

68478

THE DCP:
A VISUAL TOOL FOR PROGRAMMING PARALLEL AND DISTRIBUTED SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALTAN KOÇYİĞİT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 1997

Approval of the Graduate School of Natural and Applied Sciences



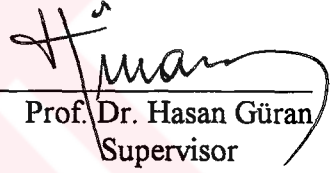
Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.



Prof. Dr. Fatih Canatan
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.



Prof. Dr. Hasan Güran
Supervisor

Examining Committee Members

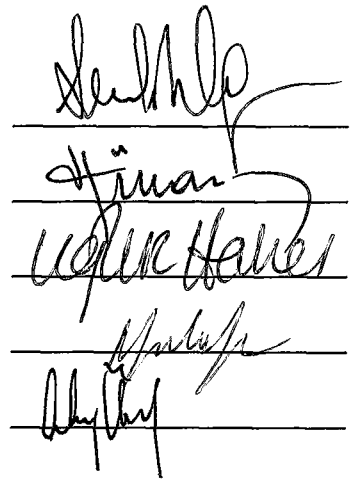
Prof.Dr. Semih Bilgen

Prof. Dr. Hasan Güran.

Prof. Dr. Ugur Halıcı

Assoc. Prof. Dr. M. Mete Bulut

Ms. Sc. Erkan Ünal



ABSTRACT

THE DCPD:A VISUAL TOOL FOR PROGRAMMING PARALLEL AND DISTRIBUTED SYSTEMS

Koçyiğit, Altan

Ms. Sc. Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Hasan Güran

January 1997, 130 pages

This thesis presents a visual programming tool called the DCPD that is used for programming parallel and distributed systems. The DCPD can be used on variety of platforms based on multiple instruction multiple data (MIMD) architectures. It presents a portable programming language for shared and distributed memory types of these machines. It is intended to provide a simple programming interface that is very close to sequential programming. In addition, the DCPD uses C++ as the base language to improve the efficiency and usefulness of the proposed programming system.

Keywords: Parallel and Distributed Programming, Programming Languages, Visual Programming Languages.

ÖZ

DCPP: PARALEL VE DAĞITIK SİSTEMLERİN PROGRAMLANMASI İÇİN BİR GÖRSEL ARAÇ

Koçyiğit, Altan

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Hasan Güran

Ocak 1997, 130 sayfa

Bu tezde paralel ve dağıtık sistemlerin programlaması için kullanılan, DCPP adında bir görsel programlama aracı sunulmuştur. DCPP çoklu komut çoklu veri (MIMD) mimarisine dayanan çeşitli sistemler üzerinde kullanılabilir. DCPP bu tür makinelerin paylaşımlı veya dağıtılmış bellekli çeşitlerinin programlanabilmesi için taşınabilir bir programlama dilidir. Bu dilde, sıralı programlamaya benzer basit bir programlama arabirimi sağlanmaya çalışılmıştır. Bunun yanında, DCPP dili yeterliliği ve kullanılabilirliği geliştirmek amacıyla C++ dilini temel dil olarak kullanmaktadır.

Anahtar Kelimeler: Paralel ve Dağıtılmış Programlama, Programlama Dilleri, Görsel Programlama Dilleri.

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Prof. Dr. Hasan Gran for his guidance and insight. This work would not have been completed without his understanding, and encouragement. I would especially thank to my friends Can, Gl, and Serkan for their help during the preparation of this thesis. I would also like to thank zlem, Deniz, and Yeim for their help on solving problems that arose during this research. Finally, I would like to thank all my friends for their continuous morale support, and encouragment.

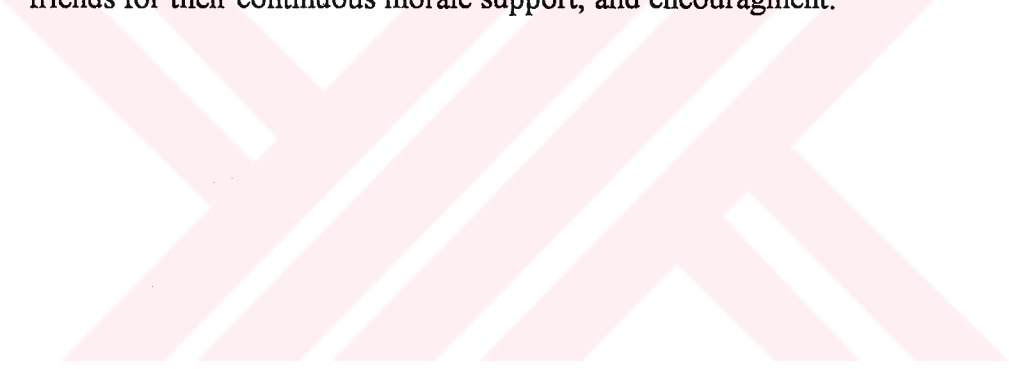


TABLE OF CONTENTS

ABSTRACT.....	iii
ÖZ.....	iv
ACKNOWLEDGEMENTS.....	v
TABLE OF CONTENTS.....	vi
LIST OF TABLES.....	ix
LIST OF FIGURES.....	x
ABBREVIATIONS.....	xii
CHAPTER	
1. INTRODUCTION.....	1
1.1 Overview	2
1.2 Organization of This Thesis	3
2. PARALLEL and DISTRIBUTED SYSTEMS.....	5
2.1 Classification of Computing Systems	6
2.1.1 SISD	6
2.1.2 MISD	6
2.1.3 SIMD	7
2.1.4 MIMD.....	8
2.2 Parallelism Classification.....	10
2.3 Operating System Support	11
2.4 Programming Language Support	13
2.4.1 Automatic Parallelization	13
2.4.2 Sequential Language Plus Extensions.....	13
2.4.3 Shared Memory Programming Languages	14
2.4.4 Data Parallel Programming Languages	14

2.4.5 Functional Programming Languages	15
2.4.6 Logic Programming Languages	15
2.4.7 Object Oriented Programming Languages	15
2.5 Visual Programming Languages	16
3. THE DCPD LANGUAGE.....	21
3.1 Observations	21
3.2 The DCPD Design Guidelines.....	23
3.3 The DCPD Programming.....	25
3.3.1 Graph Model.....	27
3.3.2 Representation of Parallel Programs by DAGs	28
3.3.3 The DCPD Programming Tools.....	48
3.4 The DCPD Rules.....	52
3.5 Program Execution.....	54
4. IMPLEMENTATION	56
4.1 Execution Environment Requirements	57
4.2 Syntax Check.....	58
4.3 The DCPD Output.....	60
4.3.1 Program Module.....	61
4.3.2 Caller Module	63
4.4 Runtime System	65
4.5 Hardware and Operating System Abstraction	69
4.5.1 Thread Class	70
4.5.2 Queue Class.....	73
4.5.3 Pool Class	74
4.5.4 Executive Function.....	75
4.6 Conversion System.....	75
4.6.1 Decision Tools	76
4.6.2 Loop Tools.....	76
4.6.3 Queue Tools.....	76
4.6.4 Block Tools.....	77
5. EXAMPLE PROGRAM.....	79
6. CONCLUSION	83
REFERENCES	87
APPENDICES	
A. RUNTIME SYSTEM FOR NETWORK OF COMPUTERS.....	91

B. RUNTIME SYSTEM FOR WINDOWS NT ON MULTIPROCESSORS.....	98
C. RUNTIME SYSTEM FOR AIX ON MULTIPROCESSORS	105
D. MATRIX MULTIPLICATION SOURCE CODES	112
E. THE DCPD APPLICATION	123



LIST OF TABLES

TABLE

2.1	Classification of Parallel Programs.....	10
3.1	C++ Fundamental Types.....	50



LIST OF FIGURES

FIGURES

2.1 MISD (Pipeline) Computer.....	7
2.2 SIMD (Array) Computer.....	8
2.3 Shared Memory MIMD.....	9
2.4 Distributed Memory MIMD	9
3.1 The DCCP's Graphical User Interface.....	26
3.2 A Simple Graph.....	27
3.3 Representation of a Function in a Directed Graph.....	29
3.4 Graph Representation of a Program That Computes the Correlation Coefficient of two Sequences	32
3.5 Graph Representation of a Simple Program Segment That Use a <i>NULL</i> arc to Synchronize Two Functions	33
3.6 Graph Representation of a Loop Node	35
3.7 Graph Representation of a Simple Program Using Loop to Calculate the Correlation Coefficients of the K Sequence Pairs	37
3.8 Graph Representation of a Decision Node	38
3.9 Graph Representation of a Simple Program That Does Error Checking at Runtime Using a Condition Node	41
3.10 Graph Representation of a Queue Node.....	42
3.11 Graph Representation of a Simple Program To Apply K Different Filters To An Image In Parallel, and Using Queues To Pass The Dependent Portions of Image Between Tasks	44

3.12 Graph Representation of Dynamic Task Creation.....	46
3.13 Graph Representation of a Simple Program Using Dynamic Job Creation to Compute the Product of Two Matrices.....	48
3.14 The DCP's Visual Programming Tools.....	49
4.1 The DCP Programming Overview	56
5.1 Matrix Multiplication $A \times B = C$	80
5.2 Matrix Multiplication Program	81



ABBREVIATIONS

DAG	Directed Acyclic Graph
DCPP	The name of the Language (implying Distributed C++)
DTC	Dynamic Task Creation
GUI	Graphical User Interface
FIFO	First In First Out
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
RPC	Remote Procedure Call
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SPMD	Single Program Multiple Data

CHAPTER 1

INTRODUCTION

Today, computers are widely used on many areas of science and engineering. Every new application requires more processing power than the previous one. Current computer industry provides new, fast computers to meet this increased processing power requirement. It can be thought that, eventually they will be fast enough to satisfy technological growth. However, history showed that, a certain technology meets known applications, as the new applications arise, it is required to develop new technology to treat them[1].

Normally, it is expected that increasing the clock rate will increase the overall performance proportionally. Although, clock rates are increasing continuously, they tend to approach physical limits. For example, a chip with 3 cm diameter propagates a signal in 10^{-9} sec with $3 \cdot 10^7$ m/sec signal transmission speed in silicon (speed of light argument) [2]. This sets a limit for sequential computing power. Reference [3] presents the relationship between the speed and overall performance for a system. The actual performance of a system also depends on several other factors. Fundamental limits on clock rates, system reliability, memory bandwidth, and cost are principal issues on the design of high performance systems[4]. These observations favor the usage of parallel architectures against sequential systems.

Actually, parallel and distributed systems have been a huge research area for many years. The vector processors, massively parallel architectures, were produced and they have been used for special large applications in laboratories and big computing centers. However, they were not available for broad use on industry due to their high prices.

Nowadays, it is possible to face parallel and distributed computing systems on many areas. Parallel and distributed computing are not a laboratory research topic any more. Parallel and distributed computing systems are widely used on many real life applications. It is possible to find several kinds of multiprocessor systems on the market with acceptable prices. Moreover, advances in networking products make it simpler to build a distributed computing environment based on network of workstations with reasonable effort, and price.

We can say that parallel and distributed computing seems to be effective on wide range of applications and systems for the near future. Computer industry will mainly focus on these systems. In addition, tendency towards these systems results in more interest in software tools and programming for parallel and distributed computing.

1.1 Overview

The parallel and distributed programs can be developed in several ways. The first method is developing the application on top of a parallel or a distributed operating system. The second method is using an existing sequential programming language with library routines that provide parallel programming primitives. The last method is using a language that is designed for parallel and/or distributed computing.

This thesis is based on the third method. In this thesis, the DCP (Distributed C++) programming language that is used for programming parallel and

distributed systems is presented. The DCPD is a general purpose programming tool for programming shared or distributed memory MIMD machines.

The DCPD is intended to be an easy to use and flexible tool for development of variety of applications on different platforms. It is a visual development environment for construction and viewing of realistically sized programs. This visual development environment makes the design process simpler and understandable compared to textual programming counterparts. The structure of the program, and interfaces between modules can be defined more clearly, and precisely by using such a visual development environment.

Although the DCPD can easily be ported to variety of platforms, its current implementation supports three platforms. These are the Microsoft NT operating system running on symmetric multiprocessors, AIX version 4.1.4 operating system running on IBM RS/6000 multiprocessor machine, and DOS operating system running on PC's that are connected by a local area network. These target platforms represent two different parallel architectures. One is a shared memory multiprocessor system, the other is a distributed memory MIMD system. All of these environments/platforms are readily available for implementing and testing the programming system.

1.2 Organization of This Thesis

Before giving the introduction to the DCPD language, Chapter 1 presents the need for parallel and distributed programming and brief overview of the language.

Chapter 2 introduces parallel and distributed systems and their classification. In this chapter, hardware and software platforms for these computing systems and currently available programming tools are presented.

Chapter 3 introduces the DCPD programming system. It starts with design guidelines of the DCPD and continues with detailed explanation of programming activity in this environment. Each primitive visual element is defined and its usage is demonstrated by simple program segments.

Chapter 4 describes the output of the DCPD, runtime system, syntax check, and conversion of the programs. The conversion process for each tool and its interface with the runtime system is presented.

Chapter 5 gives an example program written by the DCPD system. It presents a matrix multiplication example with its implementation.

Finally, chapter 6 gives the concluding remarks about the work done in this thesis.



CHAPTER 2

PARALLEL and DISTRIBUTED SYSTEMS

The parallel and distributed computing systems can be defined as the systems in which multiple processing elements are combined together to get more processing power.

Although the main goal for a distributed system and a parallel system is the same, they differ slightly in their definitions. The parallel computing systems are formed by multiple processors that are located within a small distance from each other. These systems have reliable communication between the processors. Their main purpose is to execute a computational task jointly. However, for a distributed system, processors may be far away from each other, and communication between processors is more problematic. Communication links between them may be unreliable, and communication delays may be much higher. In these systems, each processor executes its own task and cooperates with other processors in context of some large computation task[7].

A parallel, or a distributed system can be characterized by its hardware, the operating system running on top of this hardware, and the programming tools for this configuration. This chapter introduces the parallel and distributed systems along these aspects.

2.1 Classification of Computing Systems

The computer systems are first classified by Flynn. Flynn's classification [5] divides the computer systems into four main groups. These are [6]:

1. SISD: Single Instruction Single Data
2. MISD: Multiple Instruction Single Data
3. SIMD: Single Instruction Multiple Data
4. MIMD: Multiple Instruction Multiple Data

In this classification scheme, two of the classes are very important for parallel and distributed processing. Most of the parallel and distributed systems can be categorized by SIMD, and MIMD classes. Besides, it is possible to encounter *Hybrid parallel computer* systems [6]. They are derived from pipeline, MIMD, and SIMD. The multiple-pipeline, multiple-SIMD, and SPMD are examples of these systems.

2.1.1 SISD

The SISD class concerns the single processor von Neumann computer. This represents the well-known sequential machine model. The von Neumann computer model was standardized many years ago, and still a valid model for sequential computers. This computer model consists of a central processing unit, and a storage unit. In this system, the CPU executes a program that specifies the sequence of simple operations on the storage unit.

2.1.2 MISD

The MISD computers are also called as *pipeline* computers (Figure 2.1). This architecture presents a processing model in which single data is processed through multiple processors.

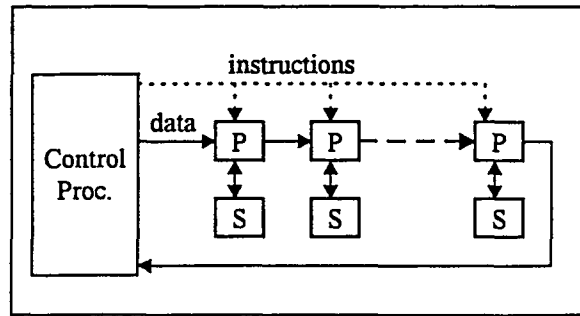


Figure 2.1 MISD (Pipeline) Computer

The main characteristic of MISD architecture is that each processor performs a pre-defined, single operation on the incoming data in the pipeline. By this way, in each clock cycle one set of instructions is finished then the data chain is further shifted in to the pipeline of processors.

2.1.3 SIMD

On SIMD systems, there is a single instruction that is to be executed by all the processors synchronously. The processors do not decode the instruction. The instruction is fetched and decoded by the central control processor. Thus the processing elements only contain the logic for Arithmetic Logic Unit (ALU), local memory and communication interface to other processing elements. This greatly reduces the complexity of the SIMD architectures and programming. The main problem in SIMD architectures is observed when the processing elements require too much communication. Special SIMD architectures are widely used in image processing today where image is split into blocks and each processor performs the same instruction on its region of the image.

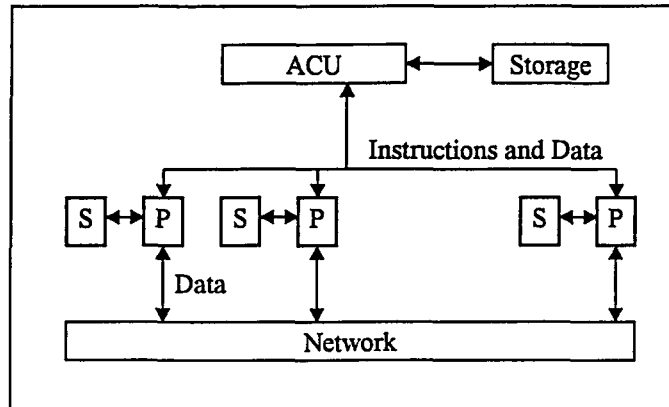


Figure 2.2 SIMD (Array) Computer

We can divide SIMD computers into two categories. The first one has no communication or only a chain communication link between processors. This kind of SIMD computer is also called as *vector computer*. The other type of SIMD computer has an interconnection network between processors. This machine is also called as *array computer* (Figure 2.2).

2.1.4 MIMD

The most complex architecture of the Flynn's classification is MIMD. In this class of computers each processing element has its own control flow. This leads to an asynchronous operation between processing elements. Each processor executes several instructions on several parts of the data.

In accordance with the memory organization of the MIMD architecture this class is divided into two categories:

The first category includes the systems having shared memory (Figure 2.3), and therefore called *tightly coupled*. This is because of the fact that the access order and interaction on shared memory somehow control the execution flow of the processing elements. This class of computers are also

called multiprocessors. The *symmetric multiprocessors* is an example for such systems.

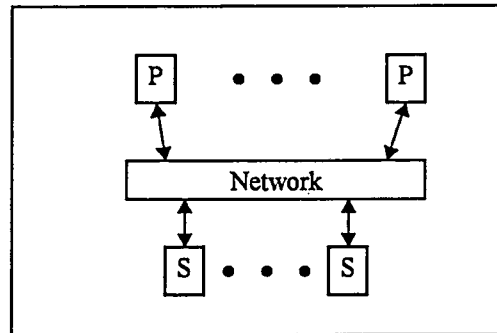


Figure 2.3 Shared Memory MIMD

In the second category, the processing elements have their own local memory (Figure 2.4). Thus the structure is 'loosely coupled' which imposes several difficulties due to the communication overhead between processing elements. Since, the processing elements have their own memory, they can also be called as *multicomputers*. The *network of workstations (or minicomputers)* is a good example for this category.

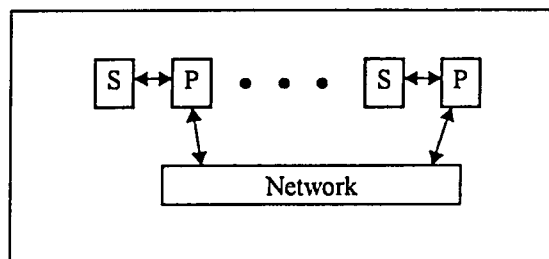


Figure 2.4 Distributed Memory MIMD

2.2 Parallelism Classification

The parallel and distributed systems are classified according to many features of their hardware and software. The fundamental classification of these systems is done according to their hardware architectures (section 2.1). The principal measure of parallelism in a given hardware is the number of processors it has. If a parallel system contains thousands of processors, this system is called as *massively parallel* system. The SIMD systems usually contain large number of processors, and they are the good examples for massively parallel systems. On the contrary, if a parallel system has ten or twenty processors, it is called as multiprocessors, or multicomputers according to their memory systems.

Besides, the parallel programs often classified by the measure of computation time between successive communications [8]. This is called as *grain* of parallelism. *Large-grain (coarse-grain)* parallel programs do computation most of the time, and communicate rarely. On the contrary, the *fine-grain* parallel programs communicate more frequently. The *medium-grain* parallel programs are in between them.

The hardware and software classification of a parallel system are also related to each other. In massively parallel architectures, the fine grain parallelism is suitable. However, in multicomputer systems the coarse grain parallelism is preferred.

Table 2.1 Classification of Parallel Programs

Level	Executed Object	Example System
Program Level	Job, Task	Multitasking Operating System
Procedure Level	Process	MIMD System
Expression Level	Instruction	SIMD System
Bit Level	Within Instruction	von Neumann Computer

In Table 2.1, classification of parallel programs according to their levels of parallelism, are shown. In this table, parallelism level is illustrated for coarse grain to fine grain, from top to bottom respectively. In reference [6], a detailed explanation is given for these levels of parallelism.

2.3 Operating System Support

It is possible to make use of an operating system to develop a parallel or a distributed application. This approach employs a sequential programming language, and the existing operating system primitives to develop a parallel or a distributed application.

Many operating systems provide multitasking capabilities, and basic communication utilities to build communication links between multiple concurrent tasks, or tasks running on multiple processing elements (computers). These tasks can be one of the separate programs, large program segments called *processes* with their own codes, data, and status information, or small processes called *threads* that share a common address space. The UNIX, and Windows NT are the most common examples for such operating systems. By using these characteristics of an operating system, a simple parallel or distributed application that is composed of many concurrent processes running on a shared memory multiprocessor system, or network of computers can be build. These processes also use the existing communication protocols (such as TCP/IP, NETBIOS, etc.), and the related tools (such as pipes) to communicate between each other. This style of programming have been used for a long time, and it is a reasonable programming method to build parallel and distributed applications on many platforms.

In addition to above services, some operating systems provide more specific and useful libraries dedicated to parallel or distributed program development.

The *message passing* [30], and *remote procedure call* (RPC) libraries are the most popular, and widely used examples of such libraries.

The message passing enables the exchange of messages between the processes running concurrently. There are two types of the message passing, synchronous, or asynchronous. In synchronous message passing, sender process stops the execution and waits until the message is delivered to the receiving process. This is a simple and easy to use method for communication. However, this type of communication reduces performance considerably due to waits. Alternative communication choice is the asynchronous message passing. In asynchronous message passing, sender immediately continues its execution after sending the message. By this way communication and executions are overlapped. This communication method is more difficult to use compared to synchronous one, but its performance is much higher.

