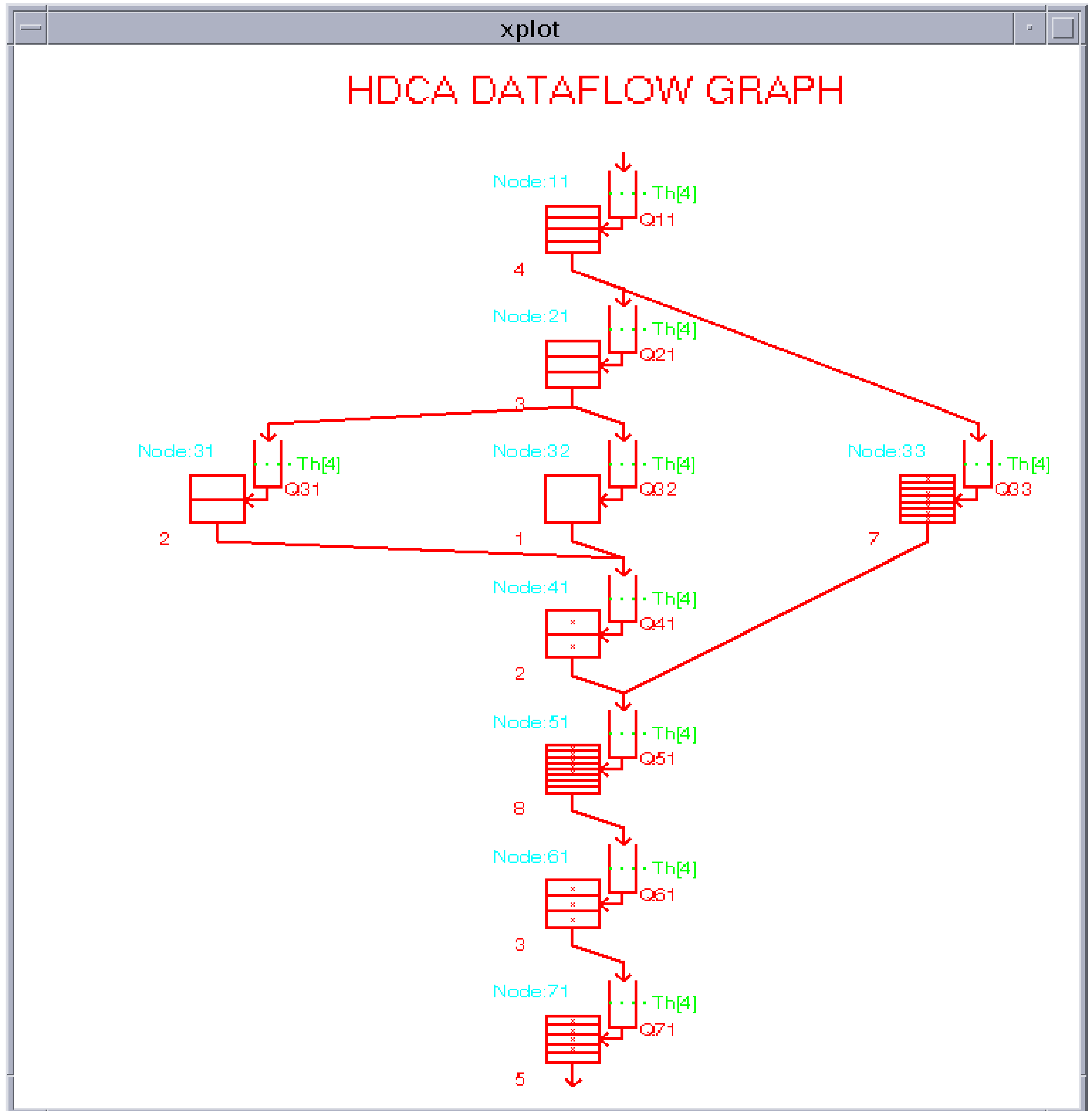


Figure 5.2 g Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=7000 micro-cycles)

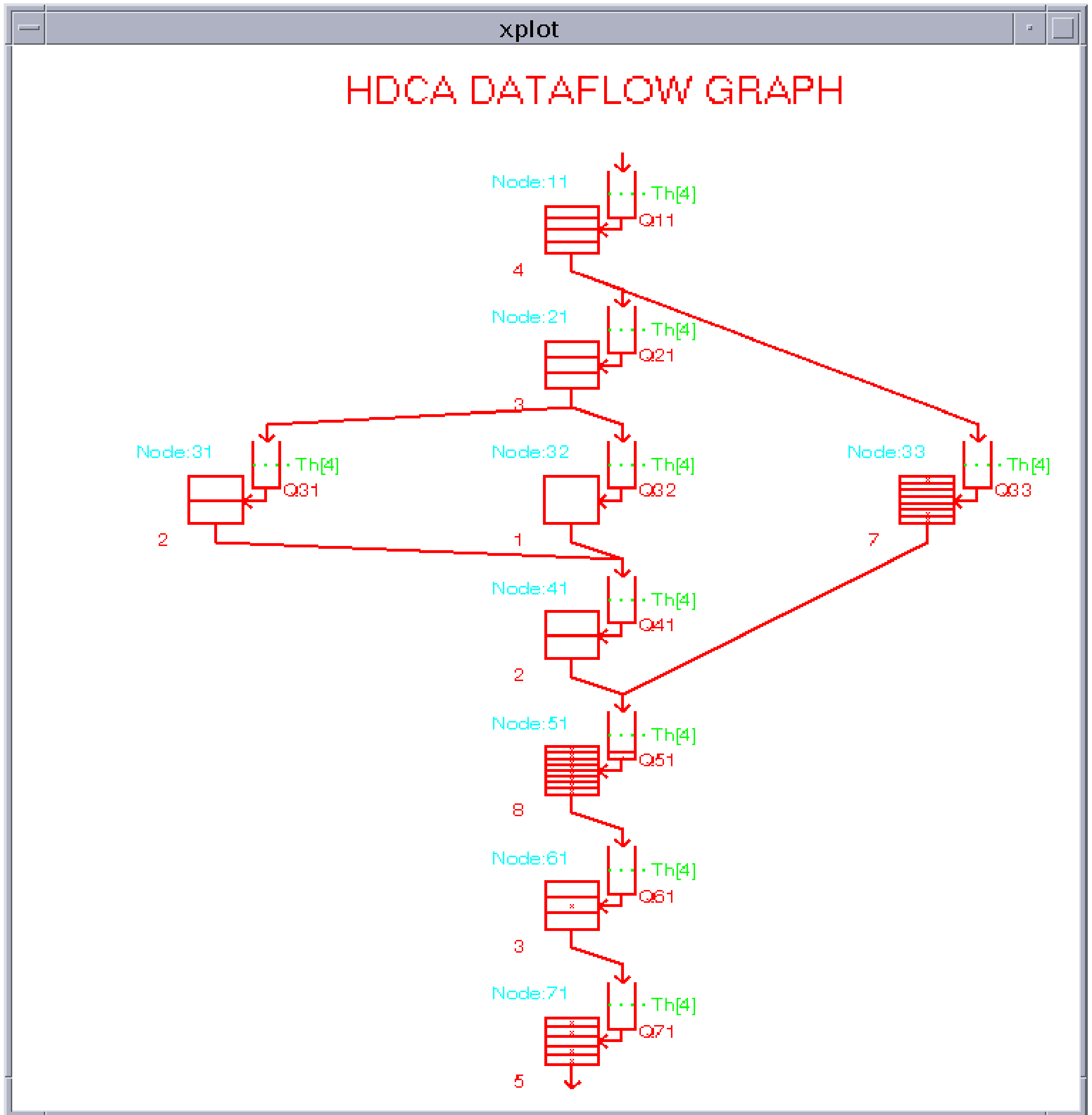


Figure 5.2 h   Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=8000 micro-cycles)

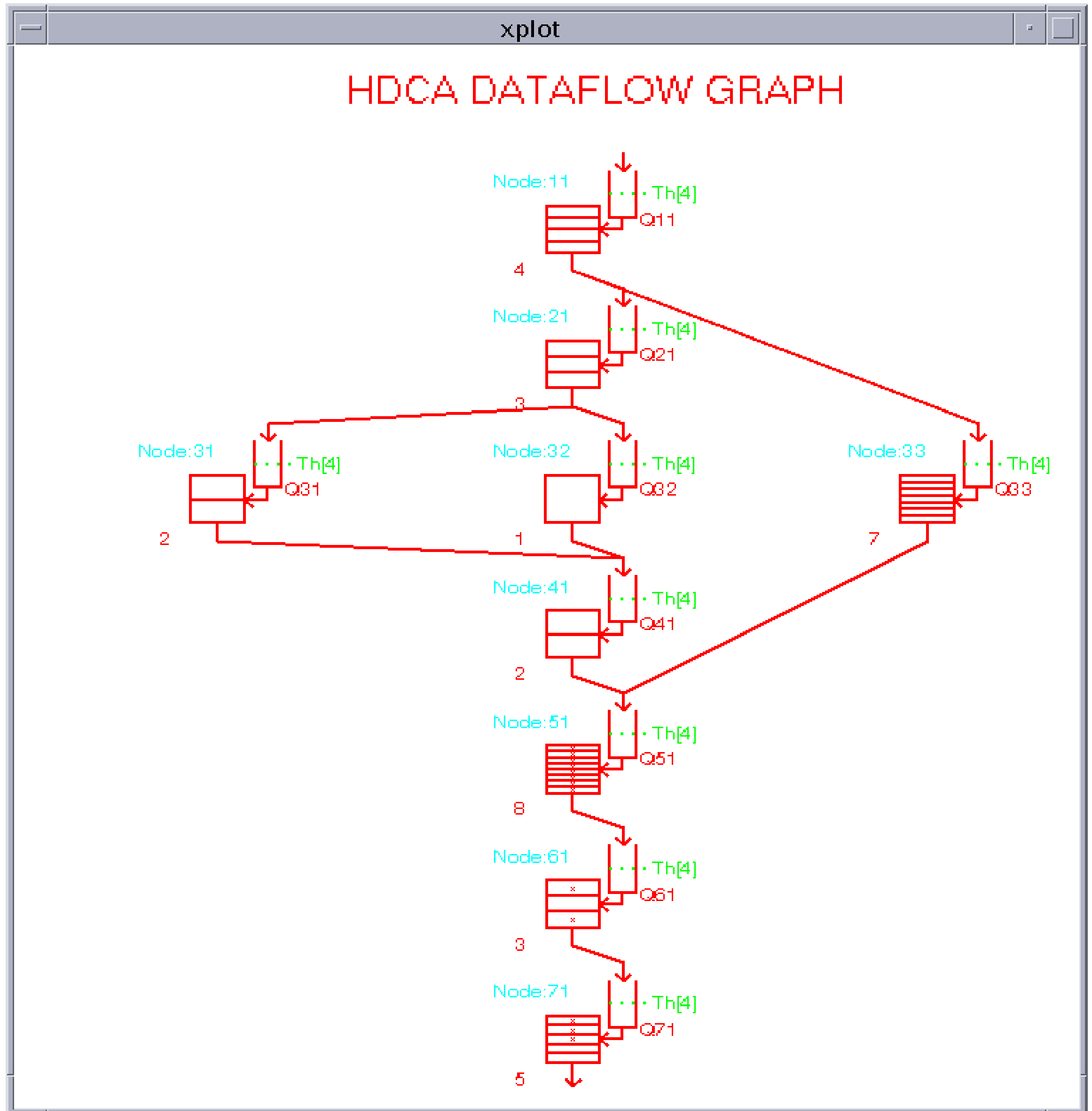


Figure 5.2 i Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=9000 micro-cycles)

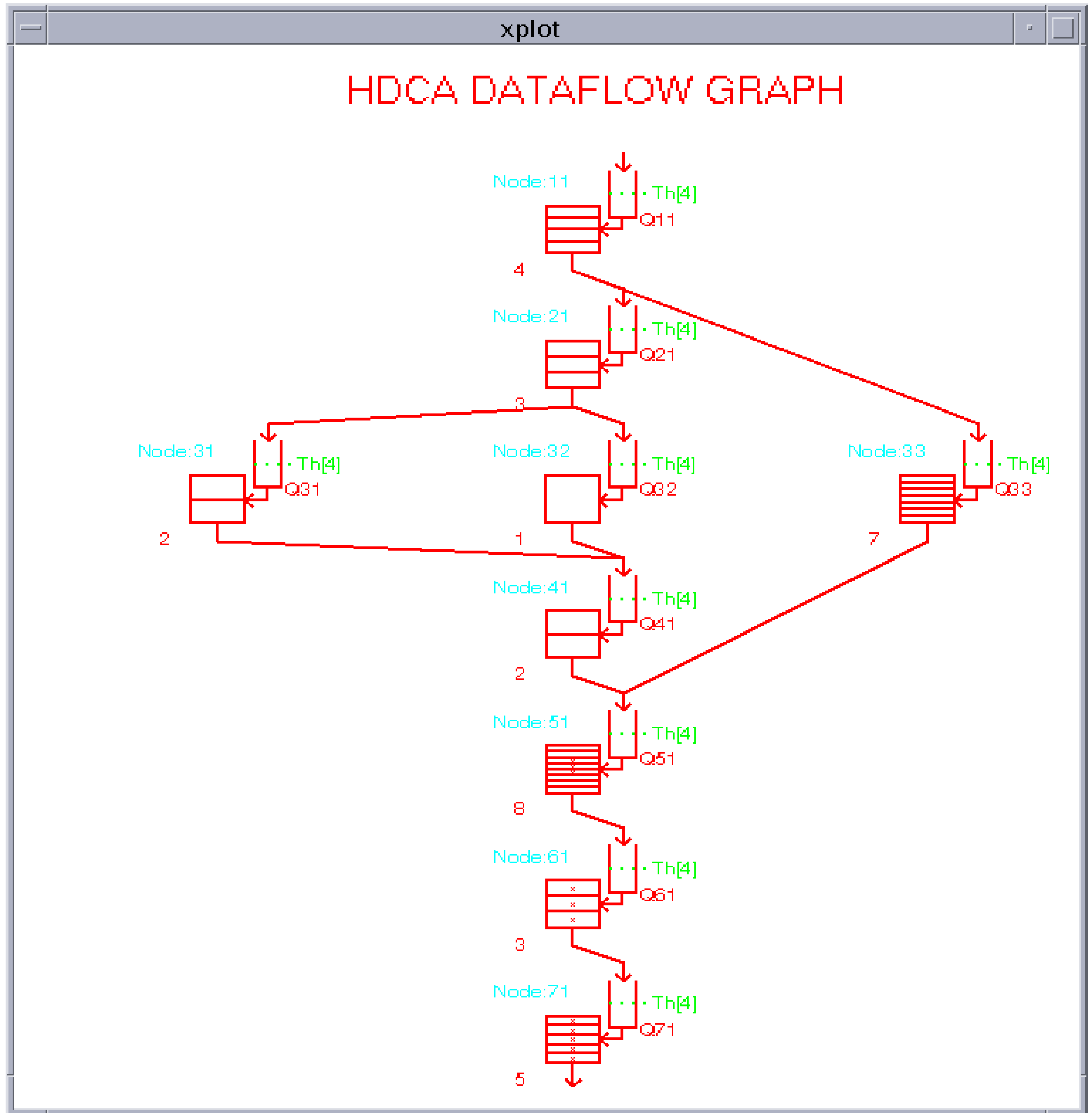


Figure 5.2 j Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=10000 micro-cycles)

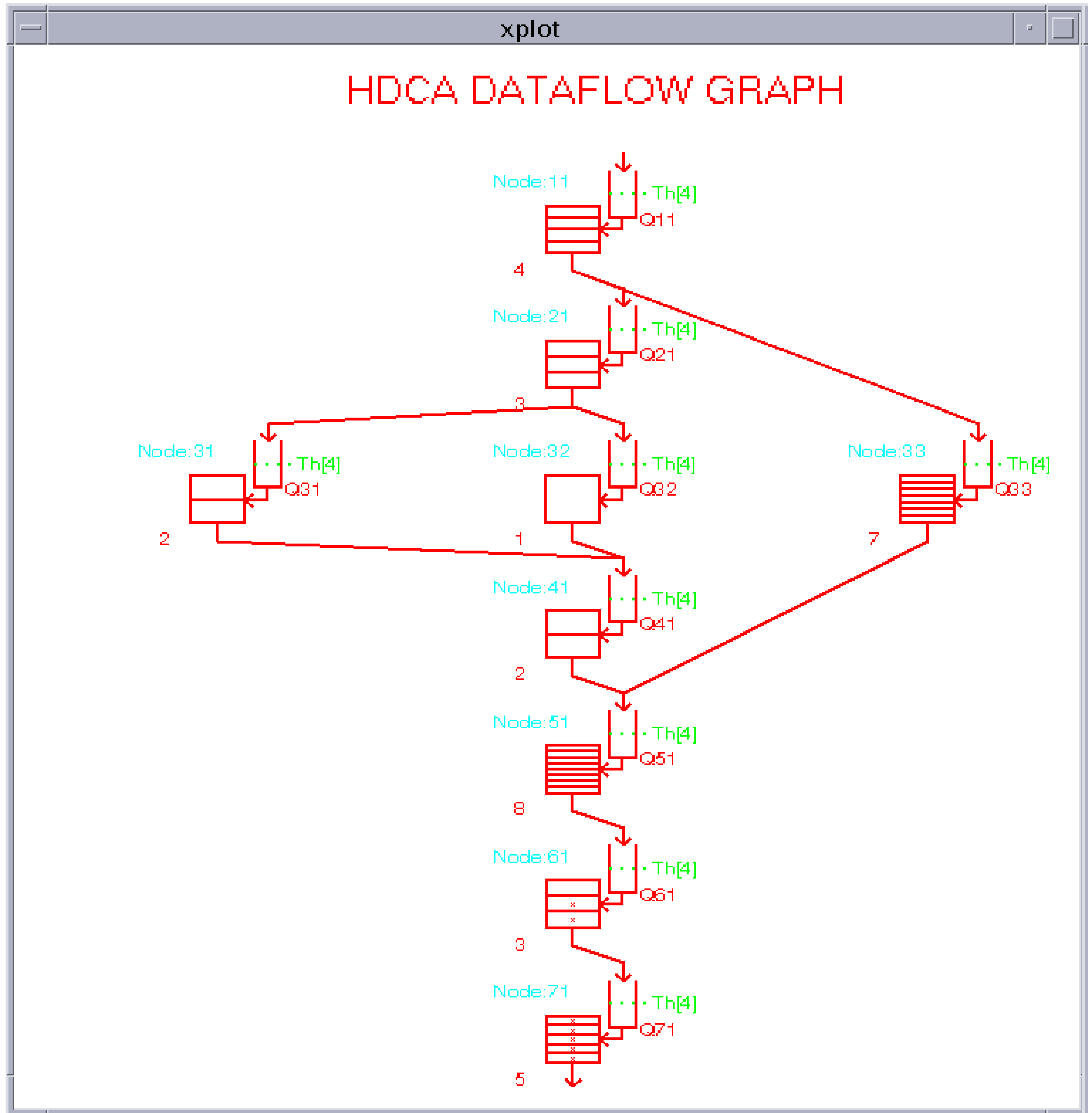


Figure 5.2 k Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=11000 micro-cycles)

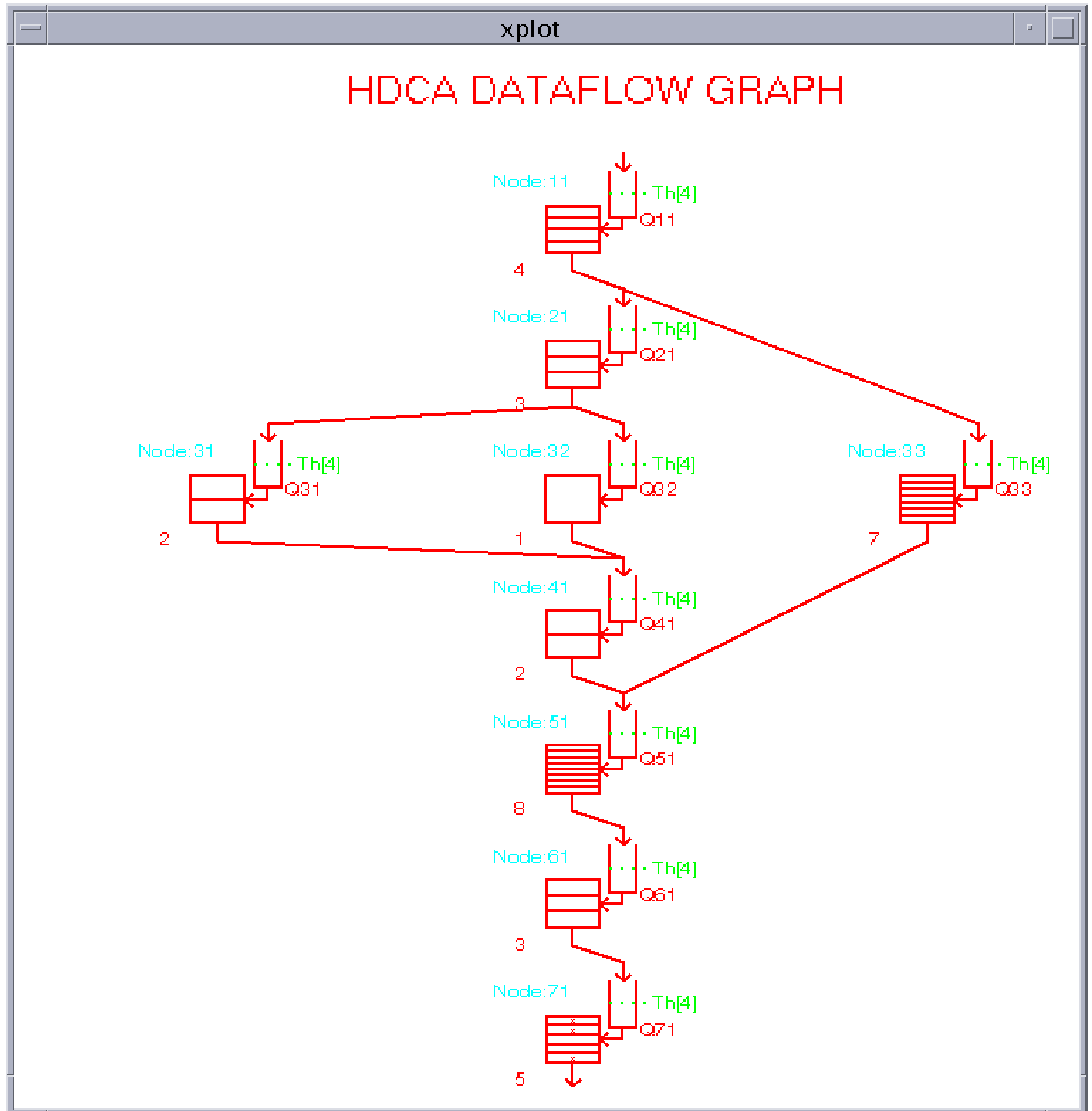


Figure 5.2 l Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=12000 micro-cycles)

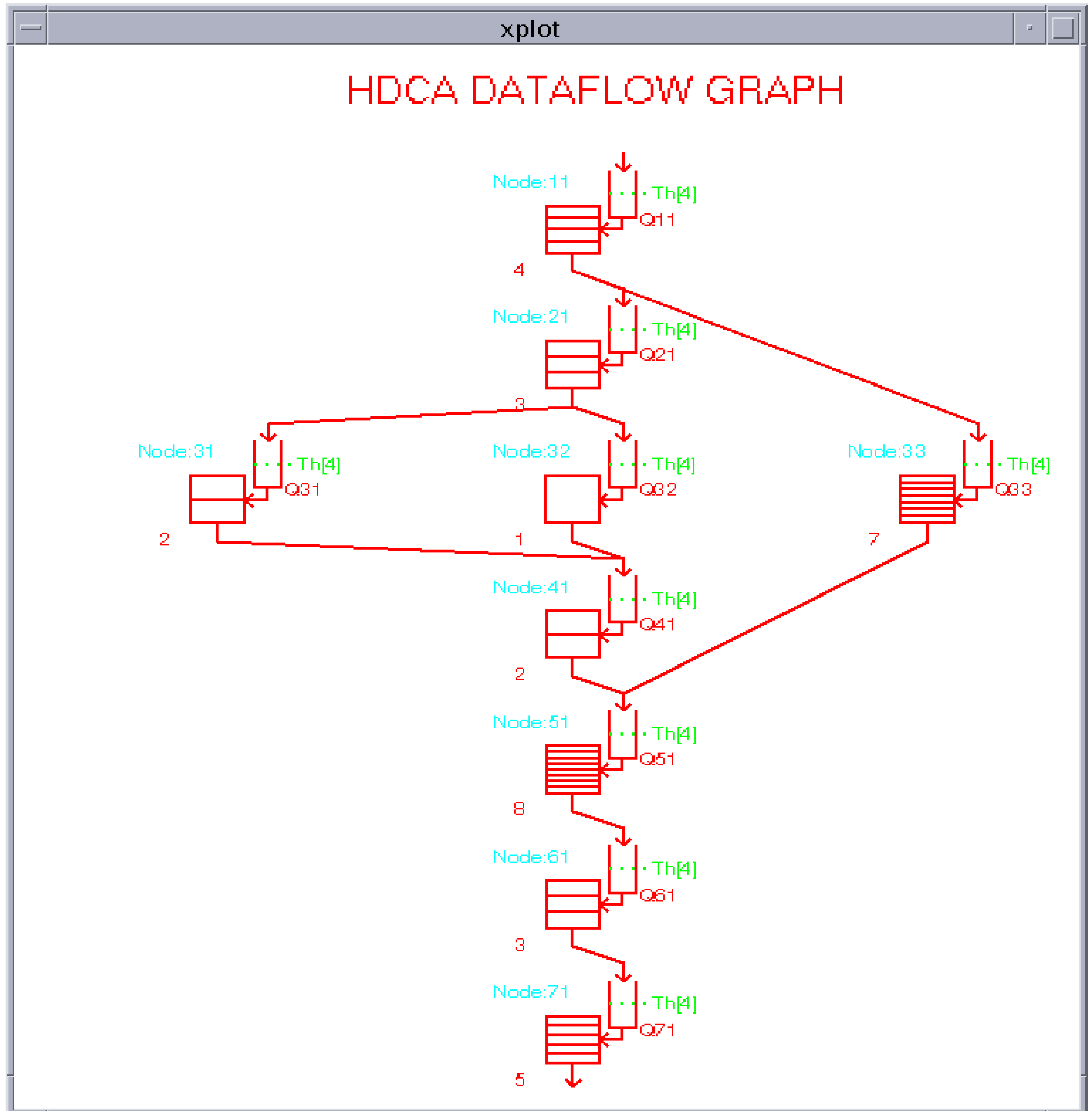


Figure 5.2 m Simu. Results of Application 1 (Input Rate=263micro-cycles/token) (t=13000 micro-cycles)

Total Simulation Time = 12633 Micro-cycles

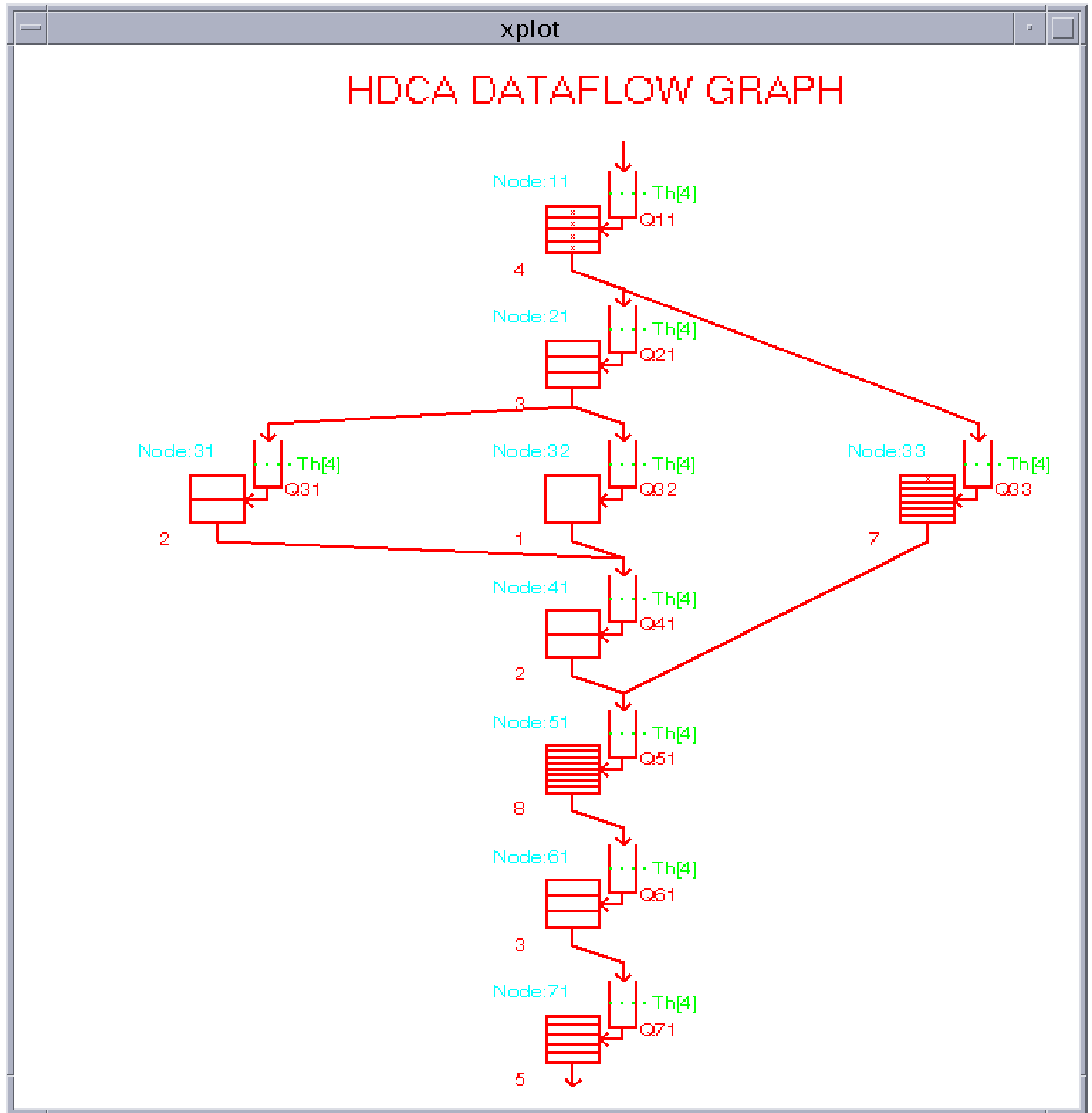


Figure 5.3 a Simu. Results of Application 1 (Input Rate=200micro-cycles/token)

(t=1000 micro-cycles)

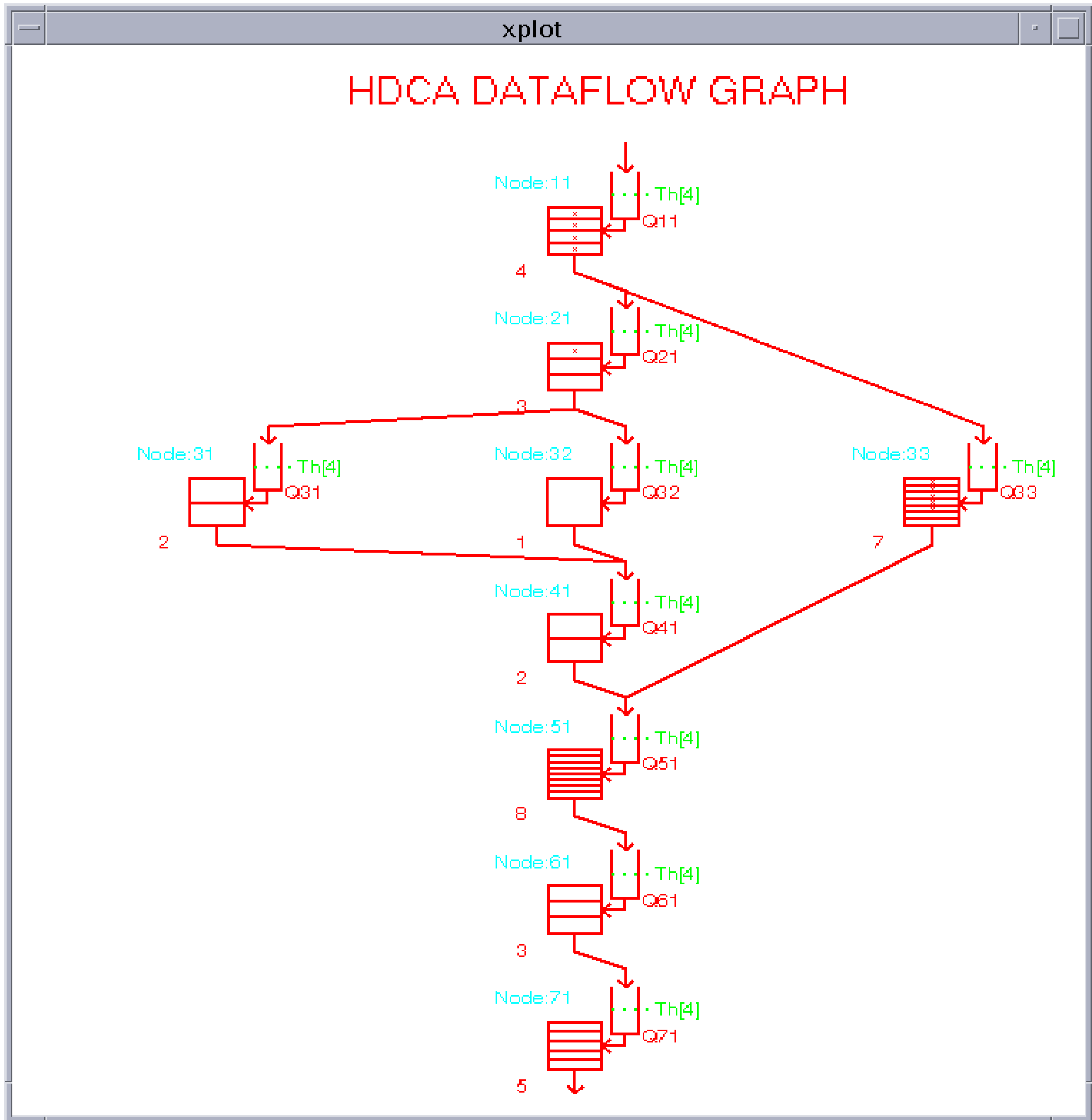


Figure 5.3 b Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=2000 micro-cycles)

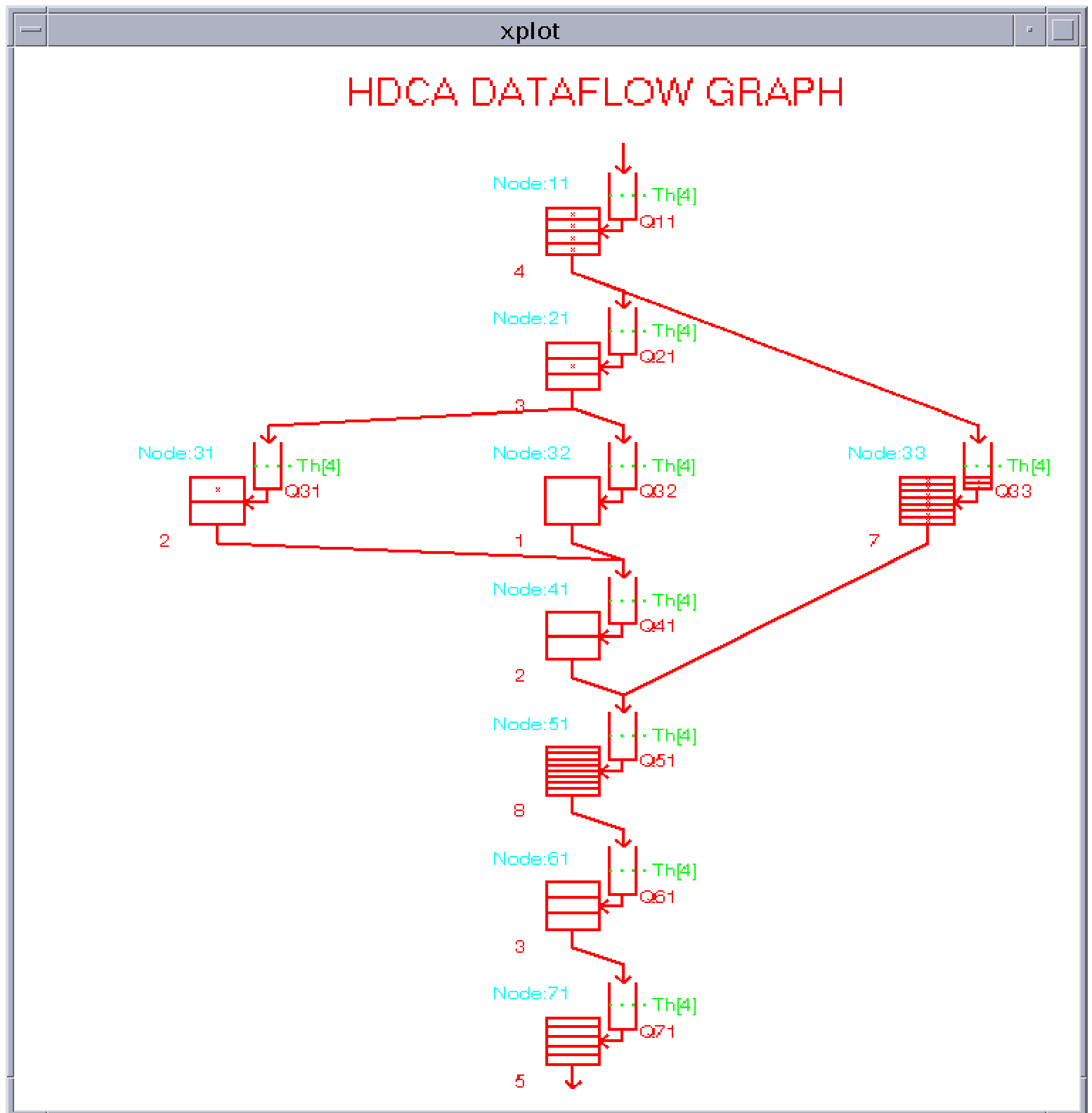


Figure 5.3 c Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=3000 micro-cycles)

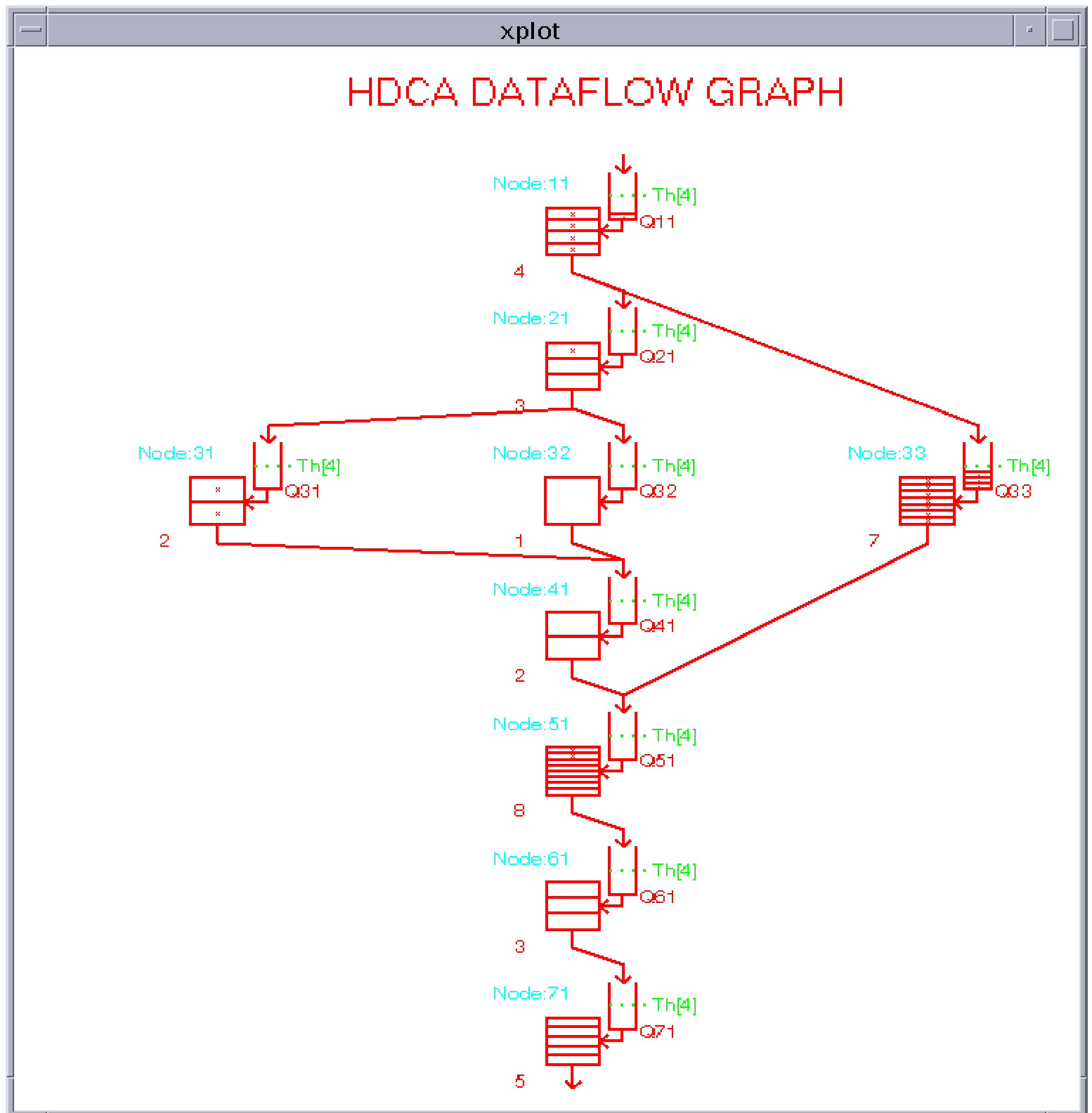


Figure 5.3 d Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=4000 micro-cycles)

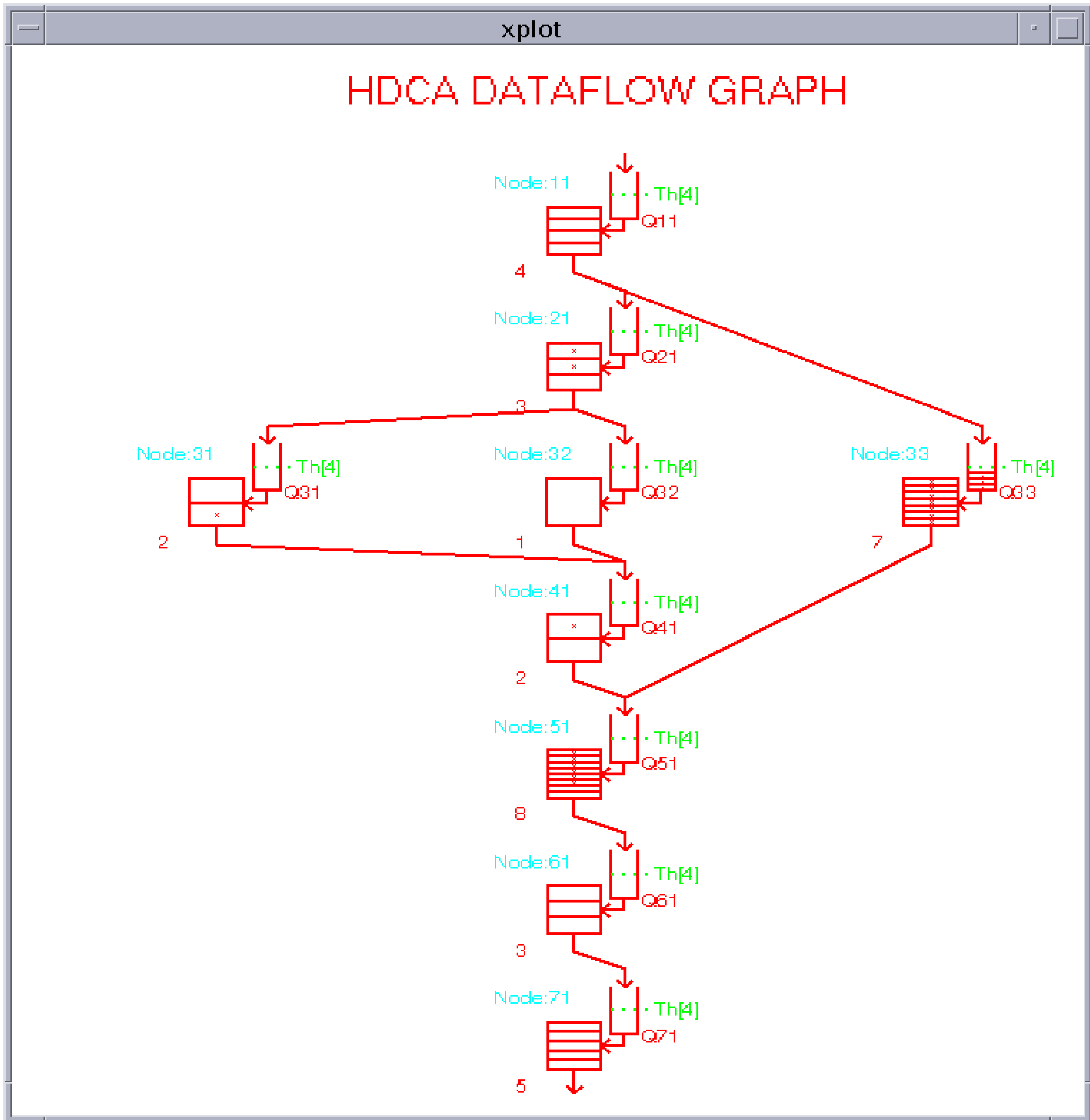


Figure 5.3 e Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=5000 micro-cycles)

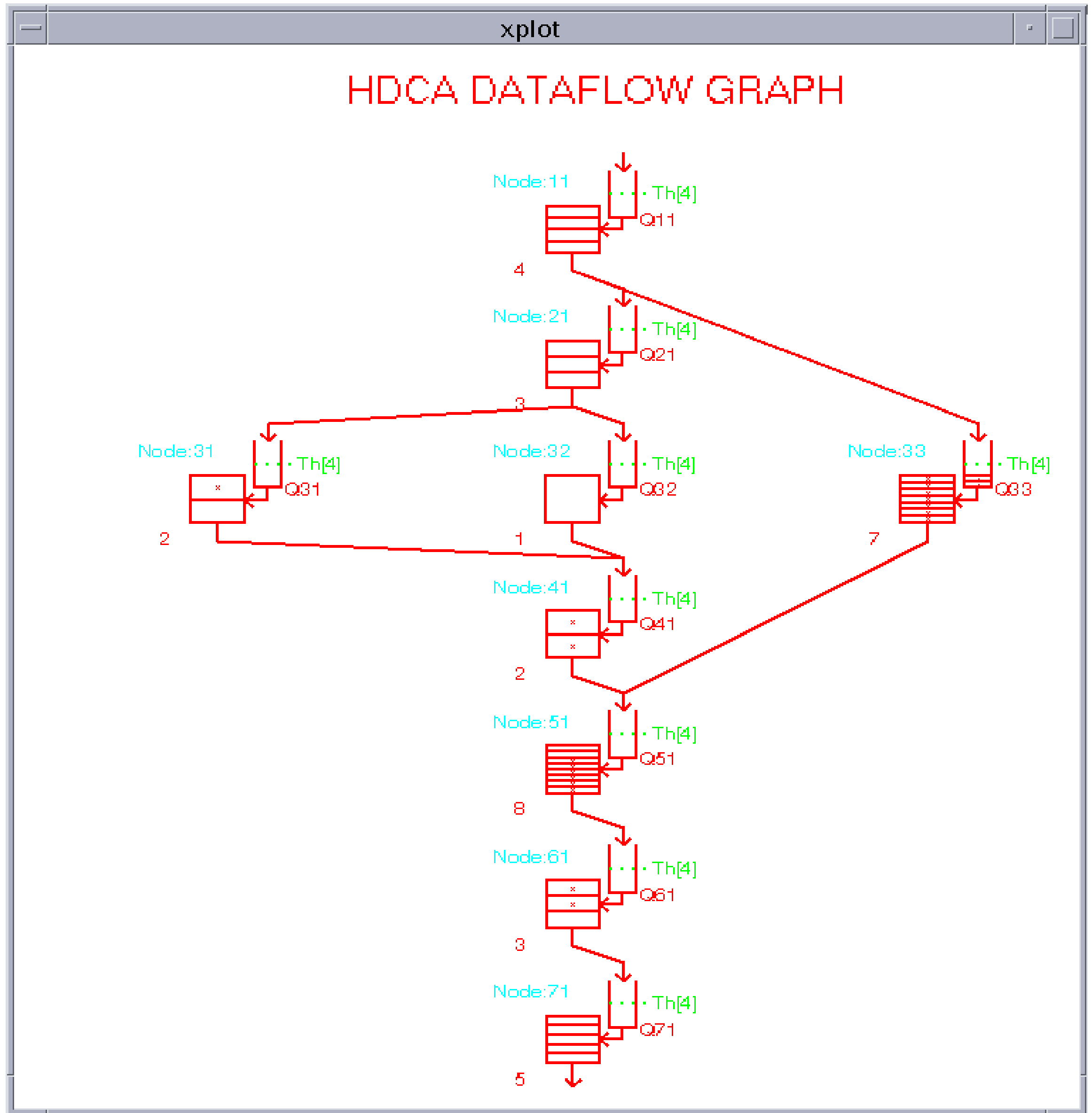


Figure 5.3 f Simu. Results of Application 1 (Input Rate=200micro-cycles/token)

(t=6000 micro-cycles)

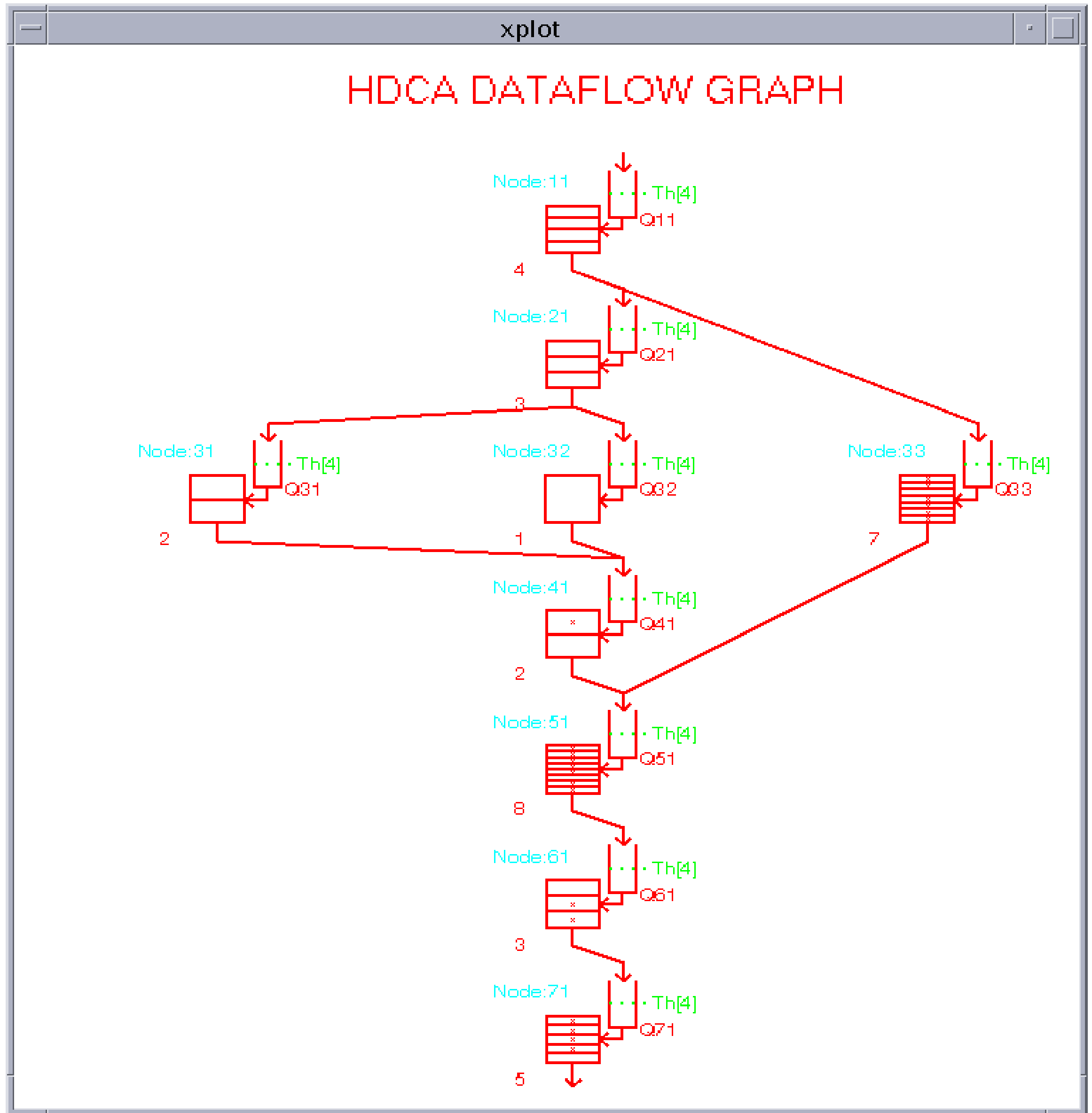


Figure 5.3 g Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=7000 micro-cycles)

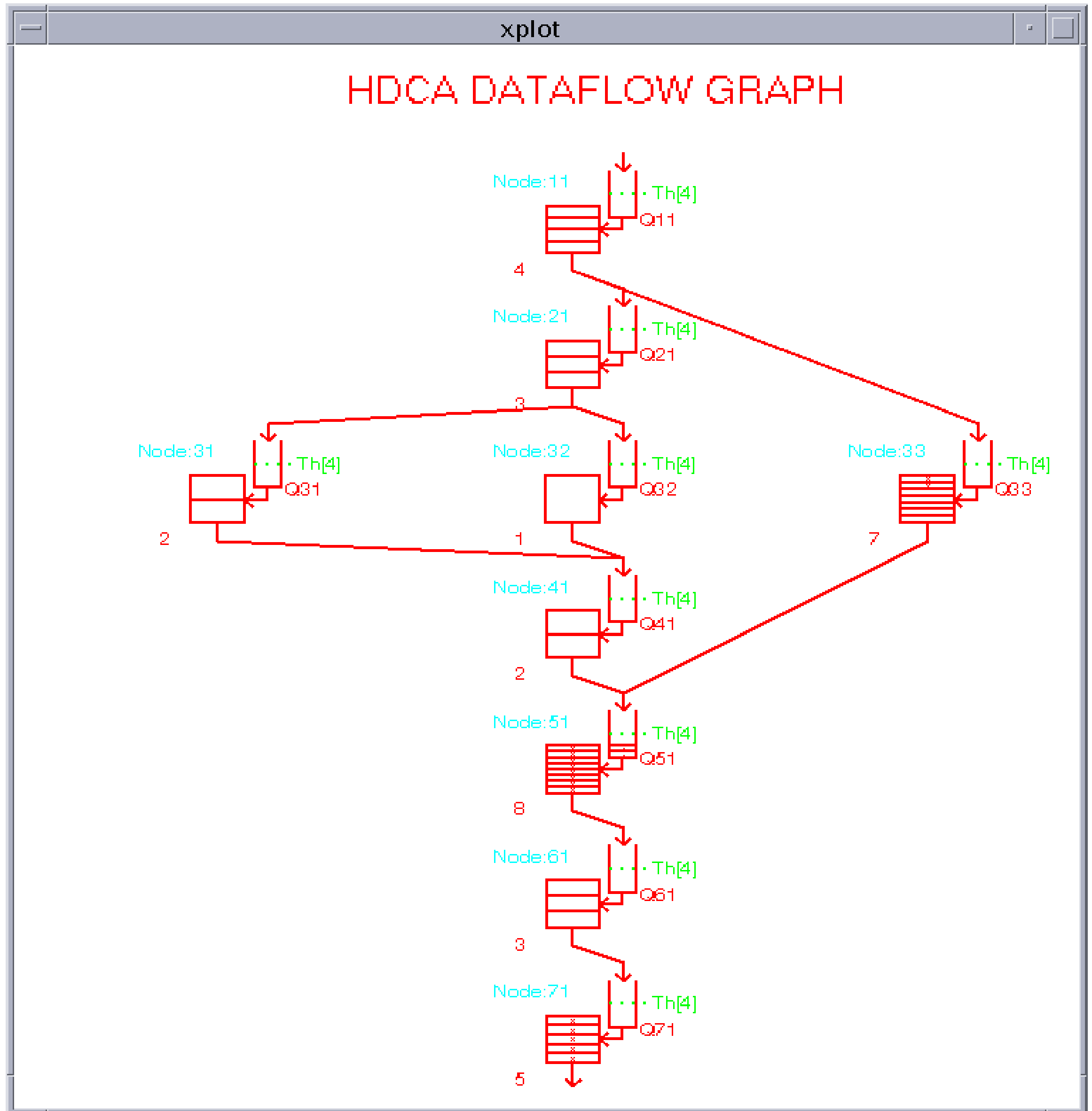


Figure 5.3 h Simu. Results of Application 1 (Input Rate=200micro-cycles/token)

(t=8000 micro-cycles)

