TOPOLOGIES: DISTRIBUTED OBJECTS IN MULTICOMPUTERS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

By

Win Bo, B.E., M.S.

* * * * *

The Ohio State University

1989

Dissertation Committee:

Dr. Karsten Schwan

Dr. Bruce Weide

Dr. Ponnuswamy Sadayappan

Dr. Richard Parent

Approved by

Bruce W. Weide

Adviser

Department of

Computer and Information Science

To My Parents, U Ba Swe and Daw Ma Ma, My Wife, Nge, and My Son, Kyaw

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Karsten Schwan, whose encouragement, guidance, understanding, and substantial aid in thesis writing, made this work possible. His advice and assistance have been immensely valuable throughout the entire period of study and research. I would also like to thank Dr. Bruce Weide whose comments on the thesis have been very beneficial and greatly appreciated.

Next, I would like to thank my parents, U Ba Swe and Daw Ma Ma, who cultivated in me the value of education and continually encouraged and supported my pursuit of it. I thank my wife, Nge, for sharing the frustrations, tolerating several lonely nights, and for continually encouraging me throughout the difficult times.

I would also like to thank other members of the research group who have helped with many suggestions and have answered many of my questions. Some of the work described in this thesis involved contributions by Nena Bauman, John Gawkowski, Ben Blake, Rajiv Ramnath, Steve Martin, Fikret Ercal, Cevdet Aykanat, and Siva Monhanachandran. Also, Paul Pierce at Intel Scientific Computers had offered several helpful tips for implementation.

This research would have been impossible without the sponsorship and support of Universal Energy Systems, the Thomas Alva Edison Program of the State of Ohio, the Intel Corporation, and the Department of Computer and Information Science at the Ohio State University.

VITA

August 9, 1953 . . .Born - Mandalay, Burma

1977 . . . . BE, The Department of Electrical Engineering,
Rangoon Institute of Technology, Rangoon, Burma

1981 . . . . MS, The Department of Computer and Information
Science, The Ohio State University, Columbus, Ohio

1985-Present . . . . PhD Student, The Department of Computer and Information
Science, The Ohio State University, Columbus, Ohio

PUBLICATIONS

1. *"OS Primitives for Implementing Distributed Objects: An Experiment in Parallel Branch-and-Bound Algorithm"*,Proceedings of the 4th. Intl. Conf. on Hypercube Concurrent Computers and Applications, March, 1989 (With Karsten Schwan, Ben Blake, and John Gawkowski).
2. *"Topologies – Computational Messaging for Structured Multicomputers"*, Proceedings of the 3rd. Intl. Conf. on Hypercubes Concurrent Computers and Applications, Jan., 1988 (With Karsten Schwan).
3. *"Mapping Parallel Applications to Hypercubes"*, Proceedings of the 2nd. Intl. Conf. on Hypercube Concurrent Computers and Applications, Sept, 1986 (With Karsten Schwan, P. Sadayappan, F. Ercal, and N. Bauman).

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Operating Systems:
Dr. Karsten Schwan

Studies in Computer Architecure:
Dr. P. Sadayappan

Studies in VLSI and VLSI Algorithms:
Dr. T. H. Lai

Table of Contents

DEDICATION ii
ACKNOWLEDGEMENTS iii
VITA v
LIST OF TABLES ix
LIST OF FIGURES xi

CHAPTER PAGE

I Introduction 1
1.1 Parallel Programs on Structured Multicomputers 1
1.2 Topologies—Distributed Objects in Multicomputers . . . 3
1.3 Motivation 5
1.4 Desirable Properties of a Topology's Communication Structure 9
1.5 Usefulness of Topologies 11
1.6 The Thesis 12
1.7 Contributions 13
1.8 Scope, Goals, and Dissertation Overview 13

II Survey of Related Work 15
2.1 IPC Structures 15
2.2 IPC Support On Structured Multicomputers 21
2.3 Activeness in Message-based IPCs 23
2.4 Novel IPC Semantics 26
2.5 Programming Environments and Program Visualization 27
2.6 Conclusions 31

III Decomposition Methods: Global Data and Operations 32
3.1 Domain Decomposition and Global Data 32
3.2 Domain Decomposition and Global Operations 33
3.3 Divide and Conquer and the Task/Queue Model 34
3.4 Functional Decomposition and Pipelining 35

3.5 Multiple Phases 36
3.6 Global Data and Operations in Operating Systems 36
3.7 Conclusions 39

IV Experimental Testbed and Sample Parallel Applications 40
4.1 Testbed Environment 40
4.1.1 Hardware 40
4.1.2 Software 43
4.2 A Finite Element Modeling Program 44
4.3 A Traveling Salesperson Program 48
4.4 Conclusions 53

V Topologies 54
5.1 Topologies—Basic Definition 54
5.1.1 Services (Object Operations) 62
5.1.2 Topologies—Extended Definition 63
5.2 Topologies—Instantiation and Binding 64
5.3 Programming with Distributed Objects 67
5.3.1 Using Topologies 67
5.3.2 Writing Services 70
5.4 Using Topologies to Program Global Data and Operations 71
5.4.1 The FEM Application 71
5.4.2 The TSP Application 77
5.5 Conclusions 84

VI Implementation and Evaluation of Topologies 86
6.1 Implementation of Topologies 86
6.2 Performance and Implementation of Topology System Calls 91
6.3 Performance of Topologies Spanning Multiple Processes 94
6.3.1 A Small Synchronous Topology 95
6.3.2 Asynchronous Topologies 97
6.3.3 Larger Synchronous Topologies 100
6.4 Performance and Usefulness of Topologies with Application Programs 104
6.4.1 The FEM Application. 104
6.4.2 The TSP Application. 106
6.5 Conclusions 110

VII PRISM—A Programming Environment for Topologies 111
7.1 A Structured Multicomputer Programming Environment 111
7.1.1 System Requirements 111

7.1.2 Basic Features of PRISM 113
7.2 Programming Topologies 117
7.2.1 Topology Compiler 117
7.2.2 Mapping of a Topology 119
7.2.3 Loader 120
7.2.4 A Vertex-monitoring Utility 121
7.2.5 Support System Calls for Services 122
7.3 Manipulation of Programmed Topologies 123
7.3.1 The Database Model and Interface 125
7.3.2 Entities, Relationships, and Sets 126
7.3.3 Action Routines 127
7.3.4 Database Operations 128
7.4 Topologies and Their Graphical Views in E-R Model . . 134
7.4.1 Representing Vertices and Edges 135
7.4.2 Representing Processors and Communication Links 136
7.4.3 Representing Mapping of a Topology to Processors 137
7.4.4 Representing Graphical Icons 137
7.4.5 Mapping E-, R-, and S-Entities to Graphical Entities 141
7.4.6 Estimating Database Size 144
7.5 Implementation and Evaluation of PRISM 147
7.5.1 Implementation of the PRISM Database 147
7.5.2 Implementation of the DML Programs 149
7.6 Conclusions 161

VIII Conclusions and Future Research **162**
8.1 Review of the Goals 162
8.2 Summary of Lessons Learned 163
8.3 Future Research 166

APPENDICES PAGE
A System Calls for Services **169**
A.1 Detailed Descriptions 171

B Grammar of Topology Description Language **180**
BIBLIOGRAPHY 183

List of Tables

6.1 Latency of TopSend System Call 93

6.2 Latency of TopRecv System Call 93

6.3 A Linear Topology vs. iPSC Messages (nonlocal communications) . 96

6.4 A Linear Topology vs. iPSC Messages (local communications) . . . 96

6.5 A Ring Topology vs. User-level Ring 98

6.6 Latencies of User-memory Update 100

6.7 A Tree Topology vs. User-level Tree 101

6.8 Best and Worst Performances of the Global-sum Topology 103

6.9 Solution Times for the FEM Application 105

6.10 Maximum Retransmission Counts of the FEM Application 106

6.11 Performance of A Shared-memory Ring Topology 107

6.12 Completion Times of the TSP Application 108

7.1 Estimated Database Storage Requirements 148

List of Figures

1.1 A Distributed Queue Object 5

4.1 The Hypercube Interconnection 41

4.2 Testbed—Cube Manager and the Hypercube 43

4.3 Node Architecture 44

4.4 Strip Mapping and the Linear Chain of Processors 46

4.5 Three-level Process Structure for Solving the TSP 50

5.1 Topology—A Distributed Object 55

5.2 Broadcast and Inverse-broadcast Trees 71

5.3 Shared-memory Ring Topology 78

6.1 Extended Node OS Kernel 88

6.2 Demultiplexing a Topology Packet 89

7.1 The PRISM Programming Environment 114

7.2 Logical and Mapped Ring 120

7.3 Topology Manipulation Tools 124

7.4 High-level and Low-level Tools Integrated Through the Database . . 125

7.5 Different Representations of a Vertex and an Edge. 154

7.6 D5 Hypercube 155

7.7 A Spanning Tree Topology 156

7.8 A Ring Topology 156

7.9 Subtree of Vertex Id 1 158

7.10 Leave Vertices at 2nd Level of the Tree 158

7.11 Vertices Mapped to a Hypercube 159

7.12 Logical Tree Structure of Processors 160

7.13 Logical Ring Structure of Processors 160

CHAPTER I

Introduction

1.1 Parallel Programs on Structured Multicomputers

Structured multicomputers such as hypercubes and toroidal or pyramidal multicomputers are gaining wide acceptance, in part due to a natural match of their physical interconnection structures to the logical communication structures of many scientific applications [25, 80, 10, 60]. However, current multicomputers are hard to program for three reasons:

1. Any parallel program with a large number of tasks and communication channels is difficult to design, implement, and debug [35].

2. Multicomputers such as hypercubes are harder to program than shared-memory multiprocessors because programmers have to construct complex communication structures such as trees or rings spanning many application processes and hypercube nodes. Such structures may be inherent parts of an application's computation, or they may be auxiliary structures superimposed

[38] by the programmer or used by the operating system [33] for purposes of maintaining multiple program views for program control, monitoring, debugging, or other purposes [74, 48, 75, 81]. In current multicomputer operating systems, such structures must be built from OS primitives and physical communication links that connect only two nodes at a time.

3. One issue in the programming of multicomputers is that (as with all other parallel applications) good speedup of execution by use of parallelism cannot be attained unless efficient use is made of the multicomputer's distributed processors and memories. This requires decomposition of the application program into multiple, independently executable processes such that global data and control, and therefore interprocess communication, are minimized [75, 35, 25].

This dissertation introduces an operating system and programming construct—termed *topology*—that allows programmers to build distributed objects that may internally consist of complex computational communication structures. It provides a versatile mechanism for interprocess communication and structuring of tasks. As an interprocess communication mechanism (IPC) it allows communications among multiple tasks transparent to the user program. As a structuring mechanism it provides an object-oriented view of shared resources in the multicomputer. It may implement distributed objects performing a wide variety of functions, including the exchange of data or control information relevant to the tasks' computations and/or communications required for task synchronization, message forwarding and filtering under program control, and others.

1.2 Topologies—Distributed Objects in Multicomputers

As a widely accepted paradigm, the *object* model has been used for designing operating systems, languages, and architectures. It is attractive for its ability to provide abstractions and a uniform and simple model of programming. A distributed object, similar to the conventional object, provides abstractions for parallel and distributed programs and consists of a distributed data representation and a set of operations that can be performed on an instance of the object. For instance, a distributed queue object may be abstracted as a single queue consisting of queue elements distributed across several processors and insert and delete operations performed in a parallel and cooperative manner on an instance of the queue. The operations can be implemented to enforce different queuing disciplines, such as a priority queue or a first-in-first-out queue.

Topologies are distributed objects implementing abstractions on a structured multicomputer such as a hypercube. A topology consists of a distributed data type and a set of operations concurrently executable on the multicomputer's processors. However, unlike object implementations on network architectures [46, 2, 67] or other multiprocessors [22, 7, 73, 28], it enforces the distribution of data representing an object and of the object's operations to match the architecture and the parallel applications using the object. Distribution is expressed in terms of explicitly defined interconnections of a set of nodes implementing the object's data and operations, hence the term "topology".

Each topology implements a distributed object as an arbitrary communication graph linking multiple nodes, which in turn may be attached to specific processors of the parallel program. The operations of a topology are performed cooperatively by the nodes linked by the graph. Specific attributes of a topology are

- *multiple connectivity*—multiple nodes may be linked by a communication structure defined by the graph,
- *activeness*—a topology employs an active message-passing mechanism that performs distributed computations, and
- *separation of concerns*—topologies can be constructed independently of the applications using them.

As a simple example, a distributed queue object among three producers and one consumer, each on a different processor, can be implemented as a tree topology with the root node (consumer) and three children (producers) (Figure 1.1). In the figure the gray rectangular area denotes the queue object, consisting of directed edges representing the communication links, and the black dots representing the tree's nodes, where distributed data (queue elements) are stored and distributed

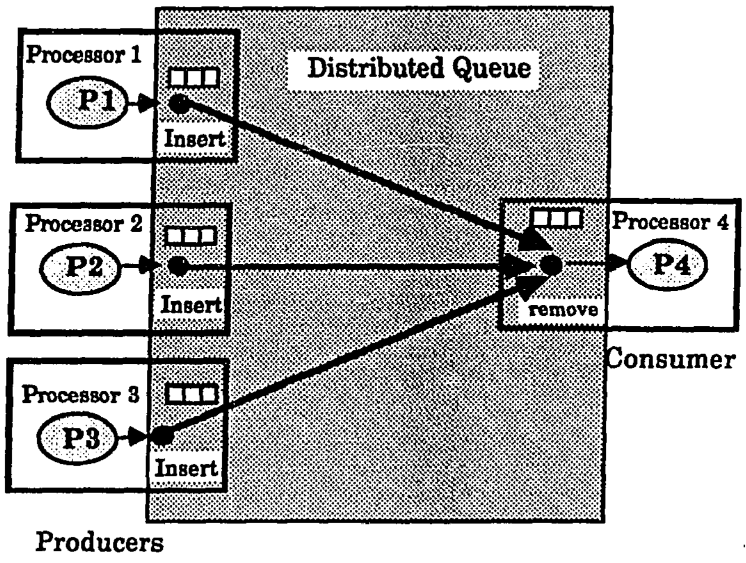


Figure 1.1: A Distributed Queue Object

operations (such as insert and remove) are executed. Parallel insertions by producers are not ordered, and therefore each node attached to a producer either buffers the data locally or forwards it to the consumer. Removal by the consumer retrieves data buffered locally or on one of the producers' processors. The routines to perform insert operations are distributed and replicated on the producers' processors so they can be executed in parallel. Thus, the tree topology provides the abstraction of a *queue*, but its structure and internal interprocess communication are invisible to the user of the queue.

1.3 Motivation

As stated in the previous section, topologies differ from other implementations of distributed objects in that they are distributed implementations of objects; data and operations in each topology have a well-defined communication structure which is crucial to its performance. Most parallel systems do not provide extensive support for such explicitly structured abstractions. A set of processes in a parallel application either communicates in an unstructured manner, or the communication structure is hidden in the processes' implementations. Some exceptions are the Hierarchical Process Composition (HPC) system [45] where an object-oriented model is used to specify the connections between processes and Thoth [14] where a process may have several sub-processes within its address space. However, in HPC, communication structures are composed of low-level connections that are incapable of performing computations. In Thoth, high level descriptions of the communication structures are not represented as long-term connections among processes.

Difficulty in structuring mainly stems from the currently available IPC mechanisms in multicomputers. Existing mechanisms are based on a message passing between two processes. In general, there are two basic communication primitives: *send* and *receive.* The *send* primitive specifies the destination process identifier and a message. The *receive* primitive specifies the sender process identifier and the buffer to receive the message. One process executes a send call while the other executes a matching receive call. The process identifiers of both sender and receiver must be globally known to use these primitives.

The premise of the topology concept is that the communication structures within a distributed object on a multicomputer should not be hidden from their implementor. Instead, such structures should be explicitly programmable as separable units in implementing the objects.

Although topologies define communication structures among several tasks, they differ from other IPC's in several ways. First, conventional message-based IPC's provide general-purpose channels with which a variety of functions may be implemented. In contrast, topologies are abstractions implementing well-defined operations. For instance, a tree structure of processes can form a logical multicast bus, allowing values from one process (the root) to send values to the remaining processes. A topology describes such a structure as a *multicast bus object* offering the operations *multicast-send* and *multicast-receive.* The implementation of such a structure using IPC constructs would require that addressing information be maintained throughout the processes using the structure. Such low-level information is not visible to the users of a topology. Instead, a topology has a global name and the semantics of its operations determine which of the processes invoking it are affected by the operations' executions.

A second difference between topologies and conventional IPC constructs concerns the implementation of a topology's operations. With conventional IPC constructs, the precise semantics of *send* and *receive* are determined at operating system design time. This may not provide a convenient or efficient basis for the implementation of the specific semantics required by an application. For example, it may be impossible to route messages using an application-specific routing algorithm or to terminate the message forwarding based on the state of a particular process participating in the IPC. In contrast, for a topology, the ability to perform computations in conjunction with the messages transferred on its communication links facilitates implementation of a wide range of link semantics.

Since topologies may be used to implement arbitrary, distributed abstract data types, their efficient implementation requires support by low levels of the operating system. Efficiency is gained by avoiding unnecessary layers of protocol [85]. In addition, low-level support must be provided such that message-sending and -receiving semantics remain variable as with packet filters in Mach [1]. An example of such variability is the implementation of a user-defined routing algorithm as part of a topology.

Two additional benefits arise from the decision to support topologies at the operating system level. First, once a topology has been constructed, it can be shared by multiple applications and it can be reused. The client thus treats it like any other operating system resource. Second, language-independent interfaces to topologies encourage their use in a wide variety of parallel applications.

As with the programming of communication protocols, programming topologies can be complex. For instance, it is difficult to debug a large structure mapped onto a hypercube of dimension 4 or more partly because it is hard to visualize the structure. Complexity in debugging may be compounded by the fact that computations are performed on messages in transit which makes it difficult to keep track of partial or incomplete results. Thus, an effective use of topologies in the development of parallel applications requires support tools for topology specification, loading, and debugging. In addition, manipulation of topologies' structures and selection of sub-structures based on certain attributes should be facilitated. Such manipulation should be provided in conjunction with tools for the visualization of multiple views of topologies. The development and evaluation of tools for topology visualization and manipulation for debugging is the second thrust of this thesis.

To conclude, we note that the communication structure of a topology depends on its intended use by a parallel application. Thus, it may not be identical to the structure of the underlying multicomputer. The efficient mapping of the topology to the multicomputer [64] is application dependent and is left to the applications programmer.

1.4 Desirable Properties of a Topology's Communication Structure

To provide abstractions through the distributed object model, a topology's communication structure must be powerful enough to implement a wide range of functionalities required in the distributed objects. An ideal communication structure should have the following properties:

1. *Flexibility*—Efficiency and ease of abstraction require that the composite structure of communication links should be changeable. It should support simple one-to-one connections to a regular structure such as a mesh, a tree, or arbitrary graphs. The structure should be scalable with the number of processes linked by it and should not depend on the underlying hardware. Furthermore, it should be able to be created and deleted dynamically, it should be possible to add and delete communication links dynamically, and links should be migratable from one processor to another.

2. *Variable semantics of link protocols*—Ease of representation of arbitrary, distributed operations requires that the semantics of each communication link should be variable along the following dimensions:

    (a) protocol attributes such as time-out periods, flow control, or acknowledgments;

    (b) send/receive semantics such as blocking/nonblocking, buffering/nonbuffering, or return/no return information;

    (c) multiplexing of logical onto physical communication channels;

    (d) typed/untyped logical channels; and

    (e) implicit logical channels that accept and forward (to the intended recipient) a message even when there is no matching *receive* operation.

3. *Ability to map communication structures to physical machines*—Independence of physical communication structures requires that topologies with arbitrary structures be mappable to the physical hardware.

4. *Ease of programming/debugging*—The programming of communication structures and of the computations associated with the structures should be straightforward. In addition, it requires the availability of debugging and powerful program visualization tools since communication structures may span a large number of processes and processors. For debugging, the state of communication structures and the messages in these structures must be monitorable and controllable. This implies that communications structures should be compositional such that complex structures may be formed as compositions of multiple, simple structures.

1.5 Usefulness of Topologies

Topologies are useful and important elements of any parallel application on a multicomputer. For instance, a linear topology may act as a simple communication link object connecting two processes. In this case, data is not manipulated but simply transferred. Alternatively, a broadcasting object may be implemented as a binary-tree topology, and this tree may be used to broadcast values sent by the root process to all connected processes. In this case, the computations performed by each link consist of buffering and forwarding the values at intermediate nodes.

The simple examples above demonstrate the usefulness of topologies. More importantly, however, topologies are a necessary element of most parallel applications because they can efficiently represent the global data and operations required by the application. It is well known[76, 35] that the appropriate implementation of global data and operations in parallel programs is important for several reasons:

1. Global data or operations are unavoidable. Any decomposition method for a serial program will require the resulting parallel program's use of some global operations or data as shown in Chapter 3.

2. Global data and operations significantly affect performance. Application programs may spend significant amounts of time in nonlocal program portions[35, 76]. More generally, in the absence of other serial phases of a parallel program, sequential implementations of global data and operations limit the parallel program's maximal possible speedup to $1/a$, where a is the fraction of the program's total execution time consumed by global data access or global operation execution (a version of Amdahl's law).

1.6 The Thesis

The thesis of this dissertation is that topologies are useful and desirable constructs for programming parallel programs on multicomputers. Furthermore, they are efficient and appropriate for programming global data and operations.

1.7 Contributions

The major contributions of this thesis are

1. the development of topologies, which are novel high-level constructs for the specification and implementation of distributed objects in a parallel program, and

2. the demonstration of the ease of programming of topologies by the provision of a set of tools for their programming and debugging, and examples of their use.

1.8 Scope, Goals, and Dissertation Overview

This dissertation focuses on two issues—the runtime support for topologies and the ease of programming of topologies. Although some of the design decisions resulted from the specific characteristics of *nex*, the Intel iPSC/1 hypercube's node operating system, the basic concept of topologies is readily applicable to other parallel systems.

To prove the thesis, this dissertation addresses three major topics:

1. **Desirability of topologies.** Chapter 3 and 4 show that topologies are desirable primitives for the interconnection of parallel tasks. Chapter 3 describes various methods of decomposition of parallel applications; each of which results in global data and operations that are efficiently programmed

with topologies. Chapter 4 describes two typical parallel applications and identifies their global data and operations, and Chapter 5 shows how topologies are conveniently used for programming them.

2. **Appropriateness of the topologies construct with respect to implementation.** Chapter 5 describes the topology construct and its components, and Chapter 6 describes the implementation and performance evaluation of the construct. These two chapters show that the construct can be built and used and that it is efficient. Thus, it is a realistic primitive.

3. **Ease of programming of topologies.** Chapter 6 describes the tools and facilities for specifying, compiling, and loading topologies, and it shows that topologies are easily programmable demonstrating tools for the visualization of topologies for debugging and monitoring.

CHAPTER II

Survey of Related Work

A topology is characterized by (i) multiple process interconnection, (ii) its active message-passing mechanism, and (iii) its capability to be constructed and programmed separately from application programs. In this chapter, other research related to these aspects of topologies is described. Accordingly, the chapter is organized as three sections: (i) process connection and IPC structures, (ii) IPC support on multicomputers, and (iii) programming environment and program visualization support for software components.

2.1 IPC Structures

Few systems offer IPC constructs that can establish entire IPC structures connecting many processes. However, simpler IPC constructs are an integral part of many operating systems. Those systems that offer some support for the establishment of IPC structures are reviewed here.

Demos' Switchboard Process. In the Demos Operating System [5] process connections are established by means of a *switchboard* process. Initially, each process only possesses a link to the switchboard process. A link is a simplex communication path that is addressable by a link identifier and can be created and destroyed dynamically. When a process wishes to communicate with another named process, it sends a message to the switchboard process. The switchboard process matches pairs of processes based on the name specified and passes the link from one process to another. This allows a process to form a connection with another consenting process. While complex structures may be built in this fashion, they must be constructed explicitly from many process-to-process connections as is the case with IPC constructs in other research and commercial operating systems such as StarOS [33], Medusa [59], Argus [46], Mach [87], and Unix [63].

Unlike Demos' IPCs which perform several process-to-process connections through a switchboard process to create an IPC structure, the communication structure of a topology is set up as a whole, independent of the processes using the topology.

SMP's IPC Structures. SMP [44] is a message-based programming environment that supports the construction of process families—a set of processes that communicate via asynchronous messages according to a given interconnection structure and naming scheme. It is very similar to topologies in that arbitrary structuring is allowed. The interconnections within the family are defined in a special data structure called *topology*, which is provided during process creation. This data structure is used by the SMP implementation to allocate buffers and to enforce the interconnection restrictions. Communications employ the family name and a list of family members to specify message recipients. Similarly, the recipients use the family name and list of family members to identify the messages to be received.

Enforcing a process structure based on a given interconnection pattern is similar for both SMP and topologies. But naming differs in that SMP requires both the name of the family of processes and the list of family members while topologies only require the structural name. Also SMP channels are passive while topologies have links that can perform computations.

HPC's IPC Structures. HPC—Hierarchical Process Composition [45], is a design that supports hierarchical communication structures. Each process is defined as an object consisting of an internal state, an executable code, and a set of communication interfaces. A composite object is constructed by a combination of already created objects, communication channels among those objects, an encapsulation *shell*, and a set of interfaces to the external world. Object connections are unidirectional channels, and they are established by connect calls on the objects' local interfaces. The location and identity of the partners of such connections are transparent. Connect calls are executed and channels are created by third parties called *controllers.* The controller associated with each shell modifies and maintains the arbitrary connections among the processes or objects within the shell and enforces protection within the shell. Among shells, communication structures are hierarchical since shells are composed hierarchically.

HPC's IPC structures differ from topologies in that they are hierarchical, resulting in tree structures, whereas topologies can have arbitrary graph structures. One interesting aspect of HPC is the semantics of its communication interface. A communication interface can be a simple, a bundle (where a number of channels may be combined), a multicast or a multiplex interface. These semantics allow one-to-one, many-to-one, one-to-many, and many-to-many connections among objects. These semantics are fixed and cannot be modified whereas the semantics of the interface for topologies to an application process may be varied arbitrarily.

Charlotte's Links and Connector. Charlotte [3] is a distributed operating system with a communication package called *nugget* which provides process control and IPC. Its IPC provides full duplex links addressable by a link identifier and employs nonblocking send and receive calls. It can also cancel a send or a receive and destroy a link or transfer a link from one process to another. A process connection is established by two utility processes—Switchboard and Connector. The Switchboard helps client processes find server processes. Given a description of the processes and the links connecting them, the Connector searches for the process in the Switchboard, creates a new process from a given program file if it is not found, and sets up the links. An interesting aspect of Charlotte's IPC is the ability to cancel a message after it is sent and to perform link migration. However, Charlotte does not provide any facilities for connecting a set of processes or for addressing such a set as a single unit as in topologies.

Although Charlotte provides several interesting aspects, such as its ability to cancel a message after it is sent and link migration, it does not provide any facilities for connecting a set of processes or for addressing such a set as a single unit as topologies can. In addition, as in Demos and unlike topologies, Charlotte has a switchboard process to help establish the communication links.

Thoth's Multi-process Structuring. In Cheriton's Thoth system [14] a concurrent program is described at 3 levels: procedure call, data abstraction, and process. Thoth processes communicate via messages, and they may be grouped into teams, each consisting of a proprietor process which controls access to a resource and a set of helper processes (subprocesses) executing within the same address space as the proprietor, much like threads in Mach [1]. Connections between processes are implicit and short term, lasting from the time the sender sends a message to the time the receiver replies to it. During this connection, the receiver may transfer data to or from the sender or send a reply to the sender without blocking. Dynamic process creation and destruction allow the process structure to change dynamically.

In the Thoth System, the connections last until the receiver sends a reply message to the sender. Thus, the connections are short term and they are not reusable by another set of processes. In contrast, connections in the topologies are long term.

Other Systems. The summary of related research presented above is not complete regarding designs of systems offering IPC constructs. They are merely a representative sample of modern IPC systems. Implicit communication structures exist in IPCs which use a process identifier for addressing. One such facility is broadcasting and multicasting in Intel's iPSC/1 nex OS. It provides limited broadcasting and multicasting on the hypercube using process identifiers. The restriction is that the receiving processes must have the same process identifier and must be on a sub-cube including the sending node.

Several other systems exist which support the connection of multiple processes. A multi-RPC facility [66] allows an RPC call to be multicasted to several servers with at-least-once semantics. In Cooper's Replicated Distributed Programs [16], Troupes form a group of replicated programs addressable by a single name. Also in Cheriton's V System, V process groups allow multicasting to a set of processes in a group using a group identifier. Caltech's MOOSE operating system has included two levels of process grouping in its design: teams—a collection of processes, and leagues—a collection of teams. These systems either offer a simple communication structure or have no explicitly defined structures at all.