It is possible to develop a message passing library on top of an operating system that have communication tools. For example, using the UNIX's sockets, or pipes one can develop such applications easily.

Another important communication method is *remote procedure call* (RPC). RPC is a different form of synchronous message passing. In this method, one process sends a message to another process. Then, receiving process gets the message, and process it. Finally, the result is sent back to the sender process. RPC is very similar to the normal procedure call mechanism. It forms a two-way communication mechanism between processes. One of the best example of the operating system support for distributed application development using RPC is the Microsoft's RPC [9].

2.4 Programming Language Support

Implementation of a parallel, or distributed application has many difficulties. However, use of a suitable language for programming could ease the implementation of such applications..

Like sequential programming, there are many parallel languages, and programming tools for application development on different architectures and operating systems. Each of them offers different programming styles, and each of them are suitable for different classes of applications. This section presents an overview of some programming models and corresponding tools or languages.

2.4.1 Automatic Parallelization

Automatic parallelization of a given sequential code is the easiest way to build a parallel application. In this scheme, compiler accept a sequential program, and produce efficient parallel object code without any additional effort of the programmer. However, automatic parallelization is a very hard problem and it seems to be almost impossible for the current compiler technology.

2.4.2 Sequential Language Plus Extensions

This approach uses an existing sequential language and extensions to that language to support parallel programming. These extensions supply explicit control over locality, concurrency, communication, and mapping.

These languages are intended to shield the programmer from both the operating system and the underlying hardware. These programming languages present a higher level, more abstract model for programming to ease the programming task.

Compositional C++, and *FM* [1] are the examples of such languages.

2.4.3 Shared Memory Programming Languages

In this programming style, parallel tasks communicate, and synchronize through shared data. The programming language enables users to define shared memory area, or shared variables. For distributed memory multiprocessors, this approach simulates physical shared memory architectures.

This approach has many advantages over message passing type communication. For example, in message passing systems, a message generally transfers information between two specific tasks, but shared data is accessible over all tasks. Besides, in shared memory programming, the programmer does not need to know the physical distribution of main memory.

Linda [4], and *Tuple Space Smalltalk* [8] are the examples for shared memory programming languages.

2.4.4 Data Parallel Programming Languages

The data parallel parallelism is obtained by applying same operation to some or all elements of a data ensemble. In data parallel programming, the data is distributed over processing elements, and sequence of operations are performed on that data. Therefore, a data-parallel program is formed by a sequence of operations that are to be done on data ensemble.

pC++, *FORTTRAN 90*, *HPF* (High Performance FORTRAN) [6] are the examples for this type of programming languages.

2.4.5 Functional Programming Languages

In this programming model, the functions are like mathematical functions. That is, the result of computation depends on function arguments only. If this condition is satisfied, order of execution of functions makes no difference, and they can be evaluated in parallel. For example, to evaluate the value r :

$r = f(x, y, z)$, where $x = g(l, m)$, and $y = h(n, o, p)$, and $z = i(q)$. Here, l, m, n, o, p , and q represent the values, and f, g, h , and i represent the functions.

the functions g, h , and i can be evaluated in parallel, and in any order then f is evaluated using the results of these functions. This kind of parallelism is well suited for fine-grain applications and corresponding architectures, such as data flow computers.

Concurrent Lisp, and *ParAlfl* [8] are the examples for functional parallelism programming.

2.4.6 Logic Programming Languages

Logic programming is implicit, parallel execution of a logic rule base. Most of the logic programming languages are based on AND/OR parallelism. These kinds of programs are usually communicate through shared logical variables.

Concurrent Prolog [6], and *PARLOG* [8] are the examples of logic programming languages.

2.4.7 Object Oriented Programming Languages

In object oriented programming model, the unit of parallelism is the objects. The objects are assumed to be schedulable entities, and they are distributed

over processors. They may be either passive or active at any instant. Each active object completes its task, and goes into passive state. While the active objects are running, they send messages to other passive objects. Then these passive objects go into active state and start to execute.

POOL, and *Emerald* [8] are the examples of object oriented programming languages.

2.5 Visual Programming Languages

Visual programming is not a new concept. Today, there exists many programming languages based on visual programming. The implementations of these visual programming languages vary from visualization of existing textual languages to inherently visual implementations. References [10], [11], and [12] discuss the visual programming languages, their evaluation, and advantages of these languages.

The visual programming languages are also applied for programming parallel and distributed systems. The visual approach in parallel and distributed processing has many advantages compared to textual counterparts [21, 22]. This is because the visual languages are easy to use and more understandable. In such programming environments, the structure of program, and interfaces between modules are supposed to be more clear, and definite.

Most of the textual parallel programming languages introduce implicit parallelism with complex compilation techniques, or they integrate communication and synchronization with the sequential programming causing large, and complex program structure. These programs are certainly difficult to understand. However, in order to achieve a high performance, the programmer must understand and keep the large structure of the program under control.

The parallel programs are not linear like the sequential ones. A parallel program is multidimensional, because it has multiple concurrently running tasks, and communication among multiple tasks [22]. Therefore, the directed graphs may become a natural means to model the parallel programs. It is possible to combine the directed graph model with the programming language to express the parallelism more clearly. Such languages can present the graphical representation of the program graph to the programmer, as a result, they have an important advantage in parallel programming over textual languages.

The directed graph representation for the parallel programs enables the creation of tasks and communications between these tasks are to be defined more abstractly. The abstraction may bring out the usage of simple and easy to use primitives. Therefore the implementation of the language can cover many of the parallel architectures. In addition, it is possible to implement a language for heterogeneous parallel environments.

Most of the visual languages make use of graph models to represent parallel programs. In these graphs, the nodes represent the sequential program parts, and the arcs represent the data dependencies. In the literature, there are many graph models present such as, Petri nets, form based, process graph, program dependence graphs, etc. These models help to display the structure of the program, and describe the programming activity according to various aspects. Reference [26] classifies and compares these graph models according to their functionality, parallelism, formalism, etc.

Poker [13], PFG [15] STILE [16] , Clara [14], and VERDI [17] can be given as the earliest examples for the visual parallel and distributed programming languages.

The first example is the Poker programming environment that allows visual programming on a special hardware platform: CHIP parallel processor architecture. It is based on grid based specification of a graph that shows communication paths between processes.

The second example is the PFG (Parallel Flow Graphs) that is a language for expression of concurrent, time-dependent computations. It is based on control flow modeling of programs. The control flow diagram is composed of non-overlapping basic blocks, called threads, and sequential function calls in these threads. In this model the threads run parallel with each other. The PFG's execution semantics are defined by timed Petri nets [6] and hierarchical graphs.

The third example is the STILE environment. The STILE is a graphical environment to design and develop logical relationships between components. This approach can be applied to different computation and concurrency methods. STILE programs are composed of boxes that contain sequential computations in a typical programming language. The boxes represent the sequential processes that communicate through the data ports. These ports are used to communicate with other boxes.

The fourth example is the Clara environment. Clara supports Milner's CCS as a specification and design language. The CCS is a tool to define systems in terms of processes communicating through ports. Clara environment provides an ideographic syntax for CCS expressions.

The last example is the VERDI visual environment for distributed system design. In VERDI, designer creates a system control flow diagram of application, then specifies the points of interprocess communication and synchronization. Normal computations (assignments, expressions, etc.) are specified in a standard language (e.g., C, PASCAL).

There are some more recent systems that are based on visual parallel and distributed programming. Some of these are currently under development. The CODE 2.0 [29], HeNCE [23, 24], VPE [25], and PADS [27] are the famous examples for such systems. All of these systems are based on directed graph modeling of the parallel programs. All of the four systems are based on the idea that the nodes represent the computation, and the arcs represent the data dependencies.

CODE 2.0 programs consist of the graphs, and each graph has many nodes, and arcs. Arcs carry data between the nodes, and they behave as FIFO buffers. In addition, it is possible to call a graph from any graph by using a special node called *call node*. The computation nodes get data from the input arcs and process them. Finally the results are pushed to the output arcs. There also special firing rules of the nodes. The firing rules determine the execution start inside a node.

HeNCE is built on top of a distributed programming software package PVM [28] that is used for process management, and communication over heterogeneous network of computers. The HeNCE programs are modeled by directed acyclic graphs (DAG) where the nodes represent the procedures (in C or FORTRAN), and the arcs represent the dependencies. The graphs express the control flow of a program.

VPE is also a heterogeneous programming environment and uses PVM (like HeNCE). It is also be readily implemented on top of other message passing libraries like MPI [30]. VPE is based on the graphs that describe the process structure of the program. The arcs represent the message ports for the processes, and messages flow on these arcs. It is also possible to call the other graphs from a graph to allow hierarchical program development.

PEDS is a visual programming tool, that is used for programming heterogeneous distributed environments. It can utilize various different software packages and integrate them in a visual development environment. The programs are modeled by computational graph model based on a directed graph. The nodes represent the computational units, and the arcs represent inter-node data dependency or communication relationship.



CHAPTER 3

THE DCPD LANGUAGE

This chapter introduces the DCPD programming language. The DCPD is a language that is used for parallel and distributed program development. For this reason, before giving the details of the language, some observations on the parallel and distributed program development are given in the following sections. The DCPD proposes a programming tool that tries to reduce the difficulties presented in those sections.

3.1 Observations

Designing and building parallel and distributed programs is a complex and hard process compared to sequential programs [22, 24, 20]. The complexity comes from the design of a parallel algorithm, and implementing this algorithm on a parallel and distributed platform. The present programming tools are trying to ease the implementation of a designed program. These tools hide the low-level details of the run-time system from the programmer, and offer a hardware and operating system independent interface.

Although the implementation difficulties can be solved by using a good programming tool, there are still problems with the design of the programs. Design of a parallel algorithm is effected by several factors. The most important factor is the selection of a suitable software structure with the

present implementation tool and present hardware. It is sometimes difficult to find the best algorithm for a specific platform and development tool. Besides, if a specific language is used for the implementation, some restrictions of that language and familiarity problems for the programmers will arise. The programmers also suffer from the lack of the ready to use, and tested library routines. Certainly, this will increase the programming time, and program bugs.

Another important point to consider about the parallel programming is to visualize the program as whole. This is even more important for parallel programming, because parallel program writers must check many things, such as, the states of simultaneously running tasks, shared data areas, etc. As program gets larger, the programmer may lose his/her control over the whole program. Sometimes, such a long program with a complicated structure may cause serious programming errors. In many cases, there is no way to detect these errors during programming. Such errors can not be detected until runtime, and produce confusing results. To avoid these errors, the structured programming model, or more simple algorithms can be employed. Since simple algorithms may degrade the performance, and some languages may not be suitable for structured programming, the above situation may be inevitable.

Portability is also an important design issue for parallel and distributed programs. There are many problems associated with writing parallel programs that run on several platforms. One of the main problems in porting an application to a different platform is the implementation language. There may not be an implementation of the development language on the target system. Thus, to run an application on that platform, the program must be rewritten with a new language that is designed for that platform. In addition, if an application is built on top of a parallel or distributed operating system, it may not be possible to find the required system calls on the new target

operating system. So, it may not be possible to transfer the application to a new domain.

Efficient use of available processors is also important for parallel programs. Application writers want to keep all processors as busy as possible. For a fixed partitioned application, it may not be easy to assign a job for each idle processor all the time. Therefore, some processors wait idle most of the time. This results in a reduced parallelism. To overcome this problem, an application should be partitioned to large number of small tasks and they are distributed over the idle processors. However, this may complicate the application unless there is mechanism to handle the execution of large number of tasks.

3.2 The DCPD Design Guidelines

The main goal of the DCPD is to help programmer to write parallel and distributed applications easily and more systematically. It is intended to be an easy to use, and flexible tool for building general purpose applications on a parallel or distributed system. It provides a simple and flexible programming interface for the programmer to make things simpler, and more understandable.

The DCPD is a visual programming language. The visual languages have many advantages on the parallel programming [21]. The DCPD combines most of the benefits of the visual programming with the parallel and distributed programming. The usage of a visual programming environment enables the programmer to visualize the whole program structure easily, and gives a complete control over the designed program. The program structure is always kept under the control of the designer, and serious design errors that are caused by poor program structures are prevented at design time.

The DCPD is designed to present a programming model that is very close to the sequential programming. So that programmers who are working on sequential programs, and inexperienced on parallel computing have no trouble in writing programs by using this system. Therefore, the amount of time that developers spend learning the new environment is tried to be minimized.

Instead of designing a special programming language, the DCPD is designed to use C++ [1] as a base language. The base language selection is an important factor on the usefulness of this system. The C++ language is a very powerful programming language for the development of complicated programs. In addition, its extensive and widespread libraries ease writing sophisticated, and large programs with improved reuse. Many implementations of C++ exist on variety of platforms and they are more or less compatible with each other. The DCPD extends all the advantages of the C++ language to write parallel programs. By using C++, the problems that may appear due to usage of a special language are also prevented. Besides, the programmers, who are using C++ for their programs, can easily adapt to programming with the DCPD.

Another design goal of the DCPD is to provide a hardware and operating system independent interface for the programmer. The DCPD users are not required to know about the details of the underlying platform. They are isolated from platform specific tasks like the distribution of tasks over multiple processors, required operating system calls, communication between processors. This provides creation of a program independently from the low-level programming details and hardware configurations. The design of DCPD system also presents full flexibility to switch the run-time system. With small modifications, the DCPD system can easily be adapted to any MIMD machine that has a C++ compiler (this requirement is very important, but almost all the computing systems have a C++ compiler), without the need to

modify the applications already written on the DCP. This presents a great flexibility to select run-time system for any application. Therefore, we can say that the DCP system is a portable system.

The DCP's another important feature is the dynamic task creation (DTC) which will be explained in section 3.3.2.5. The DTC improves the parallelism and efficient usage of the processors. It is an easy to use and powerful method to handle large number of tasks that require independent computations.

3.3 The DCP Programming

The DCP is a visual programming environment for parallel and distributed application development. It comprises a graphical user interface (GUI) (Figure 3.1), and the run-time systems (see Chapter 4). The programs are developed in this GUI by the visual programming elements, and the textual parts of the programs are written by DCP's built-in editors. Finally these programs are converted to the source files in C++ language by the DCP compiler.

The DCP's output (C++ source files) can be compiled and linked with a suitable run-time system to have executable files. The run-time systems provide a hardware and operating system independent interface for the DCP's output. The platform specific run-time systems are used for each of the target platforms.

Program development steps for the DCP system can be summarized as follows:

1. Control flow diagram (or data flow diagram) of the program is designed.
2. This control flow diagram is implemented in the GUI by the suitable visual tools.

3. The textual parts of the program, and required type declarations are implemented in C++ language by using the built-in editor.
4. The program structure is checked against the DCPD programming rules by using the *check* command of the DCPD's GUI. If there are errors, they are fixed by the programmer.
5. The program is converted to C++ source files by using the *convert* command of the DCPD's GUI.
6. The outputs of the DCPD (the C++ source files) are compiled and linked with a suitable run-time system by a C++ compiler and linker to produce the executable file.

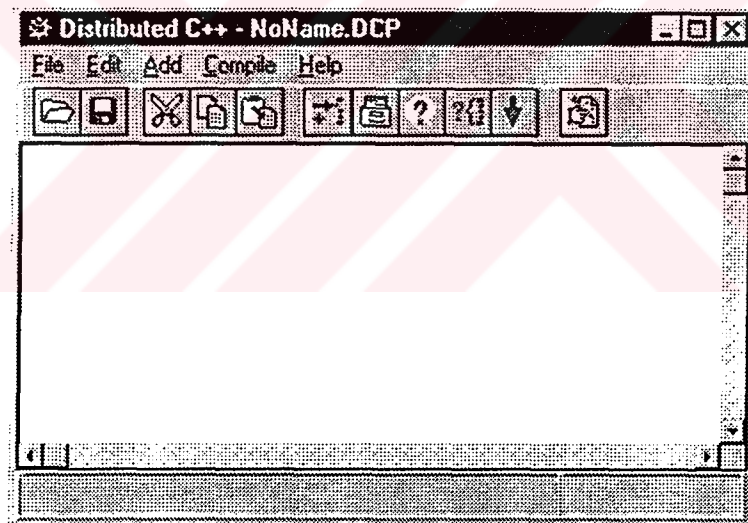


Figure 3.1 The DCPD's Graphical User Interface

3.3.1 Graph Model

Before giving the details of the DCP language a simple graph model to represent parallel algorithms, is given in this section. The remaining sections will refer to this graph model to illustrate the DCP programming concepts.

Let, $G = (N, E)$ be a directed graph, where $N = \{B_1, B_2, \dots, B_{|N|}\}$ is the set of nodes, and $E = \{l(i, j) \mid i = 1 \dots |N|, j = 1 \dots |N|, i \neq j, \text{ and there is an arc from } B_i \text{ to } B_j\}$ is the set of directed arcs connecting these nodes (in this model $|N|$ is used to represent the number of elements in the set N). In Figure 3.2, a simple graph is given with $N = \{B_1, B_2, B_3, B_4, B_5, B_6\}$, and $E = \{l(1, 2), l(1, 3), l(1, 4), l(2, 5), l(3, 4), l(3, 5), l(4, 6), l(5, 6)\}$.

It is said that, the node B_i is a *predecessor* of node B_j , and B_j is a *successor* of B_i , if $B_i, B_j \in N$, and $l(i, j) \in E$. The *in-degree* of a node $B_i \in N$ is the number of the predecessors of that node, and the *out-degree* of a node $B_i \in N$ is the number of the successors of that node. If in-degree of a node is zero, that node is called as a *source node*. If out-degree of a node is zero, that node is called as a *sink node*. For example, in Figure 3.2, B_1 is the source node, and B_6 is the sink node.

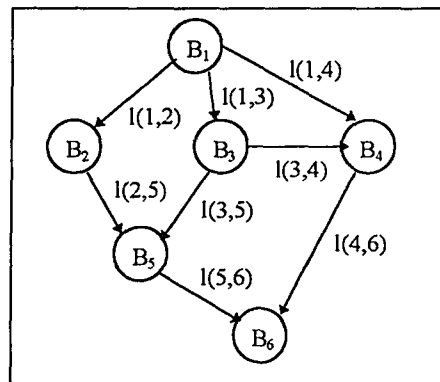


Figure 3.2 A Simple Graph

In this model, each $l(i, j) \in E$ that is leaving the node B_i is also represented by $r(B_i, j)$, and each $l(i, j) \in E$ that is going into the node B_j is represented by $a(B_j, i)$. Therefore, it can easily be seen that $l(i, j) \equiv r(B_i, j) \equiv a(B_j, i)$. By using these definitions, the set of arcs going into the node B_j can be defined as $A_j = \{a(B_j, i) \mid l(i, j) \in E, i=1, \dots, |N|\}$, and the set of arcs leaving the node B_i can be defined as $R_i = \{r(B_i, j) \mid l(i, j) \in E, j=1, \dots, |N|\}$. For example, Figure 3.2, $A_3 = \{a(B_3, 1)\}$, and $R_3 = \{r(B_3, 4), r(B_3, 5)\}$. It can be easily seen that in-degree for a node B_i is equal to $|A_i|$, and out-degree for a node B_i is equal to $|R_i|$. Moreover, if $|A_i|$ is equal to zero then B_i will be a source node, and if $|R_i|$ is equal to zero then B_i will be a sink node.

A *path* is the sequence of nodes $B_{i_0}, B_{i_1}, \dots, B_{i_K}$ of nodes such that $l(i_k, i_{k+1}) \in E$ for $k=0, 1, \dots, K-1$. For example, in Figure 3.2, $B_1 B_3 B_4 B_6$ is a path with length of 3 arcs. If all of the nodes B_{i_k} are distinct nodes in a path, it is said that there are no *cycles* in that path, otherwise the path has at least one cycle. If there is no path that has a cycle, that graph is also called as a *directed acyclic graph* (DAG).

3.3.2 Representation of Parallel Programs by DAGs

The parallel programs are multidimensional objects [22]. Therefore the natural models for the parallel programs are the directed graphs. It is also possible to use the graph model for the implementation of the parallel programs. This section presents a DAG representation model for the parallel programs. This model establishes the basis for the DCP programming. The following sections describe the construction of parallel programs using some special programming tools such as functions, condition expressions, loop expressions, and queues.

3.3.2.1 Functions and the Programs Constructed by Functions

Let a directed graph $G = (N, E)$ be used to represent a parallel program. In this model, each $B_i \in N$ represents a C++ function, and each arc $l(i, j) \in E$ represents the data dependency between the nodes $B_i, B_j \in N$. In particular, an arc $l(i, j) \in E$ indicates that the function represented by the node B_j uses the results of the function represented by the node B_i . According to this model, the set A_i represents the arguments, and the set R_i represents the return values of the function B_i .

Example 3-1:

To clarify the above representation model, a simple function's representation in a directed graph, and its implementation in C++ language is given below. In Figure 3.3, a function, that calculates the mean and variance of a sequence A with probabilities pA , is given in graph representation model.

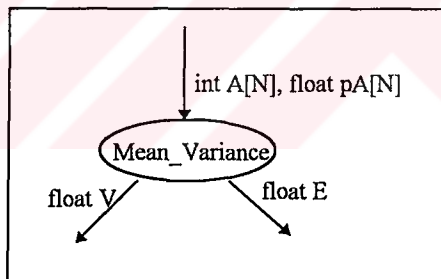


Figure 3.3 Representation of a Function in a Directed Graph

The function *Mean_Variance*, takes the sequence A , and the probabilities of the numbers in pA as the arguments that returned from the predecessor node. Then it calculates the mean E , and variance V of the sequence, and returns

them. The return values of this function are used by the two successors of this node. This function is implemented by C++ language as follows:

```
struct Ret_Mean_Variance{
    float E; // E stands for the mean of the sequence A
    float V; // V stands for the variance of the sequence A
};

Ret_Mean_Variance* Mean_Variance(int A[N], float pA[N])
{
    Ret_Mean_Variance* ret = new Ret_Mean_Variance;
    // first calculate the mean of A as  $E = \sum A_i * pA_i$ 
    ret->E = 0;
    for(int i=0; i<N; i++) ret->E += A[i] * pA[i];
    // then calculate the variance of A as  $V = (\sum A_i^2 * pA_i) - E^2$ 
    ret->V = -1*ret->E*ret->E;
    for(int j=0; j<N; j++) ret->V += A[j] * A[j] * pA[j];
    // finally return the results V, and E
    return ret;
}
```

Note: In the above implementation, a *struct* of type *Ret_Mean_Variance* is declared to specify the return values from the *Mean_Variance* function. This declaration is necessary, because a C++ function can not return more than one return value. Instead of returning values directly, a dynamic storage of type *Ret_Mean_Variance* is allocated, and the pointer to this storage is returned by the function. The successor nodes (functions) will use this pointer, and extract their arguments from the storage, and when the last node gets its argument, the dynamic storage will be de-allocated.