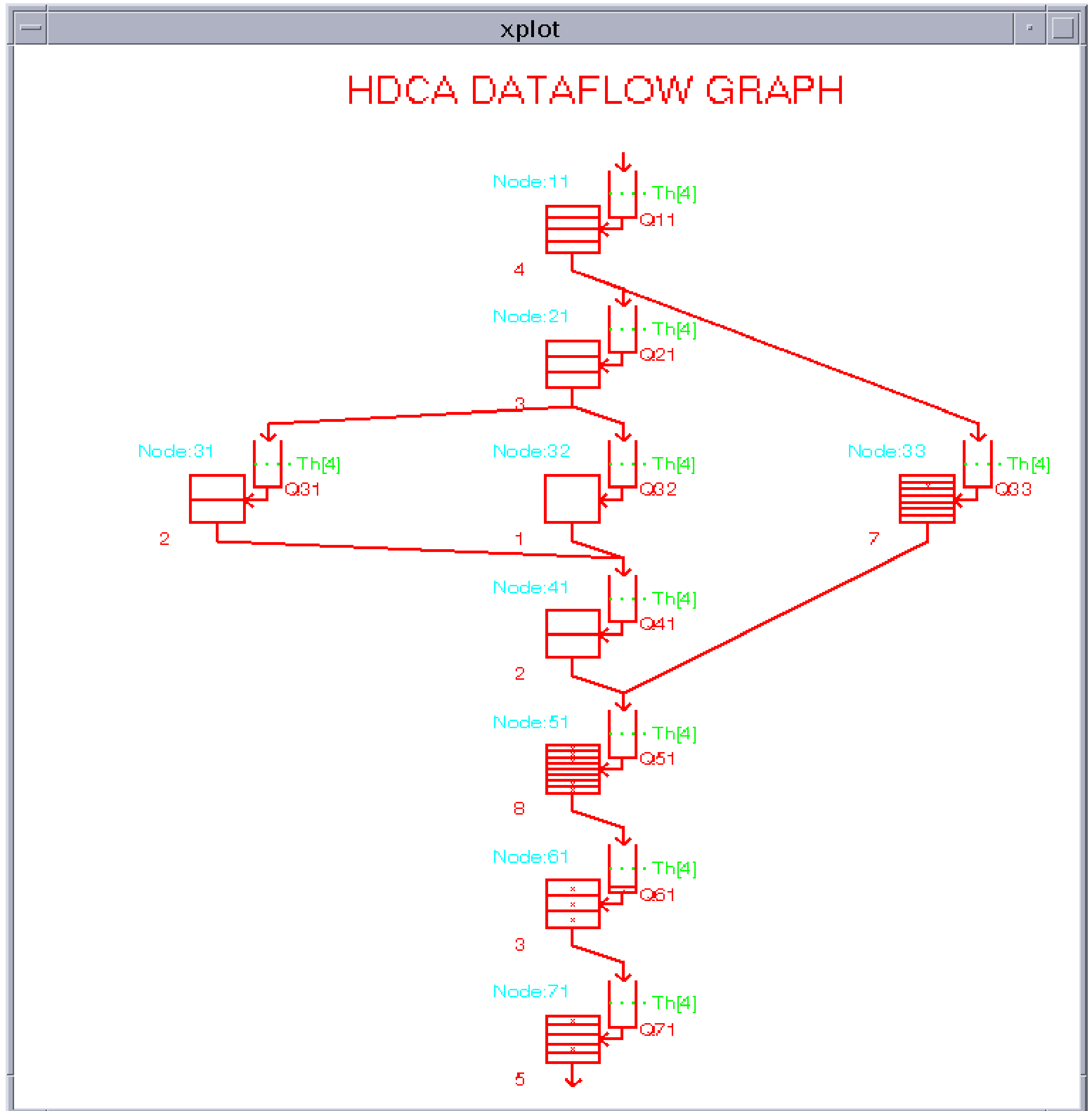


Figure 5.3 i Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=9000 micro-cycles)

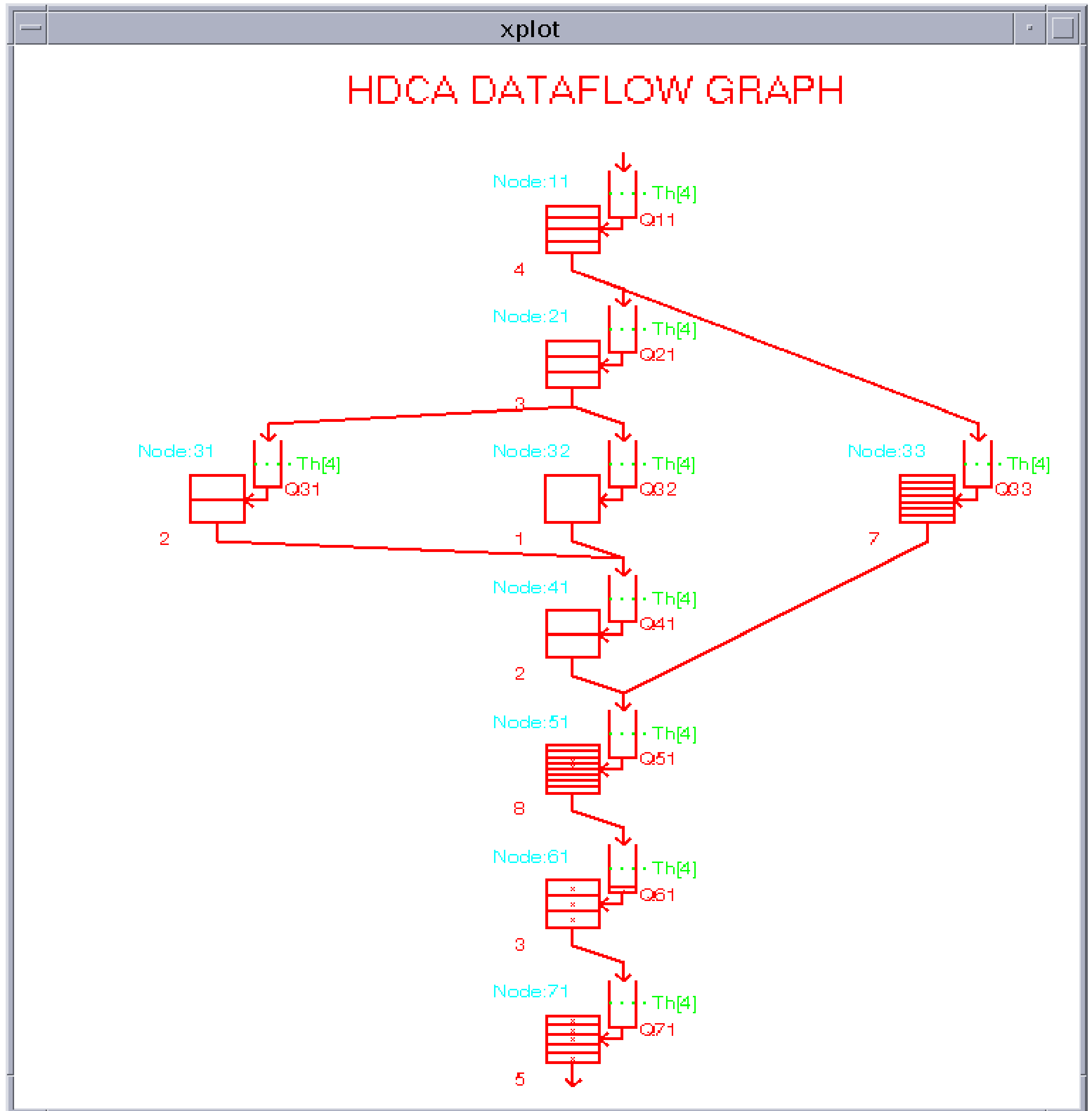


Figure 5.3 j Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=10000 micro-cycles)

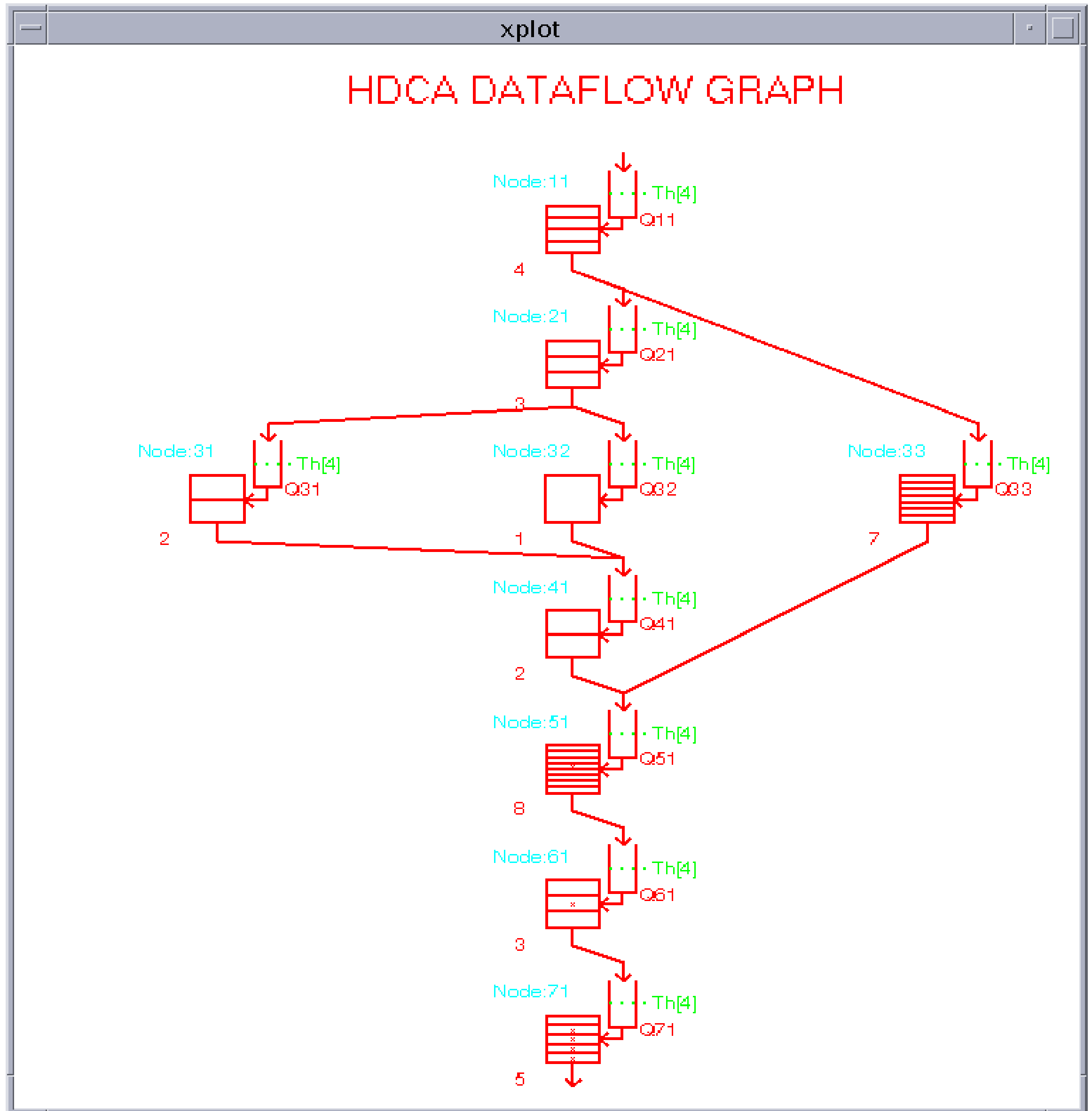


Figure 5.3 k Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=11000 micro-cycles)

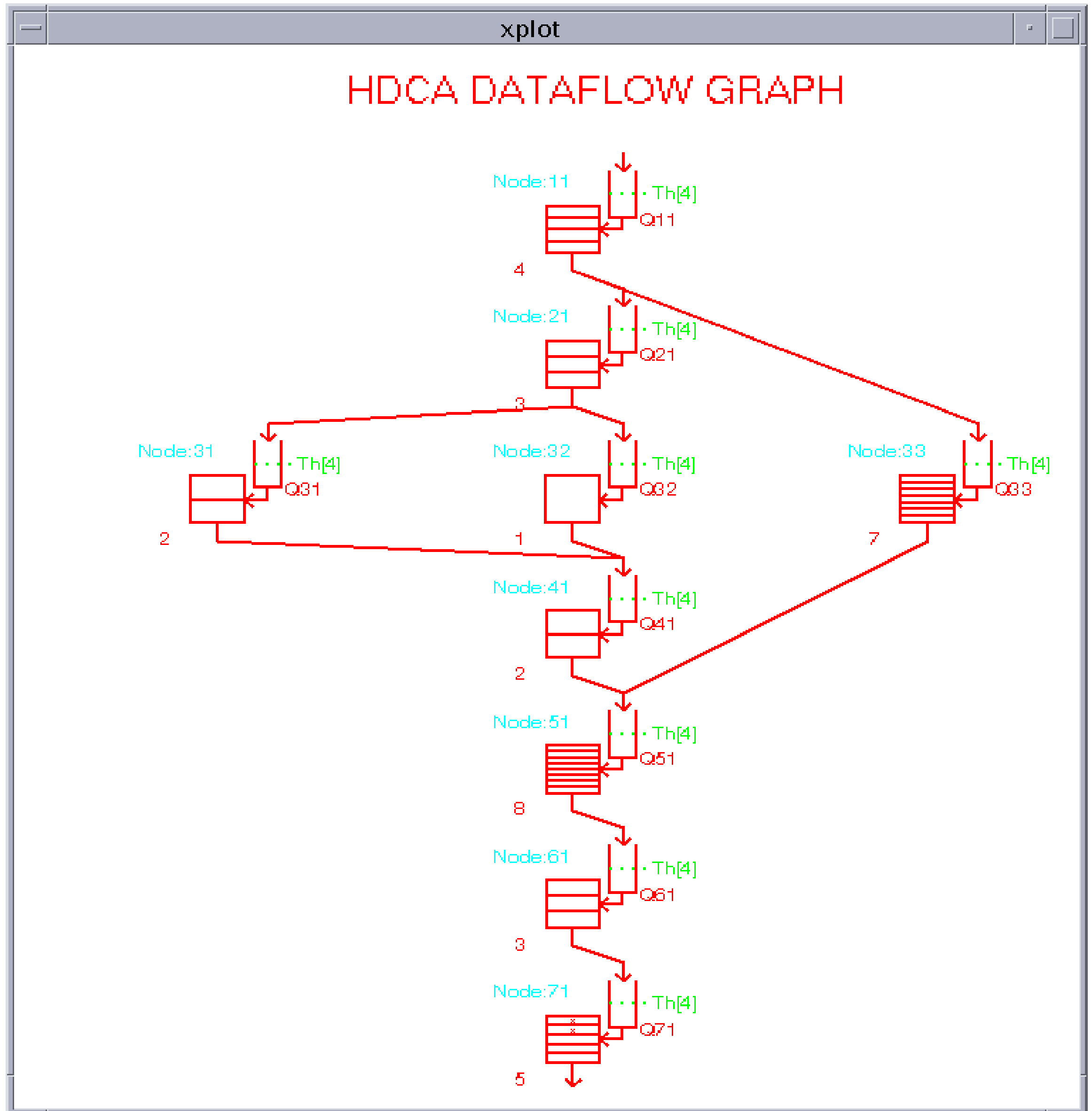


Figure 5.3 l   Simu. Results of Application 1 (Input Rate=200micro-cycles/token)

(t=12000 micro-cycles)

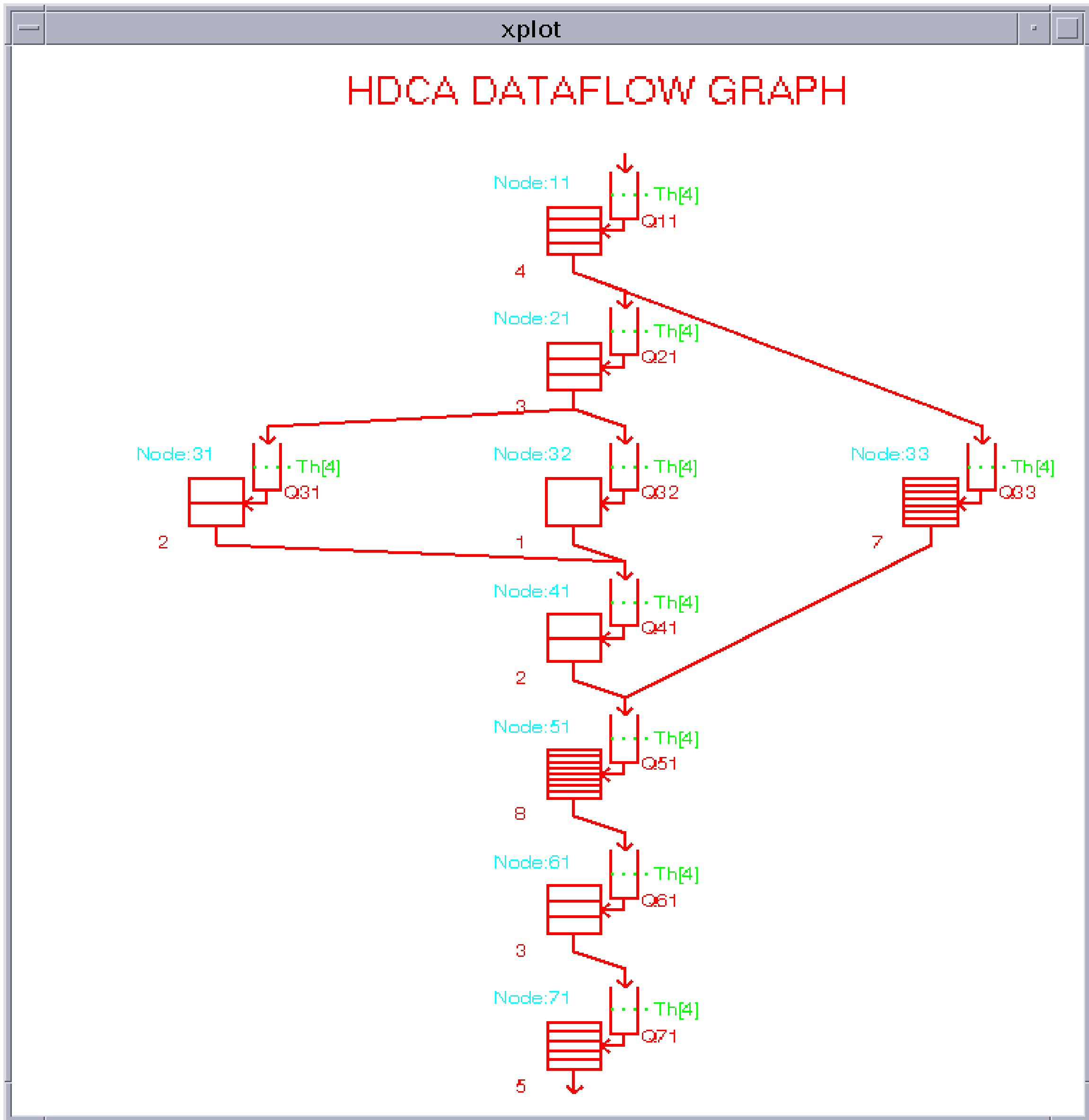


Figure 5.3 m Simu. Results of Application 1 (Input Rate=200micro-cycles/token) (t=13000 micro-cycles)

Total Simulation Time = 12834 Micro-cycles

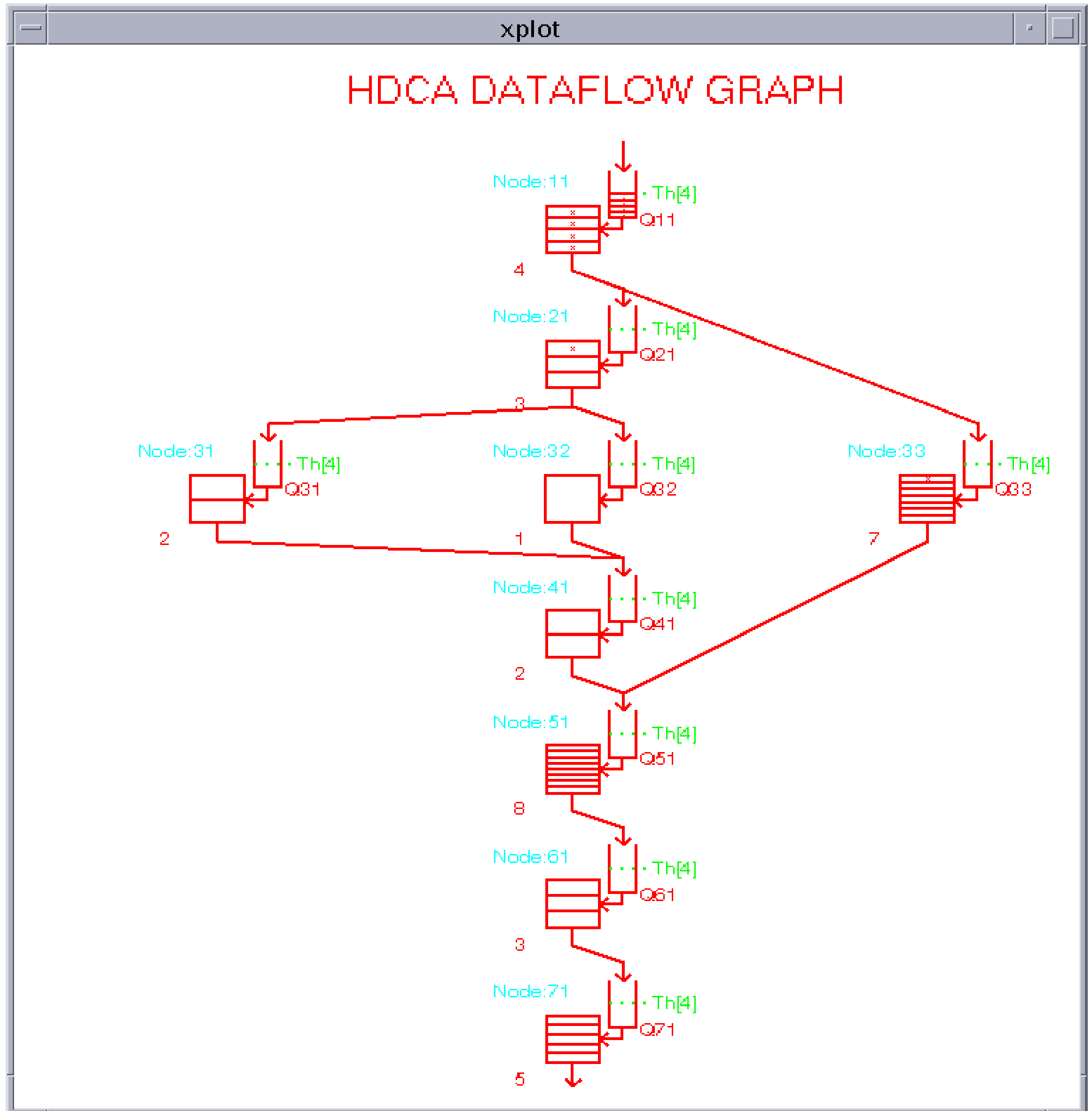


Figure 5.4 a   Simu. Results of Application 1 (Input Rate=100micro-cycles/token)

(t=1000 micro-cycles)

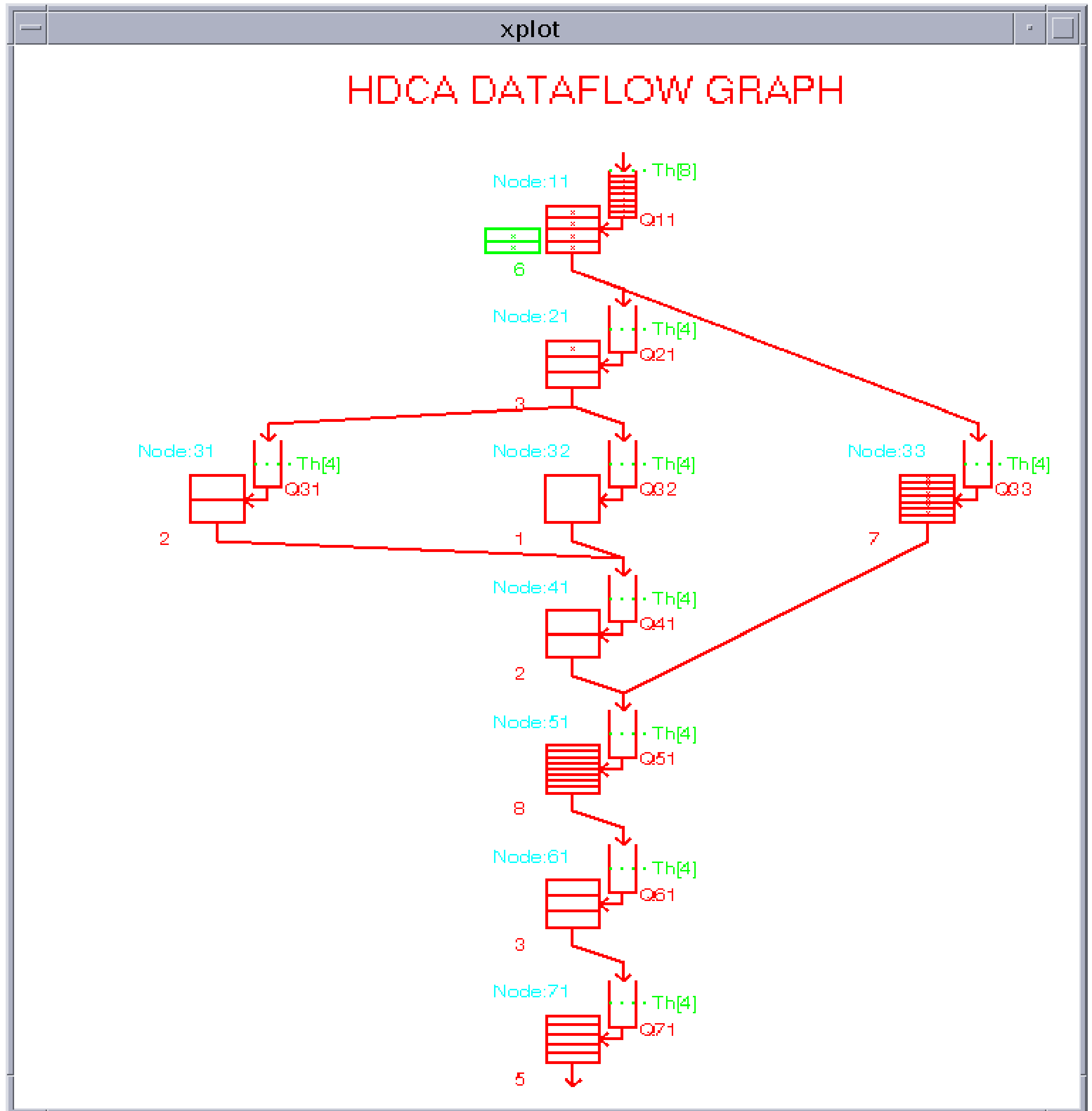


Figure 5.4 b Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=2000 micro-cycles)

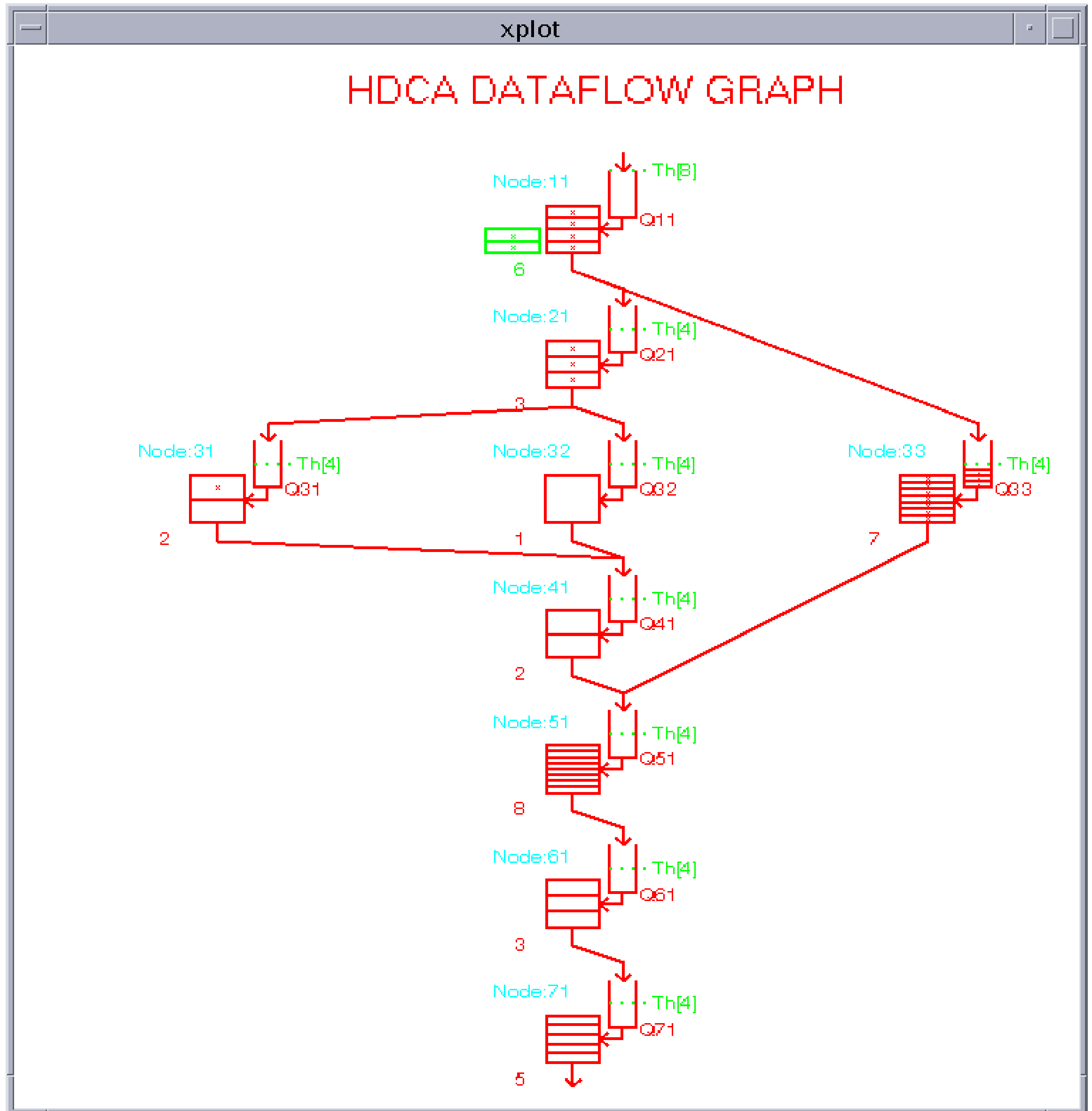


Figure 5.4 c Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=3000 micro-cycles)

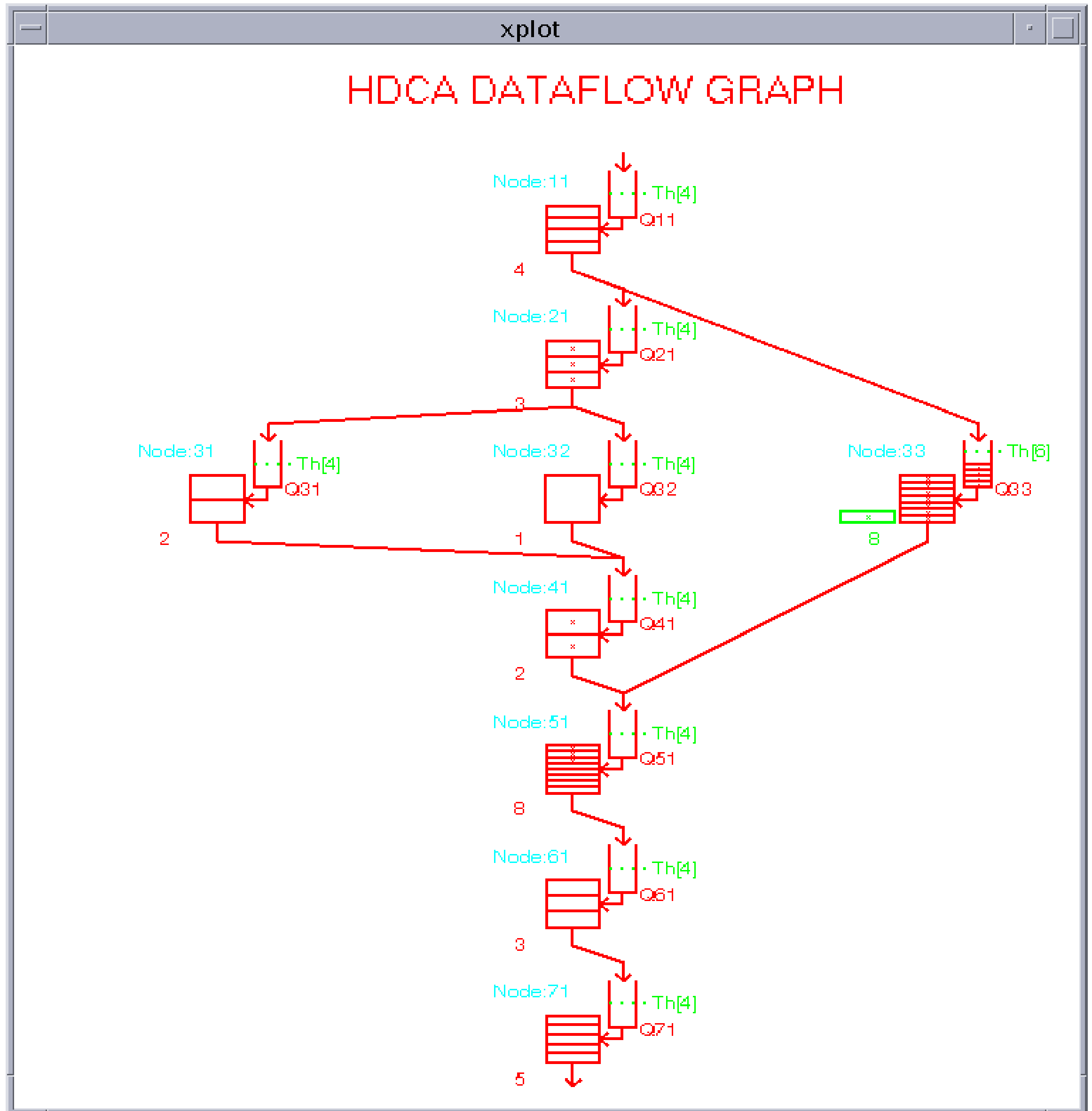


Figure 5.4 d Simu. Results of Application 1 (Input Rate=100micro-cycles/token)

(t=4000 micro-cycles)

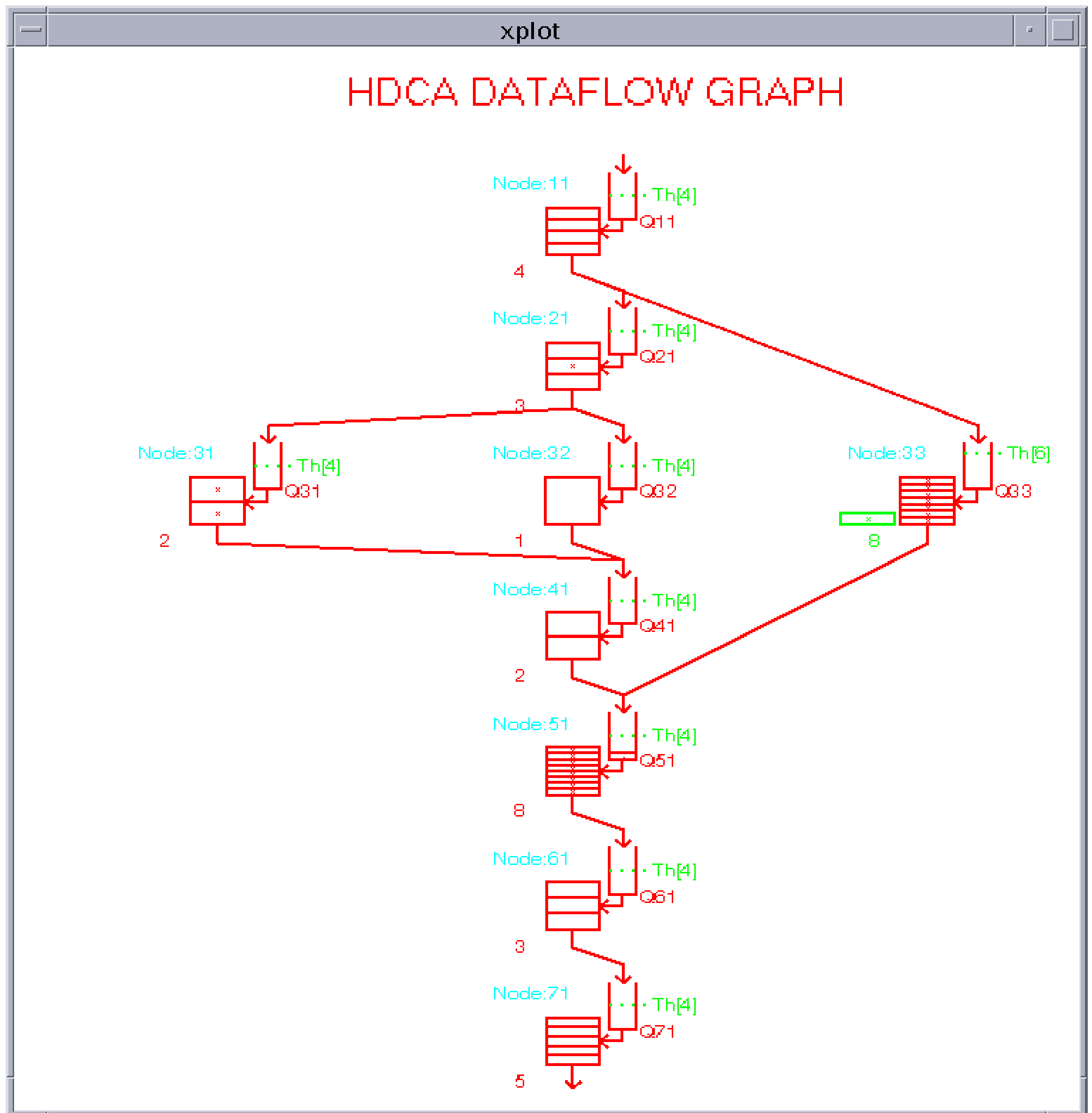


Figure 5.4 e Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=5000 micro-cycles)

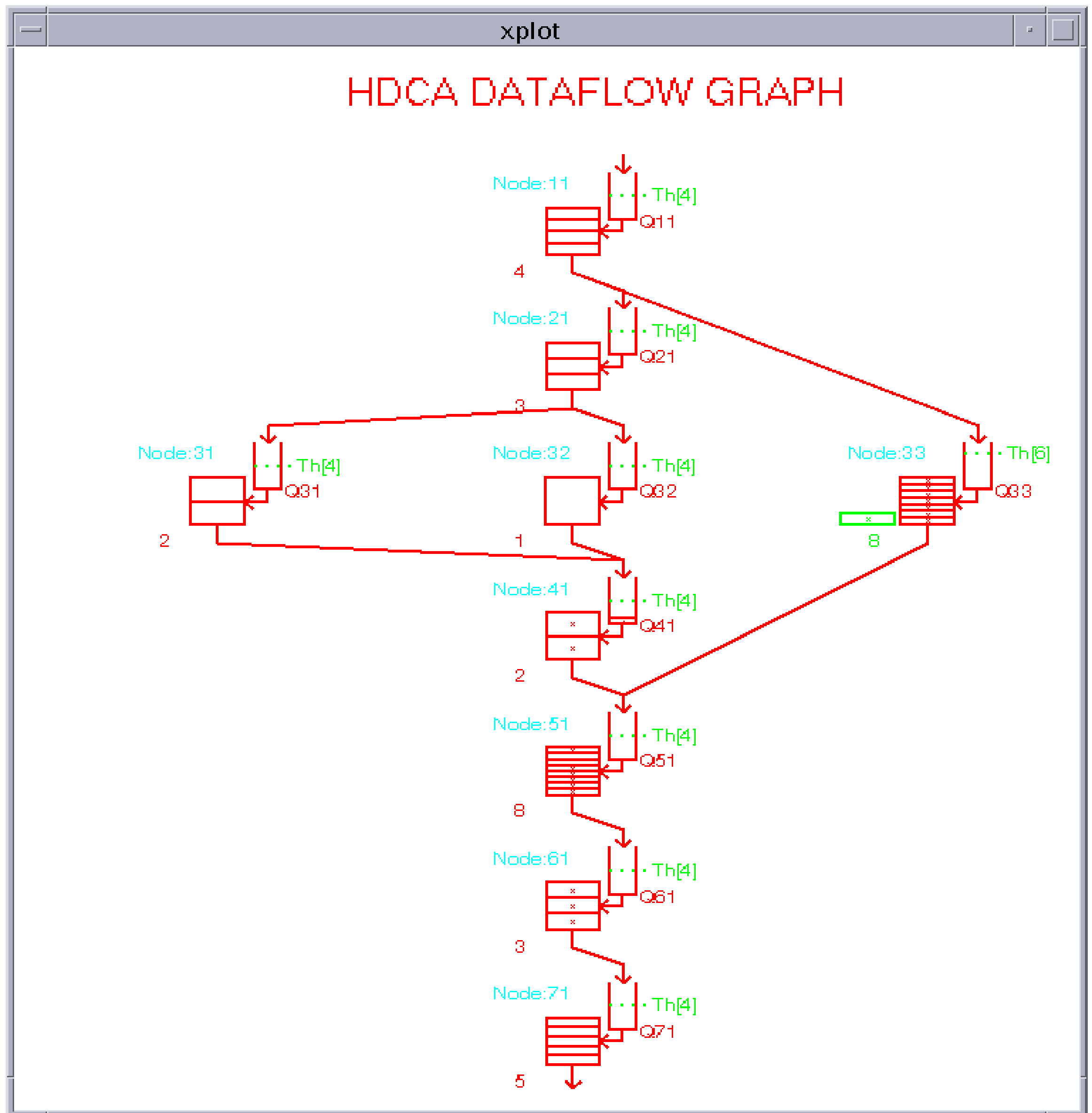


Figure 5.4 f Simu. Results of Application 1 (Input Rate=100micro-cycles/token)

(t=6000 micro-cycles)

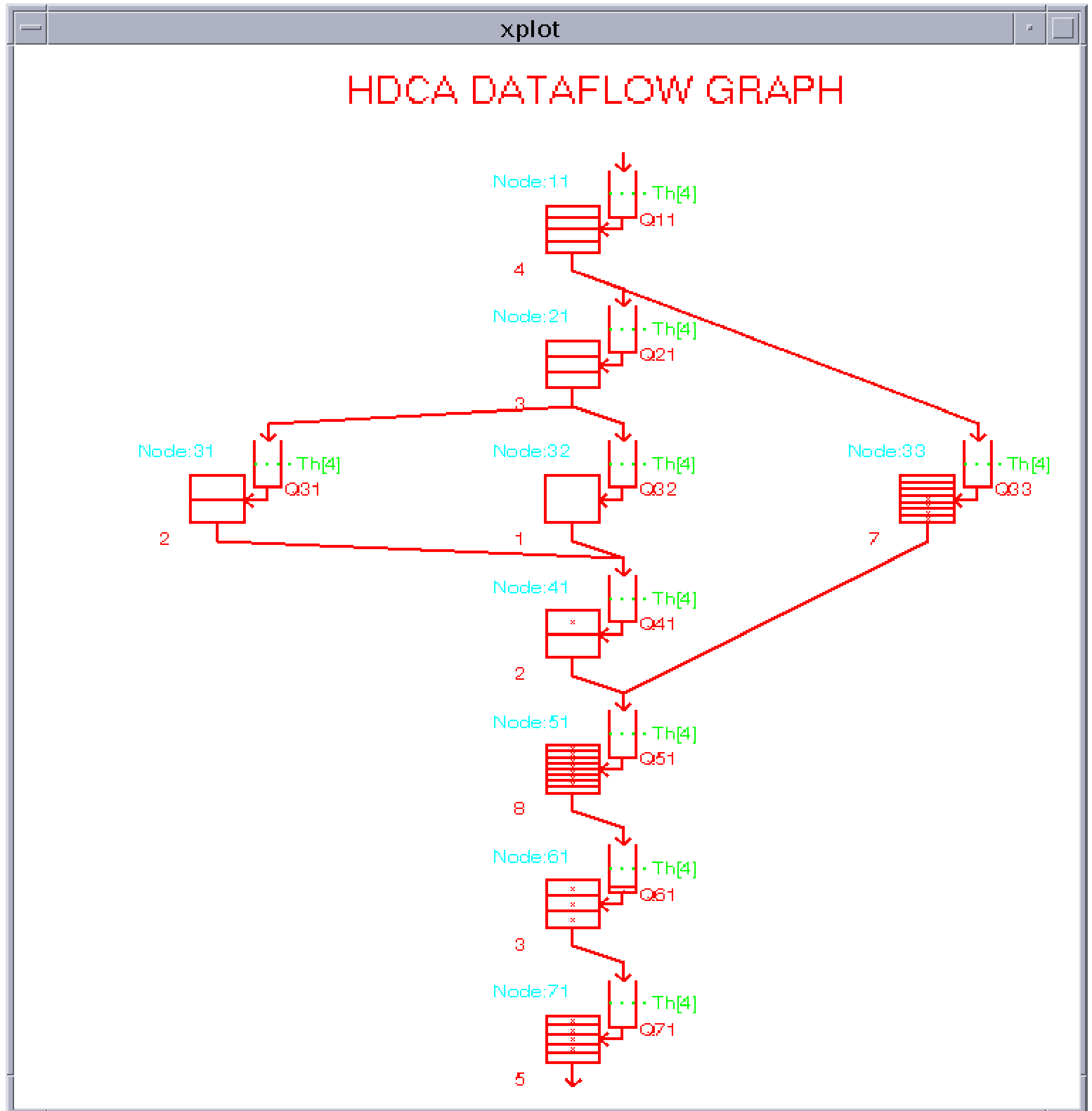


Figure 5.4 g Simu. Results of Application 1 (Input Rate=100micro-cycles/token)

(t=7000 micro-cycles)

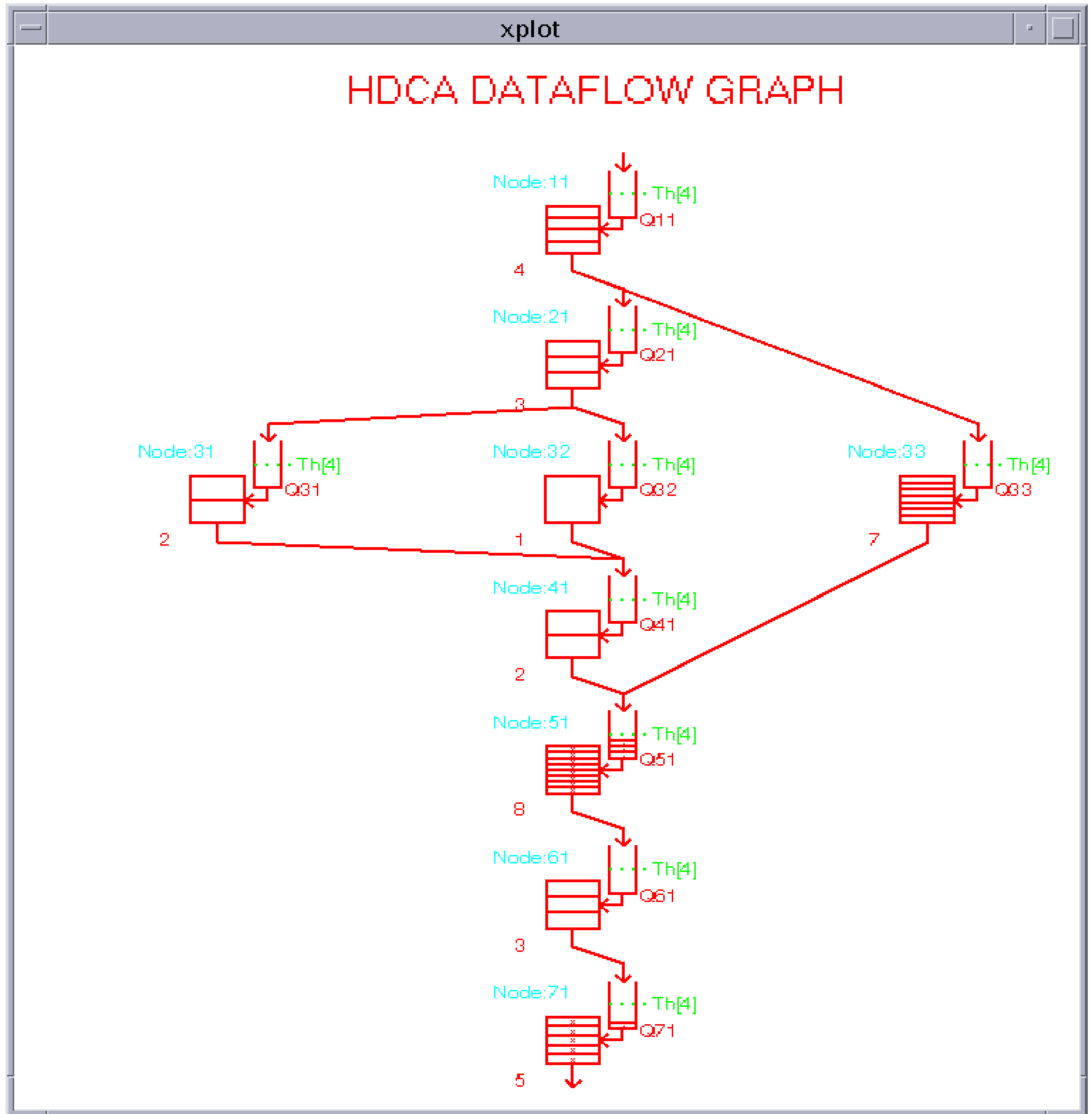


Figure 5.4 h Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=8000 micro-cycles)

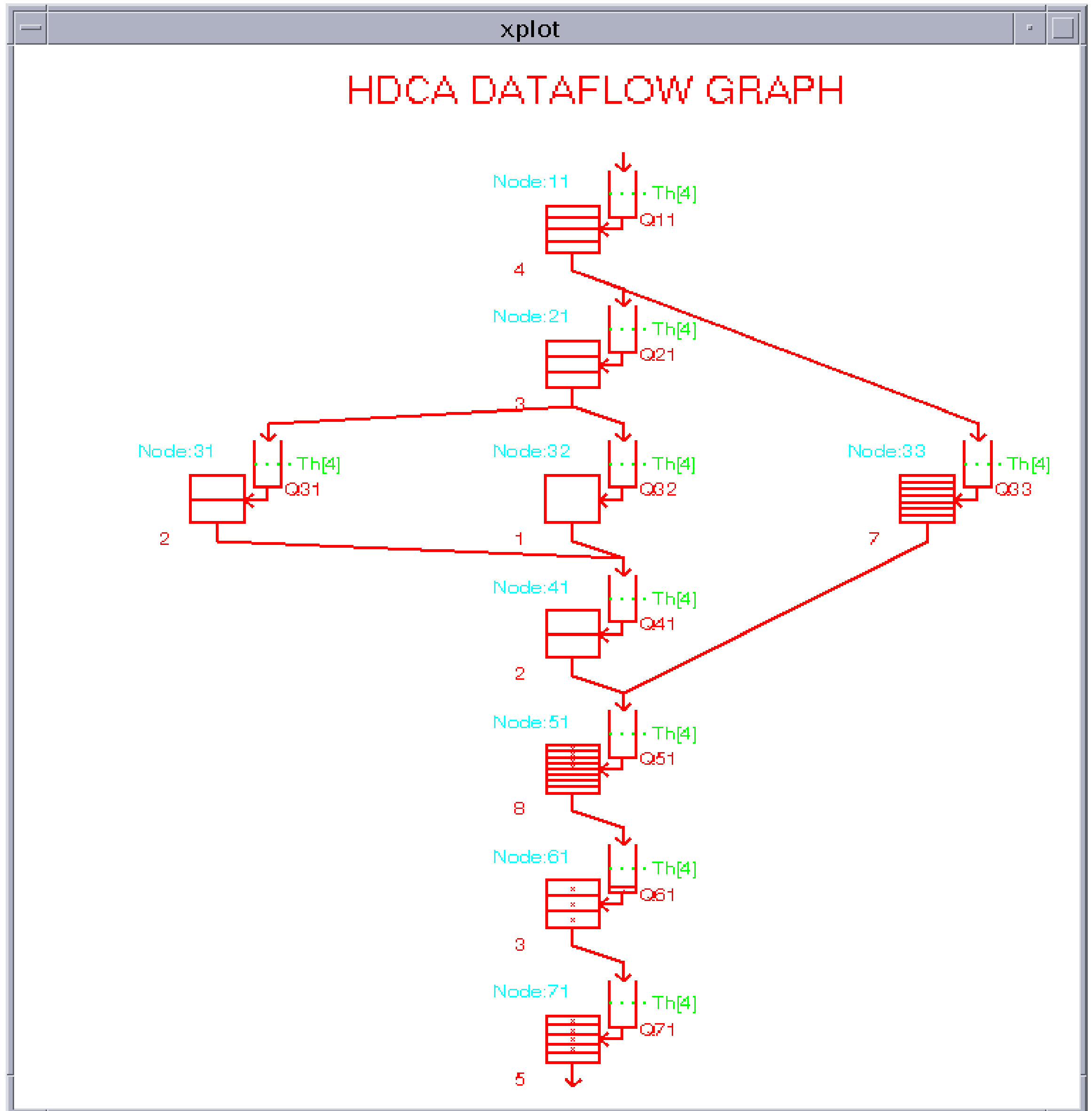


Figure 5.4 i   Simu. Results of Application 1 (Input Rate=100micro-cycles/token)

(t=9000 micro-cycles)

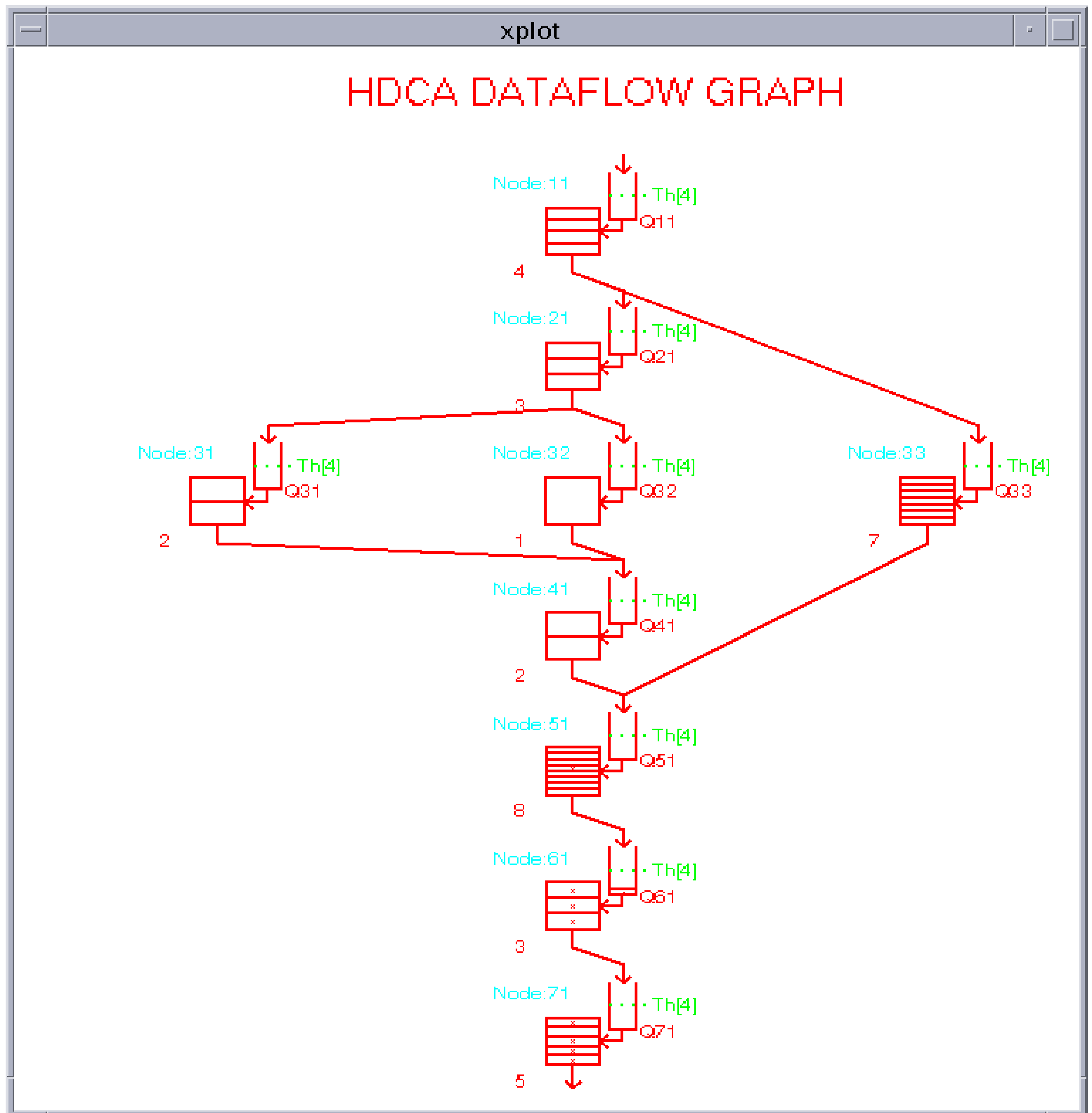


Figure 5.4 j Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=10000 micro-cycles)

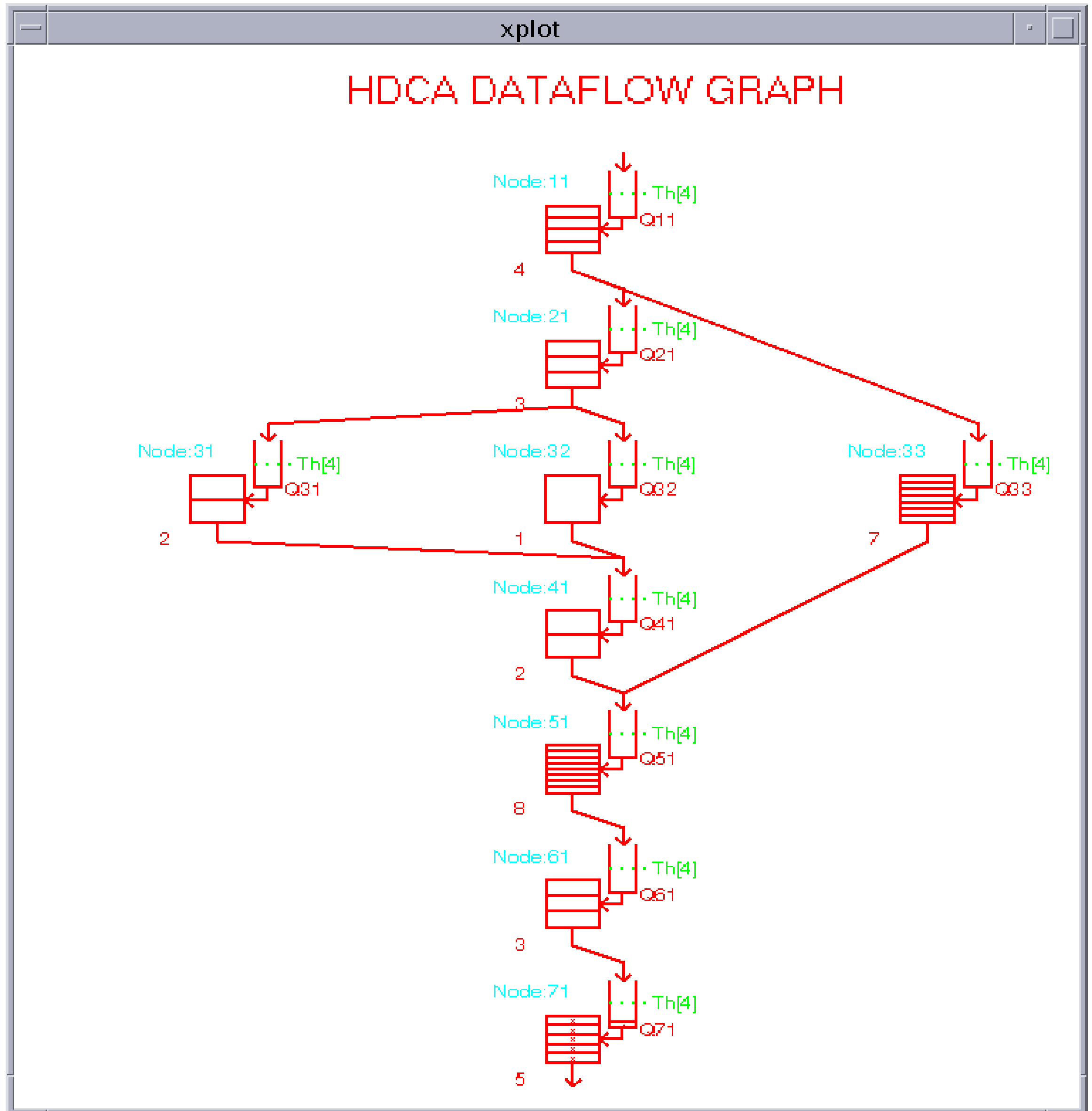


Figure 5.4 k Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=11000 micro-cycles)

