Specifying Communication for Massively Parallel Ensemble Machines

Duane A. Bailey

COINS Technical Report 88-83
September 1988

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

SPECIFYING COMMUNICATION FOR MASSIVELY PARALLEL ENSEMBLE MACHINES

A Dissertation Presented
by

Duane Andrew Bailey

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

September 1988

Department of Computer and Information Science

SPECIFYING COMMUNICATION
FOR
MASSIVELY PARALLEL ENSEMBLE MACHINES

A Dissertation Presented

by

Duane Andrew Bailey

Approved as to style and content by:

____

Janice E. Cuny, Chairperson of Committee

____

James F. Kurose, Member

____

Donald F. Towsley, Member

____

Sheldon B. Akers, Member

____

W. Richards Adrion, Department Head
Department of Computer and Information Science

© Copyright by Duane Andrew Bailey 1988
All Rights Reserved

This work was supported, in part, by the Office of Naval Research
(under contract N000014-84-K-0647), and by a ComputerVision Fellowship
granted through the American Electronics Association.

To Mary, Megan and Katelin.

ACKNOWLEDGEMENTS

No thesis is written in absolute isolation, and this is no exception. A great many people helped make this work possible; the thanks expressed here necessarily falls short.

Above all, Jan Cuny, made this dissertation a reality. Her advice and friendship can never be repaid. But, then, her pencil budget will decrease.

Several people read and reread various versions of this document. I thank Shelly Akers, Dave Carlson, and Don Towsley for their guidance. Special thanks go to Jim Kurose who was there from beginning to end, and to Larry Snyder whose discussions shaped much of this work.

The Simple Simon Group absorbed my comments and humor (perhaps a little too often) and a few people — Victor, Bruce, Gary, and Badri — helped make time fly.

I thank Mary, Megan, and now Katelin: they knew it could be done.

ABSTRACT
Specifying Communication For Massively Parallel Ensemble Machines

September 1988

Duane Andrew Bailey, B.A., Amherst College
M.S., Ph.D., University of Massachusetts

Directed by: Professor Janice E. Cuny

The work presented in this dissertation resolves problems particular to explicit specification of logical communication structures for massively parallel algorithms. Primary contributions include (1) the identification of characteristics of communication structures induced by standard parallel algorithm design paradigms, (2) the design of a communication abstraction that extends current message passing facilities, and (3) the development of a model for graphically specifying abstract aspects of communication structures.

Paradigms for designing sequential algorithms are important to our understanding of the sequential programming process. Their use suggests basic abstractions which aid in implementation. In this dissertation we identify characteristics of communication — including connectivity, symmetry and low diameter — that are fundamental to massively parallel algorithms. These characteristics are metrics for evaluating proposed methods for specification of communication.

Communication is often considered at a more abstract level than the message passing provided in many programming environments. The various techniques of constructing parallel algorithms require important abstractions of communication. These abstractions often consist of patterns of message transmission whose coordination is more global than point-to-point message passing. This thesis proposes

canister communication, which supports correct access to reusable message carriers traveling among processes on specified paths.

Communication is poorly specified in most environments because parallelism is provided as an extension of sequential data structuring techniques. A new specification mechanism is developed, based on *aggregate rewriting graph grammars*. This mechanism is unique in its uniform treatment of logically related nodes, and describes the structure of communication among processes more accurately than parallelized declarative mechanisms. Graph grammars are also investigated as theoretical support for an annotated graph editor.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v

ABSTRACT vi

LIST OF FIGURES xii

CHAPTER

1 Introduction 1

1.1 Initial Assumptions 2

1.2 The Problem 2

1.3 Thesis Contributions 4

1.4 What's Ahead 5

2 Related Research 7

2.1 Architectural Considerations 7

2.1.1 Dedicated Path Architectures 8

2.1.2 Reconfigurable, Nonshared Resource Architectures 11

2.1.3 Reconfigurable Shared Resource Machines 13

2.1.4 Summary 14

2.2 Programming Constraints 15

2.2.1 Languages Supporting Monolithic Codes 15

2.2.2 Languages Supporting Multilithic Codes 17

2.2.3 Summary 23

2.3 Conclusion 23

3 **Communication Structures** 25

3.1 Communication: Control *and* Structure 26

3.2 Programming Paradigms 28

3.2.1 Divide-and-Conquer 29

3.2.2 Systolic Methods 35

3.2.3 Relaxation 38

3.2.4 Routing Problems 41

3.2.5 Composite Communication Structures 44

3.3 Metrics of Communication Structures 46

3.3.1 Regularity Metrics 47

3.3.2 Irregularity 49

3.4 The Graph Abstraction 49

3.4.1 Labeling of Communication Structures 53

3.4.2 Supporting Flexible Annotations 55

3.5 Conclusion 58

4 **Canister Communication** 59

4.1 Coordinated Message Passing 59

4.2 Canister Communication 67

4.2.1 The Itinerary: Encoding Message Behavior 67

4.3 The Canister: A Message Wrapper 70

4.4 Examples of Canister Communication 73

4.5 The Use of Canister Communication in Debugging 78

4.5.1 Recognizing Patterns in Communication 78

4.5.2 Radioactive Tagging 78

4.6 Conclusion 79

5 Graph Grammars 83

5.1 Graph Rewriting Systems 83

5.2 Aggregate Rewriting Graph Grammars 85

5.2.1 The Formalism 86

5.2.2 Labeling Extensions 95

5.3 Structural Properties 99

5.3.1 Basic Properties 99

5.3.2 Connectedness 103

5.3.3 Symmetry Properties 105

5.4 Conclusion 111

6 Graph Editors 113

6.1 The Simple Simon Environment 113

6.2 The Graph Editor 114

6.3 The Graph Building Cycle 115

6.3.1 Domain Identification 116

6.3.2 Transformation Specification 118

6.3.3 Transformation Application 120

6.3.4 Graph Analysis 120

6.4 Conclusion 120

7 Conclusions and Evaluation 123

7.1 Overview 123

7.2 Contributions of This Research 123

7.3 Limitations and Future Work 124

LIST OF FIGURES

2–1 The Illiac IV array processor 9

2–2 A systolic array for matrix-vector multiplication. 9

2–3 A four dimensional binary cube and an embedded mesh. 10

2–4 A complete binary tree embedded in a CHiP lattice. 12

2–5 An 8x8 SW-banyan multistage switching network. 13

2–6 Communication induced by dependencies in distributed code. 16

2–7 A skewed array used in PDE solution in the Illiac processor. 16

2–8 Image histogram computation using structured processes. 18

2–9 Tree summation on the Transputer using Occam. 19

2–10 A Wavefront code for PDE solution 21

2–11 Specification of a binary tree structure in the Prep-P environment. . 22

2–12 A CSL cycle specification. 22

3–1 A sequential `bubble sort` program. 26

3–2 Communication as control: barriers in a multi-phase algorithm. . . . 27

3–3 A parallel Mergesort program. 30

3–4 Overlapping divide-and-conquer domains. 32

3–5 Divide-and-conquer layouts for multidimensional domains. 33

3–6 Communication induced by various recurrences. 38

3–7 A rectangular domain for PDE approximation. 39

3–8 Stencils used in solution of PDEs. 40

3–9 A three stage logical network, and its projection. 42

3–10 Communication supporting PDE approximation. 45

3–11 A spatially distributed macropipeline. 45

3–12 Hierarchical composition. 46

3–13 Wavefront `matrix multiplication` communication. 50

3–14 `Jacobi iteration` communication. 50

3–15 `Band matrix multiplication` and underlying graph. 52

3–16 Two constructions of banyans. 56

4–1 Paths of communication in `band matrix multiplication`. 60

4–2 Paths for `solution of lower triangular systems`. 61

4–3 Cyclic data flow in a `transitive closure` algorithm. 61

4–4 The tree-shaped paths of prefix-type operations. 62

4–5 The itineraries for Jacobi's PDE approximation technique. 62

4–6 Communication needed to generate and test palindromes. 63

4–7 A Wavefront Array Processor implementation of *LU* decomposition. 64

4–8 Mesh support for butterfly paths. 65

4–9 Three pairings which must be considered in the problem solved at processor [1, 4]. 65

4–10 Paths supporting the solution of a system of system of linear equations. 66

4–11 Specification of unbounded cycling (a), and fixed cycling (b). . . . 70

4–12 The logical structure of a canister 71

4–13 Canister access states 71

4–14 Radioactive tagging in `dynamic programming` 81

5–1 A cell mass and its representative graph. 84

5–2 Graph aggregates. 86

5–3 Aggregate manipulation. 87

5–4 A graph with $a - b$ occurrences. 88

5–5 The various effects of inheritance functions on production application. 89

5–6 Derivation of a two stage butterfly network. 93

5–7 A production and derivation for Theorem 5.2. 100

5–8 Examples of the three conditions for preserving connectedness. . . . 104

5–9 Necessity for π-symmetric functions. 106

5–10 Necessity for preservation of inheritance. 107

5–11 Daughter-induced symmetry. 109

5–12 The functions relating descendents of different mother node occurrences. 110

6–1 Components of the proposed graph editor. 121

Chapter 1
INTRODUCTION

Interprocess communication is the essential distinction between sequential and concurrent programming. As the designer of an algorithm increases parallelism, the communication needed to insure a coordinated effort in solving the problem also increases. For massively parallel computations — composed of thousands of processes — interprocess communication is pervasive. It is fundamental to our understanding of the algorithm, our accuracy in specifying it as a program, and our ability to efficiently execute it on the target architecture.

Massive parallelism has already been successfully exploited in a number of areas including linear algebra, finite element methods, partial differential equations, and image processing. Yet programming support for these endeavors has been largely inadequate. Because current environments are based on experience with structuring data in sequential systems, the parallel programmer's conceptualization of his software is not directly supported. The need for this support is made clear by Synder[153]:

> "Programming is a conversion activity and, as such, will be easy or difficult depending on whether the algorithmic form is similar or dissimilar to the desired program form .... The ideal programming environment, then, cannot make parallel programming effortless, since there will always be some dissimilarity due to the inherent properties of abstraction. It could greatly simplify the programming task, however, by supporting a specificational form close to that used in the technical literature to describe algorithms."(pages 27–28)

If a parallel programming environment is to support specifications similar to those found in the technical literature, the description of interprocess communication is essential. *This dissertation focuses on interprocess communication in parallel programming environments, providing new abstractions of communication and new parallel-specific tools for their specification.*

The remainder of this chapter discusses the initial assumptions, the problem, and contributions of this work.

1.1 Initial Assumptions

This work focuses on the process of implementing algorithms for ***massively parallel ensemble architectures*** — processors composed of thousands of independently programmed processor elements, each with its own dedicated "local" memory, coordinating their activity entirely by messages passed through a sparsely connected network. Typical examples include various binary cubes[71,145], array processors[16,100], and tree machines[20,32,105,158]. We believe that ensemble architectures show the greatest promise of supporting massive parallelism because they are less susceptible to memory-access latency and thus more likely to be scalable[6]; however many of our results can be extended to a shared memory model.

We limit ourselves to explicitly specified parallelism because, while environments supporting automatic extraction of parallelism from a monolithic code (a single code body) exist[4,91,96,123], they are not yet as efficient. Even if these methods should become more successful, there will be many algorithms for which explicit specification of communication structures is still important because "efficient concurrent algorithms may be quite different from their sequential counterparts"([145], page 26).

One cost of explicit parallelism is management of an overwhelming amount of information associated with process code and interprocess communication. A flexible framework for controlling the specification of these large systems is useful. We shall often rely on the ***program-as-database*** concept[39,154]. Each program is fully specified by a database of program attributes which may be queried to provide a number of logical program views and abstractions.

1.2 The Problem

Effective programming stems from decomposing an algorithm into manageable logical components — through various abstraction mechanisms — and cor-

rectly specifying each component as it is envisioned by the designer. For parallel environments, understanding how to extract parallelism has often distracted efforts to provide abstractions necessary for effective programming. In particular, *current parallel programming environments do not adequately support the specification of interprocess communication.*

Many algorithms have a fundamental *logical communication structure*. This topology is defined by the processes and channels of communication which are necessary to support a correctly functioning algorithm. Matrix problems, for example, often require communication structures which are similar in structure to the matrices which are being considered. In contrast, the *physical communication structure* is composed of the processors and hardware channels actually used in supporting an algorithm and may differ from the logical structure due to the limitations of the physical connectivity of the machine.

The specification of these communication structures in current parallel environments is often opaque and, in most cases, is present only as a side effect of *process* specification[35,78,108]. Even when communication structures are formally specified, they often represent the physical structure[92,100,101,154,156], while the logical structure is obscured by the idiosyncrasies of the underlying architecture, such as channel dilation, and multiplexing. Ideally, evolution of parallel programming environments will remove these low-level details from the specification of logical communication.

The specification of the logical communication structure is only a first step toward specifying complex communication. When tightly coupled processes communicate, messages participate in global communication patterns that are significant to the programmer's understanding of the algorithm. Because the semantics of these communication patterns are often more abstract than those provided by plain message passing systems, current parallel programming paradigms do not directly support this aspect of interprocess communication either.

We now review the contributions of this dissertation.

1.3 Thesis Contributions

The work presented here is dedicated to resolving the problems particular to explicit specification of logical communication structures in massively parallel systems. Primary contributions include (1) the identification of characteristics of communication structures induced by standard parallel algorithm design paradigms, (2) the design of a communication abstraction that extends the current message passing facilities, and (3) the development of a model for graphically specifying abstract aspects of communication structures. We detail these contributions:

Parallel algorithm design paradigms. Paradigms for designing sequential algorithms (*e.g.* divide-and-conquer) are important to our understanding of the sequential programming process. Their use suggests basic abstractions which aid in implementation (*e.g.* recursion and procedural abstraction). In this dissertation, we identify characteristics of communication, such as connectivity, symmetry and low diameter, that are fundamental to parallel programming. These characteristics are seen as metrics for evaluating proposed methods for specification of communication.

A communication abstraction. Communication should be considered at a more abstract level than the message passing provided in many programming environments. The various techniques of constructing parallel algorithms require important abstractions of communication. These abstractions often consist of patterns of message exchange whose coordination is more global than point-to-point message passing. In this dissertation, we describe *canister communication*, which supports correct access to reusable message carriers traveling among processes on specified paths.

A model of communication specification. Communication is poorly specified in most environments because parallelism is provided by an extension of sequential data structuring techniques. We develop a new specification mechanism based on *graph grammars* which describes the structure of communication among processes more accurately than parallelized declarative mechanisms. This mech-

anism is unique in its uniform treatment of *aggregates* of logically related nodes. Furthermore, it supports the implementation of algorithm *families*, removing the tendency to respecify algorithms for minor variations in host architectures. Finally, we demonstrate the use of graph grammars as theoretical support for tools that specify communication.

1.4 What's Ahead

Chapter 2 discusses the current status of programming environments which support the programming of parallel ensemble architectures.

Chapter 3 focuses on the support for communication which is required by various parallel algorithm design methods. In addition, certain inherent characteristics of "regularity" are identified.

Chapter 4 investigates the concept of *persistent message passing,* introducing a new programming paradigm, called *canister communication*, which more adequately supports global communication patterns.

Chapter 5 describes a graph grammar formalism which is designed to support specification of communication in parallel systems. We provide a number of theorems which show that these *aggregate rewriting graph grammars* have characteristics which are desirable to parallel algorithm design.

Chapter 6 brings theory to implementation. We describe an editor for communication structures and canister itineraries which is useful in parallel programming environments.

Chapter 7 reviews the results presented in this dissertation.

Chapter 2
RELATED RESEARCH

We consider, in this chapter, both architectures and environments which are likely to support massive parallelism. This review is two fold. First, we consider the constraints that architectures have traditionally imposed on programming environments. Second, we consider the specification of communication in current parallel programming environments.

Only a small number of systems currently support massively parallel programming, however, many of the environments we review extend naturally, to support a thousand or more processors. While the research in this dissertation considers strictly local memory machines, we also review several shared memory machines which promise massively parallel implementations; in many cases shared memory machines can run in a restricted local memory mode with communication through shared memory.