2.2 IPC Support On Structured Multicomputers

Several of the currently available operating systems on structured multicomputers and on the hypercube in particular provide IPC as send and receive calls. These are low-level calls using as the process name a <processor identifier, process identifier> pair. The process name is global, and routing is simplified by the symmetry of the hardware topology. IPC's of some of the current operating systems are summarized below:

CrOS. The Crystalline Operating System (CrOS) [25] is an operating system for Caltech's Mark II and Intel iPSC hypercubes and is most suitable for algorithms that exhibit regular communication structures. Basic calls in CrOS II are *rdelt*, which receives a communication packet from a (physical) channel; *wtelt*, which sends a packet through (physical) channel; *rdres*, which receives a unique packet from the host; and *sendres*, which sends a unique packet to the host. In CrOS III, basic calls are *cread* and *cwrite*, which are similar to *rdelt* and *wtelt* respectively. They are all synchronous calls, and the operating system provides no routing of the packets. Special calls such as *pass*, *shift*, and *pipe* in CrOS II and *cshift*, *vshift*, *broadcast*, *combine*, and *concat* in CrOS III are provided to assist routing which must be performed by the node program. These calls are also synchronous and must be executed by all the node programs at roughly the same time. This forces the complete parallel program to be written in a synchronous style, whereas all communications are done in a lock-step fashion. It is a very low-level operating system with little concern for programmability.

MOOSE. MOOSE [65] is an operating system implemented on Caltech Mark II hypercubes for automatic load balancing. It provides multi-tasking, dynamic creation of processes, and priority-based scheduling. The IPC mechanism is *pipes* in the flavor of UNIX although the actual implementation is more like mailboxes with fixed record-length entries. A pipe is created by the system call *pipopen* with a depository on the named processor. System calls *pread* and *pwrite* can be used for exchange of data using pipes. Shared memory may be used if the processes are on the same processor. MOOSE also provides globally accessible semaphores for locking and synchronization.

Reactive Kernel. Reactive kernel [79] is an operating system for the Ametek Series 2010 toroidal multicomputer which, together with the Cosmic Programming Environment, addresses portability of the operating system. It consists of a set of daemon processes, utility programs, and libraries. The IPC is based on a message-passing model using process identifiers. The kernel is reactive or message driven since it schedules a process when a message it is waiting for has arrived, and it does not force a context switch until some system exception occurs or until it is blocked for another message. Another interesting aspect of its IPC is that messages are sent and received using buffers from a message heap. These buffers are explicitly allocated and deallocated for sending. A receive call returns a pointer to the buffer, avoiding buffer copy completely. Although sending and receiving messages occur between two processes only, it does allow multicasting by providing a list of destination processes. The reactive kernel has a notion of process groups, but it was only intended for cube sharing in terms of space rather than manipulating a set of processes as a unit.

2.3 Activeness in Message-based IPCs

Operations in the distributed objects implemented by topologies can be executed in parallel, in a cooperative manner. Such cooperation is implemented with active message-passing. Active message-passing has been used in several related research efforts.

RPC and Remote Invocation. Nelson's remote procedure call (RPC) [9] is an IPC mechanism for which the manner of association of application-dependent computations with IPC actions is particularly obvious. In addition, some changes in RPC semantics are easily made, as shown by Spector's work on remote memory operations [85], by Schwan's work on remote object invocation with varying invocation semantics [73], and by similar recent research at the University of Washington in the Presto Programming Environment [7]. Most of this research is similar in spirit to topologies in that it avoids the strict layering of communication protocols, thereby facilitating the efficient association of application-dependent computations with communications. In fact, a simple topology with a link from the caller to the remote node and a return link to the caller constitutes an implementation of a remote procedure call.

Active Channels. Linvy and Manber's work on active channel [47] in Active TOken ring Network (ATON) is also related to active links in topologies. Here the token ring protocol is extended so that shift arithmetic operations may be performed directly on the node interfaces. Operations consist of 3 groups: arithmetic, selection, and counting. Each operation is executed as a sequence of 1 bit operations by the ATON interface hardware consisting of a modem, an input state machine, an output state machine, and a 1-bit ALU. This makes active channels highly efficient for implementing a number of algorithms for determining global information such as minimum load on the ring network.

Fetch-and-Φ. Another primitive similar in spirit to topologies is the Ultracomputer's fetch-and-Φ construct [76]. Although fetch-and-Φ operations are designed for accessing shared locations on shared-memory machines, they can be viewed as specialized topologies for process synchronization, where topologies are implemented as computational structures embedded in the machine's combining network. The queueing of memory requests and performing operations, as defined by Φ functions implemented in the combining switches, are in essence identical to similar operations performed in topologies. In fact, Φ exactly defines the computations of analogous topologies' links.

Packet Filters. Packet filters [49] in the Mach operating system are another means of associating application-dependent operations with the IPC. A filter is a kernel- resident packet demultiplexer that can distinguish between packets according to arbitrary and dynamically variable user-specified criteria. A simple language is defined for specifying such filter predicates. The filter predicates can be loaded in the kernel and interpreted, or they can be compiled together with the kernel. A received packet is matched with all local filters' predicates in the order of filter priority until it is either accepted by some filters or is rejected. An accepted packet is delivered to the user process while a rejected packet is dropped from the network.

2.4 Novel IPC Semantics

Recent work [12, 11, 72] is beginning to look at process communication activities as the sharing of information among processes by use of message sending. If interpreted as *shared memory* in the sense of multiprocessor hardware, such global sharing in a multicomputer using messages can be expensive due to the communication cost imposed on each process. Such costs arise from the *shared-memory* requirements of reliable communication, consistent sequencing, and efficient updating. They grow linearly with the number of processors involved [12] and worse than linearly if network failures are considered. Cheriton's ideas regarding problem-oriented shared memory are similar to our intended and actual uses of topologies [71] in that we contend that costs should be reduced by use of application-dependent forms of consistency and reliability. Thus, some of the generic characteristics identified by Cheriton are trivially and efficiently implemented as computational messages in topologies—e.g., detection of stale data on use, sufficient accuracy of stale data, optional data, discardable updates, and democratic memory.

In contrast to our notion of application-dependent sharing, Linda [11] supports a programming model based on a globally shared associative memory mechanism called *tuple space* that consists of data elements called *tuples.* Processes communicate by adding (writing), reading, and retrieving tuples from tuple space. However, it offers only one type of semantics for read or write—all shared information is strictly consistent and fully reliable. Application-dependent forms of sharing cannot be done in Linda. For instance, shared variables in Linda are distributed across the processors using a hashing function, and the user cannot define the distribution of the variables directly. Also, Linda's implementation has a high demand for multicasting on the system, and this may cause substantial communication overhead in many systems where the multicast operation is not efficiently supported.

2.5 Programming Environments and Program Visualization

Our programming system for implementing topologies called PRISM consists of a number of tools including a topology compiler, a loader, and a graphical display system integrated through a database. Using a data-manipulation language, topologies can be controlled and manipulated. Some current work related to PRISM concerns systems which provide an integrated environment for display of data structures and/or code of parallel programs. Four examples are discussed here.

Poker. Poker [83] is a parallel programming system in which a parallel program is described as a graph defining the communication structure of the program. The user defines the communication graph by drawing its connections on a two-dimensional stylized lattice using an interactive display system. Graphical symbols depict processes and ports respectively. Ports are linked to sequential program text that is entered textually. Poker does not allow the user to customize displays. Poker's internal database is mainly used to store program segments and other information; it is not intended for tool integration. Hence, user-written mapping schemes are not easily incorporated into a Poker application.

In contrast to Poker, the PRISM system described in Chapter 7 of this thesis is not intended as a graphical programming system. Its displays are not hardwired; they are defined by the user and can be easily changed or manipulated. The database in PRISM stores interesting attributes of a parallel program and is used for manipulating the program. In addition, it provides a basis for uniformly integrating several programming tools.

GARDEN System. The GARDEN system [62] is an experimental system for large-scale graphical programming; it allows pictures to be described and *executed* using an object-oriented programming environment. A set of types defining the program's objects and their semantics are defined for the visual language. An evaluation function associated with each type defines its semantics. Then for each type, the display prototypes are created and the graphic editor is augmented to support these new types. Thus, in GARDEN, several visual languages can be easily prototyped to describe programs conveniently. The GARDEN system also has a package called GELO, which performs automatic layout of graphical objects.

PRISM differs from GARDEN in several ways. First, it offers no facilities for defining any visual languages, nor does it encourage programming using different visual languages for different applications. Instead it uses the same language to manipulate displays as well as the program structures. Second, PRISM differentiates manipulation of program components from manipulation of their displays while GARDEN considers graphical objects as the same as their corresponding program components. Third, GARDEN is not intended for integration of nongraphical tools whereas PRISM is intended to integrate several tools in addition to the visualization tool, such as program monitoring and resource allocation. Unlike GARDEN, the current implementation of PRISM does not have an automatic layout package similar to GELO. However, it could be implemented as a separate tool and integrated into PRISM.

PV Environment. Program Visualization environment [40, 26] developed at the Computer Corporation of America is an interesting environment that provides the user with representations of the program structure and monitoring information, and allows animation of program execution. A program's code and data structure are displayed both graphically and textually, and dynamic changes in data structures or in the control sequence of program execution can be displayed graphically. Users can define their own graphical pictures, which are linked to code and data, permitting the display of different levels of abstraction.

As with the PV environment, PRISM also allows user-defined graphical icons, which may represent different levels of abstractions of the program. However, PRISM is a parallel-programming environment while the PV environment is not, and PRISM does not display code.

Incense. Incense [54], as part of Xerox's MESA environment, provides graphical displays of data structures defined in the strongly typed Mesa language and also depicts the execution sequence graphically. To display a variable, a collection of graphic procedures called the *Artist* is called with parameters that specify the variable's location on the screen. Artist retrieves data from run-time tables or calls other artists associated with each component of the display.

PRISM is similar to Incense in that its display concerns data structures rather than code and in that it is capable of monitoring the execution of the topologies. In both systems, users can define their own displays and can use multiple displays for a single data structure. Incense automatically updates all displays of the same data structure if they are currently displayed, while in PRISM a user-written data-manipulation program must update the displays.

2.6 Conclusions

As with distributed objects, topologies address issues of parallel programming However, the concept of topologies focuses on establishing distributed objects' IPC structures. Remote execution on more than one processor has been addressed by Troupes as well as by topologies. Program visualization is also an area addressed by systems such as PV and also by topologies. However, fundamental differences in architecture (i.e., structured multicomputers versus unstructured or simple ethernet-based multicomputers) have implications on the design of topologies that make this work different from others. For instance, it would be inappropriate to have a switchboard (as in Demos or Charlotte) to establish connections on a machine like a hypercube. In the next chapter, the testbed for implementing topologies is described to establish the experimental environment together with the sample applications with which they are evaluated.

CHAPTER III

Decomposition Methods: Global Data and Operations

This chapter summarizes several well-known decomposition methods for parallelizing serial programs and indicates the unavoidable global data and operations resulting from such decompositions. It also discusses the importance and the need for a facility for programming global data and operations efficiently in order to obtain good performance.

3.1 Domain Decomposition and Global Data

This is one of the most commonly used methods for programming structured multicomputers such as hypercubes. Domains are typically defined as data structures that model physical systems [58, 25, 36, 27], such as a matrix representing a finite element mesh [10], and they are decomposed and mapped onto the parallel machine so that replicated code operates in parallel on different partitions of the structure [34, 35] (also termed SPMD—single program, multiple data). The need for global data arises when a domain's structure is not easily decomposed to match the underlying multicomputer. However, even when the application can be decomposed to match the target hardware perfectly, global data may exist (i) as data shared between processes iterating over adjoining partitions in PDE solution methods [36, 35], (ii) as aggregations of distributed local data and the subsequent broadcast of the aggregates, as in the computation of global objective functions [23, 72] and as in the computation and broadcast of a global error norm from partial local norms when performing convergence testing in the solution part of FEM applications [10], (iii) as global minima or maxima in other *Compute-Aggregate-Broadcast* algorithms [32], or (iv) as cumulative output from partial local outputs, such as alternate scanlines generated by replicated tasks and combined to a complete image in computer graphics applications [24, 60].

3.2 Domain Decomposition and Global Operations

Some domains are constructed during program execution, such as the search tree constructed during the execution of a parallel branch-and-bound algorithm [72] on a hypercube machine. This dynamic tree must be mapped and remapped to the application's processes performing the search, thereby balancing workloads among processes [51, 23]. Remapping involves the dynamic selection of a process able to offload work by a process seeking additional load. This entails the sending of a request seeking work to loaded processes, which may accept or reject it based on local, application-dependent information. Work may be migrated as data describing it [72] or as a process able to perform it [88]. In either case, the selection and migration of work constitute examples of global operations implemented as multiple distributed actions that cooperatively implement such worksharing. Additional global operations in parallel programs generated by domain decomposition concern synchronization between processes that are working independently on data of different *age*. Some parallel algorithms require strict lock-step synchronization of neighboring [6] or all [43, 42] nodes (i.e., all data must be of the same *age*), as with barrier synchronization [32, 84]. Parallel algorithms developed using relaxation techniques can tolerate the existence of distributed values of differing ages [35] but still require occasional synchronization. The distributed actions implementing global synchronization will use state information about application processes based on which synchronization is performed, and for process control, they will interact with the multiple-node operating system kernels.

3.3 Divide and Conquer and the Task/Queue Model

The task/queue model [17] is a generalization of the notion of work sharing. Here each process defines computational tasks dynamically, typically using divide and conquer techniques [35], and offers them for execution to other processes by task insertion into single or multiple, global, or regional [30] queues (this paradigm has also been called *generate and solve* [23, 32]). Sample tasks are subtrees of the search tree in branch and bound algorithms [51], sublists of a list to be sorted [21], *futures* in Butterfly Scheme, or *tasks* in the Butterfly's Uniform System [17]. Such global or regional queues constitute abstractions implementing global operations. Their implementations should be distributed ones. (See Section 5.4.2 for a design of a distributed global pool.) Dynamic task generation also occurs with the use of *fork* and *join* instructions in automatic methods for program parallelization, such as the *DoLoop* parallelization methods and constructs used by Kuck [41, 32] and at IBM [84]. Here a single thread of execution spawns (forks) multiple threads, which may later synchronize (or join) at barriers embedded in the code. Spawned threads must be submitted for execution by entry in a global queue, which again (like barriers) constitutes an abstraction used for global operations.

3.4 Functional Decomposition and Pipelining

Parallelism generated in this fashion does not pose new problems for the programming of multicomputers. However, the nonhomogeneous parallelism generated in this fashion leads to complications with the previous decomposition methods' uses. Additional instances of global operations may arise in the form of multiple, global task queues [30], and nonuniform communication structures may need to be used, such as multicasting instead of broadcasting [16]. An interesting class of applications typically decomposed functionally are robot control programs [57, 73, 69]. Such applications may require that global deadlines are guaranteed during execution [52]. Global deadline scheduling can be implemented efficiently using distributed objects, which may be explored in future research. In addition, robot control programs may require communication constructs in addition to those provided by the node operating system [69, 73].

3.5 Multiple Phases

Most realistic, large parallel applications consist of multiple phases each containing substantial parallelism. For example, the first phase of the FEM application described in Section 4.2 and [4, 10] concerns the parallel generation of distributed data: its second phase concerns the solution of a system of equations based on that data. Global operations are required due to the fact that the second phase cannot start before the first phase has completed. In other applications global data may occur due to data redistribution between phases, which often constitutes a limiting factor in the performance of a parallel program [78].

3.6 Global Data and Operations in Operating Systems

Many global computations in parallel applications on structured multicomputers implement functionalities typically provided by the operating systems of shared-memory multiprocessors, such as I/O, exception handling, multicast communications, etc. [33, 45]. For example, the output of a multicomputer application may be performed by a distributed object that implements a spanning tree of processes, where each tree node explicitly collects its neighbors' outputs, possibly performing some post-processing, and then forwards the resulting concatenated output data, etc. Other communication structures may be designed to carry certain control messages, such as instructions to processes to resume or terminate execution.

In general, any large parallel program will exhibit many such distributed abstractions among its independently executable processes, where the importance of some of those abstractions is shown by their explicit support in multiprocessor operating systems. For example, an abstraction expressing the *dependencies* [33] between a program's multiple processes in the StarOS operating system for the Cm* multiprocessor allowed certain processes to terminate or control others, as with parent and child processes in Unix. Similarly, a *bailout* abstraction allowed one process within a parallel application to handle the exceptional conditions of the application's remaining processes (i.e., to *bail out* other processes [33]).

In contrast to the OS support for multiprocessors described in the previous paragraph, the abstractions implementing OS functions in multicomputers must be *programmable* by application programmers to attain suitable performance. For example, a specific parallel application may require partially ordered output data, which may not be provided by the *standard* output utility. Instead the standard utility may be optimized to produce output at the fastest rate possible using a distributed object with nodes that simply concatenate and forward the outputs from neighboring nodes within a spanning tree. Similarly, for specific applications, it may be desirable to associate some output post-processing the output abstraction's concatenation actions, such as filtering, correlation of output data, etc. For example, one implementation of a computer graphics application for the hypercube designed by our research group required pixel data output by one process to be correlated with data on the same pixels known to other processes.

Programmability will also be required for other inherently global operating system facilities, such as program input, program monitoring [55, 82], and debugging, support for process scheduling and/or migration [70, 72], and the implementation of process groups [13, 45]. Monitoring is of particular interest because the obvious approach of sending all monitored data collected at a certain node to a central monitor is infeasible [82]. Monitoring is best performed by distributing both the collection and analysis functions associated with a particular monitoring query across the multicomputer [55, 56]. Thus, a distributed monitoring object consists of cooperating nodes that filter and partially analyze collected data and then forward the reduced amounts of data to other nodes, thereby reducing total communication as well as parallelizing the potentially time-consuming analysis actions. Similarly, the global operations desired for debugging a parallel program may be implemented using a distributed object with nodes that are able to affect their local application processes' execution states.

Higher-level operating system facilities currently being developed for multicomputers also exhibit global data or operations. For example, the names of objects or procedures being invoked are global in RPC and in distributed object implementations [8, 2, 19, 77]. Similarly, tuple space is global in Linda [11], which causes substantial overheads for Linda's hypercube implementation [78].

3.7 Conclusions

Several commonly used decomposition methods for parallelizing serial programs exhibit global data and operations. The most significant point of this chapter is that regardless of the decomposition method used, global data and operations will arise, either between different computation phases or as a part of the parallel computation, and that without having a facility to program them efficiently, performance may be drastically affected.

CHAPTER IV

Experimental Testbed and Sample Parallel Applications

This chapter describes the testbed for implementing topologies and the sample parallel applications used to evaluate topologies and identifies the global data and operations in these applications.

4.1 Testbed Environment

4.1.1 Hardware

The underlying nonshared-memory machine which serves as a testbed for implementing topologies and sample applications is a 32 node Intel iPSC/1 hypercube. The node processors are connected as a hypercube interconnection, as shown in Figure 4.1. In a hypercube interconnection an n dimensional cube has 2^n total of processors, with each processor having n directly connected neighboring processors. The average distance between two arbitrary node processors is $n/2$ and the maximum distance is n. A hypercube is a versatile structure onto which trees, rings, meshes, or other regular structures can be mapped easily [64, 25].

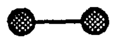
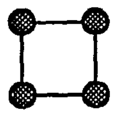
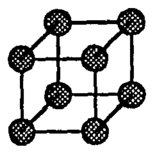
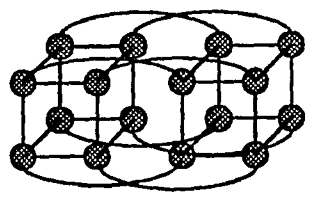
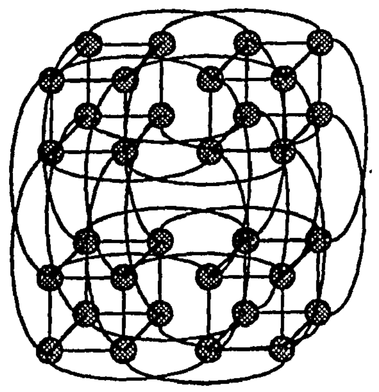
Cube Dimension	Number of Nodes	Interconnections
0	1	
1	2	
2	4	
3	8	
4	16	
5	32	

Figure 4.1: The Hypercube Interconnection

Each node consists of a 12 MHz Intel iAPX 286 processor, 512 kilobytes of memory, an iAPX 287 floating point co-processor, and 8 ethernet communications channels built from local area network (LAN) controllers (each controller is the iAPX 82586 chip), of which 6 channels are in use. Five channels are connected to the neighboring processors to form a hypercube, and the last channel is a global channel which links all the processors to the host, also called the *cube manager.* The cube manager is an Intel System 310/AP running the Xenix 3.4 operating system on an iAPX 286/10 processor. A framebuffer attached to the cube manager provides 640 by 480 pixels display and a total of 1024 colors with 256 colors displayable at a time. Figure 4.2 shows the hardware testbed. Note that a global ethernet connects all the node processors and the cube manager. The cube manager is also connected through the departmental ethernet to a number of Sun workstations. A Sun 3/50 is used as a remote host to the hypercube in place of the Intel System 310/AP for the PRISM environment for programming topologies.

The internal architecture of a node processor is shown in Figure 4.3. The memory is organized as a dual-port memory with two independent buses—one connected to the processor and the other to the communication controllers. Communication between each controller and the processor is provided by data structures called system control blocks in the memory. The hardware controllers understand the system control blocks and are capable of retrieving the list of messages and buffers linked to each of the system control blocks. A message to be sent is linked

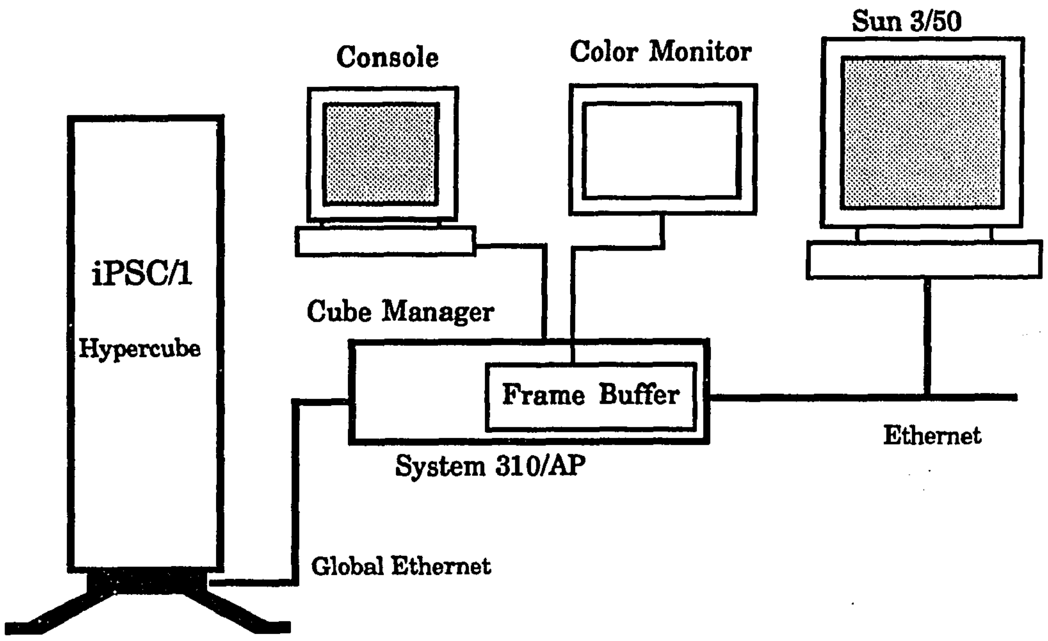


Figure 4.2: Testbed—Cube Manager and the Hypercube

to the system control block through a command block and a buffer header. A set of buffers for loading incoming messages is linked through buffer headers [31].

4.1.2 Software

Software provided by Intel for the iPSC/1 testbed consists of a node operating system called *nex*, which supports multi-tasking; a dynamic loader for loading processes; and a logger for debugging user applications. The interprocess communication primitives provided by nex can send or receive messages of sizes up to 16 Kbytes using send, recv, sendw or recvw calls. A *send* call initiates the sending of a message, and a *sendw* call causes the calling process to be blocked until the message transmission is done. A *recv* call initiates the receipt of a message but does not block the calling process, whereas a *recvw* call does. Each process on the

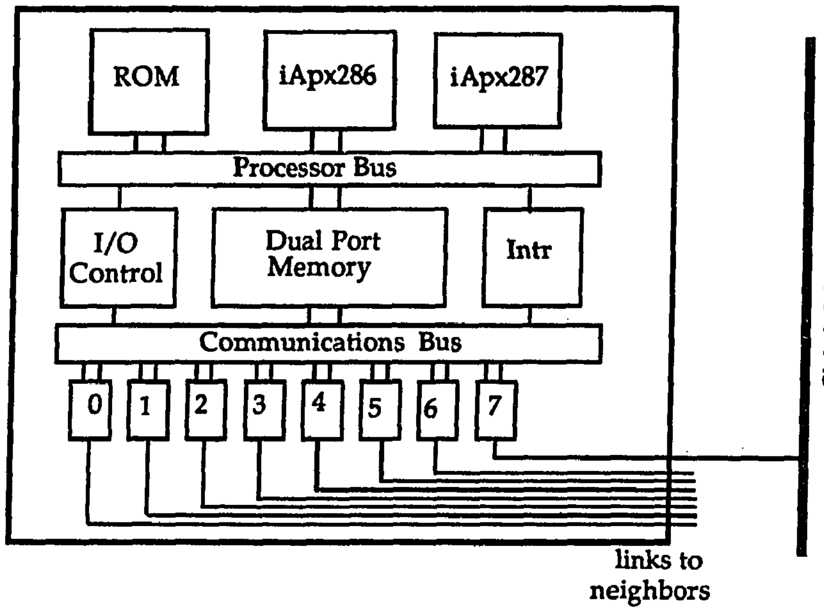


Figure 4.3: Node Architecture

hypercube is uniquely addressed by the node processor number on which it resides and by the process identifier, which is unique on each node.

4.2 A Finite Element Modeling Program

The first application used in this thesis is a finite element modeling (FEM) program [10] for metal-forming problems. It is a parallel version of a code called ALPID. This application typifies large-scale, scientific applications in that a large domain represented by some data structure is decomposed to generate parallelism. Briefly, the FEM program computes an approximate solution to the behavior of *nonlinear materials* [10] by repetitively performing a computation consisting of two phases. In the first phase, a matrix [K] is generated that contains the coefficients of a system of linear equations [K][u] = [F]. This generation is based on an initial guess for $[u]_0$ or on information from the second phase ($[u]_i$) and on information about the elements of the finite element mesh, such as elemental stiffness matrices, etc. In the second phase, the system of linear equations with the given coefficients is solved with an iterative solution method [4].

The two-phased computation continues until the solution vectors $[u]_i$ and $[u]_{i-1}$ are sufficiently close to each other. For problems of reasonable size, total execution time is distributed roughly equally across both phases. As problem size grows, the solution phase starts to dominate. Thus, subsequently, we concentrate on the solution phase of the computation.

A parallel program for the solution step of the FEM application is generated by partitioning the matrix [K] and the vectors [u] and [F] and replicating the solution code, followed by mapping partitions of the data and replicated solution code (*solver* processes) onto different nodes of the hypercube. To guarantee that only nearest-neighbor communications are required between solver processes when computing matrix/vector products, the matrix [K] is partitioned and mapped using a one-dimensional strip-mapping method [64]. This method partitions [K] into a set of rows containing the coefficients of a number of consecutive finite element nodes of the mesh; the total number of partitions is equal to the total number of processors. Nearest-neighbor communications are obtained by mapping consecutive partitions onto a linear chain of processors, where each processor is one hop away from its neighbors. The elements of the vectors [u] and [F] are mapped onto processors

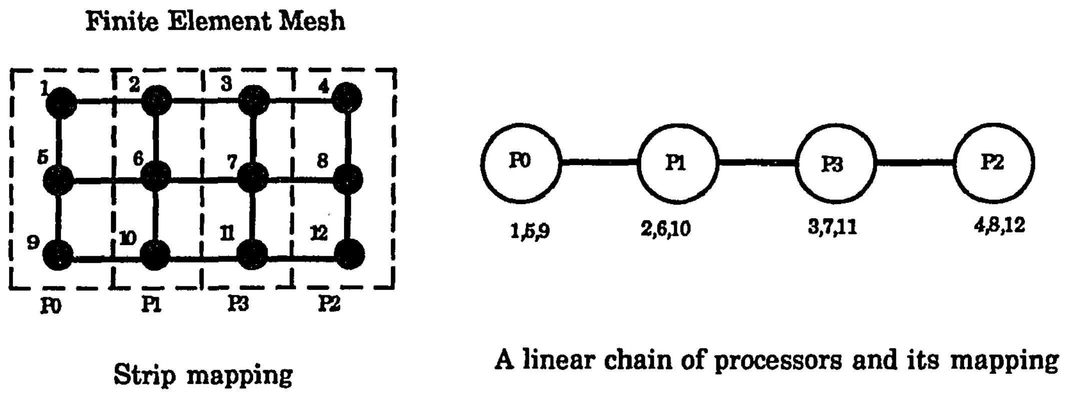


Figure 4.4: Strip Mapping and the Linear Chain of Processors

in accordance with the mapping of the coefficient matrix [K]. The resulting linear chain of processors and the finite element nodes mapped to these processors are shown in Figure 4.4. The strip mapping will also perfectly balance processor loads under certain conditions [64].