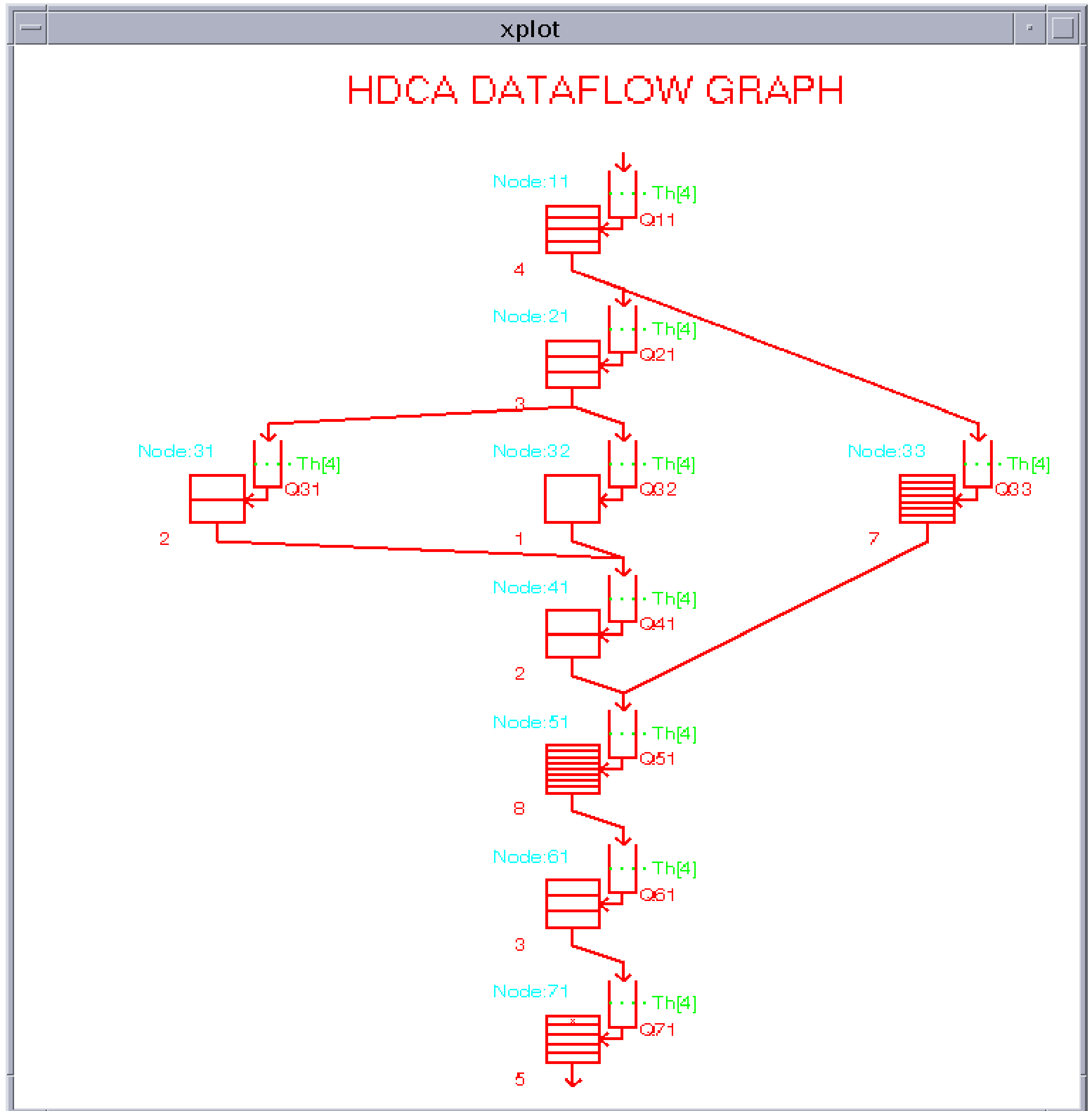


Figure 5.4 l Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=12000 micro-cycles)

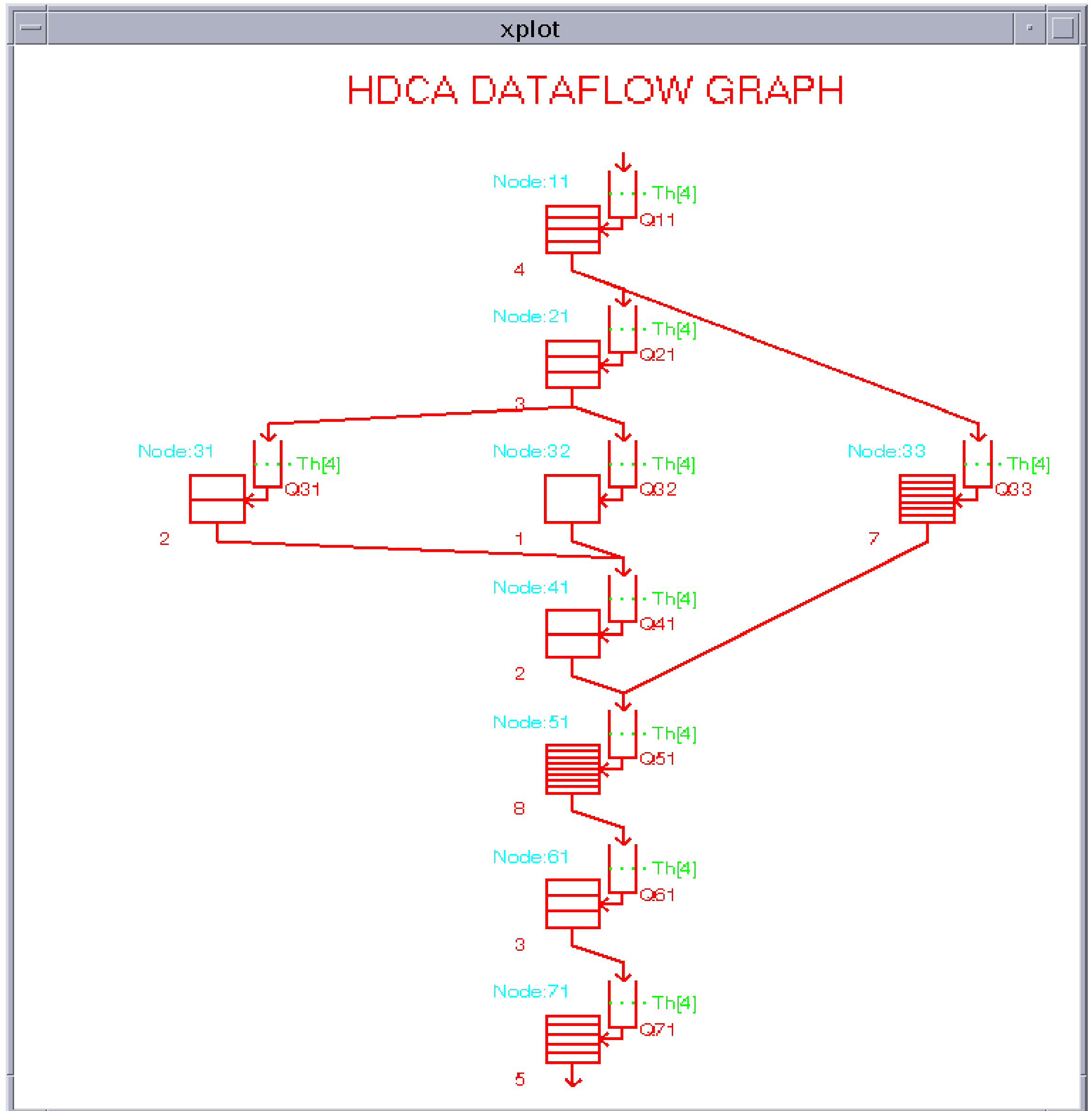


Figure 5.4 m   Simu. Results of Application 1 (Input Rate=100micro-cycles/token) (t=13000 micro-cycles)

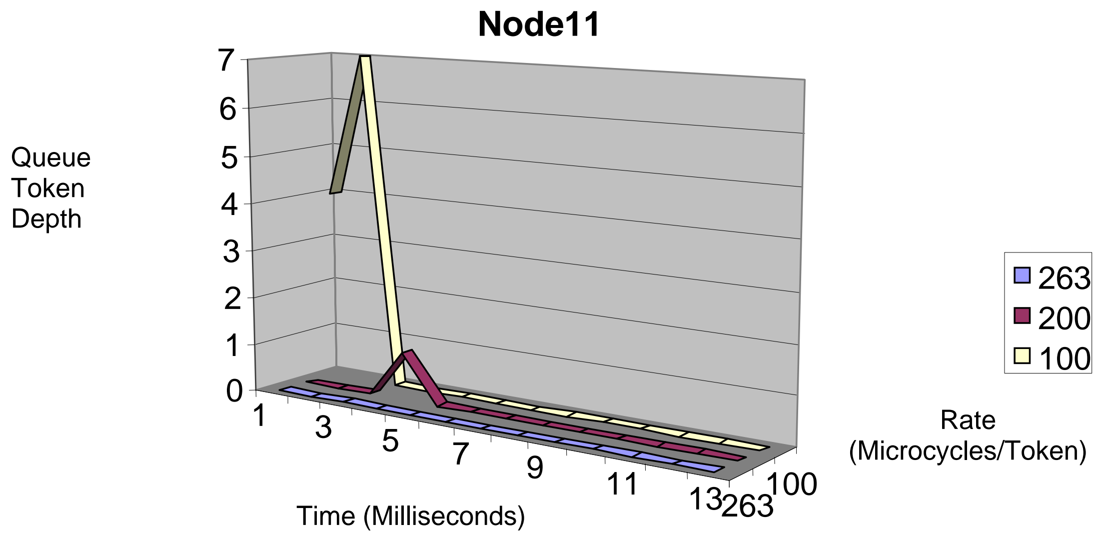
Total Simulation Time = 12132 Micro-cycles

Table 5-3 shows all the relevant data at each node at different input rates and the different simulation times. The first row represents the time in “microcycles”. The first colum is the node name, with the number of initial copies of a processor which is calculated by “COPY” inside the parenthesis. The first number in the table is the queue token depth; the second number represents how many processors are busy; and the third number is the number of extra processor copies. If the third number is 0, then the threshold of the queue is 4. Then the threshold will increase by two each time the extra copy number increases by one. So if the third number is 2, two extra copies of the processor have been initiated to help reduce the queue depth, and meantime, the threshold of this queue has changed to 8.

In order to show that the HDCA architecture is very sensitive to the input rate, the queue token depth has been plotted at the three different input rates at each node. The plots are shown in Figure 5.5. Plots for nodes 21, 31, and 32 are omitted because the queue depth at these nodes, and 32 is always 0. One thing worth mentioning is that among three different input rates: 263 microcycles/token, 200 microcycles/token, and 100 microcycles/token, 100 microcycles/token is the highest input rate, while 263 microcycles/token is the lowest input rate. According to table 5-2, the input rate at peak load is 3.8 DI/milliseoconds, that is 3.8 tokens per 1000 microcycles. By inverting this number we get 1000/3.8 = 263.158 microcycles/token. So by using the input rate of 263 micro-cycles/token, which is slightly over the maximum input rate, theoretically we should get a very smooth dataflow graph without any tokens in the queue or very few tokens in the queue. By looking at the plots in Figure 5.5, we can see the queue token depth increases as the input rate increases. This is an indication that this simulator works correctly for this application.

Table 5-3: Application 1 Results: 20 Tokens

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000
Input rate = 263 Micro-cycles/Token													
Node11(4)	0/3/0	0/3/0	0/3/0	0/4/0	0/4/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(3)	0/0/0	0/1/0	0/2/0	0/3/0	0/3/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node31(2)	0/0/0	0/0/0	0/0/0	0/0/0/	0/1/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node32(1)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node33(7)	0/1/0	0/4/0	0/7/0	0/6/0	0/6/0	1/7/2000	0/6/0	0/3/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node41(2)	0/0/0	0/0/0	0/0/0	0/1/0	0/0/0	0/2/0	0/2/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node51(8)	0/0/0	0/0/0	0/0/0	0/2/0	0/6/0	0/6/0	0/5/0	1/8/0	0/8/0	0/3/0	0/0/0	0/0/0	0/0/0
Node61(3)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/2/0	0/3/0	0/1/0	0/2/0	0/3/0	0/2/0	0/0/0	0/0/0
Node71(5)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/4/0	0/4/0	0/3/0	0/5/0	0/5/0	0/0/0	0/0/0
Input rate = 200 Micro-cycles/Token													
Node11(4)	0/4/0	0/4/0	0/4/0	1/4/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(3)	0/0/0	0/1/0	0/1/0	0/1/0	0/2/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node31(2)	0/0/0	0/0/0	0/1/0	0/2/0	0/1/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node32(1)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node33(7)	0/1/0	0/5/0	2/7/0	3/7/0	3/7/0	2/7/0	0/6/0	0/2/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0
Node41(2)	0/0/0	0/0/0	0/0/0	0/0/0	0/1/0	0/2/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node51(8)	0/0/0	0/0/0	0/0/0	0/2/0	0/6/0	0/6/0	0/7/0	2/8/0	0/5/0	0/2/0	0/1/0	0/0/0	0/0/0
Node61(3)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/2/0	0/2/0	0/0/0	1/3/0	1/3/0	0/1/0	0/0/0	0/0/0
Node71(5)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/4/0	0/5/0	0/2/0	0/4/0	0/4/0	0/2/0	0/0/0
Input rate = 100 Micro-cycles/Token													
Node11(4)	4/4/0	7/6/2	0/6/2	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(3)	0/1/0	0/1/0	0/3/0	0/3/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node31(2)	0/0/0	0/0/0	0/1/0	0/0/0	0/2/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node32(1)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0/	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node33(7)	0/1/0	0/6/0	3/7/0	4/8/1	0/8/1	0/8/1	0/4/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node41(2)	0/0/0	0/0/0	0/0/0	0/2/0	0/0/0	1/2/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node51(8)	0/0/0	0/0/0	0/0/0	0/3/0	1/8/0	0/6/0	0/7/0	3/8/0	0/4/0	0/1/0	0/0/0	0/0/0	0/0/0
Node61(3)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/3/0	1/3/0	0/0/0	1/3/0	1/3/0	0/0/0	0/0/0	0/0/0
Node71(5)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/4/0	1/5/0	0/4/0	0/4/0	1/5/000	0/1/0	0/0/0



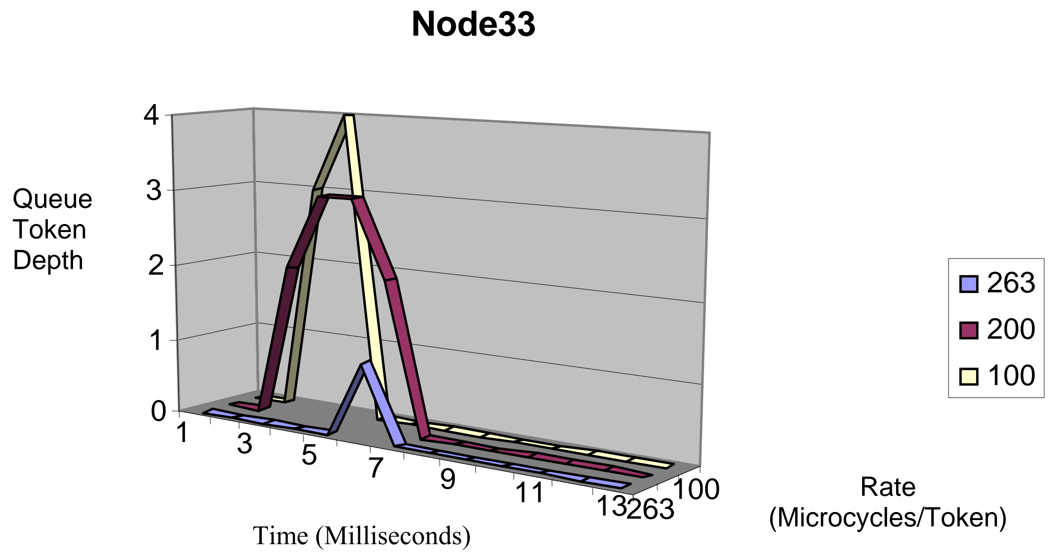
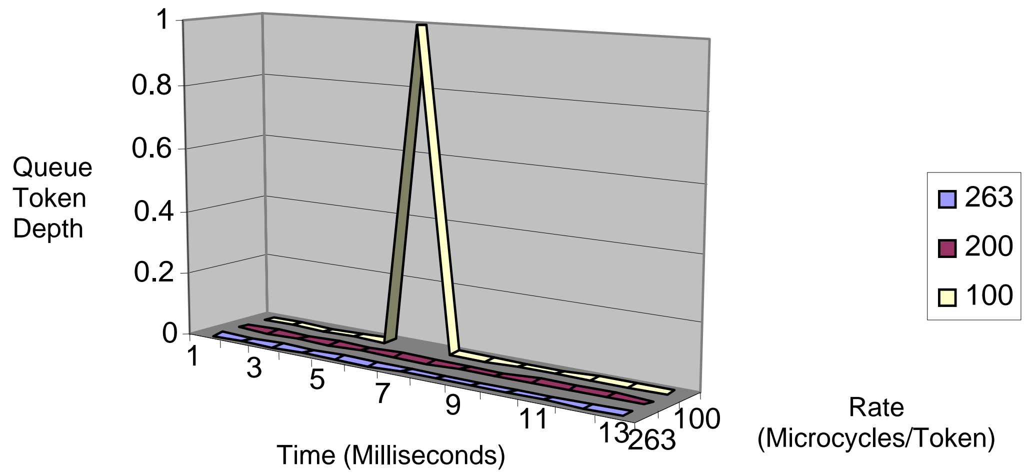


Figure 5.5 Queue Depth Plot for Application 1



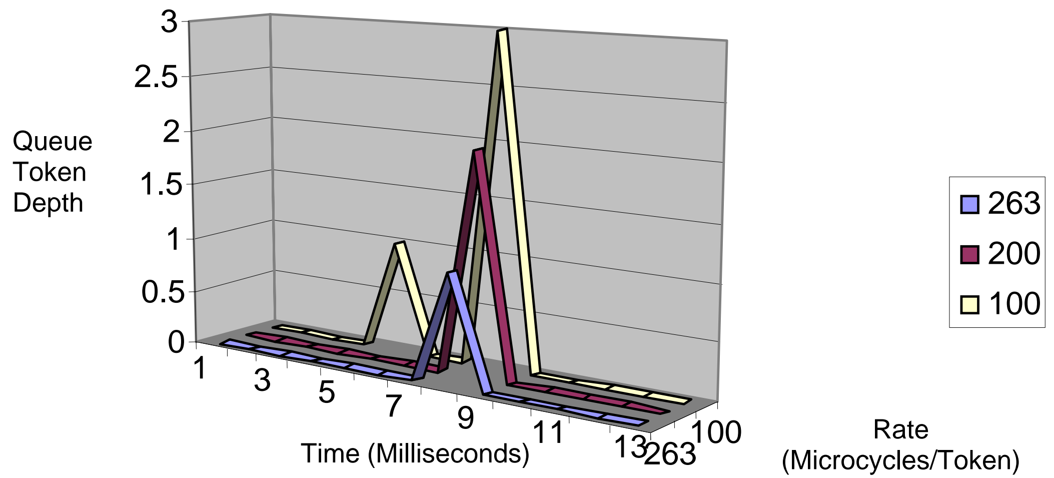
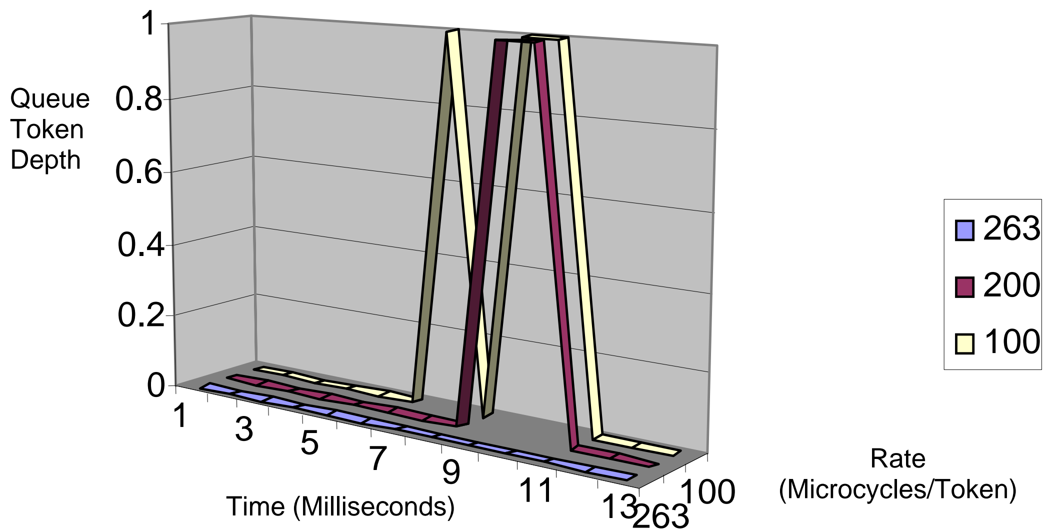


Figure 5.5 Queue Depth Plot for Application 1 (Continue 2)



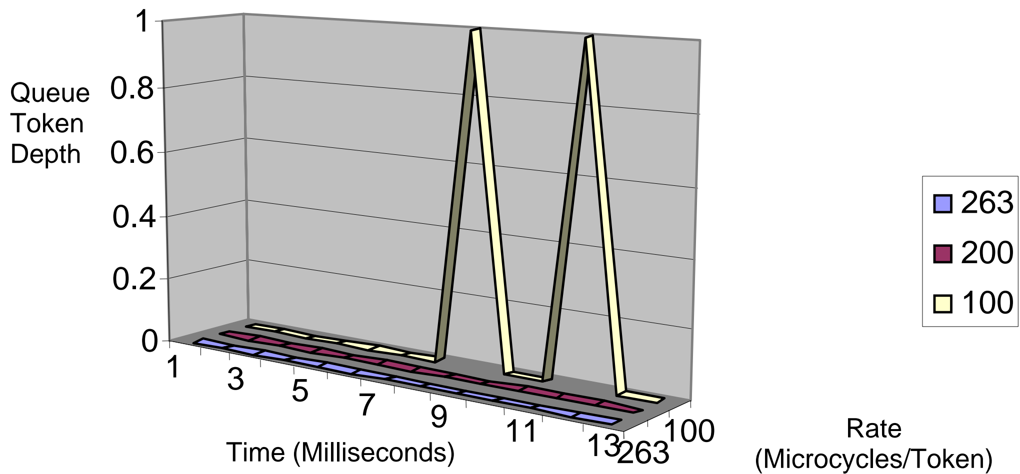


Figure 5.5 Queue Depth Plot for Application 1 (Continue 3)

5.2 Application 2

Application 2 is represented by the dataflow graph that is shown in Figure 5.6. This graph has two top nodes that take input data for the system. Table 5-4 shows all the parameters that are needed to do the simulation.

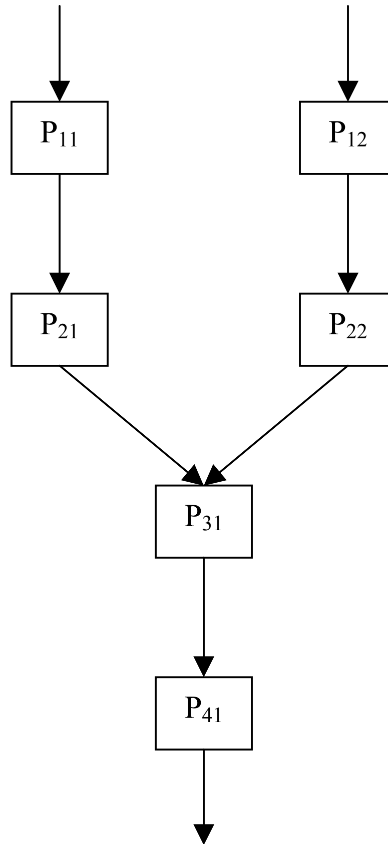


Figure 5.6 Data Flow Graph of Case 2

Table 5-4: **Parameter Values for Data Flow Graph of Application 2**

Process Designation	Execution Time (milliseconds)	Process Length (Kilobytes)
11	0.85	0.425
12	1.87	0.45
21	1.63	2.5
22	0.69	10.0
31	1.0	0.5
41	0.96	0.85

Input Data Rates (data items/millisecond)

Peak Load at Node 11: 3.8

Average Load at Node 11: 2.5

Peak Load at Node 12: 3.8

Average Load at Node 12: 2.5

Program Memory/Computing Element: 16 kilobytes

Copyset:

2 4 1 1 2 1 3 1 4 1 4 1 2 2 2 3 1 4 1 0.85 0.425 1.63 2.5 1.0 0.5 0.96
0.85 1.87 0.45 0.69 10.0 16 3.8 2.5 3.8 2.5 1 1

Dataset:

4 2 2 1 1 1 1 2 1 2 1 3 1 3 1 4 1 1 2 2 2 2 2 3 1 0 0 0 0

Informationset:

0 0 0 0 0 2

Since there is no fork in this graph, "probabilityset" is not needed.

Figures 5.7, 5.8 and 5.9 are the simulation results that are taken every 1000 micro-cycles at the input rates of 263, 200 and 100 micro-cycles per token for both top nodes. There are 20 data items at each top node. So there are 40 tokens that need to be processed altogether. Table 5-5 shows the queue depth, number of free copies, and the number of extra copies of each node at different times based on the Figures 5.7, 5.8 and 5.9. Plots for queue depth at different input rates and different simulation times for all nodes are shown in Figure 5.10.

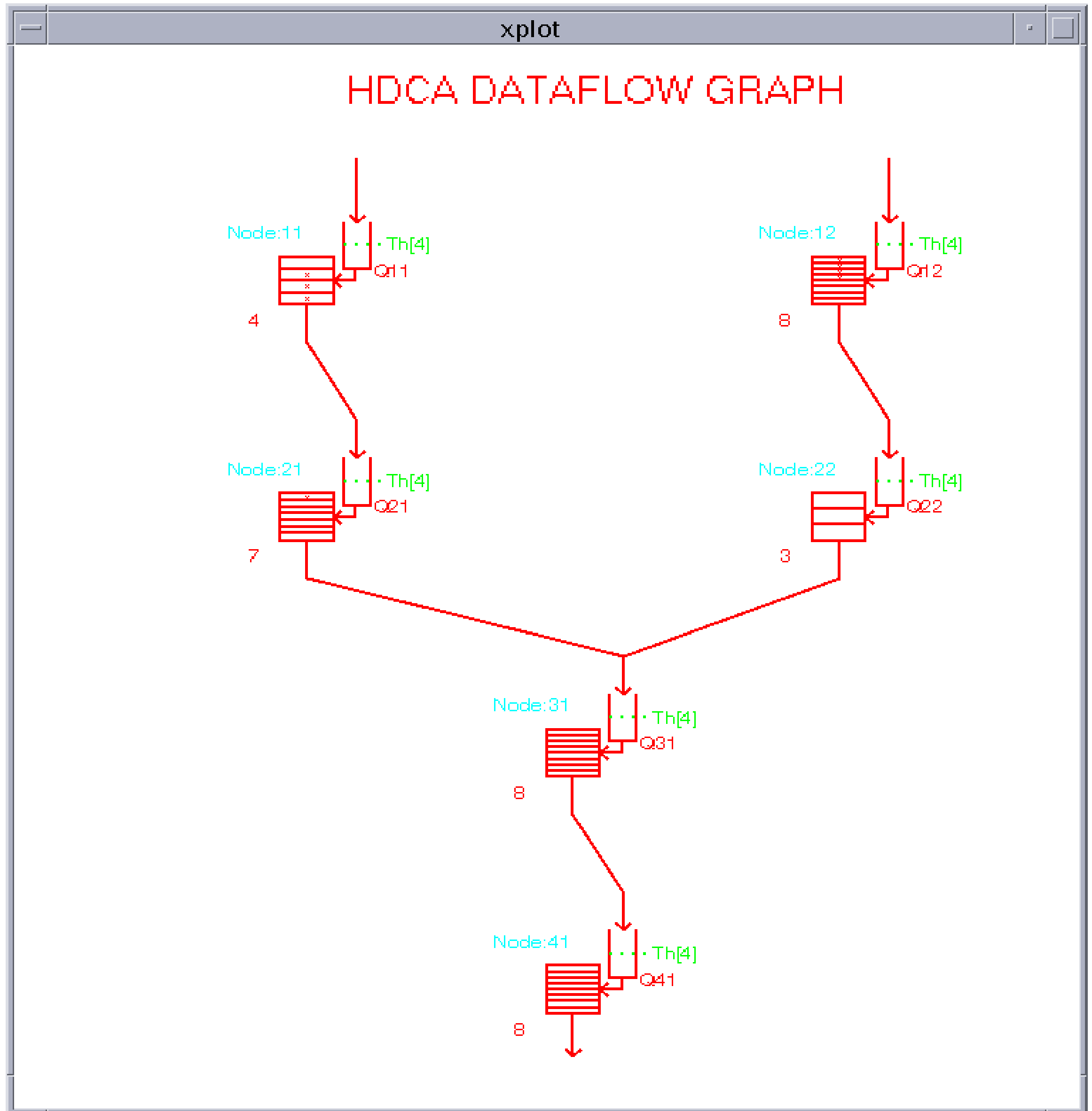


Figure 5.7 a Simu. Results of Application 2 (Input rate = 263 micro-cycles/token) (t =1000 micro-cycles)

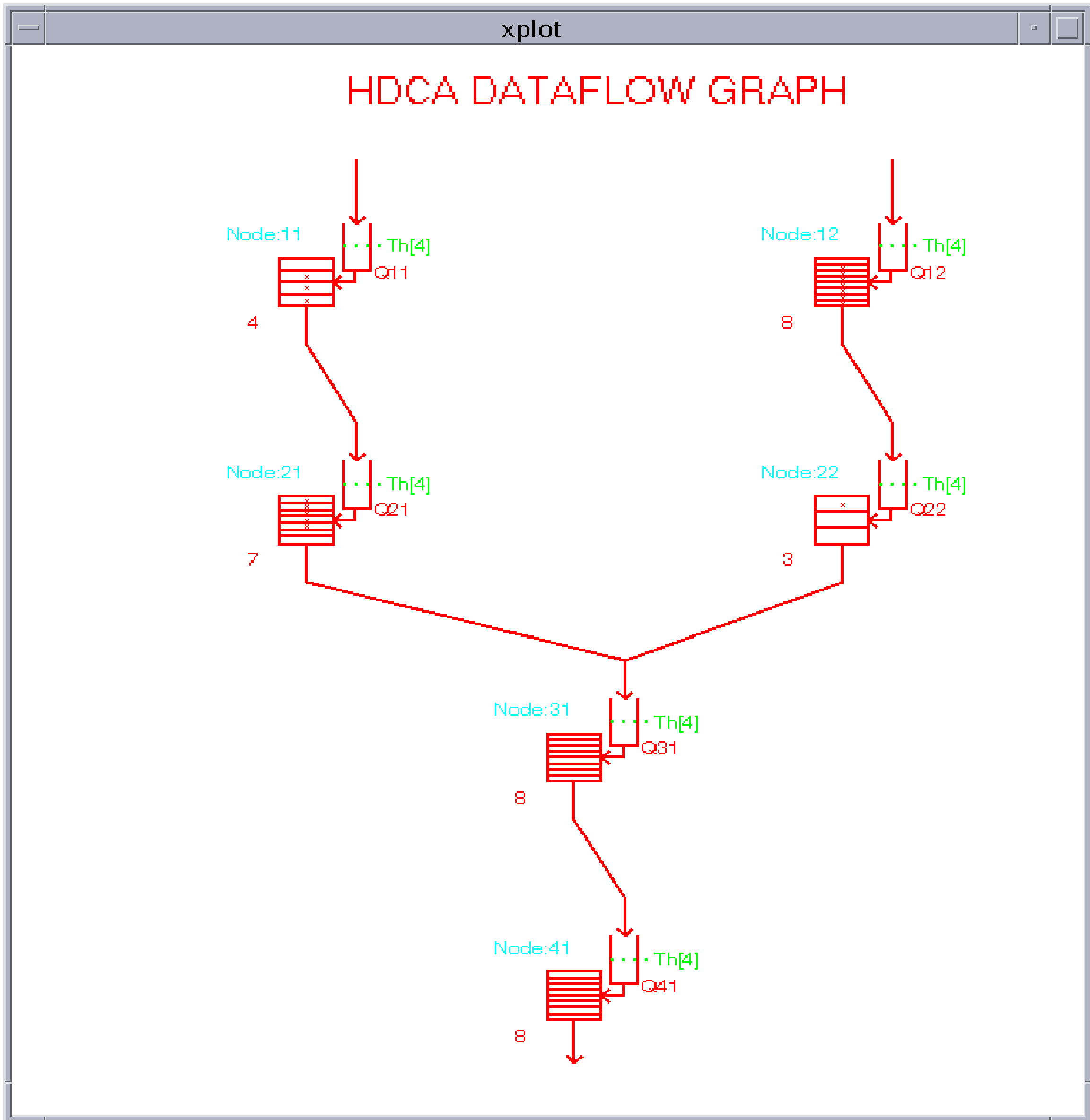


Figure 5.7 b Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)

(t =2000 micro-cycles)

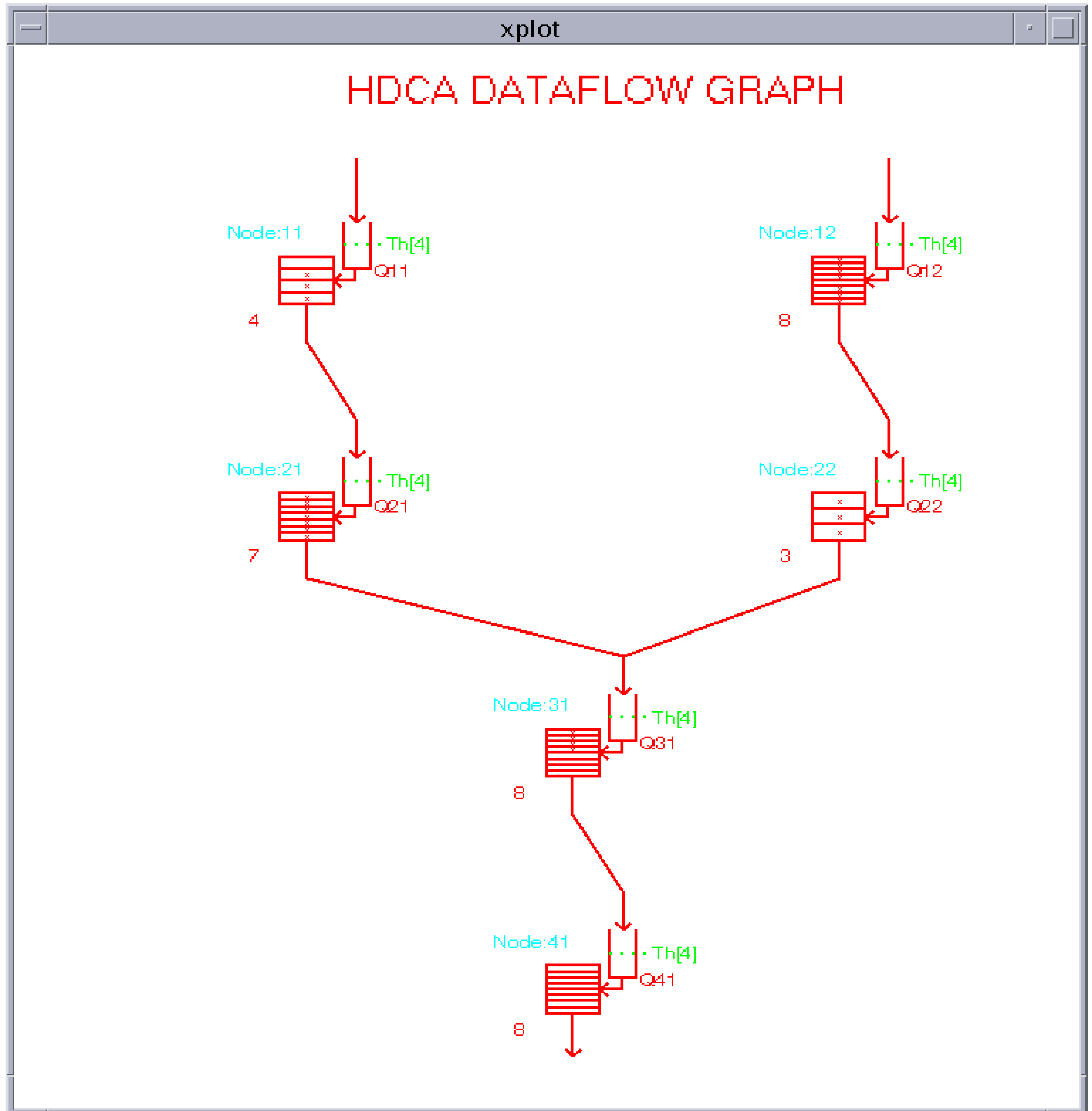


Figure 5.7 c Simu. Results of Application 2 (Input rate = 263 micro-cycles/token) (t =3000 micro-cycles)

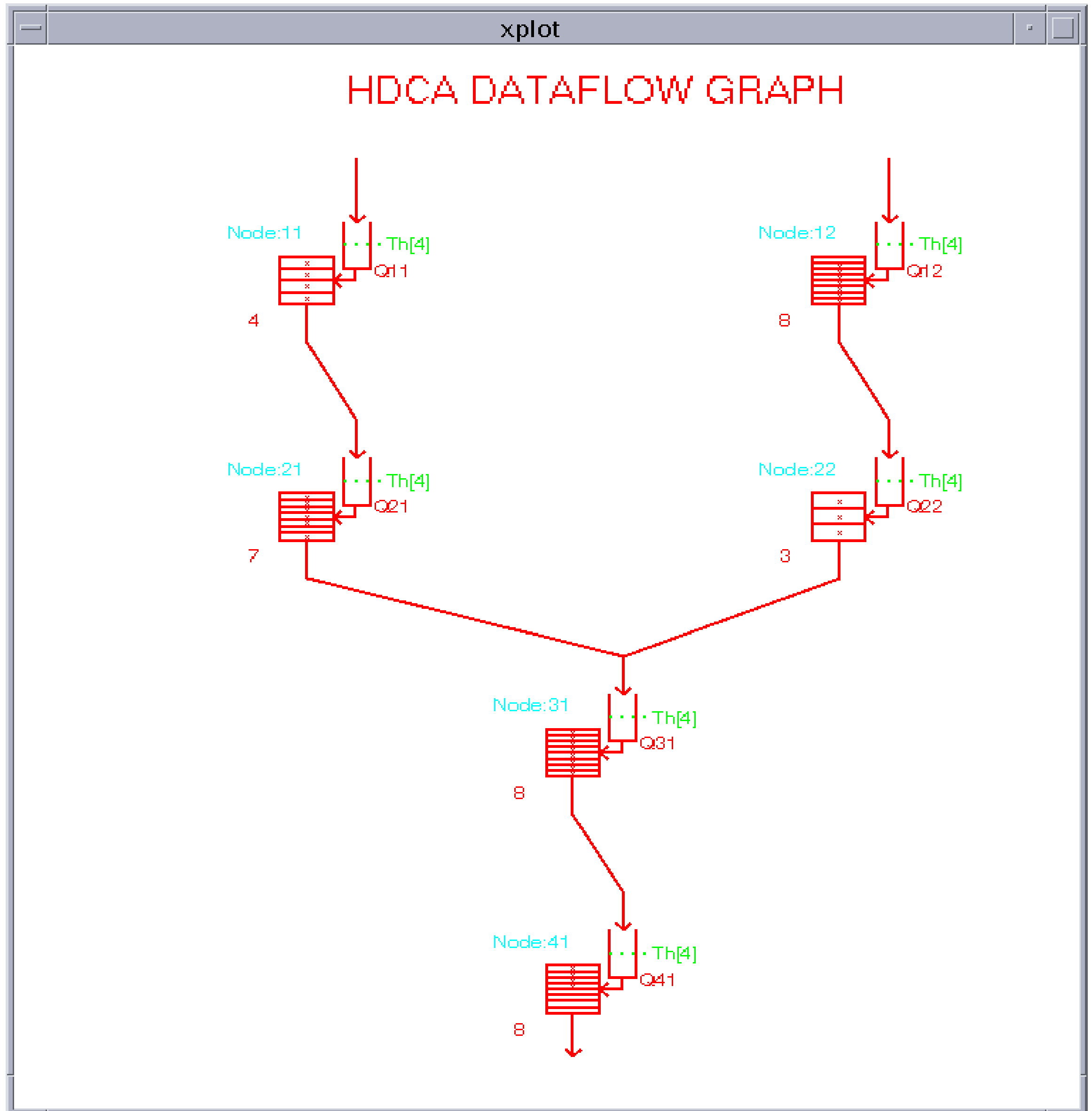


Figure 5.7 d   Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)

(t =4000 micro-cycles)

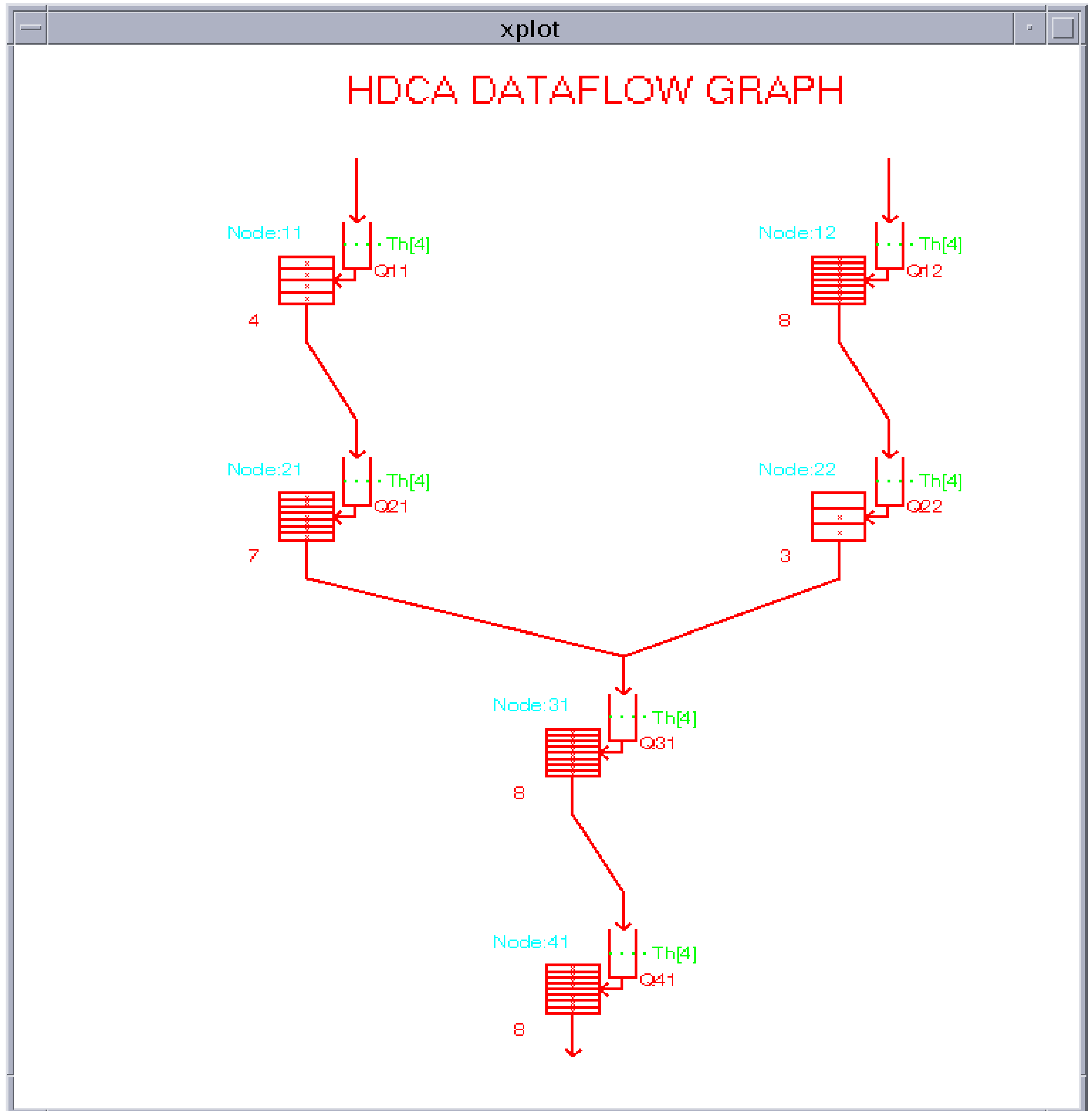


Figure 5.7 e Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)

(t =5000 micro-cycles)

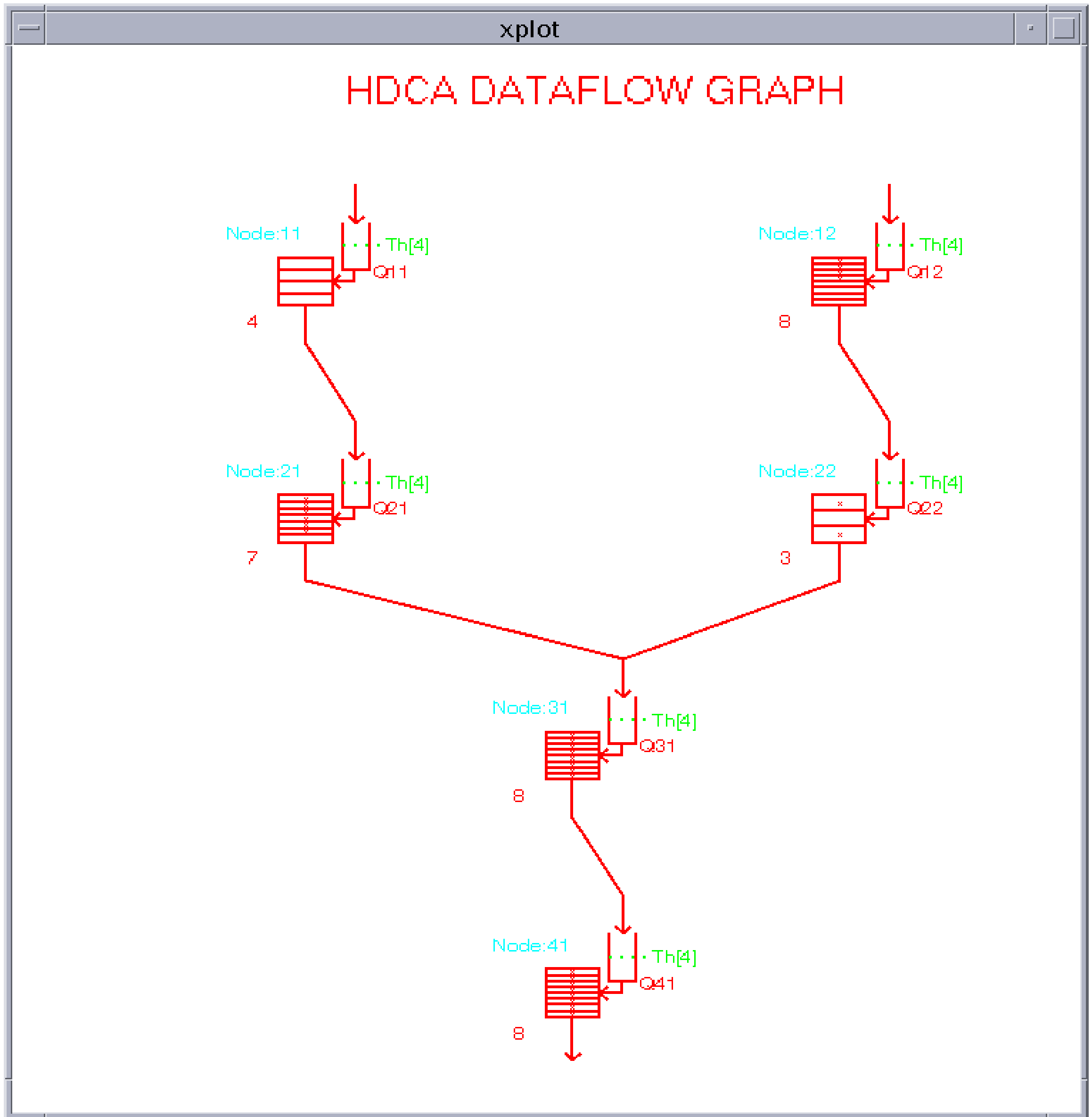


Figure 5.7 f Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)

(t =6000 micro-cycles)

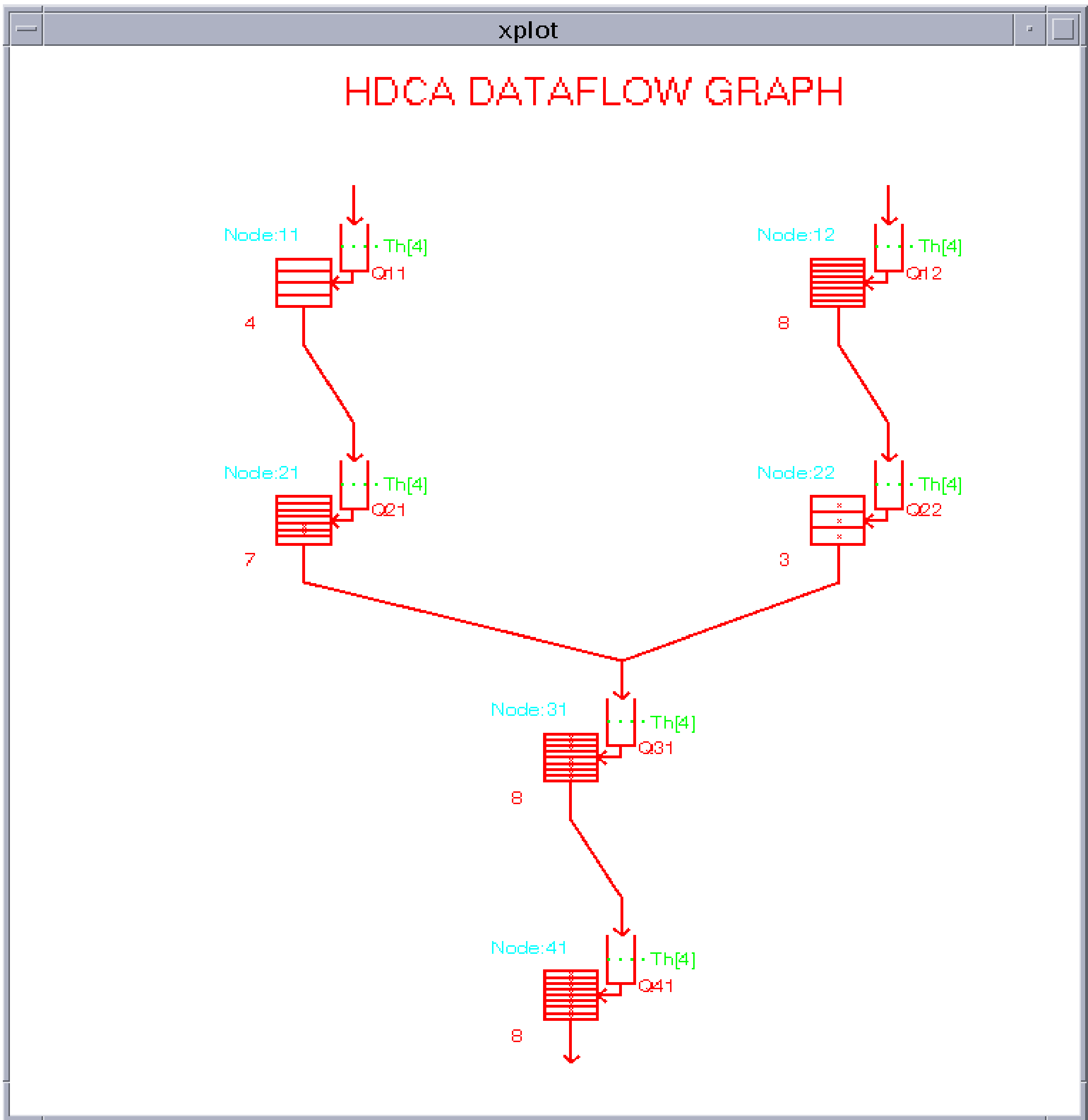


Figure 5.7 g Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)

(t =7000 micro-cycles)

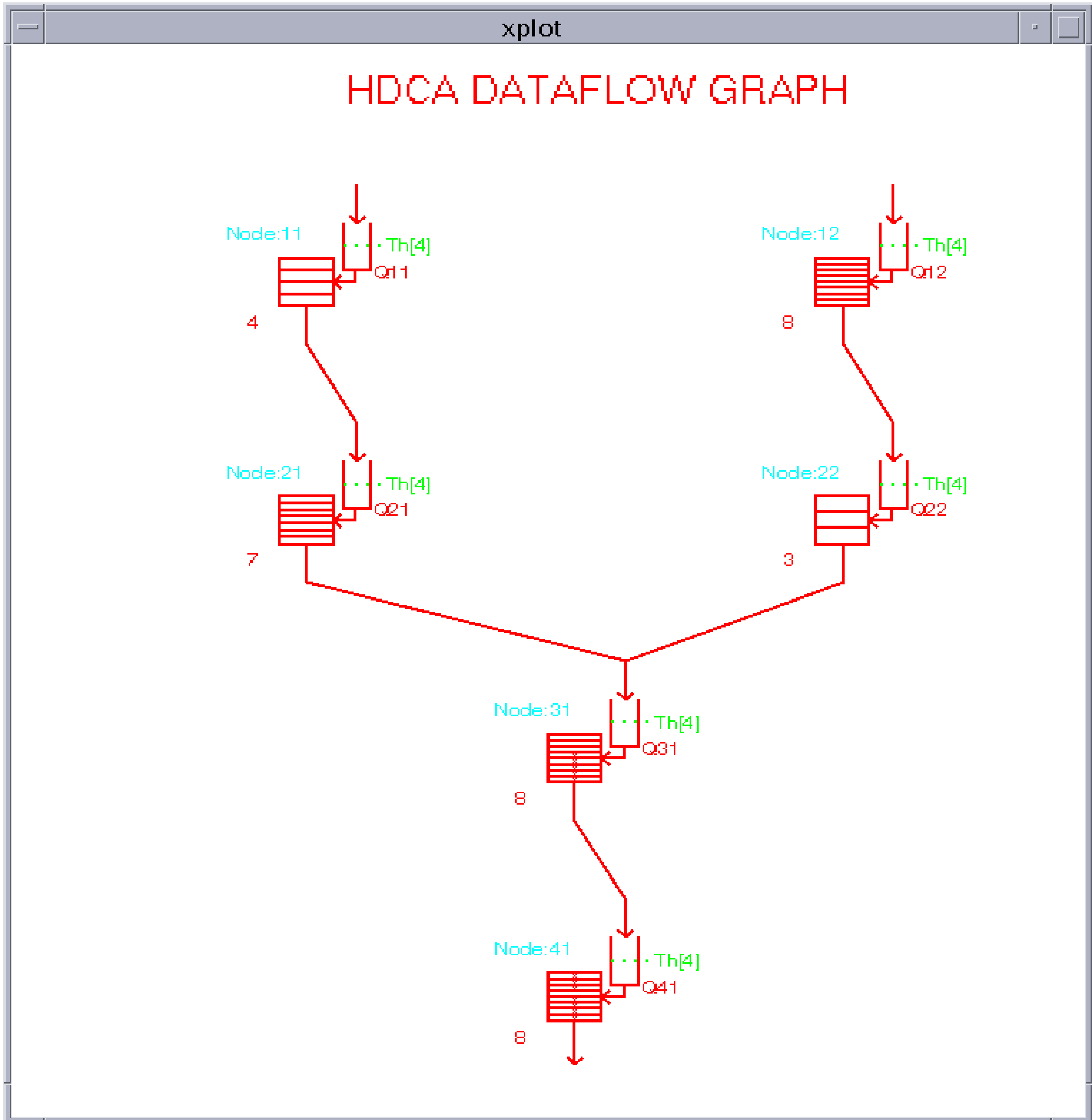


Figure 5.7 h   Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)
(t =8000 micro-cycles)

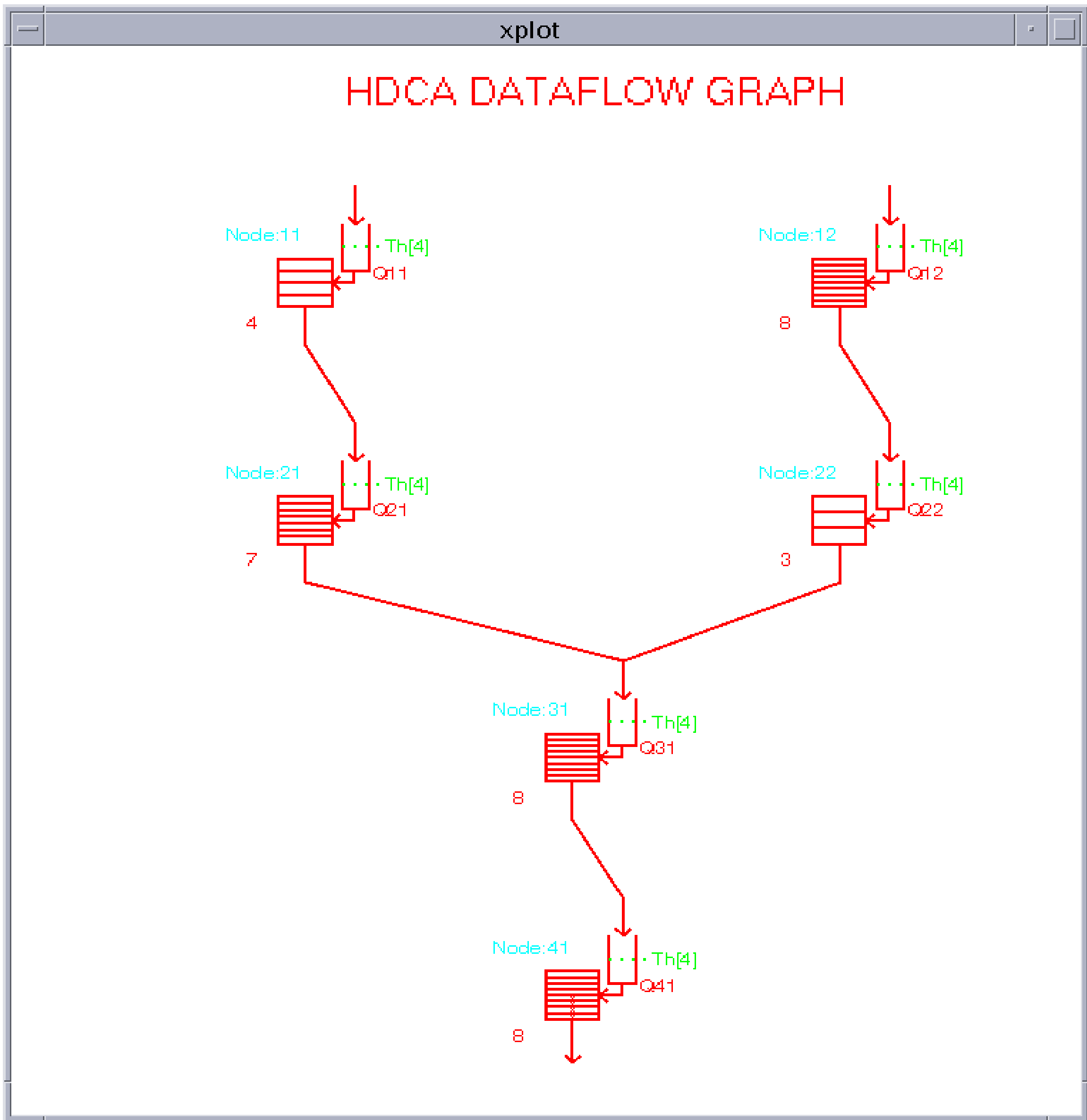


Figure 5.7 i Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)