■

The function model described above can be used to represent any parallel program that consists of the functions. To adapt this model to DCP programming some rules should be specified. Lets consider the following rules to represent a program with a directed graph. These preliminary rules ensure the creation of proper program structures for the rest of the programs that will be given.

Preliminary rules:

Let $G=(N,E)$ be the directed graph representation for a parallel program,

1. G must be a directed acyclic graph to prevent deadlocks.
2. In G , there should be only one source node, and one sink node with names' *dmain*, and *dret* respectively. The source node has arguments "*int argc, char *argv[]*" (these are the program arguments of a C++ program). The sink node has a return value "*int ret*" (*ret* is the program's return value).
3. For all nodes $B_i \in N$, all the argument variable names in the set of arcs A_i should be unique.
4. For all nodes $B_i \in N$, the return value variable names in the set of arcs R_i should be unique.

Example 3-2:

In Figure 3.4, a program that is represented by a DAG is given. This program computes the correlation coefficient of two sequences A , and B with the probability densities p_A , p_B , and the joint density p_{AB} . The program simply computes the numbers:

$$\begin{aligned} \text{covariance} &= C = E\{AB\} - E\{A\}E\{B\} = E\{AB\} - E_A E_B \\ \text{correlation coefficient} &= r = \frac{C}{\sigma_A \sigma_B} = \frac{C}{\sqrt{V_A V_B}} \end{aligned}$$

In this program, the *dmain* function prepares the sequences and related probability densities. The *MV_A*, and *MV_B* represent the functions that compute the mean E and variance V of the sequences A , and B . The function *E_AB* computes the $E\{AB\}$ using the $s_A = A$, $s_B = B$, and p_{AB} . The *Covariance* function computes the covariance of the sequences A , and B

using the results of the functions E_AB , MV_A , and MV_B . Finally, the correlation coefficient r is computed at the *dret* function using the return values of the functions *Covariance*, MV_A , and MV_B .

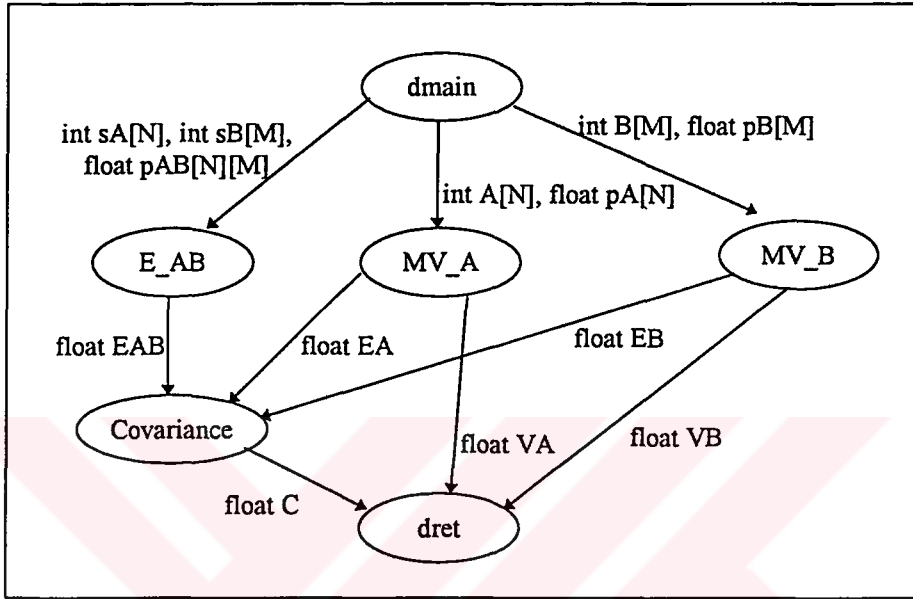


Figure 3.4 Graph Representation of a Program That Computes the Correlation Coefficient of two Sequences

■

Sometimes, a mechanism for synchronizing the executions of some functions required in parallel programs. It is possible to use arcs for this purpose, in this model. The proposed mechanism employs the arcs that carry no arguments or return values. These types of arcs are also called as the *NULL* arcs. If a function is required to run prior to the execution of another function (or a function can only be executed after another function's execution is completed) a *NULL* arc is placed between these functions. In

this case, the function that the *NULL* arc goes in can go into execution if and only if the predecessor function executed completely.

Example 3-3:

In Figure 3.5, a program segment that has a *NULL* arc is shown. For this program segment, *func1* will be executed completely before the *func2* goes into execution.

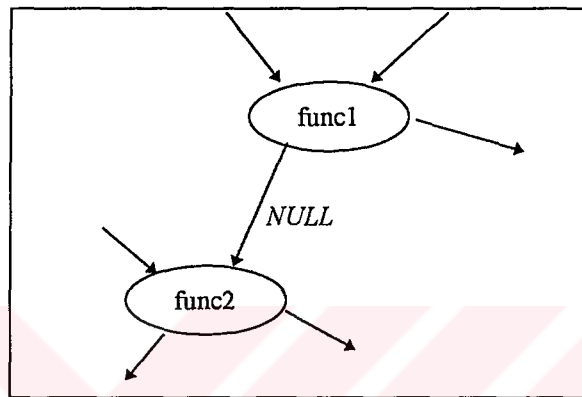


Figure 3.5 Graph Representation of a Simple Program Segment That Use a *NULL* arc to Synchronize Two Functions

■

The last thing to mention about the model described so far is the execution order of the functions. This is called as the *execution mechanism* for the programs. The execution mechanism presents a method to execute the functions with the correct order, and the correct parameters. The execution mechanism for this program model can be expressed as follows:

Let each arc have a state value of *waiting*, *ready*, and *passed*.

1. Set the *NULL* arcs to *passed* state with the argument values of all zero, and remaining arcs to *waiting* state.

2. Pass program arguments to *dmain*, and execute *dmain*.
3. Find a node B_i with all $a(B_i, j) \in A_i$ are in passed state. Set all $r(B_i, j) \in R_i$ to passed state with the values of all zero (to indicate passed state). This means, if a node is only synchronized (no arguments are passed to it), then it is not executed, just the successor nodes are synchronized.
4. Find a node B_i with all $a(B_i, j) \in A_i$ is in ready or passed state, then execute the function by proper arguments. The arguments of the arcs that are in *ready* state will carry the predecessor functions' return values. The arguments of the arcs that are in passed state will carry zeros (to indicate passed state).
5. Find a node B_i that completes the execution. If B_i is *dret* then jump to step 6, else set all $r(B_i, j) \in R_i$ to ready state with the related argument values. Jump to step 3.
6. Execute *dret* with the return values of the predecessor nodes.

To illustrate mechanism described above, consider the program in Example 3-2. The execution of this program starts with the execution of the *dmain* function. After *dmain* function is executed, the arcs leaving it go into the *ready* state, and the functions E_AB , MV_A , and MV_B can be executed in parallel. When all of these functions are executed, all the arguments of the *Covariance* function become *ready* and it is executed. Finally, all the arguments of the *dret* become *ready* and execution completes after the *dret* is executed.

3.3.2.2 Loops

The loops enable a program segment (with one or more nodes) to be executed repeatedly while the condition expression supplied to the loop

statement evaluates to non-zero. Using the loop ensures that the corresponding program segment is executed at least once because the Boolean expression is evaluated after the execution of that segment.

The loop item is represented with a thick bordered circle in the graph model (see Figure 3.6). It has only three arcs to connect it to the other nodes (functions) in the graph. These arcs have special names to distinguish them from the other types of arcs. The arc that is going into the loop node is called as the *condition* link, the thin arc leaving the loop node is called as the *false* link, and the thick arc leaving the loop node is called as the *true* link.

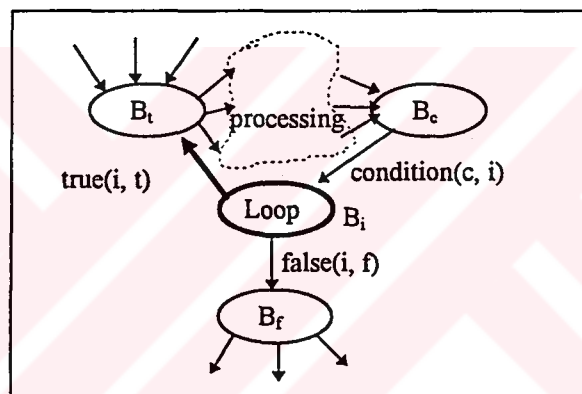


Figure 3.6 Graph Representation of a Loop Node

The implementation of the loop node in C++ language is given as (the following function can also be called as *loop function*):

```
int LoopName(/*arguments from the predecessor node */)
{
    return /* condition expression supplied to the node */
}
```

The execution mechanism for the loop is as follows:

1. At the beginning of the iteration (refer to Figure 3.6), the true link is set to passed state to enable the execution of the B_t node.
2. After B_c node is executed, the condition statement of the loop node is evaluated by calling the loop function, and if the result is non-zero, true link is set to ready state and all other arcs of A_t is set to passed state to start the next iteration.
3. If the condition statement evaluates to zero then only the false link is set to ready state (terminate the iteration and pass results to successor node).
4. if the condition link is in the *passed* state, the false link is set to *passed* state (i.e., if the condition node is synchronized, just the successor node that terminates the loop is synchronized).

This execution mechanism can be combined with the one in the previous section to have the execution mechanism for the programs that has loops.

This system works properly only if all paths starting from B_t pass through the loop node. Otherwise, in the second and further iterations there will be many nodes waiting arguments from their predecessor nodes to go into the execution, and this is not possible. In addition, except the true links of the loop nodes, the whole program must be a directed acyclic graph to preserve previous rules(i.e. If all the loop tools' true links are removed from the whole graph, the remaining graph must be a DAG). As long as the programs obey these rules, it is also possible to have nested loops in a program.

Example 3-4:

In this example, a loop is added to the program in the Example 3-2 to compute correlation coefficients for the K pairs of sequences A_i and B_i . There is some modifications on the previous program to adapt it to the loop

usage. First, the *NextSeq* function is used to supply arguments to *E_AB*, *MV_A*, and *MV_B* functions. The *Calculate* function is used to compute the correlation coefficient for the current sequence pairs and pass arguments to next iterations.

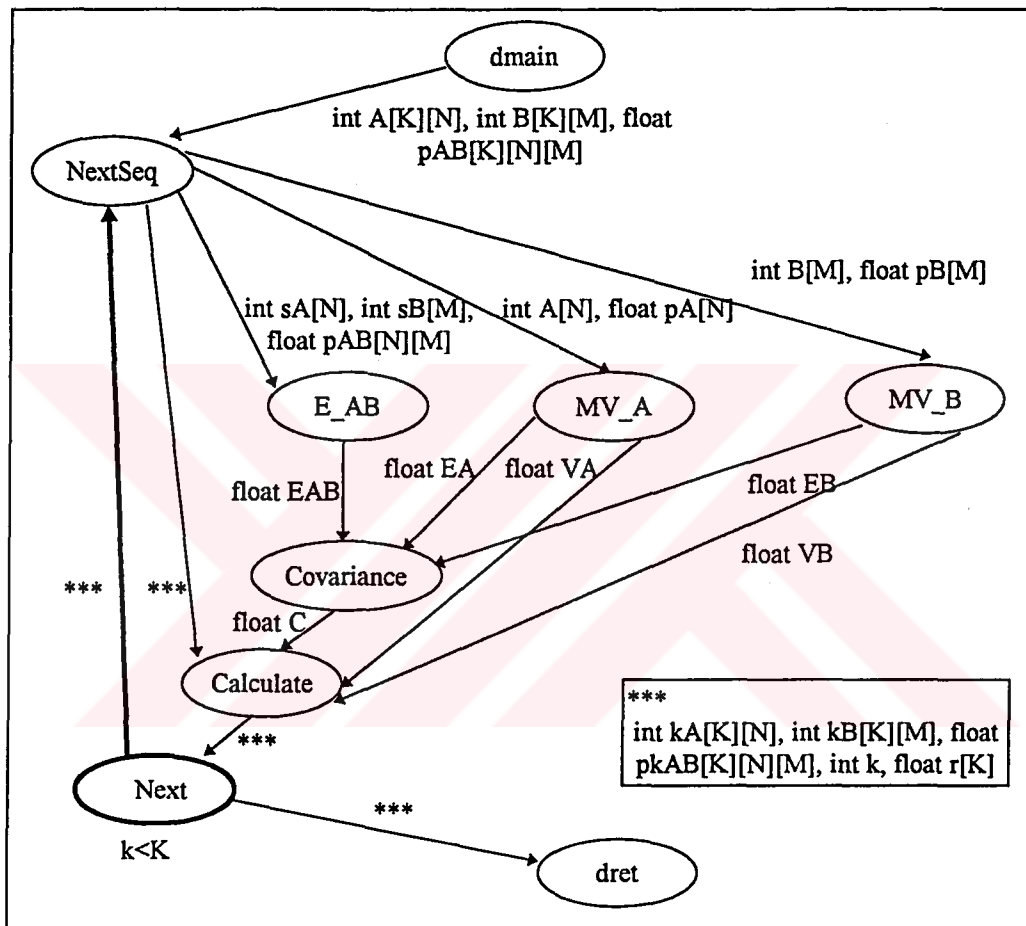


Figure 3.7 Graph Representation of a Simple Program Using Loop to Calculate the Correlation Coefficients of the K Sequence Pairs

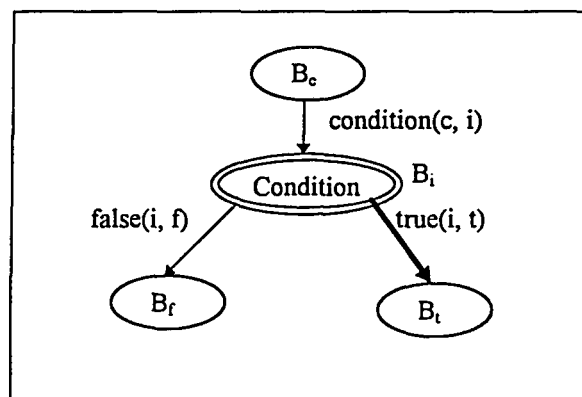
Note: For this example, passing the huge arrays through the nodes may not be an efficient method. However, this program, with this structure can run on

all of the target platforms without taking the care of whether the target system is a shared memory or distributed memory architecture. If performance improvement is necessary, for a shared memory system, global variables for the arrays can be used, and only the pointers to these variables can be passed between the nodes. The same type of improvements can also be done in distributed memory systems, for example one can distribute the arrays over all the processors in some way, and only the array descriptors are passed between the nodes.

■

3.3.2.3 Decisions

The decision nodes implement conditional execution of some portions of a program. It specifies the conditions for the selection of one of the two successor nodes to go into execution. When one of them is selected, the arguments of the condition node are passed to that node, and the other node is just synchronized (like it is connected to the condition node with a *NULL* arc).



**Figure 3.8 Graph Representation of a
Decision Node**

The decision node is represented with a double-line circle in the graph representation (see Figure 3.8). It has three arcs to connect it to the other nodes (functions) in the graph. These arcs have special names to distinguish them from the other types of arcs. The arc that is going into the decision node is called as the *condition* link, the thin arc leaving the decision node is called as the *false* link, and the thick arc leaving the decision node is called as the *true* link.

The implementation of the decision node in C++ language is given as (the following function can also be called as *decision function*):

```
int DecisionName(/*arguments from the predecessor node */)
{
    return /* condition expression supplied to the node */
}
```

The execution mechanism for a decision node is as follows:

1. When the condition link becomes the ready, the condition expression is evaluated by calling the decision function.
2. If the result is non-zero the true link is set to *ready* state and the arguments in the condition link are transferred to true link. The false link is set to *passed* state with all arguments of zeros (to indicate the passed state).
3. If the result is zero the false link is set to *ready* state and the arguments in the condition link are transferred to false link. The true link is set to *passed* state with all arguments of zeros (to indicate the passed state).

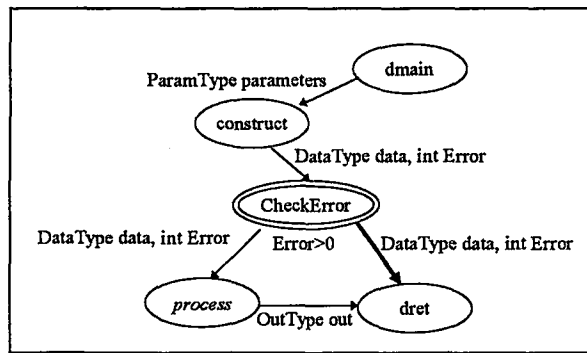
4. If the condition link is in the *passed* state, both of the outgoing arcs are set to *passed* state (i.e., if a condition node is synchronized, all of the successor nodes are synchronized).

This execution mechanism can be combined with the ones in the previous sections to establish the execution method for the programs that has decision nodes.

For the programs in which the decision nodes are used, the whole graph must still be a directed acyclic. As long as this rule is preserved, it is possible to use more than one decision node in a program.

Example 3-5:

In this example, a program that does error checking at run-time is given (see Figure 3.9). The decision node called *CheckError* is placed after the *construct* function to handle the errors that may arise in the execution of *construct* function. According to the value of the *Error* variable the *CheckError* function chooses one of its successors to execute. If *Error* is greater than zero (means some errors are occurred), the true link is set to ready state, and the false link is set to passed state. Therefore, the program terminates after the execution of the *dret* function. If *Error* is less than or equal to zero (this means no error is occurred), the false link is set to *ready* state and *data* is transferred to that link, and true link is set to *passed* state. Then execution continues with the execution of *process* function.



**Figure 3.9 Graph Representation of a
Simple Program That Does Error
Checking at Runtime Using a Condition
Node**

■

3.3.2.4 Queues

According to the model described so far, functions in a program can communicate with each other via the argument or return values. This is a limited method, because the functions can communicate with the other functions only at the start of the execution, and the end of the execution. This is not the only way for communication of functions in DCPD programs. The DCPD has a queue tool that enables the communication of the functions between each other during their execution period. This is an important approach, the program model introduced so far may be awkward, because it may lead to large and complicated programs without some additional communication methods [25].

Queues provide communication links between the functions in a program. The queues provide an abstract communication interface with strongly typed data transfer through it. This is an easy to use and secure method of

interaction of the functions at run-time. In addition, a function can use the queue more than once during its execution period to transfer large data to more than one function.

A queue is *first in first out* (FIFO) container, and it manages a train of objects, where the objects are added to the train from the tail position, and removed from the train from the head position. In this model, each queue is associated with a record type for the items. Different queue nodes can have different record types in a program. However, the record type is unique within each queue.

Every queue can be used by the functions in a program. To use a queue in a function, an arc from the queue to that function must be present. This arc carries the queue identifier to the functions, and functions can operate on the queue by using this identifier.

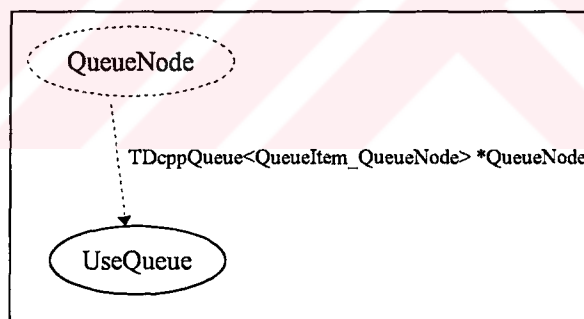


Figure 3.10 Graph Representation of a Queue Node

The queue node is represented with a dashed line circle in the graph representation model (see Figure 3.10). The arcs leaving the queue node are

represented with dashed lines (to identify the arcs carrying the queue identifiers).

The queues are created as the instances of a class template `TDcppQueue` in C++ language. The declaration of this class template is:

```
template <class T>
class TDcppQueue{
private:
    /* private declarations */
public:
    TDcppQueue (); /* constructor of the class */
    ~TDcppQueue(); /* destructor of the class */
    T* Put(T* Item); /* put an item of type T* into the queue, if result is NULL, there is
no empty space left */
    T* Get(); /* get an item of type T* from the queue, if result is NULL , there is no
item yet */
};
```

The record types can be assigned to the queues as it is done for the arcs. For each queue in the program, the record type is declared as:

```
struct QueueItem_QueueName{
// variable declarations
};
```

Finally the queues are created as:

```
TDcppQueue<QueueItem_QueueName> QueueName;
```

The previous rules still hold for the programs that have queue nodes, except one of them. In many cases the queue nodes can behave as source nodes. Therefore that rule must be modified as: There will be only one source node, and one sink node, except the queue nodes in the program.

The queues are not the executable parts of the programs. However, they effect the execution mechanisms of the programs, due to the arcs that connects queues to the functions. Therefore, a simple modification is required in previous execution mechanisms. The arcs leaving the queues

carry queue identifiers to the functions as arguments to that function. Hence, the arcs that connect queues to the functions must be initialized to *passed* state at the program initialization.

Example 3-6:

This example presents a program to apply K square filters of size M*M (M is odd) in sequence to an image of size N*N. This program employs two functions to filter the whole image, and each function filters a half of the image (see Figure 3.11). The filtering operation can be given for a filter as:

$$Image[I][J] = \sum_{i=-(M-1)/2}^{+(M-1)/2} \sum_{j=-(M-1)/2}^{+(M-1)/2} Image[I+i][J+j] * Filter[i+(M-1)/2][j+(M-1)/2]$$

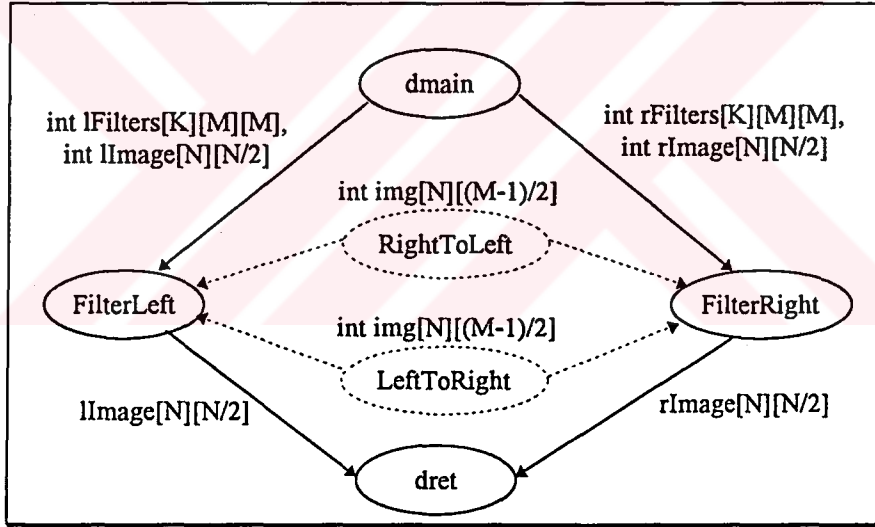


Figure 3.11 Graph Representation of a Simple Program To Apply K Different Filters To An Image In Parallel, and Using Queues To Pass The Dependent Portions of Image Between Tasks

Apparently, to filter a half of an image $(M-1)/2$ columns from the other half is required. Therefore, the functions *FilterLeft* and *FilterRight* exchange their dependent image portions before each filtering process. This exchange is achieved by two queues. These queues are called as *RightToLeft*, and *LeftToRight*. *FilterLeft* function puts the related image portion to *LeftToRight* queue, and *FilterRight* function gets this portion and uses it to filter its own half. *FilterRight* function puts the related image portion to *RightToLeft* queue, and *FilterLeft* function gets this portion and uses it to filter its own half.