2.1 Architectural Considerations

Unfortunately, there seems no general, architecturally independent method for specifying communication for parallel algorithms. At the time of this writing there are, for example, no easily portable parallel programming environments — some environments have been "ported" to several machines[156], but the portable view of the system is a restricted virtual machine. This is due to fact that the vast differences in computing properties of the various architectures have made generalized concepts in programming parallel machines difficult to identify.

A number of viable architectures may eventually yield a high degree of parallelism. Because most current environments are driven by particular host architectures, it is important to first understand distinguishing characteristics of these machines in order to better appreciate the barriers to general implementation techniques.

Two characteristics of hardware communication can be used to classify existing machines: variability in topology, and resource sharing. ***Dedicated path*** architectures, for example, consist of processors interconnected in a fixed structure. The topology of ***reconfigurable*** architectures can be changed to reduce the logical overhead of mapping the algorithm to the architecture. Architectures which potentially provide concurrent access to a single resource are ***shared resource*** architectures. We first discuss dedicated path architectures which are appealing due to their simplicity of design.

2.1.1 Dedicated Path Architectures

Dedicated path architectures restrict communication along fixed hardware channels. Due to the simplicity of constructing dedicated path architectures, these systems are most likely to provide large amounts of parallelism in the near future.

During the sixties, the SOLOMON[149] and ILLIAC IV[14] projects designed and constructed classic examples of SIMD[1] dedicated path machines. These array structured processors (Figure 2–1) achieved significant advances in performance on scientific computation, but often required a complete understanding of the machine by the programmer[102]. Because logical channels between arbitrary processors were not supported, communication within these machines was usually limited to rectangular patterns.

The financial and technological advantages of using VLSI components to construct simply programmed, special purpose processors were realized during the late seventies by H. T. Kung[97]. ***Systolic array processors*** are regularly structured, dedicated path processors that can be built cheaply from existing components in configurations that are more problem-oriented than previous array processors (see Figure 2–2). The success of this SIMD architecture is borne out by the wide attention it has received[33,67,77,98,107]. Nevertheless, dedicated paths make a specialized processor array available only to a class of similarly structured algorithms. This

[1]Single Instruction stream, Multiple Data stream. Likewise, MIMD stands for Multiple Instruction stream, Multiple Data stream.

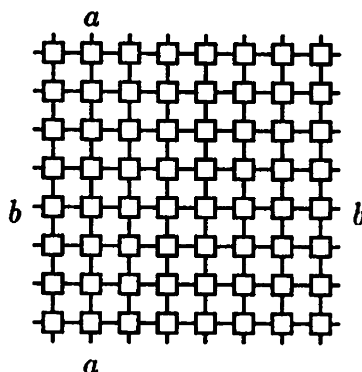


Figure 2–1: The Illiac IV array processor. Labels indicate the actual toroidal connectivity of the array.

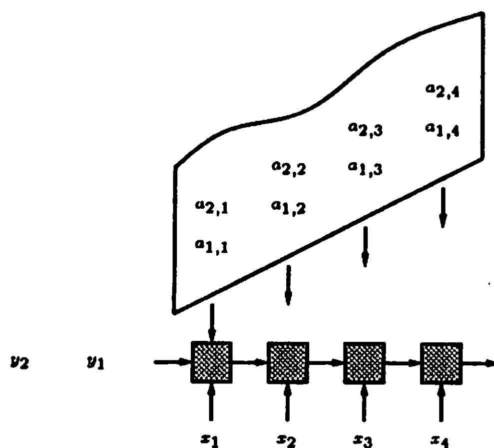


Figure 2–2: A systolic array for matrix-vector multiplication. The matrix and vector are pipelined from top and bottom, respectively, while results exit to the right.

prompted a number of groups to investigate design of generic reconfigurable processor elements for use in solving problems with a wider variety of topologies[5,98].

An offshoot of systolic arrays is S. Y. Kung's Wavefront Array Processor (WAP)[100,101] — a data driven, square mesh-connected, dedicated path processor which is useful in solving the regularly structured problems of linear algebra and signal processing. These problems can usually be transformed into algorithms which propagate waves of data through an array of SIMD computing elements. This algorithmic structure motivated the construction of a special purpose processor and associated programming environment for programming acyclic wavefront algorithms.

Dedicated path machines supporting a wider variety of communication patterns have been developed. The Cosmic Cube[145] and commercial variants from Ardent, Intel, and NCube are binary n-cubes (Figure 2–3a) consisting of 2^n processors each connected to n others in a method that relates processor addresses. Some applications can make direct use of the binary n-cube's connectivity with-

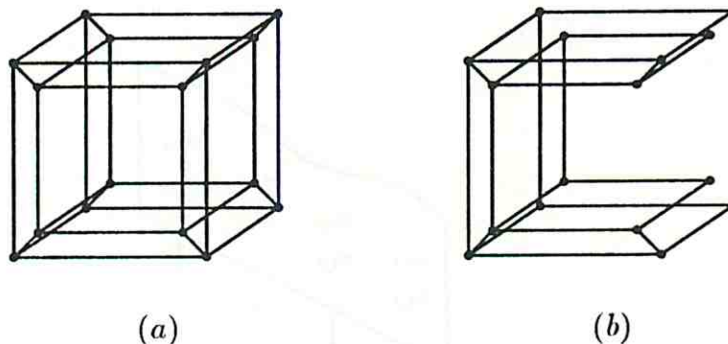


Figure 2–3: A four dimensional binary cube and an embedded mesh.

out multiple hop messages. However, environments for this class of architectures usually implement communication through transparent message passing which is supported by a software router at each node. Due to the ease of routing, many applications never consider the underlying topology of the cube during application design and, as a consequence, the expense of point-to-point communication in most cube implementations has made algorithm mapping an important program-

ming consideration. The Connection Machine™[71,72] (a 16-cube of bit processors) provides efficient support for two topologies, the cube and the mesh (Figure 2–3).

Processors with fixed connectivity allow implementation of classes of algorithms that have similar communication properties. They are, however, incapable of directly implementing general algorithms or dynamically changing *vari-structured* problems. Nonetheless, the simplicity of their design has allowed these machines to be the first to provide massive parallelism.

2.1.2 Reconfigurable, Nonshared Resource Architectures

The CHiP (Configurable, Highly Parallel)[90,154] computer and Kung's configurable systolic processor[98] allow reconfiguration of communication channels to suit varying problem structures. The unique feature of these machines is that no resources (memory, processors, or communication paths) are shared during execution. Problems suitable for these architectures are usually statically mapped by the user without complex analysis.

The CHiP processor provides a square grid of processors and custom built switches which may be configured to establish up to 4 independent paths. The switches can be reconfigured at run-time to one of a number of down-loaded settings. Highly irregular or complex topologies may be specified, provided the corridor width (the number of switches between adjacent processor elements) is sufficiently large. Due to its array-like regularity the CHiP processor requires contorted layouts of less rectangular structures (Figure 2–4).

Reconfigurable nonshared resource processors are not advantageous when the problem requires shared memory; dynamic configuration of processes, processors, or memories; or arbitrary PE-to-memory or PE-to-PE connectivity. However, for many other problems, the reconfigurability of switches makes these architectures appealing.

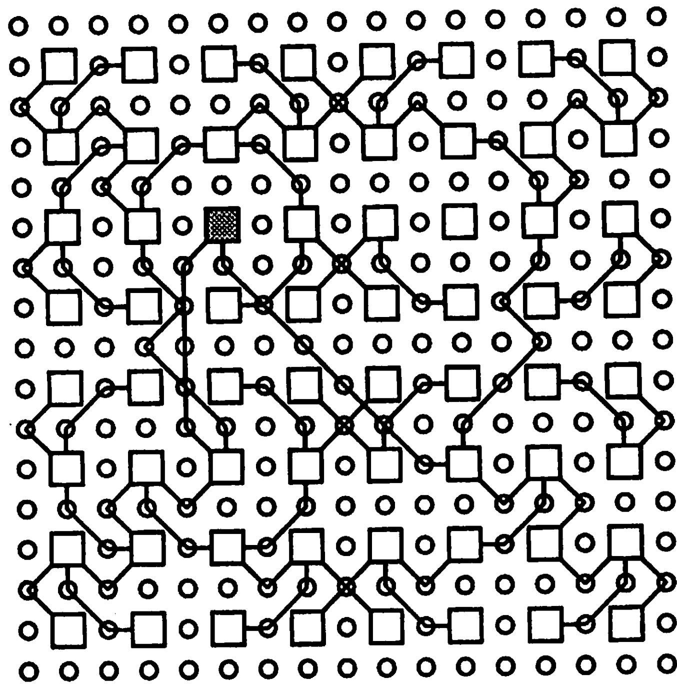


Figure 2–4: A complete binary tree (root shaded) embedded in a CHiP lattice.

2.1.3 Reconfigurable Shared Resource Machines

A large number of configurable processors share resources. The primary difficulty with reconfigurable shared resource processors is performance degradation due to resource contention. Resource competition may occur among communication channels, memories or processors. While many hardware implementations have primitive resource management, efficient use of these architectures often requires user-level knowledge of process interactions.

Many systems allow permutation of processor-to-memory assignments using multistage switching networks, as in Figure 2–5. In such systems, if communication is implemented using shared memory constructs accessed through this network, assignments of processors to memories can overburden switches and restrict potential channels of communication within the interconnection network (Figure 2–5). Current processor-to-memory interconnection networks[53] are primarily limited to a few flexible designs; banyan-type networks are common[53,64,129]. Partitionability, permutability, and fault-tolerance are among the major concerns in selecting the appropriate network[147].

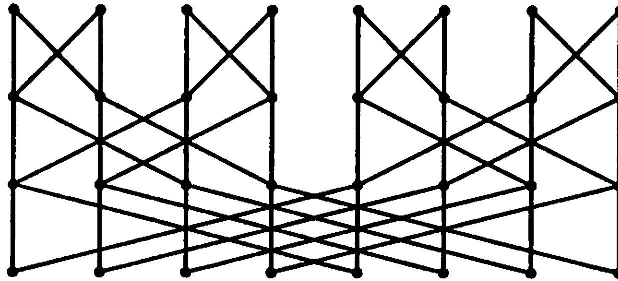


Figure 2–5: An 8x8 SW-banyan multistage switching network.

The TRAC[146] and PASM[148] architectures are designed with multiple SIMD and MIMD/SIMD computing in mind. Simple partitioning of the array and assignment of regular communication patterns make the machine reconfiguration possible from compiler generated directives. As might be expected, communication that is irregular or data-dependent requires some user direction. Contention of re-

sources demands costly multiplexing or serialization of messages to maintain data consistency — a service which often renders static resource allocation difficult.

The Cedar architecture[58] uses a hierarchical memory and processor organization to localize communication and minimize contention. Here, too, the user is not aware of the underlying hardware communication structure and must depend on operating system components to perform efficient mapping of communication.

The Ultracomputer[142], the RP3[125], and the BBN Butterfly™ are machines with a single global memory. Communication occurs through the memory and thus few restrictions have been placed on potential communication structures. Global memory is partitioned, however, so as to provide each processor local memory which has minimal access latency; other memory references are processed by a fetch-and-ϕ combining network which maintains consistency of caches and nonlocal memory. In order to efficiently map communication structures, one must account for pseudo-hierarchical memory access times and contention. There is, in fact, some indication that if contention is not properly controlled, the performance of these systems degrades considerably[126]. Some hardware models have placed processors directly at each switch component of the multistage network, in hopes of employing the regularity of these networks at increasing communication throughput[134] and decreasing the tendency toward developing "hot spots."

2.1.4 Summary

There are a number of diverse architectures capable of providing massive parallelism, but each of these architectures imposes limitations on the logical communication structures that can be efficiently implemented. Currently these algorithms must be mapped by hand, but as we improve our understanding of the physical limitations and logical requirements of communication, we expect much of the mapping process to be automated. It will then be necessary for the programmer to accurately specify logical communication structures. We investigate, next, the environments that currently support the programming of these processors, to better evaluate their effectiveness in efficient, concise specification of communication.

2.2 Programming Constraints

The introduction of parallelism has been thwarted by the costs of retooling programming environments and changing programming methods. This has caused most parallel programming languages to draw heavily on the facilities provided in their sequential predecessors. We divide these approaches into two categories: those that support *monolithic codes* in which parallelism is automatically extracted from a single program, and those that support *multilithic codes* in which parallelism is explicitly provided by the user who supplies code for each process.

2.2.1 Languages Supporting Monolithic Codes

Several parallel programming environments provide support for parallelism by generating parallel code from existing sequential languages. These compilers perform dependency[91,96,123] and data flow[56] analyses to establish independent *threads* of code that can be executed simultaneously. Since the compiler dictates the distribution of code, it also identifies the (usually irregular) logical structure of communication; these are identified by noting dependency arcs whose source and destination instructions lie on different processors after the code has been distributed (Figure 2–6). The arcs show how data migrates logically or physically. We will be less interested in these languages, as the user is fundamentally detached from the process of generating communication structures.

Another approach is to supplement existing languages with declarations and control constructs that help identify independent operations[113]. Glypnir[103] and Tranquil[95] provided array-based parallel structures and control statements for the ILLIAC IV. However, even some simple operations suggest more complex data distribution strategies[102] such as the skewed matrix distribution of Figure 2–7. More recently, Refined C[49] and Blaze[113] share this approach, combining flexible user directives with sophisticated program restructuring techniques.

A natural technique for extending sequential languages to support massive parallelism is to use sequential data structuring techniques. For example, the *structured process* of Li, Wang and Lavin[108] declares processes as an n-dimensional grid

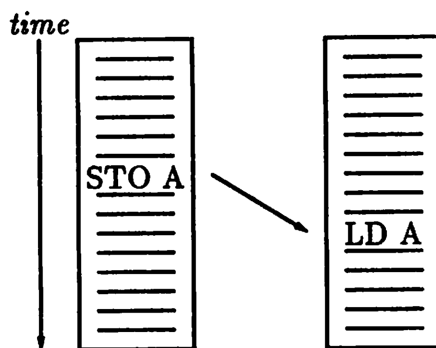


Figure 2–6: Communication induced by dependencies in distributed code.

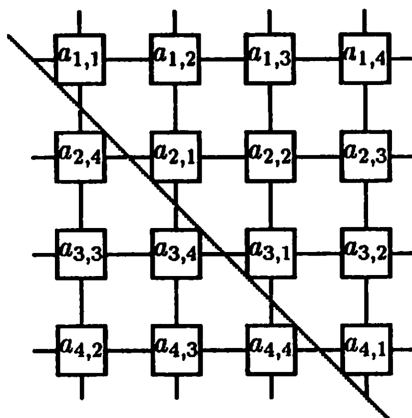


Figure 2–7: A skewed array used in PDE solution in the Illiac processor.

and provides structure macros that map distant process labels onto ports of local processors. The image histogram program of Figure 2–8, for example, demonstrates the use of structure declarations (the `strdef` lines) to set up a mesh-connected structure. These macros are then used to allow local access to remote resources. Their hypothetical hardware assumes no restrictions are placed on general processor-to-processor communication. Furthermore, the structure macros may be circumvented to generate erroneous code. The PASM project uses an extended C where existing C types may acquire n new dimensions that are associated with an n-dimensional grid of processor elements. PASM's *selectors* allow operations on restricted sets of processor elements.

The Occam and Occam-2 programming languages[78] require a monolithic code body which specifies how sets of processes are dynamically loaded onto Transputer processors. Unlike most other languages, Occam assumes an assembler-like attitude about parallel programming (note the low level code in the tree summation algorithm of Figure 2–9) and potentially provides mapping instructions (not shown).

2.2.2 Languages Supporting Multilithic Codes

The remainder of the languages we will investigate explicitly specify parallelism through the individual programming of processors. The major characteristic in this paradigm is the medium for declaring ***process structures*** — a communication structure annotated with process codes.