(t =9000 micro-cycles)

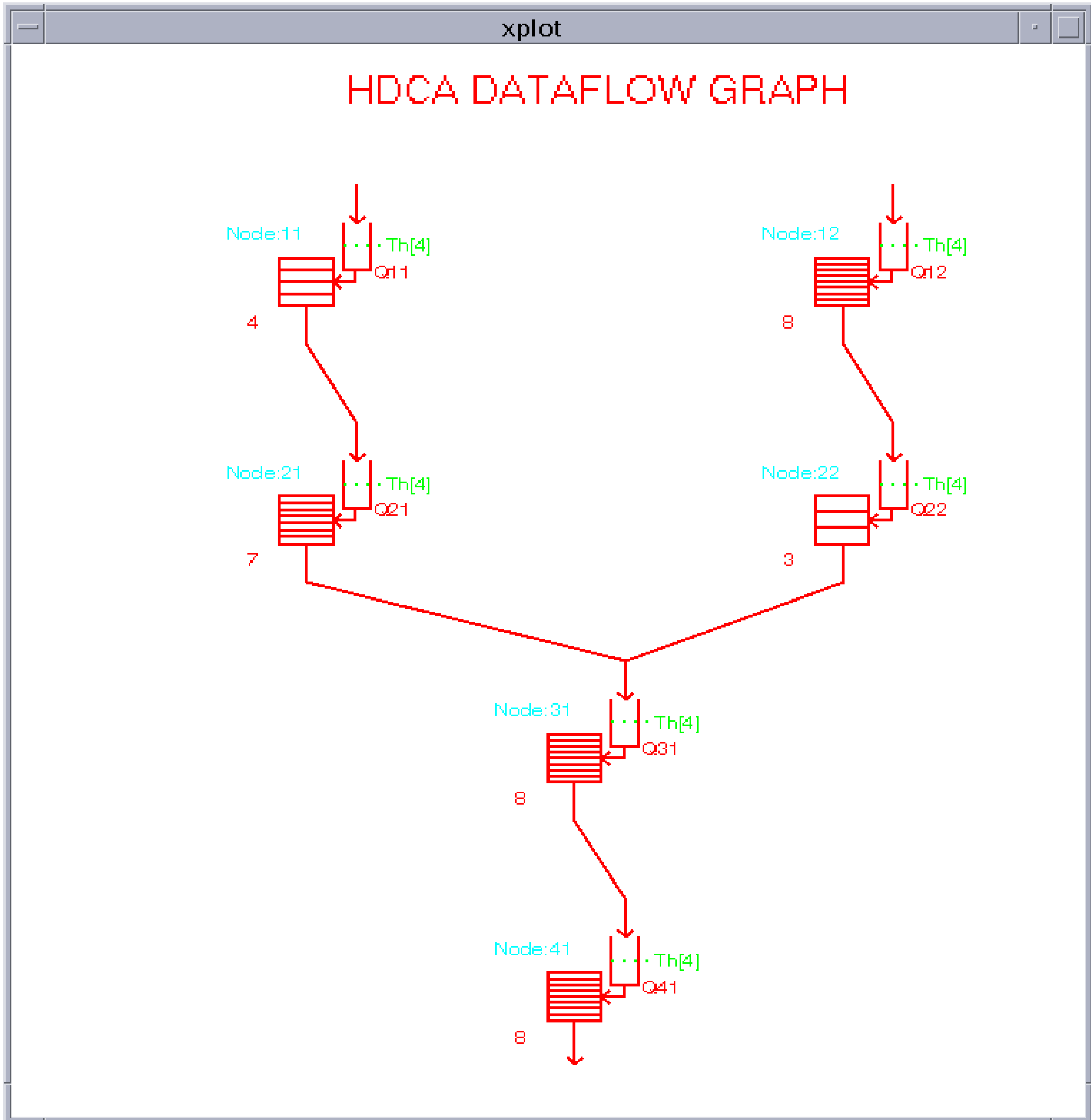


Figure 5.7 j Simu. Results of Application 2 (Input rate = 263 micro-cycles/token)
(t =10000 micro-cycles)

Total Simulation Time = 9518 Micro-cycles

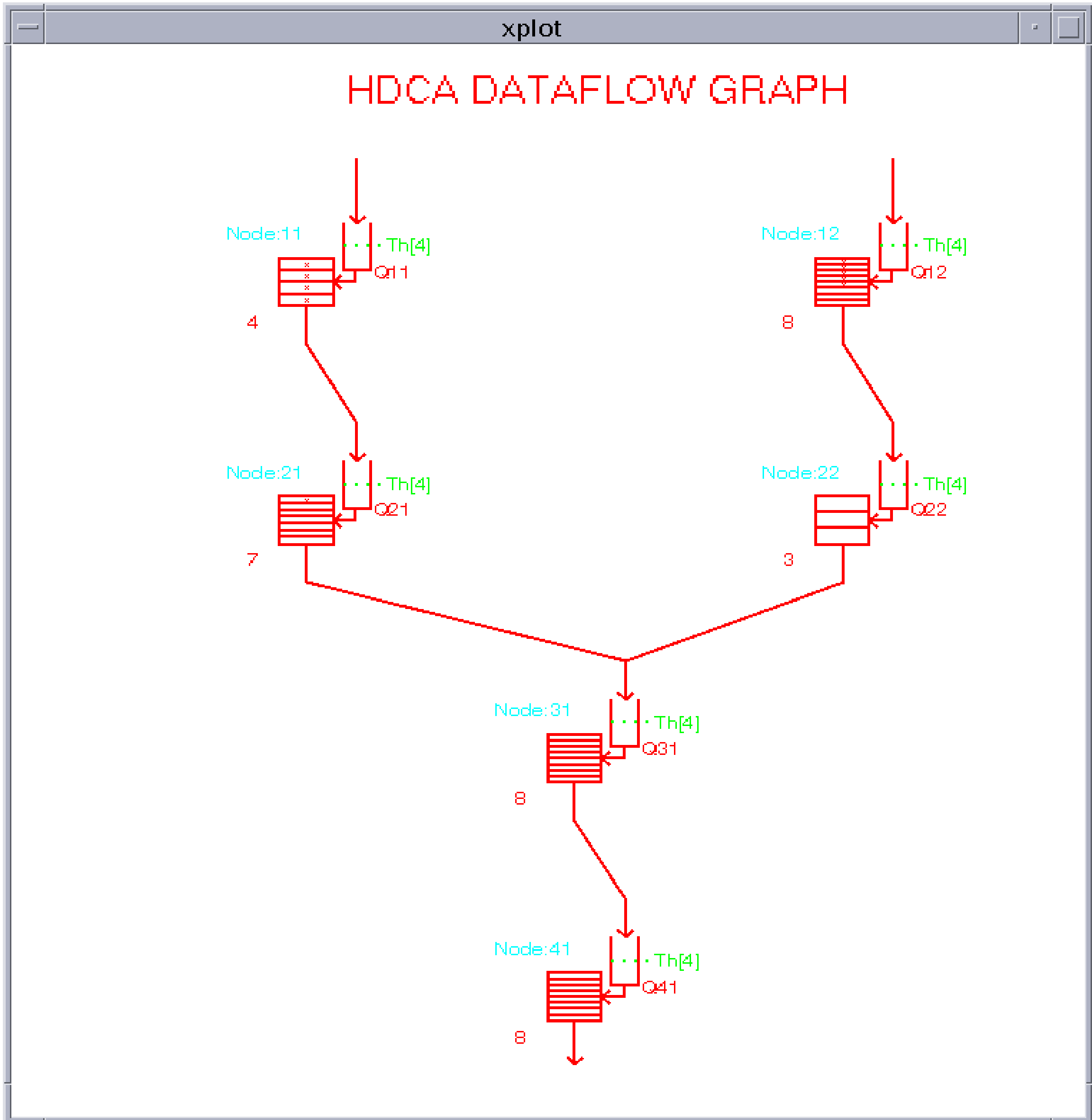


Figure 5.8 a Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =1000 micro-cycles)

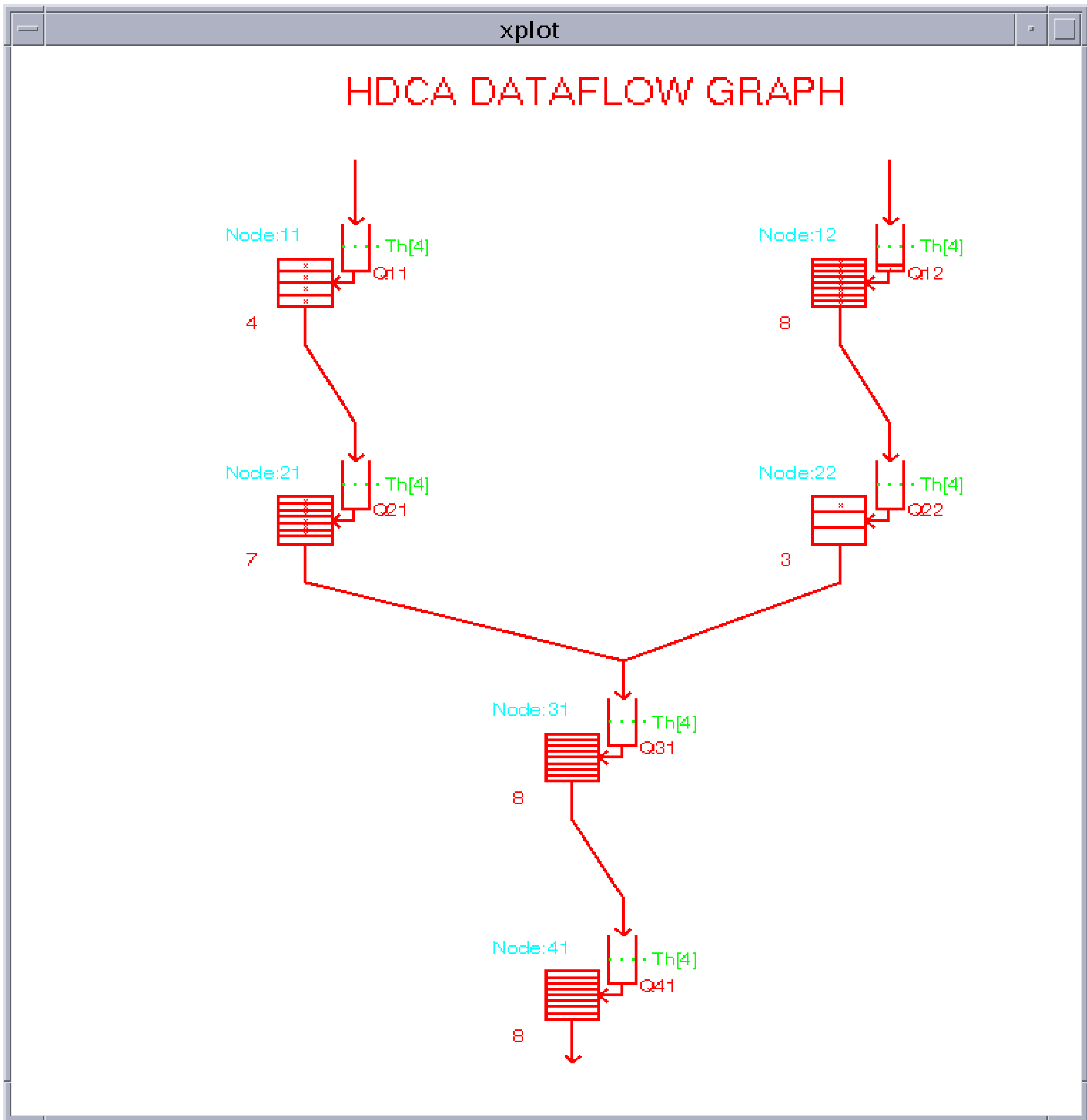


Figure 5.8 b Simu. Results of Application 2 (Input rate = 200 micro-cycles/token)

(t =2000 micro-cycles)

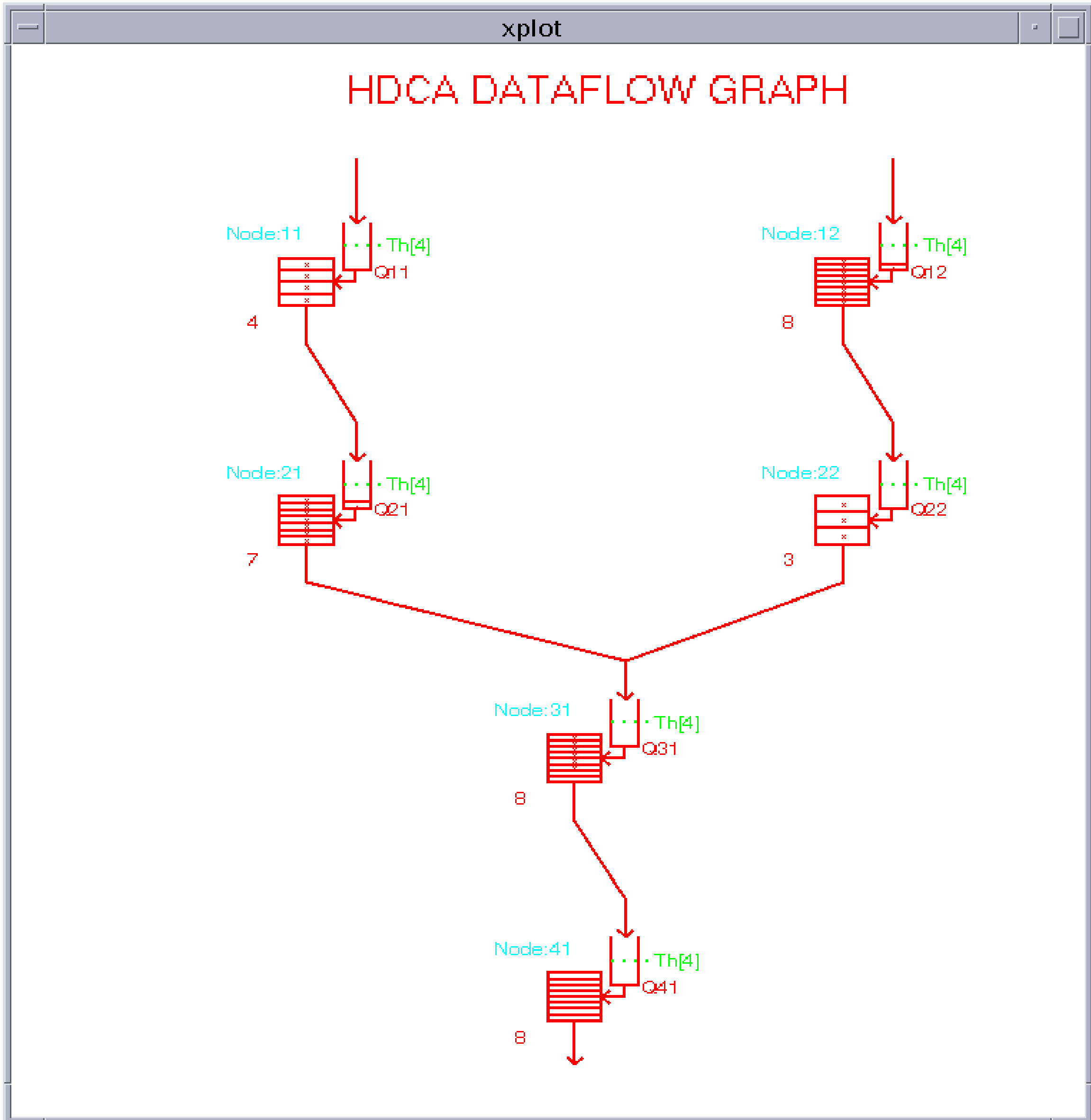


Figure 5.8 c Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =3000 micro-cycles)

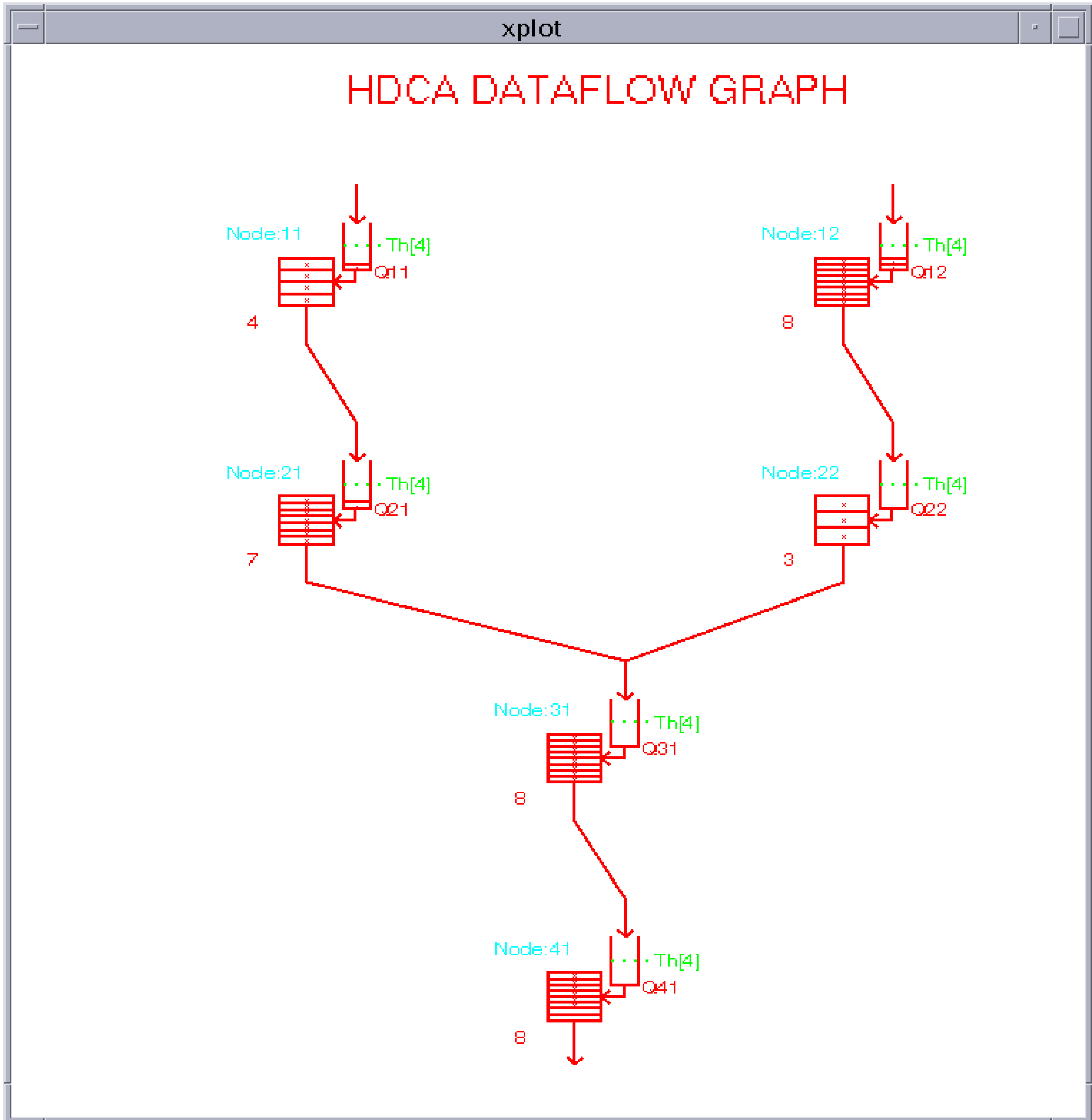


Figure 5.8 d Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =4000 micro-cycles)

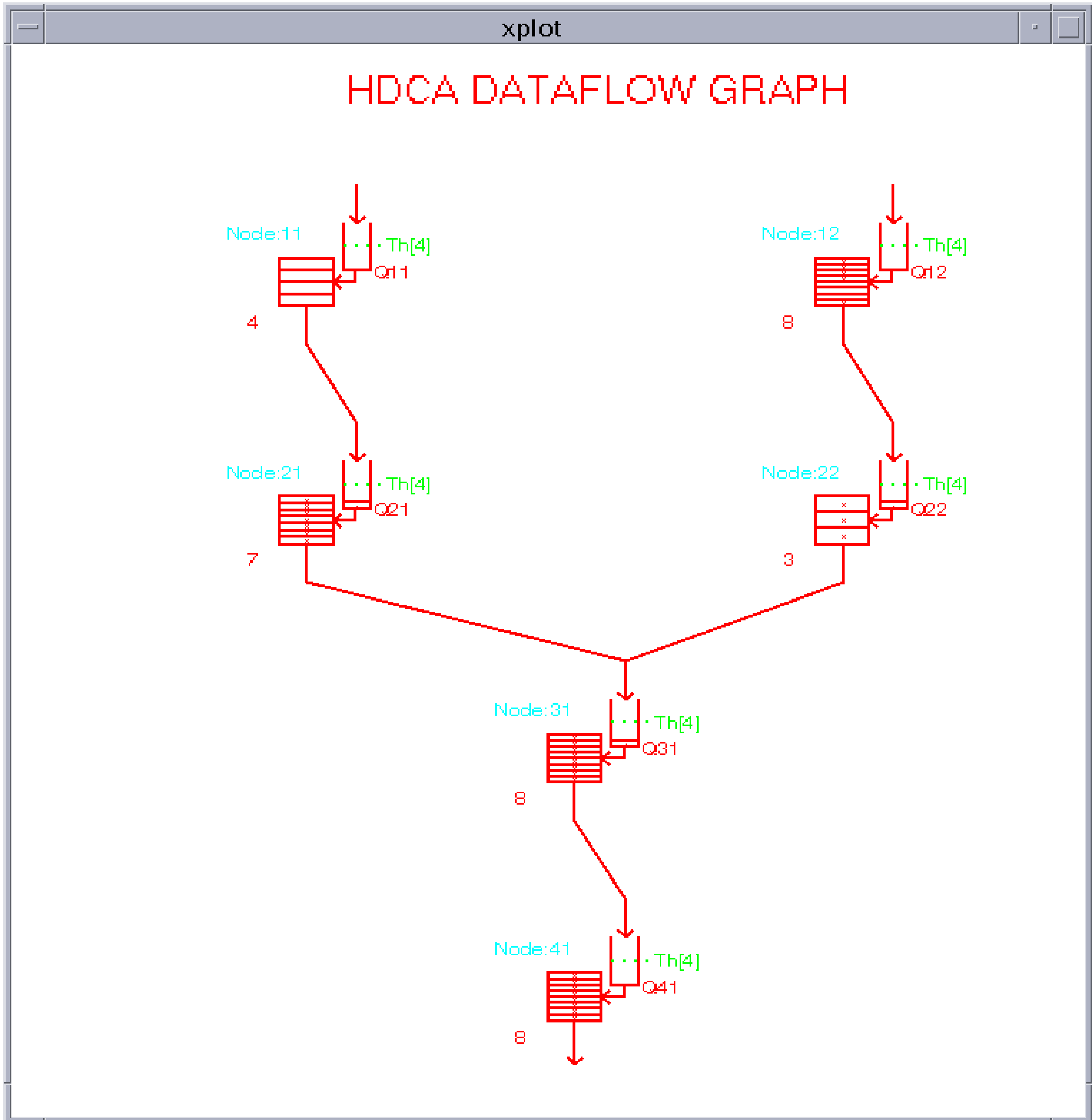


Figure 5.8 e Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =5000 micro-cycles)

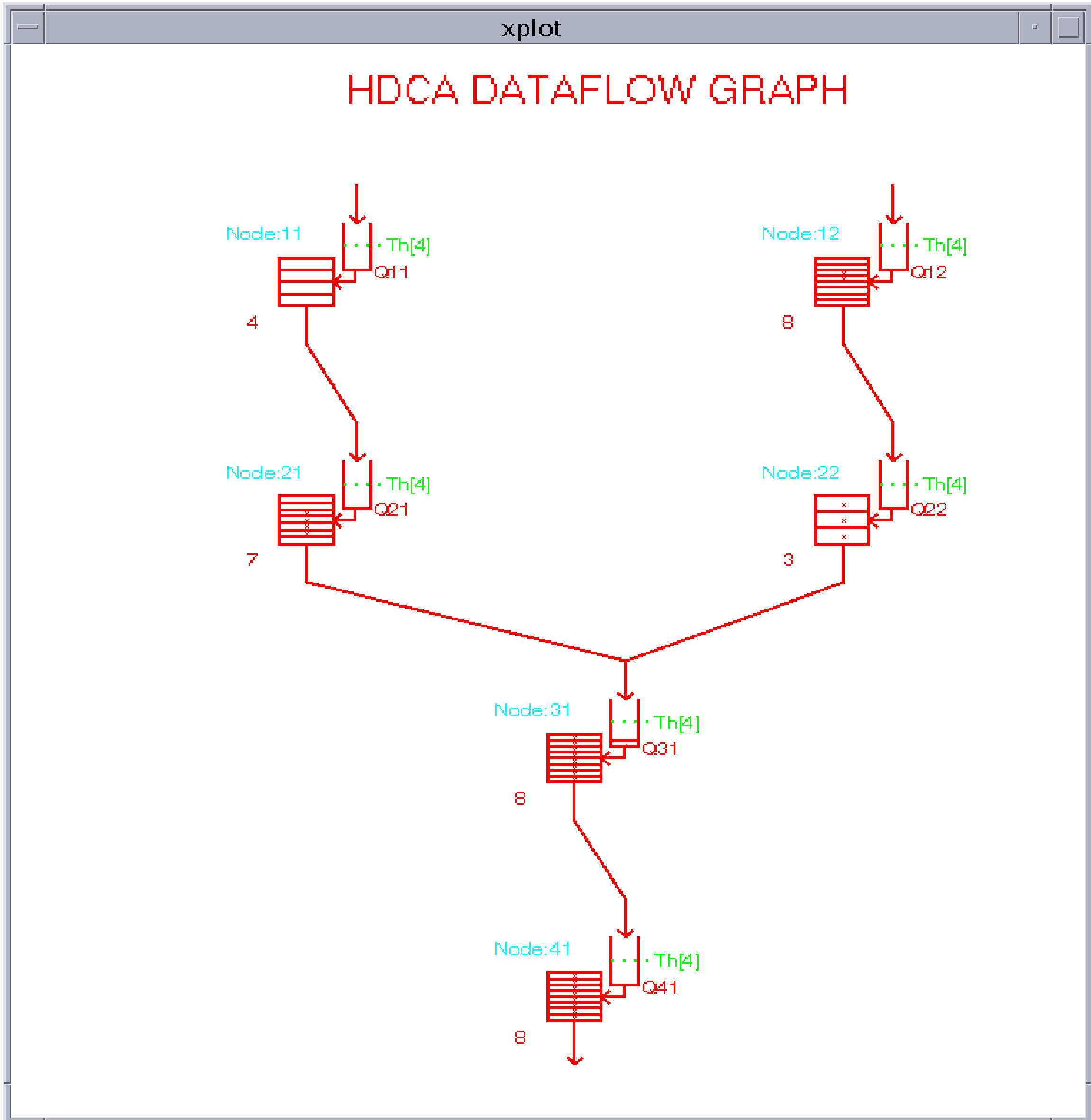


Figure 5.8 f Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =6000 micro-cycles)

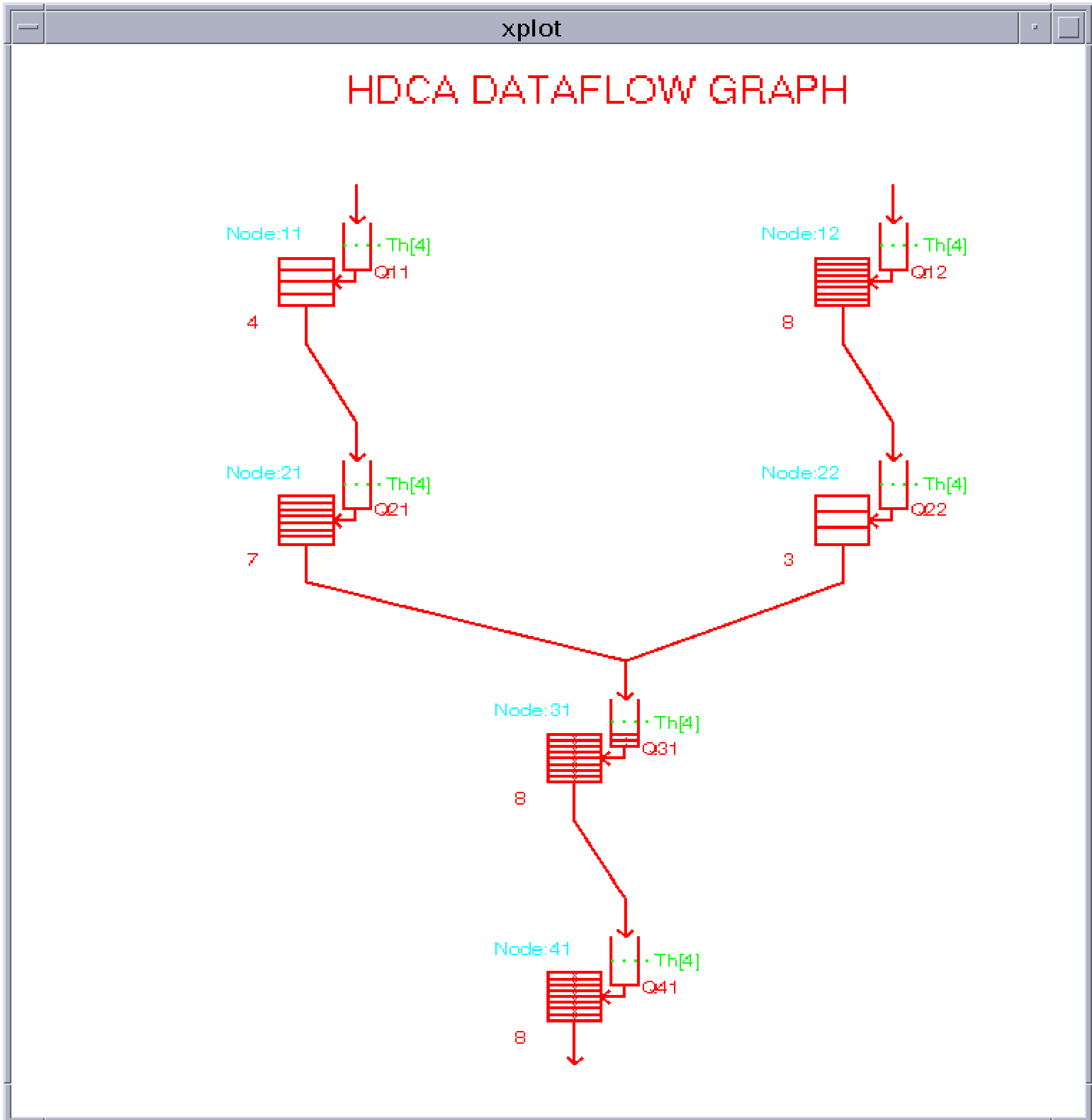


Figure 5.8 g Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =7000 micro-cycles)

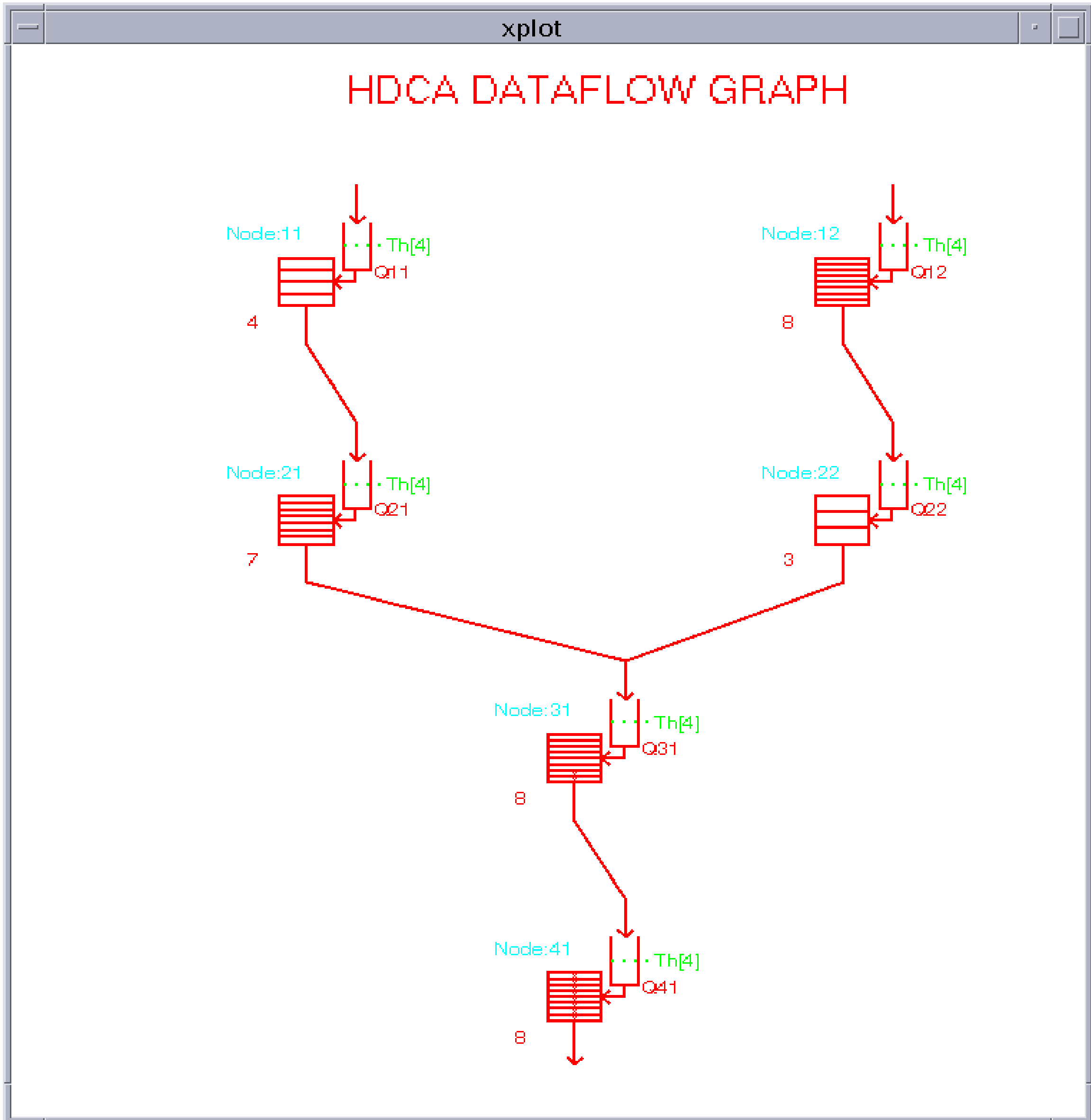


Figure 5.8 h Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =8000 micro-cycles)

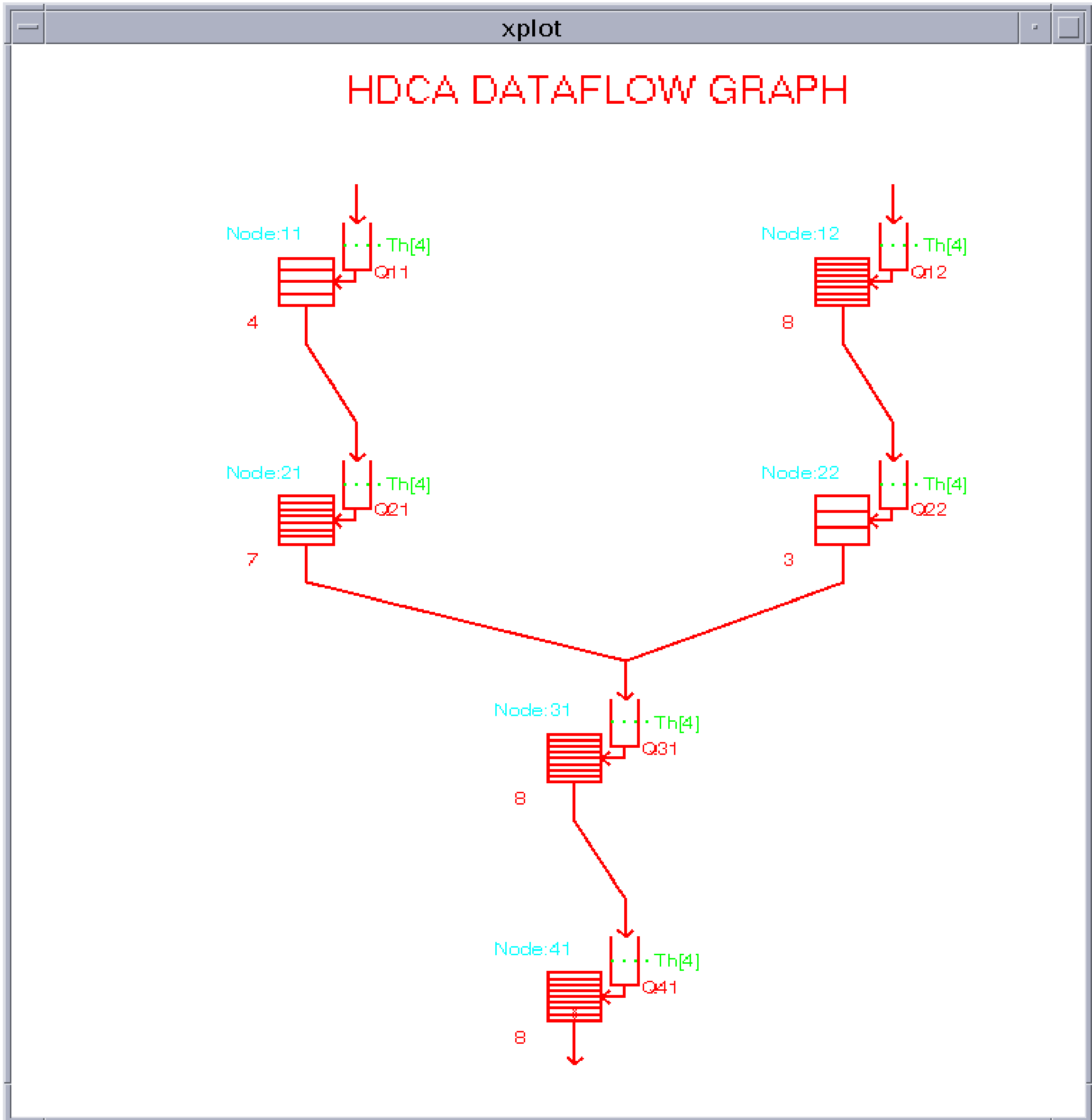


Figure 5.8 i Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =9000 micro-cycles)

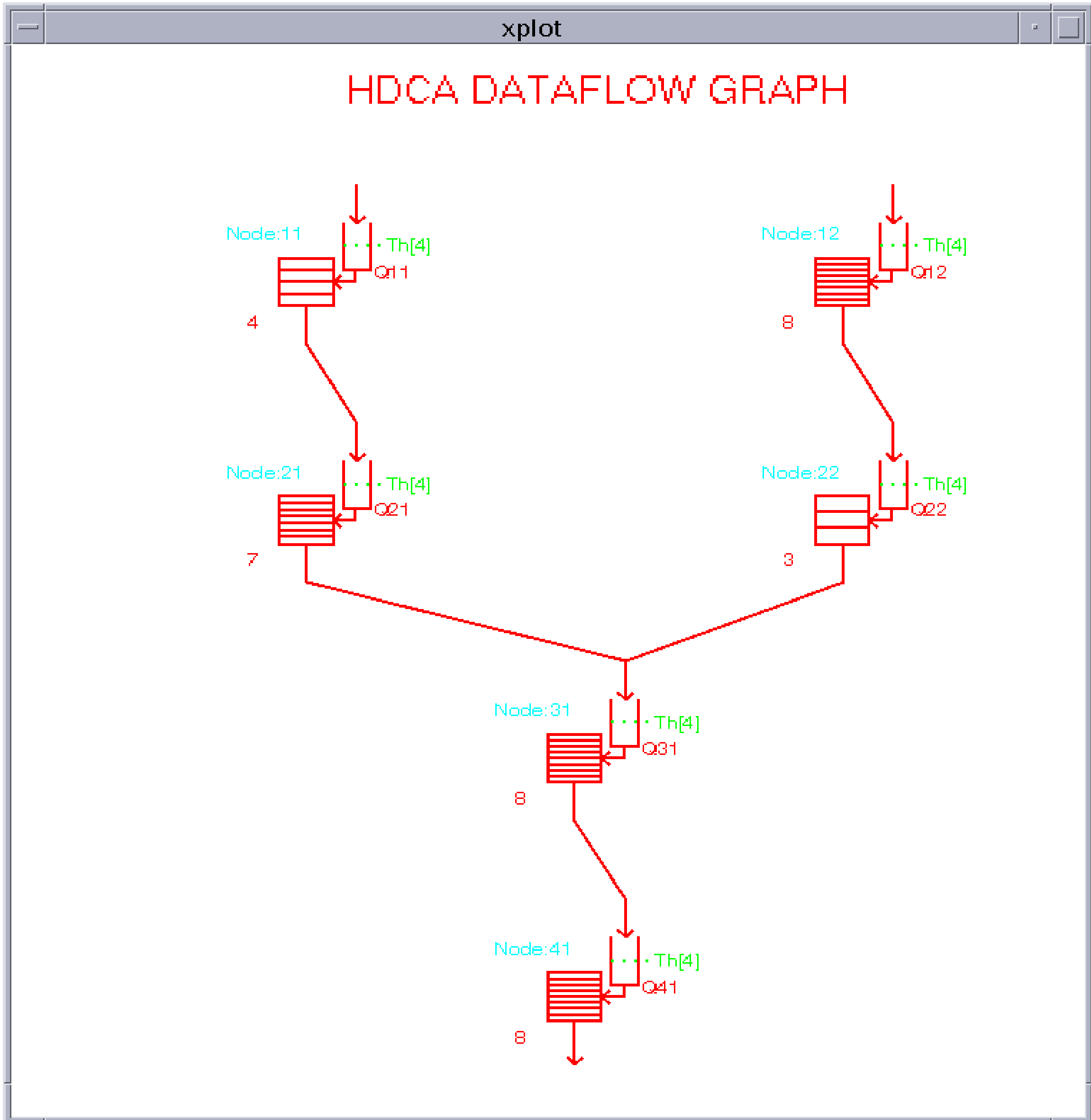


Figure 5.8 j Simu. Results of Application 2 (Input rate = 200 micro-cycles/token) (t =10000 micro-cycles)

Total Simulation Time = 9216 Micro-cycles

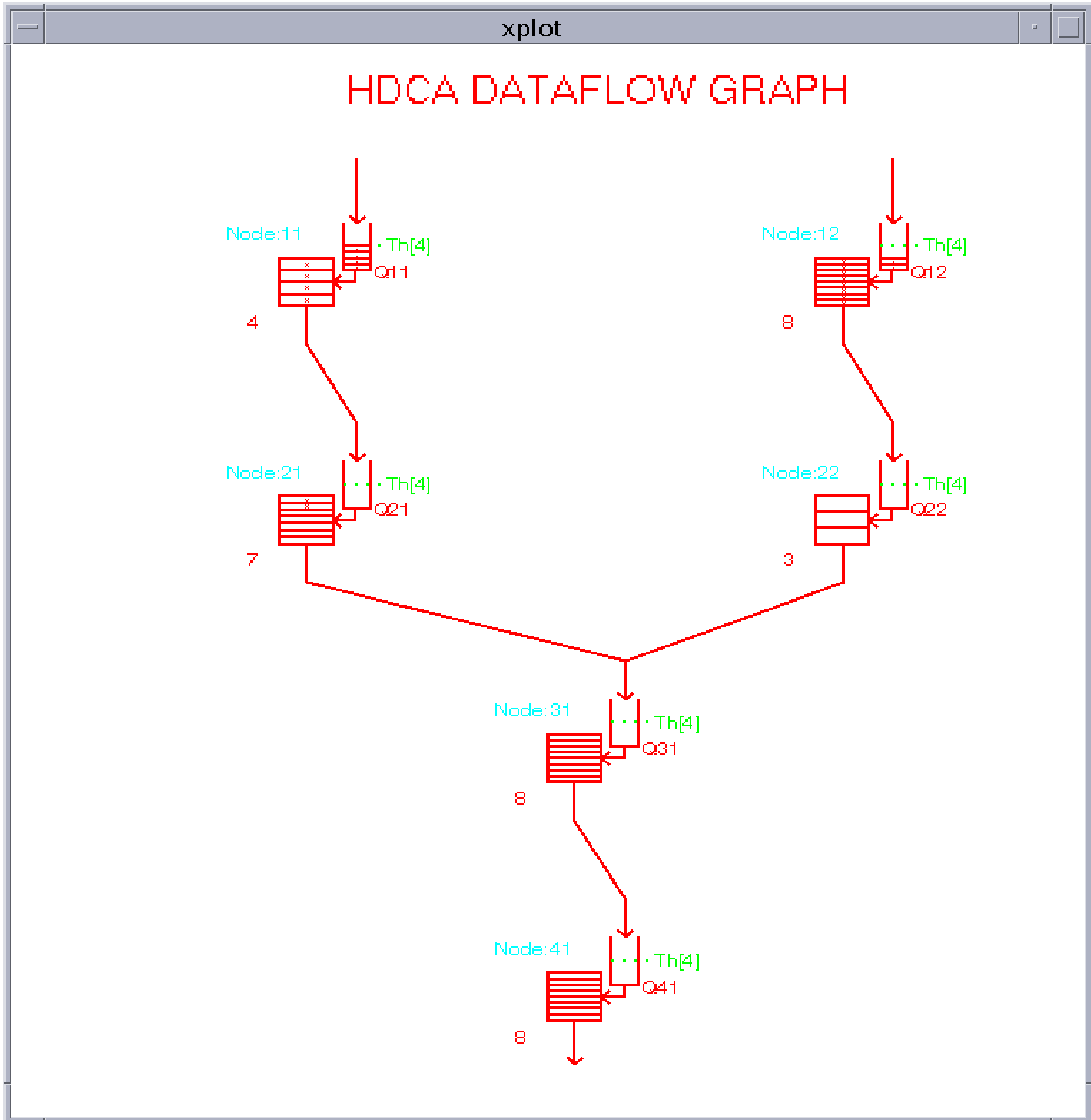


Figure 5.9 a   Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token)
(t =1000 micro-cycles)

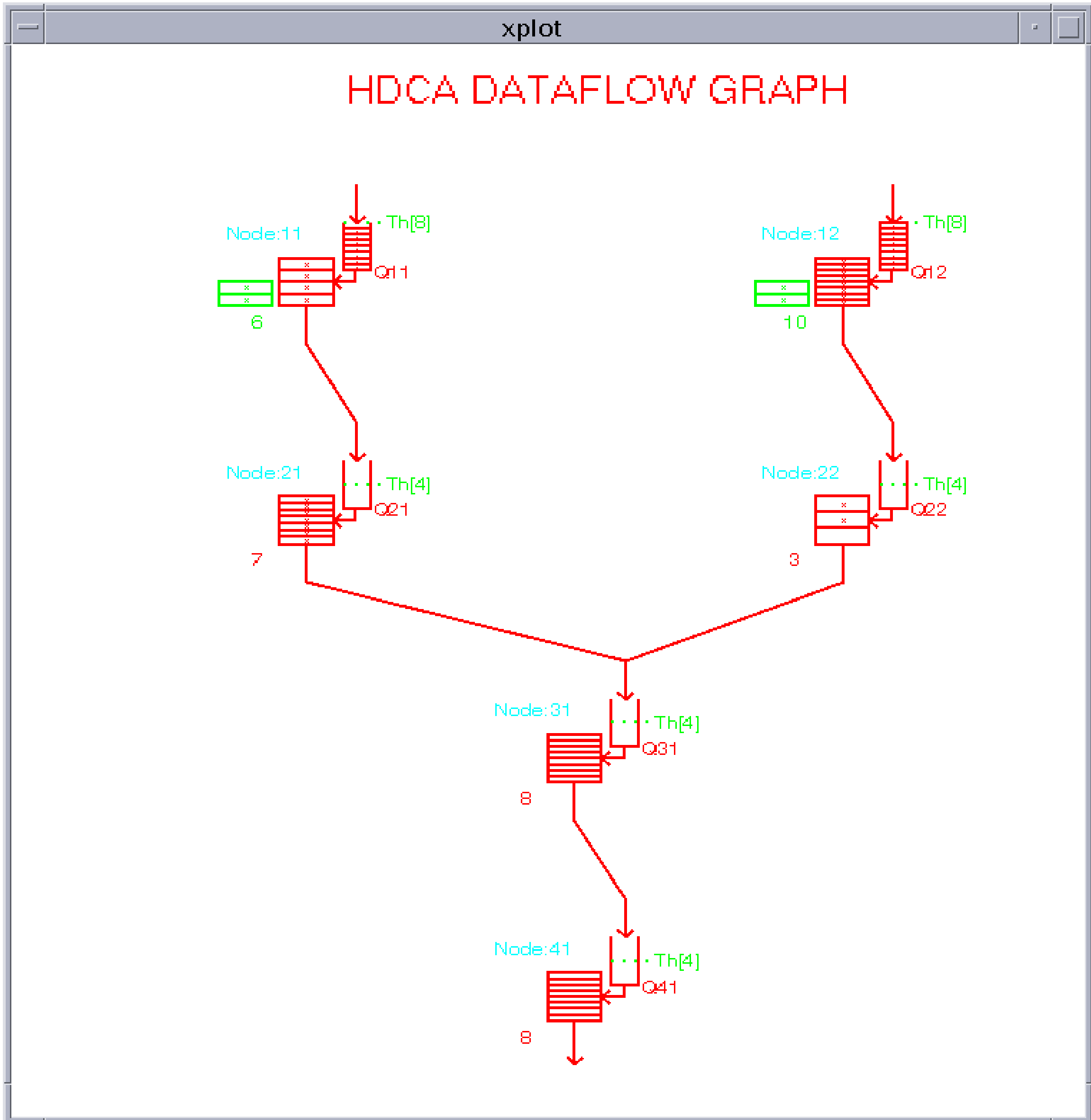


Figure 5.9 b Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token)

(t =2000 micro-cycles)

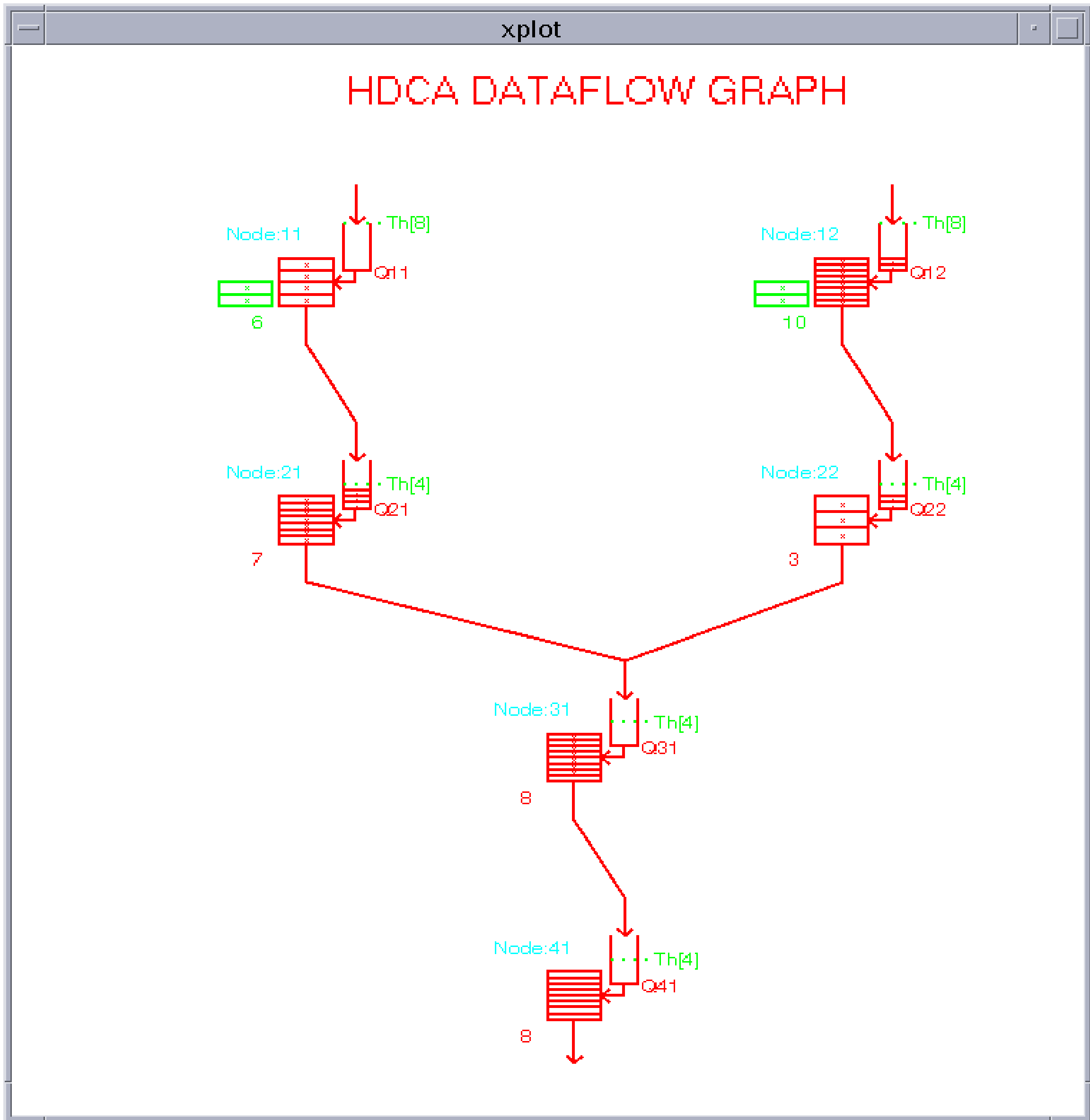


Figure 5.9 c Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token)

(t =3000 micro-cycles)

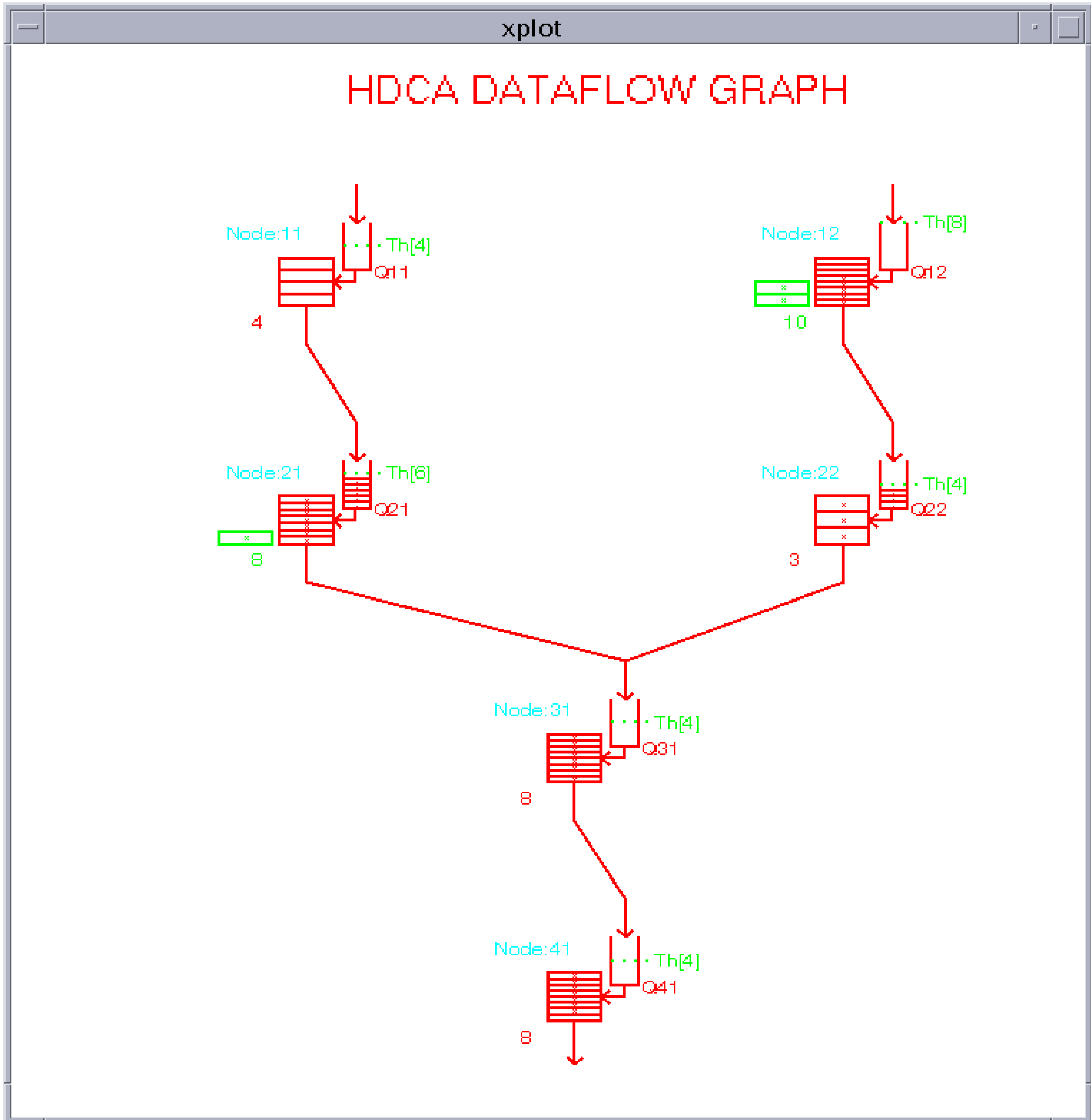


Figure 5.9 d Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token) (t =4000 micro-cycles)