■

3.3.2.5 Dynamic Task Creation

The *dynamic task creation (DTC)* is a very important feature of the DCP language. It is an easy to use and effective mechanism to implement dynamic parallelization of the parallel algorithms. The DTC is achieved by a special usage of the queue nodes. The major goal of the DTC is to improve parallelism by the efficient usage of the idle processors at run-time.

The DTC makes it possible to process many of the objects of the same type through a program portion. This program portion can utilize any types of nodes that are described so far as long as the programming rules are preserved. The result of the processing will be an object of many of the different objects (the type of these objects may be different from the initial object's type).

The DTC requires the usage of at least two queues. One of the queues is used to hold the objects to be processed and manages the start of processing. The remaining queues are used to collect the results of the processing. In this method, the objects that are to be processed are accepted as the *tasks*. The queue that holds the tasks that are to be accomplished is called as the *task*

queue, and the remaining queues are called as the *result queues*. The tasks are send to processing whenever there is an idle processor. This behavior ensures that the normal functions have a higher priority than processing of a new task.

The tasks can be placed to the task queue by some functions in the program. This is the standard usage of the queue to add records. The results can be taken from the result queues as the records like the normal queue usage.

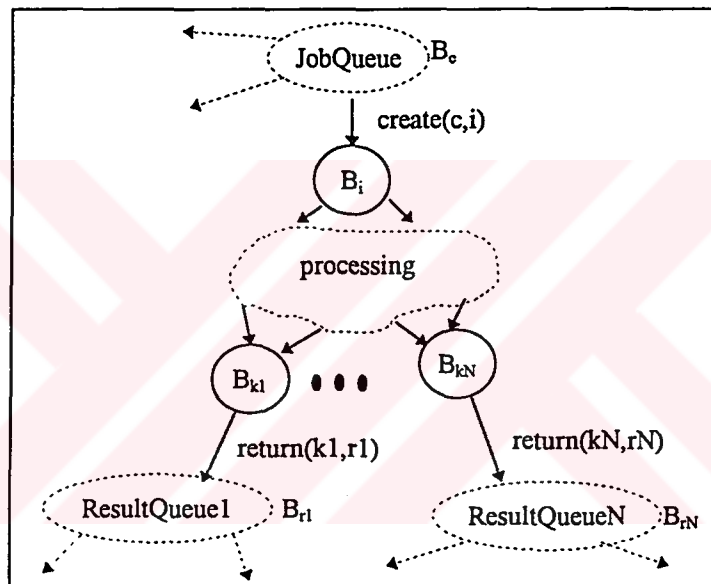


Figure 3.12 Graph Representation of Dynamic Task Creation

Representation for this type of queue usage in the graph model makes use of the arcs with the solid lines that are connected to the queue nodes (see Figure 3.12). The solid arc that is leaving the queue is called as the *create* link, and the solid arc going into the queue is called as the *return* link. There can only be one *create* link, and one *return* link for each queue node. The

node that is successor of the queue takes its arguments from the objects in the queue. To do this the object is removed from the queue, and the object's data are supplied to the function as the arguments. The node that is predecessor of the queue puts its return values to the queue as new objects.

When DTC is used in a program, the whole graph must still be directed acyclic. In addition, this method works properly if and only if all paths starting from a queue node's *create* link pass through another queue node. Otherwise, there will be many nodes waiting to go to execution, and this is not possible. As long as the programs obey this rule, it is also possible to have nested program segments that use the DTC.

Example 3-7:

This example proposes a program to find the product C of two matrices A , and B with the dimensions $N \times M$, and $M \times K$ respectively. The *CreateJobs* function puts the i^{th} row and j^{th} column of A , and B , and i, j values to the *JobQueue* for $i=1..N$, and $j=1..K$. For each item in the *JobQueue*, the *CalculateItem* function is called and this function computes $C[i][j] = \sum a[k][b[k]]$ for those items, and returns result to the *ResultQueue*. At the same time, the *CollectResults* function gets the $C[i][j]$ values from the result queue, and forms the C matrix. Finally, *CollectResults* function returns C matrix to the *dret* function.

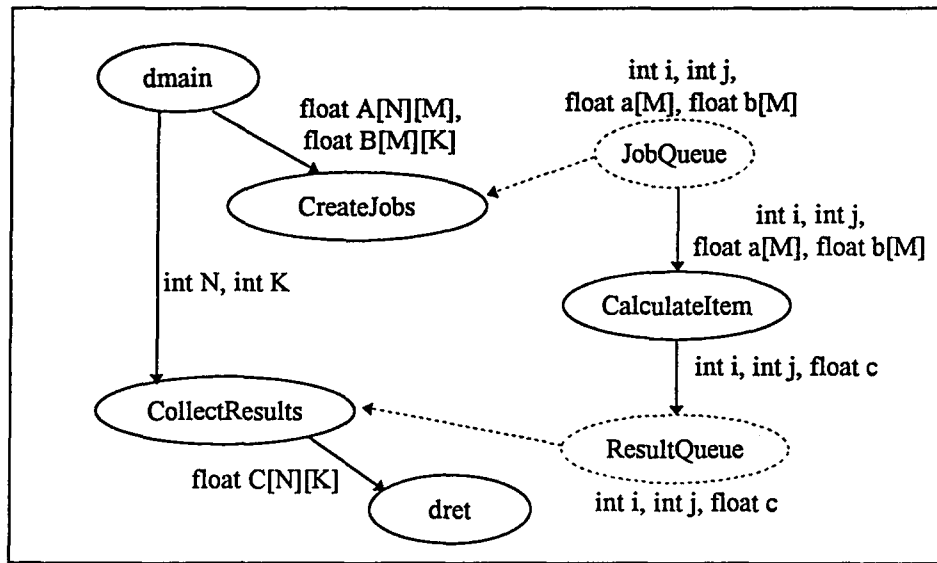


Figure 3.13 Graph Representation of a Simple Program Using Dynamic Job Creation to Compute the Product of Two Matrices

■

3.3.3 The DCPD Programming Tools

The programming model described in previous sections are realized in the DCPD environment by using the DCPD's visual tools. There are five visual tools in the DCPD system. These are the *line*, *block*, *loop*, *decision*, and *queue* tools. In Figure 3.14, these visual tools are presented as they appear in the DCPD environment.

This section describes the visual programming tools of the DCPD language. In the following sections these visual tools and their properties are presented.

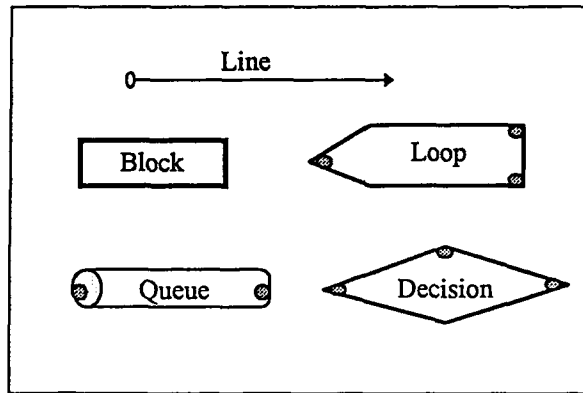


Figure 3.14 The DCP's Visual Programming Tools

3.3.3.1 Lines

The arcs in the graph model are realized by the *Line* (Figure 3.14) tool in the DCP environment. In DCP programs six types of arcs are used. These are the normal (argument or return value carrying), condition, true, false, create, and return arcs. To distinguish the different arc types they are indicated by different colors in the programming environment. The normal arcs have black, the condition, and return arcs have gray, true and create arcs have green, false arcs have red colors.

The normal (except the queue identifier carrying arcs), and condition lines can be associated to some variable declarations. Every line of these kinds can be associated to multiple variables. For variable declarations, the DCP accepts the C++ argument declaration rules for the functions (i.e., the variable declarations are separated by the commas). The format of a declaration can be like:

<type> <argument1>, <type> <argument2>, ..., <type> <argumentN>

The *type* identifiers shown above can be any C++ fundamental type (see Table 3.1), and arrays of that type (at most 9 dimensions are allowed). In addition, the variable names should be valid C++ identifiers. The variable names assigned on the lines are used as argument names and return value names for the functions that are connected by that line. While selecting variable names, the programmer must be careful. Because, argument names and return value names must be unique within each function.

Table 3.1 C++ Fundamental Types

integers of different sizes	char
	short int
	int
	long int
floating point numbers	float
	double
	long double
unsigned integers	unsigned char
	unsigned short int
	unsigned int
	unsigned long int
explicitly signed types	signed char
	signed short int
	signed int
	signed long int

In DCPD environment, the lines can be drawn between the programming tools by means of mouse, and variable declarations can be entered by simply double clicking on the line and editing the line variables.

3.3.3.2 Blocks

The function nodes in the graph model are realized by the *Block* (Figure 3.14) tool in the DCPD environment. The label on the Block tool indicates

the name of that node (function name) in the program. In the DCPD environment, it is possible to draw lines between the blocks and the other nodes, assign or change the block name, and edit the function (using the *Function Editor*).

It is possible to draw line that is leaving the block, or going into the block by clicking with the mouse anywhere on the tool. That is, the block tools do not have special connection points to draw lines.

3.3.3.3 Loops

The loop nodes in the graph model are realized by the *Loop* (Figure 3.14) tool in the DCPD environment. The label on the loop tool indicates the loop identifier in the program. In the DCPD environment, it is possible to draw lines between the loop tool and the blocks, assign or change the loop identifier, and edit the loop's condition expression.

The loop tool has three connection points to the other blocks. It is possible to draw lines to connect the loop tool to any block only at these connection points. These connection points are shown by gray circles on the loop tool in Figure 3.14. The upper right circle stands for the *condition* link, the lower right circle stands for the *false* link, and the left circle stands for the *true* link. These are represented by gray, green, and red circles for condition, true, and false links respectively, in the DCPD environment.

3.3.3.4 Decisions

The decision nodes in the graph model are realized by the *Decision* (Figure 3.14) tool in the DCPD environment. The label on the decision tool indicates the decision identifier in the program. In the DCPD environment, it is possible to draw lines between the decision tool and the blocks, assign or change the decision identifier, and edit the condition expression.

The decision tool has three connection points to the other blocks. It is possible to draw lines to connect the decision tool to any other block only at these connection points. These connection points are shown by gray circles on the loop tool in Figure 3.14. The upper circle stands for the *condition* link, the right circle stands for the *false* link, and the left circle stands for the *true* link. These are represented by gray, green, and red circles for condition, true, and false links respectively, in the DCPD environment.

3.3.3.5 Queues

The queue nodes in the graphs are realized by the *Queue* (Figure 3.14) tool in the DCPD environment. The label on the queue tool shows the queue name in the program. In the DCPD environment, it is possible to draw lines between the queue tool and the other function nodes, assign or change the condition identifier, and edit the condition expression.

It is possible to draw line that is leaving the queue by clicking with the mouse anywhere on the queue tool. These lines shows the successor blocks that will use this queue for communication. Besides, the queue tools also have two special connection points for dynamic task creation. It is possible to draw lines to connect the queue tool to any block at these connection points. These connection points are shown by gray circles on the queue tool (see Figure 3.14). The circle on the right stands for the *return* link, and the circle on the left stands for the *create* link. These lines are represented by gray, and green circles for return, and create links respectively, in the DCPD environment.

3.4 The DCPD Rules

The DCPD programs are developed by putting components together and making connections between these building blocks. To develop a valid DCPD program, there are some rules that must be obeyed. In the previous sections,

the rules for each visual tool were presented separately. This section presents complete set of these rules using the graph representation model given in section 3.3.1.

Let $G=(N, E)$ be the directed graph representation for the program under development. Obtain, $E'=E-\{l(i, j) \mid B_i \text{ is a loop tool, } l(i, j) \equiv \text{true}(i, j)\}$, and $G'=(N, E')$.

The rules are as follows:

1. G' must be a directed acyclic graph.
2. There should be only one source (except queue tools), and one sink node in G , and G' . The source node's name must be '*dmain*', and the sink node's function name must be '*dret*'.
3. For all nodes $B_i \in N$, the node names must be valid C++ identifiers, and they must be unique among all other nodes.
4. For all decision and loop tools a valid C++ expression for the condition must be supplied.
5. For all $B_i \in N$, and B_i is a decision or loop tool, there must be the arcs $\text{condition}(i, j) \equiv l(i, j) \in E$, $\text{true}(k, i) \equiv l(k, i) \in E$, and $\text{false}(t, i) \equiv l(t, i) \in E$ for any $B_j, B_k, B_t \in N$, and all B_j, B_k, B_t 's must be block tools.
6. For all $B_i \in N$, and B_i is a decision or loop tool, $\text{condition}(i, j)$ can not be a null line.
7. For all loop tools $B_i \in N$, and $\text{true}(i, j) \in E$, all paths starting from B_j must pass through B_i in G' .
8. For all queue tools, a queue's object types must be declared.

9. For all queue tools B_i , there must be at least one arc $l(i, j) \in E$ such that $l(i, j) \neq \text{create}(i, j)$, and $B_j \in N$ is a block tool.
10. For all queue tools B_i , and $\text{create}(i, j) \equiv l(i, j) \in E$, all paths in G' , starting with arc $l(i, j)$ must pass through one or more $B_k \in N$ such that $k \neq i$, and B_k is a queue tool.
11. For all queue tools B_i , and $\text{create}(i, j) \equiv l(i, j) \in E$, A_j must be equal to $\{l(i, j)\}$. For all queue tools B_i , and $\text{return}(k, i) \equiv l(k, i) \in E$, R_k must be equal to $\{l(k, i)\}$.

3.5 Program Execution

This section presents the complete definition of the program execution process.

Each line $l(i, j) \in E$ can be in one of three states at run-time. These states are *waiting*, *ready*, and *passed*. The waiting state shows that B_i is not executed or execution of that block is not completed yet. The ready state indicates that block B_i executed already, and all return values are set with B_i 's return values. Finally, the passed state means that block B_i is not executed but execution must continue without the arguments on $l(i, j)$. In this case, all variables of $l(i, j)$ (arguments of B_j) are set to zeros (in passed state array variables' first item is set to zero only).

Execution of a program starts with execution of d_{main} block, and ends with the execution of d_{ret} block. All blocks except these two are executed on other processors than the first processor on which the program started execution. After the execution of d_{main} block, all elements of $R_{d_{\text{main}}}$ are set to ready state, and return values of d_{main} are put to those arcs' variables. Then the execution goes on following way:

1. Find a block B_i with all $l(j, i) \in A_i$ are set to passed state. If not found jump to step 3.
2. For all $l(i, k) \in R_i$, set $l(i, k)$ to passed state except the line $\text{true}(i, k)$ if B_i is a loop tool. Then go to step 1.
3. If there is an idle processor, find a block B_i with all $l(j, i) \in A_i$ are set to passed, or ready state. Otherwise, jump to step 7.
4. If B_i is a block tool, set all arguments of B_i with the values on $l(j, i)$, and start execution of B_i on an idle processor.
5. If B_i is a decision tool, execute condition expression. If result is not zero, pass all values of $l(j, i)$ to $\text{true}(i, t)$ and set $\text{true}(i, t)$ to ready state, set zeros to all variables of $\text{false}(i, f)$ and set $\text{false}(i, f)$ to passed state. If result of condition is zero, set zeros to all variables of $\text{true}(i, t)$ and set $\text{true}(i, t)$ to passed state, pass all values of $l(j, i)$ to $\text{false}(i, f)$ and set $\text{false}(i, f)$ to ready state.
6. If B_i is a loop tool, execute condition expression. If result is not zero, pass all values of $l(j, i)$ to $\text{true}(i, t)$ and set $\text{true}(i, t)$ to ready state, set all other elements of A_i to passed state, and set all variables of those arcs to zeros.
7. If there is an idle processor, and there are queues B_i with $\text{create}(i, c) \in E$, if there is an item in B_i , take the item from the queue and set all variables of $\text{create}(i, c)$ with that item's data, and set $\text{create}(i, c)$ to ready state.
8. If there is a block B_i that is completed the execution already, set all return values of B_i to the elements of R_i , and set all those arcs to ready state. Except if $R_i = \{l(i, q)\}$ and B_q is a queue tool then create a queue item for B_q and fill this item with return values of B_i , then put that item to queue.
9. If dret is not started yet, jump to step 1.

CHAPTER 4

IMPLEMENTATION

The DCPD implementation can be divided into two parts. The first part is the conversion of user programs into C++ language, and the second part is the runtime system that executes threads in that program, and manages the queues.

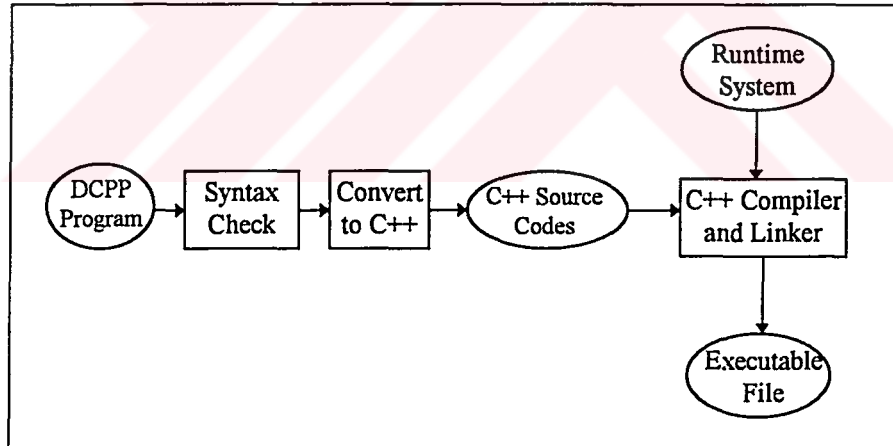


Figure 4.1 The DCPD Programming Overview

Figure 4.1 shows the program development process with the DCPD. In this figure, circles represent the files and boxes represent the processing parts.

This chapter presents these development steps, generated output source code and runtime system.

The syntax check and conversion parts are the same for all execution platforms. However, the runtime system is specific to every target system. This means, by creating a runtime system for any given architecture and operating system that is compatible with the system defined in section 4.1, makes it possible to run DCPD programs on that system. Therefore, it can be said that runtime system isolates hardware and operating system details from the DCPD programs. For this reason, the runtime system is called the *hardware and operating system abstraction module*, and it will be explained in section 4.5

4.1 Execution Environment Requirements

The DCPD is intended to be a portable programming tool on variety of parallel and distributed systems. This section presents basic requirements for execution environments in which the DCPD programs can run. The execution environment expressed here stands for the operating system, and the hardware platform used for program execution.

The DCPD uses C++ as the base language. It produces C++ source codes for the application under development. Therefore, the main requirement is the availability of a C++ compiler on target system. Today, the C++ compilers are available for most of the systems, and it is almost standard throughout different implementations. Therefore, it is possible to fulfill this requirement on most of the systems.

The second important criterion is about the thread execution mechanism. In a shared memory symmetric multiprocessor system, there must be a way to define and run threads simultaneously, and get their states such as running, finished, etc. This requirement is fulfilled by almost all the multiprocessor

systems, and their operating systems. For example, Windows NT operating system, that is running on a multiprocessor architecture, supports threads [18], and some UNIX implementations (like AIX operating system) provide such a multi-threaded program execution system. In a distributed system, this requirement is solved by a remote procedure call (RPC) mechanism. By this way, all the distributed operating systems based on RPC can also run the DCPD programs.

The last requirement is related to communication of multiple threads during program execution. As stated before, in the DCPD system, the queues are used for this purpose. In a shared memory multiprocessor system, it is possible to implement such queues easily. It is also possible to implement, similar queue structure on a distributed memory system by means of available communication tools.

4.2 Syntax Check

Before converting a DCPD program, the syntax of the program must be checked with the rules presented in section 3.4. If the syntax check operation fails, the program can not be converted. The syntax checking operation is done in the following order with the corresponding operations (the following lines use the definitions in section 3.3.1):

- I. Check names of all nodes in the program whether they are valid and unique C++ identifiers, or not (rule 3).
- II. Check all decision and loop tools whether a valid C++ expression for the condition is supplied, or not (rule 4).
- III. Check all decision and loop tools B_i whether they have arcs $\text{condition}(i, j)$, $\text{true}(k, i)$, and $\text{false}(t, i)$, or not. Then, check whether B_j , B_k , B_t 's are the representatives for block tools, or not (rule 5).

- IV. Check all decision and loop tools B_i whether return variables from B_j for $\text{condition}(i,j)$ assigned, or not (rule 6).
- V. Check all queue tools whether a queue item type is declared, or not.
- VI. Check all queue tools B_i , whether there is at least one arc $l(i, j) \in E$ such that $l(i,j) \neq \text{create}(i, j)$, and B_j is a block tool, or not (rule 9).
- VII. Check all queue tools B_i with $\text{create}(i, j)$ is defined whether A_j is equal to $\{l(i, j)\}$, or not. Then check all queue tools B_i with $\text{return}(k, i)$ is defined whether R_k is equal to $\{l(k, i)\}$, or not (rule 11).
- VIII. Check G' , whether it is acyclic or not (rule 1). This check is done as follows:
 - A. Form sets $M = N$, and $V = E'$.
 - B. Find a node $B_i \in M$, and $l(i, j) \notin V$, or $l(j, i) \notin V$ for all $B_j \in N$. If not found go to step D.
 - C. Remove B_i from M , and $l(i, j)$, and $l(j, i)$ from V for all $B_j \in N$. Go to step B.
 - D. If $M = \emptyset$ then G' is acyclic, else G' has cycles.
- IX. Check whether there is only one source (except queue tools), and one sink node in G , and G' , or not. If true check whether source node's function name is '*dmain*', and the sink node's function name is '*dret*', or not (rule 2).
- X. Check all loop tools B_i with $\text{true}(i, j)$ whether all paths starting from B_j pass through B_i in G' , and not pass through any queues, or not (rule 7). This check is done as follows:

- A. Let $B_c = B_j$, and use G' .
 - B. If B_c is the block dret, or a queue then check fails.
 - C. If $R_c = \{ \text{condition}(c, i) \}$ return okay.
 - D. For all successors B_k of B_c follow steps b through d with $B_c = B_k$. If all of them returned okay, then return okay.
- XI. Check all queue tools B_i with $\text{create}(i, j)$ is defined whether all paths in G' , starting with arc $l(i, j)$ pass through one or more $B_k \in N$ such that $k \neq i$, and B_k is a representative for a queue tool, or not (rule 10). This check is done as follows:
- A. Let $B_c = B_j$, and use G' .
 - B. If B_c is the block dret then check fails.
 - C. If $R_c = \{ \text{return}(c, q) \}$, and $B_q \neq B_i$ is a queue return okay.
 - D. For all successors B_k of B_c follow steps b through d with $B_c = B_k$. If all of them returned okay, then return okay.