The actual method for solution of [K][u] = [F] is iterative, using the *restructured* conjugate gradient solution method [4]. In this solution method, one of two required global operations performed during each iteration has been removed. The pseudo-code of each solver appears below. Operations requiring nonlocal communication are labeled with '*' for nearest neighbor communications and with [G] for global communications/operations. Nonlocal communications performed once during initialization are labeled with 'I':

```
(I)Receive and distribute mapping information
(I)Receive [K] matrix
(I)Receive [F] vector
```

```
Initialize
    direction vector [p]
    residual vector [r]
    solution vector [u]

repeat
    (*)perform matrix-vector product [m] = [K][p]
    (G)perform inner-products [p].[m], [m].[m] to compute global scalars
    (G)distributed global scalars to all processors
    compute global residual norm from global scalars
    update vectors [p], [r], [u]
until
    global residual norm < tolerance
```

In each iteration the sparse matrix-vector product of the [K] matrix and the direction vector [p] are formed[1]. This requires communications with neighboring processors only, which is indicated by '*' in the program code. Next, two inner products involving the vector [m] are formed. Since [m] and [p] are distributed, these inner product computations are global computations—global sums (first G). In fact, for a problem of matrix size 25x25 on a 32-node Intel iPSC hypercube, speedup was improved by 13 percent when changing the basic CG method, in which two global sums were performed, to a *restructured* CG method, in which the global sum to update [p] was avoided by accumulating the two inner products in one step (as indicated by the first G in the program code above).

The computation of global scalars from the inner products and the computation of the residual norm as well as the updates of vectors [p], [r], and [u] do not require

[1]The direction vectors constitute a sequence of vectors from which the sequence of approximate solution vectors are constructed, starting from the initial guess.

any nonlocal communication. The criteria for terminating the iterative process is a tolerance that must be achieved by the global residual norm.

It is apparent that the efficient computation of the global sum is critical to the performance of this parallel application. However, additional global data and operations exist in this application; they stem from the cooperation between the two phases since the results of Phase 2 must be fed into the next execution of Phase 1 and since Phase 2 cannot start before Phase 1 has completed. Furthermore, the first phase of the application must compute a global minimum from local, partial minima in a fashion similar to the computation of the global residual norm.

In the next chapter we formulate a topology for performing the global sum computation in the FEM application.

4.3 A Traveling Salesperson Program

The second application is a branch-and-bound algorithm for solving a traveling salesperson problem [72, 51] (TSP). This application performs optimization in a dynamically constructed search space [51], which is typical of many nonnumerical computations. The algorithm is the LMSK branch-and-bound algorithm [50] which heuristically constructs and traverses a search tree. It starts out at the root with the entire graph representing the initial problem. Two independent subproblems are generated on each expansion of a node by selecting an edge and excluding it in one subproblem; while in the other subproblem the selected edge is the only edge included between the corresponding two nodes. Each of these subproblems is expanded further until a tour value is found. This value is used to prune some branches of the search tree [72].

The application's parallel implementation exhibits a three-level process tree. At the root of the tree is a single process that coordinates the actions of multiple subprocesses. Each subprocess independently computes a solution for a different subproblem. The subproblem is solved either by the subprocess itself, resulting in a flat two-level tree, or the subprocess employs additional *searcher* processes (up to four) that each solve a well-defined part of the subproblem. All computations at the leaf level of the tree (the *searcher* processes) proceed by expansion of nodes of a dynamically constructed, distributed search tree. This search tree consists of multiple subtrees, where each subtree root is represented by a matrix that describes a particular subproblem. This is shown in Figure 4.5 in which a three-level process structure is shown. The problem to be solved is indicated by a graph associated with the coordinator process, and subproblems defined by elimination of edges are indicated next to the subprocesses and searcher processes, along with the search trees constructed when searching along specific edges (the edges being searched are bold-faced in the problem graph.)

The pseudo-code describing the coordinator, subprocesses, and the searcher processes appears below.

```
Coordinator:
    generate and distribute one subproblem for each subprocess
```

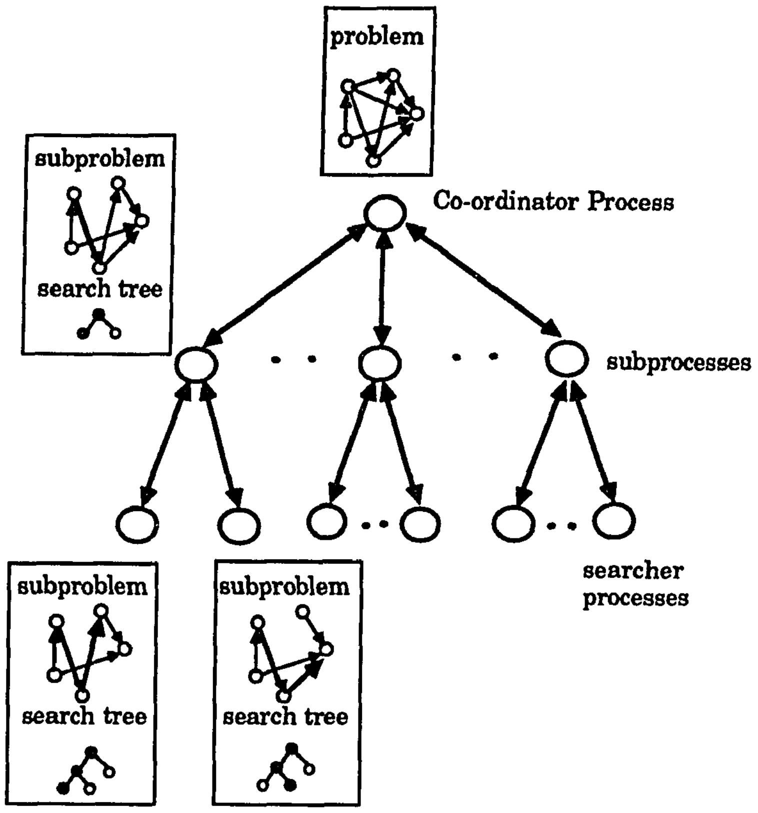


Figure 4.5: Three-level Process Structure for Solving the TSP

```
    repeat
        if a better tour value is received
            send this value to all subprocesses
        if a subprocess is idle
            ask a busy subprocess to send work to idle subprocess
    until all subprocesses are idle
```

Subprocess:

```
    receive work from coordinator
    do forever
        repeat
            while there are idle searchers and unexpanded subproblems
                send a subproblem to the idle searcher
            if a better tour value is received
                update the low tour value
            if a request to share work is received
                send a subproblem to idle subprocess
            if expanded subproblems are received from searchers
                add to the list of subproblems
        until all searchers are idle and there are no more nodes to expand
        send idle message to coordinator
        wait for work
    end
```

Searcher Processes:

```
    do forever
        receive work from a subprocess
        expand the nodes into a subtree of a given size
        send the subtree to subprocess
    end
```

A *Global Shared Memory* in TSP. TSP is interesting because it exhibits several unavoidable global data and operations, the first of which resembles global shared memory [12]. Namely, whenever the solution of a subproblem results in the detection of a tour, the *value* or length of this tour must be communicated to all other processes expanding the search tree. By comparing the current values of their own, incomplete tours to the value of a found tour, the search tree can sometimes be pruned, thereby reducing the amount of *useless* work being performed. Thus conceptually, all processes expanding nodes should share the value of the variable *best_tour* associated with their subproblems. However, the *shared memory* semantics of *best_tour* do not require that the global consistency of *best_tour* is maintained at all times.

Work Sharing in TSP. A significant source of complexity in the implementation of the parallel TSP program is the sharing of work among subprocesses. Specifically, each searcher must acquire a new subproblem whenever it completes the solution of (finds a tour) or aborts (prunes) its current subproblem. This entails finding a *good* subproblem currently known to a different searcher and then interacting with that searcher to share some of the subproblem. The direct implementation of the above TSP program demonstrates that even with dynamic work sharing, the parallel TSP implementation may spend a significant amount of its total execution time idling waiting for work. Therefore, implementing a global pool which allows the rapid sharing of work can lead to substantial performance improvements for the following reasons:

- A global pool as a repository for all work to be shared may be manipulated to remove units of work that are less useful than others, and it may order work units by their potential usefulness. Then, the *scheduling* of work now part of the user's application, is performed by a part of the pool's abstraction.

- If it can be assured that the pool always contains at least one unit of work while there is still work to do, the total waiting time of subprocesses for requested work is reduced.
- Larger problems may become solvable due to improved balancing of memory usage per processor (i.e., processors with problems that are too large could divest themselves of work).

The global pool can be implemented as a distributed pool object connecting the processes which need to access the pool. The detailed design of such a pool is described in the next chapter.

4.4 Conclusions

Summarizing the discussion in this chapter, it is apparent that multicomputers will not become more easily programmable until a wide variety of global data and operations programmed as distributed objects can be implemented conveniently and efficiently. The topology construct described in the next chapter can be used to program such global data and operations by designing multiple, arbitrary communication graphs that link the independently executable processes of a parallel application program and by defining operations performed in parallel at the nodes of such graphs.

CHAPTER V

Topologies

Topologies, briefly introduced in Chapter 1, are defined in detail in this chapter. Designs of topologies for implementation of global data and operations in the two sample parallel applications of Chapter 4 are also described, thereby demonstrating how such distributed objects can be used in parallel applications.

5.1 Topologies—Basic Definition

Operationally, a topology defines a communication protocol among a number of identified, communicating parties. Figure 5.1 depicts the basic components of a topology associated with an application program. These are (i) the communicating processes' identifications (the object users), (ii) their interfaces to the topology (object operations), and (iii) the topology's implementation (object representation), which consists of the logical communication links used for data transmission and the vertices containing data buffers, execution state, code, and intermediate results. The topology's code consists of (a) software for the efficient, reliable trans-

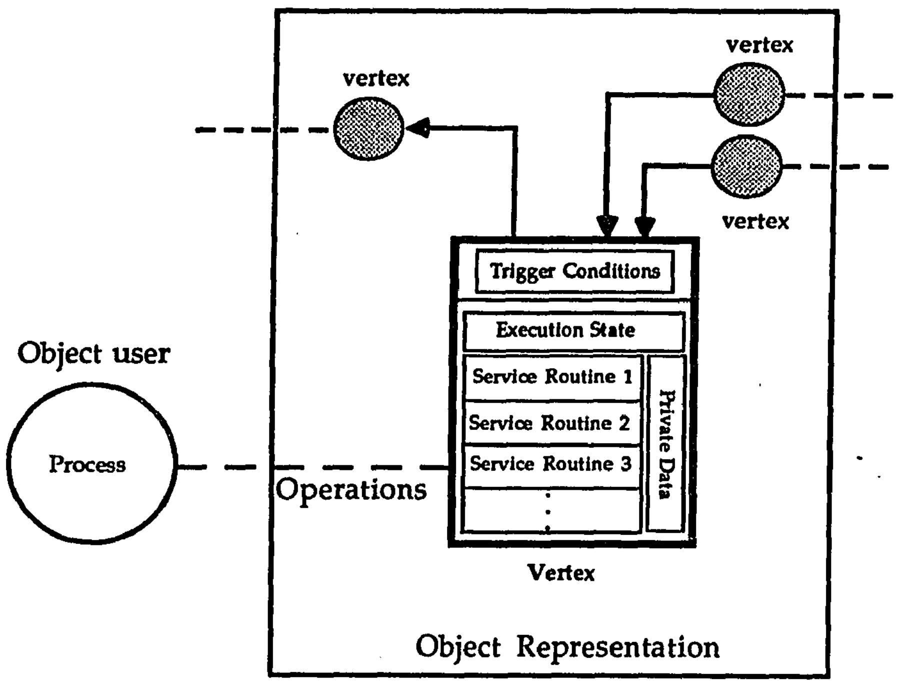


Figure 5.1: Topology—A Distributed Object

fer of data between two specific parties using some given means of data transfer (i.e., *link-level* software in the ISO model of data communication), (b) software that has knowledge of all connections from/to each vertex (roughly equivalent to *network-level* software in the ISO model) and also implements the topology's operations (called *service routines*).

Therefore, using the topology construct defined next, application programmers may define a distributed, abstract object consisting of the following components:

1. *Object Structure and Type Definition*—It defines the object as a topology with a certain *logical communication structure* and the topology's *service routines* implementing the object's operations and interaction with the struc-

ture's communication links. Service routines are executed when certain application-dependent or system-dependent *trigger conditions* (e.g., receipt of a data packet at the *link-level*) become true. It also defines execution state and intermediate data to be stored in the vertices of the communication graph.

2. *Object Mapping*—The object's logical communication structure and vertices (the object's distributed representation) are mapped to the underlying physical machines and communication channels of the multicomputer;

3. *Object Instantiation*—An object is instantiated statically (i.e., at compile time), and a unique name is associated with the instance.

4. *Object Binding*—Binding of an application to an object is done by identifying the parties (i.e., application processes) involved in communications with the topology (i.e., with the instance of the distributed object).

An object's structure is a directed graph, which may be described as a list of vertices connected with edges. (The grammatical and sample language construct descriptions below use boldfaced keywords; nonterminals are italicized; '::=' defines a production as in Backus-Naur form, whereas '=' is a terminal symbol; for convenience in notation, grammar operators like *sequence_of* are used. '. . .' indicates the omission of some details. A complete description of the grammar appears in Appendix B.)

```
type_declaration ::= Type_of_Topology TopologyType is {
   Structure is {
      Vertices = sequence_of (vertex_id);
      Connections = list_of (triple_of (vertex_id));
      }
   . . .
   . . .
   }

triple_of (vertex_id) ::=
   host_vertex_id : input_vertex_id : output_vertex_id

sequence_of(vertex_id) ::=
   smallest_vertex_id . . largest_vertex_id

list_of(elements ) ::=
   first_element; list_of(remaining_elements)
```

Type_of_topology is an identifier for a specific type of topology, as described by its set of triples. Thus, multiple instances may be created of each type of topology. *Vertex ids* are integers. *Host_vertex_id, input_vertex_list*, and *output_vertex_id* are defined below.

The sample type of distributed object below is a ring of 8 vertices, where integers are used as vertex id. In this ring, a vertex with id i is connected to another vertex with id (i + 1) modulus 7. (For brevity, vertex descriptions and edge descriptions are elided below.)

```
Ring TopologyType is {
   Structure is  {
      Vertices = 0..7;
      Connections =
         0 : 7 : 1;
```

```
        1 : 0 : 2;
        2 : 1 : 3;
        3 : 2 : 4;
        4 : 3 : 5;
        5 : 4 : 6;
        6 : 5 : 7;
        7 : 6 : 0;
        }
    . . .
    }
```

A topology's vertices (described by the *host_vertex_ids* above) contain the application-dependent components (object operations) of the topology's communication protocol—termed *services*. Each vertex contains information about its execution state, storage for intermediate data, and services represented as executable code segments.

An edge in a topology defines a uni-directional, logical communication path between two vertices. Edges provide a *link-level* communication protocol that guarantees the correct sequencing and reliable delivery of single, variable-size packets. Flow control for multiple packets and automatic buffering of multiple packets are not performed at the *link-level*; such actions may be application dependent and are therefore performed by services.

Object operations in a vertex must be associated with a specific *service routine*. However, in our implementation, a single service routine may be associated with multiple topologies, where each topology uses a different *service_type* for the routine:

service_routine_name **on** *node_id* **as** *service_type*

Thus, multiple topologies may share the same physical copy of a service routine's reentrant code.

Application-dependent *services* in a vertex are executed based on the truth values of *trigger conditions* stated with the vertices' specifications. For example, a two-vertex topology connecting two processes may filter messages such that only nonredundant values are seen by the receiving process. In this case, the application programmer might associate a service with the source vertex that performs value filtering by comparison of the current value being provided by the source with the previously transmitted value. The trigger condition for activation of the service in the sending vertex is *provision of input value by process bound to the source vertex*, and its trigger condition for producing an output (i.e., sending a value across the edge) is *new value*. Note that such filtering may be useful in robotics applications where a value produced by an iterative sending process is of interest to an iterative receiving process only when it differs from the value seen in the previous iteration [69, 73, 57].

More specifically, an edge and a vertex of a topology are described by the following abstract data type (nonterminals are italicized):

Vertex is
    host_vertex_id;
    type_of_topology;
    Map_id (defined below);
    bound_process_id;
    private_data_size;

stack_size;
services: list_of (*service_type*);
inputs:list_of(tuple_of(*input_edge*, *input_queue*)); *input_condition*;
outputs:list_of(tuple_of(*output_edge*, *output_queue*)); *output_ condition*;

Edges are
transmission_protocol_attributes;
state;

In these descriptions the *transmission_protocol_attributes* of the **edge** specify the attributes of the network-level protocol, such as number of retries, network-level flow control, routing, or whether the packets are to be acknowledged or not.

Regarding the **vertex**, the permanent execution state and intermediate data maintained in *private_data* can be arbitrarily complex[1], and it may be used for implementation of user-specified scheduling code and queueing structures that define arbitrary orders of input processing, service invocation, and output generation. Temporary execution state is maintained in the vertex's *stack*.

Note that trigger conditions are stated separately for input and output. *Input conditions* may be used to control the activation (scheduling) of services based on the availability of inputs. For instance, an input condition may state that a service is executed incrementally as each input arrives (e.g., when the service is a simple addition of incoming data values). Alternatively, a service may require that all inputs are present for it to execute. An input condition does not identify

[1]Since storage and buffer space in the 286-based hypercube are limited, the current implementation limits the number of queued input packets to one for each input link. Similarly, the *private_data* segment associated with each vertex must be small.

the services to be used for certain inputs. However, such identifications are found within headers of packets traversing the topology, and they are provided by the user programs accessing vertices (see the next section). Input conditions may also be used to implement different models of computation. For example, a topology implementing the data-flow model [20, 29] would contain conditions that cause services to execute when all inputs for a particular vertex have received new data. Other boolean conditions stating that a service should execute when 2 out of 3 inputs or any one input has new data may be formulated as well.

Output conditions control the scheduling of output generation; an output may be generated after each service execution or only after some delay required or desired by the application. Furthermore, the results of a service's execution may be sent to one, some, or all output links of a vertex.

Implicit in the descriptions above is the distinction of queueing and nonqueueing vertices. Specifically, when a service may be invoked incrementally because its order of execution viz. incoming messages is immaterial, then the vertex need not queue incoming messages. If a vertex must wait for a set of inputs for the input conditions to come true, then the input messages will be queued. For performance reasons both types of vertices are supported in our system (see Section 5.1.2).

5.1.1 Services (Object Operations)

The distributed object operations are performed by *services* which are threads of execution distributed across the processors. A service maintains a state, which is stored in the data part of a vertex, and a user-definable segment of code called a service-routine. It is activated by an arrival of a message or a set of messages. Once activated, it executes until the activation is completed. Only then can it be reactivated by the arrival of another message. The activation may terminate by generating one or more messages, or it may generate no message at all. These messages may initiate operations at other nodes or simply be queued at the user process bound to the service node. Services interact with that process through such messages in either one or both of the following two ways: (i) control of execution or (ii) transfer of data via (a) system buffers and (b) shared data areas.

For a given topology, a particular service may be activated by specifying a service id unique to the topology. When an object operation is executed concurrently and cooperatively by replicating the service routines on different processors, these routines are named by the same service id. For instance, a global sum performed as a tree structure may have partial summing service routines distributed on the processors at the tree's nodes; the *summing* services on the processors have the same ids. When the *sum* operation is invoked by a user process, the local *summing* service routine is executed. If such an operation is not completed or cannot be performed locally, remote service routines are executed by sending a topology message carrying the remote service id and the parameters required for remote execution.

5.1.2 Topologies—Extended Definition

The descriptions in the previous section are overly simplified because, in general, the services performed by a topology's vertices may range from simple, low-latency message switching to complex, application-dependent computations. Thus, our actual implementation offers alternative representations for the services of each vertex. This results in significant differences regarding a service's execution overhead, latency, and predictability in performance. The following alternative representations of services are provided:

- Small-grain computations may be performed by services implemented as procedures called at interrupt level within the operating system kernel following the execution of the *link-level* communication protocol. Such services may be invoked and executed with very low and predictable overhead and latency (see Section 6.3 for performance measurements). Their execution is atomic.

- Medium-grain computations may be performed by services represented as interruptable small tasks executing within the kernel's address space and scheduled in a round-robin fashion. Such representations may increase fairness and average throughput in the execution of multiple topologies and services within a single operating system kernel but may increase the latency of execution for specific services, as shown by our implementation of kernel tasks for character processing within the Unix System V kernel [39].

- Large-grain computations may be performed with higher latency and lower predictability by embedding services within a process that executes in user space. Such a process may be notified using an upcall-like mechanism [15] when the condition determining the service's execution becomes true, or it may service a queue of inputs associated with such a condition.

5.2 Topologies—Instantiation and Binding

A distributed object is instantiated separately from an application program so that it may be reused or changed dynamically. For its instantiation a topology's vertices and edges must be mapped to the nodes and physical communication links of the underlying computer ensemble. A mapping is defined separately from an object's structure by assigning vertices to physical nodes, where *Map_id* is a numeric identifier for the mapping:

Map_id **is map** list_of (tuple_of (*node-id:host_vertex_id*))

or

Map_id **is mapping_routine** { *routine_name()* }

Thus, whether they are user-defined or generated by automatic mapping algorithms [64], mappings may be varied independently of an object's structure, thereby resulting in nearest-neighbor communications, balancing loads, or optimizing other performance criteria.

Given a structure definition, the vertex mapping is either stored as data (as suggested by the first grammatical description above), or it is computed dynamically. A *stored mapping* is implemented by partitioning the structure definition and mapping information such that each processor maintains information only for those vertices that are mapped to it. This permits dynamic changes to a topology without having to broadcast those changes to all nodes. A *computed mapping* is implemented by services in a vertex that derive the physical nodes for the vertex's inputs and outputs. Sample computed mappings are dynamically spanning broadcast or multicast trees, rings, and other regular structures [29]. Computed mappings may be enhanced to implement dynamic routing algorithms [18].

Given a topology structure definition (a topology type) and a mapping, a global unique name is associated with a specific topology instance by combining the type with the mapping:

TM_id **Topology is** *type_of_topology* **and** *map_id*

Once named in this fashion, the topology is fully instantiated and may be loaded onto the nodes of the distributed architecture. Once loaded, it can be used by application programs. A topology may also be generated by a user-written topology generating routine. This routine generates the topology structure, vertex descriptions, and edge descriptions which are then down loaded into the node operating system kernel. Such a topology may be defined as:

TM_id **Topology is** { *routine_name* }

A distributed object cannot be used by an application program until a binding of its vertices with the application's processes has been established. Each vertex may be bound to zero or one process of the application program, and each process may be bound to multiple vertices of the same or of different topologies. Bindings are made dynamically and under program control by reference to a specific topology using the call:

thandle = **TopOpen** (*TM_id, host_vertex_id, init_tag*)

Here *host_vertex_id* identifies the name of the specific vertex of topology *TM_id* to which the calling process wishes to be bound (recall that multiple vertices of one topology may be mapped to the same node). *Init-tag* is an initial value for the *tag* to be used by this vertex for this process (tags are described in Section 5.3.1). *Thandle* is a system-provided object handle for the topology identified by *TM_id.*

The effect of a binding of a vertex to a process is the association of two predefined input and output links of the vertex with the process. Thus, the binding is bi-directional. The input and output conditions associated with those links are ignored when no process is bound to the vertex. A binding is explicitly broken using the call:

TopClose (thandle)

This allows processes to ignore communications they do not wish to handle, but it need not deactivate the vertex itself given that its input and output conditions are written to ignore the links to the specific process when necessary.

5.3 Programming with Distributed Objects

5.3.1 Using Topologies

Once defined, instantiated, and bound, a topology represents an instance of a distributed, abstract object that can perform certain user-defined operations. While each such topology should appear to the application as a distributed, abstract object [70, 2, 23] encapsulating some desired functionality, the operating system's interface to arbitrary topologies should be type independent. Toward this end, we provide calls resembling message operations with which an application process may access the object via a vertex to which it is *bound* and identify the operation (service) to be executed:

TopSend (*thandle, service_id, parameters, count, tag*)

The effect of TopSend is that the *parameters* are made available to the service routine identified by *service_id* associated with the bound vertex of topology with object handle *thandle.* The object handle is obtained from TopOpen call described in the previous section. *Parameters* is a pointer to a buffer into which the required parameters have been packed (in the order expected by the service routine), and *count* denotes the size of the buffer. *Tag* will be explained below. [2]

The service routine identified by *service id* is activated when its input condition becomes *true*, which may be immediately following the execution of a TopSend call. The call returns with the value *true* to the calling process either after the *parameters* have been queued as an input to the queued service or after the service routine of the nonqueued, incremental service has completed its execution. If the service is nonincremental or if the service's execution entails the generation of an output across one of the vertex's output edges, then the user may wish to ascertain that an output has been generated and transmitted successfully to a target vertex. In this case, the calling process may use the call:

TopSendW (*thandle, service_id, parameters, count, tag*)

This call blocks the calling process until an output has been generated by the vertex.

In both TopSend calls the *tag* argument is a user-provided value that is incremented with each call. *Tag* may be used to identify a particular set of input data. For instance, service routines can use the tags on their input data for the explicit matching/sequencing of their inputs, as required for systolic programs. Similarly, in a ring topology the tag can be used to distinguish messages from different nodes.

A process obtains the result of its operation invocation by performing the call:

TopRecv (*thandle, service_id, parameters, count, tag*)

[2] *Service_id*, *count*, and *tag* are all of type integer; *parameters* is of pointer type which may point to any other types. *thandle* is of type object_handle.

This call returns the result *parameters* of the service *service_id* if the value of *tag* specified by the caller matches the tags of the inputs based on which the outputs of the service were generated (thus, tag values are matched for outputs, as well). If such outputs are not currently available, then the call returns with an error status. Service routines may be programmed to maintain a short history of the tag values of their inputs and outputs so that they may return the most recent value of their input and output tags into *tag*. In addition, TopRecv has a wild-card value of *tag*. This results in the return of the service's most recent output.

While TopRecv is a nonblocking call, the following call will block until a result with a matching *tag* becomes available.

TopRecvW (*thandle, service_id, parameters, count, tag*)

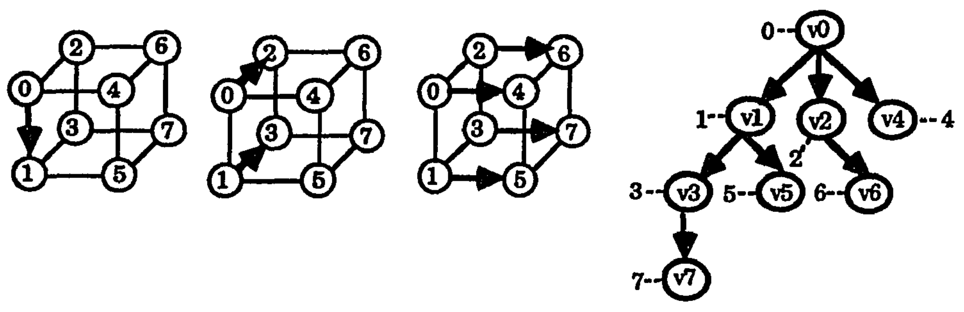
However, such a call will not block forever; it will time-out using a specification-time time-out value.

Note that the send/receive primitives shown here are not the only ways in which the computations performed by a topology's services may be associated with the computations of the bound application processes. Services may implement their own interfaces, including those that directly activate the execution of user processes (such as upcalls [15]) or directly access the processes' address spaces (see Section 5.4.2 for an example of such an interface).