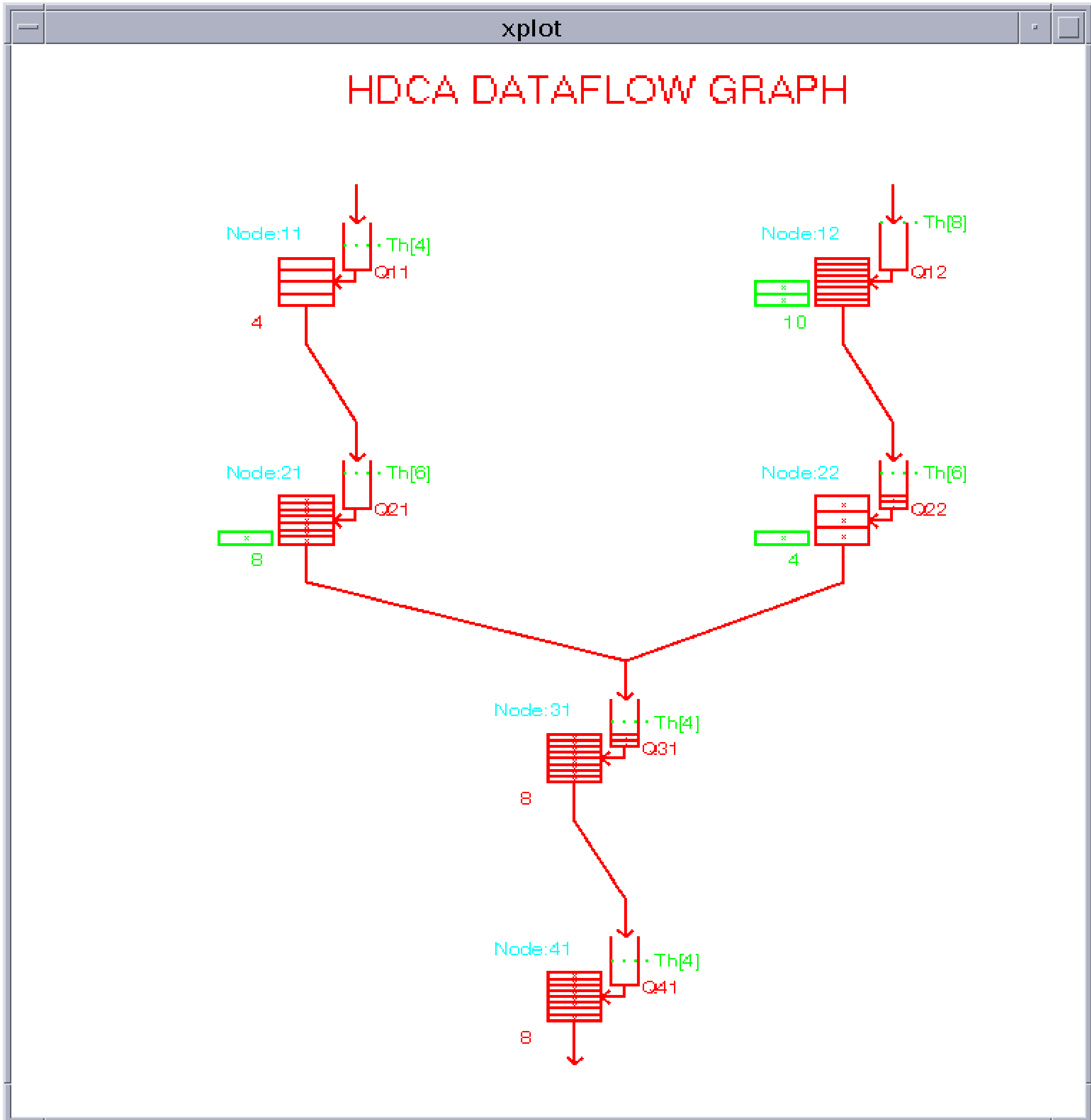


Figure 5.9 e Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token)
(t =5000 micro-cycles)

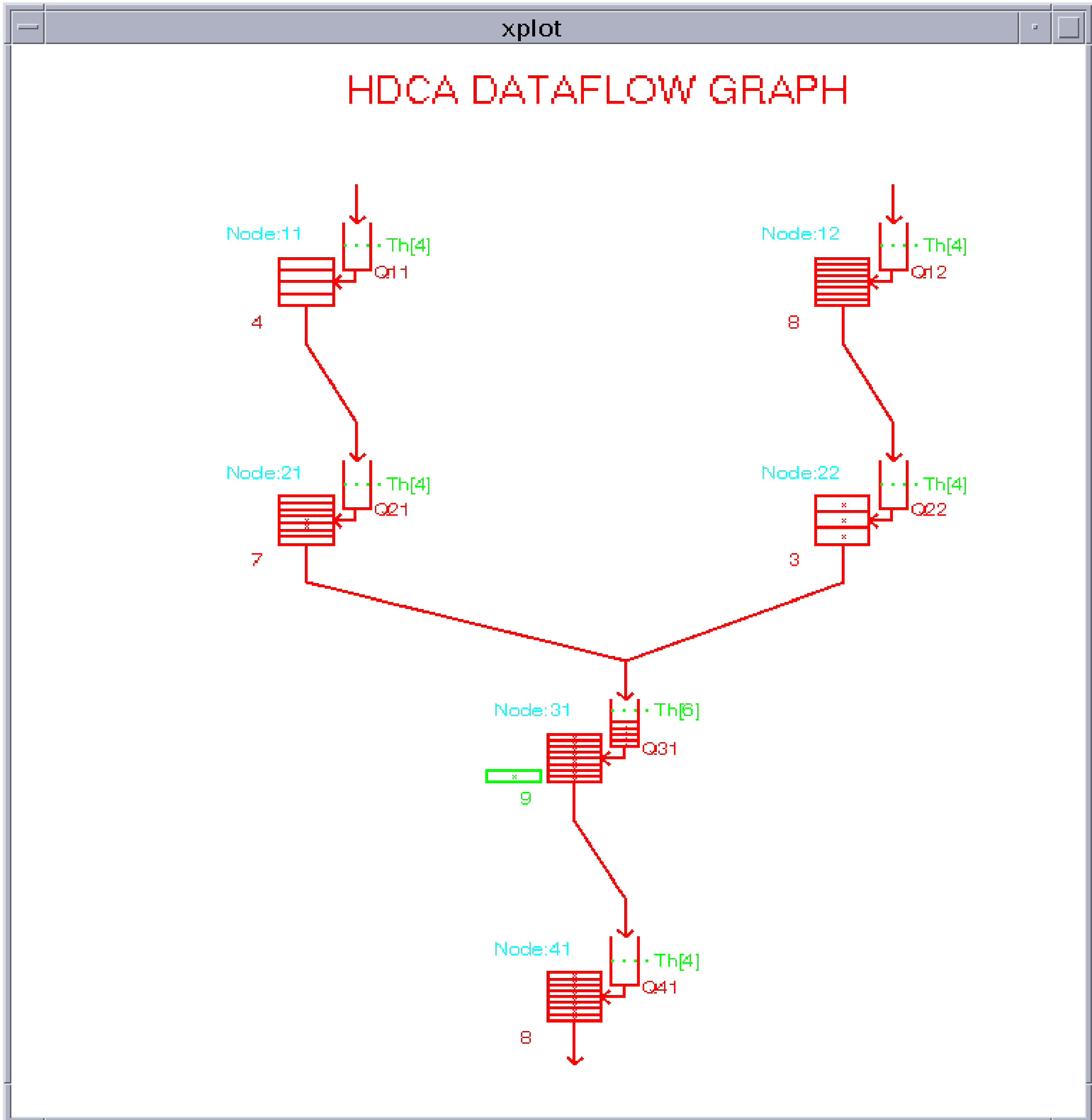


Figure 5.9 f Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token) (t =6000 micro-cycles)

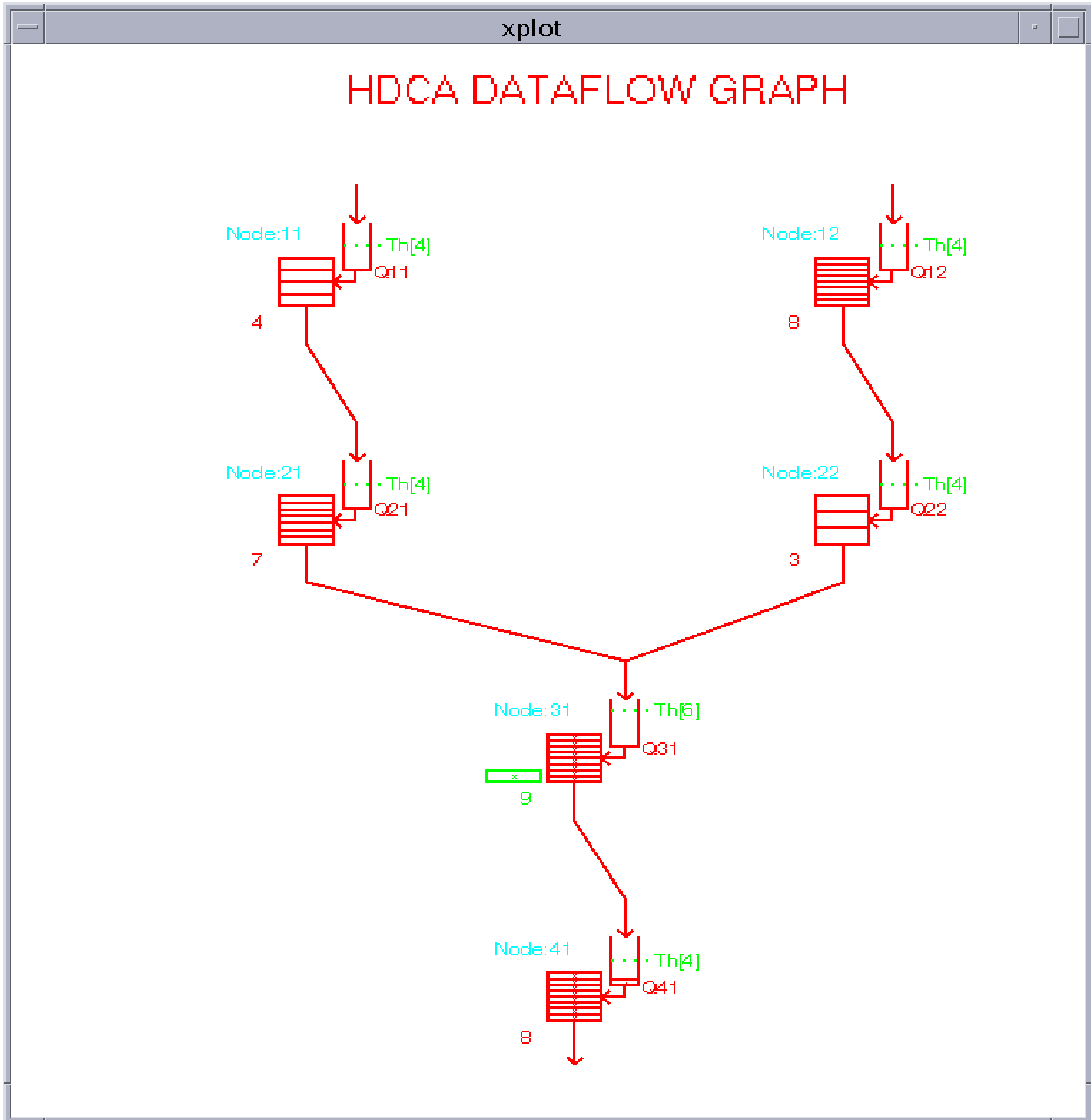


Figure 5.9 g   Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token)

(t =7000 micro-cycles)

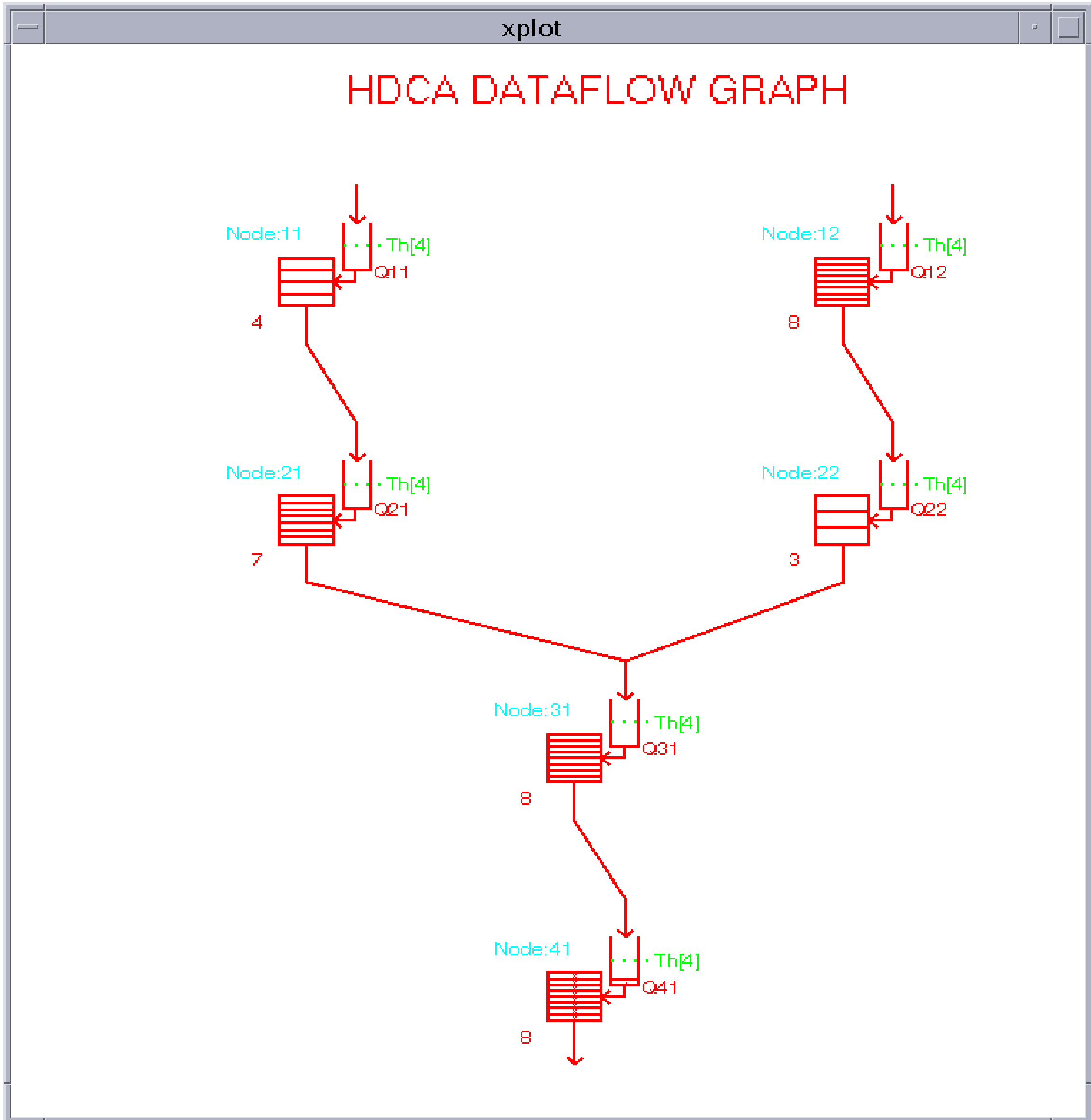


Figure 5.9 h Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token) (t =8000 micro-cycles)

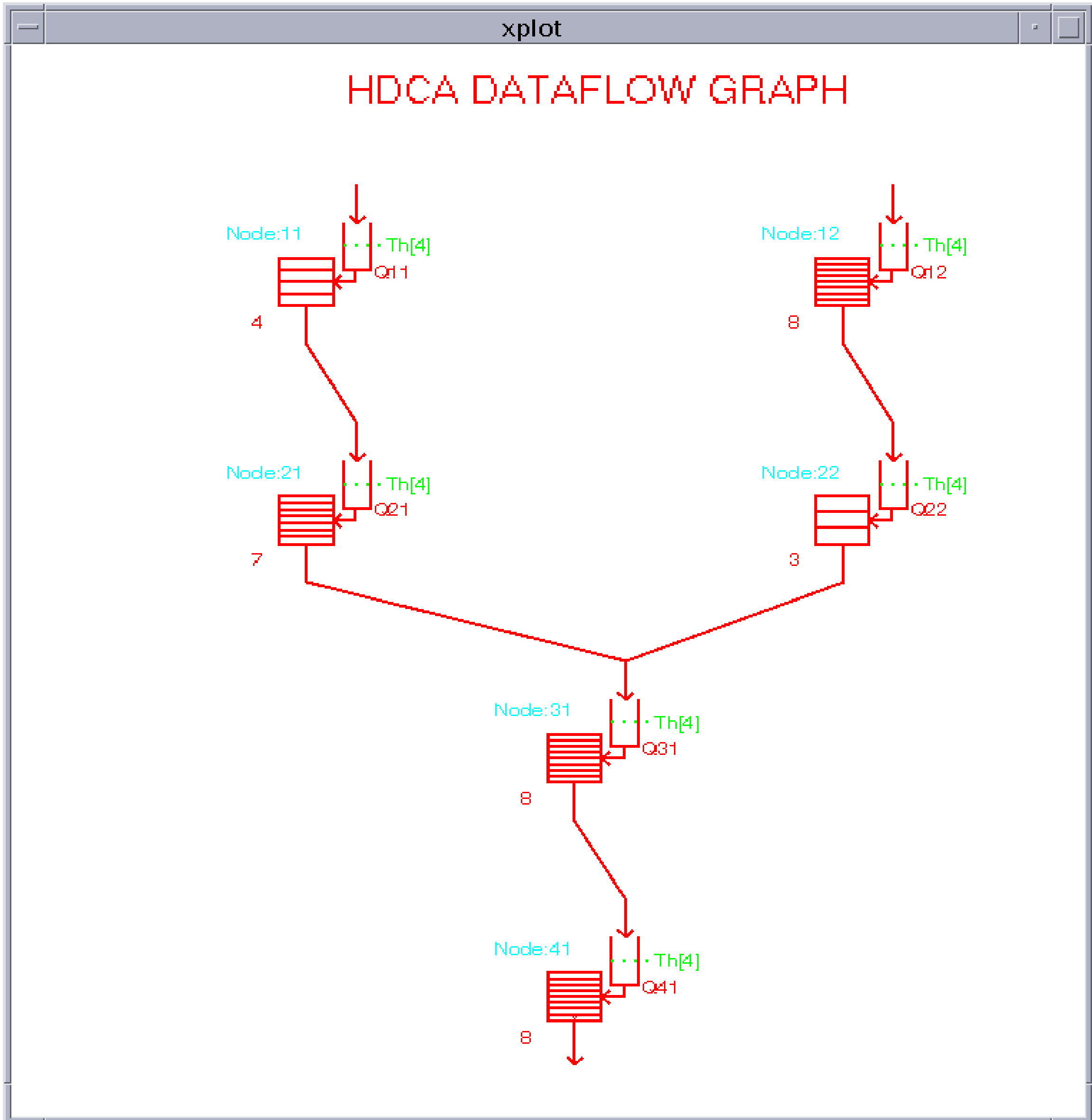


Figure 5.9 i Simu. Results of Application 2 (Iinput rate = 100 micro-cycles/token) (t =9000 micro-cycles)

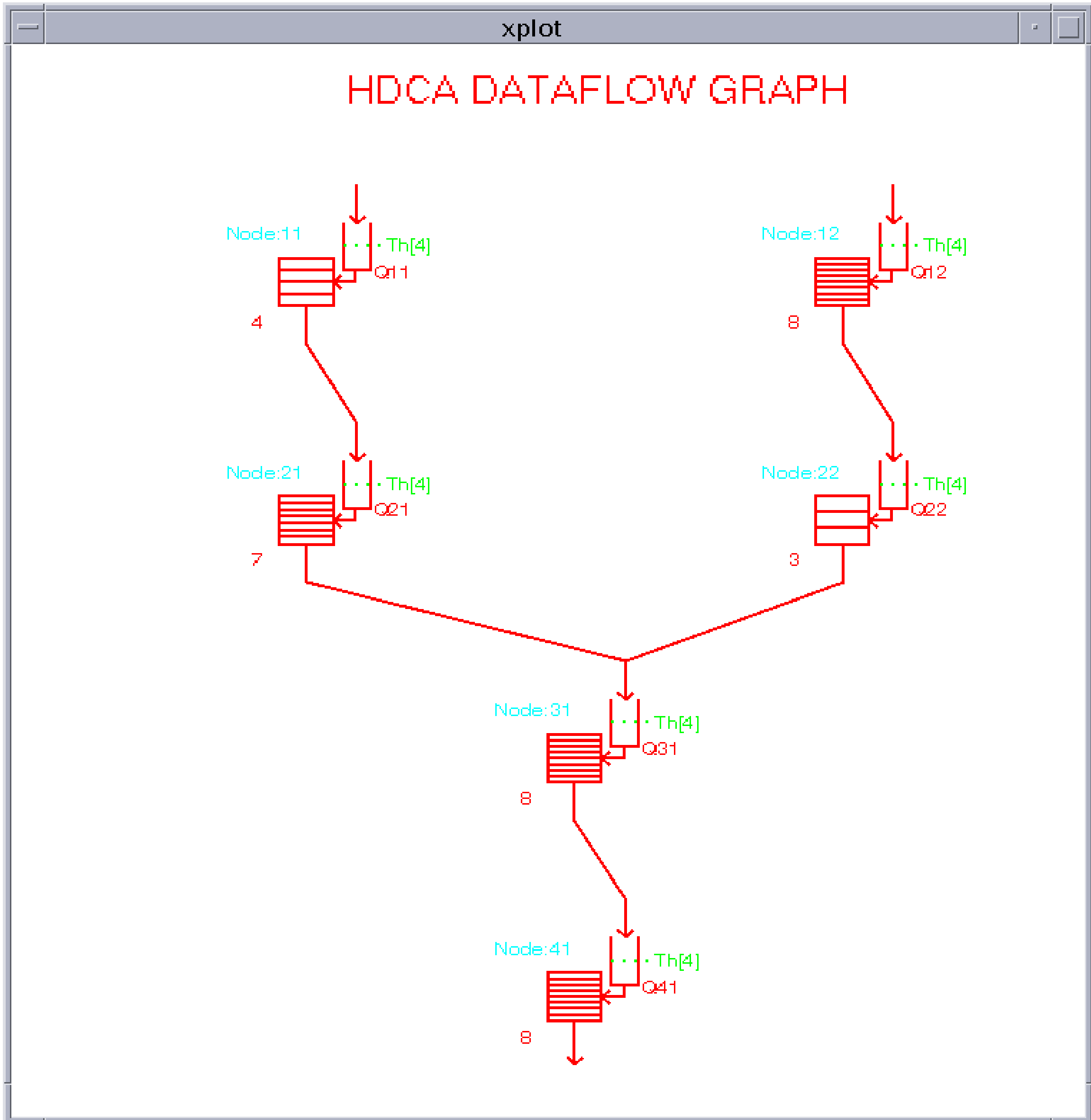


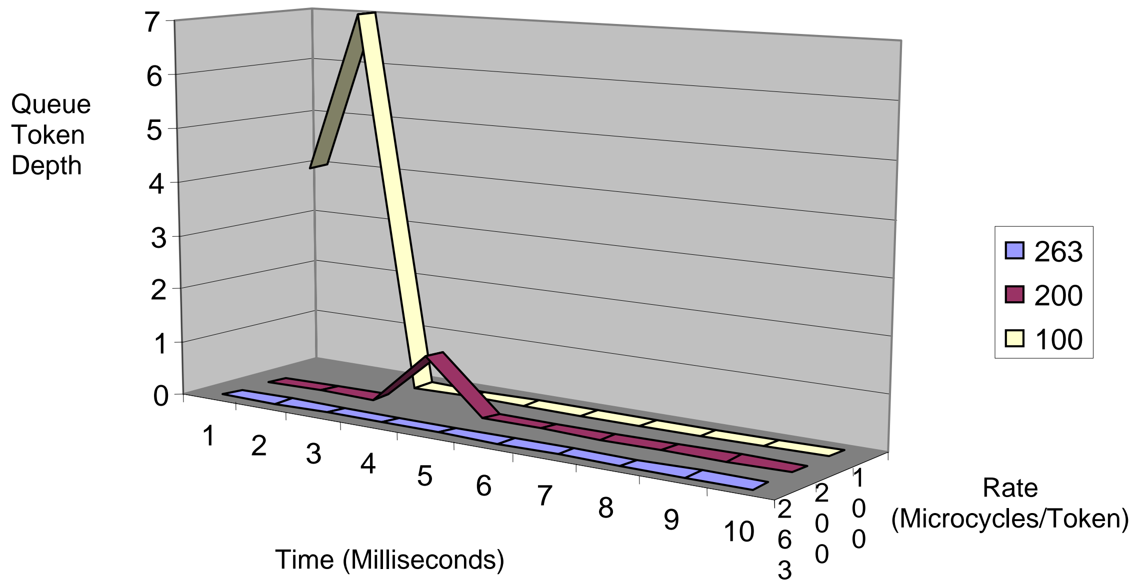
Figure 5.9 j Simulation Result for Application 2 for input rate = 100 micro-cycles/token (t =10000 micro-cycles)

Total Simulation Time = 9136 Micro-cycles

Table 5-5: Application2 Results: 20 Tokens at Node11, 20 Tokens at Node 12

Case2: 20 Tokens at Node11, 20 Tokens at Node12

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Input rate = 263 Microseconds Per DISV										
Node11(4)	0/3/0	0/3/0	0/3/0	0/4/0	0/4/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node12(8)	0/4/0	0/7/0	0/7/0	0/7/0	0/8/0	0/4/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(7)	0/1/0	0/5/0	0/7/0	0/6/0	0/6/0	0/6/0	0/2/0	0/0/0	0/0/0	0/0/0
Node22(3)	0/0/0	0/1/0	0/3/0	0/3/0	0/2/0	0/2/0	0/3/0	0/0/0	0/0/0	0/0/0
Node31(8)	0/0/0	0/0/0	0/4/0	0/8/0	0/8/0	0/8/0	0/7/0	0/5/0	0/0/0	0/0/0
Node41(8)	0/0/0	0/0/0	0/0/0	0/4/0	0/7/0	0/8/0	0/8/0	0/7/0	0/4/0	0/0/0
Input rate = 200 Microseconds Per DISV										
Node11(4)	0/4/0	0/4/0	0/4/0	1/4/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node12(8)	0/5/0	1/8/0	1/8/0	2/8/0	0/5/0	0/2/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(7)	0/1/0	0/6/0	1/7/0	1/7/0	1/7/0	0/4/0	0/0/0	0/0/0	0/0/0	0/0/0
Node22(3)	0/0/0	0/1/0	0/3/0	0/3/0	1/3/0	0/3/0	0/0/0	0/0/0	0/0/0	0/0/0
Node31(8)	0/0/0	0/0/0	0/6/0	0/8/0	1/8/0	1/8/0	2/8/0	0/2/0	0/0/0	0/0/0
Node41(8)	0/0/0	0/0/0	0/0/0	0/6/0	0/8/0	0/8/0	0/8/0	0/8/0	0/2/0	0/0/0
Input rate = 100 Microseconds Per DISV										
Node11(4)	4/4/0	7/6/2	0/6/2	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node12(8)	2/8/0	8/10/2	2/10/2	0/7/2	0/2/2	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(7)	0/2/0	0/7/0	3/7/0	5/9/1	0/9/1	0/2/0	0/0/0	0/0/0	0/0/0	0/0/0
Node22(3)	0/0/0	0/2/0	2/3/0	3/3/0	2/4/1	0/3/0	0/0/0	0/0/0	0/0/0	0/0/0
Node31(8)	0/0/0	0/0/0	0/7/0	0/7/0	2/8/0	4/9/1	0/9/1	0/0/0	0/0/0	0/0/0
Node41(8)	0/0/0	0/0/0	0/0/0	0/7/0	0/7/0	0/8/0	1/8/0	1/8/0	0/1/0	0/0/0



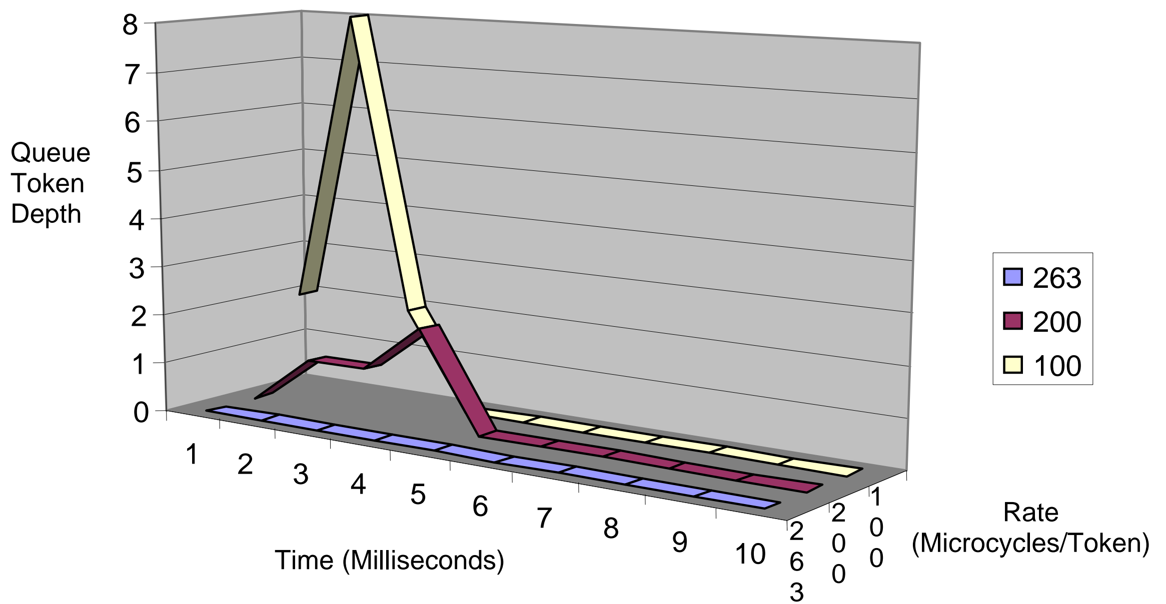
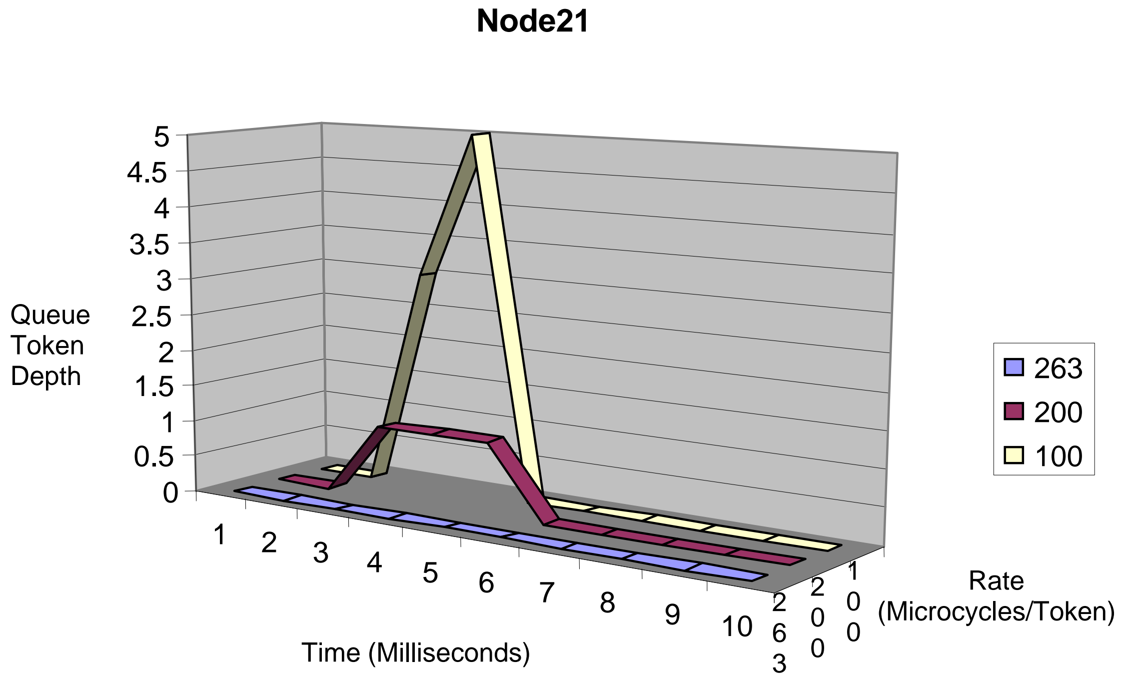


Figure 5.10 Queue Depth Plot for Application 2



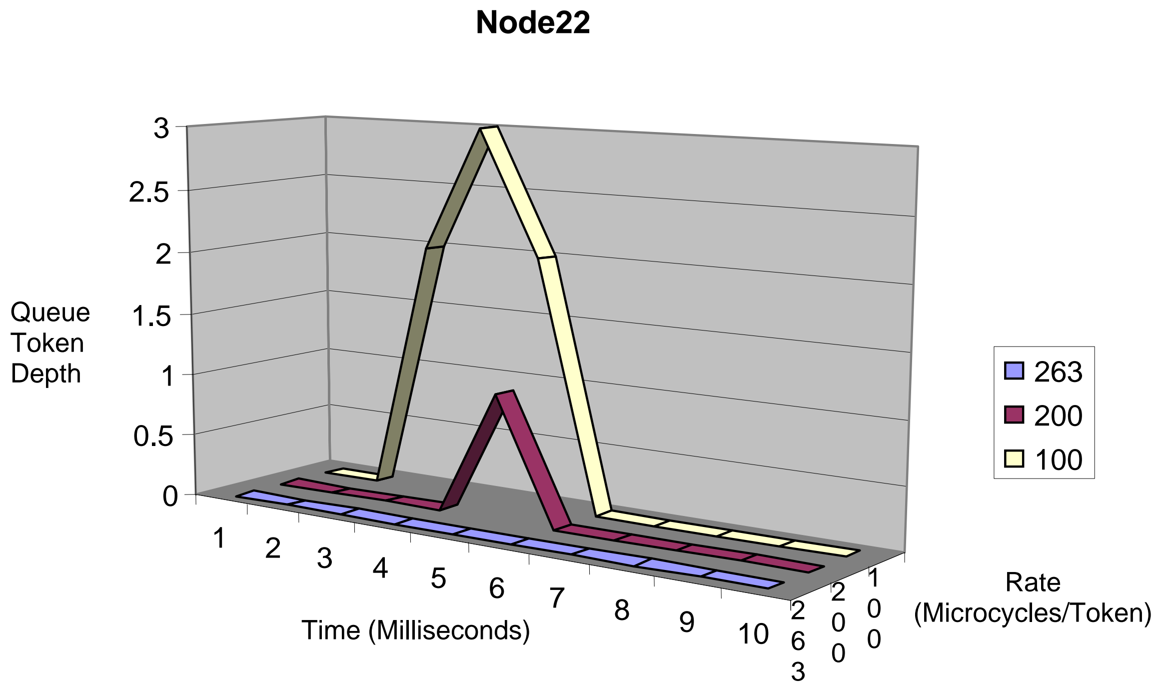
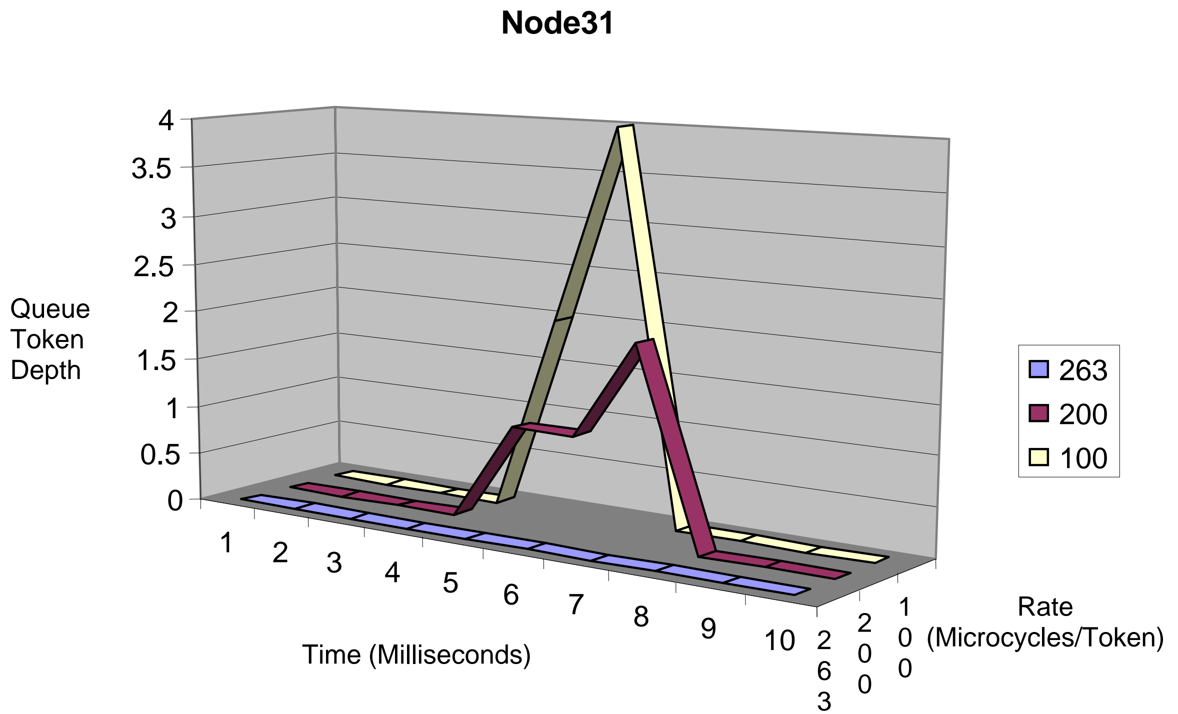


Figure 5.10 Queue Depth Plot for Application 2 (Continued 2)



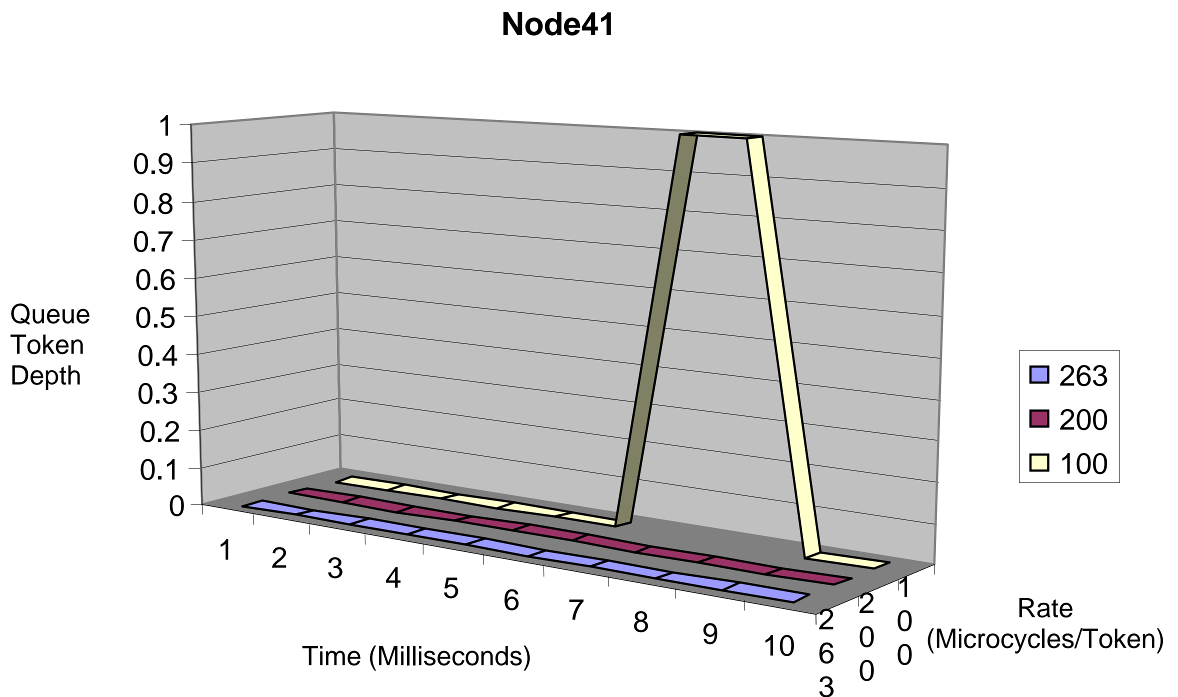


Figure 5.10 Queue Depth Plot for Application 2 (Continued 3)

5.3 Application 3

The dataflow graph for the 3rd application is shown in Figure 5.11. This graph has two feedback paths. One feedback path is from node 41 back to node 21 through node31; the other feedback path is from node 41 back to node 22 through node 33. There are two forks in this graph, node 11 and node 41. Node 11 has two branches, while node 41 has three branches. The α parameters and input files are shown in Table 5-6. Graphical simulation results are shown in Figures 5.12, 5.13 and 5.14. Table 5-7 shows the queue depth, number of free copies, and the number of extra copies of each node at different time based on the Figures 5.12, 5.13, and 5.14.

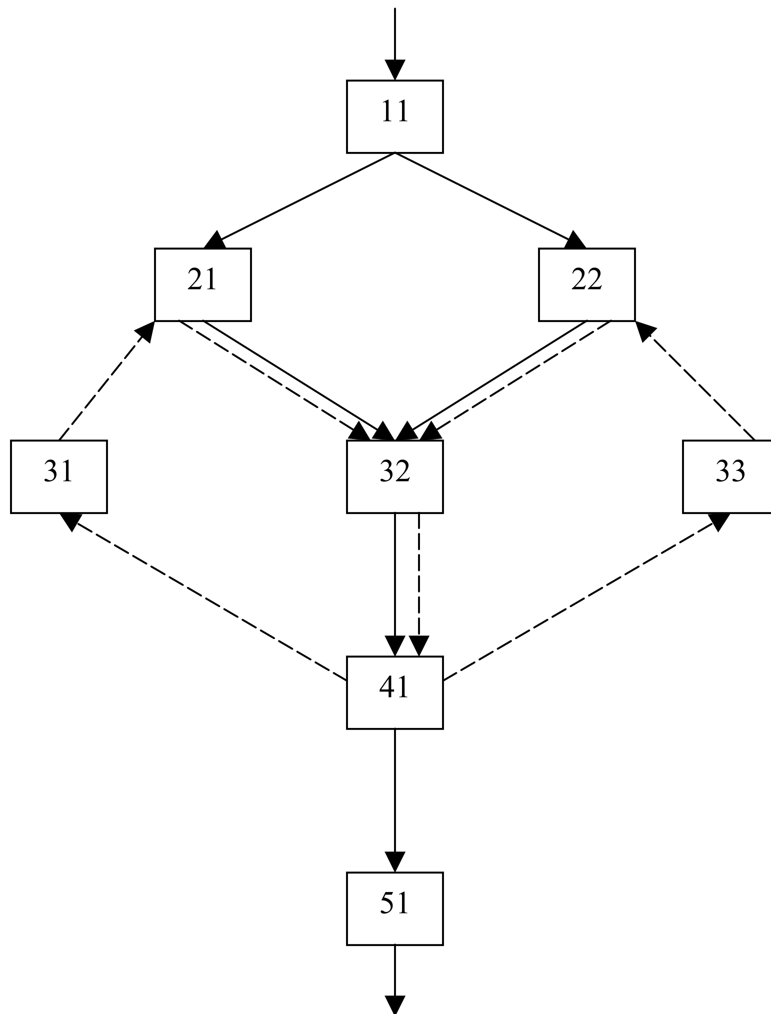


Figure 5.11 Dataflow Graph of Application 3

Table 5-6: **Parameter Values for Data Flow Graph of Application 3**

Process Designation	Execution Time (milliseconds)	Process Length (Kilobytes)
11	0.85	0.425
21	0.65	2.5
22	0.68	0.9
31	0.2	2.0
32	0.4	0.5
33	0.3	1.3
41	0.69	0.85
51	0.75	1.0

Input Data Rates (data items/millisecond)

Peak Load: 3.8

Average Load: 2.5

Probability Distributions for Forks

$P_{11->21}$: 0.6

$P_{11->22}$: 0.4

$P_{41->51}$: 0.8

$P_{41->31}$: 0.1

$P_{41->33}$: 0.1

Program Memory/Computing Element: 16 kilobytes

Copyset: 4 5 1 1 2 1 3 2 4 1 5 1 9 1 1 2 1 3 2 4 1 3 1 2 1 3 2 4 1 5 1 5 1 1 2 2 3 2 4 1 5 1 9 1 1 2 2 3 2 4 1 3 3 2 2 3 2 4 1 5 1 0.85 0.425 0.65 2.5 0.4 0.5 0.69 0.85 0.75 1.0 0.2 2.0 0.68 0.9 0.3 1.3 16 3.8 2.5 1 0.6 0.6 0.4 0.4 0.8 0.1 0.8 0.8 0.1 0.8 1

Dataset: 5 1 2 3 1 1 1 1 2 1 1 1 2 2 2 1 3 2 2 2 3 2 3 2 4 1 3 1 2 1 3 3 2 2 4 1 3 1 4 1 3 3 4 1 5 1 0 0 0 0

Informationset: 1 0 0 0 0 0 1 2

Probabilityset: 0.6 0.4 0.1 0.1 0.8

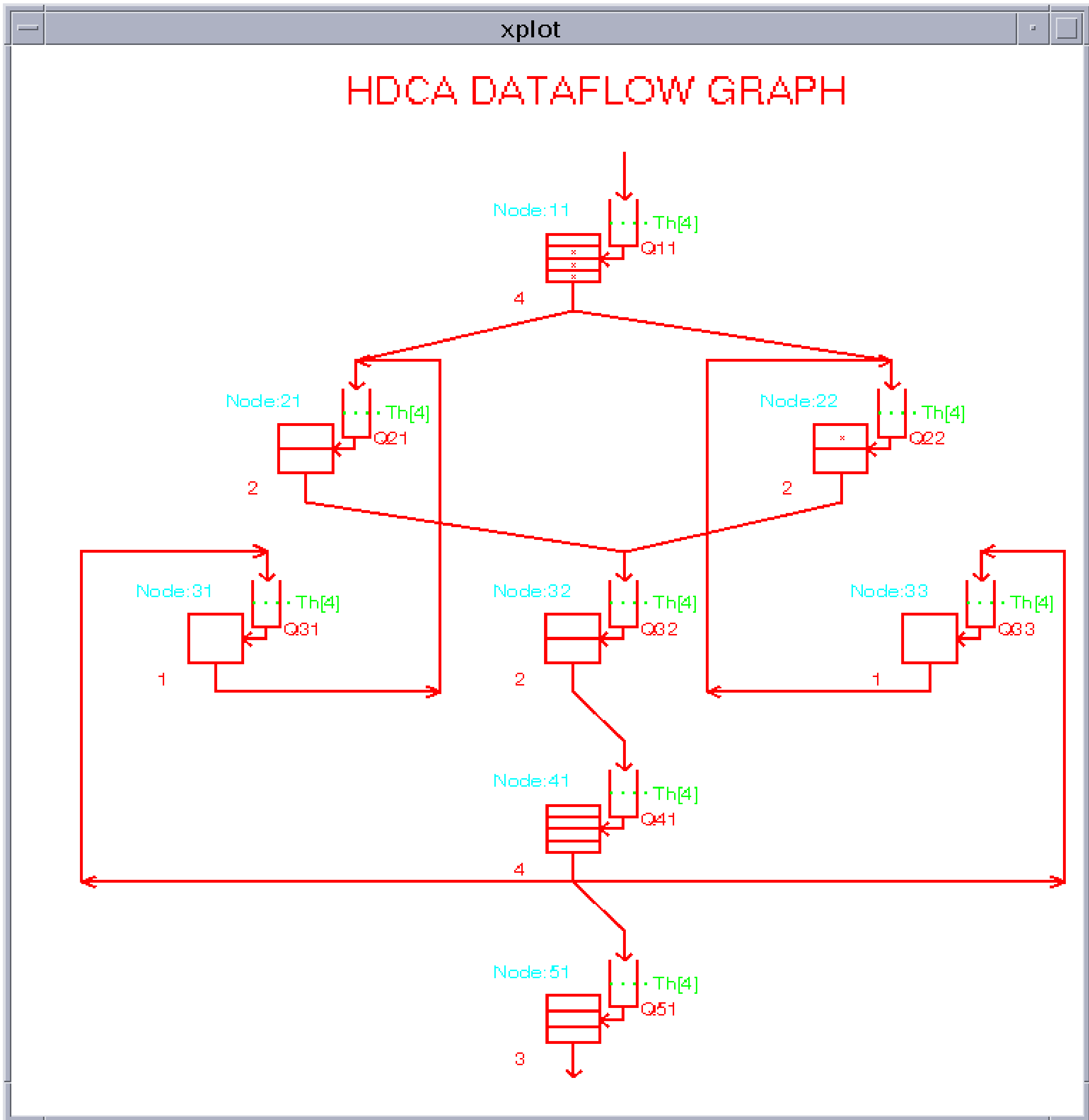


Figure 5.12 Simu. Results of Application 3 (Input rate = 263 micro-cycles/token) (t =1000 micro-cycles)

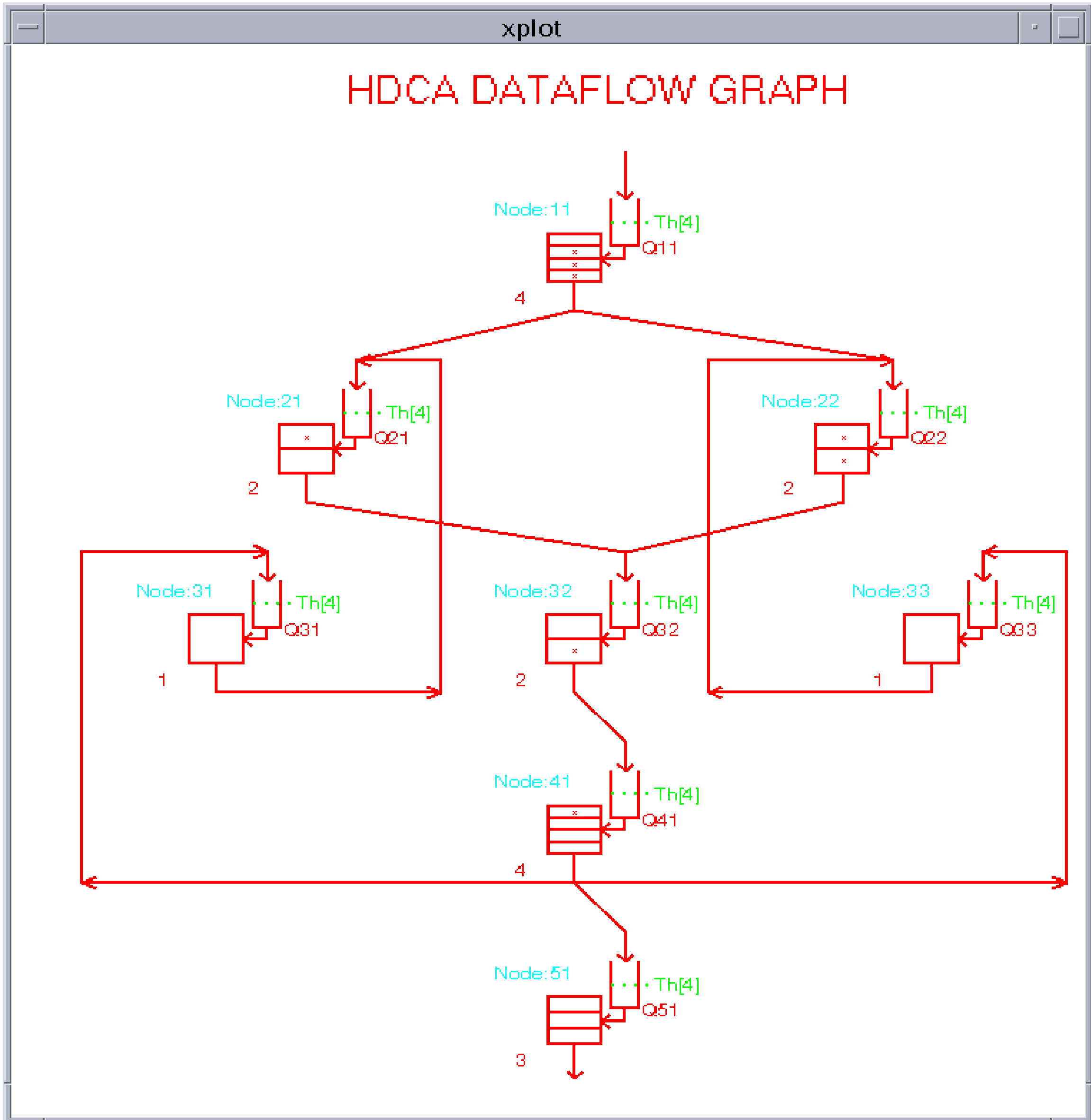


Figure 5.12 b Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =2000 micro-cycles)

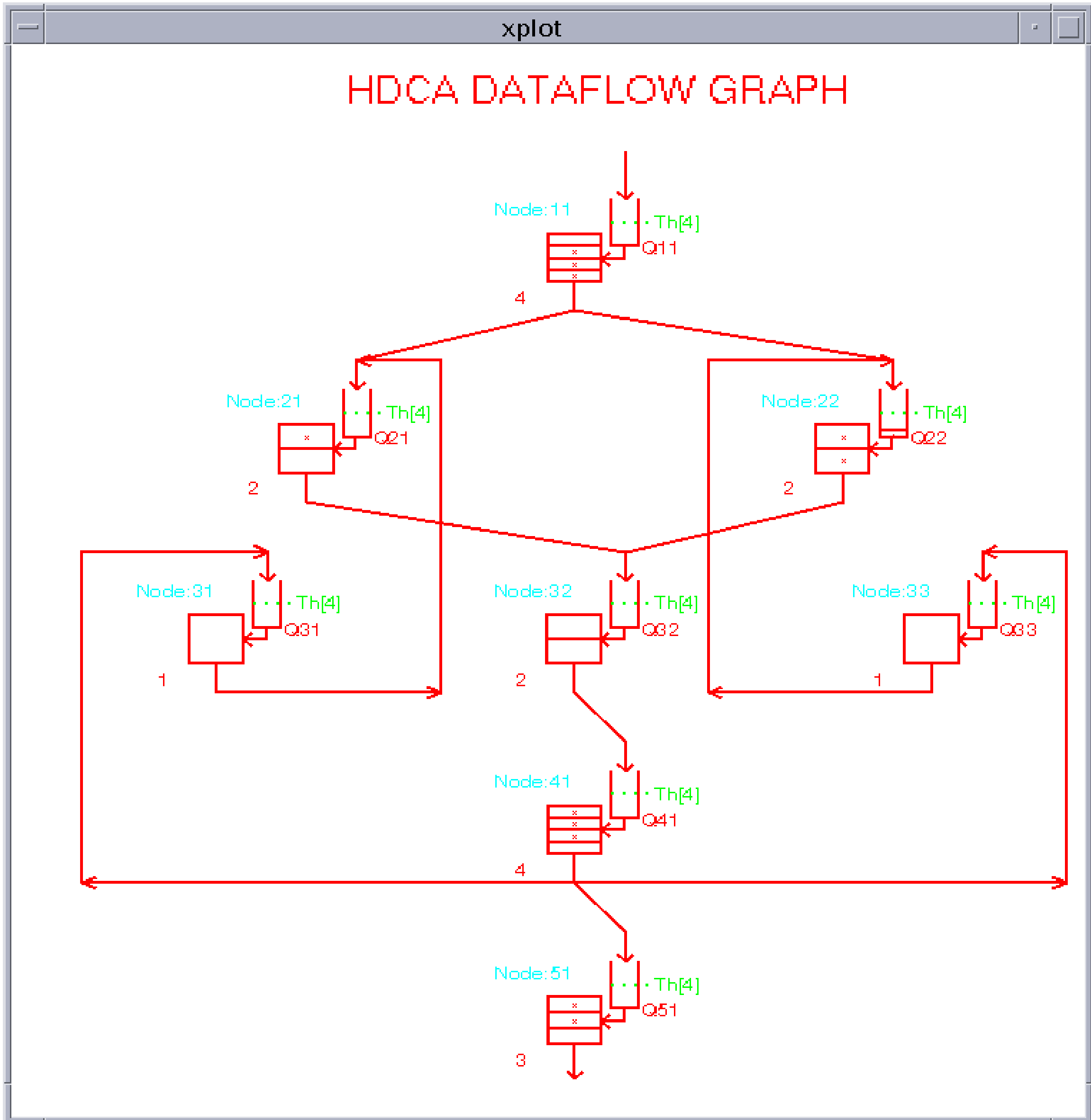


Figure 5.12 c Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =3000 micro-cycles)

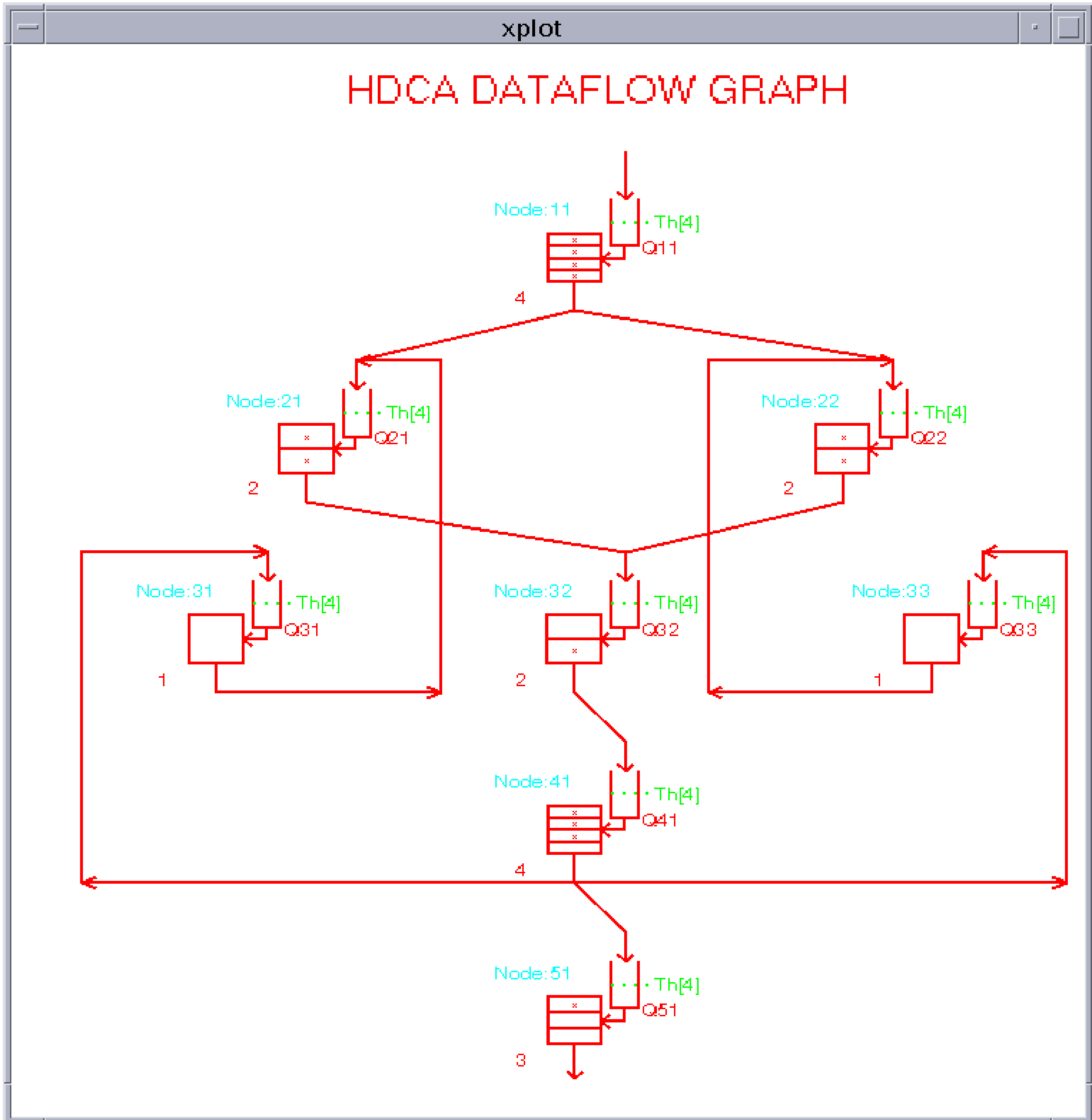


Figure 5.12 d Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =4000 micro-cycles)