If any of the check steps above fails, the DCPD will stop the check process, and report the error to the user.

4.3 The DCPD Output

The conversion system produces two C++ modules corresponding to program under development, *caller module* and *program module*. The caller module is named as *CALLER.CPP* (and *CALLER.H* as header file), and the same for every DCPD program. The program module takes its name from the program name as *<Program Name>.CPP* (and *<Program Name>.H* as

header file). This section describes these modules briefly. The detailed description for the caller module contents will be given in section 4.6.

4.3.1 Program Module

Program module contains the functions and required type definitions for all the block, decision, loop, and queue tools. This module contains the block, decision, and loop functions. Actually, the program source file contains all the source lines written by the programmer. Therefore, if any error is detected on converted source codes, by C++ compiler, this module can easily be edited to correct that error without reconverting the program.

The header file of this module contains the required type definitions and the prototypes of the functions that are located in the program source file. Every block tools' argument and return variables, and every queue tool's item record types are defined as records in the header file. These records are used by program source file, and other modules.

The header file starts with including the header file of the runtime system module. The queue record-type definitions for every queue tool follow this include directive. Every queue record's type name looks like `QueueItem_<queue name>`, and all other modules assume this naming convention. For example record type for a queue tool named `queue1` is defined as:

```
struct QueueItem_queue1{
    // queue1's Item Variables
};
```

Then, the argument and return record types for blocks are defined. The names of these record types look like `Arg_<function name>`, and `Ret_<function name>` for argument and return record types respectively. All

the other modules assume this naming convention. For example, argument and return record types for a block named as func1 are defined as:

```
struct Arg_func1{  
    // Argument Variables  
};
```

```
struct Ret_func1{  
    // Return Variables  
};
```

Finally, the function prototypes for the functions that are declared in program source file are put in header file, to enable access from other modules.

The program source file starts with including the header file. Then, global definitions and include files that are written by the user are put in program source file. The function declarations for the blocks follow these definitions. For example, a block named as func1 is transferred as:

```
Ret_func1* func1(/* arguments of func1 */)   
{  
    Ret_func1* ret = new Ret_func1;  
  
    // Function Body of func1  
  
    return ret;  
}
```

The declarations for the condition functions of loops and decision follow the block functions. These functions are used to evaluate the corresponding condition by other modules. In these functions, loop and decision tools' names are used as function names. These functions return an integer value for the condition expression's result. Following lines show an example for the declaration of condition functions corresponding to loop1, and decision1.

```
int loop1(/*arguments of loop1*/)   
{  
    return (/* Condition Expression of loop1 */);  
}
```

```

int decision1(/*arguments of cond1*/)
{
    return (/* Condition Expression of decision1 */);
}

```

4.3.2 Caller Module

The caller module includes the definitions, and functions required to execute the threads, and manage queues in the program.

The header file (caller.h) starts with the *#include* directive for the program module header file to access functions, and type definitions in program module. Then, it includes the runtime module header file to access type definitions, and global variable declarations. Then, the macro identifiers define distinct identification number macros for the blocks. The identification macro names look like *ID_<block name>*. For example for a block named as *func1*'s identification number declaration with number equal to 12 is:

```
#define ID_func1 12U
```

The caller function prototypes follow identification number definitions. For every block in the program, one caller function is assigned. The caller functions provide standard interface for calling the block's function with one of its argument and returning the return values of that function in its other argument. For example, the caller function prototype for a block named as *func1* looks like:

```
void Call_func1(char *arg, int argsize, char *ret, int retsize);
```

The external global variable definitions for every queue in the program follow caller function prototypes. By this way, all queue's in the program can be accessed over all modules (the actual declarations for queue variables are located in caller source file). In DCP, queues are defined as classes. All

queue's in the program, are the instances of a template class *TDcppQueue* (section 4.5.2) with its item record. For a queue named as queue1, external definition looks like:

```
extern TDcppQueue<QueueItem_q1> *q1;
```

For the dynamic task creation queues (queues with create link is connected to a block), a function creating a thread with an item in the queue is prototyped afterwards. If there is an item in the queue, this function creates a thread instance by assigning that item as an argument to thread function. A pointer to thread instance of type *TDcppThread* (section 4.5.1) returned by this function. The runtime system, if there are idle processors, periodically calls these functions. The returned threads are executed by the runtime system. An example function prototype for a queue named as queue1 is declared as:

```
TDcppThread* QueueThread_q1();
```

The last thing located in the header file is class declarations of threads for all blocks. The thread classes are derived from the abstract base class *TDcppThread*. All block specific parameters are passed in the constructor. The member functions *ReturnThread*, and *PassThread* define the operations done when the block is executed and return values are get, and when all the argument lines are set to passed state are. More information about the thread classes can be found in section 4.5.1. Here is an example class declaration for a block named func1:

```
class Thread_func1 : public TDcppThread{
private:
    static unsigned SeqNo;
public:
    Arg_func1 arg;
    Ret_func1 ret;
    Thread_func1(unsigned aSeqNo, int InLoop=0)
        : TDcppThread(ID_func1, (InLoop ? aSeqNo: SeqNo++), 1,
```

```

        Call_func1, (char*)&arg, sizeof(arg), (char*)&ret, sizeof(ret))
    {
        // queue arguments initialization, and loop true line initialization
    }
    virtual char ReturnThread();
    virtual char PassThread();
};

```

The program source file (caller.cpp) for this module implements the caller functions and the class member functions for the blocks.

4.4 Runtime System

The DCP's runtime system is provided as a C++ module called *DCPP.CPP* (and *DCPP.H* as header file). This module prepares types, functions, and variables that are used for thread execution, and queue management mechanism. The converted source code uses these types, variables to generate threads and queues corresponding to every block, and queue tool in the program. After converting user program into C++ source, this module and produced source codes are compiled, and linked together by a C++ compiler, and linker to get an executable program.

The implementation of runtime system has to be different for every specific platform. However, it is possible to classify runtime systems into two different groups. The first group is the runtime systems for the shared memory multiprocessor systems, and the second group is the runtime systems for the distributed memory multicomputers. Although, it is possible to implement the runtime systems in various ways, in this thesis, three runtime systems corresponding to these two groups are introduced (one of them for distributed memory and two of them for shared memory MIMD).

The distributed memory multicomputers approach makes use of a parallel programming library designed for network of PC's, running DOS operating system [19]. All DCP programs running on this system can also use the

tools provided in this library. The implementation for this runtime system is given in Appendix A.

The first example for shared memory multiprocessors approach makes use of the Windows NT operating system, and the Borland C++ v4.5 multithread libraries. The Win32 [18] kernel of Windows NT operating system provides multithread execution system for a single, or multiple processor hardware. It is possible to run the DCPD programs in this environment with a single, or multiple processor hardware. However, the performance of the DCPD programs directly related to number of processors. For single processor systems, it is possible to use Windows 95 operating system, with its Win32 subsystem. However, this system only runs the programs, it does not provide an increase on performance. The implementation for this runtime system is given in Appendix B.

The second example for shared memory multiprocessors approach makes use of the AIX version 4.1.4 operating system and its *pthread* library. This operating system provides multithread execution system for a single, or multiple processor hardware. It is possible to run the DCPD programs in this environment with a single, or multiple processor hardware. Again, the performance of the DCPD programs directly related to number of processors. The implementation for this runtime system is given in Appendix C.

In the shared memory multiprocessor runtime systems, all blocks are executed as threads. The execution of user program starts with the execution of dmain block, and ends with the execution of dret block. All blocks are executed by calling corresponding threads.

In the distributed memory multicomputer runtime systems, all blocks are executed by making remote procedure calls to idle machines. The execution of user program starts with the execution of dmain block, and ends with the

execution of dret block on the master computer. Depending on the implementation of runtime system, the master computer can be the first computer on which the program starts execution, or the computer that is specified by the user. All the other blocks are executed by making remote procedure calls to other idle computers.

For both of the runtime system types, operation is more or less the same. At the startup, the runtime system starts a special thread called *executive* (section 4.5.4). This thread runs throughout the program execution period, and responsible for starting and returning the other threads.

In runtime system, all the threads, and queues are represented by thread objects. The thread object types are derived from an abstract base class named *TDcppThread* (section 4.5.1), and named as *Thread_<block name>* (section 4.6.4). A thread object, corresponding to a block, knows how to call the caller function on other machines, how return values are distributed over the successor blocks, etc.

The runtime system considers all the threads as an instance of *TDcppThread* type, and does not know anything about the implementation specific issues for the individual threads. The runtime system uses two specific storage objects to hold threads. These storage objects are the instances to *TDcppPool* (section 4.5.3) class, and named as *WaitPool*, and *ExecPool*. The *WaitPool* is used for holding threads that has some arguments unavailable. The *ExecPool* is used to hold the threads that are currently running on other processors.

When a thread in the *ExecPool* completes its execution, its *ReturnThread* function is called. This function is responsible for distributing the return values of its block to other threads in the *WaitPool*. If a thread that is

successor of current thread is not found in WaitPool, it is created and added to WaitPool by the ReturnThread function.

If all the argument lines of a thread in the WaitPool becomes passed state, its PassThread method is called and the thread object is removed from the WaitPool. The PassThread function sets all the related arguments of its successors to passed state. If a thread that is successor of current thread is not found in WaitPool, it is created and added to WaitPool by the PassThread function.

If all the argument lines of a thread in the WaitPool becomes ready, or passed state, and all of them are not passed state, its Start method is called. If start method does not return an error, thread object is removed from the WaitPool, and added to ExecPool. The Start method is responsible for making a remote procedure, or thread call to its related caller function.

For the dynamic job creation queues, the QueueThread_<queue name> is called to create a thread using an item from the queue. If there is an item in the queue, this function gets the item from the queue, and creates a thread object for the corresponding block. Then, it sets the arguments of the thread with the item data. Finally, it returns this thread. This thread is added to WaitPool by this function, and in the next pass it is started. In the runtime system, these functions are called only when there is no ready to start item in the WaitPool. This approach gives higher priority to threads created by ordinary ways. Because, the completion of such threads is more important than creating a dynamic thread.

Executive function is responsible for doing the above operations in the runtime system. The executive thread runs on master processor, throughout the program execution, and manages the threads created by other threads, or queues. For this reason, it can be thought as a scheduler for programs.

4.5 Hardware and Operating System Abstraction

The runtime system described in section 4.4 provides an execution system independent interface for the DCPD programs. Therefore, it is called the hardware and operating system abstraction module. This system is implemented as a C++ module named as *DCPD.CPP* (and the header file is *DCPD.H*).

The header file for this module starts with a caller file type definition. This type is used to declare variables for the caller functions inside thread class. The type definition looks like:

```
typedef void (*ThreadFunc)(char *, int , char *, int);
```

Then, the external variables for WaitPool, and ExecPool is declared. This declaration enables the usage of these pools by other modules, and these are declared in the run-time system source file as global variables. The declaration of external declaration looks like:

```
extern TDcppPool WaitPool;  
extern TDcppPool ExecPool;
```

The prototype for executive function is declared after external pool variables in header file as:

```
void Executive();
```

The next declarations are related with the queue types. First a function type for thread creation functions for queues is defined. Then an external array variable for the list of dynamic job creation queues is declared. This array is defined in the caller module with the related functions in the program. This external definition makes it possible to access these functions in runtime system. Finally, the function prototypes for creating and destroying queues

that are implemented in caller module are declared to access them in runtime system.

```
typedef TDCppThread* (*QueueFunc)();  
  
extern QueueFunc ThreadQueues[];  
  
char CreateQueues(int Master);  
  
void DestroyQueues();
```

In addition, the `TDcppQueue` (section 4.5.2), `TDcppThread` (section 4.5.1), and `TDcppPool` (section 4.5.3) class types are declared in the header file.

The `DCPP.CPP` file implements the class methods and standard functions for the runtime system including the main function. The main function initializes the runtime system, creates queues, and calls the Executive function (the initialization process also includes the runtime system related parameters).

4.5.1 Thread Class

The thread class which is called `TDcppThread` is the most important part of the runtime system. This class provides execution system independent interface for the threads. It is an abstract base class, and thread classes for every block in the program are derived from it. The class definition looks like as:

```
class TDcppThread: /* public TThread */ {  
public:  
    enum ArgState { Ready, Passed, Waiting };  
    TDcppThread(unsigned anId, unsigned aSeq, unsigned aCount,  
                ThreadFunc aFunc, char *aArg, int aArgSz, char *aRet, int aRetSz);  
    ~TDcppThread();  
    int IsThread(unsigned anId, unsigned aSeq)  
        { return (Id==anId) && (Seq==aSeq); }  
    unsigned long Start();  
    /* unsigned long Run(); */  
    virtual char ReturnThread() = 0;  
    virtual char PassThread() = 0;  
    int IsCompleted();
```

```

    ArgState GetArgState();
    ArgState& operator[](unsigned index)
        { return ArgStat[index]; }
    void SetAllPassed();
private:
    unsigned Id;
    unsigned ArgCount;
protected:
    unsigned Seq;
    ArgState *ArgStat;
    ThreadFunc Func;
    char *Arg;
    int ArgSize;
    char *Ret;
    int RetSize;
};

```

The data fields of this class are explained in the following lines:

- Id: Identification number to indicate the corresponding block.
- Func: Function variable pointing to the caller function for the block.
- Arg: Address of arguments structure (included in derived class), which is used to call the caller function.
- Ret: Address of return variables structure (included in derived class), which is used to access caller function's return variable.
- ArgSize: Size of argument structure, used to call the caller function.
- RetSize: Size of return structure, used to call the caller function.
- Func: Pointer to the caller function of the block. Used by Start method to call the caller function.
- ArgCount: Number of argument lines, going into the block.
- ArgStat: Array of size ArgCount. Indicates the states of argument lines.

- Seq: Sequence number. The sequence number with Id is used by the IsThread method to identify threads in the Pools. The sequence numbers are used to distinguish between objects that has the same identification number. This condition happens when the queues are used for creating threads. If a thread is created by a queue, this number must be set to a distinct value, and all the successors of the block must use the new sequence number.

The methods can be explained as follows:

- TDcppThread: Constructs the class with the arguments for data fields.
- ~TDcppThread: Destroys the class.
- IsThread: Returns non-zero if aSeq, and anId is equal to Seq, and Id field values. This method is used by ReturnThread, and PassThread methods of other object instances for searching successor blocks in the WaitPool.
- GetArgState: This method scans the ArgStat array. If all members of this array is set to Passed state, it returns Passed, else if all members of the array is set to Ready, or Passed, it returns Ready, otherwise, it returns Waiting. This method is used by the Executive function to decide starting, or passing the threads in the WaitPool.
- operator[]: This operator is used to access an item in the ArgState array. This operator is used to set the line state, when an argument of the thread is set.
- SetAllPassed: This method sets all items of ArgState array to Passed state.

- All derived classes must define this class as a public base class, and override the ReturnThread, and PassThread methods. The data fields in this class are filled by the constructors of the derived classes.
- The Start method defines how the threads are called on remote processors. In Windows NT implementation, this class is derived from the TThread class of Borland C++ v4.52's class libraries, and this class defines the interface between operating system kernel, and TDcppThread class for running threads. In this implementation the Start method is implemented by the base class TThread, and calls the abstract function Run(). Therefore, in this implementation, the Run method calls the caller function for the thread. In network of computers implementation, Start method makes a remote procedure call to caller function to a remote computer.
- IsCompleted: This method returns the state of running thread. If thread execution is finished, this method returns non-zero, else it returns zero to indicate the thread is already running.

4.5.2 Queue Class

The queue class defines the interface to a multiple access queue over the multiple processors, and the type name is TDcppQueue. This queue can be implemented by the operating system, or hardware tools for a parallel or distributed execution system. It is designed as a template class to cover different item types for every queue in the program. The declaration of this class looks like:

```
template <class T>
class TDcppQueue{
private:
    char *Name;
public:
    TDcppQueue (int N, char* aName);
```

```

    TDcppQueue<T>::TDcppQueue(char* aName);
    ~TDcppQueue();
    int Put(T* Item);
    T* Get();
};

```

The constructors of this class define the creation style for the class. The first constructor is used to create the multiple access queue of N items on the execution environment. The second constructor is used to connect to a created queue. The name field is used to identify the queue on the system. In a network of computers implementation, the first constructor is called on the master computer to create the queue instance with the name specified in field Name. The client computers call the second constructor to access the queue that is created on the master computer. The Name field is used for this purpose to identify the queue to be accessed.

The Put, and Get methods define the access methods for the queues. They work with pointers to items of type T, specified in instance declaration.

4.5.3 Pool Class

The pool class defines the WaitPool, and ExecPool storage constructs. It holds the pointers to items of TDcppThread type, and its name is TDcppPool. The pool class is standard for any runtime system, and declared as:

```

class TDcppPool{
private:
    TDcppThread** Pool;
    unsigned Count;
    unsigned Max;
    unsigned Delta;
    int Enlarge();
public:
    TDcppPool(unsigned aMax, unsigned aDelta);
    ~TDcppPool();
    int IsEmpty() { return Count == 0; }
    int AddThread(TDcppThread* aThread);
    void RemoveThread(TDcppThread* aThread);
}

```

```

    TDcppThread* FindThread(unsigned anId, unsigned aSeq);
    TDcppThread* FindFinished();
    TDcppThread* FindReadyOrPassed();
};

```

The methods of this class can be explained as:

- **TDcppPool:** Creates a pool of size aMax. The aDelta value is used for the define how many items are added, if the pool is required to grow.
- **~TDcppPool:** Destroys the pool, and frees all dynamic allocations.
- **AddThread, RemoveThread:** These methods are used to add, and remove threads to, or from the pool.
- **FindThread:** This method is used to find a thread in the pool with given parameters. This method is used in WaitPool instance.
- **FindFinished:** This method is used to find a thread that is completed its execution, in the ExecPool instance.
- **FindReadyOrPassed:** This method is used to find a thread whose arguments are ready or passed, in the WaitPool instance.

4.5.4 Executive Function

This function executes throughout the program execution cycle. Executive function manages the threads in the WaitPool, and ExecPool, and creates threads of dynamic task creation queues. It continuously polls these pools, and queues. Its operation strategy was defined in section 4.4.

4.6 Conversion System

The conversion system of the DCPD converts a program into C++. This system produces the program, and caller modules. It prepares all definitions,

and declarations for all decision, loop, queue, and block tools that are used in the program.

4.6.1 Decision Tools

For every decision tool in the program, a function, evaluating the condition expression of the decision tool, is constructed. The implementation process was presented in section 4.3.1.

4.6.2 Loop Tools

For every loop tool in the program, a function, evaluating the condition expression of the loop tool, is constructed. The implementation process has presented in section 4.3.1.

4.6.3 Queue Tools

The definitions and declarations for the queues that are used in the program are described in program and caller module sections. One thing to note here is the thread creation functions for queues used for dynamic task creation. For all queue tools used for this purpose in the program has a thread creation function. This function looks like:

```
TDcppThread* QueueThread_q1()
{
    QueueItem_q1* Item = q1->Get();
    if(!Item) return NULL;
    Thread_x* T_x;
    T_x = new Thread_x(0);
    memcpy(&(T_x->arg), Item, sizeof(*Item));
    delete Item;
    (*T_x)[0] = TDcppThread::Ready;
    return T_x;
}
```

The operation of this function is simple. First the Get function is called to get an item from the queue. If there is no item in the queue, function returns

NULL, else a thread of specified type in the program, is created and arguments of the thread are set by the item's data. Then the ArgState for the argument line is set to ready state. Finally, this thread is returned by the function.

4.6.4 Block Tools

In program and the caller module sections the declarations and definitions done for each block tool has given. The caller function, and thread class type for the blocks will be explained in this section, in addition to previous explanations.

The caller function for a block defines a standard interface to all blocks for calling the block function. For a block named func1, looks like:

```
void Call_func1(char *arg, int argsize, char *ret, int retsize)
{
    Arg_func1 *Arg;
    Ret_func1 *Ret;
    (void *)Arg = (void *)arg;
    Ret = func1(/* arguments of func1 */);
    memcpy(ret, Ret, retsize);
    delete Ret;
}
```

To call a block function, the arguments and return values are passed in arg, and ret arguments of caller function. In this function, the arguments are resolved from the arg buffer, and they are passed to block function. Then the block function is called with those arguments. The only exception is the queues. The queues are passed from the global variables of them (local variables are used, because, passing the pointers of the processor where the function called, is meaningless in distributed memory systems).

For every block in the program, a thread class that is derived from the TDcppThread is declared. This class implements the specific features of each block. The ReturnThread, and PassThread functions are overridden in this

class. These functions are responsible to set arguments and argument status items for the successor block threads. If a decision or loop tool is encountered as a successor of the block, the corresponding condition function is executed and the true and false lines are handled accordingly. If a *NULL* line is encountered, the corresponding argument status item is set to passed state.



CHAPTER 5

EXAMPLE PROGRAM

Before giving concluding remarks on the DCP, an example program written in DCP is presented in this chapter.

The example given here considers the problem of developing a parallel program to compute $C=AxB$, where A , B , and C are dense matrices. A dense matrix is a matrix in which most of the matrix elements are non-zero.

In the matrix multiplication process, each element of C is calculated as $C_{ij}=\sum_k A_{ik}B_{kj}$. That means, calculation of each element of C requires N multiplications, and $N-1$ additions. Therefore, it can be seen that this matrix multiplication involves $O(N^3)$ operations. If this computation is done sequentially, the time required to compute C will be proportional to this $O(N^3)$ operations.

The implementation of parallel algorithm with DCP will decompose the computation of C over multiple computers to get higher throughput. Basically, the computation is divided to computation of one row of the C matrix. The computation of a row n is done by calling a thread with argument n . By this way, the threads are executed on multiple processors, and computation is done faster than the sequential one.

To do this computation on multiple processors, the matrices A, and B are copied to each processor, and each thread computes a row of C matrix, and returns this row. Figure 5.1 illustrates this operation. The dark gray row of C is computed by using dark gray row of A and light gray columns of B.