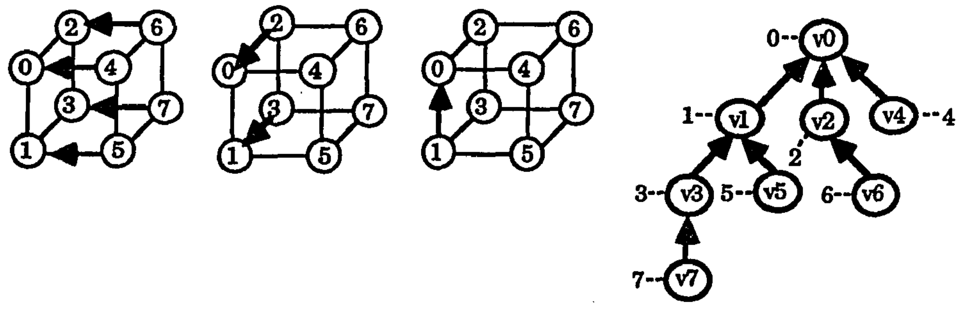
5.3.2 Writing Services

Service routines written for each topology may be classified into three kinds: initialization services, operation services, and termination services. Initialization service routines are executed at the time a topology is loaded. Such services are local and are replicated on all the processors on which the vertices reside. Thus, the topology initialization is fully distributed. Likewise, the termination service is a local operation and is executed at the time the topology is reset, in part to release the resources allocated for each vertex of the topology.

The operation services are the service routines implementing the distributed object's operations and are explicitly executed by requests from the user program. Since such service routines are event driven, their execution is activated by the arrival of a topology message requesting their services. Execution is terminated when the routine is completed. At termination the state of the service routine becomes inactive. Inactiveness is different from the blocked state of a process since reactivation transfers control to the start of the service routine whereas unblocking resumes control from the last state. However, the state of a service routine must be saved and restored explicitly if its execution is to proceed from the last activated state. Furthermore, the execution of a service routine of medium or high granularity may be preempted before the completion of the service and then resumed from the preempted state, but at the completion time of the service, control always returns to the start of the routine. This ensures that there will be one activation



Broadcast Tree



Inverse-broadcast Tree

Figure 5.2: Broadcast and Inverse-broadcast Trees

per topology message and that all the topology messages are treated uniformly.

5.4 Using Topologies to Program Global Data and Operations

5.4.1 The FEM Application

On a hypercube, a global sum is efficiently performed with a summing object having the structure of an inverse-broadcast tree since this structure minimizes the number of nonlocal communications required. The directed graphs of the broadcast and inverse-broadcast trees for a 3D-hypercube are described in Figure 5.2.

In Figure 5.2 V0-V7 are vertices of topologies mapped to the processors of a 3D-hypercube. The numbers next to the vertices are the names of the processors to which vertices are mapped. The root vertex is located at processor 0. Since a hamming code is used to number all processors on a hypercube, this mapping results in the placement of all connected vertices onto neighboring processors. The following topology types and mappings are defined to realize the broadcast tree and the inverse-broadcast tree shown in the figure.

```
BroadcastTreeType TopologyType is {
    Structure is {
      vertices = 0..7;
      connections =
        0::1,2,4;
        1:0:3,5;
        2:0:6;
        3:1:7;
        4:0:;
        5:1:;
        6:0:;
        7:3:;
    }
    Vertex is {
      VtxId: * ;
      Services: RESULTS_OP;
      Inputs: *: *+;
      Outputs: *: *+;
      }
    Edges are {
      (0,1) = ACK.FLOWCTL ;
      (0,2) = ACK.FLOWCTL ;
      (0,4) = ACK.FLOWCTL ;
      (1,3) = ACK.FLOWCTL ;
      (1,5) = ACK.FLOWCTL ;
      (2,6) = ACK.FLOWCTL ;
      (3,7) = ACK.FLOWCTL ;
    }
```

```
InvTreeType TopologyType is {
      Structure is {
         vertices = 0..7;
         connections =
            0:1,2,4:;
            1:3,5:0;
            2:6:0;
            3:7:1;
            4::0;
            5::1;
            6::2;
            7::3;
      }
      Vertex is {
         VtxId: * ;
         Services: GLOBALSUM;
         Inputs: *: *+;
         Outputs: *: *+;
         }
      Edges are {
         (1,0) = ACK.FLOWCTL ;
         (2,0) = ACK.FLOWCTL ;
         (4,0) = ACK.FLOWCTL ;
         (3,1) = ACK.FLOWCTL ;
         (5,1) = ACK.FLOWCTL ;
         (6,2) = ACK.FLOWCTL ;
         (7,3) = ACK.FLOWCTL ;
      }

M1 is map {(0:0),(1:1),(2:2),(3:3),(4:4),(5:5),(6:6),(7:7)}

Services are {
   DoSum()   on * as GLOBALSUM;
   No_op()   on * as RESULTS_OP;
   }
```

Given the type definitions and the map *M1*, unique names for both topologies are defined as:

Broadcast **Topology is** BroadcastTreeType **and** M1;

InvBroadcast **Topology** is InvTreeType **and** M1;

The service routines associated with the broadcast topology will not be described here since they are basically implementations of store and forward actions and are not highly illustrative of the distributed computations performed by the object. The computations in the *InvTree* are more interesting since this is where the global sum is performed:

- Each node in the GlobalSum topology determines, from topology structure and mapping information, a set of source nodes for partial sums.
- It then obtains a message from each source node and adds its contents to its local partial sum.
- After that addition and after all source messages have been received, it sends the partial sum to the next node up the tree.

Thus, the *DoSum* service routine sketched in C-like syntax below accumulates the value of incoming partial sums with identical tags in the static variable *partial_sum* (located in the *private_data* of the vertex and implemented as a global variable of type double precision floating point in this service routine). Addition is performed incrementally once for each input edge, including the input edge used by the process that is bound to this vertex. An output is generated to the single output edge as a result.

```
DoSum (input_buffer, count, tag)
char *input_buffer;
int count;
int tag;
```

```
  {
     (*partial_sum) = (*partial_sum) + (double) input_buffer;
     if(all inputs have been received)
        send_to_output (output_edge, partial_sum, count, tag);
        partial_sum = 0.0; /* Reset partial sum */
  }
```

Given the service routine indicated above, we can now sketch the code of the user programs bound to the root vertex and to the other vertices of the *InvTree* and *BroadcastTree* topologies. The process bound to the root vertex executes the program *Coordinate* that collects the global sum by receiving from the *InvTree*, adds its own partial sum, and then sends the result to the *BroadcastTree*, whereas the programs *Participate* at the other vertices send their own partial sums to the *InvTree* and receive the global sum from the *BroadcastTree*. *Myvid* is a variable containing the id of the vertex to which each of the processes will be bound, and *RESULTS_OP* and *GLOBALSUM* are (numeric) identifications of the specific service routines to be used for each TopSend or TopRecv.

```
coordinate() /* Root process */
{
     int tag, size;
     double partial_sum, global_sum;
     object_handle ibt, bt;
     boolean status;
        . . .
        ibt = TopOpen (InvTree, myvid, 0);
        bt = TopOpen (BroadcastTree, myvid, 0);
        status = TopRecv (ibt, GLOBALSUM, global_sum, size, tag);
        global_sum = partial_sum + global_sum;
        status = TopSend (bt, RESULTS_OP, global_sum, size, tag);
        TopClose (ibt);
        TopClose (bt);
    }

    participate () /* non-Root processes */
    {
      int tag, size;
      double partial_sum, global_sum;
      object_handle ibt, bt;
        . . .
        ibt = TopOpen (InvTree, myvid, 0);
        bt = TopOpen (BroadcastTree, myvid, 0);
        status = TopSend (ibt, GLOBALSUM, partial_sum, size, tag);
        status = TopRecv (bt, RESULTS_OP, global_sum, size, tag);
        TopClose (ibt);
        TopClose (bt);
    }
```

Note that these programs are written to use a topology exactly once after opening it. Thus, the program need not explicitly use or update tag values.

Global Minimum

Recall that the FEM application's K matrix generation phase also makes use of a global minimum operation. A topology for computation of such a value is simply constructed by using the topologies for the global sum and replacing the service routine *DoSum* with a routine that incrementally computes a local minimum from the values received as inputs and sends the resulting minimum to its outputs. The global minimum is then accumulated at the root node and broadcast to participating nodes. Since both the global sum and the global minimum topologies have the same structure and mapping, they are implemented with the same *InvTree* and

BroadcastTree topologies. Each phase simply uses a different set of service routines in *InvTree*, both of which are available in the topology.

5.4.2 The TSP Application

As described in Chapter 4, the TSP application exhibits two global data and operations—*global shared memory* and *work sharing*. Topologies to program these global data and operations are described below:

Global shared-memory Object

The following distributed object provides an implementation of *shared memory* by use of a ring structure connecting all subprocesses and searcher processes (see Figure 5.3).

The precise semantics of shared memory used in this object are that a *read* request (*Read_shared*) by a searcher returns to it the current value of *best_tour* stored at this node (in the vertex's *private_data*), whereas a *write* request (*Write_shared*) causes the propagation of a new value of *best_tour* around the ring. A new value for *best_tour* is accepted by service routines asynchronously of *Read* operations whenever the new value is less than (better than) the current value of *best_tour.* Propagation around the ring proceeds in one direction, which assures that only the best tours are actually fully propagated around the ring. However, no guarantees are made as to the consistency of the tour values stored and used at different ring

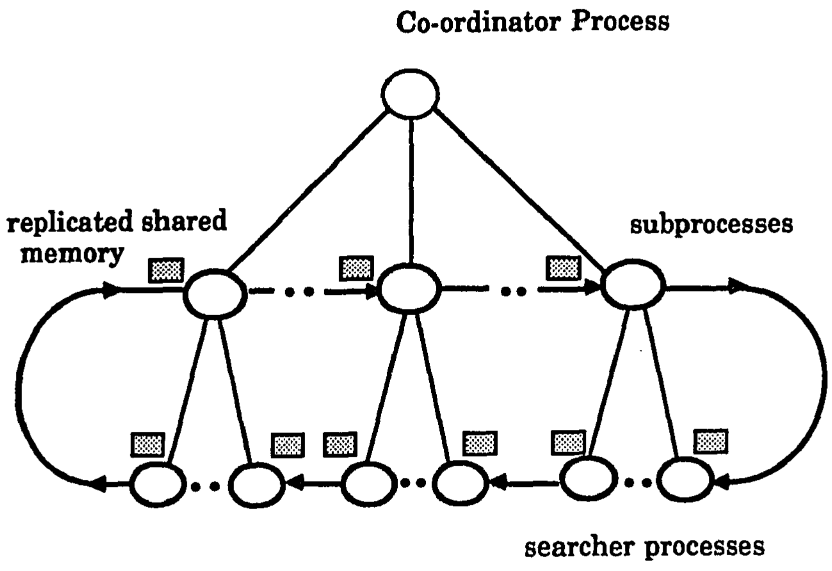


Figure 5.3: Shared-memory Ring Topology

nodes at any one time. Such guarantees are not required by this application, and their implementation and enforcement would be detrimental to the application's performance. It is for reasons such as these that we reject approaches like Linda [11] in which a single semantics of shared memory is supported.

The shared-memory object is used by the application program as defined by its interface routines:

Interface routines:

```
Setup_shared (initial_value, size)
{
  TopSend (thandle, screate, initial_value, size, tag)
}

Read_shared (best_tour, count)
```

```
{
  TopSend (thandle, sread, best_tour, count, tag)
  TopRecv (thandle, sread, best_tour, count, tag)
}

Write_shared (new_tour, count)
{
  TopSend (thandle, swrite, new_tour, count, tag)
}
```

The interface routine *Setup_shared* activates the service *screate* in order to initialize *best_tour*. *Read_shared* first communicates its own, current value of *best_tour* to the vertex then reads the vertex's *best_tour*.

The interface routines are implemented using the following service routines:

Service routines:

```
screate(initial_value)
{
  stored_best_tour = initial_value;
}

sread (best_tour, count)
{
  best_tour = stored_best_tour;
  return (best_tour);
}

swrite (new_tour, count)
{
  if (new_tour < stored_best_tour) {
    stored_best_tour = new_tour;
    send_to_output(link_to_next_vertex, new_tour,count,tag);
  }
  return;
}
```

Stored_best_tour is the location in the vertex's private data in which the current value of the best tour is stored. In this design, *Setup_shared* initializes the shared memory of a single vertex. It could be redesigned such that a single vertex may initialize the shared memory of all vertices in the topology. In addition, *Setup_shared, Read_shared* and *sread, swrite* could be redesigned to allow the vertex to directly update the value of the variable *best_tour* located in the bound process' address space. In that case, the call *Read_shared* would not be needed. It would be replaced by an efficient, noninterruptable (locking) access to the process' internal variable *best_tour.* Performance measurements and additional detail are reported in Section 6.4.

Work-sharing Object

The TSP's work-sharing object is a nearest-neighbor ring that implements a global pool of work-sharing requests. Requests are entered into the pool by processes that seek work and are removed by processes willing to share work. In addition, each vertex of the ring is connected to the single, coordinator process for purposes of termination detection, thus forming a pyramidal topology.

Requests are entered into the topology using its *Work_request* service. A request is entered at a vertex only if work is available, otherwise it is forwarded to the next vertex thereby avoiding unnecessary delays. Thus, a request message potentially travels along the entire ring. Work is assumed available if the searcher process attached to the vertex is *busy*; work is assumed unavailable if the searcher is *idle*.

Since each distributed component of the global pool associated with a particular vertex may become quite large, it is stored in the address space of the process attached to the vertex. Thus, the insertion of a request implies the interaction of the vertex with its attached process. In our implementation, this interaction is by direct access from the vertex to three variables in the process' address space: the variables *pfirst* and *plast* are pointers to the first and last elements of the pool, and *psize* is the current pool size. *Plast* and *psize* are updated by the vertex's service routine when entering a new request message at the tail of the pool. *Pfirst* and *psize* are updated by the attached process when removing a request for work from the head of the pool. *Psize* is updated atomically. The status of each searcher process is maintained in the variable *busy* in the process's address space and is inspected by the vertex's service routine and atomically updated by the searcher.

A searcher process attached to the pool inspects its local pool component prior to the time it chooses a new node of the search tree for expansion. At that time, all locally pending requests are removed, *good* subproblems to be shared are selected [72], and subproblem descriptions are generated [72, 23], which are then sent as direct messages to the processes seeking work. The requesting process and node are identified in each request message.

Note that a searcher process using the object may have received requests while expanding its last node. In addition, it may not have sufficient work to honor all requests for work stored locally. In these cases, the searcher flushes all pending requests by resending them on the object's structure (as a single message) on behalf of the original requestors. Such a re-sent request is marked *on_behalf* in the request message's status field, and if it is received by its requestor, it is re-sent around the ring. If a requestor receives a re-sent request, it concludes that no more work is available anywhere and terminates. The first vertex to discover this condition also notifies the coordinator process.

Finkel in a similar implementation [23] of work sharing for parallel TSP implementations has discussed various schemes for *fair* distributions of requests. Naively, each request may be entered at the first process that has available work. This implies that specific processes are likely to be overloaded with requests for work while others are not able to share any work at all. An alternative method of distribution is one that attempts to balance pool sizes by use of a tag value in each message. The tag value indicates the size of the pool of the last vertex visited. A request is entered into a pool only if that value is equal to or exceeds its own *psize*.

Service routines implementing the balanced pool of work requests appear below:

Service routine:

```
Work_request()
{ if (message is from local process) { /* from the bound process */
      on_behalf = 0
      send_to_output(link_to_next_vertex, request, request_size, tag)
  }
          /* check termination */
  for(i = 0; i < number of requests in the message; i++) {
      if (request is from local process){
        if(on_behalf == 0) {
           send_to_output(link_to_coordinator,stop_message,size,tag);
        } else
           on_behalf = 0;

   if (busy){ /* process has some work */
      if (psize <= tag) { /* insert the request if psize < tag */
         insert_the_request();
      } else {
         tag = psize;
         send_to_output(link_to_next_vertex, request, request_size,tag)
      }
   }
}
```

The interface of the work-sharing object presented to a searcher process is the following TopSend call:

TopSend (top_handle, WORK_REQUEST, buffer, buffer_size, 0)

Buffer contains the request consisting of the originating node's number, process id, and the initial value of *on_behalf*, which is zero.

There are several interesting attributes of the distributed, global pool. First, we maintain requests for work in the pool rather than descriptions of available work [23], due to the large size of work descriptions for problems of reasonable size (e.g., TSP problems with more than 25 cities). Second, we do not signal or interrupt a searcher process upon arrival of a request in order to minimize the disruption of the process' programming and control flow caused by its use of the global pool. Third, to be able to insert large numbers of requests or to insert larger items, the pool is stored within the address space of the participating processes. Fourth, the current implementation offers high performance of insert and remove operations because the discipline for insertion or removal is nondeterministic. Imposition of a deterministic discipline implies maintenance of some global ordering, thereby causing additional overhead even for the ring topology. Note that the use of a tree topology for the global pool would result in quite unacceptable overhead when attempting to maintain almost any global ordering. Last, pool membership is entirely dynamic since searcher processes can open and close their connections to individual vertices of the work-sharing object.

5.5 Conclusions

In this chapter, we discussed the basic concepts of the topology construct. The definition of a topology's structure, instantiation and binding, the programming of service routines, and a topology's interface to user programs were described. The topology construct provides a separately programmable communication structure, with computations performed by the service routines at the structure's nodes. Once processes are bound to a topology, they are linked with a distributed abstract object, which is programmed separately from the application and is capable of performing distributed computations.

Furthermore, this chapter demonstrated how the topology construct may be used to implement the global data and operations of the two sample applications in Chapter 4. Next, we show that the topology construct is realizable by implementing and evaluating the construct and the applications' global data and operations implemented with the construct on the Intel hypercube iPSC/1.

In place of topologies, we might have used two alternative means to implement the application-dependent global data and operations of the sample applications: (i) on the basis of simulated, globally shared memory, as in Linda [11] and (ii) using library packages or generic utilities constructed at the user level. We explicitly rejected (i) because it is apparent from the sample global data and operations described above that the desired, precise semantics of sharing (e.g., the consistency requirements of global data and control) are application dependent. Thus, mismatches are likely to occur between the single, offered semantics of simulated, shared memory and the required functionality of the global data and operations implemented with it. Our past experience on shared-memory multiprocessors [35, 69, 73] demonstrates that programmers concerned with performance will simply not use an offered mechanism when such mismatches occur.

It is also apparent that insufficient performance will result from library packages or generic utilities constructed at the user level, outside the existing operating system kernel. Such packages have been built for certain global operations (e.g., a user-level implementation of a global sum for use in numeric applications in the Intel iPSC's operating system), and they have been devised for the support of certain classes of applications, such as branch-and-bound algorithms [23].

CHAPTER VI

Implementation and Evaluation of Topologies

In this chapter the topology construct's implementation on the Intel hypercube is described and evaluated. Sample distributed objects are implemented in conjunction with the FEM and TSP parallel applications, and the performances of these applications with and without the use of these objects are compared.

6.1 Implementation of Topologies

The topology construct is implemented on a 32-node 286-based Intel IPSC hypercube. This implementation assumes the following:

- *Compatibility*—The topology addition to the iPSC kernel is performed such that any iPSC application may execute without change. Compatibility also requires that the performance of existing iPSC applications is not affected by the presence of topologies.

- *Performance*—In the best case, the use of a topology should improve the performance of an application program compared to its implementation of similar functionality using the existing iPSC communication constructs. In the worst case, the use of topologies should not degrade an application's performance.

- *Variability and Usability*—A topology's implementations should offer a variety of performance characteristics, such as different latencies and predictabilities of execution times. In addition, a constructed topology should be reusable, like any standard operating system utility.

The goals listed above are attained by an implementation of topologies within the existing iPSC 3.1 kernel. Compatibility is achieved by support of all standard iPSC kernel primitives in addition to the topology construct. High performance is achieved by representation of topologies' small grain services as short segments of code executed within the address space of the kernel at the interrupt level. This should also offer consistent, predictable performance since the execution of services is not subject to the scheduling of user processes containing them, which is particularly important when multiple user processes share a single node of the hypercube.

The variability of topologies is attained by providing alternative representations of services as schedulable, kernel-level tasks or as user processes (also see Section 5.3). Topologies are made reusable by providing system calls (TopClose, TopOpen) which user processes can attach to or detach from topologies.

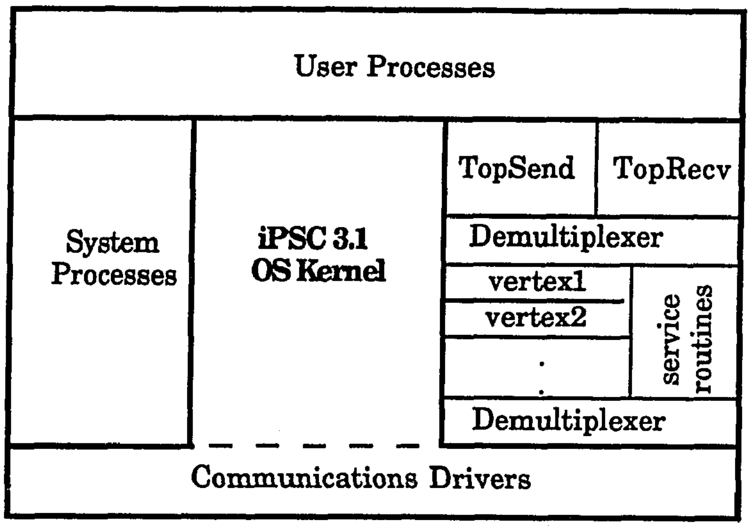


Figure 6.1: Extended Node OS Kernel

Figure 6.1 depicts the actual extensions of the iPSC 3.1 kernel:

- Additional header information is included with *topology packets.* A topology demultiplexer in the modified iPSC kernel matches topology packets with the topologies resident on the node schedules the execution of the topologies' service routines, and also schedules the topologies' outgoing packets for output across the node's physical output channels.

- The modified kernel contains the vertices and the structural information of each topology resident in the node as well as the mapping tables used by the demultiplexers. These tables map topology identifiers and service identifiers found in topology packets to the vertices and service routines of the topologies resident on the node.

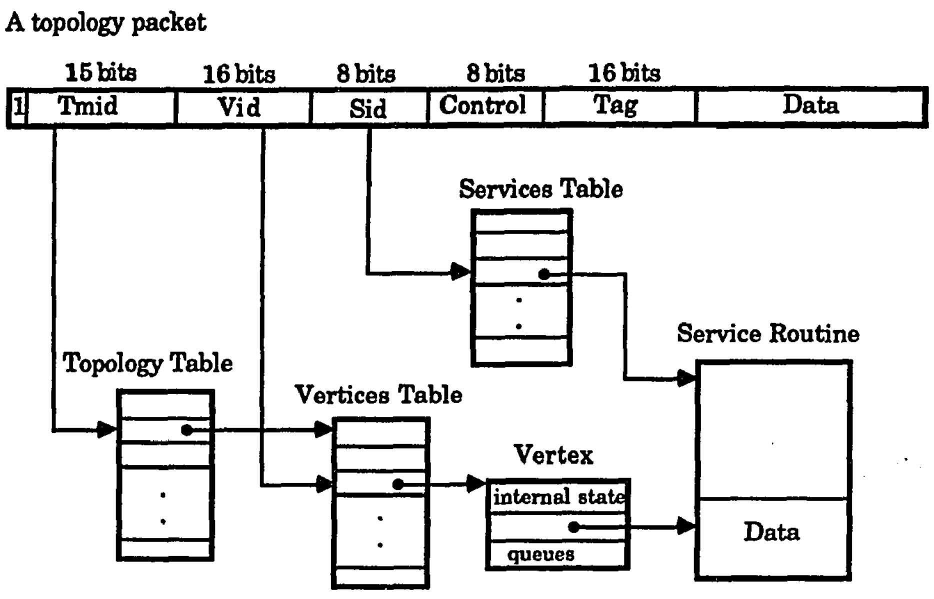


Figure 6.2: Demultiplexing a Topology Packet

- Additional system calls (TopSend, TopRecv, etc.) allow user processes to access and manipulate existing topologies, and to construct new topologies.

The headers of the iPSC 3.1 communication packets have been extended to include a topology identifier, an identifier of the destination vertex, and the identifier of the service being requested. In addition, each packet carries a tag value set by the sending vertex (also see Figure 6.2). These extensions do not increase minimum packet lengths because existing fields in the iPSC's communication packets are being reused. However, a one-bit field has been added to the header of each packet; it is used by the demultiplexer to distinguish topology packets from packets sent and received using the iPSC 3.1 communication primitives.

The demultiplexing of an incoming topology packet to a service of a specific topology is shown in Figure 6.2. Each packet is uniquely mapped to a single service of a single topology. This may cause two kinds of actions: (i) the execution of a target service in a target vertex when its input conditions are satisfied (e.g., all other inputs required for the service are present) or (ii) the queueing of the packet on the input queue of the target vertex. After execution of a service routine, the demultiplexer always evaluates the output conditions of the vertex. If such conditions are satisfied, output messages are queued on the appropriate physical output channels. Thus, the demultiplexer executes in three steps:

1. **Input**—Input conditions and queueing/nonqueueing options are checked.
2. **Services**—A service routine is executed to perform the operation requested in the topology packet.
3. **Output**—If output conditions are satisfied, the single or multiple result packets are forwarded to the single or multiple destinations.

Note that the tag field in the topology packet is not accessed by the topology demultiplexers; it is manipulated solely by services or by application code.

Topology packets are sent using the same low-level communication protocol as other iPSC communication packets. Therefore, future communication speed increases offered by Intel's new 386-based communication hardware may be realized for topology packets as well. The low-level protocol performs store-and-forward routing and flow control of multi-packet messages.

In these implementations the maximum number of input and output links per vertex is restricted to six because the amount of efficiently addressable (without manipulation of segment descriptors) memory space in the 286 kernel is limited to 64K bytes [1]. This restriction would not apply in the 386-based kernel. In addition, to minimize the overheads of buffer allocation at interrupt level, the sizes of the output packets produced by a vertex cannot exceed either 1 KBytes or the sum of the sizes of the vertex's input buffers used to produce the output.

6.2 Performance and Implementation of Topology System Calls

The semantics of the system calls TopSend, TopRecv, TopOpen, and TopClose have been defined in Sections 5.2 and 5.3. The straightforward implementation of these calls is explained below to provide insights regarding their performance.

Since topologies are not opened or closed frequently, neither the TopOpen nor the TopClose construct have been optimized for low latency. TopOpen simply performs a linear search for the named vertex on the list of vertices of the named topology. It binds the calling process to the vertex by storing the process' unique identifier (pid) in the data structure describing a vertex. It also allocates message

[1]The 32 bit length addresses of the 386-based hypercube will permit the efficient addressing of up to 4 Giga Bytes of memory space without manipulation of segment descriptors. In addition, total memory per node may be increased to 4 MBytes so that the kernel's memory can be increased easily

frame headers used by subsequent TopSend and TopRecv calls by this process. The TopClose call unbinds the process from the vertex, and it releases the message frame headers and any buffer attached to these headers.

The execution of a TopSend call involves three steps: (i) kernel entry, (ii) send processing and (iii) invoking the demultiplexer and the appropriate service routine. Similarly, execution of TopRecv involves kernel entry and receive processing, which involves searching for the desired packet. Thus, aside from the possible execution of services, the latencies of TopSend and TopRecv are determined by the latencies of send processing, receive processing, and demultiplexing. Specifically, in send processing a topology message header is built, and the demultiplexer is invoked. The demultiplexer then takes the same three steps for TopSend as for any other incoming topology packet (see the description above). In receive processing, a header is built and a buffer is allocated in the user's address space for receipt of a message, the output queue of the appropriate vertex is searched, and the found message is copied from the buffer located in kernel space to the user's buffer.

The measurements of TopSend and TopRecv in Tables 6.1 and 6.2 are attained on a single node of the iPSC hypercube running a single-user process by computing the total time of 1000 consecutive calls then reporting the average time of a single call. In both calls a single integer is used as a parameter, resulting in use of a single buffer of size 2 bytes. This and all subsequent timings use the node clocks with a resolution of 5 milliseconds. Timings are performed under *low-load* conditions.

Table 6.1: Latency of TopSend System Call

Steps in TopSend Call	execution time(μ-sec)
1. Kernel entry	55
2. Send processing before topology service	250
3. Demultiplexing and no-op service routine	145
Total:	450

Table 6.2: Latency of TopRecv System Call

Steps in TopRecv Call	execution time (μ-sec)
1. Kernel entry	55
2. Receive processing, two bytes buffer copy	420
Total:	475

That is, no extraneous processing or I/O is being done. The underlying Intel 286 processor has an 12 MHz clock and a cycle time of 83 nanoseconds. For comparison, a procedure call without parameters takes approximately 2.5 microseconds. Additional comparative information will be provided below whenever appropriate.

The latency of a TopRecv call when a packet is already available in the desired vertex's output queue is comparable to that of the TopSend operation as shown in Table 6.2.