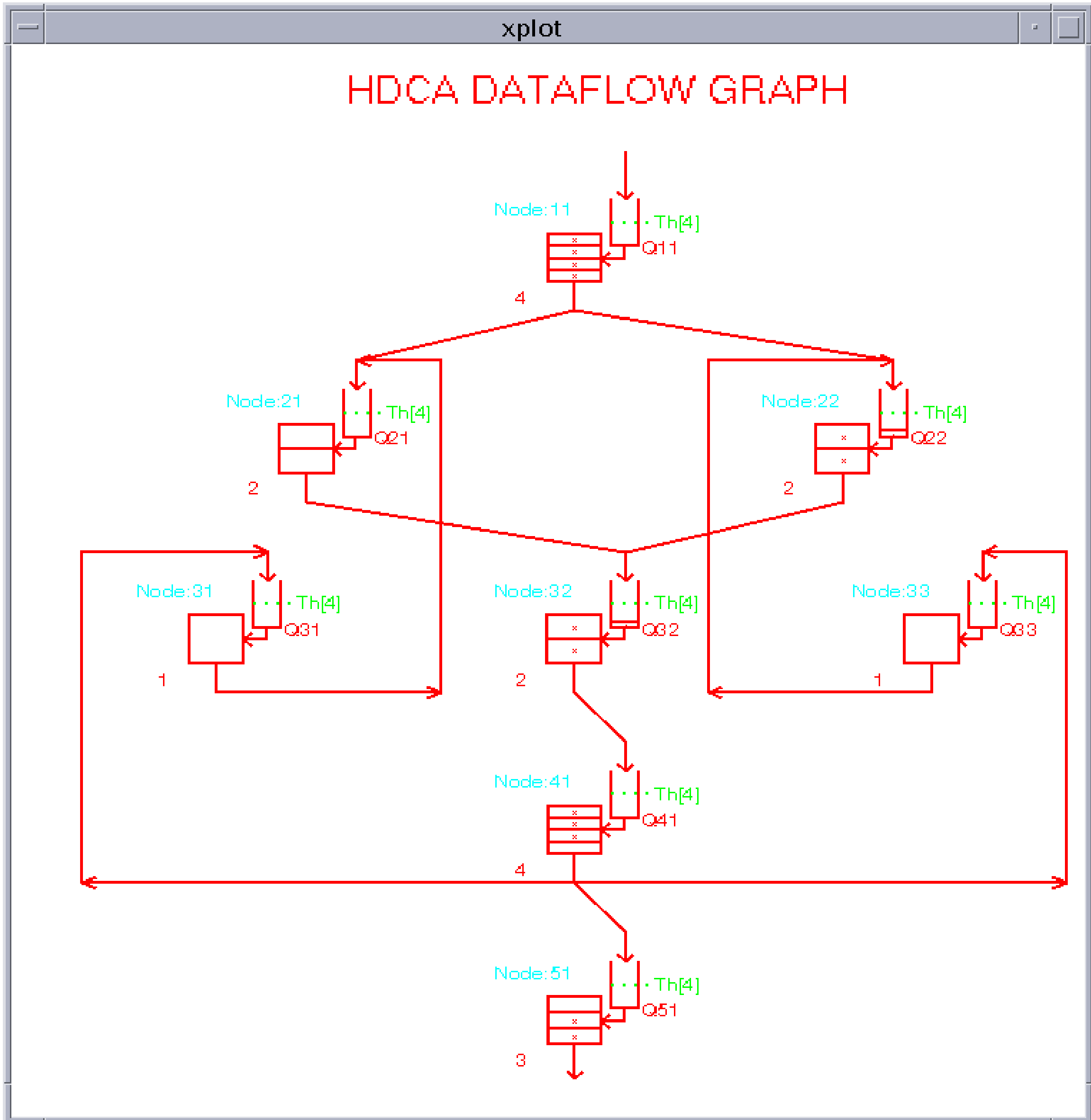


Figure 5.12 e Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =5000 micro-cycles)

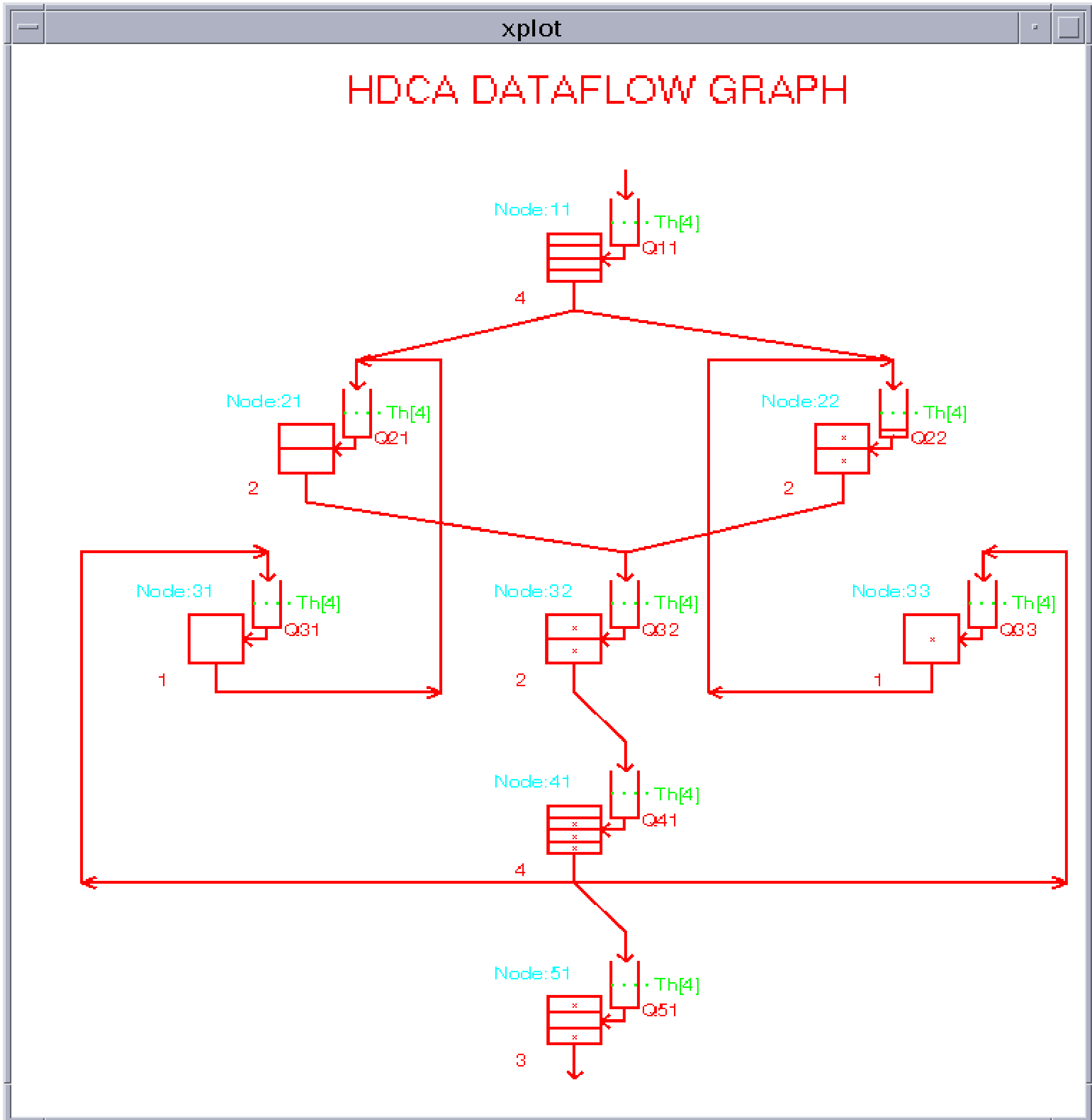


Figure 5.12 f Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =6000 micro-cycles)

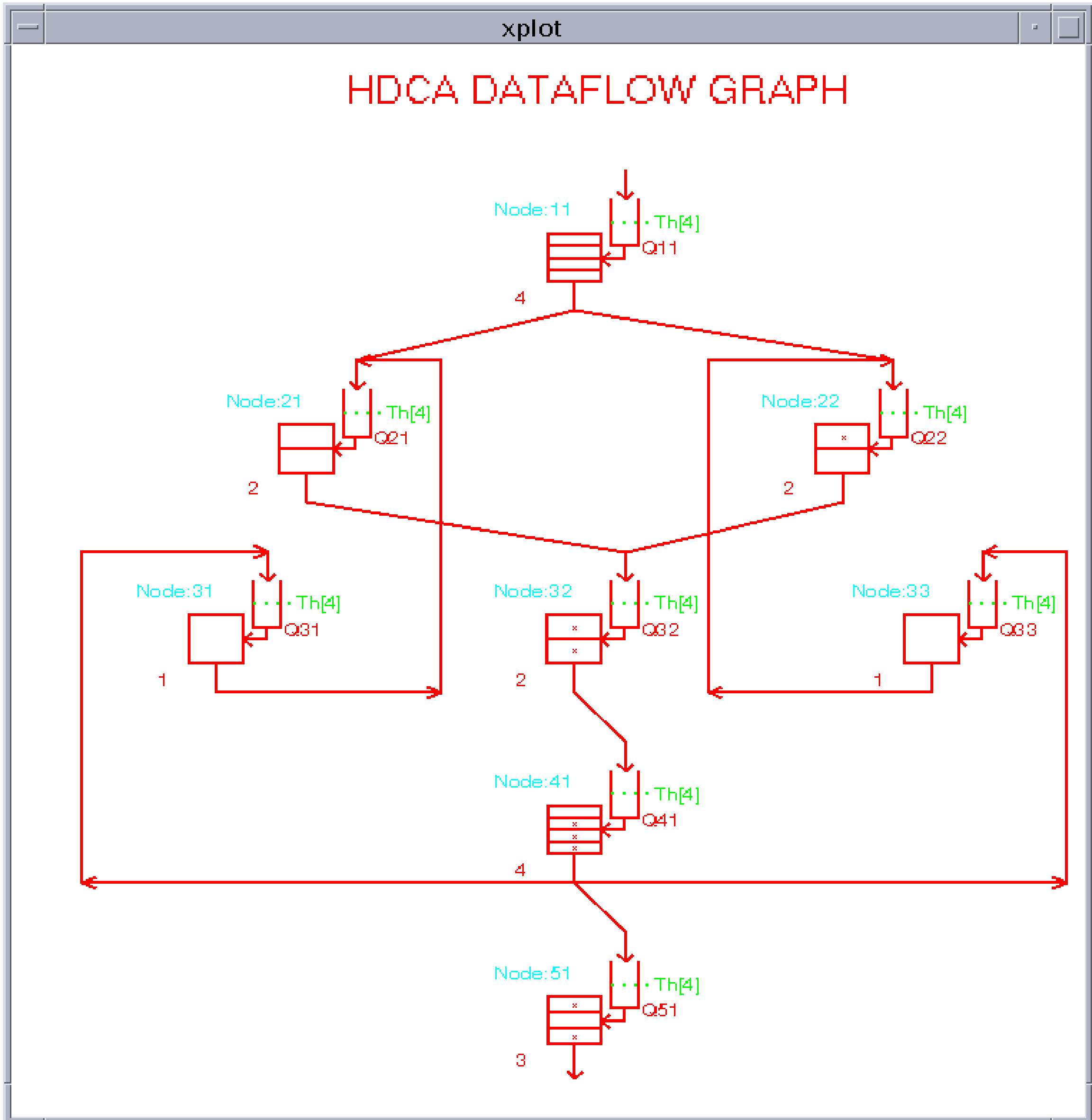


Figure 5.12 g Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =7000 micro-cycles)

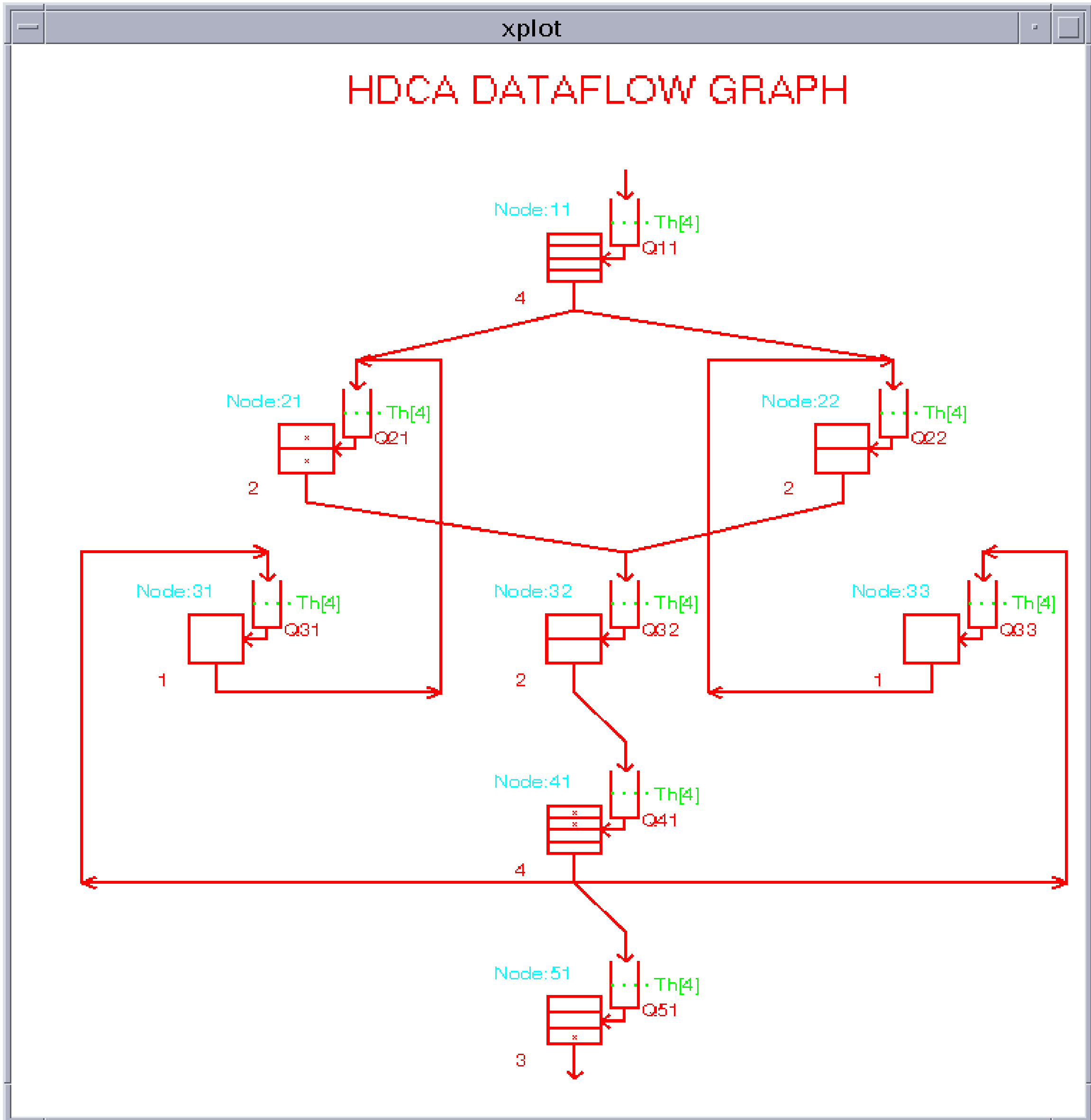


Figure 5.12 h   Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =8000 micro-cycles)

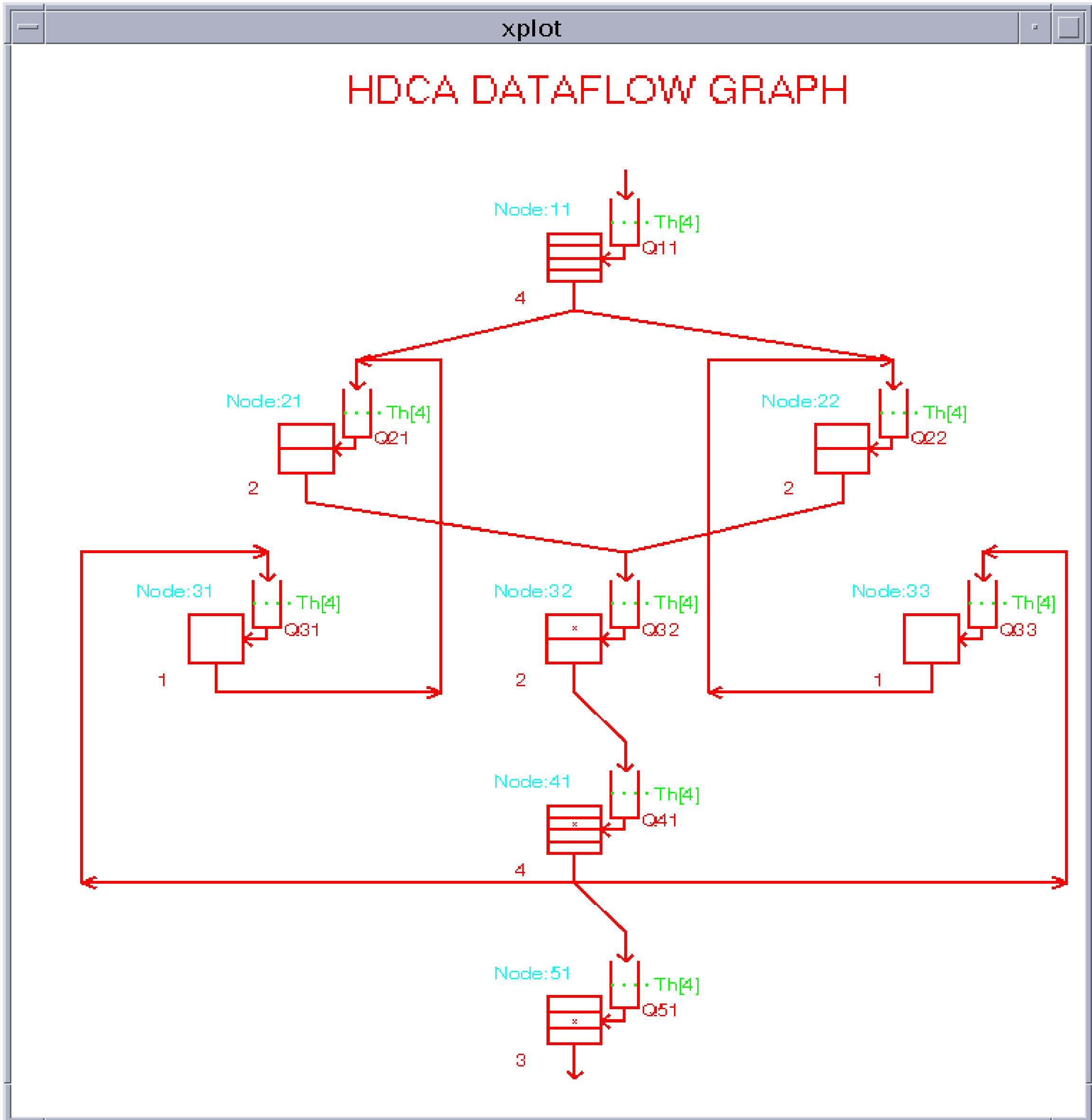


Figure 5.12 i Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =9000 micro-cycles)

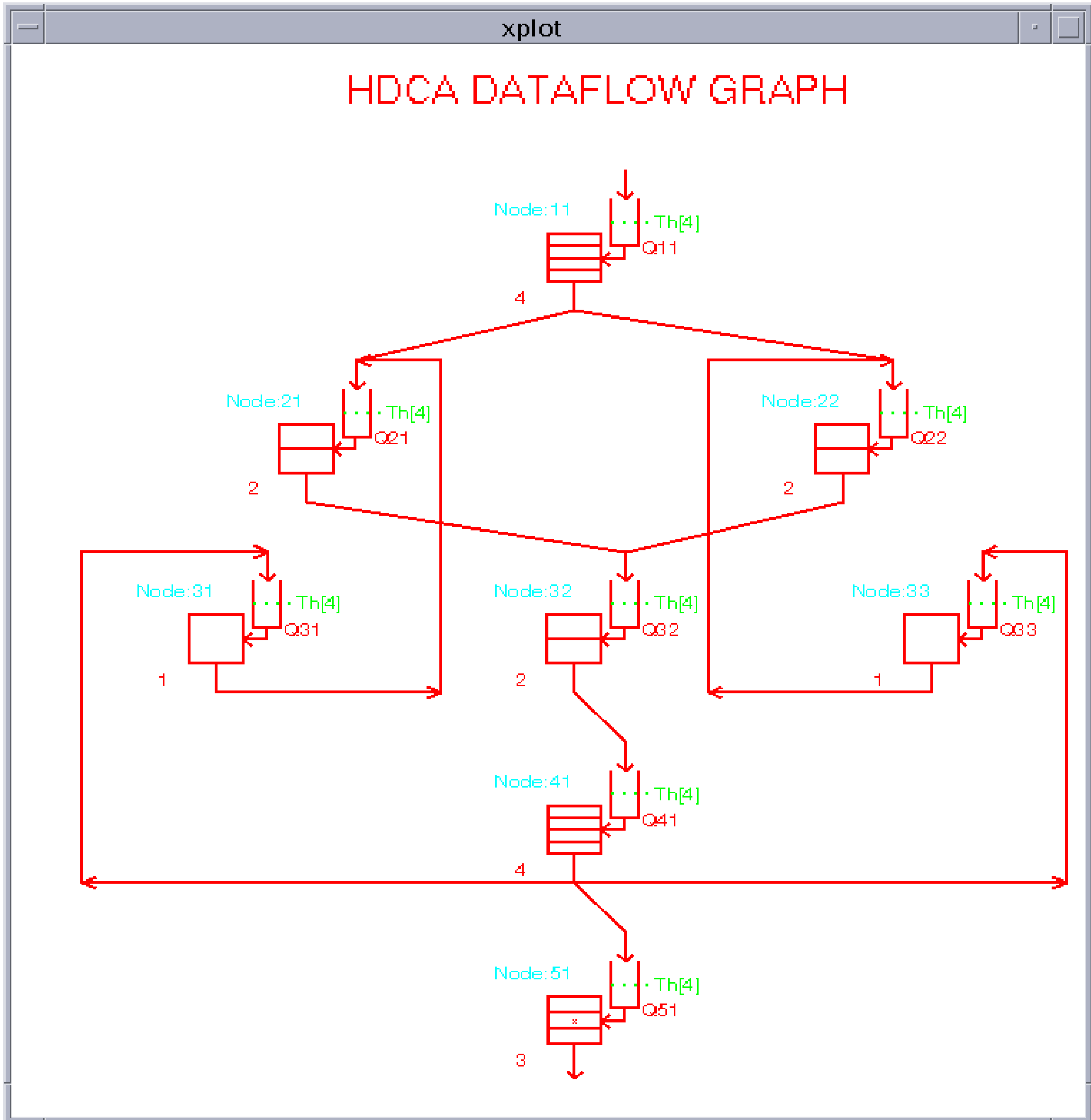


Figure 5.12 j Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =10000 micro-cycles)

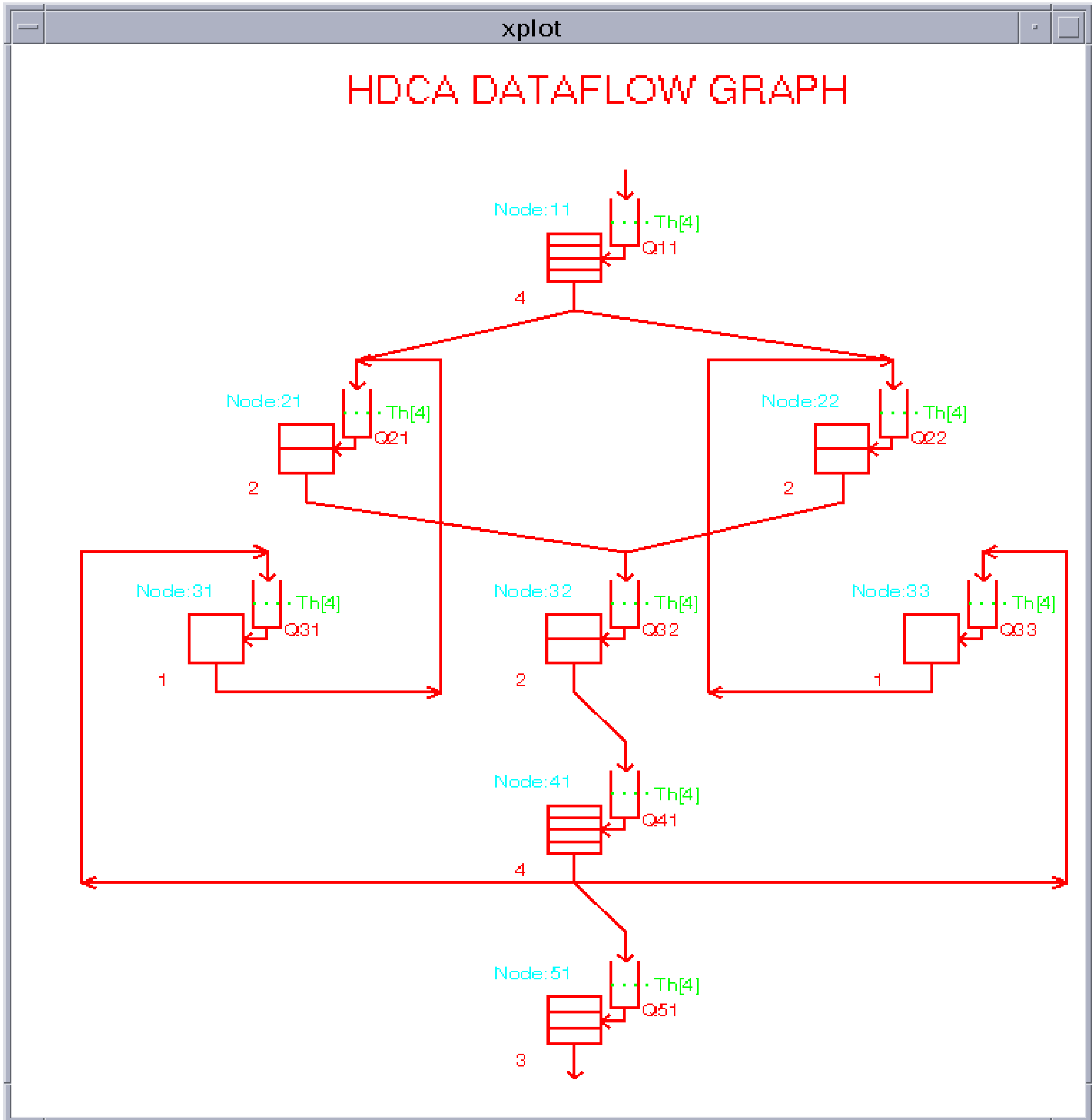


Figure 5.12 k Simulation Result for Application 3 for input rate = 263 micro-cycles/token (t =11000 micro-cycles)

Total Simulation Time = 10488 Micro-cycles

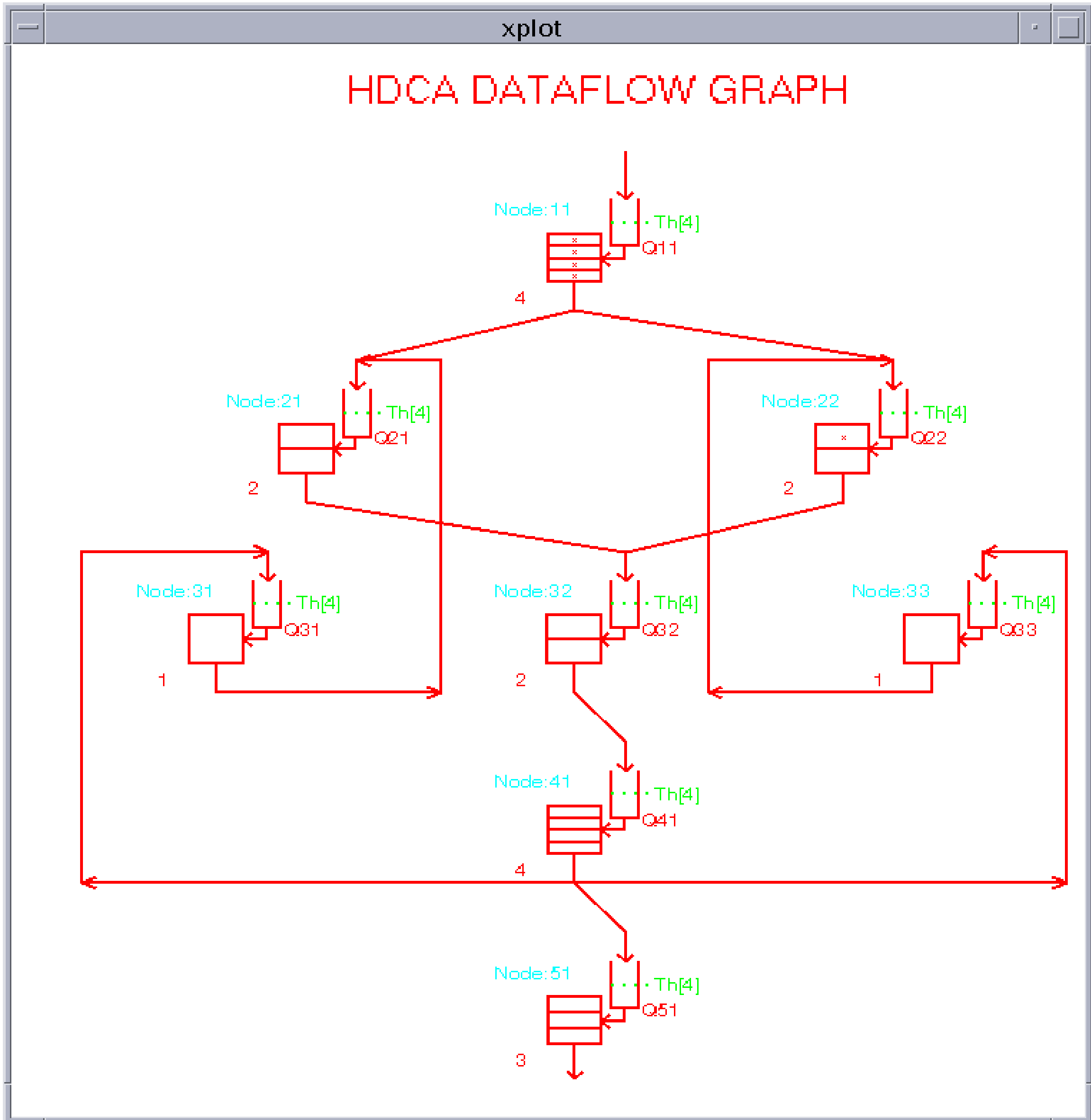


Figure 5.13 Simu. Results of Application 3 (Input rate = 200 micro-cycles/token (t =1000 micro-cycles)

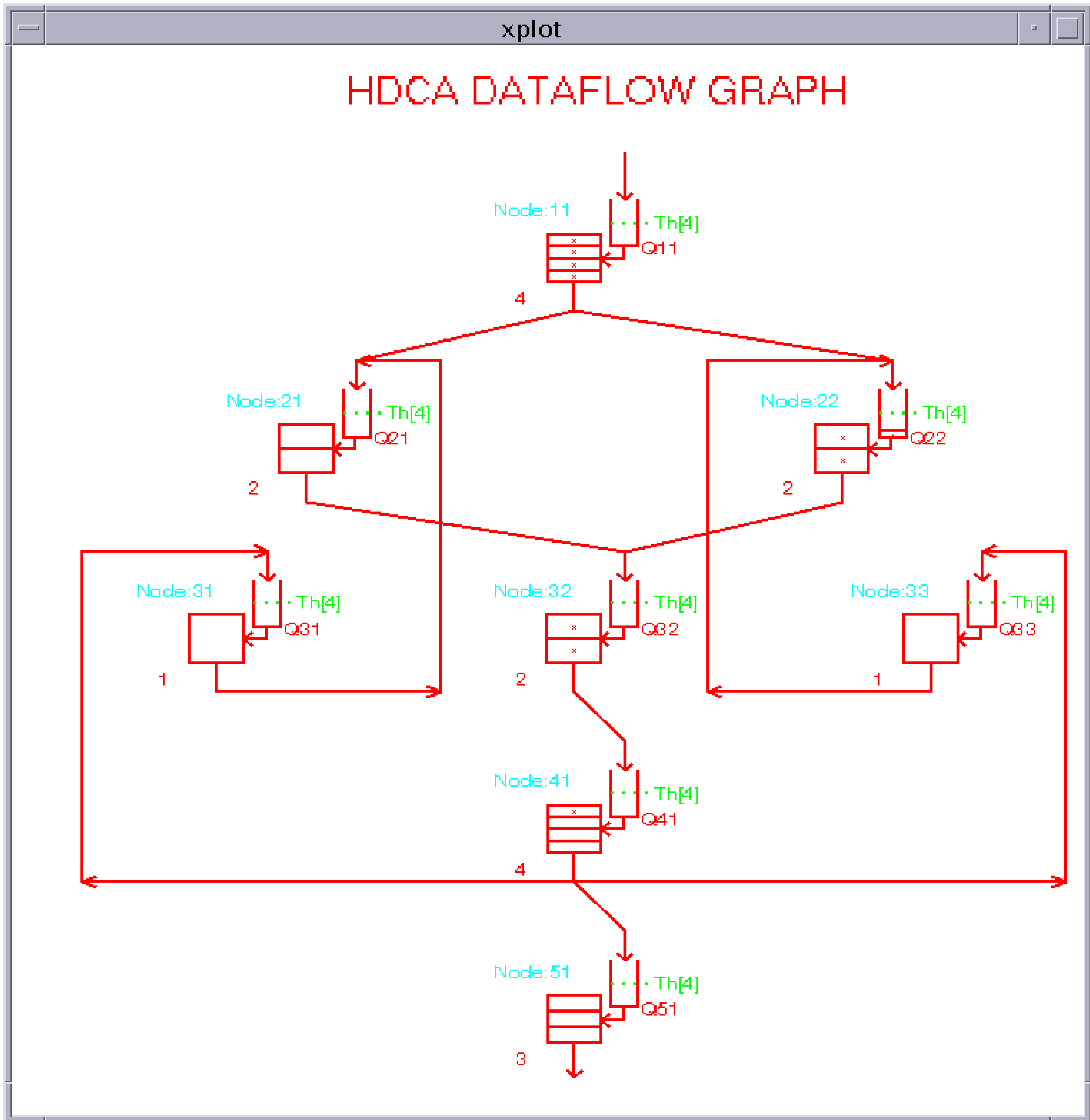


Figure 5.13 b Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =2000 micro-cycles)

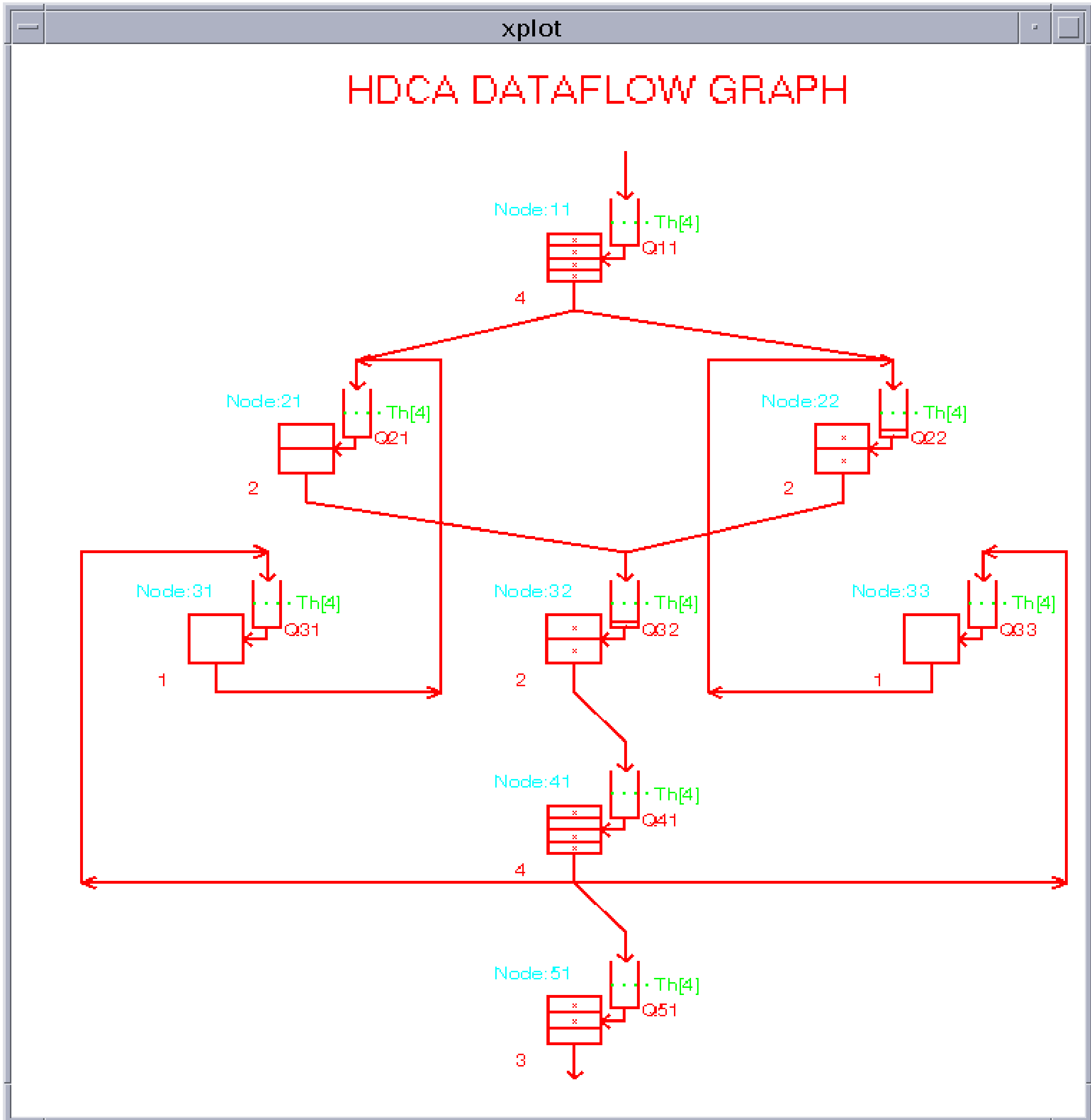


Figure 5.13 c Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =3000 micro-cycles)

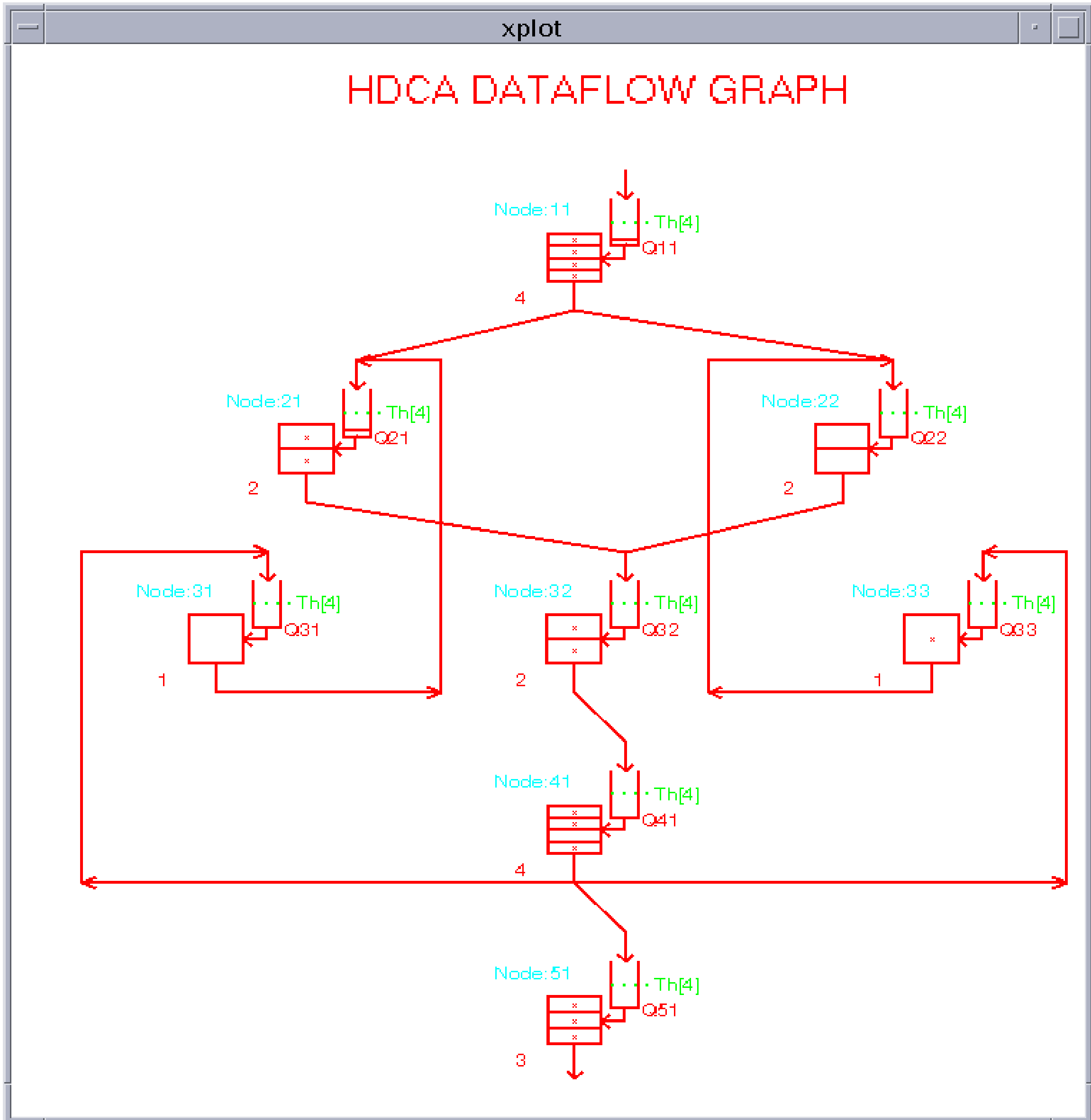


Figure 5.13 d Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =4000 micro-cycles)

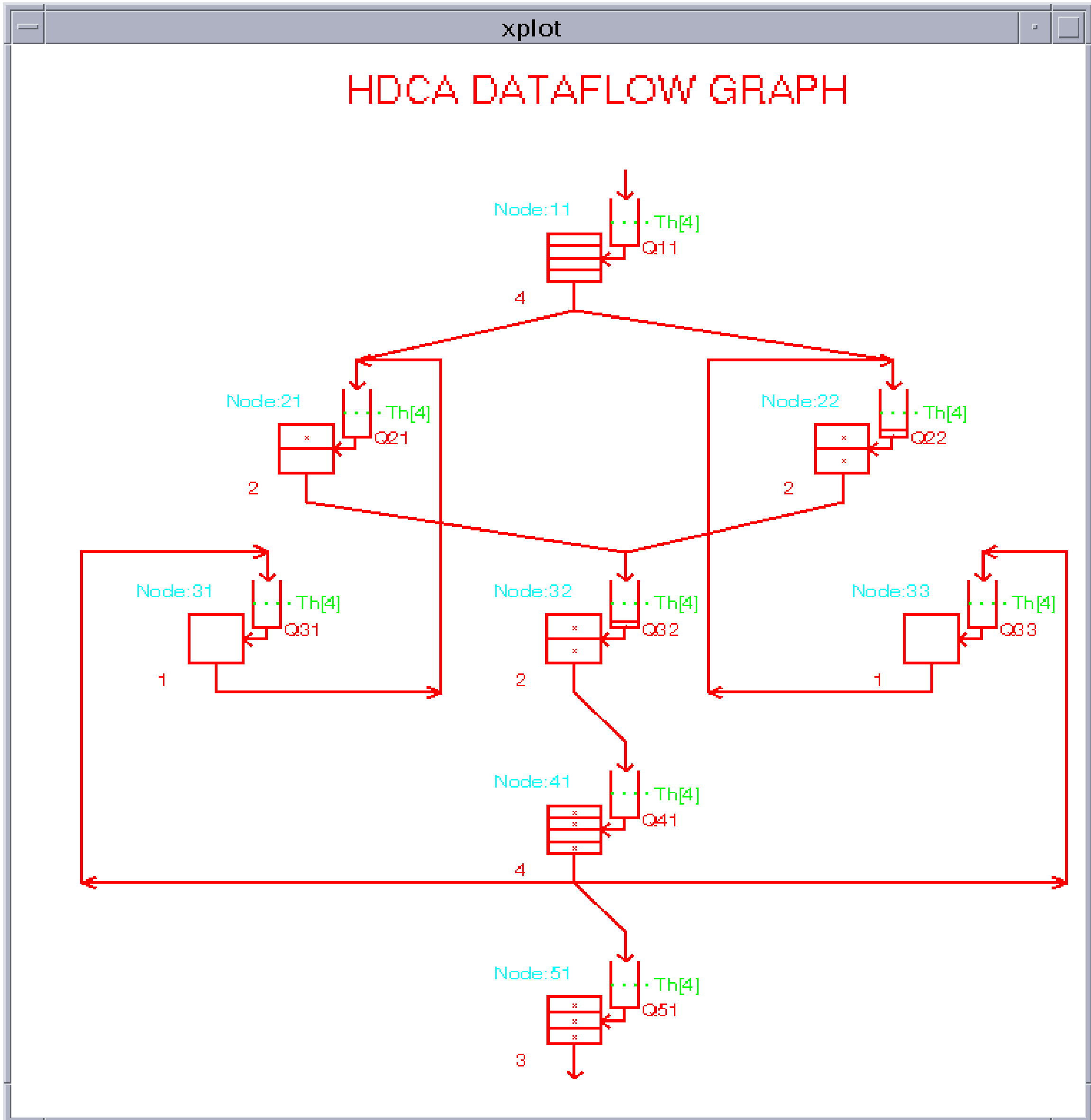


Figure 5.13 e Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =5000 micro-cycles)

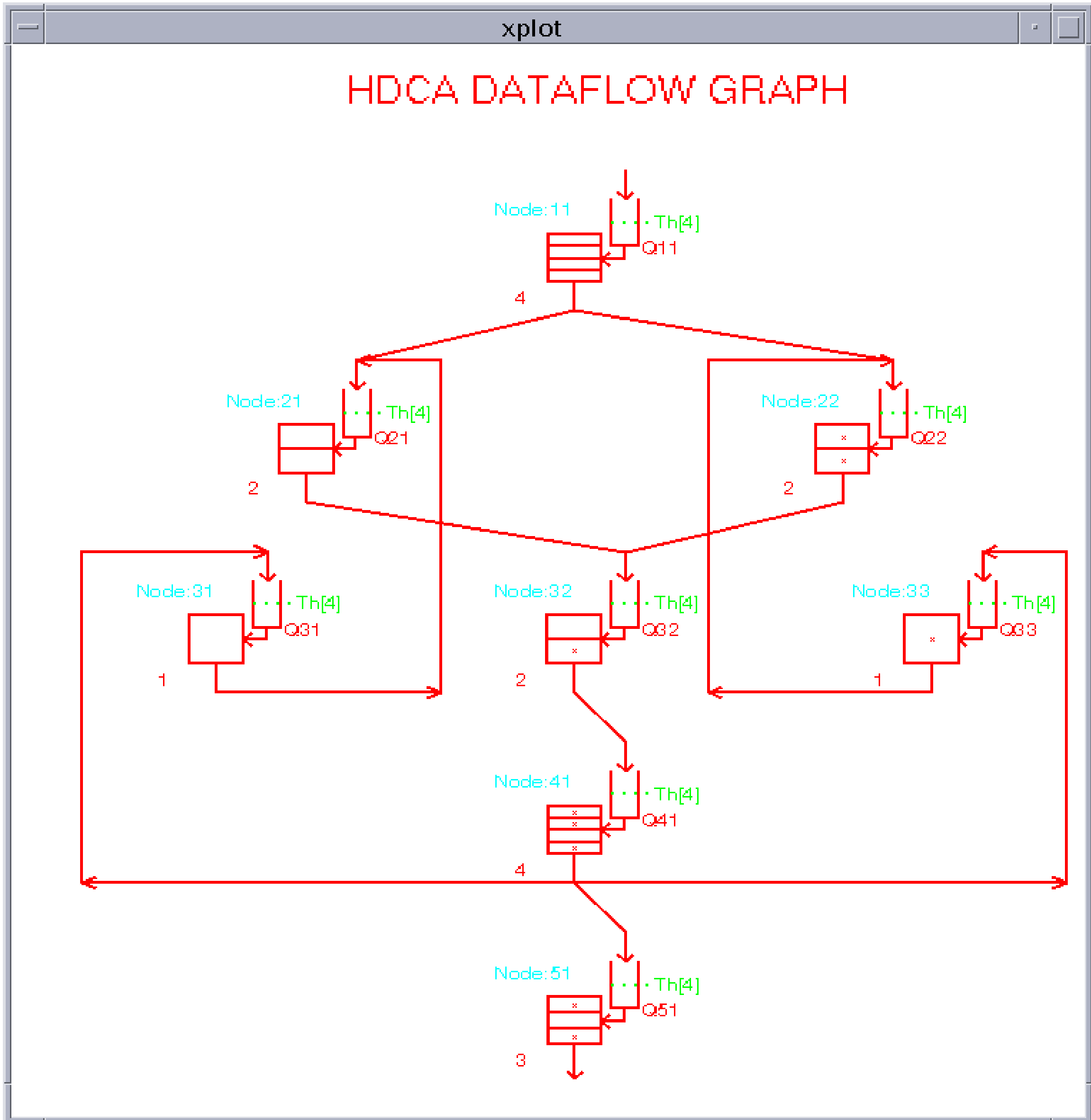


Figure 5.13 f Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =6000 micro-cycles)

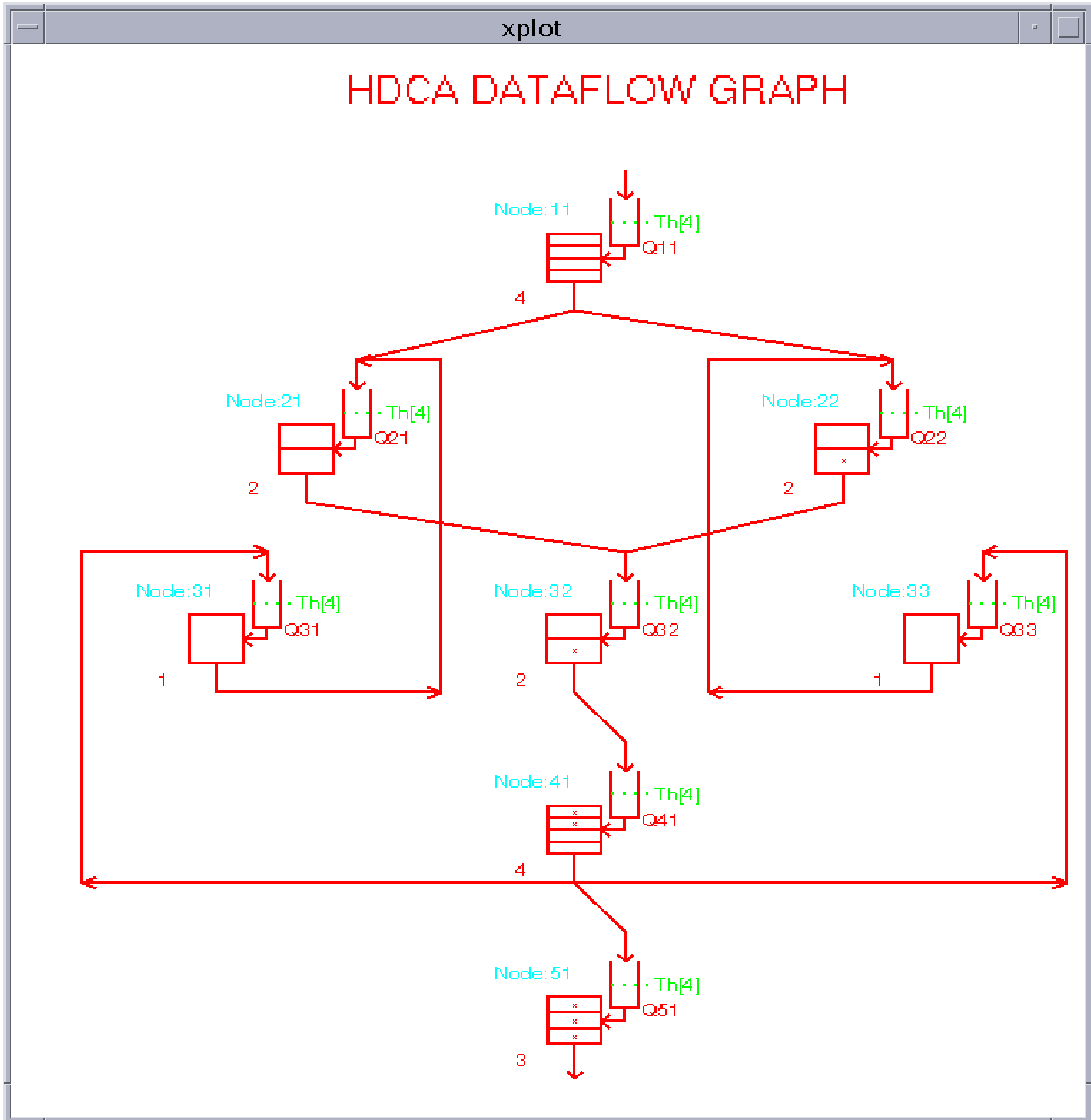


Figure 5.13 g Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =7000 micro-cycles)

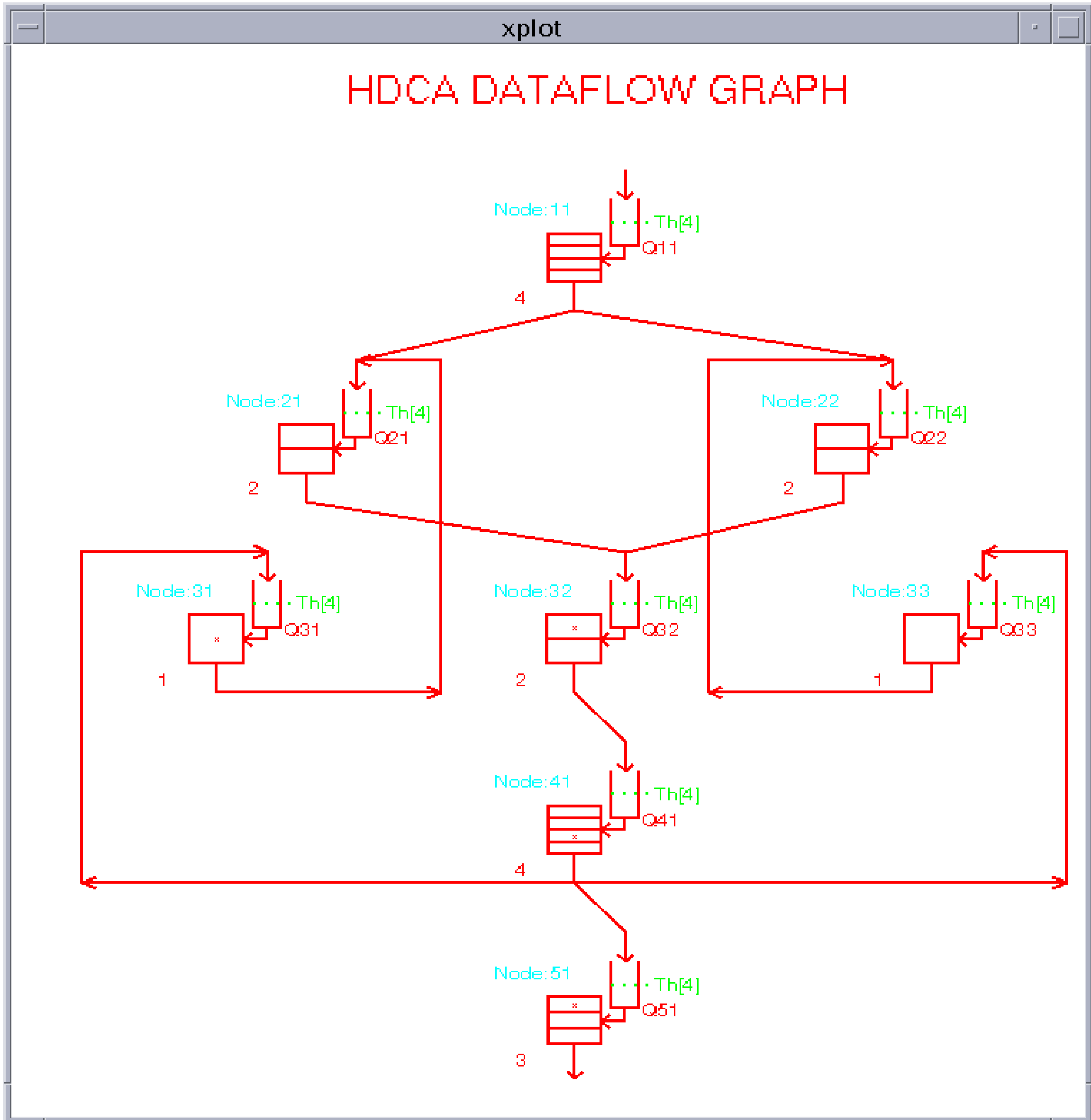


Figure 5.13 h Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =8000 micro-cycles)