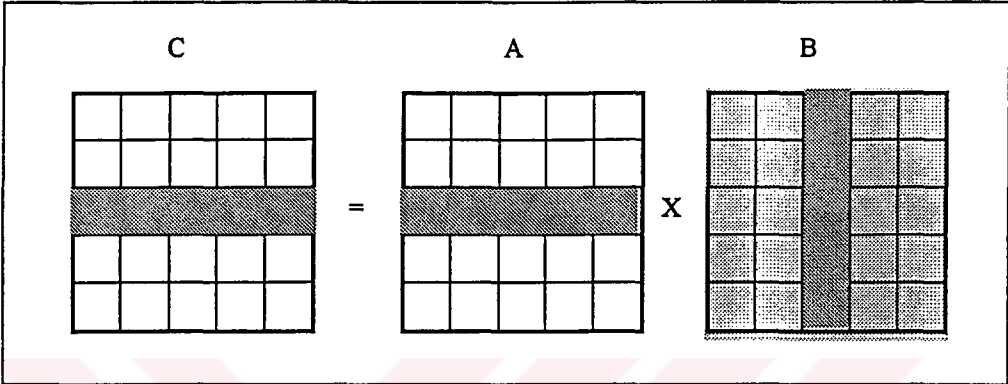


Figure 5.1 Matrix Multiplication $A \times B = C$.

The DCPD implementation of this computation makes use of dynamic job creation queues. A thread in the program puts all row numbers of C to a queue, and gets results from the other queue. For each item in the create queue, system calls the compute row thread and the row returned by this thread is put into return queue. Finally, the thread that puts all the row numbers into create queue, gets the rows of C from the return queue. The program reads the matrices A, and B from the files and writes result matrix C to output file.

The matrix multiplication program is shown in Figure 5.2. The program gets the names of the files having A, and B matrices, and the output file on which the C matrix will be written, from program arguments `argv[1]`, `argv[2]`, and `argv[3]` correspondingly. The *dmain* block checks the program arguments, and reads the matrices A, and B from the corresponding files. If all

arguments are supplied and matrices A, and B are read successfully, it returns start=1, and out=argv[3]. Otherwise, start is set to 0. Then, the decision tool *valid*, checks start and if start is nonzero it passes control to *ComputeAB* block. If start is set to zero, valid passes control to dret block, and completes program execution.

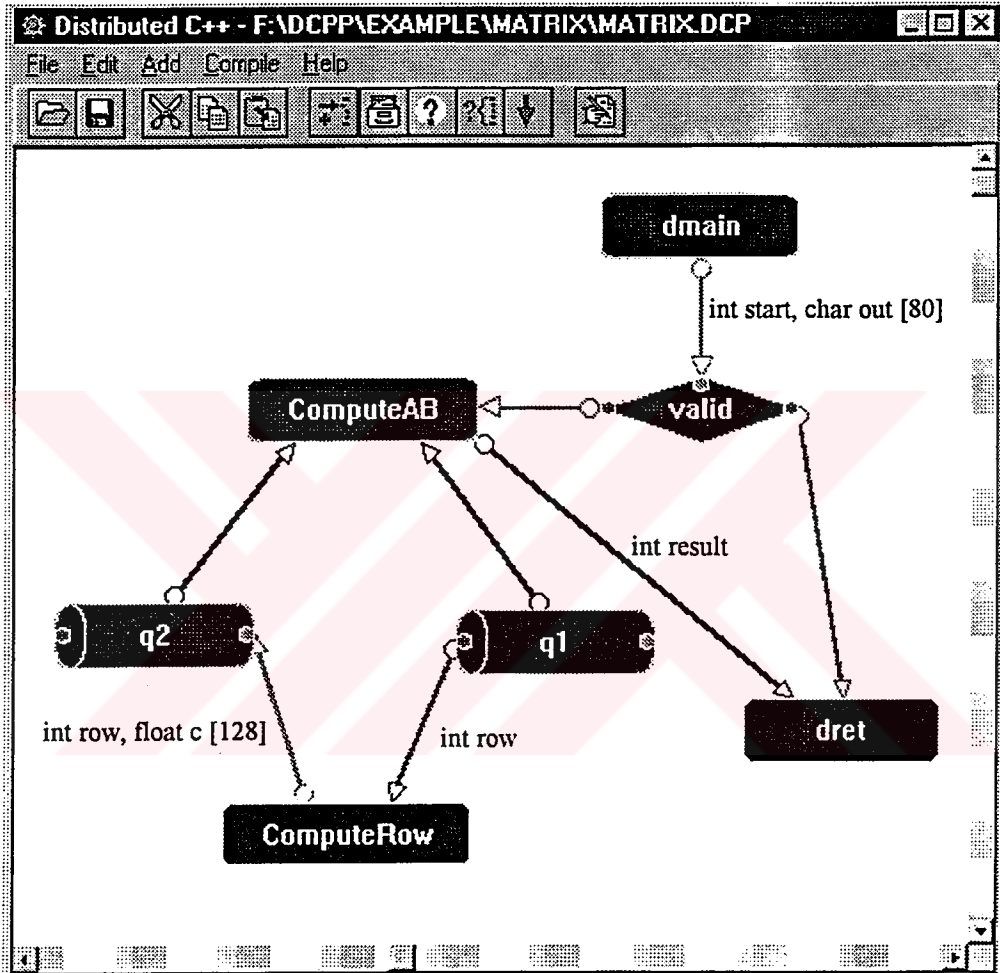


Figure 5.2 Matrix Multiplication Program

The *ComputeAB* thread puts all numbers corresponding to rows of C to *q1*, and then, it waits for the results in *q2*. After all the rows of C are completed, it writes C matrix to file named out. If any error occurs in this operation, the return value *result*, is set to zero to indicate an error, and thread returns. If

all the operations completed successfully, it sets result to 1, and returns after writing the output.

The ComputeRow block is called whenever an item is found in the queue. The ComputeRow block first checks the existence of the replicas of A, and B, in its main memory. If the replicas are not found, they are taken from the master station. Then, it gets this row number, and using the replicated copies of A, and B matrices, it computes the corresponding row of C matrix. After the row is computed, it returns the row number and resultant row, and these return values are put into q2.

The converted source codes for this program can be found in appendix E.



CHAPTER 6

CONCLUSION

In this thesis, a visual development system for programming parallel and distributed systems is presented.

The DCPD provides an easy to use interface for building programs. The programs are build on a visual development environment by using block, queue, decision, and loop tools. The programmer builds a control flow diagram for the program with these tools. The graphical development environment and DCPD's simple syntax ease the design and implementation process, and enable programmers to visualize the whole program easily. After the implementation process, the DCPD converts the program into C++ source files. These source files include the required definitions, declarations to execute the program on a parallel or distributed system. However, the approach presented in this thesis can also be used for different purposes. The programming model, and conversion system can be extended to implement a visual system for sequential, or multithreaded programming.

The DCPD hides the low level details of underlying operating system, and hardware primitives. The execution of blocks as threads or making remote procedure calls for blocks to remote machines, handling of the loop and decision tools, management of queues are done automatically. Therefore, by using the DCPD, it is possible to build programs independent from the target

platform. However, it is sometimes advantageous to access and use operating system tools to improve performance, or ease the implementation of a parallel or distributed program. For example, a programmer may want to use a tool provided by operating system in a program for distributing large data to multiple computers in an efficient way. In addition to normal programming model, the proposed system also permits the access to environment resources. This presents a great flexibility for the programmers who use the DCP.

The DCP has a simple syntax. This makes it easy to learn and use. Programming in DCP is made as close as possible to standard procedural programming model. Consequently switching from sequential programming to parallel programming is made easy for the programmers who are not familiar with parallel programming concepts. Besides, instead of designing a custom language that is used with DCP, the C++ language is selected for the base language for the DCP. Therefore, the programmers who are familiar with C++ language can easily adapt to this system. The powerful libraries, and extensive syntax of C++ language are also made available.

The current DCP implementation can build programs for three target platforms. The first target system is the network of computers with DOS operating system. The second is the shared memory, symmetric multiprocessor system with Windows NT operating system. Finally the third is also a shared memory multiprocessor system with AIX operating system. Although these two types of systems possess quite different parallel programming approach, and programming interfaces, the DCP presents a unified interface, and output syntax for programming these two systems using a hardware and operating system abstraction mechanism. Using different runtime systems to get executable files, both systems are supported with the same output files.

The hardware and software abstraction system supports portable programs for all MIMD parallel and distributed systems, by implementing a suitable runtime system for every different platform. In DCP, more general, and simple assumptions are made about the execution environment to increase portability chance. Selection of C++ as a base language is also an important criterion for covering broad range of execution platforms. As it is stated before, the current DCP implementation provides three runtime systems for the three target platforms. For Windows NT, and AIX implementations, the runtime system, makes use of the multithreaded operating system kernel. For network of computers implementations, the runtime system uses parallel programming library designed for personal computer connected by a local area network. As a future research, many runtime systems for different execution environments, or different runtime systems for these two environments with different operating systems and different parallel and distributed libraries can be prepared. For example, network of computers runtime can be build upon a shared memory distributed operating system. Therefore, like in shared memory implementation, global shared variables can be used in programs to increase usability of the DCP system.

The current implementation of the DCP, converts blocks to two functions, one is the main block function, and the other is the caller function for this block. The caller function, prepares a standard interface for calling block functions on remote computers. This standard interface is a good approach to increase the chance for porting the DCP to any other remote procedure call based distributed memory system. However, the caller functions cause an overhead of calling one more function for a block in the shared memory multiprocessor system. There are some other overheads induced by portability requirement for both target systems. It is possible to design a new system without changing the current programming model, and making

changes in conversion system to get a new and more efficient tool for any of these target platforms.

As a future improvement, it is possible to add new tools to the system. For instance, it may be possible to implement, with some modifications a tool for implementing libraries through the DCPD and using these libraries in sequential languages, or in other DCPD programs.



REFERENCES

- [1] Foster, I., *Designing and Building Parallel Programs*, Addison-Wesley Publishing Company, 1994.
- [2] White, S. et al., *How Does Processor MHz Relate to End-User Performance?*, IEEE Micro, Part I, August 1993, Part II, October 1993.
- [3] Halıcı U., Aybay I., *Lecture notes on Operating Systems*, METU, 1992.
- [4] Morse, H. S., *Practical Parallel Computing*, Academic Press, Inc., 1994.
- [5] Flynn, M., *Some Computer Organizations and Their Effectiveness*, IEEE Trans. on Computers, Vol C-21, 1972.
- [6] Bräunl, T., *Parallel Programming*, Prentice Hall International Limited, 1993.
- [7] Bertsekas, D. P., Tsitsiklis, J. N., *Parallel and Distributed Computation*, Prentice Hall, 1989.
- [8] Bal, H. E., *Programming Distributed Systems*, Slicon Press, 1990.
- [9] Microsoft Corp., *RPC On-line Help*, Microsoft Corp., 1994.
- [10] Chang, S., *Visual Languages: A Tutorial and Survey*, IEEE Software, January 1987.

- [11] Ambler, A. L., Burnett, M. M., *Influence of Visual Technology on the evaluation of Language Environments*, IEEE Computer, October 1989.
- [12] Glinert, E. P., Tanimoto, S. L., *PICT: An Interactive, Graphical Programming Environment*, IEEE Computer, November 1984.
- [13] Snyder, L., *Parallel Programming and the POKER Programming Environment*, IEEE Computer, July 1984.
- [14] Giacalone, A., Smolka, S. A. *Integrated Environments for Formally Well-Founded Design and Simulation of Concurrent Systems*, IEEE Trans. on Software Engineering, June 1988.
- [15] Stotts, P. D., *The PFG Language: Visual Programming for Concurrent Computation*, Proc. Int. Conf. on Parallel Processing, Volume 2: Software, Pennsylvania State University Press, August 1988.
- [16] Stovsky, M. P., Weide B. W., *Building Interprocess Communication Models Using STILE*, Proc. 21st Hawaii Int. Conf. on System Sciences, IEEE Computer Society Press, January 1988.
- [17] Graf, M., *A Visual Environment for the Design of Distributed Systems*, Plenum Press, 1990.
- [18] Microsoft Corp., *WIN32 On-line Help*, Microsoft Corp., 1994.
- [19] Acar C. E., *A Parallel Programming Environment for PC Compatible Computers Connected by Ethernet Local Area Network*, Masters Thesis at METU, 1996.
- [20] Newton P., *A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation*, Technical Report

TR93-28, Dept. of Computer Sciences, Univ. of Texas at Austin, 1993.

- [21] Newton P., *Visual Programming and Parallel Computing*, Workshop on Environments and Tools for Parallel Scientific Computing, Walland, TN, May, 1994.
- [22] Browne J.C., Dongarra J., Hyder S.I., Moore K., Newton P., *Experiences with CODE and HeNCE in Visual Programming for Parallel Computing*, IEEE Parallel and Distributed Technology, Vol. 3 No.1 pp 75-83, Spring 1994.
- [23] Beguelin A., Dongarra J., Geist G. A., Mancbek R., Sunderam V.S., *Graphical Development Tools for Network Based Concurrent Supercomputing*, Proceedings of Supercomputing '91, IEEE press pp 435-444, 1991.
- [24] Beguelin A., Dongarra J., Geist A., Sunderam V.S., *Visualization and Debugging in a Heterogeneous Environment*, IEEE Computer, June 1993.
- [25] Dongarra J., Newton P., *Overview of VPE: A Visual Environment for Message Passing*, Heterogeneous Computing Workshop '95, Proceedings of 4th Heterogeneous Computing Workshop, Santa Barbara CA, April, 1995.
- [26] Zhang K., Ma W., *Graphical Assistance in Parallel Program Development*, Proc. VL '94 - 10th IEEE International Symposium on Visual Languages, St Louis USA, October, 1994.
- [27] Zhang D., Zhang K., *A Visual Programming Environment for Distributed Systems*, Proc. VL '95 - 11th IEEE International

Symposium on Visual Languages, Darmstadt Germany, September, 1995.

- [28] Sunderam V., Dongarra J., Geist A., and Manchek R, *The PVM Concurrent Computing System: Evolution, Experiences, and Trends*, Parallel Computing, Vol. 20, No. 4, pp 531-547, April 1994.
- [29] Newton P., Browne J.C., *The CODE 2.0 Parallel Programming Language*, Proc. ACM Int. Conf. on Supercomputing, July, 1992.
- [30] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, Journal of Supercomputing Applications, Vol. 8, No. 3/4, 1994.



APPENDIX A

RUNTIME SYSTEM FOR NETWORK OF COMPUTERS

DCPP.H

```
#if !defined(__DCPP_H)
#define __DCPP_H

#include <iostream.h>
#include <conio.h>

#include "ppnww.h"

typedef void (*ThreadFunc)(char*, int, char*, int);

class TDcppThread{
public:
    enum ArgState { Ready, Passed, Waiting };
    TDcppThread(unsigned anId, unsigned aSeq, unsigned aCount,
                  ThreadFunc aFunc, char *aArg, int aArgSz, char
                  *aRet, int aRetSz);
    virtual ~TDcppThread();
    int IsThread(unsigned anId, unsigned aSeq)
        { return (Id==anId) && (Seq==aSeq); }
    unsigned long Start()
        { Handle = RPCCall((RPC*)Func, Arg, ArgSize, Ret, RetSize); return
        Handle!=NULL; }
    virtual char ReturnThread() = 0;
    virtual char PassThread() = 0;
    int IsCompleted()
        { return RSSstatus(Handle)==RS_DONE; }
    ArgState GetArgState();
    ArgState& operator[](unsigned index)
        { return ArgStat[index]; }
    void SetAllPassed();
private:
    unsigned Id;
    unsigned ArgCount;
```

```

RCALL *Handle;
protected:
    unsigned Seq;
    ArgState *ArgStat;
    ThreadFunc Func;
    char *Arg;
    int ArgSize;
    char *Ret;
    int RetSize;
};

```

```

template <class T>
class TDcppQueue{
private:
    /* T** Items;
    int Size;
    int Head;
    int Tail;*/
    char *Name;
    int Master;
    RQueue *Handle;
public:
    TDcppQueue (int N, char *aName);
    TDcppQueue (char *aName)
        : /*Size(0), Head(0), Tail(0), */Name(aName), Master(0), Handle(NULL) {
    };
    ~TDcppQueue() {};
    int Put(T* Item);
    T* Get();
};

```

```

template <class T>
    TDcppQueue<T>::TDcppQueue(int N, char *aName)
        : /*Size(0), Head(0), Tail(0), */Name(aName), Master(1)
    {
        Handle = RQCreate(Name, (sizeof(T)+4)*N, sizeof(T));
    }

```

```

template <class T>
T* TDcppQueue<T>::Get()
{
    T* Result = new T;

    if(!Handle) Handle=RQLocate(Name, sizeof(T));
    if(RQGetNB(Handle, Result)){
        delete Result;
        Result = NULL;
    }
    return Result;
}

```

```

template <class T>
int TDcppQueue<T>::Put(T* Item)

```



```

{
    if(!Handle) Handle=RQLocate(Name, sizeof(T));
    return !RQAdd(Handle, Item);
}

class TDcppPool{
private:
    TDcppThread** Pool;
    unsigned Count;
    unsigned Max;
    unsigned Delta;
    int Enlarge();
public:
    TDcppPool(unsigned aMax, unsigned aDelta);
    ~TDcppPool();
    int IsEmpty() { return Count == 0; }
    int AddThread(TDcppThread* aThread);
    void RemoveThread(TDcppThread* aThread);
    TDcppThread* FindThread(unsigned anId, unsigned aSeq);
    TDcppThread* FindFinished();
    TDcppThread* FindReadyOrPassed();
};

extern TDcppPool WaitPool;

extern TDcppPool ExecPool;

void Executive();

typedef TDcppThread* (*QueueFunc)();

extern QueueFunc ThreadQueues[];

char CreateQueues(int Master);

void DestroyQueues();

void Sleep(int d);

#endif

```

DCPP.CPP

```

#include <iostream.h>
#include <stdlib.h>
#include "dcpnw.h"
#include "caller.h"
#include "timeout.h"
// TDcppThread Class

```

```

TDcppThread::TDcppThread(unsigned anId, unsigned aSeq, unsigned aCount,

```

```

                                ThreadFunc aFunc, char *aArg, int aArgSz, char
*aRet, int aRetSz)
    : Id(anId), ArgCount(aCount), Seq(aSeq),
      Func(aFunc), Arg(aArg), ArgSize(aArgSz),
      Ret(aRet), RetSize(aRetSz)
{
    ArgStat = new ArgState[ArgCount];
    for(unsigned i=0; i<ArgCount; i++) ArgStat[i] = Waiting;
}

TDcppThread::~TDcppThread()
{
    delete[] ArgStat;
    RSRelease(Handle);
// TThread::~TThread();
}

TDcppThread::ArgState TDcppThread::GetArgState()
{
    unsigned i;
    int p=0;
    for(i=0; i<ArgCount; i++)
    {
        if(ArgStat[i]==Waiting) break;
        if(!p && (ArgStat[i]==Ready)) p=1;
    }
    if(i<ArgCount) return Waiting;
    if(!p) return Passed;
    return Ready;
}

void TDcppThread::SetAllPassed()
{
    for(unsigned i=0; i<ArgCount; i++) operator[](i) = Passed;
}

// TPoolClass

TDcppPool::TDcppPool(unsigned aMax, unsigned aDelta)
    : Count(0), Max(aMax), Delta(aDelta)
{
    Pool = (TDcppThread**) new char[aMax*sizeof(TDcppThread*)];
    for(unsigned i=0; i<aMax; i++) Pool[i] = NULL;
}

TDcppPool::~TDcppPool()
{
    delete[] Pool;
}

int TDcppPool::Enlarge()
{
    TDcppThread** NewPool;

```

```

NewPool = (TDcppThread**) new char[(Max+Delta)*sizeof(TDcppThread*)];
if(!NewPool) return 0;
for(unsigned i=0; i<Count; i++) NewPool[i]=Pool[i];
delete[] Pool;
Pool = NewPool;
Max += Delta;
return 1;
}

int TDcppPool::AddThread(TDcppThread* aThread)
{
    if(Count>=(Max-1))
        if(!Enlarge()) return 0;
    Pool[Count++] = aThread;
    return 1;
}

void TDcppPool::RemoveThread(TDcppThread* aThread)
{
    unsigned i;
    for(i=0; i<Count; i++)
        if(Pool[i]==aThread) break;
    if(i<Count) Count--;
    for(;i<Count; i++) Pool[i] = Pool[i+1];
    Pool[Count] = NULL;
}

TDcppThread* TDcppPool::FindThread(unsigned anId, unsigned aSeq)
{
    unsigned i;
    for(i=0; i<Count; i++)
        if(Pool[i]->IsThread(anId, aSeq)) break;
    return Pool[i];
}

TDcppThread* TDcppPool::FindFinished()
{
    unsigned i;
    for(i=0; i<Count; i++)
        if(Pool[i]->IsCompleted()) break;
    return Pool[i];
}

TDcppThread* TDcppPool::FindReadyOrPassed()
{
    TDcppThread::ArgState ArgState;
    unsigned i;
    for(i=0; i<Count; i++){
        ArgState = Pool[i]->GetArgState();
        if( (ArgState==TDcppThread::Passed)||
            (ArgState==TDcppThread::Ready) ) break;
    }
    return Pool[i];
}

```

```

}

TDcppPool WaitPool(10, 5);

TDcppPool ExecPool(10, 5);

void Executive(int argc, char** argv)
{
    TDcppThread *T;
    Thread_dmain *T_dmain;
    T_dmain = new Thread_dmain(0);
    T_dmain->arg.argc = argc;
    T_dmain->arg.argv = argv;
    (*T_dmain)[0] = TDcppThread::Ready;
    WaitPool.AddThread(T_dmain);
    while((!WaitPool.IsEmpty())||(!ExecPool.IsEmpty())){
        PPTick();
        T = WaitPool.FindReadyOrPassed();
        if(T){
            if (T->GetArgState() == TDcppThread::Passed){
                T->PassThread();
                WaitPool.RemoveThread(T);
                /*
                try {*/
                    delete T;
                /*
                } catch (...){
                ;
                }*/
            } else {
                if(T->Start()){
                    WaitPool.RemoveThread(T);
                    ExecPool.AddThread(T);
                }
            }
        } else if(HostByState(HST_IDLE)){
            for(int i=0; ThreadQueues[i]; i++){
                TDcppThread *QT;
                QT = ThreadQueues[i];
                if(QT) WaitPool.AddThread(QT);
            }
        } else RSTryUpdate();
        T = ExecPool.FindFinished();
        if(T){
            ExecPool.RemoveThread(T);
            T->ReturnThread();
            delete T;
        }
    }
}

void IdleFunc(void)
{