For systolic array processors, a small number of process codes are written and then distributed. Here, substantial programmer concern is directed toward the decomposition of programs[58,67,98,107] and proofs of correctness[33,67,77]. Information necessary for specification of communication structures (*e.g.* channel direction) is necessary for proofs of correctness. Compilers for systolic array code[100] assume the programmer has ***a priori*** knowledge of the processor array structure. This is true, even for the Wavefront Array Processor (Figure 2–10). Systolic algorithms with a variety of topologies can be embedded, but at a cost to the programmer,

```
structured process hist[M][M](limage)
int limage[N/M][N/M]
{
#strdef east(piddyn) (PID0)(PID1+(1<<piddyn))
#strdef west(piddyn) (PID0)(PID1-(1<<piddyn))
#strdef south(piddyn) (PID0+(1<<piddyn))(PID1)
#strdef north(piddyn) (PID0-(1<<piddyn))(PID1)
    int h[256], i, j; /* h stored histogram */
    int mask1, mask2;

    /* stage 1 computes the histogram of an image */
    for (i = 0; i < N/M; i++)
        for (j = 0; j < N/M; j++)
            ++h[limage[i][j]];

    /* stage 2 computes histogram per row */
    for (j = 0; (1<<j)<M; j++) {
        mask1 = ~(~0<<j);
        mask2 = ~(~0<<(j+1));
        if (((PID1&mask1)==mask1)&&((PID1&mask2)==mask2)) {
            for (i = 0; i < 256; i++)
                h[i] += h[i].west(j);
        }
        else if (((PID1&MASK1)==mask1)&&((PID1&mask2)!=mask2)) {
            for (i=0; i < 256; i++)
                h[i].east(j)=h[i];
        }
    }


    /* stage 3 computes historgram on column */
    for (j = 0; (1<<j) < M; j++) {
        mask1 = ~(~0<<j);
        mask2 = ~(~0<<(j+1));
        if (((PID0&mask1)==mask1)&&((PID0&maks2)!=mask2)
                                  &&(PID1==(M-1))) {
            for (i = 0; i < 256; i++)
                h[i].north(j)=h[i];
        }
    }
}
```

Figure 2-8: Image histogram computation using structured processes[108].

```
-- Tree summation
extern proc pnum(value n):

def MAXPROC = 128:
chan c[MAXPROC]:

-- Root has only children, sums and prints result
proc root(chan l, chan r) =
  var left,right:
  seq
    l ? left
    r ? right
    pnum(left+right)

-- Interior has both children and parent, sums
proc interior(chan l, chan r, chan p) =
  var left,right:
  seq
    l ? left
    r ? right
    p ! left+right

-- Leaf process has only parent, sends value
proc leaf(chan p, value n) =
  seq
    p ! n

seq
  par
    root(c[2],c[3])
    par i = [2 for MAXPROC/2-2]
      interior(c[2*i],c[2*i+1],c[i])
    par i = [MAXPROC/2 for MAXPROC/2]
      leaf(c[i],i)
```

Figure 2–9: Tree summation on the Transputer using Occam.

who must map the logical communication structure onto a square mesh in a manner that preserves a wavefront property.

MIMD programming of processing elements can be found in several processor arrays. The Blue CHiP Project's programming environment, Poker[154], allows individual programming of process elements or process element sets. Global information about the algorithm's communication structure is acquired in a distinct phase of specification. Poker's communication structure can be specified graphically, or textually as is done by Berman[24] (Figure 2–11). Experience suggests that unaided graphical description of large systems is tedious. Yet, the separation of communication and computation is often an advantage, especially when process code is independent of communication context.

The Cosmic Cube's C compiler allows point-to-point communication using message passing. This necessarily introduces performance penalties for certain communication structures that have large degree; a binary n-cube *requires* multihop message passing for processors with fan-out greater than n. The extra cost in message routing suggests that users who perform mapping themselves to avoid message passing will benefit by increasing performance. It should be noted, however, that a simple message routing scheme supports the view of the cube as a pool of processors. This programming paradigm is similar to those of CSP and other concurrent languages[62] (Ada, Concurrent Pascal, Modula, *etc.*) which typically assume that arbitrary and dynamic connectivities are possible.

TRAC's CSL[31] identifies communication between processes through shared memory constructs. Figure 2–12, for example, describes a ring topology. As a macro extension to traditional sequential languages, the mechanisms of CSL are easily adaptable to any MIMD programming environment. Like Poker's graphical methods, communication description is separate from program specification and is essentially a lightweight task control language. CSL has yet to be used in environments with massive levels of parallelism.

Some systems do not support *any* description of communication. Linda[63] provides a tagged message passing system which fails to describe any communication structure explicitly. All communication occurs through *tuples* added to a global

```
BEGIN
    !  Initialization phase:  one wavefront;
    SET COUNT 1;
    REPEAT
        WHILE WAVEFRONT IN ARRAY DO
            BEGIN
                CASE KIND =
                    (1,*) :     FLOW A, LEFT;
                    (*,1) : FLOW A, UP;
                    INT   : BEGIN
                                FLOW A, LEFT;
                                FLOW A, UP;
                            END;
                ENDCASE;
            END;
        DECREMENT COUNT;
    UNTIL TERMINATED;
    SET COUNT <K>;    !Represents the number of iterations.;
    REPEAT
        WHILE WAVEFRONT IN ARRAY DO
            BEGIN
                CASE KIND =
                    (*,1) : FETCH E, UP;
                    (1,*) : FETCH B, LEFT;
                    INT   : BEGIN
                                FETCH E, UP;
                                FETCH B, LEFT;
                            END;
                ENDCASE;
                ADD B,E,A; FETCH D, RIGHT; FETCH C, DOWN;
                ADD D, A, A; ADD C, A, A; DIV A, 4, A;
                FLOW A, RIGHT;  FLOW A, DOWN;
                CASE KIND =
                    (*,1) : FLOW A, UP;
                    (1,*) : FLOW A, LEFT;
                    INT   : BEGIN
                                FLOW A, UP;
                                FLOW A, LEFT;
                            END;
                ENDCASE;
            END;   !Of WHILE block;
        DECREMENT COUNT;
    UNTIL TERMINATED;
ENDPROGRAM.
```

Figure 2 10: A Wavefront code for PDE solution[101].

```
tree
nodemin = 1
nodecount = 15

procedure root
    nodetype : { i == 1 }
    port lchild : { 2*i }
    port rchild : { 2*i+1 }
    port dataout : output results

procedure internal
    nodetype : { i > 1 && i < 8 }
    port parent : { i/2 }
    port lchild : { 2*i }
    port rchild : { 2*i+1 }

procedure leaf
    nodetype : { i >= 8 }
    port parent : { i/2 }
    port datain : input data
```

Figure 2–11: Specification of a binary tree structure in the Prep-P environment[24]. Connectivity is determined by associating ports with id's of adjacent processes.

```
CHANNELS
    (moveright[i] = DATACHANNEL FROM f-c-p(i) TO
     f-c-p((i+1) MOD (N+1));
     moveleft[i] =  DATACHANNEL FROM f-c-p(i) TO
     f-c-p((i+N) MOD (N+1));)
    RANGE i = 0 TO N;
```

Figure 2–12: A CSL cycle specification[31].

tuple space. Tuples may be read and removed through a process of restricted range queries. The dynamic, data-dependent aspect of this communication technique makes analysis difficult.

2.2.3 Summary

The discrepancies between parallel algorithm and programs in existing parallel languages are significant. We expect, however, that as parallel programming becomes better understood, it will be possible to represent algorithms more directly and that the explicit specification of communication structures — which often play an important role in structuring the algorithm — will be critical.

2.3 Conclusion

The development of effective parallel programming environments has been slowed by the diversity of proposed target architectures and by the inapplicability of many sequential programming facilities. Central to both of these impediments is interprocess communication which is supported to different degrees on different architectures and which (as we will see in the next chapter) does not have a direct analog in the sequential domain. As a result, there is no general architecturally independent method for specifying communication in parallel algorithms. Yet logical communication structures are fundamental to our understanding of these parallel algorithms and Their efficient implementation is critical to the performance of parallel programs.

Chapter 3
COMMUNICATION STRUCTURES

Perhaps the greatest hurdle for implementors of massively parallel algorithms is understanding the effects of communication. The complexity of communication can be overwhelming at every stage.

Design. The algorithm designer, for any particular problem, must resolve the conflicting goals of distributing computation and minimizing communication costs. Subtle differences in parallel decomposition can drastically effect performance. Communication is arguably the most difficult consideration in algorithm design[152]. What form of specification most accurately and naturally reflects the designer's decisions?

Implementation. Often the programmer must map his algorithms onto physical hardware structures which differ in size and topology from his logical process structures. Assignment of resources is the most complex problem facing the implementor of efficient algorithms[27] and the biggest threat to portability; it is only possible when the ramifications of the algorithm's communication structure are fully understood. What assistance is necessary to support implementation of communication in massively parallel algorithms?

Debugging. The programmer, amid broken computations, must reconcile traces of distributed asynchronous processes with expected behavior. Standard debugging techniques cannot be applied because parallel systems do not have a consistent global state or reproducible behavior. Debuggers for massively parallel systems must employ new tactics for the manipulation of tightly coupled processes, including the recognition of abstract communication patterns which aid the programmer in

recognizing the system's progress[18,75,76]. What abstraction of communication extends naturally to the debugging and animation of parallel programs?

Communication issues are fundamental to every aspect of parallel programming. Thus, the success of parallel programming environments hinges on the proper treatment of communication throughout the software life cycle. In this chapter, we discuss support for parallel communication as a structuring technique.

3.1 Communication: Control *and* Structure

The execution of a sequential program is determined by its control statements and its data structures. Consider, for example, the sorting problem of Figure 3–1. The execution of the statement labeled (*a*) is determined by the surrounding looping

```
        sort(n,values)
            int n, val[];
        {
            int i, bubbled;

            do {
                bubbled = FALSE;
                for (i = 0; i < (n-1); i++) {
                    if (val[i] > val[i+1]) {
(a)                     c = val[i];
                        val[i] = val[i+1];
                        val[i+1] = c;
                        bubbled = TRUE;
                    }
                }
            } while (bubbled);
        }
```

Figure 3–1: A sequential **bubble sort** program.

and branching constructs but it is the ordering of array indices that supports the ultimate ordering of the sorted data. Thus, a correct understanding of both control and data structuring is essential. (Note that not all aspects of a program are as essential to our understanding. Engineers often use, for example, an ***abstraction of***

scale: the accuracy of the system description is independent of problem size. Thus in the `bubble sort` program we might describe bubbling as "moving the largest value to the end of the array" without reference to the size of the array.) Control and data structuring are fundamental aspects of sequential computation.

Which role does communication play in parallel computation?

Communication among loosely distributed processes is almost always used as a form of control. Because processes are inherently asynchronous, communication serves to coordinate process states. As an example, algorithms which are naturally composed of several temporal phases will perform a *barrier synchronization* between phases (see Figure 3–2). Once the barrier has been met by participating processes, each proceeds with the assumption of a coordinated global state.

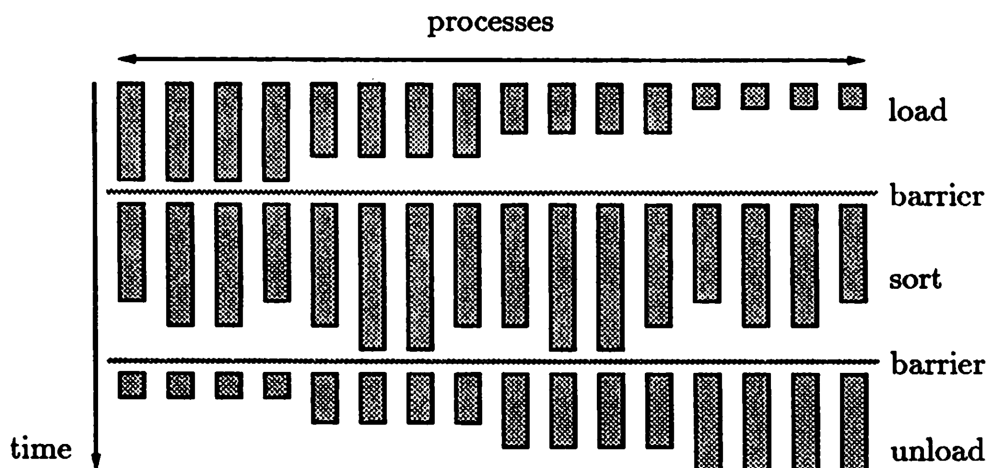


Figure 3–2: Communication as control: barriers in a multi-phase algorithm.

In contrast, massively parallel systems use communication as a mechanism of both control *and* structure. While most distributed abstractions of communication (multicasting, remote procedure calls, *etc.*) are easily adapted to massive parallelism, the nature of tightly coupled systems requires processes to communicate frequently and in highly structured patterns. The structured nature of the process interaction is so apparent that tightly coupled algorithms often manifest themselves as models of the physical systems they are used to investigate[48].

Jamieson[80] observes the two-sided nature (control *vs.* structure) of communication as the side-effect of algorithm design. At an early stage of program development, a decision is made to provide ***parallelism of function*** or ***parallelism of data***. Where parallelism of function is desired, communication serves to ***control*** function interaction; where parallelism of data is appropriate, communication serves to ***structure*** the computation. As examples, pipelined computations exhibit functional parallelism, while many physical modeling problems employ data parallelism.

As we have seen in Chapter 2, most current parallel programming environments are extensions of sequential environments which do not directly consider communication a structuring mechanism. To effectively implement tightly coupled algorithms, it is important to provide support for the specification of structured communication. It is not clear that this support can be achieved, in any general way, by extensions of distributed or sequential techniques.

In the next section, we survey structures induced by various parallel programming paradigms.

3.2 Programming Paradigms

A number of paradigms have been useful in the design of efficient parallel algorithms. These include the divide-and-conquer technique, which has also proven useful in sequential algorithm design, and others, like systolic control, which stem directly from analysis of parallelism. The relative merits of these paradigms have been investigated[30,74,120,162] but we will be more interested in the characteristics of the communication structures they induce.

We make two disclaimers. First, we do not consider all parallel design techniques. In particular, we are not interested in techniques that are not applicable to massively parallel ensemble machines. Algorithms designed with such techniques can often be replaced by algorithms which are of interest. Second, we do not consider the design of "optimal" algorithms constructed using specialized, problem-specific techniques. Such algorithms usually have less-than-optimal counterparts designed using simpler, more general methods which perform acceptably well.

We identify and investigate four major paradigms — divide-and-conquer, systolic methods, relaxation and routing techniques. These paradigms account for a large number of algorithms implemented on local memory MIMD machines. We discuss the characteristics of the communication structures required by each paradigm and we consider the communication structures arising from hybrid techniques.

3.2.1 Divide-and-Conquer

Divide-and-conquer is a sequential programming technique that constructs conceptually simple solutions to large problems. The problems are decomposed into small, related subproblems in a manner that allows their subsolutions to be recombined to form a logically simple solution. Examples of sequential use of divide-and-conquer include Mergesort, Quicksort, polynomial and matrix multiplication, database techniques, and various problems of computational geometry[144]. These algorithms depend on recursion to implement divide-and-conquer. This abstraction avoids concern for problem size and often allows divide-and-conquer to provide an optimal method of solving an otherwise unwieldy problem.