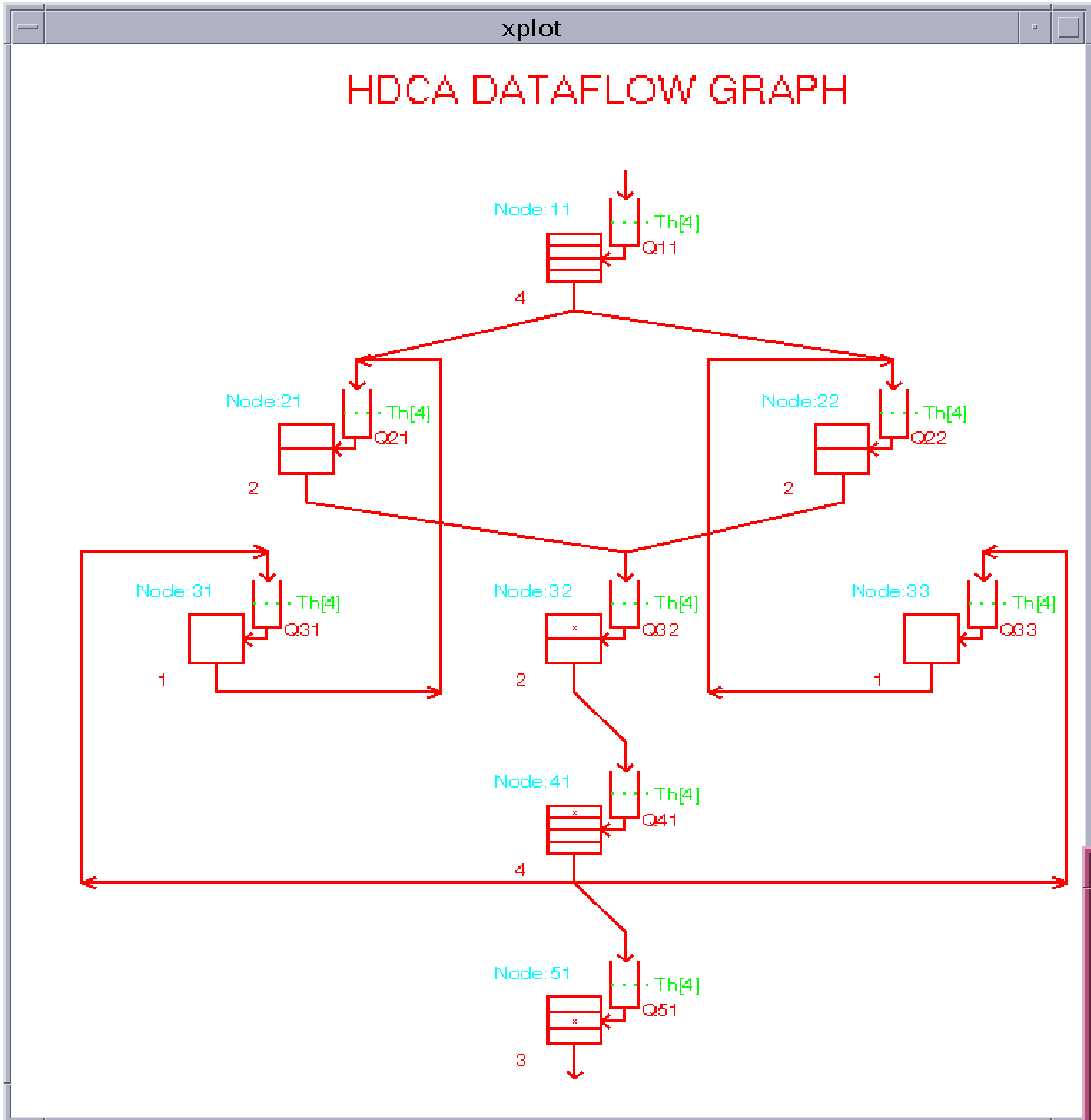


Figure 5.13 i Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =9000 micro-cycles)

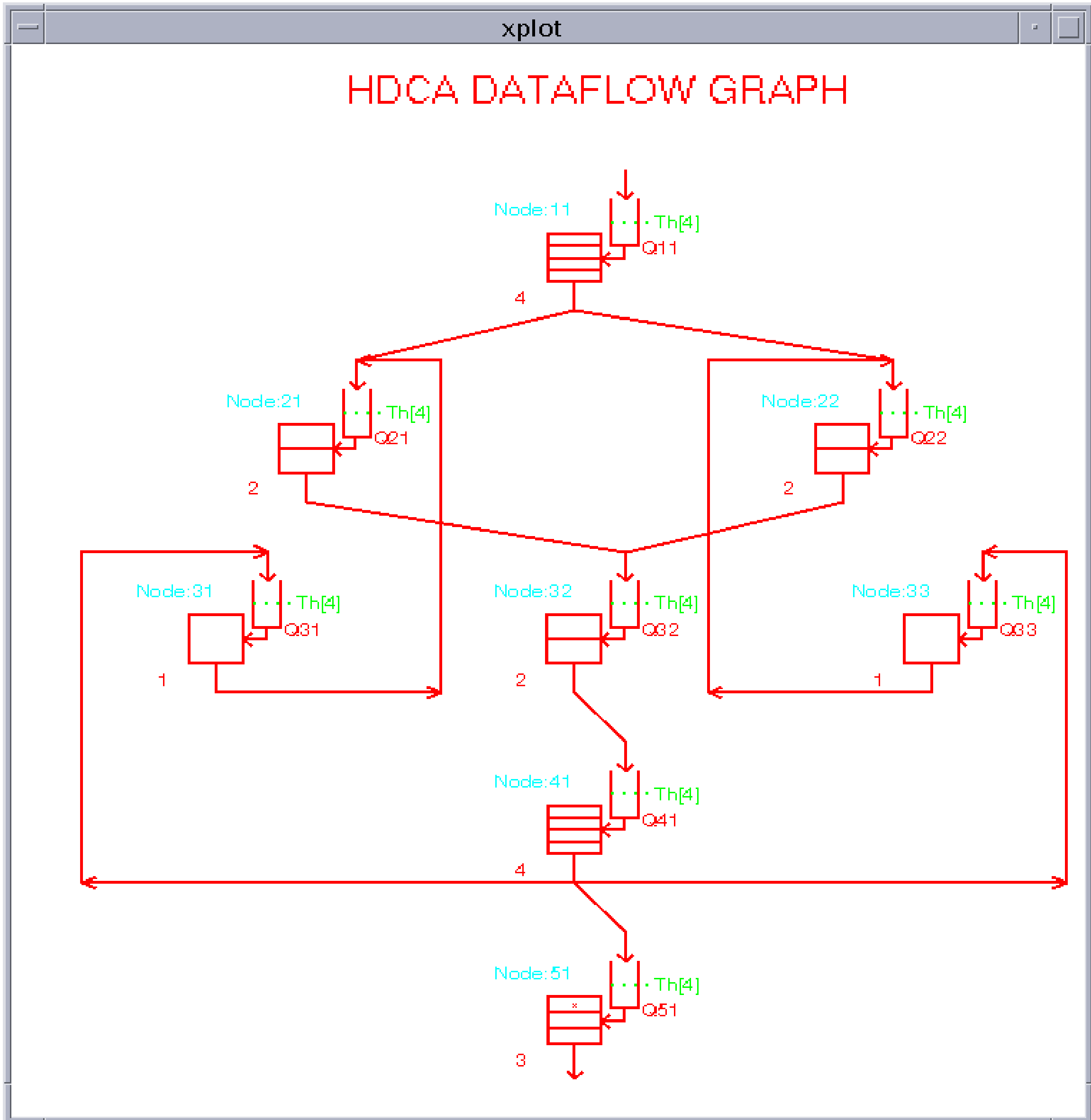


Figure 5.13 j Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =10000 micro-cycles)

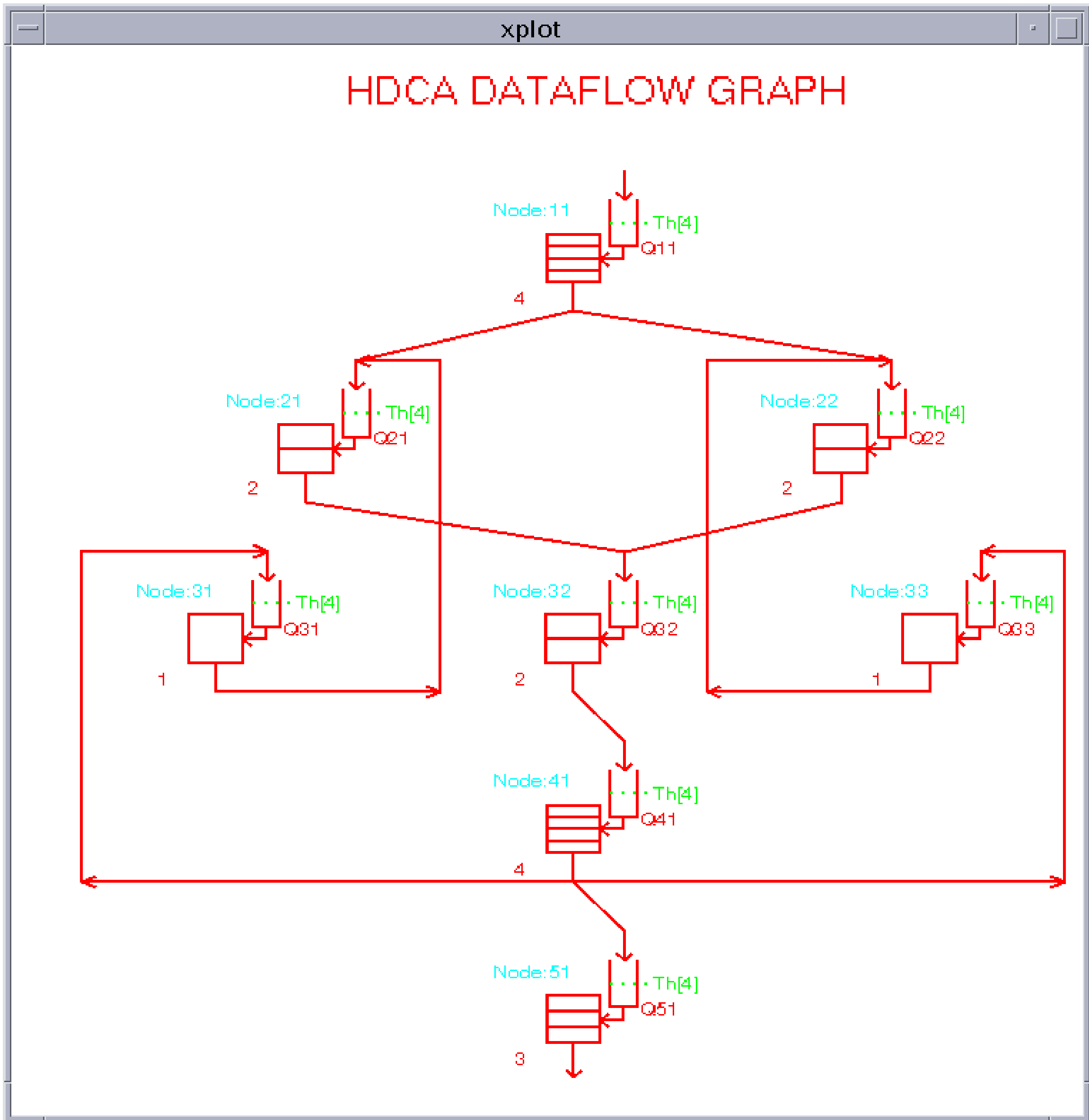


Figure 5.13 k Simulation Result for Application 3 for input rate = 200 micro-cycles/token (t =11000 micro-cycles)

Total Simulation Time = 10618 Micro-cycles

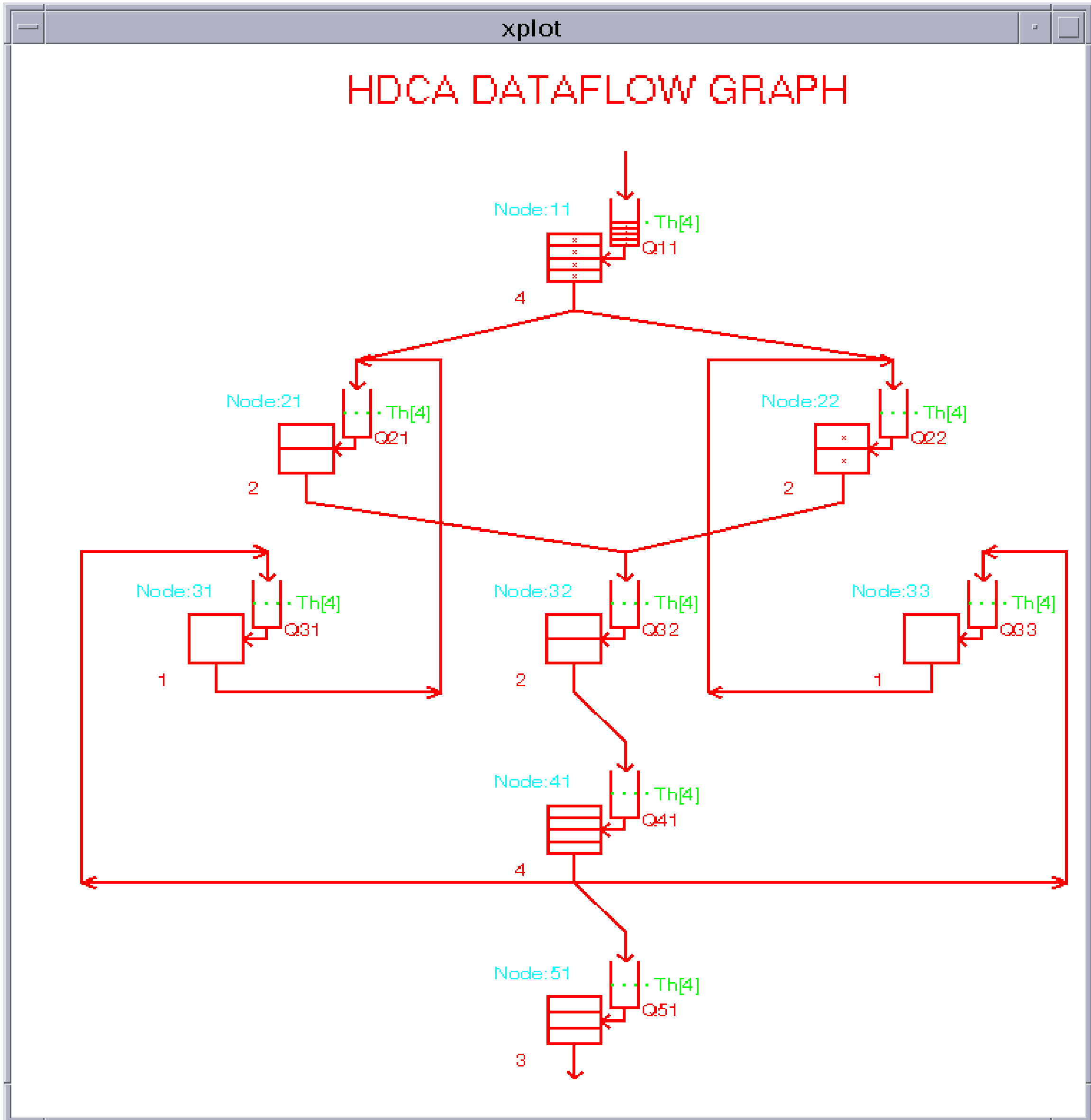


Figure 5.14 Simu. Results of Application 3 (Input rate = 100 micro-cycles/token) (t =1000 micro-cycles)

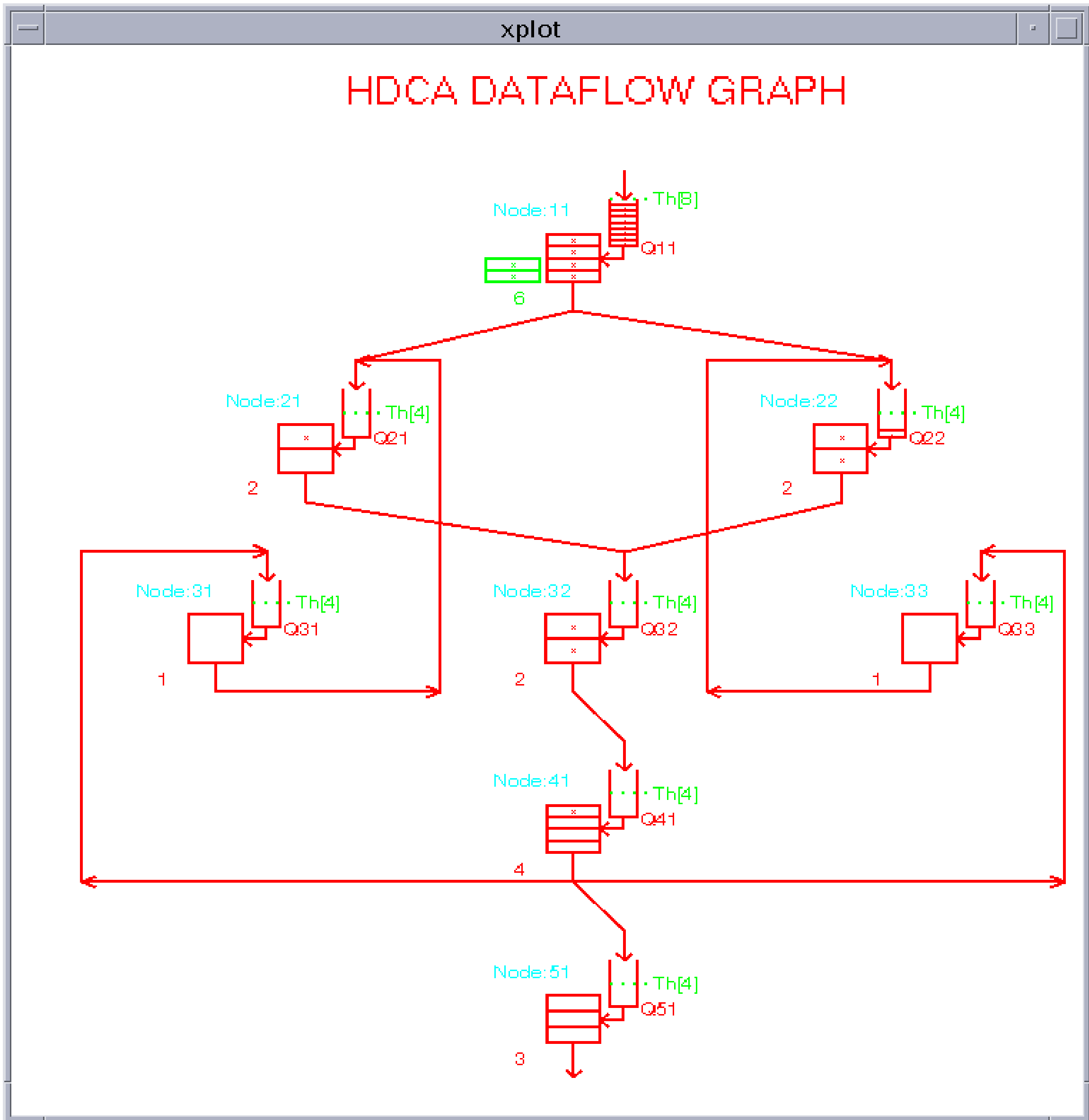


Figure 5.14 b Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =2000 micro-cycles)

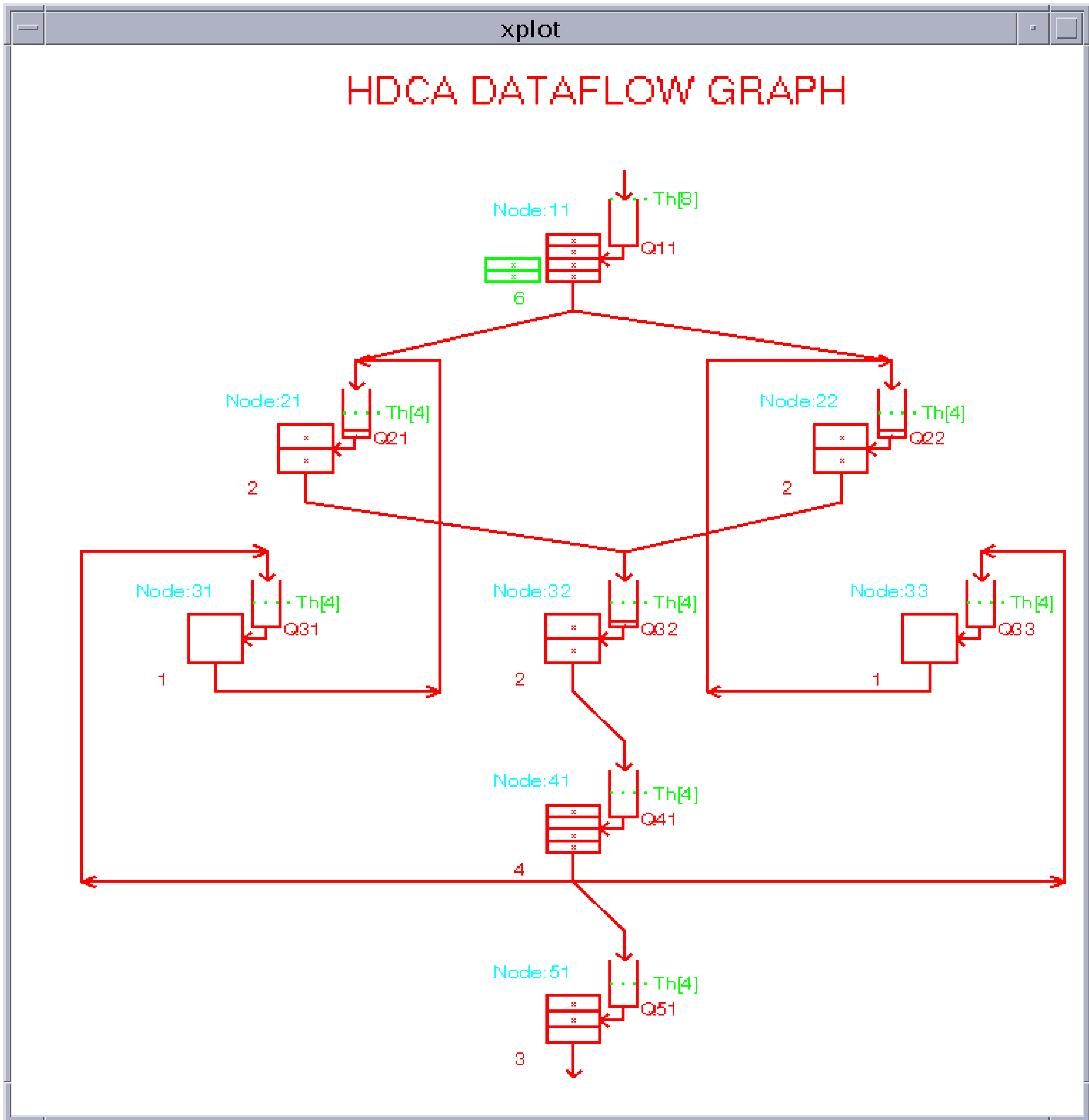


Figure 5.14 c   Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =3000 micro-cycles)

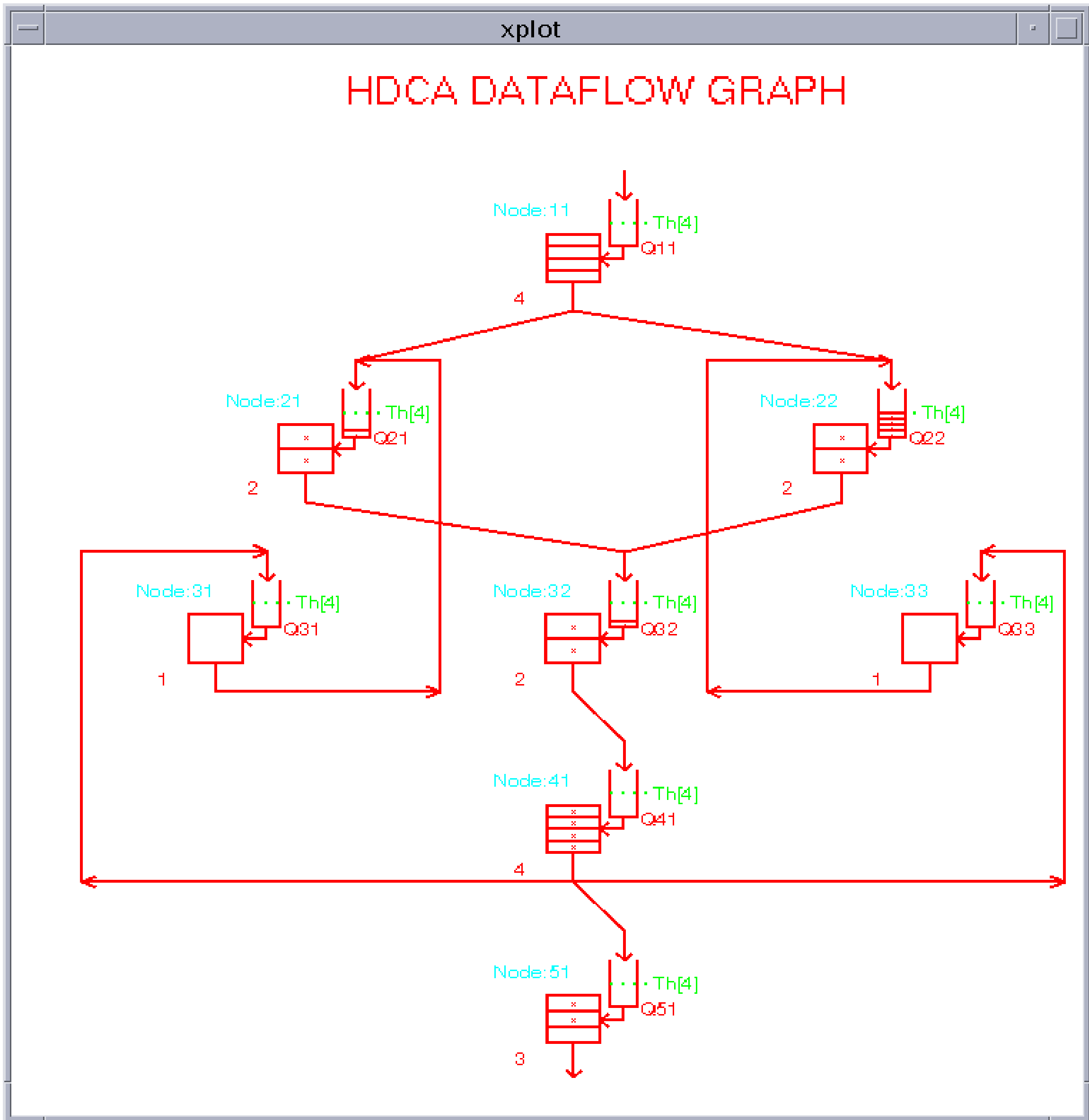


Figure 5.14 d   Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =4000 micro-cycles)

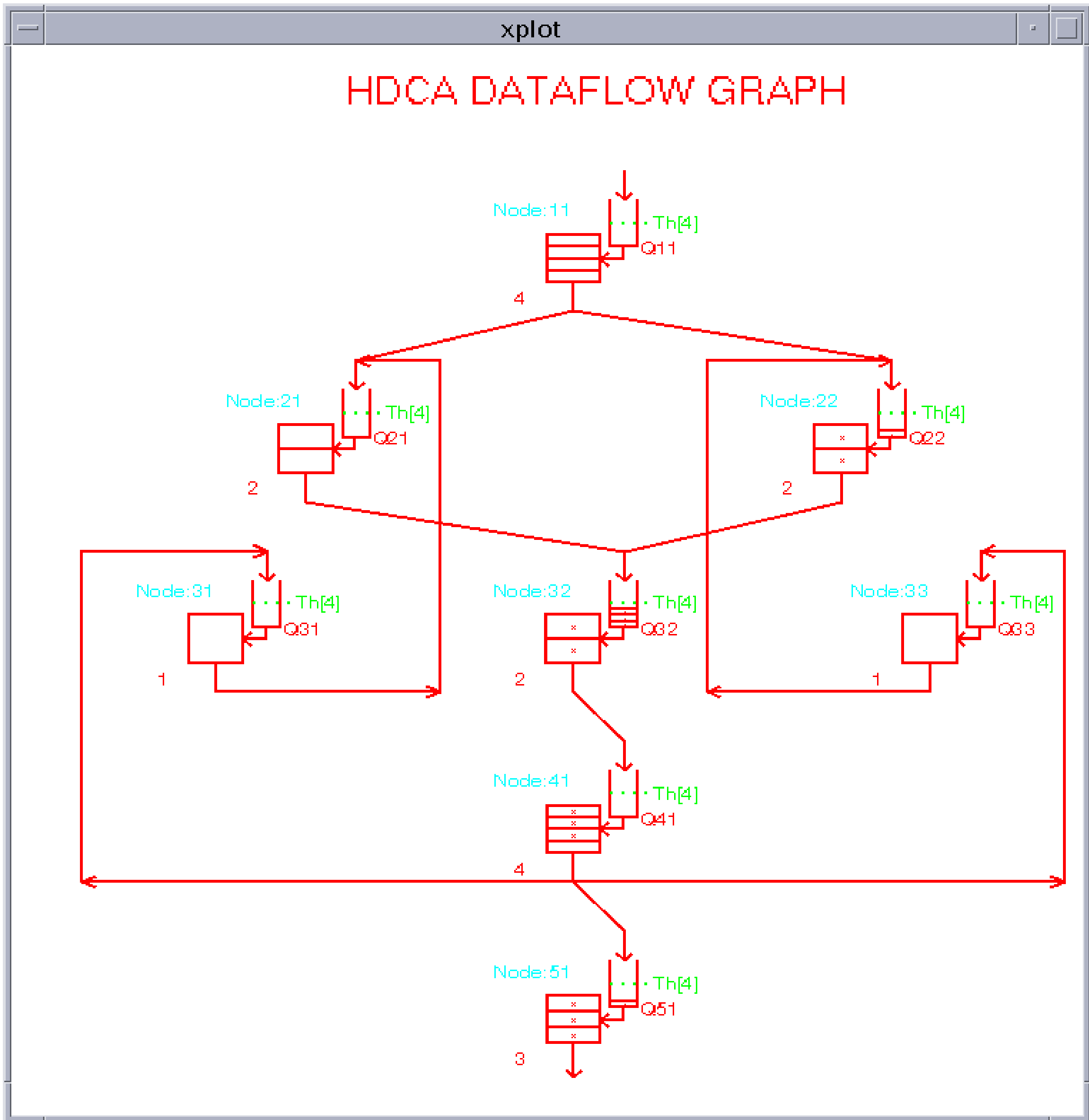


Figure 5.14 e Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =5000 micro-cycles)

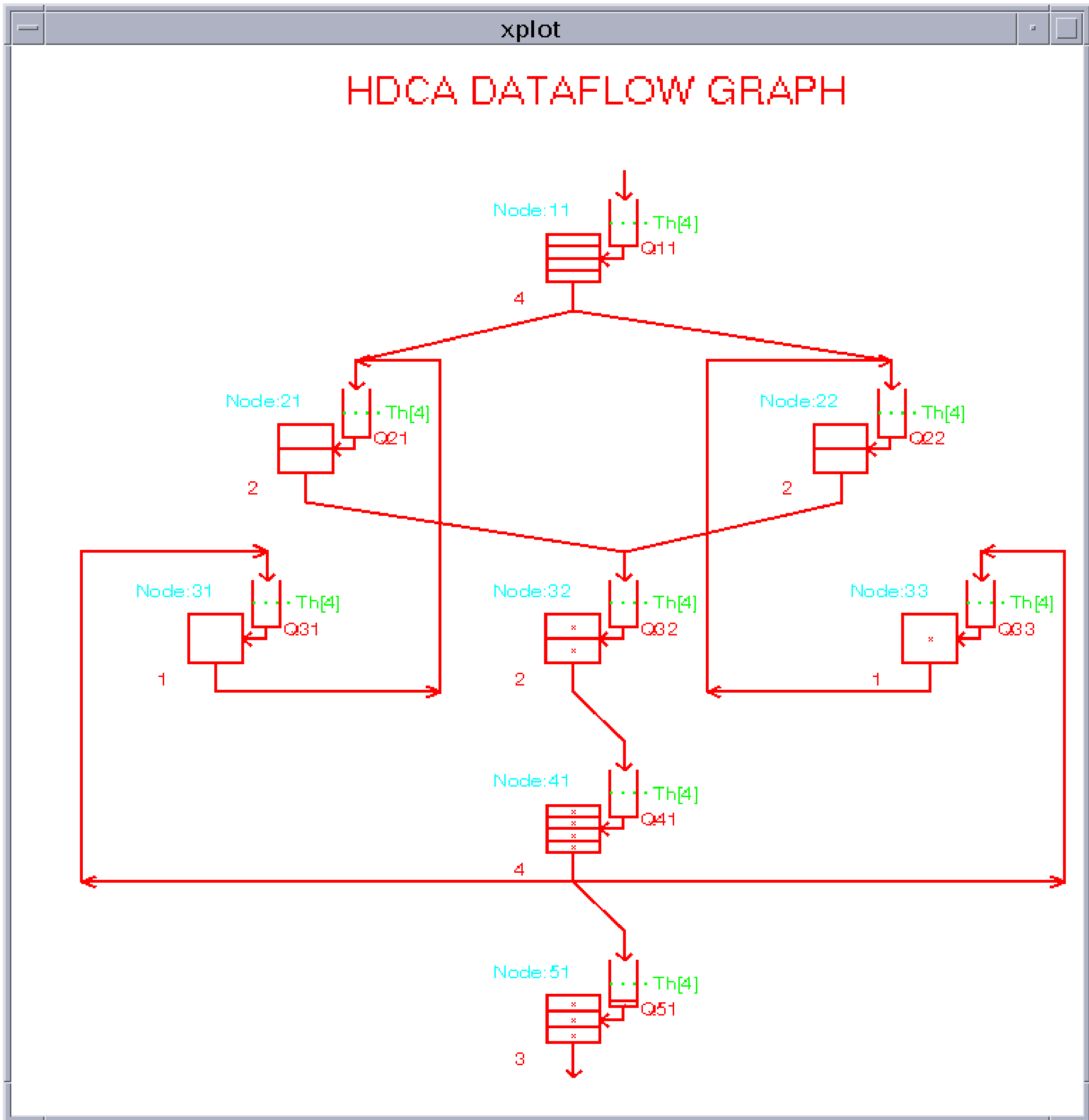


Figure 5.14 f Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =6000 micro-cycles)

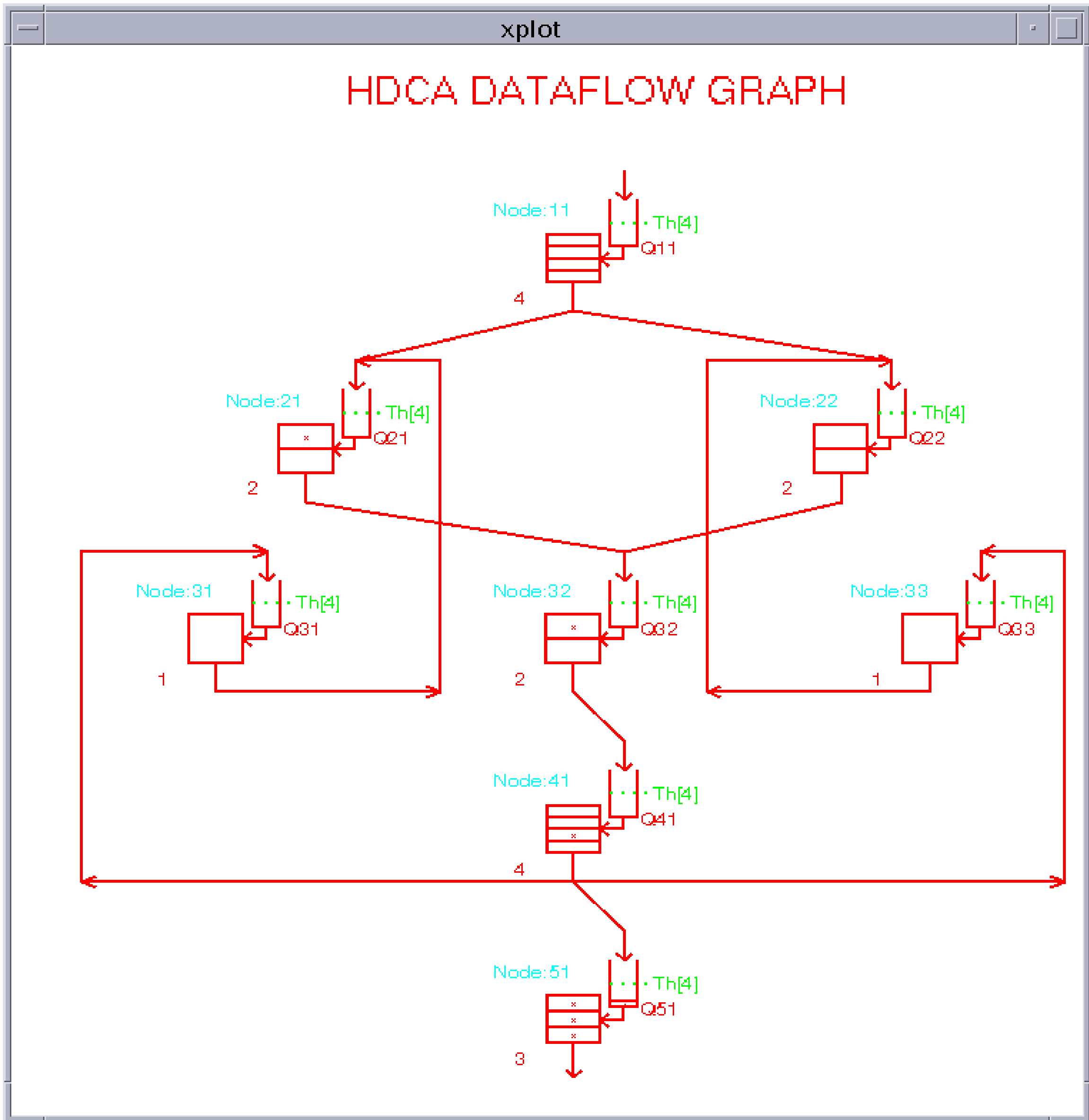


Figure 5.14 g Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =7000 micro-cycles)

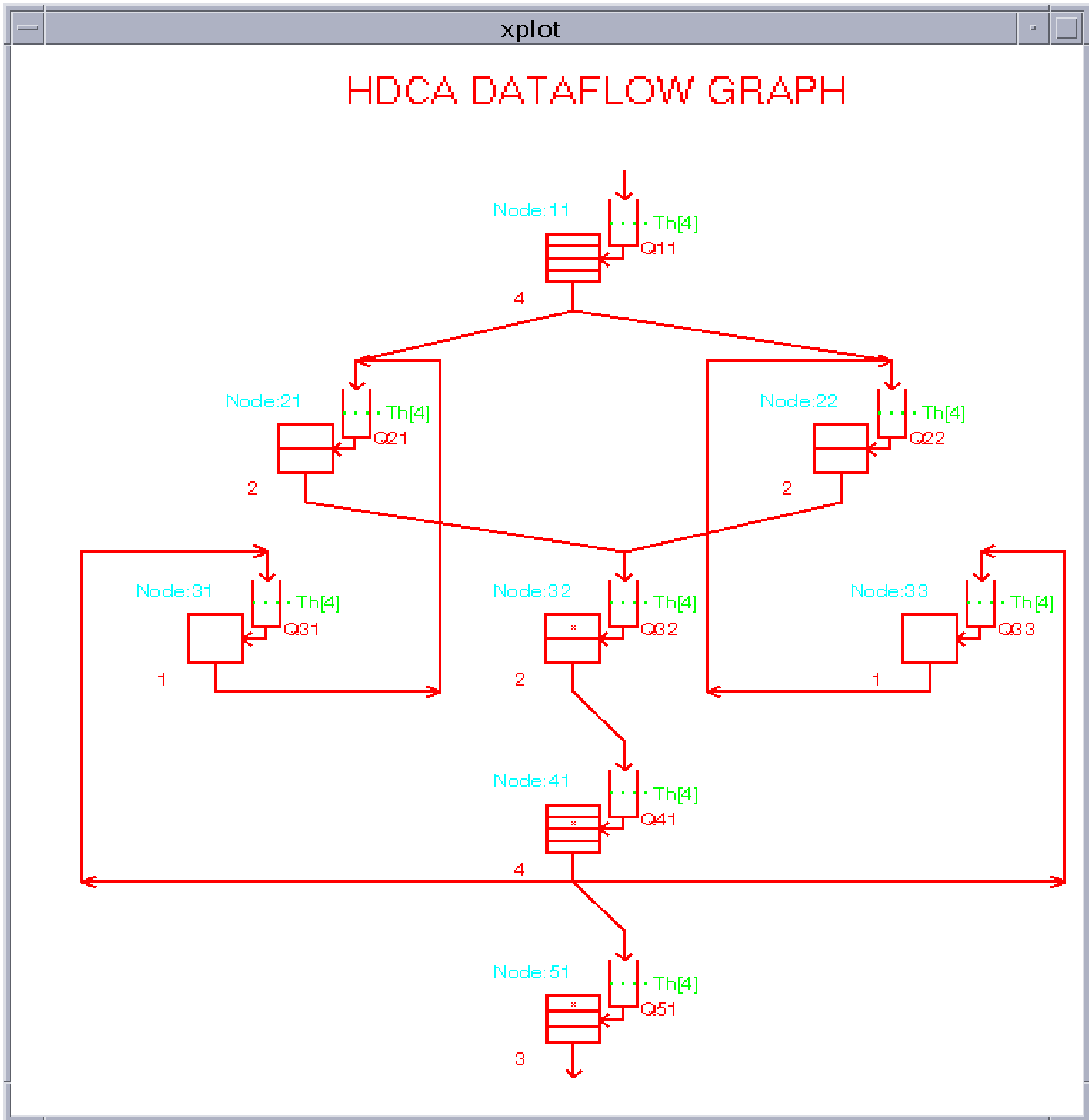


Figure 5.14 h   Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =8000 micro-cycles)

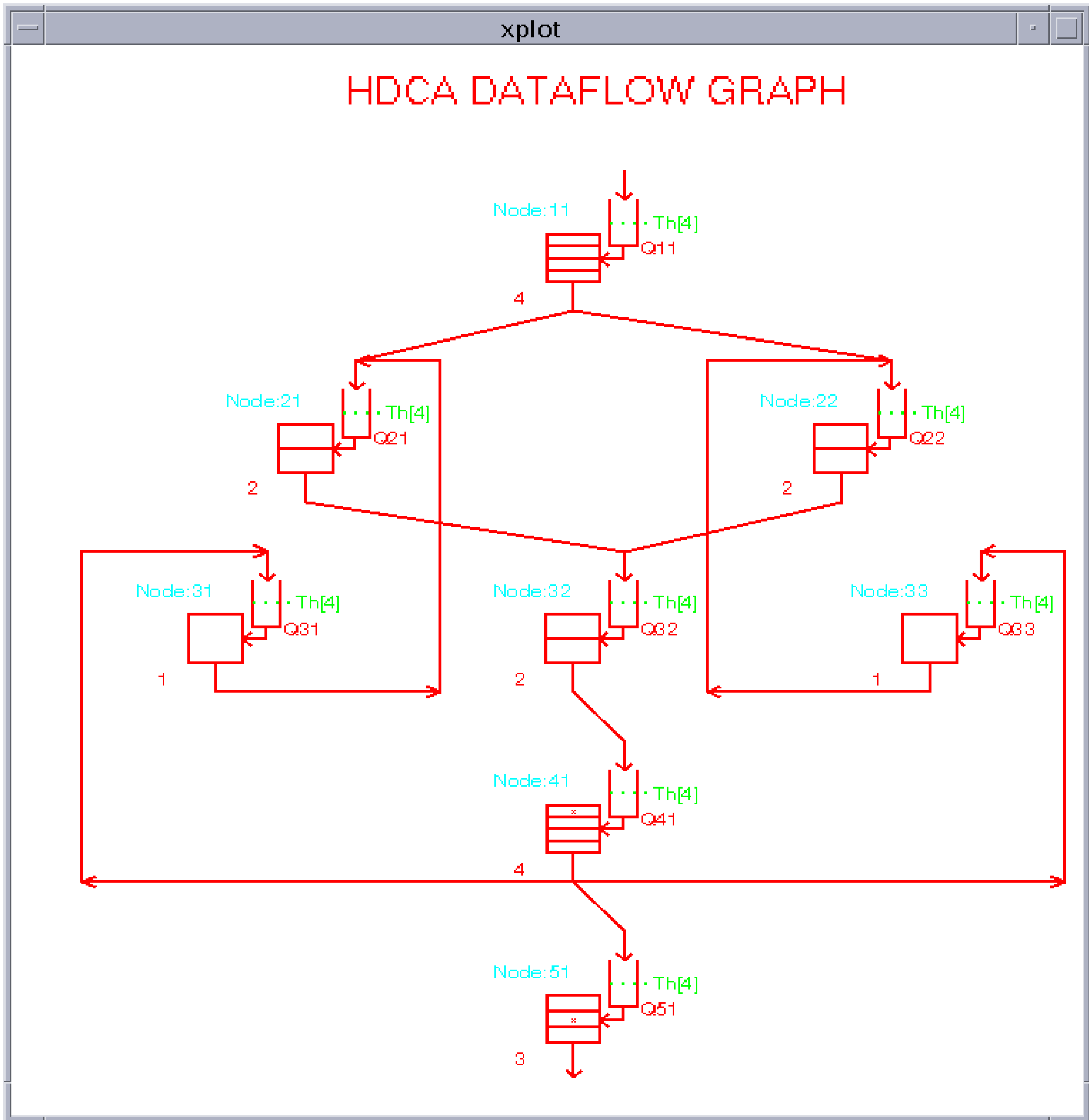


Figure 5.14 i Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =9000 micro-cycles)

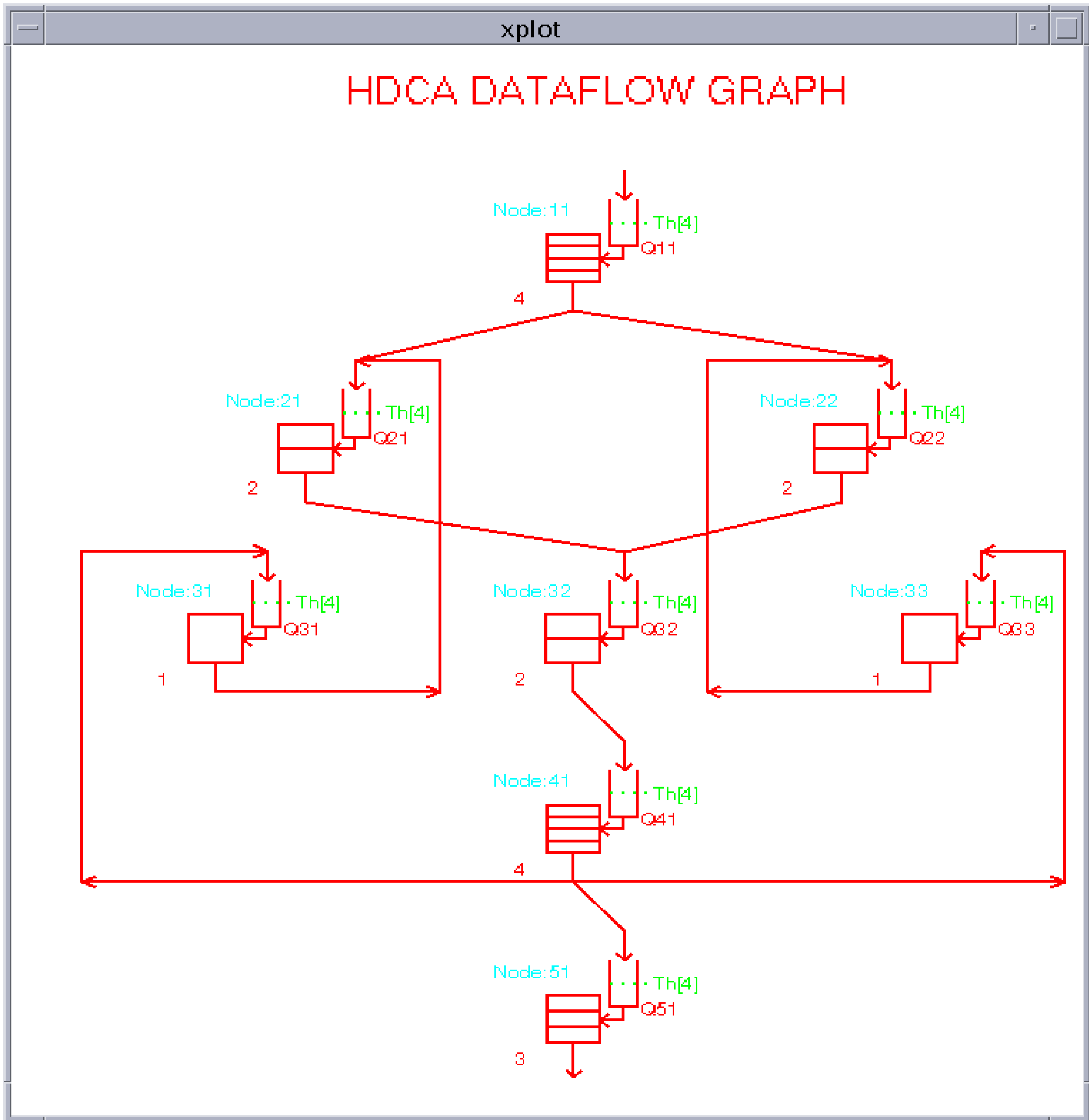


Figure 5.14 j Simulation Result for Application 3 for input rate = 100 micro-cycles/token (t =10000 micro-cycles)

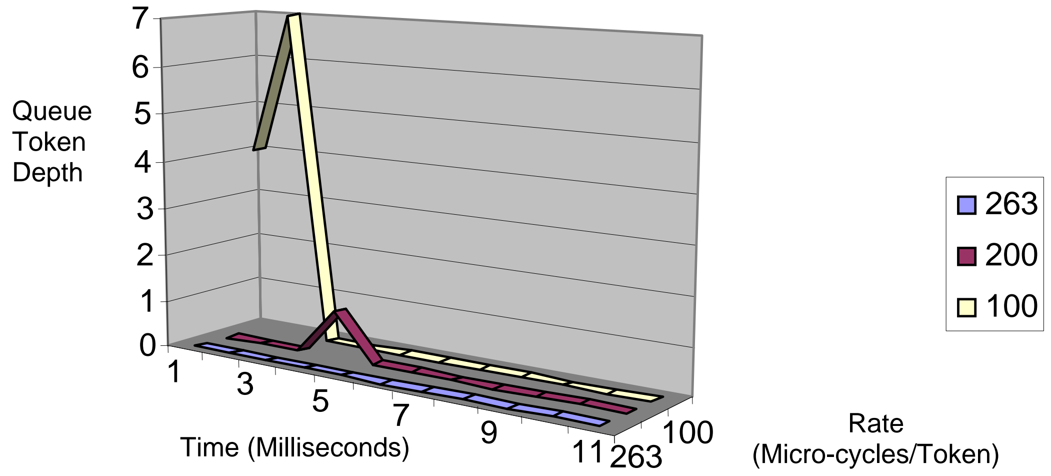
Total Simulation Time = 9782 Micro-cycles

Table 5-7: Application 3 Results: 20 Tokens at Node11

Case3: 20 Tokens

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	11000
Input rate = 263 Microseconds Per DISV											
Node11(4)	0/3/0	0/3/0	0/3/0	0/4/0	0/4/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(2)	0/0/0	0/1/0	0/1/0	0/2/0/	0/0/0	0/0/0	0/0/0	0/2/0	0/0/0	0/0/0	0/0/0/
Node22(2)	0/1/0	0/2/0	1/2/0	0/2/0	1/2/0	1/2/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0
Node31(1)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node32(2)	0/0/0	0/1/0	0/0/0	0/1/0	1/2/0	0/2/0	0/2/0	0/0/0	0/1/0	0/0/0	0/0/0
Node33(1)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node41(4)	0/0/0	0/1/0	0/3/0	0/3/0	0/3/0	0/3/0	0/3/0	0/2/0	0/1/0	0/0/0	0/0/0
Node51(3)	0/0/0	0/0/0	0/2/0	0/1/0	0/2/0	0/2/0	0/2/0	0/1/0	0/1/0	0/1/0	0/0/0
Input rate = 200 Microseconds Per DISV											
Node11(4)	0/4/0	0/4/0	0/4/0	1/4/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node21(2)	0/0/0	0/0/0	0/0/0	1/2/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node22(2)	0/1/0	1/2/0	1/2/0	0/0/0	1/2/0	0/2/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0
Node31(1)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/1/0	0/0/0	0/0/0	0/0/0
Node32(2)	0/0/0	0/2/0	1/2/0	0/2/0	1/2/0	0/1/0	0/1/0	0/1/0	0/1/0	0/0/0	0/0/0
Node33(1)	0/0/0	0/0/0	0/0/0	0/1/0	0/0/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Node41(4)	0/0/0	0/1/0	0/3/0	0/3/0	0/3/0	0/3/0	0/2/0	0/1/0	0/1/0	0/0/0	0/0/0
Node51(3)	0/0/0	0/0/0	0/2/0	0/3/0	0/3/0	0/2/0	0/3/0	0/1/0	0/1/0	0/1/0	0/0/0
Input rate = 100 Microseconds Per DISV											
Node11(4)	4/4/0	7/6/2	0/6/2	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	
Node21(2)	0/0/0	0/1/0	1/2/0	1/2/0	0/0/0	0/0/0	0/1/0	0/0/0	0/0/0	0/0/0	
Node22(2)	0/2/0	1/2/0	1/2/0	4/2/0	1/2/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	
Node31(1)	0/0/0	0/0/0/	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	
Node32(2)	0/0/0	0/2/0	1/2/0	1/2/0	3/2/0	0/2/0	0/1/0	0/1/0	0/0/0	0/0/0	
Node33(1)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/1/0	0/0/0	0/0/0	0/0/0	0/0/0	
Node41(4)	0/0/0	0/1/0	0/3/0	0/4/0	0/3/0	0/4/0	0/1/0	0/1/0	0/1/0	0/0/0	
Node51(3)	0/0/0	0/0/0	0/2/0	0/2/0	1/3/0	1/3/0	1/3/0	0/1/0	0/1/0	0/0/0	

Queue token depth plots for each node are shown in Figure 5.15 except for node 31, 33, and 41. The queue token depth plots for these three nodes are omitted because the queue depth are all zeros for all three input rates based on the simulation results shown in Figures 5.12, 5.13, and 5.14. It is easy to see from Figure 5.15 that the queue token depth is much higher when the input rate is 100micro-cycles/token for all nodes. When the input rate is 200micro-cycles/token, the queue depth equals to 1 at node 11, 21, 22, and 32 at certain time. This is what we expected. But when the input rate is 263 micro-cycles/token, theoretically there should be no token in the queue at any time, but we got one token at node 22 at time 3000, 5000, and 6000 micro-cycles and one token at node 32 at time 5000 micro-cycles. Does this mean that the simulation is not correct? Consider how the input file "copyset" was formed for program "COPY." In this file, it is listed that there are 4 pipelines in this graph: 11→21→32→41→51; 11→21→32→41→31→21→32→41→51; 11→22→32→41→51; 11→22→32→41→33→22→32→41→51. This is based on the assumption that the token only circulates in the graph once. That is, when the token reaches node 41 the second time, it will be routed to node 51. But actually it is still possible to be routed back to node 31 or node 33. So the actual input rates for node 21, 22, 31, 32, and 33 are a little bit higher than the input rates that we used. With this fact in mind, it is still reasonable to have one token in the queue occasionally. So the simulation result is still correct!



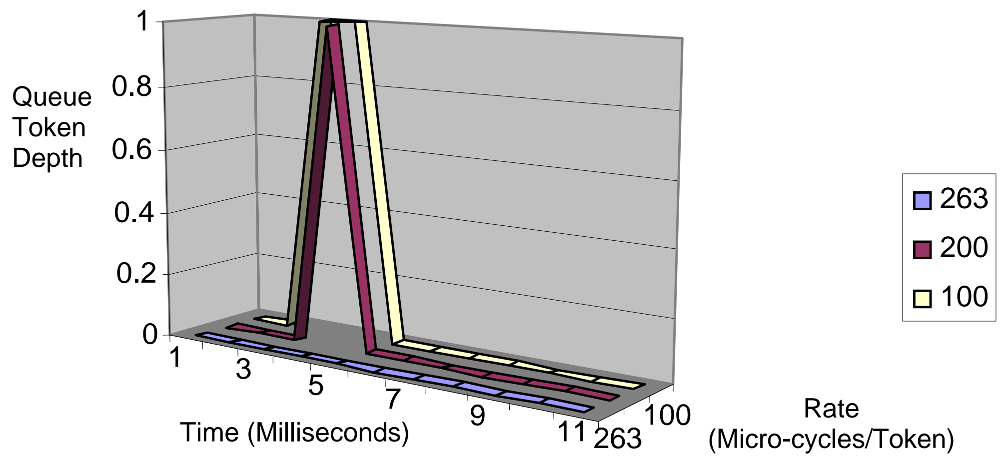
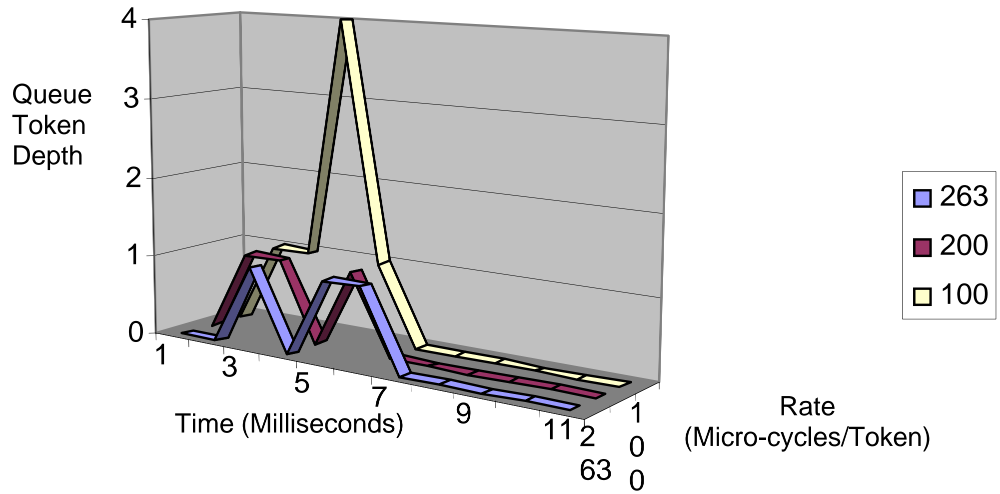


Figure 5.15 Queue Depth Plot for Application 3



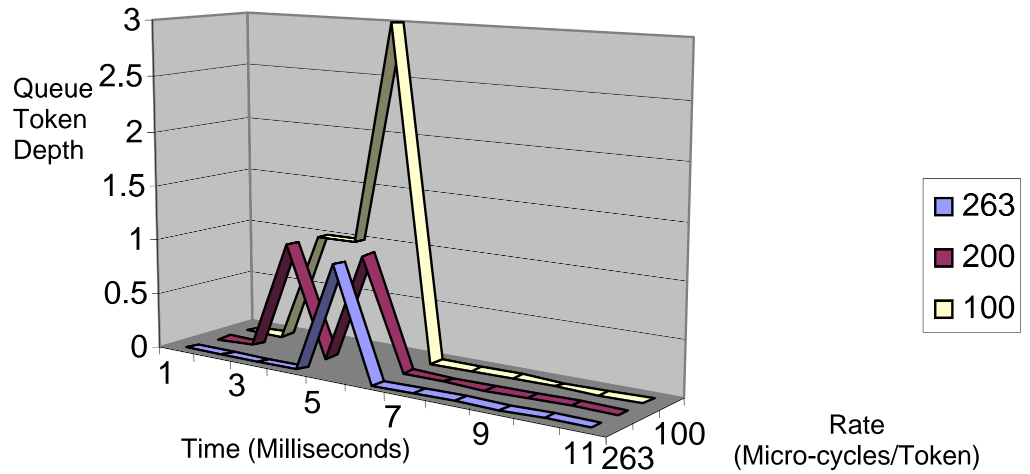


Figure 5.15 Queue Depth Plot for Application 3 (Continued 2)

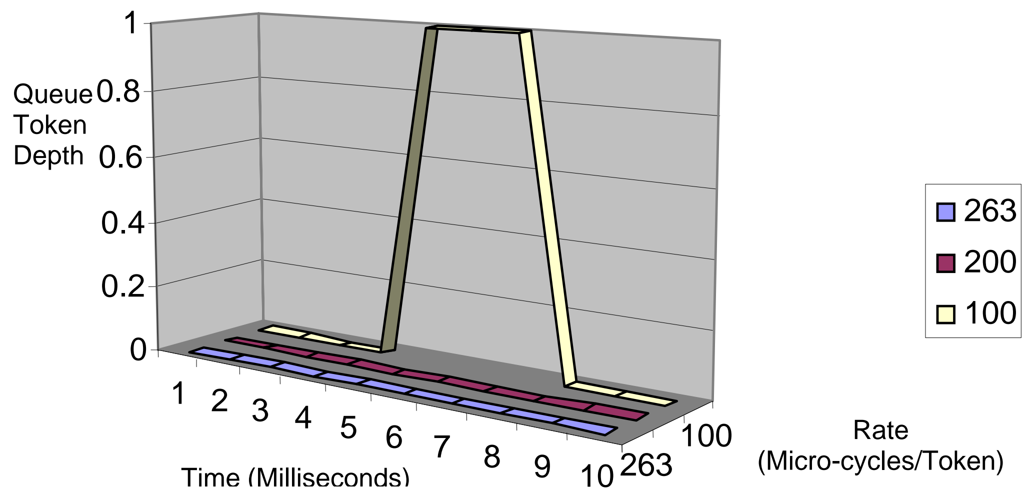


Figure 5.15 Queue Depth Plot for Application 3 (Continued 3)

6 CONCLUSIONS AND FUTURE RESEARCH

A new graphic software "hdca" has been developed, tested and evaluated for simulating an application described by a dataflow (process flow) graph running on the HDCA. This new software first utilizes the result of a "static resource allocation" algorithm to statically assign resources to meet the timing requirement of the application; then it simulates the HDCA architecture executing the application using statically assigned resources by graphically displaying the parameters which are important to the architectures operation and performance. In doing such, a user can visually observe dynamic load balancing and resource allocation characteristics of the architecture from the simulation graph. Observable characteristics include flow of control tokens, changes in queue levels, load distribution, load work-flow progress, process overload detection, start up of idle processors and their cessation after workload reduction. The user can also study the effect of different input rates on dynamic load balancing activity and overall system performance and the fault tolerance analysis of the architecture by using this new software

Chapter one included the background about the HDCA architecture. Chapter two provided a brief overview of the architecture and the application mapping and load-balancing strategy of HDCA. Chapter three reviewed queuing theory models for basic components of dataflow graphs representing computer algorithms and the static load-balancing algorithm that was used in program "COPY." Chapter four presented the algorithm for the simulation program "hdca" in great detail. Chapter five showed simulation results and analysis for three applications. From the simulation results in Chapter 5, we easily saw that the queue token level is indeed very sensitive to the input rate. When the input rate increases, the queue token depth increases too. Queue depth affects the whole architecture. The simulation behaved in a predictable manner and the results obtained were as expected. The overall performance of the simulator is very good.