The service routine used in the timing of TopRecv is a *no-op*, i.e., it does nothing.

The timings in Tables 6.1 and 6.2 demonstrate that topology messages exhibit acceptable overhead. In fact, as will be shown in the next section, topology messages may even show increased performance compared to iPSC messages because they use different message-queueing structures.

6.3 Performance of Topologies Spanning Multiple Processes

Next, we compare the performance of synchronous and asynchronous topologies with the performance of the same computational structures implemented using the message operations available in the iPSC's node operating system.

- An *asynchronous* topology is one that can execute independently of the application processes bound to it. Namely, its vertices' input conditions do not define dependencies with respect to the user processes bound to them. Such a topology may execute at the highest speed permitted by the physical communication channels and CPUs. An example of an asynchronous topology is the *shared-memory* topology in the TSP application (see Section 5.4.2) in which all vertices (with the exception of the first vertex where the new tour value is entered) may update their local copies of *best_tour* independently of the processes bound to them.

- A *synchronous* topology has vertices with input conditions that define dependencies with respect to most processes bound to them. Namely, most services cannot execute until the processes bound to them have provided inputs. In this case, the performance of the topology depends on the scheduling of the user processes bound to it. Such a topology is exemplified by the *Global-sum* topology in the FEM application.

6.3.1 A Small Synchronous Topology

The timings shown in Table 6.3 demonstrate that in the worst case (synchronous topologies not executing any services), the performance of the topology construct does not differ significantly from that of iPSC messages. Specifically, a comparison of the performance of a topology linking two processes on neighboring nodes or on the same node with the performance of the iPSC's message constructs linking two processes demonstrates that the presence of a topology slows down the *send* and corresponding *receive* operations no more than 207 to 312 microseconds, depending on message sizes. Thus, the additional overheads of demultiplexing and vertex processing incurred by each topology message compared to iPSC messages are quite acceptable. The measurements in Table 6.3 are attained by measuring the time it takes to perform a TopSend with a *no-op* service to a neighboring node, which receives the 2-byte message and returns it to the sending node, also using a *no-op* service. Thus, the timings shown in Table 6.3 are round-trip times for neighboring nodes. Again the average time of 1000 iterations is reported. Exactly one application process is resident on each participating node.

Table 6.3: A Linear Topology vs. iPSC Messages (nonlocal communications)

Number of bytes	iPSC messages (μ-sec)	Topology messages (μ-sec)
2	2340	2755
128	2515	3175
256	2685	3335
512	3330	3865
1024	4365	4980

Table 6.4: A Linear Topology vs. iPSC Messages (local communications)

Number of bytes	iPSC message (μ-sec)	Topology messages (μ-sec)
2	1085	905
512	1350	1035
1024	1620	1175

Interestingly, for message transmissions between two processes on the same node topology messages are faster than iPSC messages as shown in Table 6.4. This is because with topology messages there is only one buffer copy for local communication whereas with the iPSC messages at least two buffer copies are required.

Note that for a topology linking two processes on neighboring nodes each topology message is sent by the originating process to the local vertex, then to the remote vertex, and then to the remote process. In comparison, an iPSC message is sent directly from the originating to the remote process via buffers in the operating system kernels of each node.

6.3.2 Asynchronous Topologies

The small synchronous topology measured above does not execute any services. Typically, simple services can be executed with topologies with significant gains in performance compared to their execution by application processes using iPSC message operations. Consider an asynchronous topology resembling the *shared-memory* topology of the TSP application. This topology is a ring that links one application process to itself with a ring of vertices spanning up to 32 hypercube nodes. In this case, the topology's services simply perform routing of incoming messages (the comparison to *best_tour* is elided). The topology's mapping to the hypercube does not guarantee that all vertices are connected to their physically nearest neighbor nodes. The nontopology structure compared to the ring topology consists of up to 32 user processes, each executing on a dedicated node using the same physical and logical communication links (also not always nearest neighbor) as in the ring topology. (The topology times shown in Table 6.5 are the average round-trip times of single ring traversals measured on a 3D, 4D, and 5D cube respectively. At user-level the interval of time starting with the sending of the message onto the ring and ending with its receipt at the originating node is measured.)

Note that topologies like the ring are easily scaled to larger hypercubes. As

Table 6.5: A Ring Topology vs. User-level Ring

Cube dim.	message size (bytes)	topology (ms)	user-level (ms)
3D	2	7.75	12.52
	512	13.6	19.43
	1024	19.45	26.05
4D	2	15.4	26.17
	512	27.79	40.81
	1024	40.20	54.86
5D	2	30	55
	512	56.5	81.8
	1024	81.8	110.2

shown by the measurements in Table 6.5, increased performance improvements should be realized as cube dimensions increase.

The significant improvements in the performance of the ring topology compared to the user-level computational structure are explained as follows:

Reduced buffering. For reasons of protection, each message receipt by a user process requires a copy of the message buffer from system to user space. In comparison, a service executing in the kernel may directly access an incoming message's buffer via an address pointer. The address pointer is a segment descriptor used to translate from the kernel's logical address space to the physical addresses of the communication buffers, which is a low overhead operation.

Kernel access costs. Since services are executed at kernel level, the costs of kernel entry and exit are not accrued.

Immediate response. Since all low-granularity services like the ring-forwarding service or the *shared-memory* service operate at interrupt level, they can respond immediately to the receipt of an incoming message. Such immediate response cannot be guaranteed for user-level services, which depend on the scheduling of the processes executing them. In fact, the user-level timings shown above are best-case timings in that a typical application may have more than one process resident on each node.

Multiple message queues. Last, the iPSC message system enters all received messages in a single, multiply-linked receive queue. The receipt of such a message by a user process requires that the posted receive is matched against all packets in this queue. In comparison, a packet for a topology is either not queued at all (as with the ring topology above) or it is queued on a specific vertex, thereby avoiding extraneous searching.

Direct access to process address spaces. A last, possible advantage of topologies compared to user-level implementations of their structures is that each vertex of a topology may perform a direct access to the address space of the process bound to it. As a result, the ring topology could be extended to implement distributed shared memory in the processes bound to it by directly writing

Table 6.6: Latencies of User-memory Update

message size (bytes)	time (μ-sec)
2	95
128	130
512	230
1024	365

into reserved locations in the processes' address spaces. Such accesses require manipulations of segment descriptors, but their costs are small compared to the costs of message sending. The measurements in Table 6.6 depict latencies of writing to user from kernel space, including the costs of data movement and of the required conversions of logical to physical addresses.

The performance gains attained with direct accesses to address spaces will be evaluated jointly with TSP application's evaluation in Section 6.4.

6.3.3 Larger Synchronous Topologies

The performance advantages of topologies compared to user-level implementations are not as evident for the synchronous case. Consider the global-sum topology described in Section 5.4 in which each vertex sums the values contributed by all of its inputs. Since each vertex is associated with a user process that must contribute a value, a vertex cannot generate an output until the process' computation proceeds to the required point. Thus, the performance of the global-sum topology also

Table 6.7: A Tree Topology vs. User-level Tree

message size (bytes)	topology (ms) non-queued	topology (ms) queued	user-level(ms) (user level)
2	8.28	10.31	9.84
4	8.59	10.31	10.00
8	8.9	11.25	10.31
16	9.21	12.44	11.56
256	21.71	45.16	34.84
512	36.56	79.21	59.84

depends on the schedule of the application processes bound to it.

In Table 6.7 the global-sum implementations in queued and in nonqueued mode are evaluated. In nonqueued mode the partial sums arriving at a vertex are added as soon as they are received. In queued mode partial sums are queued and added only when all inputs to the vertex are present. This particular implementation is adding variable length vectors of 2-byte integers. Measurements are attained by first performing the global-sum operation 1000 times so that all user processes involved in the user-level implementation of the global sum or bound to the global-sum topology are executing approximately synchronously. Then, the 1001st iteration is timed. In this fashion, the performance of the global-sum construct can be evaluated without perturbations caused by differences in process schedules on individual nodes due to differences in their starting times or execution speeds. The user-level operation is similar to the nonqueued service since *add* operations are performed whenever messages are received.

As can be seen from the measurements in Table 6.7, the nonqueued global-sum topology performs significantly better than its user-level implementation. However, due to the additional overhead imposed by queueing at vertices and demultiplexing, the queued global-sum topology performs slightly worse than its user-level counterpart. This result was to be expected given the measurements of the topology mechanism presented above. However, the nonqueued topology performs consistently better than the user-level operation, and the difference in performance increases with the size of the vector being operated on. This implies that topologies supporting I/O pre- or post-processing are essential in information-intensive applications.

An interesting result not shown by the measurements above concerns the consistency in the performance of topologies vs. user-level global operations. Specifically, the iPSC's node hardware has the unfortunate property (which has been corrected in the 386-based implementation of Intel's hypercube) of having a very small effective bandwidth of the connection from the physical channels to main memory. This forces the node operating system to service one physical channel at a time. If more than one channel is actually active (e.g., trying to provide inputs to a vertex resident at a node), packets are lost, causing timeouts and retransmission at the sending node. In the current iPSC operating system, timeout values are quite high (starting at 10 milliseconds) and vary from channel to channel.

For the implementation of the global-sum topology, this implies that the upper

Table 6.8: Best and Worst Performances of the Global-sum Topology

message size (bytes)	topology (ms) nonqueued		topology (ms) queued	
	best	worst	best	worst
2	8.47	21.52	10.63	10.78
4	8.44	17.41	10.73	11.25
8	8.75	28.48	11.25	11.52
16	9.1	39.84	12.03	12.66
256	21.25	23.13	45.47	45.91
512	37.09	37.34	79.73	80.63

nodes in the *InvTree* may experience the loss and subsequent retransmission of incoming messages, which in turn causes some global-sum operations to be much slower than others. Essentially one cannot *push the hardware too hard.* This is demonstrated by the measurements in Table 6.8 in which the best and worst times of 1, 2, 3, 4, 5, 10, and 100 iterations are reported. As can be seen, the slower, queued topology shows more consistent performance than the nonqueued topology.

The global-sum measurements shown in Table 6.8 also imply that future multicomputer hardware must increase the effective bandwidth of physical communication channels to the main memory of each node. Otherwise, performance gains due to increased available total bandwidth of communication channels and to increased processor speeds will not be realized.

6.4 Performance and Usefulness of Topologies with Application Programs

In the introduction, we state that global data and operation pose one obstacle to improving the performance of parallel applications. In this section, the performance effects of the topology construct are evaluated for two different application programs—the FEM and the TSP applications.

6.4.1 The FEM Application.

In the FEM application's solution phase each solver process spends approximately 50 percent to 69 percent of its time performing global norm computations when using the user-level implementation of global sum. The measurements in Table 6.9 demonstrate that performance improvements may not always be possible even when using the nonqueued topology construct. This is because the topology must execute entirely synchronously with the application's processes. In addition, and as shown in Section 6.3, the nonqueued implementation of global sum exhibits occasional aberrations in performance because it is *driving the hardware too hard.*

The solution time of the application with and without topologies are tabulated in Table 6.9. Both worst and best times observed for the global-sum topology over 3 to 6 runs are shown.

Tolerance is the tolerance value used in the iterative method for the FEM's solution step. The total number of actual global-sum operations performed is 72

Table 6.9: Solution Times for the FEM Application

Mesh-size	Cube-size	Tolerance	User-level(sec)	Topology(sec)
15 x 20	4-D	0.00001	6.64	6.59 (6.78)
11 x 36	4-D	0.00001	10.66	11.28 (11.37)
	5-D	0.00001	7.89	7.47 (7.83)

for mesh-size 15x20 and 99 for 11x36.

It is interesting to note that use of the global-sum topology typically improves performance slightly. As expected, inconsistencies in the execution times of the FEM's solution step are due to packet retransmission. This is substantiated by the retransmission counts for the 11x36 mesh on a 4d cube shown in Table 6.10 on processors 1 and 2, which are next to the root (processor 0) of *InvTree*. This table shows the channels which have the maximum number of retransmission counts in each run for three different runs. For these measurements 99 global-sum operations were performed. Note that collisions on a communications channel with a higher timeout value result in worse performance than collisions on a channel with a lower timeout value. (See the difference in performance viz. the total number of retransmissions on the channel from node 2 to node 0 vs. the channel on node 1 to node 0.) This suggests that the current iPSC architecture is too sensitive to the proper choice of time-out values. Furthermore, it suggests that future hypercube architectures should offer hardware-implemented flow control between multiple channels connected to a single node, thereby avoiding collisions and the resulting

Table 6.10: Maximum Retransmission Counts of the FEM Application

Ch num	retrans.count	Time-Out (ms)	Solution (sec)
Node 2 to 0	7	20	11.15
Node 2 to 0	9	20	11.44
Node 1 to 0	11	10	11.41

expensive software-controlled time-outs and retransmissions.

The measured solution times shown in Table 6.10 differ slightly from those shown in the previous table due to the instrumentation required to collect retransmission counts.

6.4.2 The TSP Application.

As described in Section 4.3, the TSP application has two global data and operations: (i) global shared memory (*best_tour*) and (ii) work sharing among searcher processes. In this section the performance of TSP with and without the use of topologies are presented. First, the performance of the global shared-memory topology is evaluated as a stand-alone and in conjunction with the TSP application.

The global shared-memory topology is a nearest-neighbor ring connecting vertices resident on all nodes on which searcher processes are located. Service routines in those vertices perform forwarding of *best_tour* values and update the copies of *best_tour* values maintained in variables in the address spaces of searcher processes.

Table 6.11: Performance of A Shared-memory Ring Topology

Num of bytes	User-level (ms)	Topology (ms)	Shared-memory Update(ms)
2	35.05	17.80	19.95
4	35.05	17.90	20.20
8	35.20	17.95	20.15
128	39.30	21.60	24.50
256	43.60	24.75	29.15
512	52.30	33.09	38.55
1024	69.75	47.70	56.00

The measurements in Table 6.11 report the times required for one searcher process to simply send values around the entire ring (labelled *Topology*) or to send values and also perform the global update of *best_tour* variables in searcher processes (labelled *Shared-Memory Update*). The ring spans 32 nodes.

Note that the shared-memory topology performs consistently better than its user-level counterpart. Furthermore, the update of *best_tour* variables in searcher processes' address spaces increases the topology's execution time by less than 20 percent.

Interestingly, measurement of the actual TSP application with the shared-memory topology or with a single, centralized coordinator process (and processor) performing the sharing of *best_tour* demonstrates that performance may not always be improved as expected. This is because this version of TSP only improves performance of one of its two global data and operations, where increased pruning

Table 6.12: Completion Times of the TSP Application

Prob. size	num. of searcher	num of better tours found	Avg time (sec)	Avg num of work-sharing msgs
without topology				
25	4	3	35.11	18
25	12	8	49.34	88
30	4	3	423.74	30
30	12	3	28.26	34
with shared-memory topology				
25	4	3	35.21	25
25	12	6	49.99	94
30	4	3	426.54	35
30	12	3	27.86	42
with shared-memory and work-sharing topologies				
25	4	4	36.00	10
25	12	3	26.87	27
30	4	3	423.66	43
30	12	3	24.36	26

activity due to improved speed of global memory leads to increased worksharing (see the 'Avg num of work-sharing messages' in the versions with and without the shared-memory topology), which constitutes its second, nonoptimized global data and operation.

The timings in Table 6.12 are nondeterministic due to the probabilistic nature of the searching problem. However, the numbers shown here, which are averaged over 3 to 6 runs, are indicative of the trend of increase in work sharing when improving the speed of updating the global memory. Measured timings fluctuate from 1 to 6 percent, and the number of work-sharing messages varies from 1 to 20 percent (in very few cases) of the corresponding average values.

The third set of timing measurements using both the *shared-memory* and the *work-sharing* objects indicates that, in general, there has been improvement in performance when both global data and operations are optimized. Except for the case of the largest *number of better tours found*, execution times are improved from 1 percent to 53 percent. The case in which the application with distributed objects performs worse than the one using the coordinator is explained by the larger number of better tour values being passed. This indicates that *low tour* does not reach its final value as rapidly as the nonglobal queue version, thereby causing more useless work. Thus, the performance improvements shown above are not only due to the use of the two distributed objects but also result from complicated interactions among (i) pruning caused by *lowtour* values, (ii) the selection of *best* work units, and (iii) global queue accesses. Direct comparisons with the results shown in Section 6.3 are not possible since the performance effects of the *work-sharing* and *shared-memory* objects cannot be isolated. However, with the *work-sharing* object, good performance is attained even when the total number of work-sharing messages is large (as seen in the test configuration with problem size 30 on 4 processors).

Note that the current implementation of the *work-sharing* object indirectly affects performance by providing a time-dependent ordering of work-sharing messages: a work request is filled by the first process that has shareable work. This implementation should be outperformed by one that provides a *fair* queue in which each process having work is assumed to have equal probability of containing the solution.

6.5 Conclusions

Implementation of and experimentation with the topologies in this chapter provide several interesting results. First, it shows that the topologies are realizable constructs. Second, the results indicate that with topologies, performance improvement is possible. Third, by allowing the programmer to design the distributed objects separately from the user processes, it facilitates programming and encourages modular development. However, debugging topologies is no less complicated than debugging communications protocols. It is especially difficult to trace the execution paths of messages routed by the fine-grain services since such messages travel faster than the error-logging messages themselves. This leads us to suggest the implementation of *debugging topologies* or specialized tools for debugging topologies. Also, topologies' communication structures and their mappings to the multicomputer are difficult to keep track of when the number of processors grows. To alleviate these problems, an environment is needed to support the programming of topologies. We next describe an environment called PRISM which provides a set of integrated tools for programming topologies on structured multicomputers.

CHAPTER VII

PRISM—A Programming Environment for Topologies

This chapter describes PRISM—a PRogrammIng System for Structured Multicomputers. PRISM is designed to support the programming and debugging of topologies. Its implementation focuses on program debugging and visualization.

7.1 A Structured Multicomputer Programming Environment

7.1.1 System Requirements

A parallel programming environment for multicomputers must provide programming support for improving the performance of parallel programs in addition to standard support such as programming languages, debugging support, and user interfaces. For instance, it must facilitate the manipulation of a parallel program's processes, communication structures, and program monitoring to permit efficient use of the underlying architecture. In addition, various heuristics for mapping the parallel applications to the underlying architecture must be supported. Program manipulation, mapping, and monitoring must share information [75, 61] and operate coherently. In contrast to other research [7], our emphasis regarding programming support for structured multicomputers is the development of facilities for the efficient use of topologies' complex communication structures.

The PRogrammIng System for Structured Multicomputers (PRISM) addresses the development of high-performance parallel programs for structured multicomputers. PRISM is based, in part,on results of the ISSOS project at the Ohio State University [37]. Below, the features of PRISM that naturally evolve from ISSOS are described, followed by a discussion of its unique attributes.

Although the main focus of ISSOS is to write statically and dynamically adaptable programs for real-time and long-running applications on multiprocessors and loosely coupled networks of processors, many of its concepts are applicable to structured multicomputers. In all such environments, parallel programming implies experimentation involving static adaptations like mapping and remapping an application and dynamic adaptations like dynamic load balancing. Similarly, program tuning abounds, such as changing a multicomputer application to accumulate data and sending it as a single message rather than as several messages. Another concept originated with the ISSOS project [68, 73] is that of operating software, which is defined as application software integrated with the operating system components. Fine grain service routines in topologies linked together with the operating system are examples of such operating software for multicomputers.

In addition, research on program instrumentation and performance monitoring [55], as well as graphical tools [48] is applicable to the PRISM system. Lastly and most importantly, the integration of multiple tools in PRISM can be based on past research in ISSOS, which utilizes an active database [61].

Unique to PRISM is its support for the development and testing of topologies because PRISM addresses structured multicomputers and not shared-memory multicomputers and networked workstations.

7.1.2 Basic Features of PRISM

PRISM provides a set of tools for writing programs, monitoring them, and displaying certain program attributes. Multiple levels of programming languages are used for programming different portions of a parallel program. At the highest level each program's parallel structure is described; at the next level program modules are written using a procedural language such as C while the interface between different modules is described with a data manipulation language.

Programming Paradigm

PRISM describes a parallel program as a set of processes interacting through distributed abstract objects. The distributed objects provide abstractions hiding the underlying architecture of the parallel machine from the processes while their implementations may explicitly take advantage of the architecture.

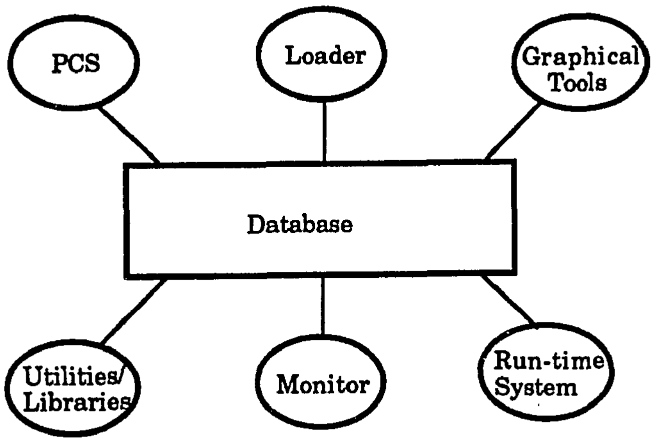


Figure 7.1: The PRISM Programming Environment

Components of PRISM

The PRISM system consists of the following components integrated through the database process as shown in figure 7.1.

Program Construction System: It consists of C and Fortran compilers, a linker for processes, and a topology compiler. Direct support of a concurrent language may be implemented on top of these basic tools.

Loader: Two kinds of loaders are available: a loader for loading processes and a loader for loading communication structures. Processes can be loaded on a given processor or on a given set of processors. The topology loader can load the communication structures independently of the processes that will use them.

Database: The database is an active database that integrates the various tools shown in Figure 7.1. Operations on the database can be performed using a data-manipulation language (DML).

Graphical Tool: The Graphical tool provides displays of various aspects of a program. It consists of a set of icons that represent the information to be displayed, and a library of graphics routines. Specific displays are generated by means of a user-written DML program.

Monitor: It monitors the program execution as specified by the DML program. It consists of a local monitor residing on each node processor and a central monitor on the host processor.

Utilities, Libraries, etc.: These consist of a set of routines for implementing heuristics for mapping standard problem structures and packaged topologies.

Run-time System: It supports the process abstraction and interprocess communications including topologies and implements the kernel interface through a set of system calls.

It has been argued in Chapter 3 that to obtain high performance, global data and operations must be programmed efficiently. The aspects of PRISM described in this chapter concern the programming and display of topologies that implement such global data and operation. Specifically we describe (i) how PRISM supports the programming of topologies using a topology compiler and a loader and (ii) how programmed topologies may be manipulated (creation, loading, resetting) and visualized using a high-level data-manipulation language.

Concerning (i) and (ii), our approach differs significantly from past work such as Poker in that we do not choose to program graphically for three reasons. First, although visually appealing, it is tedious to explicitly construct a large number of processes and the associated process-to-process connections. Second, once a process structure is constructed, unlike VLSI design, each element (process) is likely to make use of several topologies (communication links) so that a single screen is likely to contain a confusing number of overlapping communication links [83]. Third, it is difficult to specify graphically the application-dependent semantics that may be associated with communication links [86]. We conclude that a concise data model offering powerful operations for extraction, display, and highlighting of processes and communication structures is required. For example, consider a tree topology for which one might wish to monitor and display the queue length of the second-level vertices mapped to a set of processors. A graphical primitive to perform this task should be useful for the monitoring and display of queue lengths in a ring topology as well. Thus, such a primitive cannot be application specific. Instead it must be based on a uniform model of representations of topology structures, their mappings to the parallel machine and the machine's structure. We choose the entity-relationship model for this purpose. The entity-relationship model of data representation has the power and expressiveness to represent all necessary information about the target system. An added advantage is that it allows attributes that characterize an entity to be separated from those attributes that characterize the entity's relationships with other entities.

7.2 Programming Topologies

PRISM allows the programmer to build topologies by specifying them in an intermediate language, and by compiling and loading them. It offers

1. a topology compiler
2. a loader
3. mapping of a topology
4. a vertex-monitoring utility and
5. a system call library for services.

7.2.1 Topology Compiler

The topology compiler takes as input a topology specification and generates as output an initialization program that loads the kernel data structures for the topology on each node. It also generates a set of header files that defines the service routines and their names, which are to be linked and loaded with the node operating system or with the application program depending upon the service granularity. The C-code for generating the structure is generated as a stub routine that is called by a system-provided driver. The structure of the stub routine generated by the compiler is shown below. The '...' indicates unnecessary details that have been elided. In the stub routine initializing the vertex data structures involves setting the topology id, input connections, output connections, input conditions, output conditions, vertex name, stack size, and data size.

```
buildTop()
{
        switch (mynode()) of {
                0: /* executed on processor 0 */
                        for all vertices on processor 0
                        initialize the vertex data structures
                        break;
                1: /* executed on processor 1 */
                        for all vertices on processor 1
                        initialize the vertex data structures
                        break;
                2: /* executed on processor 2 */
                        for all vertices on processor 2
                        initialize the vertex data structures
                        ...
                        ...
        }
}
```

Apparently, for a large number of processors, the above stub routine can become fairly large. In that case, the user must provide a function which returns a list of input vertices and a list of output vertices given the vertex name. For example, a function for a ring topology connecting vertex name n to n-1 and n+1 will be as follows:

```
ring(n, inputvids, NumofInputs, outputvids, NumofOutputs)
int n, inputvids[], outputvids[];
{
        inputvid[0] = (n + 1) % total_vids; /* total_vids is a constant */
        outputvid[0] = (n - 1) % total_vids;
        NumofInputs = 1 ;
        NumofOutputs = 1;
}
```

In this case, the mapping of the vertices to the nodes must also be provided as a function or a mapping array which maps the vertices to the node processors.

7.2.2 Mapping of a Topology

Mapping of a topology to processors can be defined as part of the topology's specification. Mapping is done by assigning vertices to processors with a goal to optimize the execution time of the global operation the topology is implementing. As a result, a logical link between two vertices may span more than one physical link, and the default routing scheme used by the iPSC kernel ensures that messages are sent to the right destination. The topology compiler takes the mapping specification into account when generating the code so that each vertex is mapped to the right processor.

For example, a mapping for a ring topology with vertex identifiers 0 to 5 onto an 8-processor hypercube can be stated as part of the topology's specification as follows. As described in Chapter 5, the mapping specification has an identifier which may be used by more than one topology.

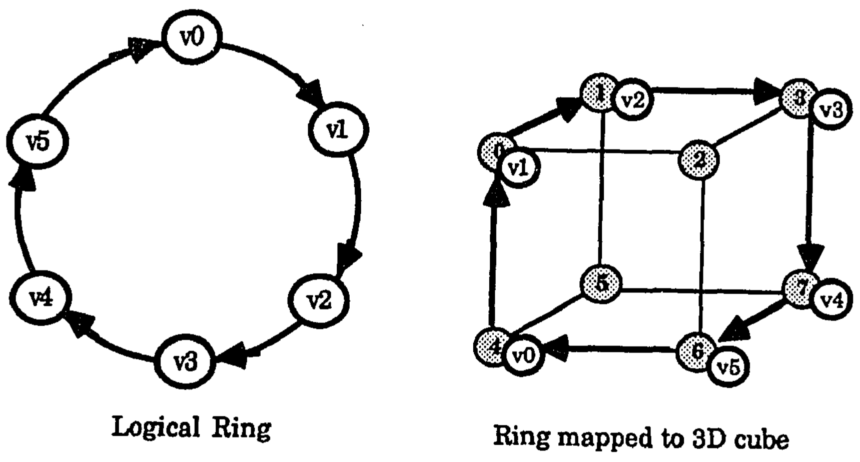


Figure 7.2: Logical and Mapped Ring

M1 **is** **map**{(0:4);(1:0);(2:1);(3:3);(4:7);(5:6)}

Ring **Topology is** RingTopologyType **and** M1

Here the first number in each tuple of the mapping description is the vertex name and the second number is the processor number. In the ring the vertex i is connected to i+1 modulo 5. The resulting ring topology is shown in the figure 7.2.2 where the physical links involves are marked.

7.2.3 Loader

The loader implements the loading of two different abstractions—processes and topologies. The loader for loading the node processes runs on the host machine and communicates with the local loader process on each node processor. This part of the loader is functionally the same as the loader provided on the Intel iPSC hypercube.

The topology loader loads a topology onto the hypercube by loading the programs generated by the topology compiler or written by the user onto the node processors and executing them. The programs register the topology in the kernels and set up the kernel data structures for the vertices. The loader can also remove previously loaded topologies.

In loading the services, the fine-grain service routines must be compiled together with the operating system and loaded as a unit with the operating system. The process loader can be used to load medium-grain and large-grain services.