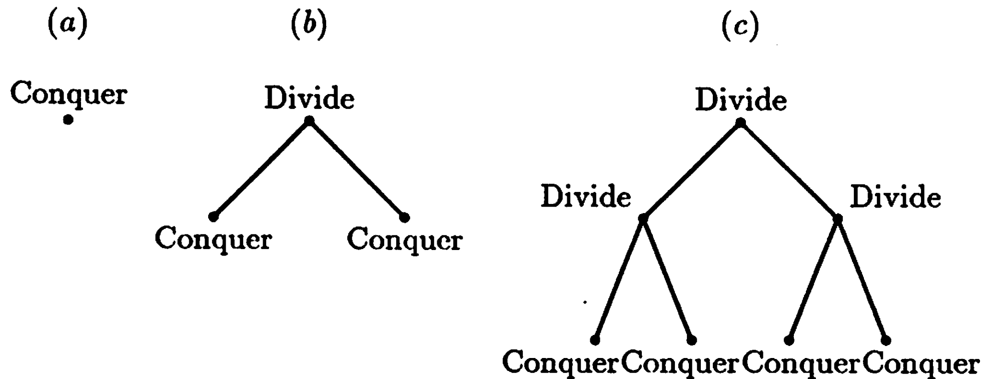
Divide-and-conquer is often also applicable in parallel algorithm design, especially for algorithms which achieve speedup through data parallelism. The method identifies parallelism inherent to the problem and then partitions the problem so that solutions are allowed to progress independently. Examples of this technique can be generated from algorithms based on sequential divide-and-conquer; data is distributed through the processor array, processed in parallel, and the results are recombined.

Example. Consider the problem of sorting a set of $N = 2^n$ numbers. Suppose efficient serial techniques exist for sorting smaller arrays of $M = 2^m$ numbers. The algorithm of Figure 3–3 outlines a parallel Mergesort. Note that this algorithm shows no recursion. The recursion has been "unrolled" across the process structure. This use of communication is similar to a remote procedure calls, but is globally structured in a manner which is determined by the structure of the data.

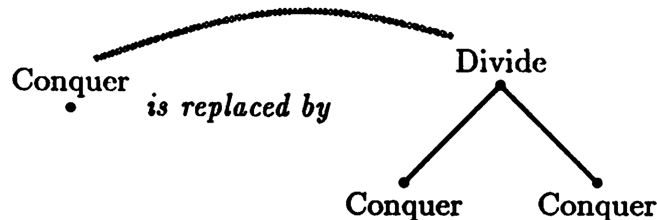
```
Sort()
{
    int n, *array;
    port ParentPort, LeftPort, RightPort;

    n <- ParentPort;
    if (n <= M) {
        read array[0..n-1] from ParentPort;
        sort array locally;
        write array[0..n-1] to ParentPort;
    } else {
        LeftPort <- n/2;
        repeat { LeftPort <- ParentPort } n/2 times;
        RightChild <- n/2;
        repeat { LeftPort <- ParentPort } n/2 times;
        merge LeftPort and RightPort values,
            forwarding to ParentPort;
    }
}
```

Figure 3–3: A parallel Mergesort program.

Characteristics. A sketch of the algorithm's progress, seen as a transformation of the communication structure, might appear as:



Each **conquer** labels a conquering computation; a **divide** labeled computation resorts to applying divide-and-conquer. Thus the algorithm is initiated as a single process (a). Then, if analysis suggests dividing is necessary, it divides into three processes (b). This step occurs repeatedly, until data can be completely conquered locally (c). We might more compactly encode this transformation in structure by



Of course, the conversion of a node from a conquerer to one that divides does not destroy its context. For example, if the conquering node has a parent, that connection is unaffected by this rewriting rule. Here, the gray line suggests the divide node's ***inheritance*** of the **conquer** node's characteristics.

Because of the recursive nature of divide-and-conquer, the algorithm usually consists of two pieces of code: one which divides and another which conquers. The conquering code, which serves to terminate the recursion, is usually uniformly distributed along among the leaves of the communication structure. The code responsible for dividing the problem is recursive and performs a similar partitioning on

successively smaller domains at each level. This leads to communication structures resembling binary trees.

While divide-and-conquer techniques could make data-sensitive divisions, designers often opt for a fixed degree of partitioning; any uneven distribution of work can always be accommodated by deeper recursion. The simplest divide-and-conquer algorithms, then, lead to fixed-degree, tree-like communication structures. While most machines cannot directly embed such structures, research has been successful at finding optimal embeddings of trees and forests in popular architectures[13,73].

Some problems cannot be partitioned without subproblem interaction. When subproblems are duplicated, solutions to smaller problems can be computed once and distributed appropriately. This method — known as *dynamic programming* — is used for solving combinatorial problems. When the subproblems share portions of the data, as in Figure 3–4, preprocessing of the data can increase subproblem independence or overlapping subproblems can work on independent copies of the overlapping area. Subproblems are sometimes allowed to interact. Care must always be taken to correctly combine subsolutions.

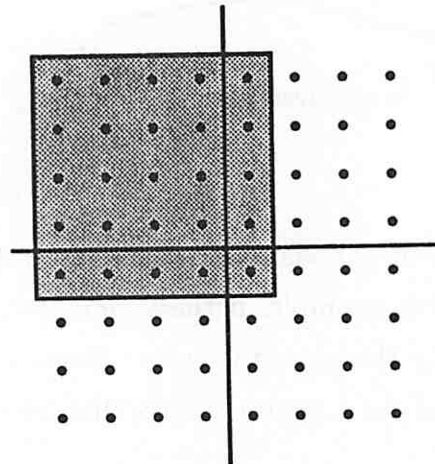


Figure 3–4: Overlapping divide-and-conquer domains.

More exotic variations on divide-and-conquer have also been successful. Bentley documents a modified divide-and-conquer paradigm for solving problems of mul-

tidimensional data[19]:

> "[T]o solve a problem of N points in k-space, first recursively solve two problems each of $N/2$ points in k-space, and then recursively solve one problem of N points in $(k-1)$-dimensional space."(page 214)

There are a number of topologies that support this type of computation. Figure 3–5 (labeled with $[N, k]$ pairs) demonstrates the use of an unbalanced ternary tree (a) and a binary tree with sibling connections (b).

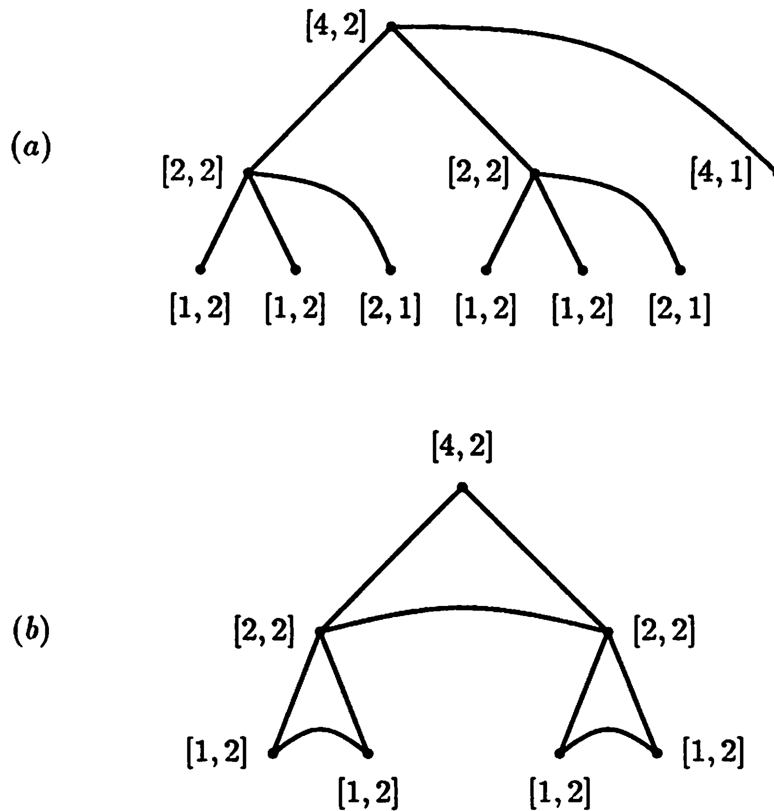


Figure 3–5: Divide-and-conquer layouts for multidimensional domains.

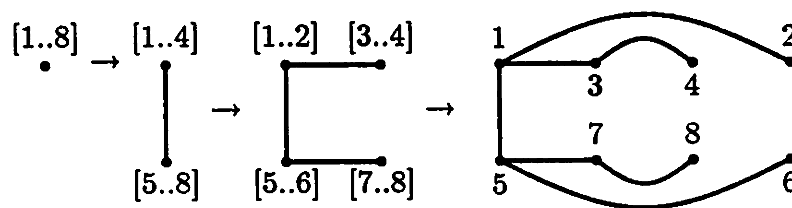
Divide-and-conquer has yielded some elegant parallel algorithms. Because an assumption of the divide-and-conquer technique has been the identification of independent subproblems, the communication requirements are rarely forced to deviate significantly from tree structures. Even so, it remains important to identify the

communication needs accurately. Efficient embedding of these structures in multiprocessors depends on characteristics of communication, as we see in the following example.

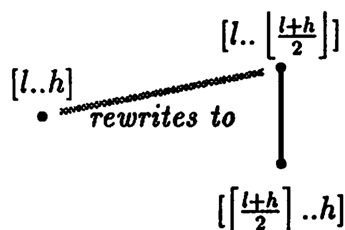
Example. Naïve implementation of divide-and-conquer in a local memory machine can lead to severe performance degradation. Because large subproblems must be communicated to subordinate conquering processes, distribution of data often becomes all-consuming. Horowitz and Zorat[74] suggest

> "[a] processor that has received its data from its parent may...immediately proceed to further subdivide the data and pass it to its own subprocessors. What this amounts to is that the subprocessor has gained access to a substantial portion of the data, without making much use of it."(page 583)

When divide-and-conquer is supported by a binary cube architecture, "reuse" of processors can achieve better performance. Instead of partitioning the problem into two subproblems which are both exported to subordinate processors, one problem is exported, and the other remains resident. Each active node then acts as a conquer node. Consider a sketch of communication required to support the divide-and-conquer on a problem composed of eight large subproblems. The expansion of the computation then becomes



and the stage-to-stage transformation is encoded as



with inheritance of attributes again suggested by the gray line.

3.2.2 Systolic Methods

Systolic algorithms stem from a dependency analysis of data in a repetitive computation. In particular, when the computation is stated as a recurrence, and each unrolling of the recurrence is assigned a process, the code necessary to support the processes is uniform. This means that special purpose hardware can be built to solve large problems at relatively low cost. Even when the target architecture has not been specially designed, systolic methods identify useful algorithms for a wide variety of problems.

Example. Multiplication of $n \times n$ matrices is defined by the summation

$$c_{i,j} = \Sigma_{k=1}^{n} a_{i,k} b_{k,j}.$$

This formula is often the basis for sequential solution to the problem, which assumes values for $a_{i,k}$ and $b_{k,j}$ are globally accessible. Systolic analysis requires identifying a recurrence, which makes the data dependencies clear:

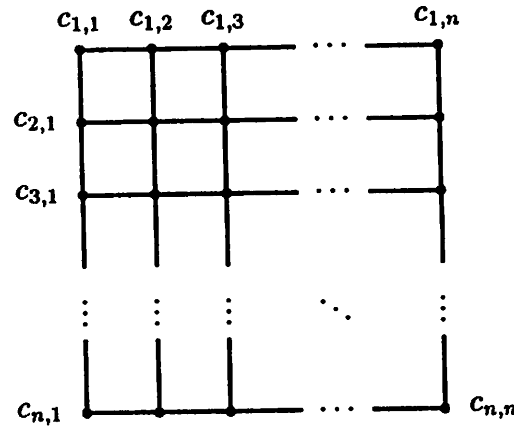
$$c_{i,j}^{(0)} = 0, \qquad c_{i,j}^{(k+1)} = c_{i,j}^{(k)} + a_{i,k} b_{k,j}, \qquad c_{i,j} = c_{i,j}^{(n)}$$

The middle expression relates successive approximations of $c_{i,j}$, and $a_{i,k}$ and $b_{k,j}$. When a virtual process is assigned to computing the recurrence relation for particular values of i, j, and k, dependencies induced by the recurrence (between $c_{i,j}^{(k+1)}$ and $c_{i,j}^{(k)}$) indicate the need for communication. Other values ($a_{i,k}$ and $b_{k,j}$) must either be initially distributed or shuffled to where they are used in the various computations.

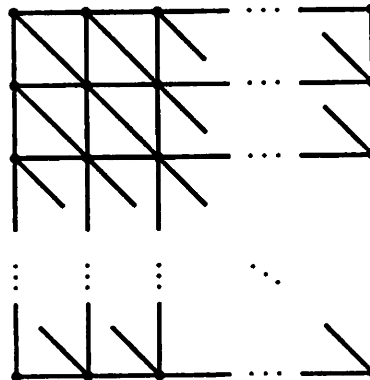
The set of virtual processes and channels necessary to compute any particular point of the recurrence relation thus induces a directed, acyclic graph (DAG). When a decision is made regarding the grouping of virtual processes into processes, channels required by the DAG's induce the communication structure necessary to support the particular algorithm.

Grouping of computations, therefore, plays an important role in determining the communication needs. If, for example, we assume that each $c_{i,j}^{(k)}$ for $1 \leq k \leq$

n is computed by the same process (*ie.* the C matrix is computed in place), a communication structure for the problem is a rectangular mesh:

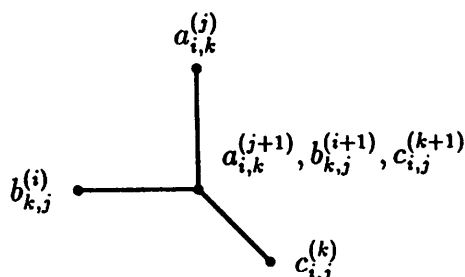


This configuration is appropriate for the Wavefront Array Processor which has the mesh topology. Another assignment might require the product matrix pass diagonally through the array. This suggests the algorithm of Kung and Leiserson, which requires a hex connected mesh, and is useful if the computation is a component of a series of communication structures, *ie.* a ***macropipeline***.



Characteristics. We consider two sketches of the communication required to implement a systolic algorithm. The communication structure for systolic systems may be constructed by "tessellating" graphs, where communication is specified from the point of view of a process with external dependencies. Each node is labeled with the value it produces. Thus the central node responsible for computing the recurrence based, in our example, on three values, is connected to nodes labeled with

values it requires. The central node is labeled with the three values it produces. For the matrix multiplication, we have the following graph.



(We have introduced two recurrences, $a_{i,k}^{(j+1)} = a_{i,k}^{(j)}$ with $a_{i,k}^{(1)} = a_{i,k}$ and $b_{k,j}^{(i+1)} = b_{k,j}^{(i)}$ with $b_{k,j}^{(1)} = b_{k,j}$, which suggest a distribution strategy for a and b values.) The process of grafting these n^2 graphs together requires *identifying* nodes whose labels are mentioned by the same recurrence. The result is a graph whose nodes are grouped to generate the communication structure.

Processors which support systolic algorithms are special purpose, and result in interconnections that directly reflect the communication structure of the algorithm. Often, this structure is identified by pipes through the array which perform affine transformations on the data at each transition step. Our second sketch relates pipelines of data. For all $1 \leq i, j \leq n$, the recurrence in a becomes

$$a_{i,k} \rightarrow a_{i,k}^{(1)} \rightarrow a_{i,k}^{(2)} \rightarrow \cdots \rightarrow a_{i,k}^{(n)},$$

the recurrence in b becomes

$$b_{k,j} \rightarrow b_{k,j}^{(1)} \rightarrow b_{k,j}^{(2)} \rightarrow \cdots \rightarrow b_{k,j}^{(n)},$$

and the recurrence in c becomes

$$c_{i,j} \rightarrow c_{i,j}^{(1)} \rightarrow c_{i,j}^{(2)} \rightarrow \cdots \rightarrow c_{i,j}^{(n)}.$$

The interaction of these $3n$ pipelines is suggested by the values produced by an instance of the recurrence computation:

$$(a_{i,k}^{(j+1)}, b_{k,j}^{(i+1)}, c_{i,j}^{(k+1)}).$$

The communication structure, therefore, is determined by the structure of the data that is consumed and produced. We provide a number of figures (Figure 3-6) which depict communication structures that result from various recurrences.