```

```

    if(HIDEErrorFlag) PPEndIdle();
}

#pragma argsused
int main(int argc, char** argv)
{
    int master,stat;

    if(argc==2){
        master=1;
        stat=atoi(argv[1]);
        if(stat<0) stat=1;
        if(stat>200) stat=200;
    }else master=0;

    if(PPInit(master,0,0)){
        cout<<"Initialization Failed\n";
        exit(1);
    }
    PPSetIdleFn(IdleFunc);

    CreateQueues(master);
    if(master){
        cout<<"Waiting for "<<stat<<" stations...\n";
        while(num_hosts<stat) PPTick();
        cout<<"Done!\n";
        Executive(argc, argv);
    }else PPGoIdle();

    DestroyQueues();
    return 0;
}

void Sleep(int d)
{
    clock_t t;
    cout << "Sleep "<<d<<endl;
    t=clock();
    while(!mstimeout(t,d)) PPTick();
}

```

APPENDIX B

RUNTIME SYSTEM FOR WINDOWS NT ON

MULTIPROCESSORS

DCPP.H

```
#if !defined(__DCPP_H)
#define __DCPP_H

#include <classlib\thread.h>

typedef void (*ThreadFunc)(char*, int, char*, int);

class TDcppThread: public TThread{
public:
    enum ArgState { Ready, Passed, Waiting };
    TDcppThread(unsigned anId, unsigned aSeq, unsigned aCount,
                  ThreadFunc aFunc, char *aArg, int aArgSz, char
*aRet, int aRetSz);
    ~TDcppThread();
    int IsThread(unsigned anId, unsigned aSeq)
        { return (Id==anId) && (Seq==aSeq); }
    unsigned long Run()
        { (*Func)(Arg, ArgSize, Ret, RetSize); return 0L; }
    virtual char ReturnThread() = 0;
    virtual char PassThread() = 0;
    int IsCompleted()
        { return GetStatus()==Finished; }
    ArgState GetArgState();
    ArgState& operator[](unsigned index)
        { return ArgStat[index]; }
    void SetAllPassed();
private:
    unsigned Id;
    unsigned ArgCount;
```

```

protected:
    unsigned Seq;
    ArgState *ArgStat;
    ThreadFunc Func;
    char *Arg;
    int ArgSize;
    char *Ret;
    int RetSize;
};

template <class T>
class TDcppQueue{
private:
    T** Items;
    int Size;
    int Head;
    int Tail;
    TCriticalSection LockQ;
    char *Name;
public:
    TDcppQueue(int N, char* aName);
    TDcppQueue<T>::TDcppQueue(char* aName)
        : Size(0), Head(0), Tail(0), Name(aName) {};
    ~TDcppQueue();
    T* Put(T* Item);
    T* Get();
};

template <class T>
TDcppQueue<T>::TDcppQueue(int N, char* aName)
    : Size(N), Head(0), Tail(0), Name(aName)
{
    Items = (T**) new char[N*sizeof(T*)];
    for(int i=0; i<N; i++) Items[i] = NULL;
}

template <class T>
TDcppQueue<T>::~~TDcppQueue()
{
    for(int i=0; i<Size; i++) if(Items[i]) delete Items[i];
    delete Items;
}

template <class T>
T* TDcppQueue<T>::Get()
{
    TCriticalSection::Lock lock(LockQ);
    T* Result;
    if(Head==Tail)
        Result = NULL;
    else {
        Result = Items[Head];
        Items[Head++] = NULL;
    }
}

```

```

        Head %= Size;
    }
    return Result;
}

template <class T>
T* TDcppQueue<T>::Put(T* Item)
{
    TCriticalSection::Lock k(LockQ);
    T* Result;
    if(Head==((Tail+1)%Size))
        Result = NULL;
    else {
        Result = Items[Tail++] = Item;
        Tail %= Size;
    }
    return Result;
}

class TDcppPool{
private:
    TDcppThread** Pool;
    unsigned Count;
    unsigned Max;
    unsigned Delta;
    int Enlarge();
public:
    TDcppPool(unsigned aMax, unsigned aDelta);
    ~TDcppPool();
    int IsEmpty() { return Count == 0; }
    int AddThread(TDcppThread* aThread);
    void RemoveThread(TDcppThread* aThread);
    TDcppThread* FindThread(unsigned anId, unsigned aSeq);
    TDcppThread* FindFinished();
    TDcppThread* FindReadyOrPassed();
    unsigned GetCount(){ return Count; }
};

extern TDcppPool WaitPool;

extern TDcppPool ExecPool;

void Executive();

typedef TDcppThread* (*QueueFunc)();

extern QueueFunc ThreadQueues[];

char CreateQueues(int Master);

void DestroyQueues();

#endif

```


DCPP.CPP

```
#include "dcp.h"
#include "caller.h"
#include <stdlib.h>

// TDcppThread Class

TDcppThread::TDcppThread(unsigned anId, unsigned aSeq, unsigned aCount,
                          ThreadFunc aFunc, char *aArg, int aArgSz, char
                          *aRet, int aRetSz)
    : Id(anId), Seq(aSeq), ArgCount(aCount),
      Func(aFunc), Arg(aArg), ArgSize(aArgSz),
      Ret(aRet), RetSize(aRetSz)
{
    ArgStat = new ArgState[ArgCount];
    for(int i=0; i<ArgCount; i++) ArgStat[i] = Waiting;
}

TDcppThread::~TDcppThread()
{
    delete[] ArgStat;
}

TDcppThread::ArgState TDcppThread::GetArgState()
{
    int p=0;
    for(int i=0; i<ArgCount; i++)
    {
        if(ArgStat[i]==Waiting) break;
        if(!p && (ArgStat[i]==Ready)) p=1;
    }
    if(i<ArgCount) return Waiting;
    if(!p) return Passed;
    return Ready;
}

void TDcppThread::SetAllPassed()
{
    for(int i=0; i<ArgCount; i++) operator[](i) = Passed;
}

// TPoolClass

TDcppPool::TDcppPool(unsigned aMax, unsigned aDelta)
    : Max(aMax), Delta(aDelta), Count(0)
{
    Pool = (TDcppThread**) new char[aMax*sizeof(TDcppThread*)];
    for(int i=0; i<aMax; i++) Pool[i] = NULL;
}
```

```

TDcppPool::~TDcppPool()
{
    delete[] Pool;
}

int TDcppPool::Enlarge()
{
    TDcppThread** NewPool;
    NewPool = (TDcppThread**) new char[(Max+Delta)*sizeof(TDcppThread*));
    if(!NewPool) return 0;
    for(int i=0; i<Count; i++) NewPool[i]=Pool[i];
    delete[] Pool;
    Pool = NewPool;
    Max += Delta;
    return 1;
}

int TDcppPool::AddThread(TDcppThread* aThread)
{
    if(Count>=(Max-1))
        if(!Enlarge()) return 0;
    Pool[Count++] = aThread;
    return 1;
}

void TDcppPool::RemoveThread(TDcppThread* aThread)
{
    for(int i=0; i<Count; i++)
        if(Pool[i]==aThread) break;
    if(i<Count) Count--;
    for(;i<Count; i++) Pool[i] = Pool[i+1];
    Pool[Count] = NULL;
}

TDcppThread* TDcppPool::FindThread(unsigned anId, unsigned aSeq)
{
    for(int i=0; i<Count; i++)
        if(Pool[i]->IsThread(anId, aSeq)) break;
    return Pool[i];
}

TDcppThread* TDcppPool::FindFinished()
{
    for(int i=0; i<Count; i++)
        if(Pool[i]->IsCompleted()) break;
    return Pool[i];
}

TDcppThread* TDcppPool::FindReadyOrPassed()
{
    TDcppThread::ArgState ArgState;
    for(int i=0; i<Count; i++){

```

```

        ArgState = Pool[i]->GetArgState();
        if( (ArgState==TDcppThread::Passed)||
            (ArgState==TDcppThread::Ready) ) break;
    }
    return Pool[i];
}

TDcppPool WaitPool(10, 5);

TDcppPool ExecPool(10, 5);

int MAXTHREADS;

void Executive(int argc, char** argv)
{
    TDcppThread *T;
    Thread_dmain *T_dmain;
    if(argc<2) return;
    MAXTHREADS = atoi(argv[1]);
    if(MAXTHREADS<2) return;
    T_dmain = new Thread_dmain(0);
    T_dmain->arg.argc = argc-1;
    T_dmain->arg.argv = &argv[1];
    (*T_dmain)[0] = TDcppThread::Ready;
    WaitPool.AddThread(T_dmain);
    while((!WaitPool.IsEmpty())||(!ExecPool.IsEmpty())){
        for(int i=0; ThreadQueues[i]; i++){
            TDcppThread *QT;
            QT = ThreadQueues[i][0];
            if(QT){
                WaitPool.AddThread(QT);
                break;
            }
        }
        if(ExecPool.GetCount()<MAXTHREADS)
            T = WaitPool.FindReadyOrPassed();
        else
            T = NULL;
        if(T){
            if (T->GetArgState()==TDcppThread::Passed){
                T->PassThread();
                WaitPool.RemoveThread(T);
                try {
                    delete T;
                } catch (...){
                    cout << "exception" << endl;
                }
            } else {
                T->Start();
                WaitPool.RemoveThread(T);
                ExecPool.AddThread(T);
            }
        }
    }
}

```

```

    T = ExecPool.FindFinished();
    if(T){
        ExecPool.RemoveThread(T);
        T->ReturnThread();
        delete T;
    }
}

#pragma argsused
void main(int argc, char** argv)
{
    CreateQueues(1);
    Executive(argc, argv);
    DestroyQueues();
    return;
}

```



APPENDIX C

RUNTIME SYSTEM FOR AIX ON

MULTIPROCESSORS

DCPP.H

```
#if !defined(__DCPP_H)
#define __DCPP_H

#include <pthread.h>
#include <stdlib.h>
#include <iostream.h>

typedef void (*ThreadFunc)(char*, int, char*, int);

struct ThreadParams{
    ThreadFunc f;
    char *arg, *ret;
    int argsz, retsz;
    int *Result;
};

void* Call(void *p);

class TDcppThread{
public:
    enum ArgState { Ready, Passed, Waiting };
    TDcppThread(unsigned anId, unsigned aSeq, unsigned aCount,
                 ThreadFunc aFunc, char *aArg, int aArgSz, char *aRet, int
aRetSz);
    ~TDcppThread();
    int IsThread(unsigned anId, unsigned aSeq)
        { return (Id==anId) && (Seq==aSeq); }
    void Start(){
        Status = 0;
    }
};
```

```

    tparams->f=Func;
    tparams->arg=Arg;
    tparams->ret=Ret;
    tparams->argsz=ArgSize;
    tparams->retsz=RetSize;
    tparams->Result=&Status;
    pthread_create(&e_th, NULL, Call, tparams);
}
virtual char ReturnThread() = 0;
virtual char PassThread() = 0;
int IsCompleted()
    { return Status; }
ArgState GetArgState();
ArgState& operator[](unsigned index)
    { return ArgStat[index]; }
void SetAllPassed();
private:
    unsigned Id;
    pthread_t e_th;
    unsigned ArgCount;
    int Status;
protected:
    unsigned Seq;
    ArgState *ArgStat;
    ThreadFunc Func;
    char *Arg;
    int ArgSize;
    char *Ret;
    int RetSize;
    ThreadParams *tparams;
};

template <class T>
class TDcppQueue{
private:
    T** Items;
    int Size;
    int Head;
    int Tail;
    pthread_mutex_t mutex;
    char *Name;
public:
    TDcppQueue (int N, char* aName);
    TDcppQueue<T>::TDcppQueue(char* aName)
        : Size(0), Head(0), Tail(0), Name(aName){
        pthread_mutex_init(&mutex, NULL);
    }
    ~TDcppQueue();
    T* Put(T* Item);
    T* Get();
};

template <class T>

```

```

TDcppQueue<T>::TDcppQueue(int N, char* aName)
    : Size(N), Head(0), Tail(0), Name(aName)
{
    Items = (T**) new char[N*sizeof(T*)];
    for(int i=0; i<N; i++) Items[i] = NULL;
    pthread_mutex_init(&mutex, NULL);
}

template <class T>
TDcppQueue<T>::~TDcppQueue()
{
    for(int i=0; i<Size; i++) if(Items[i]) delete Items[i];
    delete Items;
}

template <class T>
T* TDcppQueue<T>::Get()
{
    pthread_mutex_lock(&mutex);
    T* Result;
    if(Head==Tail)
        Result = NULL;
    else {
        Result = Items[Head];
        Items[Head++] = NULL;
        Head %= Size;
    }
    pthread_mutex_unlock(&mutex);
    return Result;
}

template <class T>
T* TDcppQueue<T>::Put(T* Item)
{
    pthread_mutex_lock(&mutex);
    T* Result;
    if(Head==((Tail+1)%Size))
        Result = NULL;
    else {
        Result = Items[Tail++] = Item;
        Tail %= Size;
    }
    pthread_mutex_unlock(&mutex);
    return Result;
}

class TDcppPool{
private:
    TDcppThread** Pool;
    unsigned Count;
    unsigned Max;
    unsigned Delta;
    int Enlarge();
}

```

```

public:
    TDcppPool(unsigned aMax, unsigned aDelta);
    ~TDcppPool();
    int IsEmpty() { return Count == 0; }
    int AddThread(TDcppThread* aThread);
    void RemoveThread(TDcppThread* aThread);
    TDcppThread* FindThread(unsigned anId, unsigned aSeq);
    TDcppThread* FindFinished();
    TDcppThread* FindReadyOrPassed();
    unsigned GetCount(){ return Count; }
};

extern TDcppPool WaitPool;

extern TDcppPool ExecPool;

void Executive();

typedef TDcppThread* (*QueueFunc)();

extern QueueFunc ThreadQueues[];

char CreateQueues(int Master);

void DestroyQueues();

#endif

```

DCPP.C

```

#include "dcpp.h"
#include "caller.h"

void* Call(void *p)
{
    ThreadParams *P=(ThreadParams*)p;
    (*(P->f))(P->arg, P->argsz, P->ret, P->retsz);
    *(P->Result)=1;
    pthread_exit(NULL);
    return NULL;
}

// TDcppThread Class

TDcppThread::TDcppThread(unsigned anId, unsigned aSeq, unsigned aCount,
                        ThreadFunc aFunc, char *aArg, int aArgSz, char
*aRet, int aRetSz)
    : Id(anId), Seq(aSeq), ArgCount(aCount),
      Func(aFunc), Arg(aArg), ArgSize(aArgSz),
      Ret(aRet), RetSize(aRetSz)
{

```



```

        ArgStat = new ArgState[ArgCount];
        for(int i=0; i<ArgCount; i++) ArgStat[i] = Waiting;
        tparams = new ThreadParams;
    }

    TDcppThread::~TDcppThread()
    {
        delete[] ArgStat;
    }

    TDcppThread::ArgState TDcppThread::GetArgState()
    {
        int p=0;
        for(int i=0; i<ArgCount; i++)
        {
            if(ArgStat[i]==Waiting) break;
            if(!p && (ArgStat[i]==Ready)) p=1;
        }
        if(i<ArgCount) return Waiting;
        if(!p) return Passed;
        return Ready;
    }

    void TDcppThread::SetAllPassed()
    {
        for(int i=0; i<ArgCount; i++) operator[](i) = Passed;
    }

    // TPoolClass

    TDcppPool::TDcppPool(unsigned aMax, unsigned aDelta)
        : Max(aMax), Delta(aDelta), Count(0)
    {
        Pool = (TDcppThread**) new char[aMax*sizeof(TDcppThread*));
        for(int i=0; i<aMax; i++) Pool[i] = NULL;
    }

    TDcppPool::~TDcppPool()
    {
        delete[] Pool;
    }

    int TDcppPool::Enlarge()
    {
        TDcppThread** NewPool;
        NewPool = (TDcppThread**) new char[(Max+Delta)*sizeof(TDcppThread*));
        if(!NewPool) return 0;
        for(int i=0; i<Count; i++) NewPool[i]=Pool[i];
        delete[] Pool;
        Pool = NewPool;
        Max += Delta;
        return 1;
    }

```

```

int TDcppPool::AddThread(TDcppThread* aThread)
{
    if(Count>=(Max-1))
        if(!Enlarge()) return 0;
    Pool[Count++] = aThread;
    return 1;
}

void TDcppPool::RemoveThread(TDcppThread* aThread)
{
    for(int i=0; i<Count; i++)
        if(Pool[i]==aThread) break;
    if(i<Count) Count--;
    for(;i<Count; i++) Pool[i] = Pool[i+1];
    Pool[Count] = NULL;
}

TDcppThread* TDcppPool::FindThread(unsigned anId, unsigned aSeq)
{
    for(int i=0; i<Count; i++)
        if(Pool[i]->IsThread(anId, aSeq)) break;
    return Pool[i];
}

TDcppThread* TDcppPool::FindFinished()
{
    for(int i=0; i<Count; i++)
        if(Pool[i]->IsCompleted()) break;
    return Pool[i];
}

TDcppThread* TDcppPool::FindReadyOrPassed()
{
    TDcppThread::ArgState ArgState;
    for(int i=0; i<Count; i++){
        ArgState = Pool[i]->GetArgState();
        if( (ArgState==TDcppThread::Passed)||
            (ArgState==TDcppThread::Ready) ) break;
    }
    return Pool[i];
}

TDcppPool WaitPool(10, 5);
TDcppPool ExecPool(10, 5);

int MAXTHREADCNT;

void Executive(int argc, char** argv)
{
    TDcppThread *T;
    Thread_dmain *T_dmain;

```

```

if(argc<2) return;
int MAXTHREADCNT = atoi(argv[1]);
if(MAXTHREADS<2) return;
T_dmain = new Thread_dmain(0);
T_dmain->arg.argc = argc-1;
T_dmain->arg.argv = &argv[1];
(*T_dmain)[0] = TDcppThread::Ready;
WaitPool.AddThread(T_dmain);
while((!WaitPool.IsEmpty())||(!ExecPool.IsEmpty())){
    for(int i=0; ThreadQueues[i]; i++){
        TDcppThread *QT;
        QT = ThreadQueues[i];
        if(QT){
            WaitPool.AddThread(QT);
            break;
        }
    }
    if(ExecPool.GetCount()<MAXTHREADCNT)
        T = WaitPool.FindReadyOrPassed();
    else
        T = NULL;
    if(T){
        if (T->GetArgState()==TDcppThread::Passed){
            T->PassThread();
            WaitPool.RemoveThread(T);
            delete T;
        } else {
            T->Start();
            WaitPool.RemoveThread(T);
            ExecPool.AddThread(T);
        }
    }
    T = ExecPool.FindFinished();
    if(T){
        ExecPool.RemoveThread(T);
        T->ReturnThread();
        delete T;
    }
}
}

#pragma argsused
void main(int argc, char** argv)
{
    CreateQueues(1);
    Executive(argc, argv);
    DestroyQueues();
    return;
}

```

APPENDIX D

MATRIX MULTIPLICATION SOURCE CODES

MATRIX.CPP

```
#include "matrix.h"

#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include "datadist.h"

float A[128][128], B[128][128];
float C[128][128];

char A_B_available=0;

Ret_dmain* dmain(int argc, char** argv)
{
    Ret_dmain* ret = new Ret_dmain;
    ret->start = 0;
    if(argc<4){
        cout << "Usage: " << endl;
        cout << " MATRIX <matrix A> <matrix B> <matrix C>" << endl;
        return ret;
    }

    FILE* inA = fopen(argv[1], "rb");
    if(!inA){
        cout << " Unable to open file: " << argv[1] << endl;
        return ret;
    }

    FILE* inB = fopen(argv[2], "rb");
    if(!inB){
        fclose(inA);
        cout << " Unable to open file: " << argv[2] << endl;
        return ret;
    }
}
```

```

    }

    if(fread(A, sizeof(float), 128*128, inA)!=128*128){
        cout << "Unable to read file: " << argv[1] << endl;
        fclose(inA); fclose(inB);
        return ret;
    }

    if(fread(B, sizeof(float), 128*128, inB)!=128*128){
        cout << "Unable to read file: " << argv[2] << endl;
        fclose(inA); fclose(inB);
        return ret;
    }

    fclose(inA); fclose(inB);
    Distribute_A_B0;

    strcpy(ret->out, argv[3]);
    ret->start = 1;
    return ret;
}

Ret_ComputeAB* ComputeAB(int start, char out [80], TDcppQueue<QueueItem_q1>*
q1, TDcppQueue<QueueItem_q2>* q2)
{
    Ret_ComputeAB* ret = new Ret_ComputeAB;

    ret->result = 0;

    ret->result = 0;
    for(int i=0; i<128; i++){
        QueueItem_q1* Item = new QueueItem_q1;
        Item->row = i;
        while(!q1->Put(Item));
    }

    for(int j=0; j<128; j++){
        QueueItem_q2* Item;
        while(!(Item = q2->Get()));
        memcpy(C[Item->row], Item->c, 128*sizeof(float));
        delete Item;
    }

    FILE *stream = fopen(out, "wb");
    if(!stream) return ret;

    if(fwrite(C, sizeof(float), 128*128, stream)!=128*128){
        fclose(stream);
        return ret;
    }
    ret->result = 1;
    return ret;
}

```

```

Ret_dret* dret(int start,char out [80],int result)
{
    Ret_dret* ret = new Ret_dret;
        if(result||start)
            cout << "Matrix multiplication completed sucessfully..." << endl;
        else
            cout << "Matrix multiplication failed..." << endl;
    return ret;
}

Ret_ComputeRow* ComputeRow(int row)
{
    if (!A_B_available){
        GetA_B_FromMaster();
        A_B_available = 1;
    }
    Ret_ComputeRow* ret = new Ret_ComputeRow;
    for(int i=0; i<128; i++){
        ret->c[i] = 0;
        for (int j=0; j<128; j++)
            ret->c[i] += A[row][j]*B[j][i];
    }
    ret->row = row;
    // cout << row << endl;
    return ret;
}

int valid(int start,char out [80])
{
    return (start==1);
}

```

MATRIX.H

```

#ifndef __MATRIX_H
#define __MATRIX_H

#include "dcpp.h"

struct QueueItem_q1 {
    int row;
};

struct QueueItem_q2 {
    int row;
    float c [128];
};

struct Arg_dmain {
    int argc;
}