7.2.4 A Vertex-monitoring Utility

A Vertex Monitor is a utility for debugging topologies. It is basically a resident monitor for the vertex on a node processor. It returns the information regarding the status of the vertex such as its internal queue sizes, input conditions, outgoing vertices, and incoming vertices. It retrieves the status of the vertex that has been requested on a certain processor node and sends it back to the host for inspection. The resident monitor can be loaded dynamically while the application is in execution.

The monitoring command invoked by the user specifies the topology id *tid* and a vertex name *vid* on a given processor as follows:

vinfo *vid tid processor-number*

This command causes a resident monitor to be loaded on the given processor, sends request messages to the monitor to retrieve the desired information and receives the information which is then presented to the user. If the processor number is not specified, a resident monitor is loaded onto each processor, and the monitors that cannot locate the vertex terminate. Although this type of command monitors only a single vertex at a time, it can be used in conjunction with a DML program to retrieve information from a set of processors.

7.2.5 Support System Calls for Services

A system call library provides calls for writing service routines, such as calls for mapping memory from the user or system address space to message buffers, filling the message headers, allocating and deallocating memory descriptors, and loading topologies from within a user program. The system calls can be grouped into (i) memory-mapping and -unmapping calls (ii) message-manipulation calls and (iii) service-binding calls.

Memory-mapping calls are used to allocate and initialize memory descriptors so that service routines can directly access a buffer or the address space of a user process without violating privilege-level protection. This allows accessing the user buffer or the system buffer directly and avoids buffer copying or repeatedly making system calls to access a certain memory location in the user space. Since the memory descriptors are a scarce resource, they must be deallocated after the activation of the routine. The unmapped_ptr() call is provided for this purpose.

Message-manipulation commands are used for setting up the fields in the message header and allow the service routines to closely control buffer copying and certain attributes of the communication protocol like routing and acknowledgement.

Service-binding calls allow medium- and large-grain services which are loaded dynamically to be linked to the operating system. They are bound by noting their entry points, service identifiers, and stack- and data-segments addresses in the kernel's topology-related data structures.

7.3 Manipulation of Programmed Topologies

A parallel program may have a set of attributes of interest to the user. Such a set of attributes is called a *program view.* A program view may consist of program module names, their interconnections or their interaction attributes concerned with other modules, etc. An interesting view of a parallel program is one describing its topologies. By representing the topologies as a tuple in a relational database and by using a data-manipulation language, various operations can be performed on the topologies.

Topology manipulation is performed using two components as shown in figure 7.3:

1. Topology management: Once programmed, the low-level tools described in

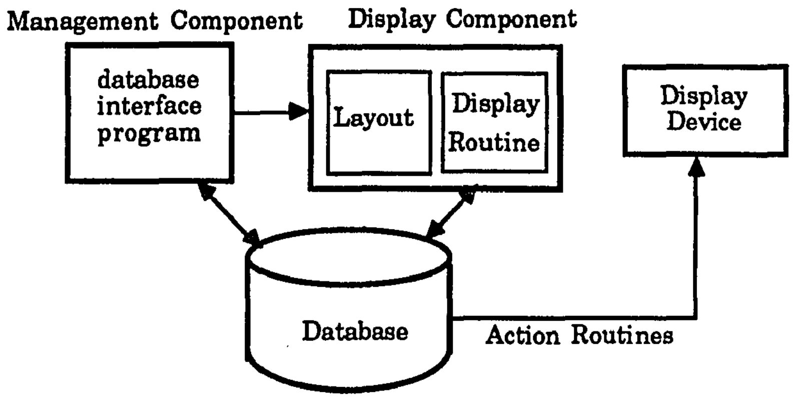


Figure 7.3: Topology Manipulation Tools

section 7.2 may be activated and controlled via the database, thus removing from the programmer detailed knowledge regarding topology construction. Specifically, a topology can be generated, compiled, and loaded through the database. A user-written DML program specifies high-level commands such as creation, loading, resetting, or removal of topologies.

2. Graphical views of the topologies: A graphical view is a graphical representation of a specific program view. In the case of a topology, such a view may contain different levels of granularity or abstraction. For example, it can display the complete structure or just a single vertex. In addition, it may convey different kinds of information about each display item.

Note that 1 and 2 are separated on purpose, because the manipulation of the program entities (topology components) is a separate activity from their display. Furthermore, there may be more than one type of display for a given entity, depending on the information to be viewed, or the same type of displays may be

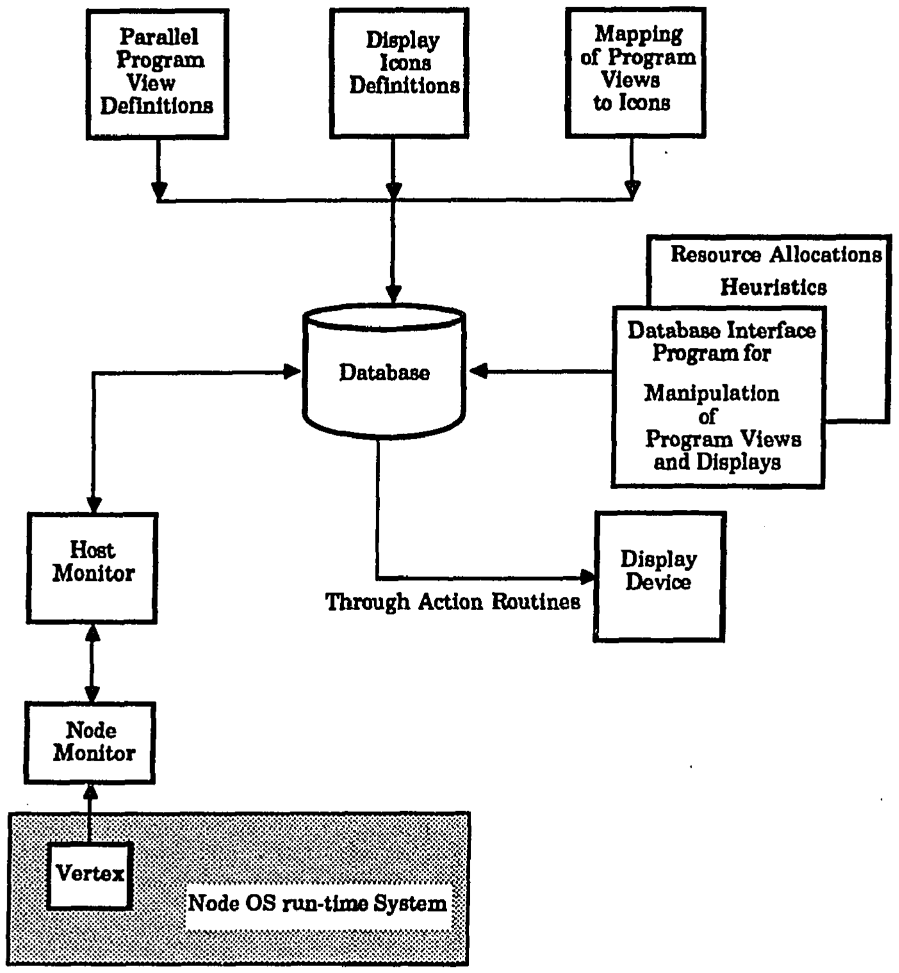

Parallel Program View Definitions
Display Icons Definitions
Mapping of Program Views to Icons
Resource Allocations Heuristics
Database Interface Program for Manipulation of Program Views and Displays
Database
Host Monitor
Display Device
Through Action Routines
Node Monitor
Vertex
Node OS run-time System

Figure 7.4: High-level and Low-level Tools Integrated Through the Database

shared among similar program entities.

7.3.1 The Database Model and Interface

The data model of the database for the PRISM environment is an *augmented Entity-Relationship* (E-R) model. The E-R model represents topologies and their graphical views using E-entities, R-entities and S-entities explained in the following section. Interface to the database is through a data-manipulation language which provides a set of operations that can be performed on the database. Below is the description of the entity-relationship model and the database operations provided in the database. The operations are described in terms of C procedures.

7.3.2 Entities, Relationships, and Sets

An *E-entity* is a collection of the attributes of a parallel program's components. It is similar to a record structure in a programming language with fields as attributes. The value of one of the attributes may serve as the identifier of an entity. Except for the identifier, the values of other attributes may vary during the existence of the E-entity.

Information about relationships among one or more components is represented by a *Relationship* relation. In the E-R model a relationship relation is represented by an *R-entity.* In addition to the identities of the E-entities involved in the relationship, an R-entity may also have attributes that characterize the relationship.

An *S-entity* represents a collection of entities of pre-defined types. They may be E-entities, R-entities, or S-entities. An S-entity may also have simple attributes related to the set.

Every entity in the system is an instance of a programmer-defined type called a *Schema.* A schema may be considered a template for an entity, which defines whether the entity is an E-entity, an R-entity, or an S-entity. In addition, the schema also defines the types and names of attributes for E-, R-, and S-entities and the schemas of component entities in the case of R- and S- entities.

7.3.3 Action Routines

An *Action Routine* is a trigger that may be associated with a type of schema. When a database operation such as create, remove, or update is performed on a relation in the database, this piece of code is executed to perform some set of semantic actions on the database as a separate thread of execution. The routine receives information on the type of database operation to be performed and on the input values of the operation, allowing it to make decisions based on these parameters. It can also access another tuple in the database, causing another action routine to fire, and this chain of actions can propagate even further. This characteristic of the action routine can be used to ensure the semantic consistency among the related tuples in the database or to initiate a graphical display of an icon associated with a tuple.

The general form of an action routine consists of a trigger condition, pre-operation part, and a post-operation part as shown below.

```
if conditions of trigger are satisfied then
        execute the pre-operation part of action routine
        perform the database operation
        execute the post-operation part of the action routine
```

7.3.4 Database Operations

Database operations are described below in terms of C procedure calls.

Operations on Schemas: To register a type of Schema with the database:

```
_tdb_schema_reg("schema_name","schema_type",
                "contains_list","relates_list",
                "attributes_list",Action_Routine_Name)
```

In registering a schema definition the *relates_list* is valid only for an R-schema while the *contains_list* is valid only for an S-schema. For example, the following calls register in the database an E-schema, R-schema, and S-schema respectively.

```
_tdb_schema_reg("VertexSchema","e","","",
              "services:int[5];vid:int;queues:int[5];",
              VertexActionRoutine);


_tdb_schema_reg("SetOfVertices","s","Vertices","",
              "Topologyid:int",
              SetofVeticesActionRoutine);


_tdb_schema_reg("ProcesstoPRocessorMap","r","",
              "ProcessID → ProcessorID",
```

MapActionRoutine);

The **_tdb_ent_getschema** call returns the schema type of an already created entity. Here, _tdb_enttype is a database entity type.

_tdb_ent_getschema(entity_name)

where:

entity_name: _tdb_enttype;

Operations on Entities: All types of entities can be created by the **_tdb-_ent_create** call and can be removed by the **tdb_ent_delete** call.

1. _tdb_ent_create("SchemaType,"ExtraAttributes");

   where:

   SchemaType: Name of a previously registered schema.

   ExtraAttributes:List of any entity-specific attributes(if any).

2. _tdb_ent_delete("EntityName");

   where:

   EntityName: Name of a previously created entity.

Operations on Attributes: There are two operations that can be performed on attributes:

1. _tdb_get_attr("EntityName","AttributeName",Index)

   where:

EntityName:	A previously created entity.
AttributeName:	Name of the attribute.
Index:	Integer offset if AttributeName represents an array.

2. _tdb_set_attr("EntityName","AttributeName",Index)

   where:

EntityName:	A previously created entity.
AttributeName:	Name of the attribute.
Index:	Integer offset if AttributeName represents an array.

Operations on R-Entities: The **_tdb_set_link** and **_tdb_ent_get** are two operations unique to R-entities.

The _tdb_ent_setlink() call can be used to set up a link from the R_Entity to a component entity. Both the R_Entity and the component entity must exist.

_tdb_ent_setlink(R_EntityName, "Field_Name", Component_Entity)

where:

R_EntityName:	A previously created R-entity.
Field_Name:	Field to which to link.
Component_Entity:	A previously created entity that is to be linked in.

The _tdb_ent_get() call returns the identifier of the entity that is linked to the component field of an R-Entity.

_tdb_ent_get(StartEntity, PtrChain)

where:

StartEntity:	A previously created R-Entity.
PtrChain:	A string of the form "FieldName→".

Operations on S-Entities: The operations defined on S-entities allow creation and initialization of a set; accessing an element of the set; and union, intersection, difference, and assignment operations between two sets.

1. Initialize a set to a NULL set. The following call returns a NULL set and is used in conjunction with _tdb_set_assign() call described later.

```
_tdb_set_init()
```

2. To determine if an element E is in Set S. If it is, the call will return true.

```
_tdb_set_inset(S, E)
```

3. Remove an element from a set S, returning a pointer to the element. Repeated remove calls return the elements of a set in sequence.

```
_tdb_set_removeelt(S)
```

4. Union of Sets S1 and S2, creating a new set. The call returns a pointer to a new set.

```
_tdb_set_union(S1, S2)
                          returns:  pointer to a new set.
```

5. Intersection of Set S1 and S2, creating a new set. The call returns a pointer to a new set.

```
_tdb_set_inter(S1, S2)
```

returns: pointer to a new set.

6. Difference of Set S1 and S2, creating a new set. The call returns a pointer to a new set.

   _tdb_set_diff(S1, S2)

7. Create a new set that contains all elements of Set S.

   _tdb_ent_ALL(S)

8. Return a new set that contains all created entities of schema type SchemaName.

   _tdb_sch_ALL(SchemaName)

   where:

   SchemaName: A previously registered schema

9. Return a new singleton set that has an Entity E as its only element.

   _tdb_set_makeset(EntityName)

where:

EntityName: Name of a previously created entity.

10. Assign all elements in a set S1 to the S-Entity S2.

    _tdb_set_assign(S2, S1)

7.4 Topologies and Their Graphical Views in E-R Model

All components of a parallel application in general are represented in the E-R model as entities, relationships, or sets. Topologies consist of sets of vertices and edges which are represented by a combination of E-, R-, and S-entities. The attributes of interest are defined in the schema of these entities. The topologies can be stored in the database together with the information about the parallel machine they are mapped to. The latter information is also represented as E-, R-, or S-entities. Furthermore, the mapping of the topologies to the machine is a relationship entity definable as an R-entity. Lastly, the graphical view of the topologies can be represented in the database as sets of icons defined as entities. Thus the entities for representing the topologies and their graphical views can be grouped into five kinds:

1. Entities representing the topologies

2. Entities representing the parallel machine

3. Entities representing the mapping of topologies to the parallel machine

4. Entities representing the graphical icons depicting the above abstractions

5. Entities representing the mapping of topologies and the parallel machine to graphical icons

7.4.1 Representing Vertices and Edges

The lowest-level representation of a topology is a vertex and an edge. A vertex is represented as an entity and an edge is defined as a relationship as follows:

```
Vertex = e-schema
        attributes   vid:       int;
                     incond:    int;
                     outcond:   int;
                     inlist:    int[5];
                     outlist:   int[5];
                     stacksize:int;
                     datasize:  int;
        action       none;

Edges = r-schema
        relates      dst_vtx → Vertex;
                     src_vtx → Vertex;
        attributes   eid:       int;
                     Ack:       int;
                     FlowControl:      int ;
        action       none;
```

Vertices are grouped into a set of vertices and the edges are grouped into a set of edges. A set consisting of these two sets is defined as a topology schema.

```
vertex_set = s-schema
          contains     Vertex;
          attributes   none;
          action       none;

edge_set = s-schema
          contains     Edges;
          attributes   none;
          action       none;

Topology = s-schema
          contains     Set_of_Vertices;
                       Set_of_Edges ;
          attributes   Topology_id:     int;
          action       none;
```

7.4.2 Representing Processors and Communication Links

Similarly, processors and physical communication links can be represented as E-entities and R-entities respectively. A set consisting of the sets of these two entities can then represent the hypercube machine.

```
Processor = e-schema
          attributes   proc_id: int;
                       vector_board:    int;
                       mem_size:        int;
                       action   none;

Commlink = r-schema
          relates      proc1 → Processor;
                       proc2 → Processor;
          attributes   link_id: int;
                       router_present:  int;
                       retransmission_count:    int ;
          action       none;

SetofProcessor = s-schema
          contains     Processor;
          attributes  none;
          action      none;

SetofLinks = s-schema
          contains    CommLinks;
          attributes  link_id: int;
          action      none;

Multicomputer = s-schema
          contains    SetofProcessors;
                      SetofLinks;
          attributes  Type: string;
                      Dimension: int;
          action      none;
```

7.4.3 Representing Mapping of a Topology to Processors

Each mapping of a vertex to a processor is defined as an R-entity. A group of mappings are organized as an S-entity.

```
VtoPmap_Schema = r-schema
          relates     vid → Processor;
          attributes  vtxid:   int;
          action      none;

SetofVPmap = s-schema
          contains    VtoPmap_Schema;
          action      none;
```

7.4.4 Representing Graphical Icons

A graphical icon is represented as a tuple in the augmented E-R database. Each icon consists of two components: graphical attributes and icon-drawing components. The graphical attributes are a set of attributes which are modifiable by the application in the course of its execution and display. The icon-drawing component is an active component which actually does the drawing on the display screen. It consists of a code segment that retrieves the attribute values, performs required computations, and makes calls to the graphics routines to display the icon.

Icons represented as database entities may be created, displayed, and deleted. An icon's various graphical attributes such as the color, position, and mode of display may be modified using the database operations.

A more complicated display for a S-entity, which has an icon for each of its elements, can be constructed by building a set of icons and a representative icon for the S-entity itself. Each element of the set is responsible for creating its own display. The icon set also has two components as above—one that consists of display elements and one that actually does the drawing, in this case scanning through its own list of icon elements and calling the display command to create the display.

A general representation of an icon is as follows:

```
Icon_Name = e-schema
          attributes
                       graphical attributes that may be modified;
          action
                       Icon_Drawing_Routine;
```

The actual drawing of an entity is performed by the action routine. The basic structure of the action routine for drawing an icon is as follows. The *display_flag*, one of the attributes of an icon entity, is used to indicate if the display is inactive, to be initiated, or already active but modifiable.

```
if action-routine-disable-flag is set
        return
if database-operation is set-attribute and the attribute is display_flag
        if attribute-value is 0
                return
        elseif attribute-value is 1
                disable further activation of the action routines
                retrieve values of other attributes
                enable further activation of the action routines
                draw the icon
else
        retrieve value of the attribute display_flag
        if  display_flag is 2
                disable further activation of the action routines
                retrieve values of other attributes if needed
                enable further activation of the action routines
                draw changes to the icon
```

Layout of the overall display is crucial for aesthetical displays as well as for conveying information. Since automatic layout of arbitrary graphs is not the focus of this thesis, it is not addressed here. It is left to the programmer to set up the layout as desired.

The following graphic icons defined as E-entities represents the vertices, edges, processors, and physical communication links.

```
Vtx_icon1 = e-schema
        attributes  xpos:      int;
                    ypos:      int;
                    diameter:int ;
                    magnification:    int;
                    color:     int ;
                    vid :      int;
                    display_flag:      int;
        action      DrawVertex1;

Vtx_icon2 = e-schema
        attributes  xpos:     int;
                    ypos:     int;
                    magnification:     int;
                    color:    int ;
                    vid:      int;
                    display_flag:      int;
        action      DrawVertex2;

Edge_icon1 = e-schema
        attributes  src_xpos: int;
                    src_ypos: int;
                    dst_xpos:int;
                    dst_ypos:int;
                    color:    int;
                    display_flag:      int;
        action      DrawEdge1;

Edge_icon2 = e-schema
        attributes  corner_xpos:       int;
                    corner_ypos:       int;
                    display_flag:      int;
        action      DrawEdge2;

Proc_icon = e-schema
        attributes  xpos:     int;
                    ypos:     int;
                    diameter:int ;
                    magnification:     int;
                    color:    int ;
                    display_flag:      int;
        action      DrawProc;

Commlink_icon = e-schema
        contains    src_xpos: int;
                    src_ypos: int;
                    dst_xpos:int;
                    dst_ypos:int;
                    linktype: int;
```

```
                length:  int ;
                color:   int ;
                display_flag:     int;
      action    DrawLinks;
```

7.4.5 Mapping E-, R-, and S-Entities to Graphical Entities

Topologies are represented as E-entities, R-entities, S-entities or a combination of these entities. Each of them can be associated with an icon entity representing the graphical icon. The mapping of the topology entities to their icons can be done by performing a relational join on the set of topology entities and the set of icons representing them. For instance, the graphical icon to which a vertex is mapped can be determined by a relational join on the set of vertices and the set of icons. The mapping can also be done by defining R-entities, which map from topology entities, to their respective icons. Defining R-entities is appropriate when the mappings remain unchanged of the times. Searching through R-entities can also be less costly than performing a join, and mapping to different types of icons can be easily done by defining one set of R-entities for each mapping. In PRISM the mapping is implemented using this approach.

The R-entities used for mapping must relate not just an entity to an icon but must also relate an icon to an entity. This is because the icon drawing routine needs to access the values of the entity and then convert them to display screen commands. Thus the mapping entity is bi-directional.

As yet another alternative for having a set of R-entities for mapping the topological entities to their graphical representations, it is possible to have an attribute in the topology entity that indicates the icon associated with it. However, icons are not logical attributes of a topological entity in general; also, topological entities may not always have an icon, and there may be more than one icon associated with each entity.

```
VtoViconmap_Schema1 = r-schema
        relates     Viconptr → Vtx_icon1;
                    Vtxptr → Vertex;
        attributes  vid:     int;
                    iid:     int;
        action      none;

VtoViconmap_Schema2 = r-schema
        relates     Viconptr → Vtx_icon2;
                    Vptr → Vertex;
        attributes  vid:     int;
                    iid:     int;
        action      none;

PtoPiconmap_Schema = r-schema
        relates     Piconptr → Proc_Icon;
                    Pptr → Processor;
        attributes  procid:  int;
                    iid:     int;
        action      none;

CtoCiconmap_Schema = r-schema
        relates     CtoCiconptr → Commlink_Icon;
                    Cptr → Commlink;
        attributes  Clink:   int;
                    iid:     int;
        action      none;

EtoEImap_Schema1 = r-schema
        relates     Eiconptr → Edge_Icon1;
                    Edgptr → Edge;
        attributes   eid: int;
                     iid: int;
        action       none;

EtoEImap2_Schema2 = r-schema
        relates      Eiconptr → Edge_Icon2;
                     Edgptr → Edge;
        attributes   eid:      int;
                     iid:      int;
        action       none;
```

The following S-entities organize each R-entity above into corresponding sets.

```
SetofVImap1 = s-schema
        contains     VtoViconmap_Schema1;
        action       none;

SetofVImap2 = s-schema
        contains     VtoViconmap_Schema2;
        action       none;

SetofEImap1 = s-schema
        contains     EtoEiconmap_Schema1;
        action       none;

SetofEImap2 = s-schema
        contains     EtoEiconmap_Schema2;
        action       none;

SetofPImap = s-schema
        contains     PtoPiconmap_Schema;
        action       none;

SetofCImap = s-schema
        contains     CtoCiconmap_Schema;
        action       none;
```

A topology can be represented by a set of icons built from a set of basic icons.

The schema that defines the set of icons is associated with an action routine which displays each of the icons in its set.

```
SetofVtx_icons = s-schema
          contains    Vtx_icon1;
          action      none;

SetofEdge_icons = s-schema
          contains    Edge_icon1;
          action      none;

Topology_Icon = s-schema
          contains    SetofVtx_icons;
                      SetofEdge_icons;
          action      DisplayTop;
```

A set of icons for displaying the multicomputer structure may be constructed from the set of icons for processors and communication links as follows:

```
SetofProc_icons = s-schema
          contains    Vtx_icon1;
          action      none;

SetofCommlink_icons = s-schema
          contains    Edge_icon1;
          action      none;

Multicomputer_Icon = s-schema
          contains    SetofProc_icons;
                      SetofCommlink_icons;
          action      DisplayMulticomputer;
```

7.4.6 Estimating Database Size

Since the database uses in-core storage, there is a limit to the size of data that can be stored. To estimate the memory required, the representation of data in the database must be considered. The database consists of (i) a *database table* which stores the information on the actual data values such as pointers to the allocated data and (ii) a *symbol table* which maintains information on each schema, such as string name of the schema, the name of attributes and pointers the schema has, the type of schema, etc. Main storage requirements for the database results from these two data structures

The size of the database table can be estimated as follows. The database table consists of entries for the schemas where each schema has a predefined number of entities and each entity has a predefined number of attributes and pointers. The storage requirement for the database table is the size of the table data structure plus the size of the real data which is stored as character strings. Let m_{schema}, $m_{entities}$ and $m_{pointers}$ be the maximum number of schema types that can be defined, the maximum number of entities per schema, and the maximum number of pointers an entity can have respectively. Also, let $S_{pointer}$ be the size of each entry of a pointer in an entity. The size S_0 of the table data structure is then

$$S_0 = (m_{pointers} \times S_{pointer} \times m_{entities} + b_e) \times m_{schema} + b_s. \tag{7.1}$$

In the above equation, b_s and b_e are fixed bookkeeping overheads for each schema entry and each entity entry respectively.

The memory-size requirement for stored data can be estimated for a given number of vertices, edges, processors, communication links, and graphical icons.

Let V be the number of vertices, E be the number of edges, P be the number of processors, and L be the number of physical communication links. Let the sizes of the entities representing a vertex, an edge, a processor, and a link be S_v, S_e, S_p and S_l respectively and let their corresponding icons be represented by entities of sizes S_{vi}, S_{ei}, S_{ci}, and S_{li}. Then the size of the data for storing the entities representing the vertices, edges, processors, and links is given by

$$S_1 = V \times (S_v + S_{vi}) + E \times (S_e + S_{ei}) + P \times (S_p + S_{pi}) + L \times (S_l + S_{li}). \qquad (7.2)$$

Another set of data stored in the database are mappings. Let S_{vp} be the size of an entity representing a mapping of a vertex to a processor. All the mappings of entities to graphical icons use the same amount of storage and this amount is denoted by S_{imap}. The total storage size S_2 required by the mappings can be computed as

$$S_2 = V \times S_{vp} + (V + E + P + L) \times S_{imap}. \qquad (7.3)$$

Thus the total storage requirements of the database table is given by

$$S_{databasetable} = S_0 + S_1 + S_2. \qquad (7.4)$$

A symbol-table entry consists of the type of schema, the address of the action routine, a set of attribute names, a set of pointer names, and a set of extra attribute entries (the entries for extending the attributes at entity creation time). The total number of entries of the symbol table is m_{schema}. If S_{att} is the size of the attribute name, S_{ptr} the size of the pointer name, S_{extra} the size of the extra attribute entry, m_{att} the maximum number of attributes a schema can have and b_{st} the fixed storage overhead the table has, then the size of the symbol table is

$$S_{symboltable} = m_{schema} \times (S_{att} \times m_{att} + S_{ptr} \times m_{pointers} + S_{extra} \times m_{entities} + b_{st}). \quad (7.5)$$

The total size required for the database is then

$$S_{Database} = S_{databasetable} + S_{symboltable}. \quad (7.6)$$

7.5 Implementation and Evaluation of PRISM

7.5.1 Implementation of the PRISM Database

Central to the PRISM enivronment is the active database implemented on the Sun 3/50 workstation, which serves as a remote host of the hypercube. It is implemented as an in-core database residing in a single UNIX process. All the entity, relationship, and set relations representing vertices, edges, and graphical icons are stored in the database. Storage is allocated statically for the tables and dynamically for the actual values. In the current implementation, the action routines are linked to the database process directly, and the database operations are provided as C-language function calls. The DML program is written as a C program making database operation calls and is compiled together with the database process. This, however, is not a limitation since the database process and the DML program can be easily modified to run separately with processes communicating with each other via IPC messages.

Table 7.1: Estimated Database Storage Requirements

Topology Type	cube dim.	numof vertices	numof edges	Storage size (bytes)
Tree	3	8	7	64,570
Tree	4	16	15	156,874
Tree	7	128	127	2,148,842
Ring	3	8	8	64,700
Ring	4	16	16	157,004
Ring	7	128	128	2,149,028

To represent the topologies, the physical machine, and the display icons as entities in the database, the amount of storage used by the database is significant. To represent a topology on a hypercube with one set of icons for each entity that represents a vertex, an edge, a processor, or a physical link, the storage requirements are shown in Table 7.1. The database is configured to meet the requirements of a given topology in determining the storage size. There are a total of 34 schemas, each with attributes as defined in Section 7.4. The size of each entity is determined from these schema definitions. In representing the data an integer value may take up to 10 characters since the value is stored as a character string. A pointer takes 4 bytes. The *Commlink_icon* has the maximum number of attributes ($m_{att} = 8$), and the *Commlink* schema has the maximum number of entities ($m_{entities} = n2^{n-1}$, where n is the dimension of the cube). Fixed overhead for the tables implemented on the Sun workstation are found to be as follows: $b_e = 16$ bytes; $b_s = 4$ bytes; and $b_{st} = 36$ bytes.