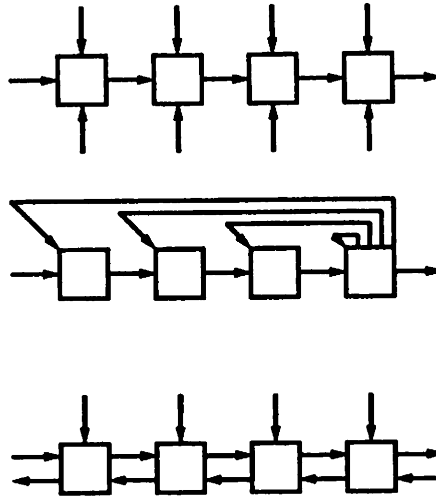


Figure 3–6: Communication induced by various recurrences.

A requirement of systolic design is uniformity in the computations that constrain the communication structure. In our example, both algorithms perform an inner product ($c = c + a * b$) in all processes of the array.

3.2.3 Relaxation

Globally consistent solutions to problems can be constructed from good local approximations. Relaxation algorithms are characterized by processes that proceed without global synchronization. These algorithms do not depend on globally communicated values *per se*, but rather generate local approximations whose consistency is determined by occasional consensus. Each process performs a computation, broadcasting its value in some local neighborhood and normalizing its local state by analyzing values broadcasted by neighbors. Algorithms for PDE approximation, and finite element analysis are naturally cast in terms of relaxation.

Example. We consider Jacobi's iterative solution to the partial differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

on the rectangle $0 < x < a$, $0 < y < b$ with Dirichlet conditions ($u(x,0) = u(x,b), 0 < x < a$ and $u(0,y) = 0, u(a,y) = f(y), 0 \leq y \leq b$). An iterative approximation can be made on an arbitrarily finely discretized $m \times n$ mesh using

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left[u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}\right]$$

where $u_{i,j}^{(k)}$ is the kth iterative approximation to $u(j \cdot \Delta x, i \cdot \Delta y)$ where $\Delta x = \frac{a}{n}$ and $\Delta y = \frac{b}{m}$ (see Figure 3–7). Parallelization occurs when each point in the mesh is

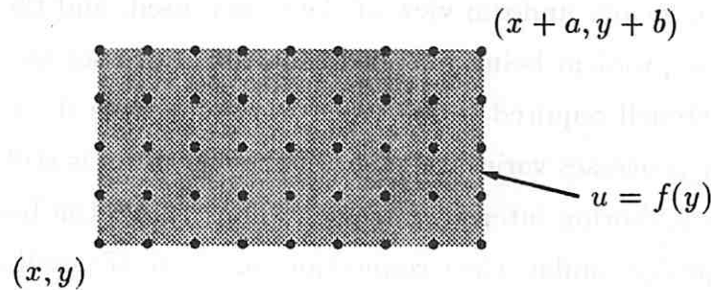
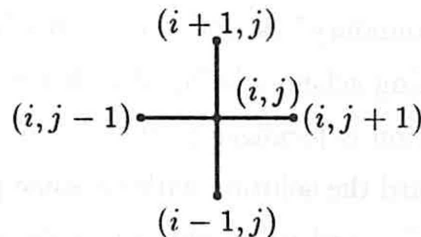


Figure 3–7: A rectangular domain for PDE approximation.

supported by a process which coordinates its computation with its neighbors using a *5-point stencil*:



The communication structure is determined by identifying similar labels of stencils which represent each point. The 5-point stencil generates a communication structure with a mesh connectivity and a rectangular shape determined by the problem

boundary. A number of other stencils have been used to improve convergence, including those of Figure 3-8.

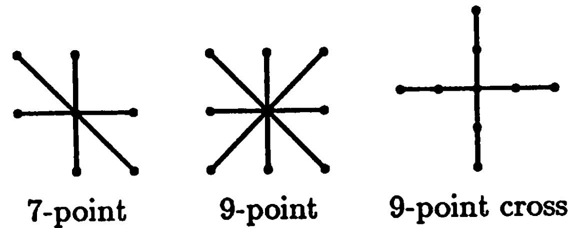


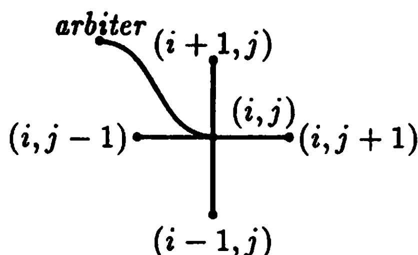
Figure 3-8: Stencils used in solution of PDEs.

Characteristics. Communication in relaxed algorithms is determined by two components: the locally uniform view of the stencil used, and the global structure suggested by the problem being modeled. Interior processes are uniformly coded to support the stencil required to solve the problem. While the exact connectivity of the boundary processes varies, the *logic* of the algorithm is still to broadcast the conditions to neighboring interior processes. Thus, while the boundary processes are not topologically similar, their connections are *logically* similar.

Like the structures of systolic design, the system can be expanded by adding layers of processes along any of the dimensions. However, because large data sets must be completely contained within the processor array during the entire computation, large computations are often divided into overlapping subproblems[170]. During the computation phase, each subproblem simulates, sequentially, a relaxation algorithm whose "boundary" is updated by the local broadcasts. A fuller treatment of grid partitioning schemes is found in Reed and Fujimoto[135].

Because communication is localized in these algorithms, they can not determine relative progress toward the solution without some periodic global evaluation. Two techniques are typically used to provide accurate computations: (1) an algorithm proceeds a number of iterations which is guaranteed to provide an answer correct within some acceptable tolerance, or (2) processes communicate with a centralized arbiter that monitors progress towards an ultimate solution. Jacobi's method,

and related techniques terminate using after a predetermined number of iterations. However, relaxed algorithms, in general, require a run-time global consensus for termination. In these cases, support for global aggregating and broadcasting of a consensus is required. This modifies our stencil to:



3.2.4 Routing Problems

Routing data among processes plays an important role in algorithms for broadcasting (or multicasting) and sorting data. Sorting algorithms require connectivities which support routing of data to arbitrary processes quickly. In these problems, it is important to support the permutation of n items using n logical channels. On all but completely connected communication structures, arbitrary permutations are only possible when data values are routed using multiple hops. It is common, therefore to consider the process connectivity as a simulator for a *multistage switching network*. In Figure 3–9, for example, each process of the binary cube simulates all switches in a column of the switching network.

Example. Bitonic sorting is accomplished by a divide-and-conquer mechanism. The array of numbers to be sorted into ascending order is divided in half and conquered — the lower half sorted into ascending order and the upper half sorted into descending order. Elements are then pairwise compared ($array[0]$ with $array[N/2]$, $array[1]$ with $array[N/2+1]$, *etc.*) and exchanged, guaranteeing the lowest $N/2$ elements appear somewhere in the lowest $N/2$ positions. Finally, each half is recursively sorted into ascending order. The result is an array of ascending numbers.

To sort a two element array, we require only a bidirectional connection between the two processes. In sorting a four element array, we construct the network

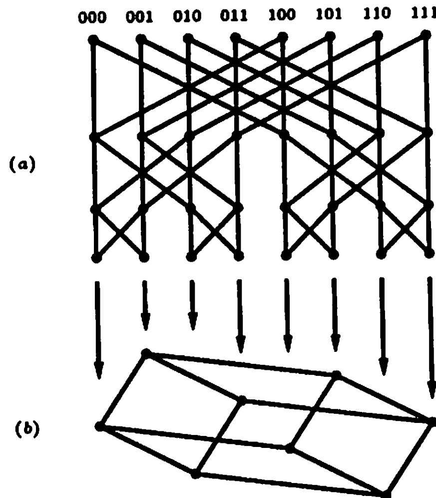


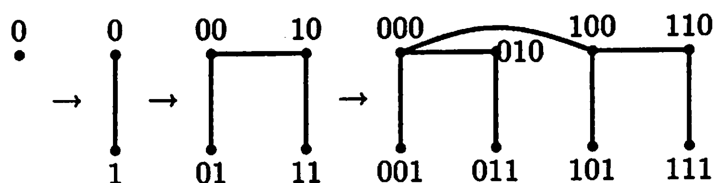
Figure 3–9: A three stage logical network (a) and its projection (b).

from a pair of two element sorters, joined by crossbars. This type of recursive construction is common among routing problems.

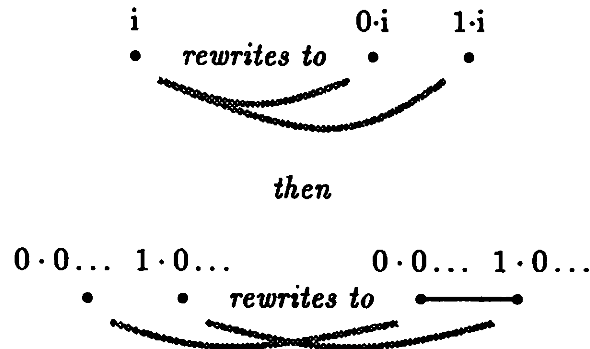
Example. Broadcasting of values is a routing problem. Broadcasting is used in the distribution of code and data in MIMD architectures. When no global channels exist to broadcast values on cube-like architectures, efficient use of communication can be difficult. A technique for efficient distribution of values on binary hypercubes uses the *binomial tree*[73]. Construction of binomial trees is recursive:

> "A 0-level binomial tree has only one node. A n-level binomial tree is constructed out of two $(n-1)$-level binomial trees by adding one edge between the roots of the two trees, and by making either root the new root."

A sketch of a series of binomial trees is thus



A transformation between trees might be depicted as follows ("..." indicates digit repetition, "·" indicates concatenation).



Characteristics. Because routing networks are so communication intensive, they often are defined and constructed in a manner that preserves desirable routing properties, such as the unique path between nodes. When large communication structures are constructed from smaller structures of the same family, the proper adjoining of the networks can insure preservation of these desirable properties.

In the specific case of sorting networks, a *good* sorter[166] must support the simultaneous use of many channels. Thus recursive construction of good sorting networks involves introduction of successively larger numbers of channels. If two n element sorters are combined, the worst case use of the network is the complete swapping of values between halves which requires the introduction of n channels.

Communication structures which support simple routing perform exactly the same routing function at each node. This leads to uniformity of the routing method throughout the network. This leads in many (but not all) cases to symmetric graphs.[1]

We next consider the composition of communication structures.

[1]The shuffle-exchange graph has a uniform routing formula, but it is not symmetric.

3.2.5 Composite Communication Structures

Each of the former paradigms might be considered to be *primitive* because the resulting communication structure cannot be logically decomposed into simpler structures which perform meaningful computations. A dual of this concept is the composition of computations, and their communication structures. In practice, there exist a number of composite communication structures.

Multiphase computing (temporal). When processes perform a series of computations in place, the result is called a *multiphase computation*. Each process in a multiphase algorithm performs a sequence of transformations on data and the steps in the sequence are separated by a barrier synchronization. Examples include PDE approximation which computes an approximation and later computes an array-maximum deviation. Observe the computation structure (Figure 3–10) which shows the communication structure of each phase: one phase requires a mesh and the other a tree. A composite structure concatenates the two structures by adding the communication necessary to align them.

Multiphase computing (spatial). Spatial multiphase computing, or *macropipelining*, is a method for distributing the computation by function. Each algorithm in a task is supported by a systolic array which pipelines data. The pipelines of successive phases are then grafted together to generate a macropipeline computation which itself may be considered a component of a larger computation. Figure 3–11 demonstrates a macropipeline: two matrix multipliers are concatenated to form a two-stage multiplier.

Hierarchical refinement. Browne[30] describes iterative refinement of the communication structure supporting a particular computation. The problem is decomposed to primitive computations at some level of abstraction. Each process at this level is run on a virtual machine which supports the primitive operations of the particular abstraction. If the virtual machine does not coincide with the target hardware, the processes are further refined, perhaps using further distribution. The

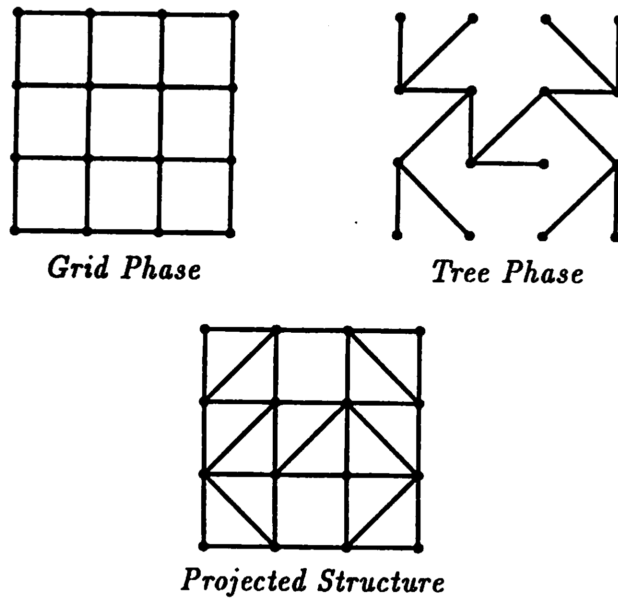


Figure 3–10: Communication supporting PDE approximation.

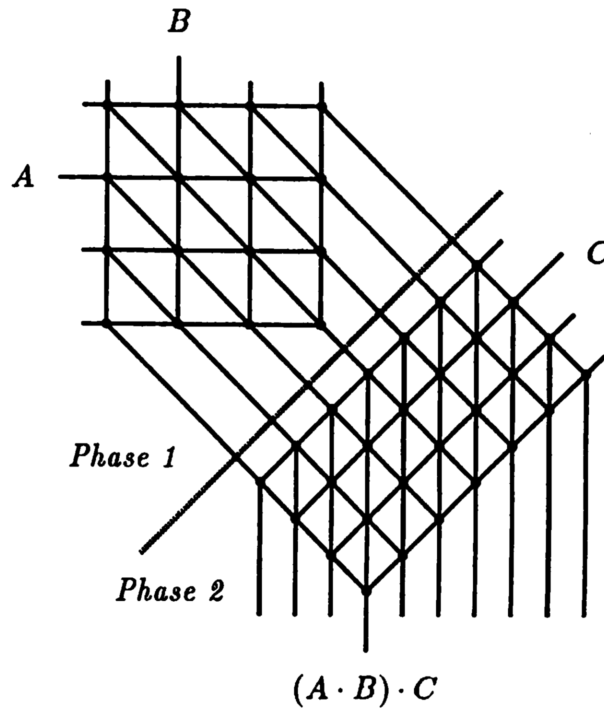


Figure 3–11: A spatially distributed macropipeline.

result is a hierarchical structure which is the product of composing a high-level communication structure with communication structures supporting the primitive operations (see Figure 3–12).

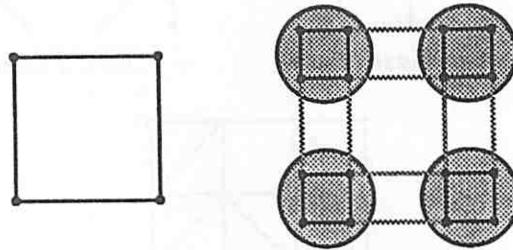


Figure 3–12: Hierarchical composition.

Each of these composition methods suggests joining of two or more communication structures. In temporal multiphase computing, the interfacing of two communication structures is a mapping problem. Macropipelines require grafting component systolic arrays along boundaries. Finally, in hierarchical refinement, the rewriting of high level nodes in communication structures by more refined communication structures resembles composition.

3.3 Metrics of Communication Structures

The construction of communication structures is not an *ad hoc* process. Parallelism is introduced into algorithms for a common purpose — efficiency — and it often results in communication structures with common characteristics. Communication structures, for example, are usually connected graphs: disconnected components would fail to come to common consensus about a solution to a problem. Of course, a full taxonomy of parallel programs will find counter examples to any metric. We use these metrics as guidelines to identifying potential difficulties in the construction of communication structures, much as type checking aids in identifying possible problems in correct assignment of values.