```

```

    char** argv;
};

struct Ret_dmain {
    int start;
    char out [80];
};

struct Arg_ComputeAB {
    int start;
    char out [80];
    TDcppQueue<QueueItem_q1>* q1;
    TDcppQueue<QueueItem_q2>* q2;
};

struct Ret_ComputeAB {
    int result;
};

struct Arg_dret {
    int start;
    char out [80];
    int result;
};

struct Ret_dret {
    int r;
};

struct Arg_ComputeRow {
    int row;
};

struct Ret_ComputeRow {
    int row;
    float c [128];
};

Ret_dmain* dmain(int argc,char** argv);
Ret_ComputeAB* ComputeAB(int start,char out [80],TDcppQueue<QueueItem_q1>*
q1,TDcppQueue<QueueItem_q2>* q2);
Ret_dret* dret(int start,char out [80],int result);
Ret_ComputeRow* ComputeRow(int row);
int valid(int start,char out [80]);

#endif

```

CALLER.CPP

```
#include "caller.h"
```

```

#pragma argsused
void Call_dmain(char *arg, int argsize, char *ret, int retsize)
{
    Arg_dmain *Arg;
    Ret_dmain *Ret;
    Arg = (Arg_dmain *)arg;
    Ret = dmain(Arg->argc, Arg->argv);
    memcpy(ret, Ret, retsize);
    delete Ret;
}

```

```

#pragma argsused
void Call_ComputeAB(char *arg, int argsize, char *ret, int retsize)
{
    Arg_ComputeAB *Arg;
    Ret_ComputeAB *Ret;
    Arg = (Arg_ComputeAB *)arg;
    Ret = ComputeAB(Arg->start, Arg->out, q1, q2);
    memcpy(ret, Ret, retsize);
    delete Ret;
}

```

```

#pragma argsused
void Call_dret(char *arg, int argsize, char *ret, int retsize)
{
    Arg_dret *Arg;
    Ret_dret *Ret;
    Arg = (Arg_dret *)arg;
    Ret = dret(Arg->start, Arg->out, Arg->result);
    memcpy(ret, Ret, retsize);
    delete Ret;
}

```

```

#pragma argsused
void Call_ComputeRow(char *arg, int argsize, char *ret, int retsize)
{
    Arg_ComputeRow *Arg;
    Ret_ComputeRow *Ret;
    Arg = (Arg_ComputeRow *)arg;
    Ret = ComputeRow(Arg->row);
    memcpy(ret, Ret, retsize);
    delete Ret;
}

```

```

unsigned Thread_dmain::SeqNo;

```

```

char Thread_dmain::ReturnThread()
{
    TDCppThread::ArgState first_branch, second_branch;

```



```

if(valid(ret.start,ret.out)){
    first_branch = TDcppThread::Ready;
    second_branch = TDcppThread::Passed;
} else {
    first_branch = TDcppThread::Passed;
    second_branch = TDcppThread::Ready;
}
{
    Thread_ComputeAB* T_ComputeAB;
    T_ComputeAB = (Thread_ComputeAB*) WaitPool.FindThread(ID_ComputeAB,
Seq);
    if(!T_ComputeAB){
        T_ComputeAB = new Thread_ComputeAB(Seq);
        WaitPool.AddThread(T_ComputeAB);
    }

    if (first_branch==TDcppThread::Passed) {
        T_ComputeAB->arg.start = 0;
        T_ComputeAB->arg.out[0] = 0;
    } else {
        T_ComputeAB->arg.start = ret.start;
        for(int i=0;i<80;i++)
            T_ComputeAB->arg.out[i] = ret.out[i];
    }
    (*T_ComputeAB)[0] = first_branch;
}
{
    Thread_dret* T_dret;
    T_dret = (Thread_dret*) WaitPool.FindThread(ID_dret, Seq);
    if(!T_dret){
        T_dret = new Thread_dret(Seq);
        WaitPool.AddThread(T_dret);
    }

    if (second_branch==TDcppThread::Passed) {
        T_dret->arg.start = 0;
        T_dret->arg.out[0] = 0;
    } else {
        T_dret->arg.start = ret.start;
        for(int i=0;i<80;i++)
            T_dret->arg.out[i] = ret.out[i];
    }
    (*T_dret)[0] = second_branch;
}
return 1;
}

char Thread_dmain::PassThread()
{
{
    Thread_ComputeAB* T_ComputeAB;
    T_ComputeAB = (Thread_ComputeAB*) WaitPool.FindThread(ID_ComputeAB,
Seq);

```

```

if(!T_ComputeAB){
    T_ComputeAB = new Thread_ComputeAB(Seq);
    WaitPool.AddThread(T_ComputeAB);
}

T_ComputeAB->arg.start = 0;
T_ComputeAB->arg.out[0] = 0;
(*T_ComputeAB)[0] = TDcppThread::Passed;
}
{
    Thread_dret* T_dret;
    T_dret = (Thread_dret*) WaitPool.FindThread(ID_dret, Seq);
    if(!T_dret){
        T_dret = new Thread_dret(Seq);
        WaitPool.AddThread(T_dret);
    }

    T_dret->arg.start = 0;
    T_dret->arg.out[0] = 0;
    (*T_dret)[0] = TDcppThread::Passed;
}
return 1;
}

unsigned Thread_ComputeAB::SeqNo;

char Thread_ComputeAB::ReturnThread()
{
{
    Thread_dret* T_dret;
    T_dret = (Thread_dret*) WaitPool.FindThread(ID_dret, Seq);
    if(!T_dret){
        T_dret = new Thread_dret(Seq);
        WaitPool.AddThread(T_dret);
    }

    T_dret->arg.result = ret.result;
    (*T_dret)[1] = TDcppThread::Ready;
}
return 1;
}

char Thread_ComputeAB::PassThread()
{
{
    Thread_dret* T_dret;
    T_dret = (Thread_dret*) WaitPool.FindThread(ID_dret, Seq);
    if(!T_dret){
        T_dret = new Thread_dret(Seq);
        WaitPool.AddThread(T_dret);
    }

    T_dret->arg.result = 0;

```

```

    (*T_dret)[1] = TDcppThread::Passed;
}
return 1;
}

unsigned Thread_dret::SeqNo;

char Thread_dret::ReturnThread()
{
    return 1;
}

char Thread_dret::PassThread()
{
    return 1;
}

unsigned Thread_ComputeRow::SeqNo;

char Thread_ComputeRow::ReturnThread()
{
    QueueItem_q2 *QItem_q2 = new QueueItem_q2;
    memcpy(QItem_q2, &ret, sizeof(ret));
    q2->Put(QItem_q2);
    return 1;
}

char Thread_ComputeRow::PassThread()
{
    return 1;
}

TDcppThread* QueueThread_q1()
{
    QueueItem_q1* Item = q1->Get();
    if(!Item) return NULL;
    Thread_ComputeRow* T_ComputeRow;
    T_ComputeRow = new Thread_ComputeRow(0);
    memcpy(&(T_ComputeRow->arg), Item, sizeof(*Item));
    delete Item;
    (*T_ComputeRow)[0] = TDcppThread::Ready;
    return T_ComputeRow;
}

TDcppQueue<QueueItem_q1> *q1;
TDcppQueue<QueueItem_q2> *q2;

QueueFunc ThreadQueues[] = {
    QueueThread_q1,
    NULL};

char CreateQueues(int Master)
{

```

```

if(Master){
    q1 = new TDcppQueue<QueueItem_q1>(130,"q1");
    q2 = new TDcppQueue<QueueItem_q2>(130,"q2");
} else {
    q1 = new TDcppQueue<QueueItem_q1>("q1");
    q2 = new TDcppQueue<QueueItem_q2>("q2");
}
return 1;
}

void DestroyQueues()
{
    delete q1;
    delete q2;
}

int Max_Thread = 3;
int Min_Thread = 3;

```

CALLER.H

```

#ifndef __CALLER_H
#define __CALLER_H

#include "dcpp.h"
#include "matrix.h"

#define ID_dmain 0U
#define ID_ComputeAB 2U
#define ID_dret 3U
#define ID_ComputeRow 6U

void Call_dmain(char *arg, int argsize, char *ret, int retsize);
void Call_ComputeAB(char *arg, int argsize, char *ret, int retsize);
void Call_dret(char *arg, int argsize, char *ret, int retsize);
void Call_ComputeRow(char *arg, int argsize, char *ret, int retsize);

extern TDcppQueue<QueueItem_q1> *q1;
extern TDcppQueue<QueueItem_q2> *q2;

TDcppThread* QueueThread_q1();

class Thread_dmain : public TDcppThread{
private:
    static unsigned SeqNo;
public:
    Arg_dmain arg;
    Ret_dmain ret;
    Thread_dmain(unsigned aSeqNo, int InLoop=0)
        : TDcppThread(ID_dmain, (InLoop ? aSeqNo : SeqNo++), 1,
            Call_dmain, (char*)&arg, sizeof(arg), (char*)&ret, sizeof(ret))
    {}
};

```

```

    {
    ;
    }
    virtual char ReturnThread();
    virtual char PassThread();
};

```

```

class Thread_ComputeAB : public TDcppThread{
private:
    static unsigned SeqNo;
public:
    Arg_ComputeAB arg;
    Ret_ComputeAB ret;
    Thread_ComputeAB(unsigned aSeqNo, int InLoop=0)
        : TDcppThread(ID_ComputeAB, (InLoop ? aSeqNo: SeqNo++), 3,
            Call_ComputeAB, (char*)&arg, sizeof(arg), (char*)&ret, sizeof(ret))
    {
        arg.q1 = q1; operator[](1) = Passed;    arg.q2 = q2; operator[](2) = Passed;
    }
    virtual char ReturnThread();
    virtual char PassThread();
};

```

```

class Thread_dret : public TDcppThread{
private:
    static unsigned SeqNo;
public:
    Arg_dret arg;
    Ret_dret ret;
    Thread_dret(unsigned aSeqNo, int InLoop=0)
        : TDcppThread(ID_dret, (InLoop ? aSeqNo: SeqNo++), 2,
            Call_dret, (char*)&arg, sizeof(arg), (char*)&ret, sizeof(ret))
    {
    ;
    }
    virtual char ReturnThread();
    virtual char PassThread();
};

```

```

class Thread_ComputeRow : public TDcppThread{
private:
    static unsigned SeqNo;
public:
    Arg_ComputeRow arg;
    Ret_ComputeRow ret;
    Thread_ComputeRow(unsigned aSeqNo, int InLoop=0)
        : TDcppThread(ID_ComputeRow, (InLoop ? aSeqNo: SeqNo++), 1,
            Call_ComputeRow, (char*)&arg, sizeof(arg), (char*)&ret, sizeof(ret))
    {
    ;
    }

```

```
}  
virtual char ReturnThread();  
virtual char PassThread();  
};
```

```
#endif
```



APPENDIX E

THE DCPD APPLICATION

The DCPD is developed by using DELPHI programming language. DELPHI is a visual programming language based on PASCAL programming language. The DELPHI is a useful tool for implementing GUI visually, and easily. For this reason it is chosen as the implementation language for the DCPD.

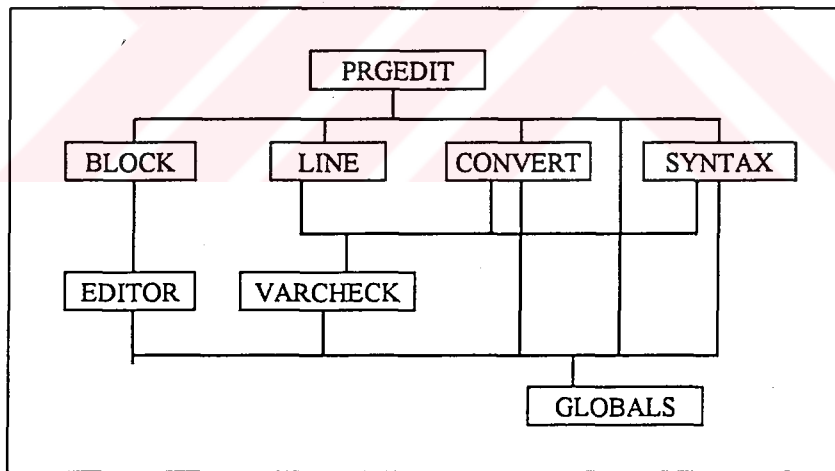


Figure E.1 The DCPD Modules

The DCPD is an object oriented application that is composed of eight DELPHI modules (*units*). Some of these modules have visual components

(forms), and some of them are the non-visual, utility procedures. In this appendix, these modules are presented.

In Figure E.1, the modules that constitute the DCPD application, and the interactions between them are shown. The following sections will introduce each module and their contents in detail. The DCPD application, DCPD application source codes, and the run-time systems are given in 3.5" floppy disk with this thesis.

E.1 PRGEDIT MODULE

PRGEDIT module defines the main form of the DCPD application. The main form is an instance of the class named *TPrgEditForm* (Figure E.2). This class is derived from the *TForm* component (a reusable component that is the ancestor of all the forms in DELPHI).

TPrgEditForm has a *Main Menu*, a *Speed Bar*, and a *Pop-up Menu* to accept the standard file related commands:

1. New Application
2. Open Application
3. Save Application
4. Save Application
5. Exit.

the commands related to program editing:

1. Add visual tools to program such as Block, Decision, Loop, Queue, and Line.
2. Remove a visual tool from the program.
3. Set a tool's identifier.
4. Edit the block's functions, set the item type of a queue, edit the condition expression for decision and loop tools.

the commands related to program conversion:

1. Syntax check of the program.
2. Conversion of the program into C++ sources.

This form captures the user commands (messages) from these menus (main menu, pop-up menu, and speed bar), and responds to them by performing that action. TPrgEditForm has also a program development area, to hold the program under development, and required message handlers (for windows messages) to draw, insert, delete, move, and edit the visual programming tools in it.

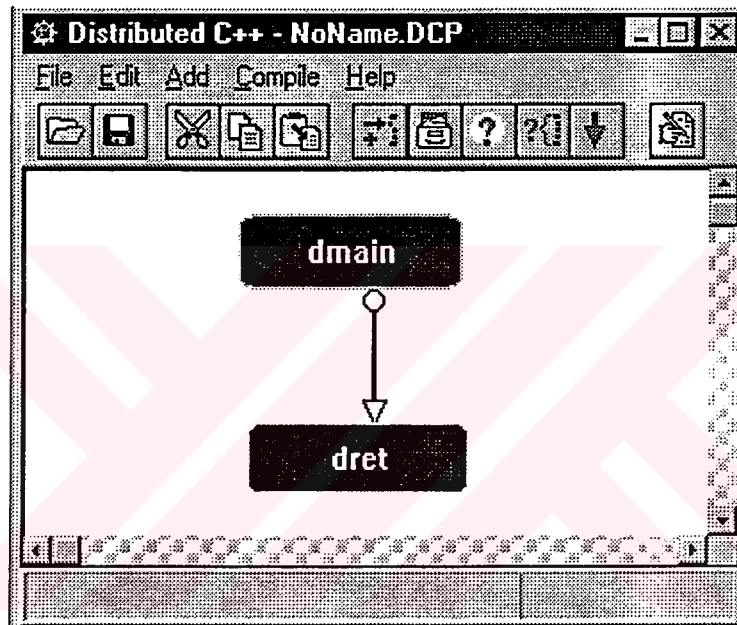


Figure E.2 TPrgEditForm Component

E.2 BLOCK MODULE

Block module defines the TBlock class. TBlock class is a DELPHI visual component that implements the DCPD visual programming tools block, queue, decision, and loop. That is, every block, queue, decision, and loop tool are the instances of the TBlock class.

Every TBlock instance has a BlockType field to identify the visual programming tool that it corresponds to. According to BlockType field, it displays itself on TPrgEditForm object, and responds to the windows messages directed to it. It also displays the block, decision, loop, or queue name on top of itself to identify itself to the programmer.

The TBlock class has the following fields to keep the related parameters:

1. Buf: Buf is a buffer that holds the function implementations for the blocks, condition expressions for the decision and loop tools, and queue item's record type for the queues.
2. Arg: Arg is a string variable that holds the argument definitions for the block, loop, and decision tools.
3. Ret: Ret is a string variable that holds the return value definitions for the block, loop, and decision tools.
4. LIn: LIn is an array variable that holds the list of lines that are going into itself.
5. LOut: LOut is an array variable that holds the list of lines that are leaving itself.
6. LTrue: LTrue is a line variable that holds the true line for decision, and loop tools, and create line for queue tools.
7. LFalse: LFalse is a line variable that holds the false line for decision, and loop tools, and return line for queue tools.
8. LCond: LCond is a line variable that holds the condition line for decision, and loop tools.
9. BlockName: BlockName is a string variable that holds the identifier for itself (block name, decision name, loop name, queue name).

The fields above defines everything about a block, decision, loop, and queue tool. Every visual tool keep the user defined data on these fields, and programmer can always edit these fields using the DCP's editors. These editors are the TFuncEditor (see section E.4) for editing functions, InputBox (standard windows dialog box for getting string inputs) for editing the condition expressions, and queue item's type, and TPrgEditForm(see section E.1) for the lines connected to itself. The textual editors can be activated by the mouse or TPrgEditForm menu commands.

These fields are also accessible from the TPrgEditForm. According to these fields TPrgEditForm can make syntax checking of the programs, and converts the programs into C++ sources.

E.3 LINE MODULE

Line module defines the TLine class. TLine class is a DELPHI visual component that implements the line tool of DCPD visual programming language. That is, every line in the program are the instances of TLine component.

Every TLine instance can represent different type of lines in the program. These types can be one of normal line, true line, false line, condition line, create line, and return line. Every different line type has different colors to identify themselves to the programmer. The normal lines have black, the true and create lines have green, false and return lines have red, and the condition lines have gray colors. According to line types, the lines draw themselves on TPrdEditForm, and connects the TBlock instances.

Every TLine instance have the following data fields:

1. Vars: Vars is a string variable and it keeps the argument (or return) variables carried by the line.
2. LStart: LStart field holds the TBlock instance which the line leaves.
3. LEnd: LEnd field holds the TBlock instance which the line goes into.

All the lines responds the mouse events click, and double click. Clicking the line with the mouse selects the line, and it is possible to work with that line by means of DCPD menu commands. It is possible to edit Vars field of the line by simply double clicking on the line. However, the true, false, create, and return lines are not editable. If the user wants to edit the other editable lines, an InputBox dialog appears, and user can edit the variables. Every editing operation is checked against the rules defined in Chapter 3, by calling the CheckVars function in VARCHHECK module (section E.7). The new variables are not accepted until they are properly defined.

E.4 EDITOR MODULE

The editor module defines the TFuncEditor class (Figure E.3). TFuncEditor is a DCPD class derived from the TForm class. TFuncEditor is a special editor to edit the function definitions of the block tools (instances of TBlock class). It simply displays a C++ function skeleton to the programmer. The function declarations, and return values etc. are automatically prepared by using the TBlock's data fields. Only the function body (white areas) is editable in this form. Therefore, programmers can easily implement the function body in this area.

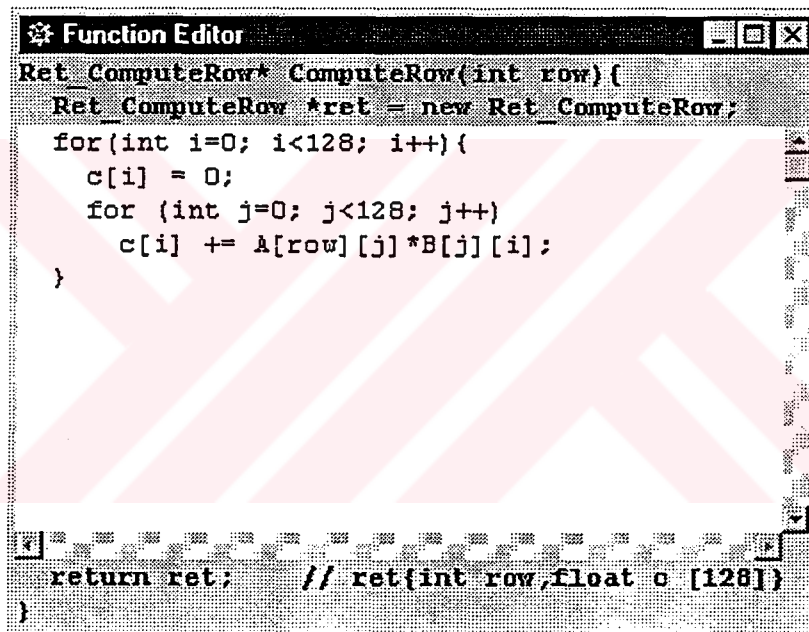


Figure E.3 TFuncEditor Form

The editor form always keep the active block's function definition, and is invisible to the user. The form is shown to the user whenever a block is double clicked by the mouse, or the edit command from the TPrgEditForm menus. Whenever the active block is changed, it saves the contents to the related block's Buf field, and loads the contents from the active block.

E.5 SYNTAX MODULE

The syntax module implements a function that is used for syntax checking the programs by TPrgEditForm. It has a single interface function called Syntax. Syntax function checks the syntax of the programs according to the rules in Chapter 3, and if there is no error it returns true. If Syntax function detects an error, it returns false, and sets the global variables SyntaxErrStr, SyntaxErrBlock, SyntaxErrLine to indicate the description and the location of the error. The interface function, and the global variables are defined as:

```
function Syntax(Lines, Blocks: TList, FullName: string);  
var  
    SyntaxErrStr: string;  
    SyntaxErrBlock: TBlock;  
    SyntaxErrLine: TLine;
```

E.6 CONVERT MODULE

The convert module implements the conversion process of the programs into C++ language. This process is described in detail in Chapter 4. There is one function called Make in the interface of this module. The Make function takes the lines, blocks, and the full name of the program as arguments. It produces the program module, and caller module for the given program. This function is called by TPrgEditForm in response to the convert command. It writes its output to disk as *FullName.cpp*, *FullName.h*, *Caller.cpp*, and *Caller.h*. The definition of the interface function is:

```
procedure Make(Lines, Blocks: TList; FullName: string);
```

E.7 VARCHHECK MODULE

This module is used by the Line, Syntax, and Convert modules to process the variable declarations for the lines, and queue item types. The CheckVars function is used to determine whether the given string is a valid declaration according to rules in Chapter 3. The IsIdentifier function is used to determine whether a given string is a valid C++ argument or not. The Normalize

function is used to format the variable declaration so that the ExtractDecl, and ExtractVars functions can work with it. ExtractDecl function is used to extract a variable declaration from a given variable definition, and deletes that declaration from the argument string (to enable iterations). Similarly, the ExtractVar function extracts a variable name from a given variable declaration string, and deletes the related declaration from the argument string (to enable iterations). The interface functions are defined as:

```
function CheckVars(var vars: string): Boolean;  
  
function ExtractDecl(vars: string; var vartype, variable: string;  
    var dim: array of Integer): string;  
  
function ExtractVar(var vars: string): string;  
  
function Normalize(s: string): string;  
  
function IsIdentifier(s: string): Boolean;
```

E.8 GLOBALS MODULE

The globals module includes the global variables, utility functions and the program constants that are used by the other modules in the DCP application.