The maximum size of a topology that can be represented in the database depends on the available run-time memory, the number of vertices and edges in the topology, and the number of processors and physical links in the machine represented as entities.

7.5.2 Implementation of the DML Programs

Manipulation of Topologies

The DML program can perform database operations on the entities that represent the topologies and retrieve sets of vertices or edges that satisfy certain conditions. For example, a set of vertices mapped onto a particular hypercube node can be determined using the following segment of DML program. The DML program determines all the vertices residing on a given processor node by scanning the set relation which defines the mapping of the vertices to the processors. The resulting vertices are obtained in the set *vertex_on_proc_set.* In this program, the variables used are defined as below:

SetofVtxS	:	schema defining the set of vertices
temp_set	:	a variable that represents a temporary set
ent_ptr	:	pointer to an entity
VPmap_set	:	set of relationship relations that map a processor to a vertex.
int_frm_ent	:	a utility routine that retrieves the value of a given

field as a character string from an entity

and returns and integer value.

DML Program Segment::

```
        vertex_on_proc_set = _tdb_ent_create( "SetofVtxS","");
        temp_set = _tdb_ent_ALL( VPmap_set );
        while( !(_tdb_set_isempty( temp_set))) {
                ent_ptr = _tdb_set_removeelt( &temp_set);
                processor_num = int_frm_ent(ent_ptr,"proc",0);
                if( proc_num == PROCESSOR )
                        _tdb_set_assign(vertex_on_proc_set,
                                _tdb_set_union( _tdb_ent_ALL(vertex_on_proc_set),
                                        _tdb_set_makeset(ent_ptr)));
        }
```

As another example of manipulating the topologies, the following segment of DML program determines a set of leaf vertices at level two of a tree topology. It first determines a set of leaf vertices and then a set of vertices at the second level of the tree. By intersecting these two sets, the set of vertices at level 2 of the tree can be determined. The leaf vertices are determined by first obtaining a set of vertices that are nonleaves and then performing a set difference from the set of vertices. The vertices at level 2 are determined by a breadth-first search starting from the root node.

The variables in the program segment are defined as follows:

nonleaves_set	:	A set of vertices that are not leaves
edg_set	:	A set of Edges

vtx_set	: A set of vertex entities
ent_ptr, ent_ptr1, ent_ptr2	: Pointer to an Entity
root_set	: A set containing the root vertex
parent_set	: A set containing the parent vertices
level_set	: A set the vertices at a level of a tree
leaves_set	: A set the leaf vertices
level2leaves_set	: A set containing the level 2 leaf vertices
src_vid	: Source vertex name
dst_vid	: Destination vertex name
vid, vid1	: Vertex name
temp_set1, temp_set2, temp_set3, temp_set4	: Temporary sets
_upstrtoint	: A utility routine which converts a string to integer
find_attinset	: A utility routine to find an entity in a given set with a given attribute value

DML Program Segment:

```
/* To determine the set of leaves vertices,
first determine the set of vertices that are not leaves */
nonleaves_set = _tdb_ent_create("SetofVtxS", "");
_tdb_set_assign( nonleaves_set, _tdb_set_init() );
/* For each edge, determine the outgoing vertex entities */
temp_set = _tdb_ent_ALL( edg_set );
while(!(_tdb_set_isempty(temp_set))) {
        ent_ptr = _tdb_set_removeelt( &temp_set );
        src_vid = _tdb_ent_getattr( ent_ptr, "srcvid", 0);
        temp_set2 = _tdb_ent_ALL( vtx_set );
        while( !(_tdb_set_isempty(temp_set2) ) ) {
                ent_ptr1 = _tdb_set_removeelt( &temp_set2 );
                vid1 = _tdb_ent_getattr( ent_ptr1, "vid", 0);
                if( vid1 == src_vid ) {
                        _tdb_set_assign(nonleaves_set,
                                _tdb_set_union(_tdb_ent_ALL(temp_set1)),
                                _tdb_set_makeset(ent_ptr1));
                }
        }
}
leaves_set = _tdb_ent_create("SetofVtxS", "");
_tdb_set_assign( leaves_set, _tdb_set_init() );
_tdb_set_assign( leaves_set,
        _tdb_set_diff( _tdb_ent_ALL(vtx_set), _tdb_ent_ALL(nonleaves_set) ));
/* Determine the set of vertices at level 2 */
parent_set = _tdb_ent_create("SetofVtxS", "");
_tdb_set_assign( temp_set,_tdb_ent_ALL( root_set ));
level_set = _tdb_ent_create("SetofVtxS", "");
level = 0;
temp_set1 = _tdb_ent_ALL(parent_set);
while( !(_tdb_set_isempty(temp_set1) )) {
level++;
_tdb_set_assign( level_set, _tdb_set_init());
while( !(_tdb_set_isempty(temp_set1) )) {
        ent_ptr = _tdb_set_removeelt( &temp_set1 );
        vid = _tdb_ent_getattr( ent_ptr,"vid",0);
        temp_set3 = _tdb_ent_ALL( edg_set );
        while( !( _tdb_set_isempty( temp_set3))) {
                ent_ptr1 = _tdb_set_removeelt( &temp_set3);
                src_vid = _tdb_ent_getattr( ent_ptr1,"srcvid",0);
                if( src_vid == vid ) {
                        dst_vid = int_frm_ent( ent_ptr1,"dstvid",0);
                        ent_ptr2 = find_attinset( vtx_set, "vid", dst_vid);
                        _tdb_set_assign(level_set,
                                _tdb_set_union(_tdb_ent_ALL(level_set)),
                                _tdb_set_makeset(ent_ptr2));
                }
        }
}
_tdb_set_assign( parent_set, _tdb_ent_ALL(level_set));
temp_set1 = _tdb_ent_ALL(parent_set);
```

```
if( level == 2 ) break;
}
level2leaves_set = _tdb_ent_create("SetofVtxS", "");
_tdb_set_assign( level2leaves_set,
        _tdb_set_inter( _tdb_ent_ALL(level_set), _tdb_ent_ALL(leaves_set) ));
```

The DML program is simple, flexible, and straightforward to implement, and it can easily compute subsets of vertices and edges satisfying certain properties.

Visualization of Topologies

The DML program, together with the PRISM database, can generate and manipulate displays of topologies in several ways. Most of these display manipulations can be done in a uniform manner—by changing certain attributes of the graphical icons of a relation or a set of relations. Also, alternate displays of the same relation are obtained by defining different schemas that represent them.

Representing different levels of detail uniformly. Topologies can be visualized in several ways. In the simplest form, each vertex can be represented by a circle and each edge by a line. As a whole, a topology can be displayed as a graph aesthetically laid out on the color screen. Or a vertex can be displayed in detail as a tabular icon indicating the details of its attributes. In accordance with the principle of least effort for effective displays [53], the programmer should be able to choose the type of display with the amount of detailed information he or she wishes to convey. The PRISM database provides a very convenient way to display

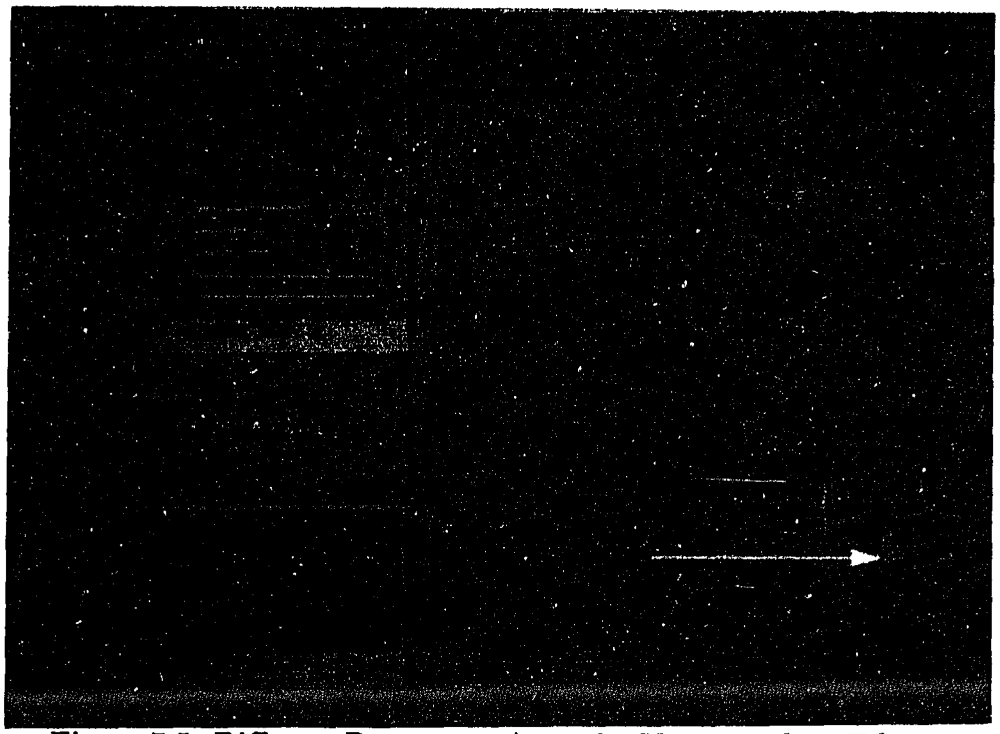


Figure 7.5: Different Representations of a Vertex and an Edge.

different levels of details of a relation by allowing different relations to be defined for each level.

Figure 7.5 indicates the different representations of a single vertex and a single edge. Simple action routines draw each icon, and they are associated with appropriate relations. They can be displayed by setting the display attribute to 1.

Displaying the structures. When displaying structures like trees, it is not trivial to generate a viewer-pleasing layout. A simple layout strategy is one that places the nodes of the structure in a regular pattern and then connects the nodes as defined in the structure.

A layout routine can be implemented as an action routine of the set relation

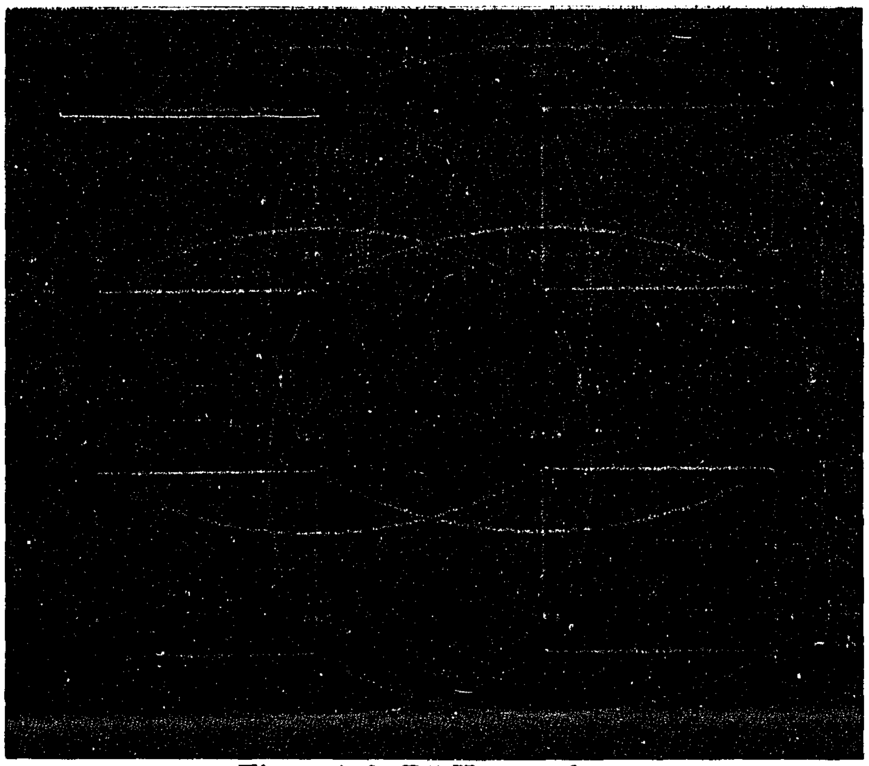


Figure 7.6: D5 Hypercube

which represents the graph when the display attribute is set, or it can be implemented separately as a routine initializing the screen coordinates of each vertex or node. Figure 7.6 displays the regular layout of the hypercube architecture, with the squares representing the hypercube processors and both curved and straight lines representing the physical communication links. The screen coordinates are determined when the database is initialized. Figure 7.7 displays a tree laid out aesthetically. The arrow represents the edges while the dots represents the vertices. Similarly, Figure 7.8 displays a ring topology of 16 vertices.

For these displays layout can be changed easily by simply changing the attributes xpos and ypos of the corresponding icons.

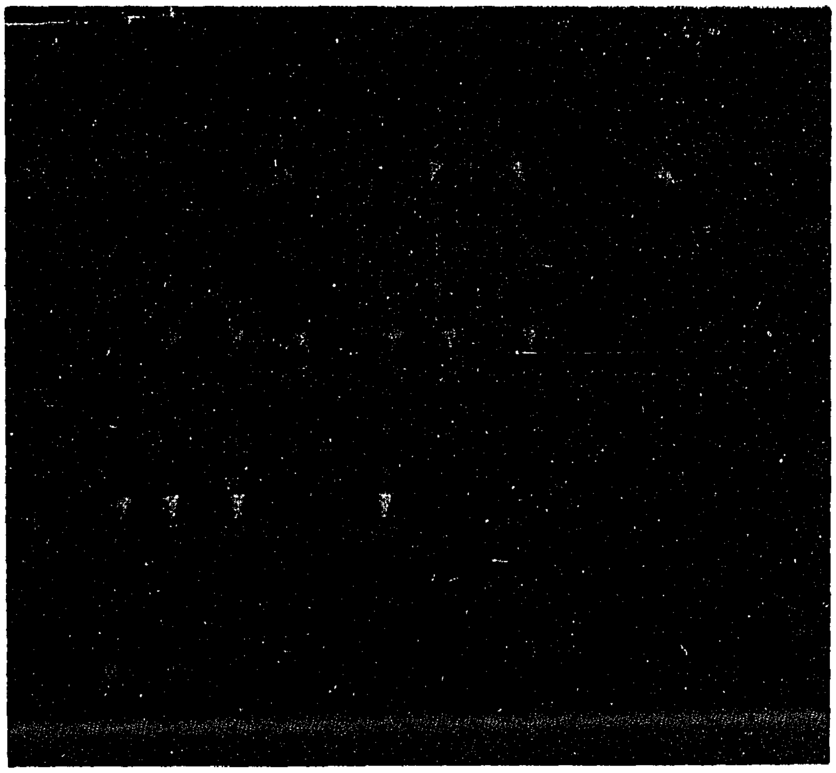


Figure 7.7: A Spanning Tree Topology

Figure 7.8: A Ring Topology

Use of color for displaying semantic information. As described in section 7.5.2, a subset of the vertices or edges can be determined by set manipulation operations provided by the database interface. Such a set can be displayed separately by setting the display attributes of the corresponding icons in the icon set. The subsets can also be displayed as part of the original set and by highlighting or changing the colors of the elements which represent them. Although such use of colors is not new, the ability to change the color attribute in the icons provides a uniform and simple way to illustrate the subset. This also allows colors to be used to represent semantic information such as *containment in a set.*

The displays generated by the following figures illustrate the use of color for displaying subsets of the vertex set. Figure 7.9 shows the subtree with the root at vertex id 1, and Figure 7.10 shows the set of vertices at the second level, which are leaves, by using a different color to indicate the vertices in a particular set.

Composition of icons. Graphical icons may be overlaid, attached, or composed in several ways to convey the semantics of the display. For example, overlaying may be used to illustrate the fact that a vertex is mapped to a hypercube node. This can be done by overlaying on top of each processor icon the corresponding vertex icon. Such composition of icons can be done easily in a DML program as follows. First, the set that represents the mapping of the two icons is determined using the database. Next, the xpos and ypos screen-coordinate attributes of the

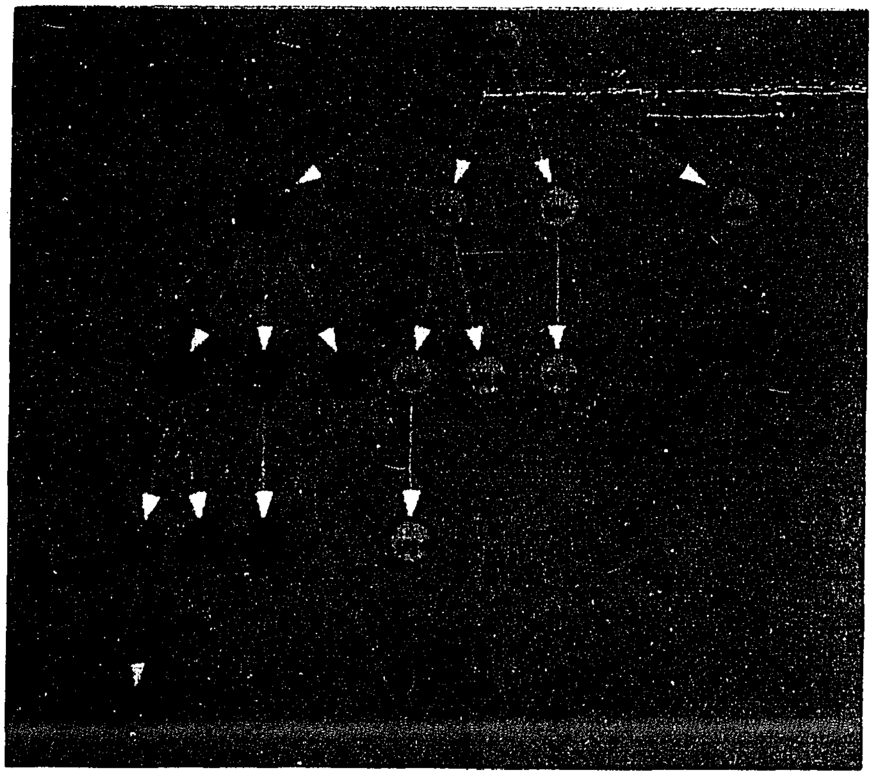


Figure 7.9: Subtree of Vertex Id 1

Figure 7.10: Leave Vertices at 2nd Level of the Tree

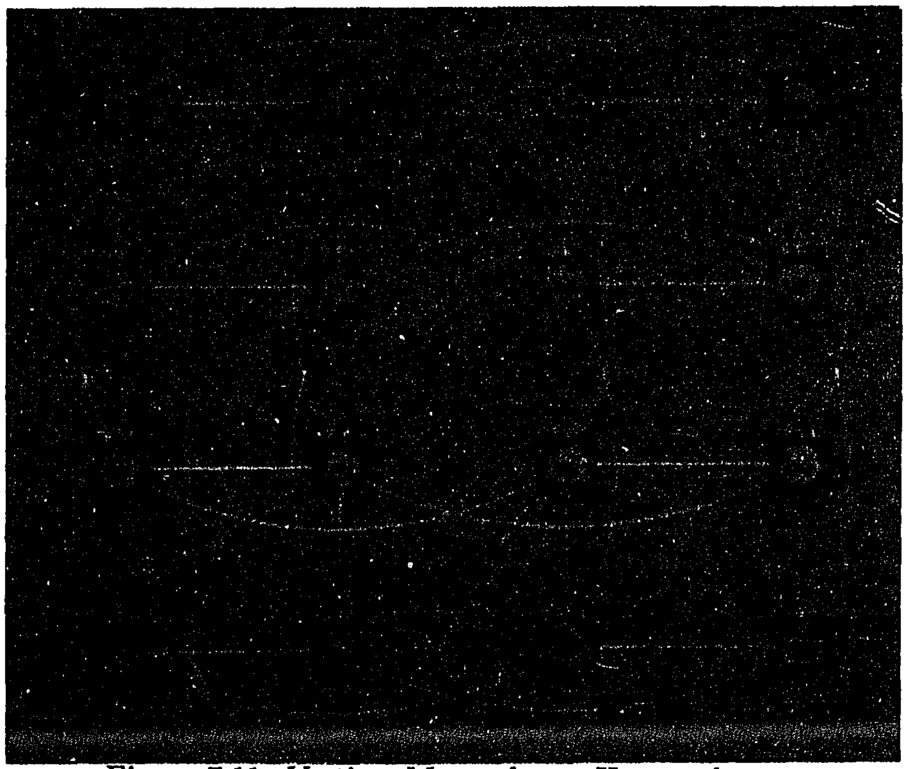


Figure 7.11: Vertices Mapped to a Hypercube

underlying icon are set to the same values as those of the overlaying icon. This causes one icon to be drawn on top of the other.

An example of such a display is shown in the following three figures. In Figure 7.11 the hypercube structure is displayed with the vertices mapped onto it. This can be done by assigning the xpos and ypos attributes of the processor icons to xpos and ypos attributes of the vertex icons. Figure 7.12 and Figure 7.13 display the logical structure of processors by assigning the xpos and ypos attributes of the vertex icons to the xpos and ypos attributes of the processor icons.

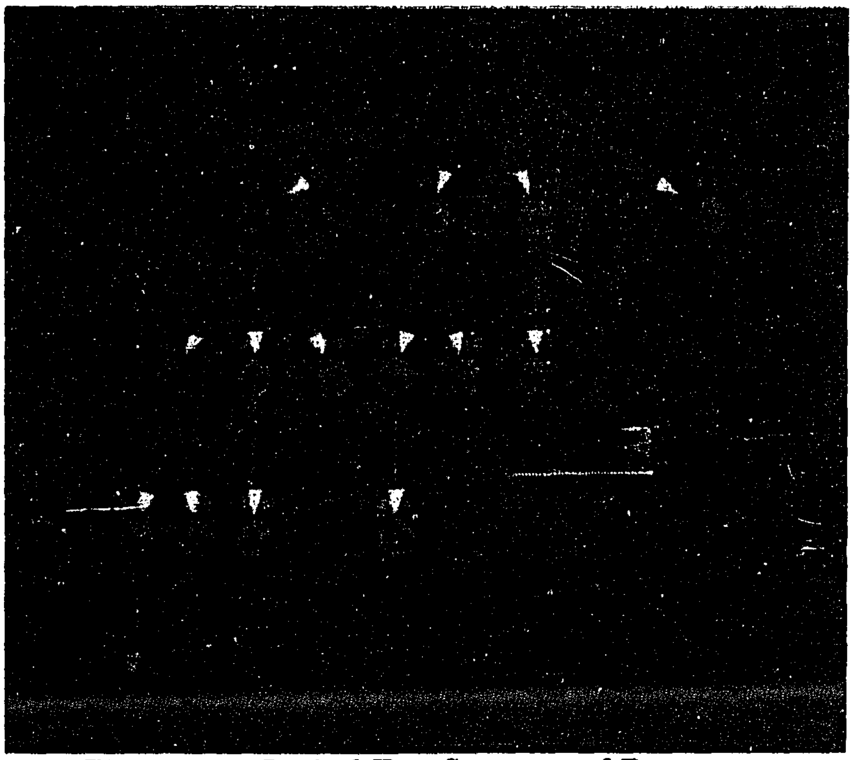


Figure 7.12: Logical Tree Structure of Processors

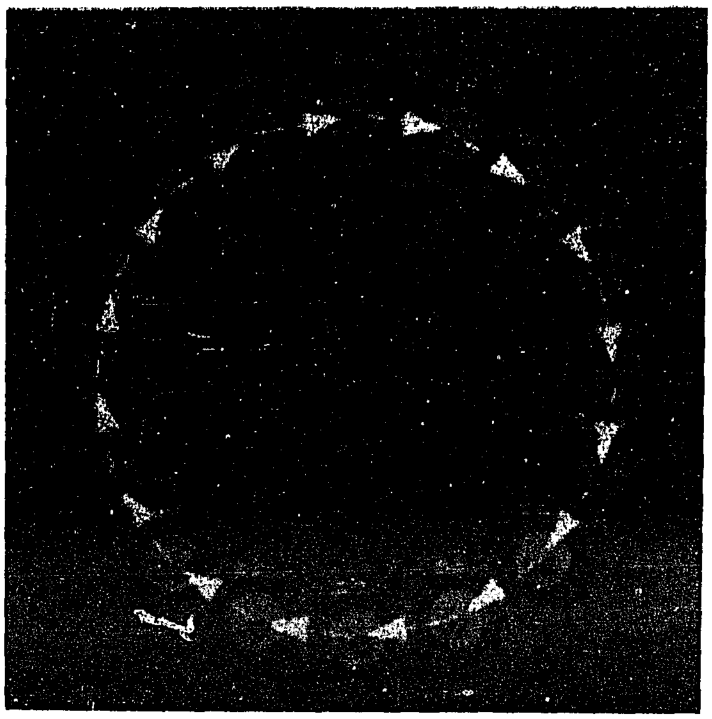


Figure 7.13: Logical Ring Structure of Processors

7.6 Conclusions

Experimentation with the partial implementation of PRISM in this chapter demonstrates not only the kinds of manipulations that can be done using the database, but also how they can be done. In using a DML as an integration language, abstractions can be conveniently represented as entities or sets of entities, and manipulating these sets is easy and convenient. Specifically, manipulation of topological entities such as determining the subsets of their structures and mappings, and manipulation of their graphical representations such as overlaying the icons and high-lighting the subsets, can be easily programmed and modified. This is because representations and operations can be done in a uniform manner.

CHAPTER VIII

Conclusions and Future Research

This chapter reviews the goals of the research, summarizes the results, and describes possible future research.

8.1 Review of the Goals

The thesis of this work is that topologies form a basis for a useful and desirable paradigm for programming multicomputers. Our motivation is that communication structures of a parallel program on parallel machines such as a hypercube must be programmable in a modular, efficient, and flexible manner. To demonstrate this thesis, the following goals were established.

1. Desirability of topologies. Chapter 3 argues that topologies are desirable because they allow efficient programming of global data and operations that arise from various decomposition methods. Chapter 4 describes two specific parallel applications that use topologies to implement their global data and operations.

2. Appropriateness of the topology construct for implementation. Chapter 5 and 6 describe the topology construct and its components, its implementation, and its performance evaluation demonstrating that the construct is appropriate because it can be built and used and can improve performance.

3. Ease of programming of topologies. Chapter 7 describes the tools and facilities for specifying, compiling, and loading the topologies, and it describes the use of a database interface for visualizing these topologies to aid debugging and monitoring.

The implication of meeting these goals is that topologies can be a viable paradigm for implementing parallel programs on multicomputers. Furthermore, they should also be a basis for structuring a set of parallel tasks in a message-passing system. They can be highly efficient when implemented properly and are appropriate for programming global data and operations in parallel programs.

8.2 Summary of Lessons Learned

In working towards the above goals we learned several lessons.

- Through our survey of decomposition methods, we observed that global data and operations arise in several of these methods from sharing of or performing operations on global information and that a single application may exhibit several global data and operations.

- Topologies are implementable and can improve the performance of the global operations, as exemplified by topologies performing a global sum or implementing global shared memory. Experiments also indicate that when the global operation is as fast as or faster than the time-out values of communication links, performance can be significantly affected by the retransmission of messages.

- Topologies are programmable and help improve the programmability of parallel computations by providing modularity, flexibility, and abstraction. Experience in using topologies indicates that the capability to load a communications structure separately from an application program is a very convenient facility. In addition, service routines are found to be a powerful mechanism. For instance, passing addresses (capabilities) to service routines and allowing them to access user memory is quite useful.

- Experience with building and using topologies indicates that they can be difficult to debug. Specific tools to build the topologies and monitor their data structures have been found to be useful in debugging them. Use of even more sophisticated tools, such as visual monitoring of topologies, would be advantageous.

- In experimenting with the TSP application, we observed that optimizing a single global data and operation in an application with multiple global data and operations may make the performance of the other global data and operations more critical.

- Reusability of topologies is helpful, especially for experimentation with different structures with minimal change in the application or when running different applications that use the same types of communication structures.

- The ISSOS Project has shown that a programming environment for the uniform integration of tools is useful for parallel programming on a network of processors. As an extended implementation of ISSOS, PRISM demonstrates such an environment is also useful in programming topologies and displaying certain views. The partial database constructed allows easy and uniform manipulation of structures and of graphical views. In particular, the action routines implementing graphical icons can be a convenient mechanism to uniformly represent topological views. Program visualization thus obtained has helped in debugging topologies because it allows different levels of detail to be displayed.

We also learned that the topology construct is not without drawbacks.

- Some of the objects' data structures implemented partly in the kernel and partly in the user process will make process migration difficult to implement. Also, scheduling of the operation execution is limited to meeting input and output conditions in a vertex. There should be a more general scheduling mechanism either to schedule locally or globally.