We first discuss characteristics which suggest "regularity." Most measures of regularity are motivated by specific biases in programming massively parallel algo-

rithms: reduction of code specification, reduction in communication overhead, *etc.* The second section suggests reasonable perturbations in otherwise regular communication structures.

3.3.1 Regularity Metrics

Metrics of regularity arise from useful properties of the graph itself and from useful methods of constructing those graphs.

Topological Characteristics

Connectedness is commonly required because disconnected components can not synchronize and could have no global coordination of results.

Low degree is often important because physical structures with large numbers of communication channels are expensive to build and hard to program efficiently. Graphs which represent communication are generally *sparse* — that is the number of channels is often linear in the number of processes or processors — to insure reasonably efficient implementations. Of course, counterexamples can be found, for example, the binary n-cube has $O(n \log n)$ channels.

Symmetry is a paramount characteristic in both algorithms and hardware. Symmetry implies that for any pair of processes, there is an automorphism of the communication structure which exchanges the processes. Thus, every process has the same view of the communication network, an important property for routing and algorithm decomposition.

Low Diameter is often a characteristic of efficient communication structures. For many algorithms, linear structures are not efficient because central nodes can become bottlenecks. Many architectures boast a diameter of $O(\log n)$, although mesh connected structures are $O(\sqrt{n})$.

High Flux represents the ability of a graph to support concurrent communication. The flux of a graph is, essentially, a measure of the worst bottleneck of

communication: if the graph is cut in two in the worst way, it is a measure of the bandwidth of the communication between the two halves. Flux has been used in understanding lower bounds on sizes of efficient sorting networks[166].

Construction Related Characteristics

Recursive constructibility — the ability of a graph to be considered the composition of smaller instances of the same graph family — is a common characteristic. These graphs are easy to describe and compose and their scalability allows the programmer to design and test algorithms in-the-small.

Iterative constructibility is a trait of communication structures which are easily catenated. Systolic arrays and algorithms whose structures are dependent linearly on problem size often have meaningful linear extensions in one or more dimensions.

Engineering Characteristics

Uniformity is a characteristic which suggests that the process structure can be composed of a small number of different process types. This is necessary because the individual programming of large numbers of distinct processes would be unwieldy.

Scalability of processes is a trait of all communication structures which will yield massive parallelism. Because behavior of processes cannot be directly sensitive to the size of the process structure, the scaling assumption assumes that processes which work for small systems will work for larger systems.

Routing Properties are characteristic of networks which sort efficiently. For example, if there is a unique path through the network between any two processors, it is often possible to have messages automatically routed, with little forwarding overhead. Many recursively constructed systems have routing properties which are guaranteed through induction motivated by their recursive construction.

Because parallel systems will exhibit many of these characteristics, construction methods that preserve these characteristics will be more efficient in describing massively parallel communication structures.

3.3.2 Irregularity

While many graphs demonstrate regular properties, a number of important algorithms have communication structures with boundaries that make the graph irregular. An example, is the square mesh, whose corners have degree two, edge nodes have degree three and interior nodes have degree four.

Some algorithms, such as the matrix multiplication of S. Y. Kung (Figure 3-13), appear to have irregularities in communication when external I/O is not considered. While external I/O is treated differently in many systems for hardware-specific reasons, the abstraction of communication should unify the concepts of external and internal communication. This treatment of communication makes many communication structures more regular.

Some communication structures exhibit pseudo-irregularities, that is irregularities which do not effect the logic of the algorithm. The Jacobi-iteration problem (Figure 3-14), for example, broadcasts values to adjacent neighbors, ***without consideration of topology***; the obvious irregularities are removed by an abstraction in the code. While structural irregularities do occur in communication structures, they are not necessarily at odds with the regularity of the algorithm.

3.4 The Graph Abstraction

A succinct description of a general algorithm should provide exactly the information necessary to understand how it is to be implemented. Ideally, the designer illustrates his novel use of data structure or flow of control. The designer of parallel algorithms furthers his design and analysis with a diagram which illustrates his use of communication.

Consider Figure 3-15a of Kung and Leiserson[114] which depicts the communication necessary in `band matrix multiplication`. Here, a picture is a more

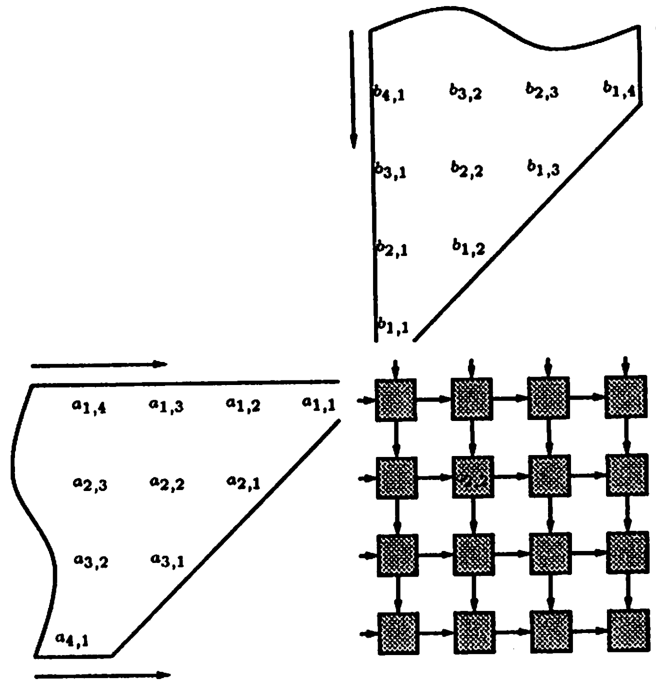


Figure 3–13: Wavefront `matrix multiplication` communication.

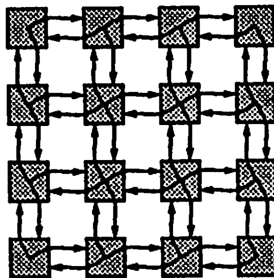


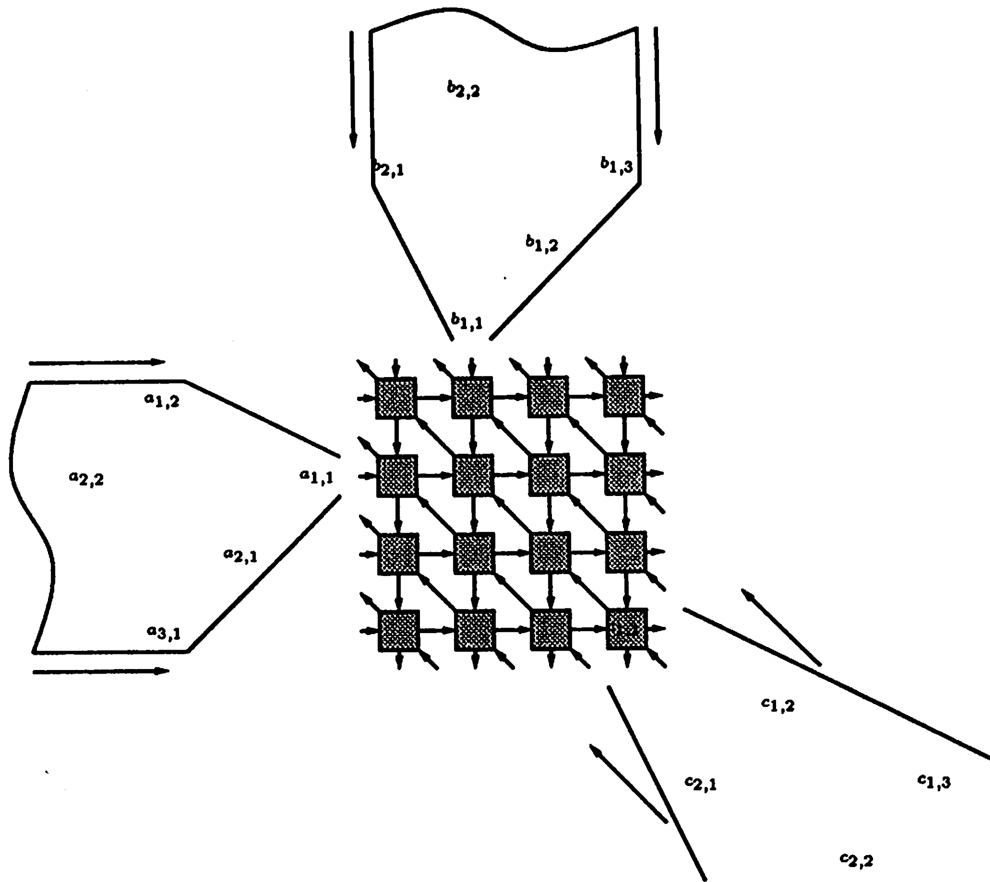
Figure 3–14: `Jacobi iteration` communication.

convenient and abstract method of specifying the algorithm's communication. Two important pieces of information can be gleaned from this type of diagram: the underlying connectivity of the computation, and the paths of data through the system. We leave to Chapter 4 consideration of data paths and instead concentrate on the computation's lattice of communication.

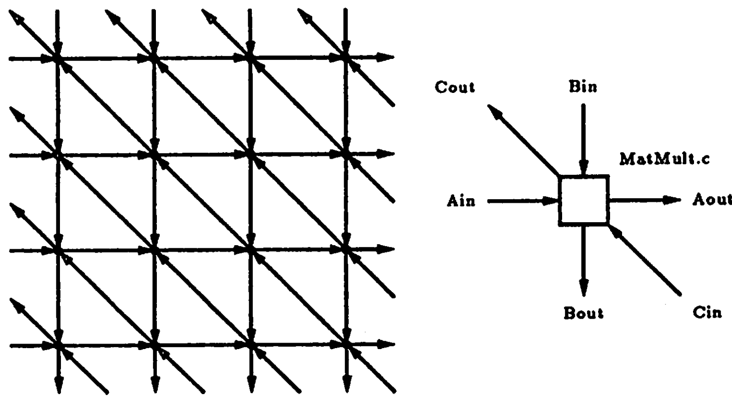
Without losing generality, our illustration may be reduced to a labeled graph. In this graph, each node represents a process and each edge a communication link. The communication structure for matrix multiplication algorithm is depicted in Figure 3–15b. Here, each node is labeled with the code supporting that process' computation.

The representation of communication as a graph in concurrent systems is a standard and straightforward abstraction, as found in the literature. In particular, taxonomies of algorithms, including those by Akl[3], Kung[97], Quinn[131], and Ullman[165], depend heavily on the graph abstraction. Much of our understanding of aspects of communication networks, such as fault tolerance, resource assignment[9,25,24,27,46], and routing problems[166], is derived from an analysis of the underlying graph. Since, as we have suggested in Section 3.1, communication is a structuring mechanism in tightly coupled systems, representation of process interaction by communication graphs is a fundamental aid to understanding the algorithm.

It is particularly important for tools which support the graph abstraction to be sensitive to graph labeling techniques. Since many aspects of the communication structure can be directly interpreted as labels on various components, support for useful labeling techniques is important. We consider, in the remainder of this section, labelings that are commonly used in concise descriptions of tightly coupled algorithms.



(a)



(b)

Figure 3-15: `Band matrix multiplication` and underlying graph.

3.4.1 Labeling of Communication Structures

To this point, we have considered communication structures as abstract graphs. Each annotation, or labeling of a graph provides a particular "view"[2] of the communication graph as a structural abstraction. As the algorithm is elaborated, the labeling of the communication graph becomes more specific. We consider some common labelings of communication structures which are useful at each of the various stages of algorithm implementation.

Process Labeling. The most important annotation is the ***process labeling*** which binds computations to the lattice of communication. Without this labeling, the program specification is necessarily incomplete. Each node of the `band matrix multiplication`, for example, supports an inner-product computation. Process labeling is provided by even the most rudimentary environments discussed in Chapter 2.

Process labelings are often associated with the method of distribution. When functional parallelism is the motivation, decomposition assigns a variety of functions to separate nodes. This leads to nonuniform labeling of nodes, as in the macropipelining of algorithms. On the other hand, when data parallelism is the motivation, processes perform the same logical task on independent data and are more uniformly labeled, as in systolic designs.

Processor Assignment. The problem of embedding the logical program structure in the physical hardware structure is termed the ***processor assignment problem***. The use of appropriate labels on communication structures can lead to direct and efficient mappings[55]. Berman and Snyder[25] use annotations of sentential forms of an edge grammar in this manner, and we consider annotations of sentential forms of a graph grammar in Chapter 5. Fishburn and Finkel use processor labeling as the basis of their quotient mappings[55].

Processor assignment can sometimes be considered independent of the structure of communication. In particular, DeGroot[45] suggests methods for map-

[2] A term borrowed from the Poker programming paradigm.

ping certain classes of regularly structured graphs to banyan-connected multiprocessors, independent of the communication structure specification.

Enumeration. Various parameterizations of processes can be considered labelings of the communication structure. When processes become sensitive to their relative *position* among other processes, algorithm-dependent *enumeration* becomes important. Meshes, for example, are often enumerated in row-major, column-major order or by coordinate pairs. In any case, the enumeration of nodes is strongly related to a method of constructing the network topology.

Various systems provide a standard enumeration which can be used for process parameterization. Poker, for example, has access to the row and column indices.

Data Distribution. When the problem domain can be statically distributed, data parallelism is often the desired mode of problem decomposition. Communication structures for these algorithms are often related to characteristics of the problem domain. Image analysis algorithms might, for example, spatially distribute each frame among the processes. Thus the distribution of data is related to the structure of the computation and labeling of the communication structure parallels the its topology.

Animation Labeling. Once the algorithm has been specified, many environments allow some form of program *animation*[38,76,153,157]. Animation provides a graphical view of the program which can be used as a framework for manipulating and observing the algorithm. Since communication is an important feature of program animation, attributes of communication animation such as the icon needed to depict a processor or its spatial coordinates can be considered attributes of the communication structure. This leads to the *animation labeling*. The needs of abstract program animation are discussed fully by Brown[29]. Animation is also discussed in Chapter 6.

Poker[153] allows the programmer to specify several program views, which are tied to the topology of the algorithm. The Simple Simon environment supports the use

of attributes, which are labels associated with processes, and channels. While some attributes correspond to views of the algorithms (for example, process assignment, channel access, *etc.*), others are user-defined and can be manipulated at run time from within the user's code.

An algorithm's communication structure is fully specified when pertinent labelings have been provided. In the following subsection, we argue that an effective programming environment allows the programmer to specify the needed annotations.

3.4.2 Supporting Flexible Annotations

We have argued that the structure of communication among tightly coupled processes is an independent facet of logical algorithm design. To impose constraints on the structure which are motivated by an underlying architecture distracts the structural specification process. Thus annotation of communication structures should be independent of the underlying architecture.