The program "hdca" has the basic framework required for simulation of the HDCA architecture and would be a good candidate for incorporating future developments and adding new functionalities to the architecture. Further research can add queues for all different processors, which can simulate HDCA more accurately. Using a 3D potting

utility may give a better graph. A better visual graph will lead to better understanding of the concepts of the HDCA architecture.

APPENDIX A C Code for Program copy.c

```
/*This Program is a "C" translation of "copy" written in "BASIC" by Jim Cochran
(see A Dynamic Computer Architecture for Data Driven System: Final report,
chap. 5 page nos.5-45,5-50), then translated by Matura Suryanarayana Rao.
This program caculates the number of copies required at each node to minimize
the clog point effect in the pipeline */

#include <stdio.h>
#define SEL 1

main()
{
  int a,a2,a3,c8,c9,i,i1,j,j1,k,l,m1,n,p,x,x2,x5,x9,w,y;
  int c5[200],c6[200][2],c7[200][12],f[20],j2[200][20],n1[20],s[20],v[200],
     v1[20],v2[20],v3[200],v4[20];
  int q,q1,q2,q3;
  int t1,t2,t3,t4,t5,t6,t9;
  int cplevel,cpnode,cpl,cpn,cp,cp1,sel;

  float b2,b3,s1,x1,y1;
  float b1[200],c[200],c4[200],p9[20][20],r[200],r4[200],t[200];

  struct out{
       int copies;
       float time;
  } copyvalue[20][20];

  struct name{
       int level;
       int node;
  }
  z[200];
  FILE *fp,*fp1,*fopen();

  /*Open the file Copyset and get data*/
  fp=fopen("copyset","r");
  x5=0;
  a=1;
  printf("If keyboard entry is desired set KBENTRY to 1,\n");
  printf("and for 'read' from file, set KBENTRY to 0\n");
  printf("KBENTRY=");
  scanf("%d",&sel);
  if(sel==SEL)
  {
     printf("pipeline\n");
     scanf("%d",&p);
  }
  else   fscanf(fp,"%d",&p);
  for(j=1;j<=p;j++)
   {
     s[j]=a;  /* a is the next empty location, s[] is the position of initial node of pipeline */
```

```
    if(sel==SEL)
    {
        printf("how many nodes in pipe %d \n",j);
        scanf("%d",&x);
    }
    else   fscanf(fp,"%d",&x);
    n1[j]=x; /*n1[] is the number of nodes in each pipeline */
    x=n1[j];
    if(sel==SEL)
       printf("nodes in pipe %d\n",x);
    f[j]=a+n1[j]-1; /* f[]: Position of the final node of each pipeline */
    if(sel==SEL)
       /*Enter the names of nodes in each pipe*/
       printf("name of each node in pipe %d\n",j);
    for(i=1;i<=n1[j];i++)
    {
        if(sel==SEL)
        {
           printf("node %d\n",i);
           scanf("%d %d",&q,&q1);
        }
        else  fscanf(fp,"%d %d",&q,&q1);
        z[a].level=q;
        z[a].node=q1;
        a=a+1;
    }
}
if(sel==SEL)
       printf("alpha parameters\n");
for(i=1;i<=f[p];i++)
{
     if(v3[i]==1) goto r630;  //Repeated node
     q=z[i].level;
     q1=z[i].node;
     /*Input Process time in milliseconds followed by amount of memory
     occupied by the Process associated with the node, in Kilobytes*/
     if(sel==SEL)
     {
         printf("%d %d\n",q,q1);
         scanf("%f %f",&x1,&y1);
     }
     else fscanf(fp,"%f %f",&x1,&y1);
     t[i]=x1;  //Process time in miliseconds
     b1[i]=y1; //Amount of memory occupied by process associated by node[i]
     /*assign alpha parameters to same nodes in other pipes*/
     for(j=i+1;j<=f[p];j++)
     {
         q2=z[j].level;
         q3=z[j].node;
         if(q1!=q3||q2!=q) goto r620;
         t[j]=t[i];
         b1[j]=b1[i];
         v3[j]=1;
r620:;
    }
r630:;
```

```
 }
 /*Input Memory available for Process in the Node, in Kilobytes*/
 if(sel==SEL)
 {
     printf("memory per node\n");
     scanf("%f",&b2);
 }
 else   fscanf(fp,"%f",&b2);
 for(i=1;i<=p;i++)
 {
     if(v4[i]==1) goto r820;
     x=s[i];  //x is the first position of each pipeline */
     q=z[x].level;
     q1=z[x].node;
     /*Input maximum and average rates(data items per millisecond)*/
     if(sel==SEL)
     {
           printf("max.rate %d %d\n",q,q1);
           scanf("%f",&x1);
     }
     else   fscanf(fp,"%f",&x1);
     r[x]=x1;
     if (sel==SEL)
     {
          printf("ave.rate\n");
          scanf("%f",&y1);
     }
     else   fscanf(fp,"%f",&y1);
     r4[x]=y1;
     /*assign maximum and averge initial rates for all the nodes*/
     for(j=i+1;j<=p;j++)
     {
          y=s[j];
          q2=z[y].level;
          q3=z[y].node;
          if(q3!=q1||q2!=q) goto r810;
          r[y]=r[x];
          r4[y]=r4[x];
          v4[j]=1;
r810:;
      }
r820:;
        /* donot change input rates*/
        v[x]=2;
        /*Assign initial rate to all nodes.*/
        for (k=s[i]+1;k<=f[i];k++)
        {
             r[k]=r[x];
             r4[k]=r4[x];
        }
}
w=f[p];
v[w+1]=2;
/*Setup the j2 matrix to contain all duplicate nodes.
A duplicate node is one which appears in more than one pipeline. */
j1=0;
```

```
for(j=1;j<=p;j++)
{
     for(i=s[j];i<=f[j];i++)
     {
          if(v[i]==1) goto r1100;
          m1=1;
          q=z[i].level;
          q1=z[i].node;
          for(k=i+1;k<=a;k++)
          {
               q2=z[k].level;
               q3=z[k].node;
               if(q1!=q3||q!=q2) goto r1090;
               m1=m1+1;
               if(m1!=2) goto r1040;
               j1=j1+1;
               j2[j1][2]=i;
r1040:;
               j2[j1][1]=m1;
               j2[j1][m1+1]=k;
               /*Alter v[] for all duplicate nodes
               (nodes appearing in the j2 matrix*/
               v[i]=1;
               v[k]=1;
r1090:;
          }
r1100:;
      }
}
/*Input probabilities if desired.*/
if(sel==SEL)
{
       printf("forks1(yes) or 0(no)\n");
       scanf("%d",&x2);
}
else  fscanf(fp,"%d",&x2);
if(x2==0) goto r1420;
/*change v2[] for all forks.*/
for (i=1;i<=j1;i++)
{
     /*skip terminal nodes.*/
     for (i1=1;i1<=p;i1++)
     {
         if(j2[i][2]==f[i1]) goto r1280;
     }
     for(j=2;j<=j2[i][1]+1;j++)
     {
         x=j2[i][2];
         y=j2[i][j];
         x=x+1;
         y=y+1;
         q=z[x].level;
         q1=z[x].node;
         q2=z[y].level;
         q3=z[y].node;
         if(q1==q3&&q==q2) goto r1270;
```

```
          v2[i]=1;
          goto r1280;
r1270:;
      }
r1280:;
}
/*v2[] now contains 1's in positions corresponding positions of forks in j2 matrix.*/
/*Input probabilities for forks in decimal.*/
if(sel==SEL)
       printf("probability (decimal)\n");
for(i=1;i<=j1;i++)
{
    if(v2[i]!=1) goto r1400;
    for(j=2;j<=j2[i][1]+1;j++)
    {
          x=j2[i][2];
          y=j2[i][j];
          q=z[x].level;
          q1=z[x].node;
          q2=z[y+1].level;
          q3=z[y+1].node;
          if(sel==SEL)
          {
              printf("from %d %d to %d %d \n",q,q1,q2,q3);
              scanf("%f",&x1);
          }
          else    fscanf(fp,"%f",&x1);
            p9[i][j]=x1;
    }
r1400:;
}
/*prob. matrix p9 is complete*/
/*calculate new rates for all the nodes except for source node. */
r1420:;
x9=0;
r1440:;
c9=0;
for(i=1;i<=j1;i++)
{
     /*skip the source node.
     i.e.,donot alter input rate*/
     for (i1=1;i1<=p;i1++)
     {
          if(j2[i][2]==s[i1]) goto r1790;
     }
     s1=0;
     for(i1=1;i1<=11;i1++)
     {
          v1[i1]==0;
     }
     for(j=2;j<=j2[i][1]+1;j++)
     {
          if(v1[j]==1) goto r1710;
          for(l=j+1;l<=j2[i][1]+1;l++)
          {
              x=j2[i][j];
```

```
                y=j2[i][l];
                x=x-1;
                y=y-1;
                q=z[x].level;
                q1=z[x].node;
                q2=z[y].level;
                q3=z[y].node;
                if(q!=q3||q!=q2) goto r1580;
                v1[l]=1;
r1580:;
            }
            /*check for previous fork.*/
            if(x2!=1) goto r1700;
            for (l=1;l<=j1;l++)
            {
                w=j2[1][2];
                x=j2[i][j];
                x=x-1;
                q=z[x].level;
                q1=z[x].node;
                q2=z[w].level;
                q3=z[w].node;
                if(q1!=q3||q!=q2) goto r1690;
                if(v2[1]==0) goto r1690;
                for(k=2;k<=j2[1][1]+1;k++)
                {
                   if((j2[i][j]-1)==j2[l][k])
                    goto r1670;
                }
r1670:;
                x=j2[i][j]-1;
                s1=s1+(r[x]*p9[1][k]);
                goto r1710;
r1690:;
            }
r1700:;
            x=j2[i][j];
            x=x-1;
            s1=s1+r[x]; /*s1 is the new rate*/
r1710:;
        }
        /*if the rate is changed alter matrix r.*/
        y=j2[i][2];
        if(s1==r[y]) goto r1790;
        for(j=2;j<=j2[i][1]+1;j++)
        {
            x=j2[i][j];
            r[x]=s1;
            c9=c9+1;
        }
        /*change rates for all singular nodes following a fork.*/
r1790:;
        for (j=2;j<=j2[i][1]+1;j++)
        {
             for(k=j2[i][j]+1;k<=a;k++)
             {
```

```
                    if(v[k]==1) goto r1920;
                    if(v[k]==2) goto r1920;
                    if(x2==1) goto r1870;
                    x=j2[i][j];
                    s1=r[x];
                    goto r1880;
r1870:;
                    x=j2[i][j];
                    s1=r[x]*p9[i][j];
r1880:;
                    if(s1==r[k]) goto r1920;
                    r[k]=s1;
                    c9=c9+1;
              }
r1920:;
        }
}
x9=x9+1;
if(x9<=20) goto r1980;
printf("converge failed\n");
goto r3010;
r1980:;
if(c9!=0) goto r1440;
/*rate matrix is finalized*/
if(x5==1) goto r2240;
/*caculate the number of copies required of each node.*/
for(i=1;i<=f[p];i++)
{
     c[i]=r[i]*t[i];
}
printf("node # of copies \n");
t9=0;
for(i=1;i<=f[p];i++)
{
       if (v3[i]==1) goto r2140;
       q=z[i].level;
       q1=z[i].node;
       printf("%d %d  %d\n",q,q1,(int)(c[i]+0.99));
       copyvalue[q][q1].copies=(int)(c[i]+0.99);
       copyvalue[q][q1].time=t[i];
       t9=t9+(int)(c[i]+0.99);
r2140:;
}
printf("total %d\n",t9);
copyvalue[0][0].copies=1;
copyvalue[0][0].time=(int)(t[0]);
/*prepare r with ave. rates.*/
for(i=1;i<=f[p];i++)
{
       r[i]=r4[i];
}
x5=1;
goto r1420;
r2240:
/*caculate the copies with ave. rate.*/
for(i=1;i<=f[p];i++)
```

```
{
        c4[i]=r[i]*t[i];
}
/*caculate # copies max. only.(refer to page 5-26 of [5])*/
for(i=1;i<=f[p];i++)
{
      c5[i]=(int)(c[i]+0.99)-(int)(c4[i]+0.99);
}
if(sel==SEL)
{
      printf("if combinable processes desired?1(yes)0(No)\n");
      scanf("%d",&x2);
}
else  fscanf(fp,"%d",&x2);
if(x2==0) goto r3010;
/*search for processes that can be combined for execution by one C.E. (node).*/
a2=0;
for(i=1;i<=f[p];i++)
{
      if (v3[i]==1) goto r2540;
r2410:
      if(c5[i]<=0) goto r2540;
      for(j=i+1;j<=f[p];j++)
      {
          if(v3[j]==1) goto r2530;
          if(c5[j]<=0) goto r2530;
          if((b1[i]+b1[j])>b2) goto r2530;
          a2=a2+1;
          c6[a2][1]=i;
          c6[a2][2]=j;
          c5[i]=c5[i]-1;
          c5[j]=c5[j]-1;
          goto r2410;
r2530:;
      }
r2540:;
}
printf("following pairs are combined in one C.E.(node)\n");
for(i=1;i<=a2;i++)
{
         x=c6[i][1];
         y=c6[i][2];
         q=z[x].level;
         q1=z[x].node;
         q2=z[y].level;
         q3=z[y].node;
         printf("%d %d  %d %d\n",q,q1,q2,q3);
}
/*search for combinable processes(more than two in a group).*/
a2=0;
for(i=1;i<=f[p];i++)
{
       if(v3[i]==1) goto r2870;
r2670:
       if(c[i]<=0) goto r2870;
       a3=0;
```

```
        c8=0;
        b3=b1[i];
        for(j=i+1;j<=f[p];j++)
        {
            if(v3[j]==1) goto r2850;
            if(c[j]<=0) goto r2850;
            if((b3+b1[j])>b2) goto r2850;
            a3=a3+1;
            b3=b3+b1[j];
            if(a3!=1) goto r2810;
            a2=a2+1;
            c7[a2][1]=i;
            c[i]=c[i]-1;
r2810:
            c7[a2][a3+1]=j;
            c[j]=c[j]-1;
            c8=c8+1;
            if(a3==4) goto r2670;
r2850:;
       }
       if(c8!=0) goto r2670;
r2870:;
}
/*print the combinable groups that can be combined within one C.E.(node).*/
printf("For absolute minimization\n");
printf("The following groups of Processes are\n");
printf("combined in one C.E.(node)\n");
for(i=1;i<=a2;i++)
{
    printf("group %d\n",i);
    for(j=1;j<=5;j++)
    {
        if(c7[i][j]==0) goto r3000;
        w=c7[i][j];
        q=z[w].level;
        q1=z[w].node;
        printf("%d %d\n",q,q1);
    }
r3000:;
}
r3010:;
fclose(fp);
a=a-1;
cpl=z[a].level;
for(i1=1;i1<=a;i1++)
{
      cp1=z[i1].node;
      for(i=i1+1;i<=a;i++)
      {
          cp=z[i].node;
          cpn=(cp1>=cp)?cp1:cp;
          cp1=cpn;
      }
      if(cpn==cp1)
          break;
 }
```

```
  fp1=fopen("labelset","w");
  fp=fopen("timeset","w");
  for(cplevel=0;cplevel<=cpl;cplevel++)
  {
   for(cpnode=0;cpnode<=cpn;cpnode++)
    {
     if(copyvalue[cplevel][cpnode].copies!=0)
      fprintf(fp1,"%d \n",copyvalue[cplevel][cpnode].copies);
     if(copyvalue[cplevel][cpnode].time!=0)
      fprintf(fp,"%f \n",copyvalue[cplevel][cpnode].time);
    }
   }
   fclose(fp1);
   fclose(fp);
}
```

APPENDIX B C Code for Program hdca.c

```
  /* In order to run this program, you need to run copy.c first to get
the copyset, which specify the copies each node needs. Then you have to
get dataset ready,  dataset is in the form of "# of level, # of nodes
from first level to the last level" */
#include <stdio.h>
#include <math.h>
#include <plot.h>
#define MAXLIMIT 11

/*  level: the number of total levels, maximum is 10;
    node[10]: the number of nodes in each level, maximus is 10;
    copy[10][10]: the number of copies for each node; */
int
level,node[10],copy[10][10],initialcopy[10][10],qthreshold[10][10],max_
extracopy[10][10];
int x,y,dec,inc,decval[10][10];
int l,n,cp,copynum,linknum,repitfactor;
char lab[3],lab2[3];
int totaljob,job[10], jobleft=0, nodejob[10][10], queue[10][10];
int varyinrate,inrate[10],speedratio[10],simutime,shut,updateflag;

/* totaljob: total number of jobs need to be processed ;
   jobleft: the number of left jobs. when jobleft=0, the simulation
ends;
   nodejob[10][10]: the number of jobs that has entered each node;
   queue[10][10]: the number of jobs in the queue for each node;
   inrate: input rate, or speed ratio;
   simutime: the simulation time;
   shut: whther need to shutdown some copies; 1: yes, 0: no */
struct link{
   int lf;   /* level of from box */
   int nf;   /* node of from box */
   int lt;   /* level of to box */
   int nt;   /* node of to box */
   float probability;  /* only for when from node is fork */
   }path[100];

struct data{
    int x1; /*bottom midpoint*/
    int y1;
    int x2; /* upper midpoint */
    int y2;
    int qx;
    int qy;
}midpoint[10][10];

struct nodeinfo{
    int fork;  /*1:fork; 0:singular; 2:tailpiece */
    int processtime;  /* the processtime for this node */

}information[10][10];
```

```
struct copyinfo{
    int shutflag[10];  /*1: this copy is shutdown; 0: this copy is not
shutdown*/
    int stopcount[10]; /* the shutdown time, the number should be the
times of processtime */
    int busyflag[10]; /* 1: this copy is busy; 0: this copy is not busy
*/
    int exetime[10]; /* execution time of this copy */
}nodecopy[10][10];

void draw_dataflow(plPlotter *plotter);
void draw_link(plPlotter *plotter);
void arrow(plPlotter *plotter, int x3, int y3);
void simulation(plPlotter *plotter);
void shutoff();
void redraw(plPlotter *plotter);
void update(plPlotter *plotter, int level, int node, int cp);
void draw_queue(plPlotter *plotter, int level, int node);
void clear_one_queue(plPlotter *plotter, int level, int node);
void draw_shut(plPlotter *plotter, int level, int node, int copy);
void clear_shut(plPlotter *plotter, int level, int node, int copy);
void draw_busy(plPlotter *plotter, int level, int node, int copy);
void clear_busy(plPlotter *plotter, int level, int node, int copy);
void draw_extracopy(plPlotter *plotter, int level, int node);
int varate(int speedratio);

main()
{

int i,w,h1,h2,h3,h4,sel;
FILE *fp,*fp1,*fp2,*fopen();

printf("If keyboard entry is desired set KBENTRY to 1,\n");
printf("and for 'read' from file, set KBENTRY to 0\n");
printf("KBENTRY=");
scanf("%d",&sel);

/*******************************************************************/
/* Input the number of levels and the number of nodes in each level */
/*******************************************************************/

if(sel==1)
 {
   printf("Input the number of levels:\n");
   scanf("%d",&level);
 }
else
{
 fp=fopen("dataset","r");
 fscanf(fp,"%d",&level);  /*Level is the total number of levels */
}
printf("NUMBER OF LEVELS =%d\n\n",level);
if (level>=MAXLIMIT)
{
   printf("\nExceeded the limit, excute with a smaller value of
LEVEL\n");
   exit(1);
```

```
}
for(i=1;i<=level;i++)
{
  if(sel==1)
  {
   printf("Enter the number of nodes of level%d:\n",i);
   scanf("%d",&w);
  }
  else fscanf(fp,"%d",&w);
 node[i]=w;
 printf("Level %d has %d nodes\n",i,node[i]);
 }
/**Input the link ***/
for(l=1;l<=100;l++)
{
 if(sel==1)
 {
  printf("Input the link in the form of [from level][from node][to
level][to node]\n");
  printf("Ending the link, enter '0 0 0 0'!\n");
  scanf("%d %d %d %d",&h1,&h2,&h3,&h4);
 }
 else fscanf(fp,"%d %d %d %d",&h1,&h2,&h3,&h4);
if((h1&&h2&&h3&&h4)==0)
   break;
path[l].lf=h1;
path[l].nf=h2;
path[l].lt=h3;
path[l].nt=h4;
path[l].probability=1.0;
linknum=l;
printf("Link:%d%d-->%d%d\n",path[l].lf,path[l].nf,
path[l].lt,path[l].nt);
}
 if(sel==0) fclose(fp);
 printf("Total %d links\n", linknum);
/****************************************************************/
/**************Input the copies for each node ********************/
/****************************************************************/
/*copyset is the result file produced by copy.c , it includes the
number of copies of each node*/
fp=fopen("labelset","r");
for(l=1;l<=level;l++)
{
  for(n=1;n<=node[l];n++)
  {
    if(sel==1)
    {
     printf("Input the number of copies of the processor for
node%d%d:\n",l,n);
     scanf("%d",&w);
    }
    else  fscanf(fp,"%d",&w);
    copy[l][n]=w;
    initialcopy[l][n]=w;
    printf("Node%d%d needs %d copies!\n",l,n,copy[l][n]);
  }
```

```
}
fclose(fp);


/***************** Initialization all the data structure
*********************/
updateflag=1;
for(l=1;l<=level;l++)
  {
    for(n=1;n<=node[l];n++)
    {
     nodejob[l][n]=0;
     queue[l][n]=0;
     qthreshold[l][n]=4;
     max_extracopy[l][n]=4;
     for(i=1;i<=copy[l][n];i++)
     {
       nodecopy[l][n].busyflag[i]=0;
       nodecopy[l][n].shutflag[i]=0;
       nodecopy[l][n].stopcount[i]=0;
       nodecopy[l][n].exetime[i]=0;
     }
    }
  }

/********Graphical Module begins here *************/

//draw_dataflow(plotter);   /* Draw all the nodes, each node represent
a process */
//draw_link(plotter);  /* Draw all the links between nodes */
/*****Graphical module ends here***********/


/***************Simulation Module begins here******************/

/* Read from files the relevent data parameters for simulation */
int word;
float processtime,prob,protemp;
fp=fopen("informationset","r");
fp1=fopen("timeset","r");
fp2=fopen("probabilityset","r");

for(l=1;l<=level;l++)
{
  for(n=1;n<=node[l];n++)
  {
    if(sel==1)
    {
     printf("Whether node[%d][%d] is fork? 1-fork, 0-singular node, 2-
tailpiece.\n",l,n);
     scanf("%d",&word);
     printf("Enter the processtime for node[%d][%d]:\n",l,n);
     scanf("%f",&processtime);
    }
    else
    {
      fscanf(fp,"%d",&word);
```

```
      fscanf(fp1,"%f",&processtime);
    }
    information[l][n].fork=word;
    information[l][n].processtime=(int)(processtime*1000+0.99);
    if(word==1)
    {
      protemp=0;
      for(i=1;i<=linknum;i++)
      {
        if(l==path[i].lf&&n==path[i].nf)
        {
          h1=path[i].lt;
          h2=path[i].nt;
          if(sel==1)
          {
            printf("Input the probability for link node[%d][%d]-
>node[%d][%d]\n",l,n,h1,h2);
            scanf("%f",&prob);
          }
          else   fscanf(fp2,"%f",&prob);
          protemp=protemp+prob;
          path[i].probability=protemp;
          printf("The cumu probability for link node[%d][%d]-
>node[%d][%d]is %f\n",l,n,h1,h2,path[i].probability);
        }
      }
    }
    printf("Information[%d][%d].fork=%d processtime=%d \n",
         l,n,information[l][n].fork,information[l][n].processtime);
  }
}

fclose(fp);
fclose(fp1);
fclose(fp2);

for(i=1;i<=node[1];i++)
{
  printf("Input the total number of jobs at the topnode 1%d:",i);
  scanf("%d",&word);
  job[i]=word;
printf("Job[%d]=%d",i,job[i]);
  totaljob=totaljob+job[i];
  printf("\nEnter the average speedratio of the input jobs at the
topnode1%d:",i);
  scanf("%d",&speedratio[i]);
}
jobleft=totaljob;
printf("\nEnter the total simulation time(in mirco cycles):");
scanf("%d",&simutime);
printf("\nWould you like to shut down any of CE copies?1-yes, 0-no");
scanf("%d",&shut);
printf("If variable input rate is desired, set varyinrate to 1, or
0!\n");
scanf("%d",&varyinrate);
```

```
/******************************************************************/
/***Initialize the plotter****************************************/
/******************************************************************/
plPlotter *plotter;
plPlotterParams *plotter_params;
plotter_params=pl_newplparams();
pl_setplparam(plotter_params,"BITMAPSIZE","750x750");
pl_setplparam(plotter_params,"VANISH_ON_DELETE","no");
pl_setplparam(plotter_params,"USE_DOUBLE_BUFFERING","YES");
pl_setplparam (plotter_params, "BG_COLOR", "white");
pl_setplparam (plotter_params, "FILLTYPE", "0");

/* Create an X plotter with the specified parameters */
if((plotter=pl_newpl_r("X",stdin,stdout,stderr,plotter_params))==NULL)
{
  fprintf(stderr,"Couldn't open Plotter\n");
  return 1;
}
if (pl_openpl_r(plotter)<0)
{
  fprintf(stderr,"Couldn't open Plotter\n");
  return 1;
}
pl_space_r(plotter,0,0,4000,4500);
pl_pencolorname_r(plotter,"red");
pl_linewidth_r(plotter,10);

simulation(plotter);
printf("\ntotaljob=%d jobleft=%d  simutime=%d shut=%d\n",
                            totaljob,jobleft,simutime,shut);
draw_link(plotter);
if(pl_closepl_r(plotter)<0)
{
  fprintf(stderr,"Couldn't close Plotter\n");
  return 1;
}
if(pl_deletepl_r(plotter)<0)
{
  fprintf(stderr,"Couldn't delete Plotter\n");
  return 1;
}

}


void simulation(plPlotter *plotter)
{
  int t,moreshut,flag=0;
  int k,inflag[10];
  for (k=1;k<=10;k++)
  {
    inflag[k]=1;
  }
  if (shut==1)
  shutoff();
  for(t=1;t<=simutime;t++)
  {
```

```
/*************************************************************/
/***********Input a DISV to topnode**************************/
/*************************************************************/
for(n=1;n<=node[1];n++)
{
if(t==inflag[n]&&job[n]>0)
 {
  nodejob[1][n]++;
  job[n]--;
  if(varyinrate==0)
    {inrate[n]=speedratio[n];}
  else
    {inrate[n]=varate(speedratio[n]);}
 printf("Input rate for node[1][%d] is %d:\n",n,inrate[n]);
  inflag[n]=inflag[n]+inrate[n];
 }
 }
 /*************************************************************/
 /*********Check all nodes************************************/
 /*************************************************************/
 for(l=1;l<=level;l++)
 {
   for(n=1;n<=node[l];n++)
   {
   /*************************************************************/
     if(nodejob[l][n]>0)
     {
       flag=0;
       for(cp=1;cp<=copy[l][n];cp++)                    // If there is
a free copy, executue the job
        {
         if(nodecopy[l][n].shutflag[cp]==0)
         {
           if(nodecopy[l][n].busyflag[cp]==0)
           {
              nodecopy[l][n].busyflag[cp]=1;
              nodecopy[l][n].exetime[cp]=0;
              nodejob[l][n]--;
              flag=1;
              break;
           }
         }
       }  //If there is not free copy, put the job in the queue
       if(flag==0) /*No free copy availabel */
       { queue[l][n]++;
             nodejob[l][n]--;
           }
     }  /* nodejob[l][n]>0 ends here!
     /*************************************************************/
     /* Do another loop to check all busy copies */
     for(cp=1;cp<=copy[l][n];cp++)

       {
         if(nodecopy[l][n].shutflag[cp]==1)
         {
```

```
if(nodecopy[l][n].stopcount[cp]!=repitfactor*information[l][n].processt
ime)
               nodecopy[l][n].stopcount[cp]++;
            else
              {
               clear_shut(plotter,l,n,cp);
               nodecopy[l][n].shutflag[cp]=0;
               printf("Node[%d][%d].copy[%d]'s shutdown time is
up!\n",l,n,cp);
               printf("If you need more shutdown, set moreshut=1, or
0.\n");
               scanf("%d",&moreshut);
               if(moreshut==1)
               shutoff();
              }

           }
           else if(nodecopy[l][n].busyflag[cp]==1)
           {

if(nodecopy[l][n].exetime[cp]!=information[l][n].processtime)
                nodecopy[l][n].exetime[cp]++;
             else
               {
                 update(plotter,l,n,cp);
             clear_busy(plotter,l,n,cp);
               }

            }
          }
      /*******************Check the queue********************/
      if(queue[l][n]>0)
      {
        /* If queue length is greater than QTHRESHOLD, start a new copy
of
        computing element. */
        if(queue[l][n]>(qthreshold[l][n]+1))
        {
          if((copy[l][n]-initialcopy[l][n])<max_extracopy[l][n])
          {
            copy[l][n]++;
            cp=copy[l][n];
            nodecopy[l][n].busyflag[cp]=0;
            nodecopy[l][n].shutflag[cp]=0;
            nodecopy[l][n].stopcount[cp]=0;
            nodecopy[l][n].exetime[cp]=0;
            qthreshold[l][n]=qthreshold[l][n]+2;
            updateflag=1;//Redraw the threshold line
          }
        }
        for(cp=1;cp<=copy[l][n];cp++)
        {
          if(nodecopy[l][n].busyflag[cp]==0)
          {
            nodecopy[l][n].busyflag[cp]=1;
            nodecopy[l][n].exetime[cp]=0;
```

```
          clear_one_queue(plotter,l,n);
            queue[l][n]--;
          }
        }
      }
      /*****************Finish checking quequ*************/
      /************Check whether extracopy need to be deactivate****/
      if(copy[l][n]>initialcopy[l][n])
      {
        if(queue[l][n]<qthreshold[l][n])
         {
           cp=copy[l][n];
           if(nodecopy[l][n].busyflag[cp]==0)
              {
                 copy[l][n]--;
                 qthreshold[l][n]=qthreshold[l][n]-2;
                updateflag=1;//Erase the extra copy
              }
          }
      }
     }
   }

/**********************************************************************
/
   /*****Finish checking all level and all nodes, redraw the dataflow
****/

/**********************************************************************
/

   redraw(plotter);

   /*******Check simulation time***************************/
   if(t==simutime&&jobleft>0)
   {
     printf("Simulation time is not sufficient. Set simutime to a
larger value\n");
     break;
   }
   if(jobleft<=0)
   {
     printf("No DISV's at the input. Execution interrupted.\n");
     printf("Change the number of input jobs if desired, and run the
program again!\n");
     printf("Total simulation time is %d micro cycles.\n", t);
     break;
   }

  }
//Finish one time simulation

}

void update(plPlotter *plotter, int l, int n,int cp)
{
```

```
  int word,j,lt,nt,a,b,c,d,e,f,s=dec/5,a1,b1;
  float prob,w;
  word=information[l][n].fork;
  switch(word)
  {
    case 0:  /* Singular Node */
    for(j=1;j<=linknum;j++)
    {
      if(l==path[j].lf&&n==path[j].nf)
       {
         lt=path[j].lt;
         nt=path[j].nt;
         nodejob[lt][nt]++;
         a=midpoint[l][n].x1;
         b=midpoint[l][n].y1;
         e=midpoint[lt][nt].qx;
         f=midpoint[lt][nt].qy;
         if(l<lt)
         {
         pl_line_r(plotter,a,b,a,b-s);
         pl_line_r(plotter,a,b-s,e,f+s);
         pl_line_r(plotter,e,f+s,e,f);
         arrow(plotter,e,f);
         }
         else
         {
          if(e<=a)
            a1=midpoint[lt][nt].x1-500;
          else
            a1=midpoint[lt][nt].x1+500;
          b1=b-s;
          pl_line_r(plotter,a,b,a,b-s);
          pl_line_r(plotter,a,b-s,a1,b1);
          if(e<a){
            pl_line_r(plotter,a1+50,b1+25,a1,b1);
            pl_line_r(plotter,a1+50,b1-25,a1,b1);
           }
           else{
            pl_line_r(plotter,a1-50,b1+25,a1,b1);
            pl_line_r(plotter,a1-50,b1-25,a1,b1);
           }
          pl_line_r(plotter,a1,b1,a1,f+s);
          pl_line_r(plotter,a1,f+s,e,f+s);
          if(e<a){
            pl_line_r(plotter,e-50,f+s+25,e,f+s);
            pl_line_r(plotter,e-50,f+s-25,e,f+s);
          }
           else{
          pl_line_r(plotter,e+50,f+s+25,e,f+s);
          pl_line_r(plotter,e+50,f+s-25,e,f+s);
          }
         pl_line_r(plotter,e,f+s,e,f);
         arrow(plotter,e,f);
         }
         nodecopy[l][n].busyflag[cp]=0;  /*execution is over, set
busyflag to 0 */
         nodecopy[l][n].exetime[cp]=0;
```

```
    }
}
break;

case 1:  /* Fork */
for(j=1;j<=linknum;j++)
{
  if(l==path[j].lf&&n==path[j].nf)
  {
    lt=path[j].lt;
    nt=path[j].nt;
    prob=path[j].probability;
    w=random();
    w=w/1000000000;
    if(prob>=w)
    {
      nodejob[lt][nt]++;
      a=midpoint[l][n].x1;
      b=midpoint[l][n].y1;
      c=midpoint[lt][nt].qx;
      d=midpoint[lt][nt].qy;
        if(lt>l)
        {
          pl_line_r(plotter,a,b,a,b-s);
          pl_line_r(plotter,a,b-s,c,d+s);
          pl_line_r(plotter,c,d+s,c,d);
          arrow(plotter,c,d);
        }
      else
      {
         if(c<=a)
        {
         a1=midpoint[lt][nt].x1-500;
        }
        else
        {
         a1=midpoint[lt][nt].x1+500;
        }
       b1=b-s;
       pl_line_r(plotter,a,b,a,b-s);
       pl_line_r(plotter,a,b-s,a1,b1);
       if(c<a){
       pl_line_r(plotter,a1+50,b1+25,a1,b1);
       pl_line_r(plotter,a1+50,b1-25,a1,b1);
       }
       else{
       pl_line_r(plotter,a1-50,b1+25,a1,b1);
       pl_line_r(plotter,a1-50,b1-25,a1,b1);
       }
       pl_line_r(plotter,a1,b1,a1,d+s);
       pl_line_r(plotter,a1,d+s,c,d+s);
       if(c<a){
       pl_line_r(plotter,c-50,d+s+25,c,d+s);
       pl_line_r(plotter,c-50,d+s-25,c,d+s);
       }
       else{
       pl_line_r(plotter,c+50,d+s+25,c,d+s);
```

```
             pl_line_r(plotter,c+50,d+s-25,c,d+s);
             }
             pl_line_r(plotter,c,d+s,c,d);
             arrow(plotter,c,d);
            }
            nodecopy[l][n].busyflag[cp]=0;  /*execution is over, set
busyflag to 0 */
            nodecopy[l][n].exetime[cp]=0;
            //printf("nodejob[%d][%d]=%d\n",lt,nt,nodejob[lt][nt]);
            break;
          }
        }
     }
     break;

     case 2: /* Tailpiece */
     nodecopy[l][n].busyflag[cp]=0;  /*execution is over, set busyflag
to 0 */
     nodecopy[l][n].exetime[cp]=0;
     jobleft--;
     a=midpoint[l][n].x1;
     b=midpoint[l][n].y1;
     pl_line_r(plotter,a,b,a,b-s-20);
       arrow(plotter,a,b-s-20);
   } //switch ends here!


}



void shutoff( )
{
     int yesorno;
repeat:;
     printf("Input the LEVEL,NODE,and COPY# of which you want to shut
down!");
     printf("\nLevel=");
     scanf("%d",&l);
     printf("\nNode=");
     scanf("%d",&n);
     printf("\ncopy=");
     scanf("%d",&cp);
     nodecopy[l][n].shutflag[cp]=1;
     nodecopy[l][n].stopcount[cp]=0;
     printf("\nSet the interval of shut down time!\n");
     printf("Interval should be the interger time number of the
processtime of this node!\n");
     scanf("%d",&repitfactor);
     printf("Any other copy need to be shut down? 1-Yes, 0=No\n");
     scanf("%d",&yesorno);
     if(yesorno==1)
     goto repeat;

}
```

```
void redraw(plPlotter *plotter)
{
  int s=dec/3;
  int a,b;
  if(updateflag==1)
  {
    pl_erase_r(plotter);
    updateflag=0;
    draw_dataflow(plotter);
  }

  /*******************Draw input data line
****************************/
 for(n=1;n<=node[1];n++)
 {
  if(job[n]>0)
  {
    a=midpoint[1][n].qx;
    b=midpoint[1][n].qy;
    pl_line_r(plotter,a,b+s,a,b);
    arrow(plotter,a,b);
  }
 }
  /****************Draw data link *********************************/
  for(l=1;l<=level;l++)
  {
    for(n=1;n<=node[l];n++)
    {
      if(queue[l][n]>0)
      draw_queue(plotter,l,n);
      for(cp=1;cp<=copy[l][n];cp++)
      {
        if(nodecopy[l][n].shutflag[cp]==1)
        draw_shut(plotter,l,n,cp);
        if(nodecopy[l][n].busyflag [cp]==1)
        draw_busy(plotter,l,n,cp);
      }
    }
  }
}

void draw_shut(plPlotter *plotter,int l, int n, int cp)
{
  int a2,b2,d;
  int x1,x2,y1,y2;
  a2=midpoint[l][n].x2;
  b2=midpoint[l][n].y2;
  d=decval[l][n];
  x1=a2-100;
  x2=a2+100;
  y1=b2-d*(cp-1);
  y2=b2-d*cp;
  pl_line_r(plotter,x1,y1,x2,y2);
  pl_line_r(plotter,x1,y2,x2,y1);

}
```

```
void clear_shut(plPlotter *plotter,int l, int n, int cp)
{
  int a2,b2,d;
  int x1,x2,y1,y2;
  a2=midpoint[l][n].x2;
  b2=midpoint[l][n].y2;
  d=decval[l][n];
  x1=a2-100;
  x2=a2+100;
  y1=b2-d*(cp-1);
  y2=b2-d*cp;
  pl_pencolorname_r(plotter,"white");
  pl_line_r(plotter,x1,y1,x2,y2);
  pl_line_r(plotter,x1,y2,x2,y1);
  pl_pencolorname_r(plotter,"red");
}

void draw_busy(plPlotter *plotter, int l, int n, int cp)
{
  int a2,b2,d,x1,y1;
  a2=midpoint[l][n].x2;
  b2=midpoint[l][n].y2;
  d=decval[l][n];
  if(cp>initialcopy[l][n])
  {
    pl_pencolorname_r(plotter,"green");
    x1=a2-225;
    y1=b2-200+50*(cp-initialcopy[l][n]-1)+25;
    pl_marker_r(plotter,x1,y1,5,10);
    pl_pencolorname_r(plotter,"red");
  }
  else
  {
    x1=a2;
    y1=b2-d/2-(cp-1)*d;
    pl_marker_r(plotter,x1,y1,5,10); /* Marker type 5 is a asterisk; 10
is marker size */
  }
}

void clear_busy(plPlotter *plotter, int l, int n, int cp)
{
  int a2,b2,d,x1,y1;
  a2=midpoint[l][n].x2;
  b2=midpoint[l][n].y2;
  d=decval[l][n];
  pl_pencolorname_r(plotter,"white");
  if(cp>initialcopy[l][n])
  {
    x1=a2-225;
    y1=b2-200+50*(cp-initialcopy[l][n]-1)+25;
    pl_marker_r(plotter,x1,y1,5,10);
  }
  else
  {
    x1=a2;
    y1=b2-d/2-(cp-1)*d;
```

```
    pl_marker_r(plotter,x1,y1,5,10); /* Marker type 5 is a asterisk; 10
is marker size */
  }
  pl_pencolorname_r(plotter,"red");
}

void draw_queue(plPlotter *plotter, int l, int n)
{
  int a2,b2,x1,y1,x2,y2,copies,i;
  copies=queue[l][n];
  a2=midpoint[l][n].x2;
  b2=midpoint[l][n].y2;
  x1=a2+140;
  y1=b2-50;
  for(i=1;i<=copies;i++)
  {
    x2=x1+100;
    y2=y1+25;
    if(i>qthreshold[l][n])
    {
      pl_pencolorname_r(plotter,"green");
      pl_box_r(plotter,x1,y1,x2,y2);
      pl_marker_r(plotter,x1+50,y1+12,4,4);
      pl_pencolorname_r(plotter,"red");
    }
    else
    {
      pl_box_r(plotter,x1,y1,x2,y2);
      pl_marker_r(plotter,x1+50,y1+12,4,4);
    }
    y1=y1+25;
  }
}

void clear_one_queue(plPlotter *plotter, int l, int n)
{
  int a2,b2,x1,y1,x2,y2,copies,i;
  copies=queue[l][n];
  a2=midpoint[l][n].x2;
  b2=midpoint[l][n].y2;
  x1=a2+140;
  y1=b2-50;
  pl_pencolorname_r(plotter,"white");
  x2=x1+100;
  y2=y1+25*copies;
      pl_line_r(plotter,x1,y2,x2,y2);
      pl_marker_r(plotter,x1+50,y2-13,4,4);
      pl_pencolorname_r(plotter,"red");
 }

void draw_dataflow(plPlotter *plotter)
{
 int a1,a2,b1,b2;
 int y1,lab1;
 x=1250;
y=4300;
pl_move_r(plotter,x,y);
```

```
pl_ffontsize_r(plotter,150);
pl_alabel_r(plotter,'l','c',"HDCA DATAFLOW GRAPH");
pl_ffontsize_r(plotter,75);
 y=4000;
 dec=(y/level)-200;
 y=y-(dec/2);

 for(l=1;l<=level;l++)
 {
  x=4000;
  inc=(x/node[l])-200;
  x=100+inc/2;
  for(n=1;n<=node[l];n++)
  {
   pl_box_r(plotter,x,y,x+200,y-200);
   copynum=copy[l][n];
   itoa(copynum,lab);  //copynum was changed after this instruction!
   x=x-120;
   y=y-270;
   pl_move_r(plotter,x,y);
   copynum=copy[l][n];
   if(copynum>initialcopy[l][n])
    {
     pl_pencolorname_r(plotter,"green");
     pl_alabel_r(plotter,'l','c',lab);
     pl_pencolorname_r(plotter,"red");
    }
   else
   pl_alabel_r(plotter,'l','c',lab);
   x=x+120;
   y=y+270;
   copynum=initialcopy[l][n];
   decval[l][n]=200/copynum;
   y1=y;   /*bring the cursor back to original position */
   midpoint[l][n].x1=x+100;
   midpoint[l][n].y1=y-200;
   midpoint[l][n].x2=x+100;
   midpoint[l][n].y2=y;
   a1=midpoint[l][n].x2+140;
   a2=a1+100;
   b1=midpoint[l][n].y2-50;
   b2=b1+200;
   midpoint[l][n].qx=a1+50;
   midpoint[l][n].qy=b2;
   pl_line_r(plotter,a1,b2,a1,b1);
   pl_line_r(plotter,a1,b1,a2,b1);
   pl_line_r(plotter,a2,b1,a2,b2);
/* Label the threshold for each node */
   pl_pencolorname_r(plotter,"green");
   pl_linemod_r(plotter,"dotted");

pl_line_r(plotter,a1,b1+25*qthreshold[l][n],a2+50,b1+25*qthreshold[l][n
]);
   pl_linemod_r(plotter,"solid");
   pl_move_r(plotter,a2+60,b1+25*qthreshold[l][n]);
   pl_alabel_r(plotter,'l','c',"Th[");
   lab1=qthreshold[l][n];
```

```
   itoa(lab1,lab2);
   pl_alabel_r(plotter,'l','c',lab2);
   pl_alabel_r(plotter,'l','c',"]");
   pl_pencolorname_r(plotter,"red");
/* Label the queue name for each queue */
   pl_move_r(plotter,a2+10,b1-10);
   pl_alabel_r(plotter,'l','c',"Q");
   lab1=l;
   itoa(lab1,lab2);
   pl_alabel_r(plotter,'l','c',lab2);
   lab1=n;
   itoa(lab1,lab2);
   pl_alabel_r(plotter,'l','c',lab2);
/* Label then node name for each node */
   pl_pencolorname_r(plotter,"cyan");
   pl_move_r(plotter,midpoint[l][n].x2-300,midpoint[l][n].y2+100);
   pl_alabel_r(plotter,'l','c',"Node:");
    lab1=l;
   itoa(lab1,lab2);
   pl_alabel_r(plotter,'l','c',lab2);
   lab1=n;
   itoa(lab1,lab2);
   pl_alabel_r(plotter,'l','c',lab2);
   pl_pencolorname_r(plotter,"red");

  pl_line_r(plotter,a1+45,b1,a1+45,b1-50);
  pl_line_r(plotter,a1+45,b1-50,a1-40,b1-50);
  pl_line_r(plotter,a1-40,b1-50,a1-10,b1-20);
  pl_line_r(plotter,a1-40,b1-50,a1-10,b1-80);
   for(cp=1;cp<=(copy[l][n]-1);cp++)
   {
    if(cp>(copynum-1))
      draw_extracopy(plotter,l,n);
    else
     {pl_line_r(plotter,x,y1-decval[l][n],x+200,y1-decval[l][n]);/*
draw copies */
      y1=y1-decval[l][n];
     }
   }

   x=x+200+inc;
  }
  y=y-200-dec;
 }
}

// The following program is taken verbatim from page59 if the C
programming Language
reverse(s)
char s[];
{
    int c,i,j;
    for(i=0,j=strlen(s)-1;i<j;i++,j--)
    {
        c=s[i];
        s[i]=s[j];
        s[j]=c;
```

```
    }
}

// The following program is taken verbatim from page60 of The C
Programming Language
itoa(n,s) // convert n to characters in s
char s[];
int n;
{
    int i,sign;
    if ((sign=n)<0) // record sign
         n=-n;
    for(i=0;i<strlen(s);i++)
    {
      s[i]=' ';
    }

    i=0;

    do{   // generate digits in reverse order
        s[i++]=n%10+'0';  // get next digit
    }
    while((n/=10)>0);  //delete it
    reverse(s);
}

void draw_link(plPlotter *plotter)
{
 int i,j,a,b,c,d,e,f,e1,f1,e2,f2,a1,b1,s=dec/5;
 /* Draw lines to the top level boxes */
 for(i=1;i<=node[1];i++)
 {
   a=midpoint[1][i].qx;
   b=midpoint[1][i].qy;
   pl_line_r(plotter,a,b+s,a,b);
   arrow(plotter,a,b);
 }


 for(i=1;i<=node[level];i++)
 {
   a=midpoint[level][i].x1; //bottom midpoint x
   b=midpoint[level][i].y1; //bottom midpoint y
   pl_line_r(plotter,a,b,a,b-s-20);
   arrow(plotter,a,b-s-20);
   for(j=1;j<=linknum;j++)
   {
    e1=path[j].lf;
    f1=path[j].nf;
    e2=path[j].lt;
    f2=path[j].nt;
    if(e1==level&&f1==i)  //Then draw feedback line!
    {
      c=midpoint[e2][f2].qx; //queue upper midpoint x of the
destination node
      d=midpoint[e2][f2].qy; //queue upper midpoint y of the
destination node
```

```
      b1=b-s;
      if(f2<f1) a1=midpoint[e2][f2].x1-500;
      else a1=midpoint[e2][f2].x1+500;
      pl_line_r(plotter,a,b,a,b-s);
      pl_line_r(plotter,a,b-s,a1,b1);
      if(f2<f1)
        {
         pl_line_r(plotter,a1+50,b1+25,a1,b1);
         pl_line_r(plotter,a1+50,b1-25,a1,b1);
        }
      else
        {
         pl_line_r(plotter,a1-50,b1+25,a1,b1);
         pl_line_r(plotter,a1-50,b1-25,a1,b1);
        }
      pl_line_r(plotter,a1,b1,a1,d+s);
      pl_line_r(plotter,a1,d+s,c,d+s);
      if(f2<f1)
      {
        pl_line_r(plotter,c-50,d+s+25,c,d+s);
        pl_line_r(plotter,c-50,d+s-25,c,d+s);
      }
      else
      {
        pl_line_r(plotter,c+50,d+s+25,c,d+s);
        pl_line_r(plotter,c+50,d+s-25,c,d+s);
      }
     pl_line_r(plotter,c,d+s,c,d);
     arrow(plotter,c,d);
    }
  }
}
/* Draw connections to the rest of the boxes */
for(i=1;i<=linknum;i++)
{
 e1=path[i].lf;          /* Level value of "from" box */
 f1=path[i].nf;          /* Node value of "from" box */
 a=midpoint[e1][f1].x1;   /*Bottom x coord. of "from" box */
 b=midpoint[e1][f1].y1;   /*Bottom y coord. of "from" box */
 e2=path[i].lt;          /* Level value of destination box */
 f2=path[i].nt;          /* Node value of desitination box */
 c=midpoint[e2][f2].qx;   /* Upper x coord. of destination box */
 d=midpoint[e2][f2].qy;   /* Upper y coord. of destination box */
 if(e1<e2)
 {
   pl_line_r(plotter,a,b,a,b-s);
   pl_line_r(plotter,a,b-s,c,d+s);
   pl_line_r(plotter,c,d+s,c,d);
   arrow(plotter,c,d);
 }
 else
 {
  if(c<=a)
  {
   a1=midpoint[e2][f2].x1-500;
  }
  else
```

```
    {
      a1=midpoint[e2][f2].x1+500;
    }
    b1=b-s;
    pl_line_r(plotter,a,b,a,b-s);
    pl_line_r(plotter,a,b-s,a1,b1);
    if(c<a)
         {
          pl_line_r(plotter,a1+50,b1+25,a1,b1);
          pl_line_r(plotter,a1+50,b1-25,a1,b1);
         }
       else
         {
          pl_line_r(plotter,a1-50,b1+25,a1,b1);
          pl_line_r(plotter,a1-50,b1-25,a1,b1);
         }
    pl_line_r(plotter,a1,b1,a1,d+s);
    pl_line_r(plotter,a1,d+s,c,d+s);
    if(c<a)
       {
         pl_line_r(plotter,c-50,d+s+25,c,d+s);
         pl_line_r(plotter,c-50,d+s-25,c,d+s);
       }
       else
       {
         pl_line_r(plotter,c+50,d+s+25,c,d+s);
         pl_line_r(plotter,c+50,d+s-25,c,d+s);
       }
    pl_line_r(plotter,c,d+s,c,d);
    arrow(plotter,c,d);
   }
  }
}

void arrow(plPlotter *plotter, int x3, int y3)
{
 pl_line_r(plotter,x3-30,y3+30,x3,y3);
 pl_line_r(plotter,x3+30,y3+30,x3,y3);
 pl_move_r(plotter,x3,y3);
}

int varate(int mean)
{
  int n,drand;
  float rand1,rand2;
  float pi=3.14159;
  rand1=random();
  rand2=random();
  rand1=sin(2*pi*rand1);
  drand=sin(2*pi*rand1)*rand2/100000000;
  n=mean+drand;
  return(n);
}
```

```
void draw_extracopy(plPlotter *plotter, int l, int n)
{
int a1,b1,x1,y1,copies,i;
copies=copy[l][n]-initialcopy[l][n];
a1=midpoint[l][n].x1;
b1=midpoint[l][n].y1;
x1=a1-325;
y1=b1;
pl_pencolorname_r(plotter,"green");
for(i=1;i<=copies;i++)
{
  pl_move_r(plotter,x1,y1);
  pl_box_r(plotter,x1,y1,x1+200,y1+50);
  y1=y1+50;
}
pl_pencolorname_r(plotter,"red");
}
```

REFERENCES

[1] Maturi Suryanarayana Rao, "A Graphic Simulation of a Dynamic Pipeline Computer Architecture", *Master's Thesis*, Department of Electrical Engineering, University of Kentucky, Lexington, KY,1983.

[2] J. Cochran, "Mathematical Modeling and Analysis of a Dynamic Pipeline Computer Architecture", *Master's Thesis*, Department of Electrical Engineering, University of Kentucky, Lexington, KY, August. *1982.*

[3] J. R. Heath, G. D. Broomell, A. Hurt, J. Cochran, and L. Le, "A Dynamic Pipeline Computer Architecture for Data Driven Systems: Final Report", *Digital Engr. Research Rept. 82-1 (Contract No. DASG60-79-C-0052),* Dept. Of Elect. Engr, Univ. of KY. Feburary, 1982.

[4] Heath, J.R, Ramamoorthy, S, Stroud, C.E, and Hurt, A.D, "Modeling,Design, and Performance Analysis of a Parallel Hybrid Data/Command Driven Architecture System and Its Scalable Dynamic Load Balancing Circuit", *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing,* vol. 44, no 1, pp. 22-40, Jan. 1997.

[5] Heath, J.R, and Balasubramanian, S, "Development, Analysis, and Verification of a Parallel Dataflow Computer Architectural Framework and Associated Load-Balancing Strategies and Algorithms via Parallel Simulation", SIMULATION, vol. 69, no.1, pp. 7-25, July.1997.

[6] U. Chameera. R. Fernando, "Modeling, Design, Prototype Synthesis and Experimental Testing of a Dynamic Load Balancing Circuit for a Parallel Hybrid Data/Command driven Architecture", *Master's Project Report,* Department of Electrical Engineering, University of Kentucky, Lexington, KY. 1999.

[7] Xiaohui Zhao, "Hardware Description Language Simulation and Experimental hardware Prototype Validation of a First-Phase Prototype of a Hybrid Data/Command Driven Multiprocessor Architecture", *Master's thesis,* Department of Computer Science, University of Kentucky, Lexington, KY. 2003.

[8] Mahyar R. Malekpour, "(NASA-CR-191545) Simulator for Heterogeneous Dataflow Architectures Report, 1 Jun. 1991- 31 Aug. 1992", Lockheed Engineering & Sciences Company, Hampton, Virginia.

[9] Harry F. Jordan, Gita Alaghband, "Fundamentals of Parallel Processing", Pearson Education, Inc., Upper Saddle River, NJ 07458.

[10] William E. Biles, Susan T. Wilson, "Animated Graphics and Computer Simulation", Industrial Engineering Department, Louisiana State University, Baton Rouge, LA 70803.

[11] Gosselin, CM; Laverdiere, S; Cote, J;"SIMPA: A graphic simulator for the CAD of parallel manipulators", *COMPUT ENG PROC INT COMPUT ENG CONF EXHIB.,* ASME, New York, NY(USA), vol. 1, pp. 465-471, 1992.

[12] Windham, W A; Schrieder, J E; "An animated graphical simulator for multiple switch architectures", *NAECON '97; Proceedings of IEEE 1997 National Aerospace and Electronics Conference,* Dayton, OH; pp. 353-359; 14-17 July 1997.

[13] Jerry Banks, "Software for Simulation", *Proceedings of the 28th conference on Winter Simulation,* Coronado, California, United Stateds. Pages: 31-38.

[14] H. Welch and W. McDonald, "An Example BMD Problem fir Experimentation on Dynamically Reconfigurable Distributed Computing System", *Technical Report No. TM-HU-301/000/01, System Development Corporation, Huntsville, AL, April 1981.*

[15] Website:
http://www.delorie.com/gnu/docs/plotutils/plotutils_toc.html#SEC_Contents

Vita

Chunfang Zheng was born on October 22nd, 1975 in Hebei, China. She attended Suzhou High School in Jiangsu Province and graduated in 1992. She obtained her Bachelor of Science in Communication Engineering Degree in July 1996 from Beijing University of Posts and Telecommunications, Beijing, China. She enrolled in the University of Kentucky's Graduate School in the fall semester of 2002.

Copyright © Chunfang Zheng 2004