- Another drawback is that topologies should be able to change dynamically either with process migration or by addition and deletion of communication links. The current naming scheme must be extended to support such functionality.

- The major drawback in the implementation is the difficulty of debugging topologies. Messages themselves are computational and their routing may be application dependent. Debugging can be even more difficult if services are written such that the interaction between user applications and topologies is not defined clearly. It might be useful to extend the TopSend and TopRecv calls to impose some limitation on the implementation of such interactions.

- The major drawbacks of the current programming system are its slow interaction, and the limitation of the size of the database for representing topologies with large numbers of vertices and edges.

8.3 Future Research

From our implementation of topologies, we learned that debugging may be difficult, and highly complex topologies spanning hundreds of processors may be tedious to specify. An interesting area to investigate is the automatic generation of topologies. Because both the structure and the services of a topology must be designed jointly to match the application's characteristics, both must be generated automatically. Although an optimum structure to perform a certain global function may be hard to determine, libraries of structures for typical decomposition methods may be possible. Automatic generation of services can be done for well-defined domains such as user-defined protocols or channel semantics. For instance, a user-defined high-level protocol may be compiled into service routines implementing the protocol. Likewise, service routines that enforce certain semantics of the communication channels (such as multiplexing channel or a multicast channel) may be automatically generated for a given set of channel types.

One aspect of topologies not taken into account in our current implementation is the failure of communication links or of the processor nodes in the multicomputer system. Various reliability semantics may be explored for the distributed remote operations implemented by the topologies in the manner similar to the reliability semantics of RPC calls.

For highly efficient implementation of topologies, service routines must be highly reactive and must be executed with minimum processing overhead. Architectural support for topologies must provide not only hardware level routing but also execution of services concurrently with the user application. It may be worthwhile to implement on each node a communication co-processor capable of performing message processing, service execution, and of interacting with the node processor.

Several programming environment aspects of topologies can be explored further. First, additional tools can be integrated into PRISM to implement a more powerful and comprehensive environment. Examples of such tools are an interactive graphical interface to define the entities and relationships and to build graphical icons, a tool to automatically generate icons and to lay them out, a more complete and efficient monitoring system for monitoring performance or program states, and tools for automatic resource allocation. Second, the currently implemented program visualization system depicts only one type of view of the parallel program–namely, the topological view. Further work should be done to display other types of view, such as performance statistics, execution traces, load distribution, and the higher-level abstractions implemented by topologies, such as a representative display of an object rather than its internal communication structure.

To summarize, the work of this thesis presents several areas that call for further exploration for multicomputers, such as the automatic generation of topologies, architectural and hardware support for their efficient implementation, reliability semantics, and an environment for monitoring and visualization.

Appendix A

System Calls for Services

Following are the system calls or kernel-level calls (calls made within the kernel) implemented as extension to iPSC 3.1 that are used in conjunction with the topology primitive.

1. Basic communication calls as seen by the user:
    - TopReset()
    - TopLoad()
    - TopSend[w]()
    - TopRecv[w]()
    - TopOpen()
    - TopClose()
2. Other Utility calls (used by the tload/vinfo commands)
    - tquery()

3. Large-grain service loading and utilities

    - set_lservices()
    - waitq()
    - mapped_fd()
    - mapped_buf()
    - mapped_vtx()
    - output()
    - load_msgbuf()
    - fd_on_link()
    - unmapped_ptr()

4. Medium-grain service loading and utilities (in addition to the above calls)

    - set_mservices()
    - mapped_fd()
    - mapped_buf()
    - mapped_vtx()
    - output()
    - load_msgbuf()
    - fd_on_link()

- unmapped_ptr()

5. Fine-grain services utilities (these are not system calls but are just macros or simple routines inside the kernel.)

    - getbufferaddress()
    - output()
    - load_msgbuf()

A.1 Detailed Descriptions

1. **TopReset()**

    This system call will clean up the vertex data structures by setting the block memory storing them to zeroes. The whole structure is cleaned up and all the topologies previously loaded on the processor that executes this call will be lost. This must be considered as a privileged call and should be used with care since this will clean up *all* the vertices in spite of the caller not being bound to any of them. This is helpful when the previous programs have used the topology and did not close the topology properly or if the user wishes to have a clean start of loading a topology from scratch. Note that the topologies are still left around even if the processes which created them have died.

2. status = **TopLoad**(&vtx_structure)

struct tvertex vtx_structure

Returns : if status is -1, a topology with same id has already been loaded; otherwise okay status

This parameter is the initialized data structure of type *struct tvertex* which basically contains a list of input/output links, queues, and conditions. The user must know the structure of the tvertex to be able to make use of this call or must use the stub routine generated by the topology compiler (tcc).

Loading of the vertex on a processor causes the systems' internal data structure to be initialized to the vertex structure that is loaded. Specifically, it performs the following steps:

(i) It reads the topology id and vertex name of the new vertex, (ii) checks if any identical topology id and vertex name have been already loaded, (iii) if not, loads the vertex into an empty slot of vertex structures set the pids to -1, indicating no process is bound and (iv) sets the output and input channel descriptors to -1 (i.e.,none have been allocated).

3. tcount = **TopSend[w]** (top_handle, service_id, &buffer, count, tag)

int top_handle : The handle of the bound topology.

int service_id : An integer value which is designated as a specified service or operation

char buffer[] : Address of the buffer

int count : size of the buffer

int tag : value of tag

Returns -1 if the top_handle is illegal; -2 if the process is not bound to the vertex

This call is used to send a topology packet on a given topology structure to perform an operation specified by the *service_id.* The contents of the buffer of size *count* is sent with a tag value *tag.* The command will cause the execution of the operation in the vertex which the process is bounded. If the service forwards the packet, it will also cause remote execution of the service on the target node.

4. tcount = **TopRecv[w]** (top_handle, service_id, &buffer, count, &tag)

   int top_handle : The handle of the bounded topology.

   int service_id : An integer value which is designated as a specified service or operation

   char buffer[] : Address of the buffer

   int count : size of the buffer

   int tag : address of tag – if null address, then tag is ignored; if value is negative, the received packet's tag is returned.

int tcount : value of the number of bytes received; if negative, error in received packet.

Returns -1 if the top_handle is illegal; -2 if the process is not bound to the vertex.

This call causes the user process to search for the corresponding message in the output queue of the vertex that is bound to the process. If there is no message received yet, the process is put on a wait queue and is woken up when the packet arrives. Data is copied from the system buffer into the user buffer if the packet is found.

5. thandle = **TopOpen** (tmid, vid, initial_tag)

   int tmid : The topology-mapping id

   int vid : vertex name

   int initial_tag : initial value of tag

   Returns int thandle : Handle to the vertex of the topology bounded

   This call binds the user process to the vertex specified by tmid and vid. The value of the tag in the vertex is set to *initial_tag* at the binding time. Also channels for sending messages are allocated for the vertex at this time.

6. status = **TopClose** (thandle)

   int thandle : Handle of the topology to be closed.

returns int status : if negative, illegal handle.

This call unbinds the process from the vertex allowing other user processes to gain access to the topology and be able to use the services.

7. status = **tquery**(function, buffer, count)

   int function : Integer code for the following functions

   (a) NUMTOP - Number of topologies loaded Buffer must be of size 2 bytes at least

   (b) GETMID - Read out the tmid's of the topologies loaded The buffer should be as long as 2*number of topologies

   (c) RTMS - Determine number of retransmission counts. Number of channels that will be read is 7 or count/2 channels which ever is less.

   (d) RDVTX - Read out the vertex data structure. Must provide the topology and vertex name in buffer[0] and buffer[1] respectively. The results are returned in buffer[0] up to the size of the struct tvertex

   int buffer[] : buffer for input and output parameters

   int count : size of the buffer in number of bytes

   returns int status : if negative, illegal function is given

   This call is not for user processes but is used by system utilities to down load, send an inquiry, and retrieve information about the vertices and topologies.

8. **set_lservices**(service_id,routine_name, local_stack)

   int service_id : integer value of the service to be loaded

   int routine_name: name of the service routine identified as service_id.

   int local_stack : The stack for the interrupt task. Should be defined as a linear array of integers of size 2K or whatever amount desired.

   returns none.

   This call sets up the service routine located inside the large-grain service process' address space. When this large-grain service is requested, the topology demultiplexer will wake up the large-grain service process.

9. **waitq** (topid, vertex_id, service_id)

   int topid : integer values of the topology mapping identifier

   int vertex_id : integer value of the vertex name

   int service_id : service identifier

   This call puts the large-grain process on the wait queue of the large-grain service routine.

10. **set_mservices**(service_id, trap_number, routine_name, local_stack)

    int service_id : integer value of the service identifier to be loaded

    int trap_number : trap number assigned to the service. Currently, it can be from 18 to 27.

int routine_name: The address of the routine_name which is basically a stub routine that calls the actual service routine.

int local_stack : The stack for the interrupt task. Should be defined as a linear array of integers of size 2K or whatever amount desired.

returns none.

This call sets up the trap-handler routine located inside the user address space, which calls the service routine. Allowable trap numbers are 18 to 27.

11. address = **mapped_fd**(fd_index)

    int fd_index : Index of the message header to be mapped.

    returns struct fd * address : address of the fd as seen from user address space.

    This call sets up the global descriptor that maps the user-defined pointer to the frame descriptor (message header) inside the system space. Any attempt to access past the header structure will result in access violation.

12. address = **mapped_buf**(fd_index, ithBuffer)

    int fd_index : Index of the message header attached to the buffer.

    int ithBuffer : The ith buffer if the buffers are linked as a list.

    returns struct fd * address : address of the fd as seen from user address space.

This call is similar to the above call except that it maps the system buffer attached to the message header in the system space into the user space. Access beyond the buffer size will result in access violation.

13. address = **mapped_vtx**(fd_index)

    int fd_index : Index of the message header of the message destined to the vertex.

    returns struct fd * address : address of the fd as seen from user address space.

    This call is similar to the above call except that it maps the vertex to which the messages have arrived into the user space. Access beyond the buffer size will result in access violation.

14. fd_index = **output**(fi, service_id, link_num, tag)

    int fi : Index of currently received message header.

    int service_id : Service Identification in the message to cause an operation to be performed.

    int link_num : Link number to be output.

    int tag : Tag value to be set in the message.

    This call basically sets up the destination of the topology packet by loading appropriate fields in the header.

15. **load_msgbuf**(fi, buffer, count)

    int fi : Index of the message header to attach a buffer.

    char buffer[] : User buffer.

    int count : size of the buffer.

    This call loads a local buffer into a system buffer attached to the message header.

16. fd_index = **fd_on_link**(link_num)

    int link_num : Link number of the vertex that the user wishes to retrieve.

    Returns int fd_index : Index of the message header of the message on link number "link_num". Note that link_num = 0 refers to the bounded process.

    This call returns the frame descriptor (message header) of a packet queued on an input queue of link 'link_num'.

17. **unmapped_ptr**(address)

    char *address: Address or pointer that is desired to be removed from the global descriptor table.

    This call allows garbage collection of the descriptors.

Appendix B

Grammar of Topology Description Language

In the grammar rules below, '*' is a special terminal symbol expressing 'any of the above'. For example '**VtxId:*;**' implies that the vertex ids are those that are defined in the structure definition. Similarly, '**Inputs:*;*+;**' implies that the input edges are as defined in the structure definition, and the input condition is that all the input edges are to be or-ed ('+'). Terminal symbols are bold-faced. A null string is denoted by ϵ.

<Top-Desc>	::=	<top-type-decl-list>
		<mapping-decl-list>
		<service-block>
		<top-decl-list>
<top-decl-list>	::=	<top-decl-list> <top-decl> | <top_decl>
<top-decl>	::=	<top-id> **Topology is** <top-type> **and** <mapping>
		| <top-id>**Topology is** { <routine-name> } ;
<top-type>	::=	<id>;
<mapping>	::=	<id>;
<mapping-decl-list>	::=	<mapping-decl-list> <mapping-decl>
		| <mapping_decl>
<mapping-decl>	::=	<id> **is map** { <mapping-list> }
		| <id> **is mapping_routine** { <routine-name> }
<mapping-list>	::=	(<processor-id>:<set-of-vertices>)
		| <mapping-list>,(<processor-id>:<set-of-vertices>)
<top-type-decl-list>	::=	<top-type-decl-list> <top-type-decl>

```
                              | <top-type-decl>
<top-type-decl>          ::= <id> TopologyType is
                              { <structure-decl> <vertex-desc-list> <edge-decl>}
<structure-decl>         ::= Structure is { <vertex-decl> <conn-decl> }
<vertex-decl>            ::= Vertices = <set-of-vertices> ;
<set-of-vertices>        ::= <range-of-vertices> | <vertex-list>
<range-of-vertices>      ::= <vertex-no>. .<vertex-no>
<conn-decl>              ::= Connections = <conn-list> ;
<conn-list>              ::= <conn-list> <connection> | <connection>
<connection>             ::= <host-vertex-id>:<in-vertex-list>:<out-vertex-list>;
<in-vertex-list>         ::= <vertex-list> | ϵ
<out-vertex-list>        ::= <vertex-list> | ϵ
<vertex-list>            ::= <vertex-list>,<vertex-no> | <vertex-no>
<host-vertex-id>         ::= <vertex-no>
<vertex-no>              ::= <numerals>
<edge-decl>              ::= Edges are { <edge-attributes-list> }
<edge-attribute-list>    ::= <edge-attribute-list> <edge-attributes>
                              | <edge-attributes>
<edge-attributes>        ::= <edge-list> = <transmission-protocol-attributes>;
<edge-list>              ::= <vertex-pair> | *
<vertex-pair>            ::= <vertex-pair> (<vertex-no>,<vertex-no>)
                              | (<vertex-no>,<vertex-no>)
<vertex-desc-list>       ::= <vertex-desc> | <vertex-desc-list> <vertex-desc>
<vertex-desc>            ::= Vertex is {
                              VtxId :<vid>;
                              Services: <service-list>;
                              Inputs: < input-edge-list>;<input-cond >;
                              Outputs:<output-edge-list>;<output-cond>;
                              }
<vid>                    ::= <host-vertex-id> | *
<service-list>           ::= <service-type> | <service-type>,<service-list>
<input-edge-list>        ::= <vertex-list> | ϵ | *
<output-edge-list>       ::= <vertex-list> | ϵ | *
<input-cond>             ::= <cond>
<output-cond>            ::= <cond>
<cond>                   ::= <op-vertices> <vertex-no> | *+ | *.
<op-vertices>            ::= <op-vertices> <vertex-no>.
                              | <op-vertices> <vertex-no>+ | ϵ
<transmission-
protocol-attributes>     ::= <attributes>
                              | <transmission-protocol-attributes>.<attributes>
<attributes>             ::= ACK | NOACK | FLOWCTL | NOFLOWCTL
```

```
<service-type>      ::= <id>
<service_block>     ::= Services are { <service-decl> }
<service-decl>      ::= <service-name> | <service-name> <service-decl>
<service-name>      ::= <routine-name>on<processor-id>as<service-type>
<routine-name>      ::= <id>();
<processor-id>      ::= <numerals>
<top-id>            ::= <id>;
<id>                ::= <id> <letter> | <letter>
<letter>            ::= a..z | A..Z
<numerals>          ::= <digit> | <digit><numerals>
<digit>             ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Bibliography

[1] Mike Acetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. Technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, May 1986.

[2] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The eden system: A technical review. *IEEE Trans. Softw. Eng.*, SE-11(1):43–58, Jan. 1985.

[3] Yeshayahu Artsy, Hung-Yang Chang, and Raphel Finkel. Interprocess communication in charlotte. *IEEE SOftware*, January 1987.

[4] C. Aykanat, F. Ozguner, S. Martin, and S.M. Doraivelu. Parallelization of a finite element application program on a hypercube multiprocessor. In *Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee.* Oakridge National Laboratories, Sept. 1986.

[5] Forest Baskett, John Howard, and John Montague. Task communication in demos. *Proceedings of Sixth ACM Symposium on Operating Systems Principles*, pages 23–31, Nov. 1977.

[6] Gerard M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Computer Science Department, Carnegie-Mellon University, April 1978.

[7] Brian Bershad, Edward Lazowska, Heny Levy, and David Wagner. An open environment for building parallel programming systems. In *Proceedings of the ACM SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems.* ACM SIGPLAN, Jul. 1988.

[8] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[9] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), Feb 1984.

[10] S.M. Doraivelu C. Aykanat, F. Ercal, P. Sadayappan, K. Schwan, and B. Weide. Parallel computers for finite element analysis. In *1986 ASME International Conference on Computers in Engineering, Chicago.* ASME, Aug. 1986.

[11] Nicholas Carriero and David Gelernter. The s/netś linda kernel. *IEEE Transactions on Computer Systems*, 4(2), 1986.

[12] D. R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed system design. *The 6th. International Conference on Distributed Computing Systems, Cambridge,MA.*, pages 190–197, May 1986.

[13] David R. Cheriton and Willy Zwaenepol. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[14] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. Thoth, a portable real-time operating system. *Comm. of the Assoc. Comput. Mach.*, 22(2):105–115, Feb. 1979.

[15] D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating System Principles, Orcas Island, Washington*, pages 171–180. ACM, Dec. 1985.

[16] E. C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operation Systems Principles*, pages 63–78. ACM, Dec. 1985.

[17] W. Crowther and others. The butterfly parallel processor. *IEEE Computer Architecture Technical Committee Newsletter*, pages 18–45, Dec. 1985.

[18] W. Dally and C. Seitz. The torus routing chip. *J. Distributed Computing*, 1(4), 1986.

[19] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The clouds distributed operating system: Functional description, implementation details and related work. *The 9th. International Conference on Distributed Computing Systems, San Jose, CA.*, June 1988. To appear.

[20] J.B. Dennis, J.B. Fosseen, and J.P. Linderman. Data flow schemas. Technical report, Project MAC, M.I.T., Cambridge, MA, July 27 1972.

[21] Eds. I. Durham, S.F. Fuller, and A.K. Jones. The cm* review report. Technical report, Comp. Science Dept., Carnegie-Mellon Univ., 1977.

[22] Guy T. Almes Edward D. Lazowska, Henry M. Levy, Michael J. Fisher, Robert J. Fowler, and Stephen C. Vestal. The architecture of the eden system. In *Proceedings of the 8th Symposium on Operating System Principles, Asilomar, Ca.*, pages 148–159. Assoc. Comput. Mach., Dec. 1981.

[23] R. Finkel and U. Manber. Dib - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–255, Apr 1987.

[24] E. Fiume and A. Fournier. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. *Computer Graphics*, 17(3):141–150, July 1983.

[25] Geoffrey C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems On Concurrent Processors.* Prentice-Hall, 1988.

[26] Mark Friedell, Jane Barnett, and David Kramlich. Context-sensitive, grpahic presentation of information. *Computer Graphics*, 16(3), Jul 1982.

[27] Edward F. Gehringer, Daniel P. Siewiored, and Zary Segall. *Parallel Processing: The Cm* Experience.* Digital Press, Digital Equipment Corporation, 1987.

[28] Ahmed Gheith, Prabha Gopinath, Karsten Schwan, and Peter Wiley. Chaos and chaos-art: Extensions to an object-based kernel. In *IEEE Computer Society Fifth Workshop on Real-Time Operating Systems, Washington, D.C.* IEEE, April 1988.

[29] Paul Hudak. Exploring parafunctional programming: Separating the what from the how. *IEEE Software*, 5(1), Jan 1987.

[30] I.Durham, R.C. Dugan, A.K. Jones., and S.N. Talukdar. Power system simulation on a multiprocessor. In *Text of Abstracts of the IEEE Power Engineering Society Summer Meeting, Vancouver, British Columbia, Canada*, July 15-20 1979.

[31] Intel. *iPSC Programmer's Reference Guide*, 1987.

[32] Jamieson, Gannon, and Douglass. *The Characteristics of Parallel Algorithms.* The MIT Press, 1987.

[33] A.K. Jones, R.J. Chansler, I. Durham, J. Mohan, K. Schwan, and S. Vegdahl. Staros, a multiprocessor operating system. In *Proceedings of the 7th Symposium on Operating System Principles, Asilomar, CA*, pages 117–127. Assoc. Comput. Mach., Dec.10-12 1979.

[34] Anita K. Jones and Karsten Schwan. Task forces: Distributed software for solving problems of substantial size. pages 315–329, Sept.14-16 1979.

[35] Anita K. Jones and Peter Schwarz. Experience using multiprocessor systems: A status report. *Surveys of the Assoc. Comput. Mach.*, 12(2):121–166, June 1980.

[36] Eds. A.K. Jones and Edward Gehringer. The cm* multiprocessor project: A research review. Technical report, Comp. Science Dept., Carnegie-Mellon Univ., CMU-CS-80-131, July 1980.

[37] Rajiv Ramnath Karsten Schwan, Soumitra Sarkar, and Sridhar Vasudevan. An integrated programming/operating system for parallel programming. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-TR-85-13, Sept. 1985.

[38] Shmuel Katz. A superimposition control construct for distributed systems. Technical report, MCC Technical Report Number STP-268-87,Software Technology Program, Aug 1987.

[39] Karl Kleinpaste. On structural and performance improvements to peripheral interface software. Master's thesis, The Ohio State University, Columbus, Ohio, 1987.

[40] D. A. Kranlich, P.B. Gretchen, R. T. Carling, and C.F. Herot. Program visualization: Graphics support for software development. pages 143–149. Proceedings ACM/IEEE 20th Design Automation Conference, 1983.

[41] D. Kuck and others. The effects of program restructuring, algorithm change, and architecture choice on program performance. In *Proceedings of the International Conference on Parallel Processing*, pages 129–138. IEEE, Assoc. Comput. Mach., Aug. 1984.

[42] Kung, Lo, Jean, and Hwang. Wavefront array processors - concept to implementation. *IEEE Computer*, 20(7):18–33, Jul 1987.

[43] H.T. Kung. Why systolic architectures? *IEEE Computer Magazine*, 15(1), Jan. 1982.

[44] T. LeBlanc, N. Gafter, and Ohkarni T. Smp:a message-based programming environment for the bbn butterfly. Technical report, Goergia Institute of Technology, School of Information and Computer Science, July 1986.

[45] Thomas J. LeBlanc and S.A. Friedberg. Hierarchical process composition in distributed operating systems. In *Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado*, pages 26–34. IEEE, ACM, May 1985.

[46] B. Liskov, D. Curtis, P.Johnson, and R. Scheifler. Implementation of argus. In *Proceedings of the Eleventh ACM Symposium on Operation Systems Principles*, pages 111–122. ACM SIGOPS, Nov. 1987.

[47] M. Livny and U. Manber. Distributed computation via active messages. *IEEE Transactions on Computers*, C-34(12):1185–1190, Dec 1985.

[48] Jim Matthews and Karsten Schwan. Graphical views of parallel programs. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-TR-85-11, Sept. 1985.

[49] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operation Systems Principles*, pages 39–51. ACM SIGOPS, Nov. 1987.

[50] Joseph Mohan. A study in parallel computation - the travelling salesman problem. Technical report, Computer Science Department, Carnegie-Mellon University, Aug. 1982.

[51] Joseph Mohan. Experience with two parallel programs solving the parallel salesman problem. In *Proceedings of the 12th International Conference on Parallel Processing*, pages 191–193. IEEE, Aug. 1983.

[52] Aloysius Ka-Lau Mok. *Fundamental Problems of Distributed Systems for the Hard Real- Time Environment.* PhD thesis, Laboratory for Computer Science, Massachussetts Institute of Technology, May 1983.

[53] Alan Morse. Some principles for the effective display of data. *ACM SIGGRAPH*, 1979.

[54] Brad A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[55] Dave Ogle, Karsten Schwan, and Richard Snodgrass. The real-time collection and analysis of dynamic information in a distributed system. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-TR-85-12, Sept. 1985.

[56] David Ogle. *The Real-Time Monitoring of Distributed and Parallel Systems.* PhD thesis, Department of Computer and Information Sciences, The Ohio State University, Aug. 1988.

[57] D.E. Orin and W.W. Schrader. Efficient computation of the jacobian for robot manipulators. *International Journal of Robotic Research*, 4(1), Spring 1985.

[58] Neil S. Ostlund, Robert A. Whiteside, and Peter G. Hibbard. Computational chemistry and computer science. *Journal of Physical Chemistry*, 86:219–297, 1982.

[59] John K. Ousterhout, Donald A. Scelza, and Pradeep Sindhu. Medusa: An experiment in distributed operating system structure. *Comm. of the Assoc. Comput. Mach.*, 23(2):92–104, Feb. 1980.

[60] F.I. Parke. Simulation and expected performance analysis of multiple processor z-buffer systems. In *SIGGRAPH'80*, pages 48–56. ACM, July 1980.

[61] Rajiv Ramanth. *An Integrated Environment for Tuning Parallel Programs.* PhD thesis, The Ohio State University, 1988.

[62] Steven Reiss. Visual languages and the garden system. In *Program Visualization.* Springer-Verlag, 1987.

[63] D. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the Assoc. Comput. Mach.*, 17(7), 1974.

[64] P. Sadayappan and F. Ercal. Nearest-neighbor mapping of finite element graphs onto processors meshes. *IEEE Transactions On Computers*, C-36(12), Dec 1987.

[65] J. Salmon and S. Callahan. Moose: A multitasking os for hypercubes. In *3rd Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA*, pages 391–391. ACM, JPL, Jan. 1988.

[66] M. Satayanarayanan and Ellen H .Siegel. Multirpc: A parallel remote procedure call mechanism. Technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, CMU-CS-86-139, Aug. 1986.

[67] R.E. Schantz, R.H. Thomas, and G. Bono. The architecture of the cronus operating system. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 250–259. IEEE, May 1986.

[68] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. Gem: Operating system primitives for robots and real-time control. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-TR-85-4, Feb. 1985.

[69] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[70] Karsten Schwan and Ben Blake. A fast scheduling mechanism for real-time systems. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-5/87-TR16, Sept. 1987.

[71] Karsten Schwan and Win Bo. Topologies - computational messaging for multicomputers. In *Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA*, pages 580–593. ACM, Jet Propulsion Laboratories, Jan. 1988. Also available as technical report, Department of Computer and Information Science, Ohio State University, OSU-CISRC-3/88-TR11, March 1988.

[72] Karsten Schwan, John Gawkowski, and Ben Blake. Process and workload migration for a parallel branch-and-bound algorithm on a hypercube multicomputer. In *Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA*. ACM, Jet Propulsion Laboratories, Jan. 1988.

[73] Karsten Schwan, Prabha Gopinath, and Win Bo. Chaos - kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[74] Karsten Schwan and Anita K. Jones. Flexible software development for multiple computer systems. *IEEE Transactions on Software Engineering*, SE-12(3):385–401, March 1986.

[75] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A language and system for parallel programming. *IEEE Transactions on Software Engineering*, April 1988.

[76] J.T. Schwarz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–543, Oct. 1980.

[77] Michael Scott, Thomas LeBlanc, and Brian Marsh. Design rationale for psyche, a general-purpose multiprocessor operating system. IEEE, 1988.

[78] Z. Segall and Eds. L. Snyder. Position papers: Workshop on performance efficient parallel programming, seven springs, pa. Technical report, Computer Science Department, Carnegie-Mellon University, National Science Foundation, CMU-CS-86-181, Aug. 1986.

[79] C. Seitz. Reactive kernel. In *The 3rd Conf. On Hypercube Concurrent Computers and Applications*, pages 1520–1528. ACM, Jan. 1988.

[80] Charles L. Seitz. The cosmic cube. *Comm. of the Assoc. Comput. Mach.*, 28(1):22–33, Jan. 1985.

[81] Richard Snodgrass. *Monitoring Distributed Systems.* PhD thesis, Computer Science Dept., Carnegie-Mellon Univ., Sept. 1982.

[82] Richard Snodgrass. Monitoring in a software development environment: A relational approach. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 124–131. Assoc. Comput. Mach., SIGSOFT/SIGPLAN, April 1984.

[83] L. Snyder. Parallel programming and the poker programming environment. *IEEE Computer Magazine*, 17(7):27–37, July 1984.

[84] K. So, F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. Psimul - a system for parallel simulation of the execution of parallel programs. *Research Report, IBM T.J. Watson Research Center, Yorktown Heights, NY*, 1986.

[85] A. Spector. Performing remote operations efficiently on a local computer network. *Comm. Assoc. Comput. Mach.*, 25(4):246–260, April 1982.

[86] B.W. Weide. Graphical programming of process control software using stile: A progress report and future directions. Technical report, Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, OH, Sep. 1983.

[87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, D. Black, W. Bolosky, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operation Systems Principles*, pages 63–76. ACM SIGOPS, Nov. 1987.

[88] Edward R. Zayas. Attacking the process migration bottleneck. pages 13–24. Proceedings of the eleventh ACM symposium on Operating System Principles, ACM SIGOPS, Nov 1987.