Even when an algorithm is dissected at an abstract level, flexibility in structure labeling is necessary. Consider, for example, the construction of process labels for banyan networks, as described by two different algorithm designers (Figure 3–16). The first, is from Ullman[166]:

> "In this network, there are $k+1$ ranks, each with 2^k nodes. If we number the ranks from the bottom, starting at 0, and we number the nodes in each rank from the left, starting at 0, then there is a connection between nodes j and l on ranks i and $i+1$, respectively, if and only if either $j = l$ or the binary representations of j and l differ in only the $i + 1^{st}$ place from the left."(p. 5)

DeGroot[44] describes the same graph in the following manner:

> "A uniform one-level SW-banyan is simply a crossbar. A two-level uniform SW-banyan can be constructed as follows. Consider any $m \times n$ crossbar, and select t of them. Choose any integer $k > 0$, and construct another SW-banyan as follows. Take the first (leftmost) apex of each of the t crossbars and connect them to k new nodes. Take the second apex of each crossbar and connect each of these apexes to k other new

nodes. Continue until all apexes have been connected to groups of k new nodes ... This procedure can be recursively applied to construct an SW-banyan with any number of levels."(p. 20)

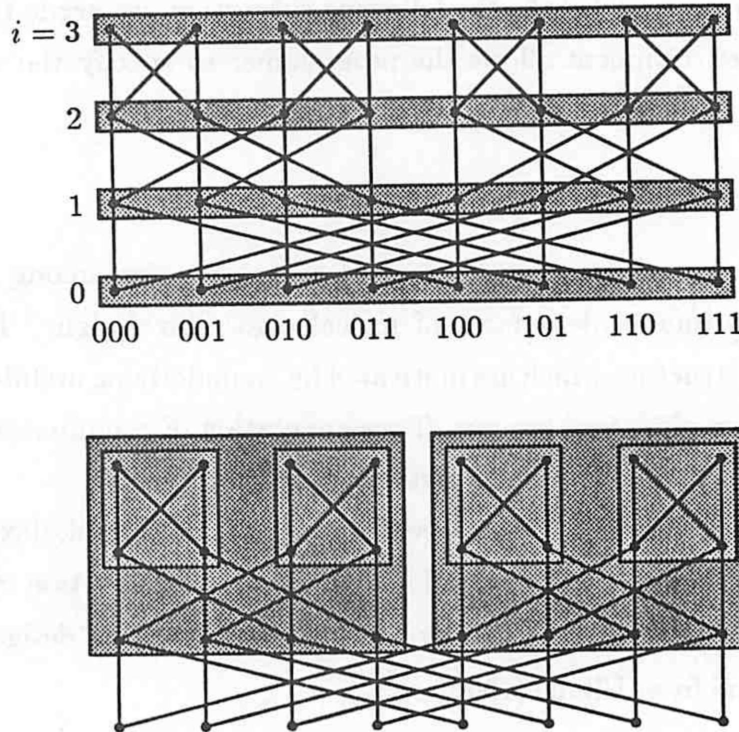


Figure 3–16: Two constructions of banyans: pool-of-processors (top), and recursive (bottom).

Obviously, both descriptions were "natural" to their respective authors. However, the approaches were entirely different. Ullman considers the from-scratch construction of the banyan from a pool-of-processors, while DeGroot's is a recursive description of a family of graphs. Furthermore, Ullman's construction naturally groups all nodes on one rank, while DeGroot's groups all nodes associated with each subbanyan. The design of algorithms for these networks will reflect the differences in views of the communication structures. ***Designers will manipulate the algorithm using their own annotations.***

While the annotations of Section 3.4.1 are common, user-specified labels will be important. The semantics of these labels are user specific, but some general guidelines can be observed.

Process Identity. During specification, debugging and animation, it is often necessary to uniquely identify each process. At some level of abstraction, the set of labels which annotate a particular process distinguish it from all other processes. A context sensitive view of the computation provides an argument: if two processes can not be distinguished, both serve exactly the same purpose, and one could be removed without effect. We shall often assume that communication graphs are uniquely labeled.

Process Equivalence. Aggregation allows the programmer to use labels as a method of abstraction. Liskov[111] states this as follows:

> "The process of abstraction can be seen as an application of many-to-one mapping. It allows us to forget information and consequently to treat things that are different as if they were the same. We do this in the hope of simplifying our analysis by separating attributes that are relevant from those that are not."(p. 3)

The manipulation of large systems requires that many processes be considered as identical at some level of abstraction. For example, in most systolic algorithms a functional abstraction reveals a uniformity of processes. Thus an abstraction induces a view of the communication structure which defines an equivalence based on a restricted view of node labels.

The equivalence class of nodes with a particular label we call an *aggregate*. Aggregates define nodes which are logically similar in some way. In the above example, Ullman considers a "rank" to be the aggregate of all nodes whose i values are the same. DeGroot, however, would aggregate nodes based on the level of recursion, an "i-level banyan".

Later we shall consider transformations on communication structures which are based on labeling. These transformations can be considered refinements of a view the structure: a node is replaced by a set of nodes which simulates

its logic[30]. If two nodes are members of the same aggregate then they are "indistinguishable" and they will be transformed in the same manner.

3.5 Conclusion

The use of communication in tightly coupled systems is unlike that of sequential or distributed processing. It follows that methods of specifying communication that are based on principles of these domains are likely to be unsuccessful at providing the necessary abstractions. We have shown, in this chapter, that communication is used as a structuring technique — so much so that the communication graph has been adopted as a central abstraction for massively parallel communication. In developing support for this abstraction, we must understand two issues: (1) How are communication structures typically derived? and (2) How do they interact with other views of the algorithm?

To address the first issue, we studied four major paradigms for developing algorithms, each of which leads to a different technique for constructing communication graphs. Other paradigms will undoubtedly arise but at a minimum, any specification technique must support these constructions. We further identified a number of characteristics or "regularities" exhibited by structures and we expect that these regularities will provide metrics for evaluating both graph descriptions and proposed specification techniques.

To address the second issue, we demonstrated that it is possible to view the communication graph as a lattice that structures all other aspects of algorithm design. With appropriate tools for annotation or labeling, communication graphs can be used to coordinate the different aspects of specifying massively parallel algorithms. Finally, we discussed common labelings of communication graphs and the support needed for controlled but flexible annotation.

Chapter 4
CANISTER COMMUNICATION

Many environments for programming parallel, ensemble architectures depend on message passing as a communication paradigm. In this chapter, we discuss a new, higher-level paradigm — canister communication — of global patterns composed of multiple point-to-point communications. Canister communication is a simple extension to existing message-based environments. Its use allows increased programming support for parallel computation throughout the software life cycle, including the code specification and debugging processes.

A pivotal component of canister communication is a path which is essentially a restricted view of the communication graph. To the extent that current environments do not support concise specification of communication graphs, they will be ineffective in their support of canister communication.

4.1 Coordinated Message Passing

It is the interprocess flow of data and control that distinguishes parallel from sequential computation. For tightly-coupled, massively parallel algorithms, communication is often structured: streams of related data are pipelined along sequences of channels, values are broadcast to (and aggregated from) sets of processes, and messages are routed through intermediate processes to their intended destinations. In each case, the resulting communications are best understood not as isolated point-to-point message transmissions, but as components of *global patterns of communication.* These patterns are fundamental to our understanding of parallelism.

In systolic applications[67,98], for example, data is pipelined along logical paths in fire-brigade fashion: messages pass from one process to the next across connecting channels. Figure 4–1 shows a systolic `band matrix multiplication`

algorithm with three such paths: the A matrix is carried from left to right, the B matrix is carried from top to bottom, and the resultant C matrix is carried diagonally from the bottom right to top left. An inner product process ($c = c + a \cdot b$) reads A and B values, and modifies C values. Thus, for each path, there exist clear modes of access which must be obeyed. Incidently, the same process, connected with different paths, implements a `lower triangular systems solver` (Figure 4–2). The solution involves four paths: the A matrix enters from the top, the constant vector b enters from the bottom, and x and y vectors travel horizontally in opposite directions. Finally, Figure 4–3 shows a systolic `transitive closure` algorithm. The transitive closure of matrix A is computed in a processor array of the same shape. Two bands of intermediate results cycle through the array, one rotating horizontally, and the other vertically.

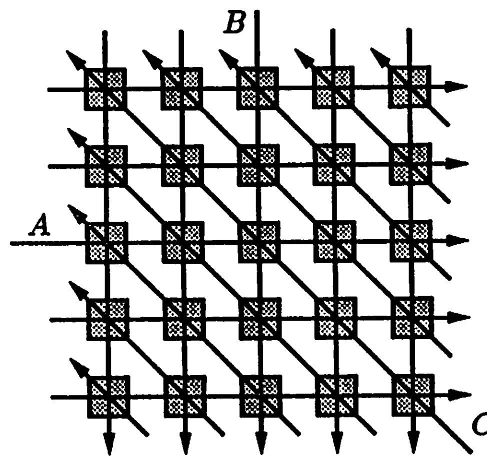


Figure 4–1: Paths of communication in `band matrix multiplication`.

Nonsystolic algorithms also require logical paths. Parallel prefix algorithms for MIMD architectures accumulate results through several iterations, as is demonstrated by Figure 4–4. Data values are accumulated through communications structured as a complete binary tree, but actual communication depends on the position of the process in the processor array and the progress of the computation. In a `Jacobi iteration` for solving PDE's[131], processes repeatedly broadcast their

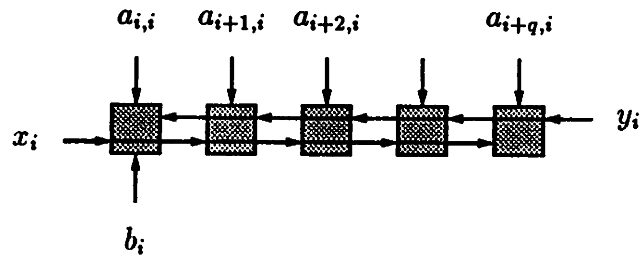


Figure 4–2: Paths for solution of lower triangular systems.

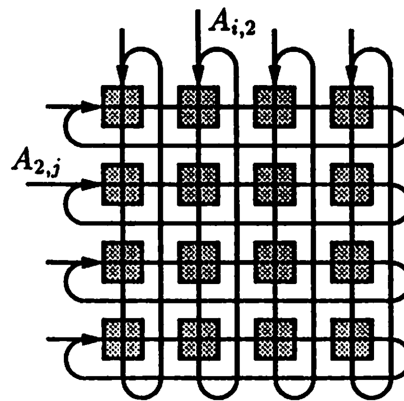


Figure 4–3: Cyclic data flow in a transitive closure algorithm.

current value to neighboring processes and then average the values they receive. Each message is sent on an export path (a) and received on an import path (b), as shown in Figure 4–5. Notice that the logic of the algorithm is independent of the specifics of communication (such as degree of fan-in or fan-out). An obvious attempt to support messages which persist across several communication channels is found in `palindrome`[54] (Figure 4–6). Each process p_i processes a sequence of digits in a large palindrome. Each iteration requires a summation which causes a carry flag to ripple through the network.

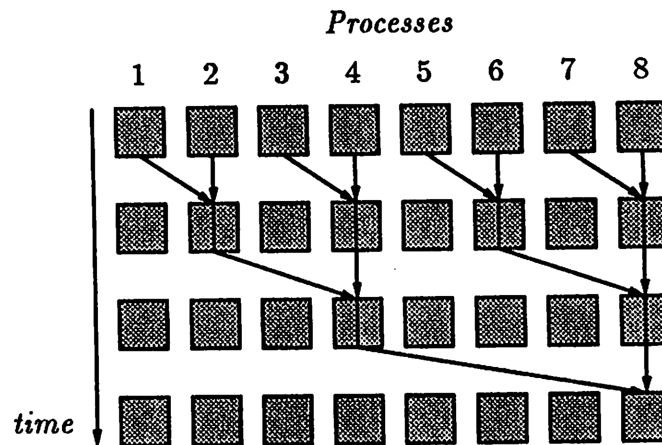


Figure 4–4: The tree-shaped paths of prefix-type operations.

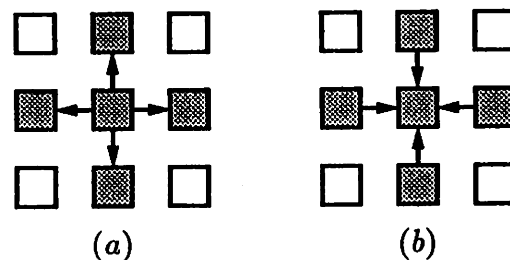


Figure 4–5: The itineraries for Jacobi's PDE approximation technique.

Logical communication patterns can also serve as virtual circuits in which a message traverses successive channels in a sparse network to arrive at a logically

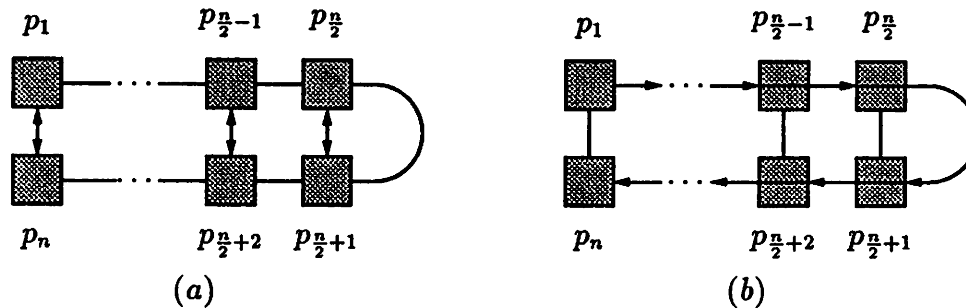
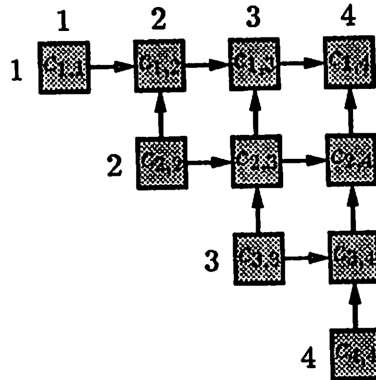


Figure 4-6: Communication needed to generate and test palindromes.

adjacent process. This may be needed because of limitations in the connectivity of the hardware or it may be inherent in the algorithm itself. Examples of routing induced by hardware constraints include diagonal shift of the data in a rectangular mesh. Figure 4–7 depicts communication in an LU-decomposition, including paths that transmit a pivot value (a), generate L-values (b), and generate U-values (c). Successive reductions of the problem require logically shifting the matrix diagonally (d) which, on a square mesh, corresponds to physically shifting in two steps as in (d'). The support of `bitonic sorting` and `fast Fourier transform` in square meshes[163] requires statically determined multihop communication patterns differing at each stage of the computation (Figure 4–8). Even in systems that provide automatic routing, the programmer may want to code it directly for efficiency[73]. Examples of algorithms with inherent routing include `dynamic programming` and `lower triangular systems solver`. `Dynamic programming` uses routing as a queuing mechanism. It is implemented on an upper triangular array,



with intermediate results moving upward and rightward, generating the solution in the upper right corner. Each process pairs incoming intermediate values to compute its own cost estimation. Figure 4–9 depicts the three pairings of values needed at process [1,4]. Correct ordering of messages is important to the correctness of the pairing — note, for example, the multiplexing of the physical channel between processes [1,3] and [1,4]. Algorithms, such as `lower triangular systems solver` (Figure 4–10), require complex routing to correctly interface each of several logical stages of the algorithm. Paths result from the composition of `LU decomposition` (a), two lower triangular system solvers (b), and a stack of processors for intermediate matrix values (c).

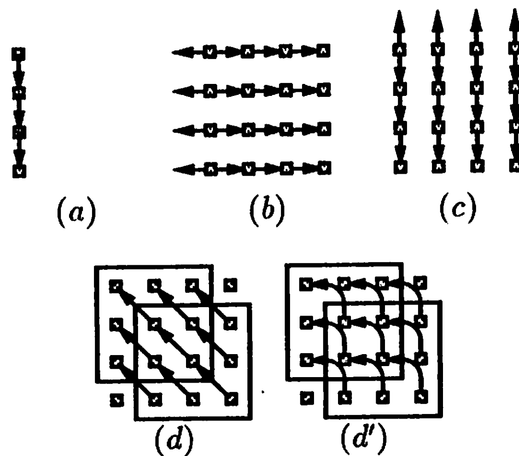


Figure 4–7: A Wavefront Array Processor implementation of *LU* decomposition.

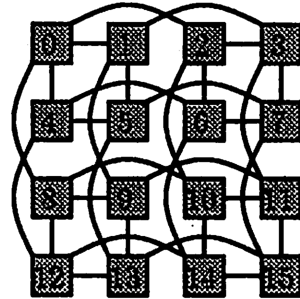


Figure 4–8: Mesh support for butterfly paths.

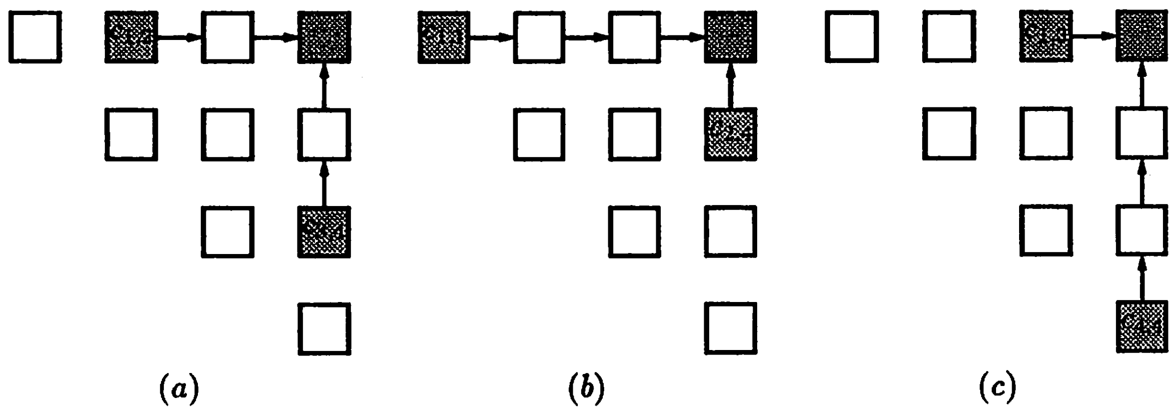


Figure 4–9: Three pairings which must be considered in the problem solved at processor [1, 4].

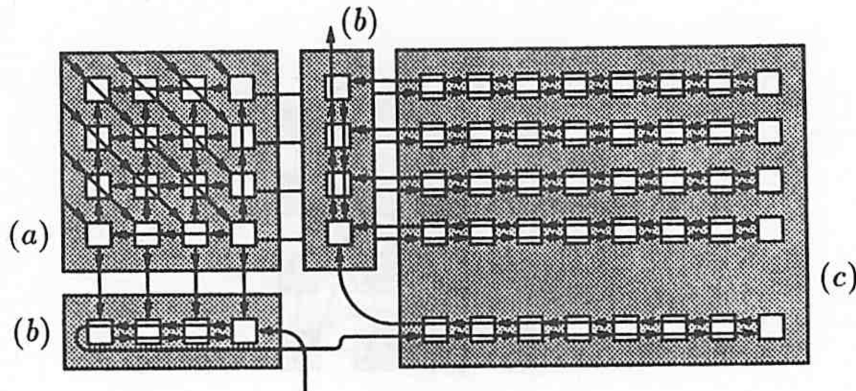


Figure 4–10: Paths supporting the solution of a system of system of linear equations.

In each of these examples, global communication patterns are fundamental to our understanding of the algorithm. How are these patterns specified by the programmer?

The abstractions provided by a programming environment determine the programmer's effectiveness in implementing and debugging algorithms. Few abstractions, however, support the specific needs of tightly-coupled, massively parallel computation. In particular, most parallel programming languages offer only modest extensions to sequential languages, supporting point-to-point interprocess communication. Thus, the programmer cannot directly describe the important global characteristics of his interprocess communication.

In this chapter, we introduce a new programming abstraction, called canister communication *that enables programmers to specify global communication patterns directly, narrowing the conceptual gap between the design of an algorithm and its implementation.* Canister communication is a straightforward extension to existing message-passing environments for tightly coupled computation. It allows the programmer to specify three pieces of information which aid in avoiding common communication errors in message passing systems: (1) the path of the data through the processor array, (2) the local access rights of each process along the path, and (3) the global type of the data. By separating communication details from the

logic of the algorithm, canister communication often leads to concise specification of process code and communication-specific error detection.

4.2 Canister Communication

The essential concept in canister communication is that messages are carried in a logical container, called a *canister*.[1] The canister is created, filled, transmitted (possibly many times), emptied, and destroyed in a manner which must be consistent with a pattern of point-to-point transmissions, called an *itinerary*. Canister communication provides a more abstract "layer" of support for the coordination of point-to-point passing mechanisms. Each program written using canisters is equivalent to a program utilizing standard message passing. In fact, the worst case of canister communication is equivalent to straightforward message passing: each point-to-point communication is itself an itinerary.

We first discuss a mechanism for specifying paths of data.

4.2.1 The Itinerary: Encoding Message Behavior

An *itinerary* is a path through a process array, together with the type of data that it is to carry, and the permissible accesses to that data. It is represented as a node-labeled directed graph in which nodes represent processes and edges represent communications. Processes may appear in more than one place in an itinerary and they may have more than one "local" name for that itinerary. As a result, we label each node with a `(process-name,itinerary-alias)` pair. In addition, we label each node with a set of access rights that determine which canister operations — create (*crt*), destroy (*dst*), read (*rd*), write (*wrt*), and view (*view*) — are permissible.

An itinerary is defined as follows:

1. A domain M from which messages are selected. This set codes the *type* of the

[1]The term *canister* is motivated by the pneumatic canisters used to perform a two step transaction in old banks and stores. Like the incorrect passing of values in a tightly coupled system, errors in the monetary transaction were not well-received.

itinerary. A unique null message, $\perp$, is not in any domain and is the message returned by a non-blocking read when no message is available on a channel.

2. A directed graph where each node represents a process and where each edge connects processes which are adjacent through physical communication channels (processes may be self-adjacent).

3. Each node is uniquely labeled with a `(process-name,itinerary-alias)` pair.

4. For each node with outdegree $d > 0$, an associated broadcast function

$$broadcast^d : m \mapsto (\overbrace{m, ..., m}^{d \text{ times}})$$

which generates a copy of the message for each exiting edge. Note that nodes with no fan-out are labeled with the identity function $I = broadcast^1$.

5. For each node with in-degree $d > 0$, an associated merge function

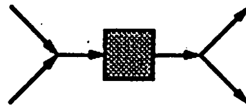
$$merge^d : (M \cup \{\perp\})^d - \{\perp\}^d \rightarrow M$$

which combines available messages from inbound channels. Depending on the function, the merge blocks until one or all of the channels have input available. A class of merge functions is "consistent" if it reduces to the identity function when $d = 1$.

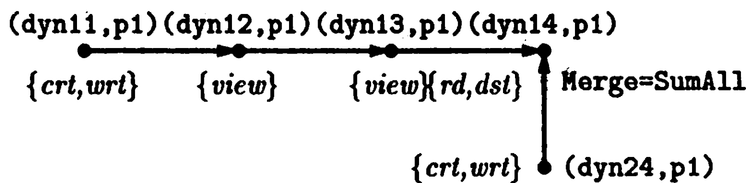
6. For each node, a subset of the access rights $\{read, write, view, create, delete\}$ to canisters and the messages they contain. *create* and *delete* respectively support sources and sinks for canisters in the itinerary, while *read*, *write* and *view* respectively provide destructive read, write, and nondestructive read access to messages in a canister.

The simplest itineraries are lines which enter and exit distinct processes at most once:

If an itinerary enters (exits) a node in the graph more than once, multiple inputs (outputs) are assumed to be indistinguishable to the process:



Where this results in fan-in, arriving values are combined with a merge function; where it results in fan-out, copies of the departing value are broadcast along each edge. A number of merge functions are possible, such as `SelectAny` (select one message from a random channel), `SelectFirst` (select messages in order of arrival), and `SumAll` (add an incoming value from each channel). Each of these is a *class* of functions semantically independent of the details of the communication; thus, for example, `SelectAny` has a consistent interpretation for all degrees of fan-in (degenerating to a nonblocking read in the trivial case). `Dynamic Programming`, uses a number of paths with fan-in to accomplish the pairings of values; the costs for $c_{1,1}$ and $c_{2,4}$, for example, are merged along path `p1`:



Fan-in may also be used to produce cycles. An itinerary for the `transitive closure`, for example, could be given as the cycle in Figure 4–11a using the `SelectAny` merge function. Alternatively, this itinerary could have been specified as an unrolled line in which each process appears as a node label exactly three times as in Figure 4–11b. To distinguish different points on the itinerary from within a process, the user defines local aliases for the itinerary and thus in the example above each process has three names for the itinerary, corresponding to the three times it reads from the itinerary. (Arrays of alias names are possible.)

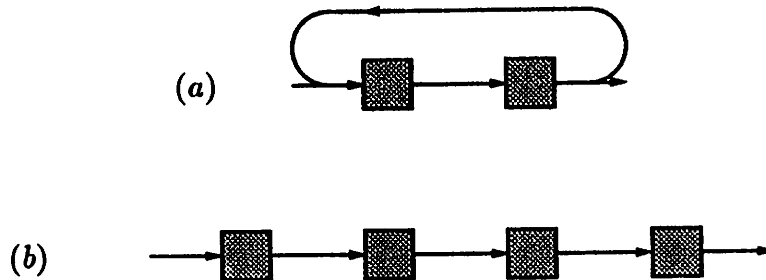


Figure 4–11: Specification of unbounded cycling (*a*), and fixed cycling (*b*).

More complicated itineraries are possible. The `jacobi iteration` (Figure 4–5) and `dynamic programming` (Figure 4–9), for example, use overlapping itineraries; while `palindrome generation` (Figure 4–6), `LU decomposition` (Figure 4–7), the `fast Fourier transform` (4–8), and the `linear systems solver` (Figure 4–10) all utilize compositions of paths.

We next discuss canisters.

4.3 The Canister: A Message Wrapper

The *canister* provides a reusable logical wrapper (as illustrated in Figure 4–12) for messages whose semantics must be preserved across multiple transmissions. Conceptually, it is transmitted intact from one process to another allowing its internal state (including message validity, message value, associated itinerary and other canister attributes) to persist. Furthermore, canister manipulation is restricted to a small number of access functions, and thus the state of the canister is preserved between transmissions.

A canister is created in association with an itinerary, carries messages from point-to-point, and is eventually emptied and destroyed. Intermediate processes may read, modify or view the contents of the canister in accordance with the behavior mandated by its itinerary. The allowable ordering of accesses to canisters is depicted in Figure 4–13.

All the `matrix multiplication` itineraries of Figure 4–1, for example, allow

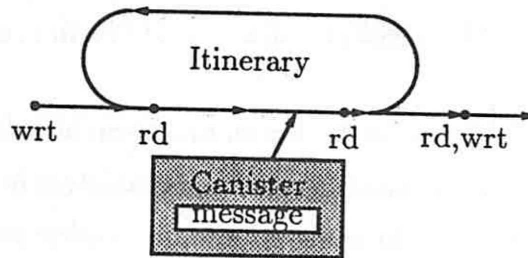


Figure 4–12: The canister is a wrapper which carries a user message. The message is constrained to accesses provided in the associated itinerary.

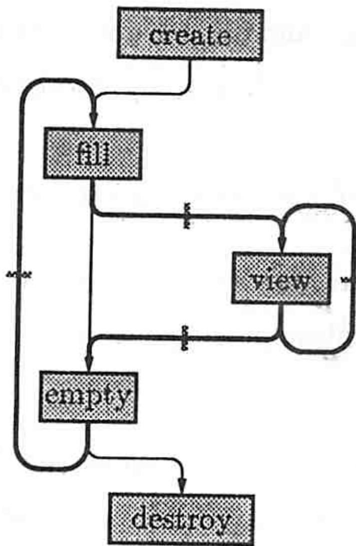


Figure 4–13: The possible access orders of canisters. Cuts indicate points where logical message transmissions must occur.

access to the values at each supporting process; in Figure 4–7, however, the diagonal routing processes are not allowed access to canisters they forward. The use of canisters guarantees that values will not be accessed in a manner that is inconsistent with the desired behavior.

The interest in providing canister communication as a *layer* of abstraction on an existing message passing mechanism constrains canisters in certain obvious ways. For example, each transmission in canister communication must be nontrivial: the transmission of an empty canister indicates a non-message which artificially breaks the continuity of data flow.

Broadcasting and merging of messages is often a function of the process' location in structure of the computation, rather than some inherent characteristic of the process code. Canisters hide the tailoring of communication on boundary processes because the details of broadcasts and merges are logically removed from the process code. The result is that processes that perform logically identical operations are coded identically. In its support of message passing, canister communication makes source code more uniform, and often reduces the number of distinct process codes necessary to support an algorithm — *e.g.* the tree merge (as will be seen in Example 2 below).

The following seven logical operations on canisters are sufficient.

- *Create(canister-ptr,itinerary).* Creates an empty canister associated with a specific itinerary. The creating process must have `create` access at that point in the itinerary.

- *Destroy(canister-ptr).* Destroys empty canisters. The canister pointer must be defined, and the associated itinerary must provide `destroy` access.

- *Put(canister-ptr).* Writes a *copy* of the canister to all exiting channels which branch at this point in the associated itinerary. Each channel receives a *logically indistinguishable* copy of the message in a separate canister.

- *Get(canister-ptr,itinerary[,merge-function]).* Reads the next canister available on the specified itinerary. Where there is fan-in, the canister is the result of

the itinerary's merge function applied to the set of next available canisters on incoming channels. No explicit access is necessary.

- *Fill(canister,message)*. Fills an empty canister. The process must have `write` access to the canister and the type of the message must match that required by the itinerary.

- *Empty(canister,message-ptr)*. For a full canister, retrieves the message in the canister and changes its contents state to `empty`. The process must have `read` access to the canister and the message type must match that required by the itinerary.

- *View(canister,message-ptr)*. Retrieves a *copy* of the message in the canister. The process must have `view` or both `read` and `write` access to the canister. The message type must match that required by the itinerary.

Itineraries encode global information that is a redundant specification of local canister operations. Thus the abstraction provides information potentially useful in error detection and debugging, as we shall see later. First, however, we provide some illustrative examples of the use of canisters as implemented in the Simple Simon programming testbed.

4.4 Examples of Canister Communication

We have implemented canister communication as part of the Simple Simon[40] programming testbed, an environment which supports programming of the Simon multicomputer simulator[70]. Central to the Simple Simon programming methodology is the concept of a program database[153] which enables different utilities to access and modify different logical views of the program. This database approach to specification is particularly well suited for supporting the description of itineraries. A textual database language (TDL) allows the programmer to describe and annotate the communication structure associated with any particular algorithm.

Itineraries are supported by TDL's `path` directive which establishes the relation between channel connections and paths, as well as access rights and local

aliasing. The programmer writes generic C language code bodies which are automatically tailored for use at different points in the process structure. Canisters are implemented as a library which augments the standard Simon message passing mechanisms. This implementation was intended as a prototype; the interest is not in questions of elegance or syntax but in providing a testbed for experimenting with the concept of canisters. In particular, a graphical interface to the database (which is currently under development) will provide more natural views of communication structures and itineraries. The design of this editor is considered in Chapter 6.

We now present a number of algorithms which illustrate the utility of canister communication. (These algorithms are not presented in their entirety.)

Example 1. The pipelined nature of the communication in `band matrix multiplication` causes canister-based and the traditional message-based code to be very similar. Because, however, itineraries for A, B, and C values have been specified, mismatched communications (for example, a message sent as an A value which was read as a B value) would be detected in the canister-based implementation. The following code is used by all processes.

```
matmult() {
    /* declare canisters */
    CanTypeDecl(double,ACan);
    CanTypeDecl(double,BCan);
    CanTypeDecl(double,CCan);
     ...
    for (...) {
        CanPut(CanView(CanGet(ACan,APath),a));
        CanPut(CanView(CanGet(BCan,BPath),b));
        CanEmpty(CanGet(CCan,CPath),c);
        CanPut(CanFill(CCan,c + a*b));
    }
}
```

Example 2. A systolic tree summation algorithm performs parallel summation of vectors which are piped in from the leaves of the tree. Sums appear at the root. Since each process sends the sum of its subtree to its parent for inclusion in the sum

at the next level, a binary tree of communication channels is the itinerary necessary to support the algorithm. The fact that the itinerary has the same topology as the underlying communication structure is a coincidence imposed by the simplicity of the algorithm. Even so, the information that "all channels support the same logical operation" is information that cannot be gleaned from lower level abstractions of communication.

Code for the processes in the tree summation appears as:

```
int sumfun(a,b) { return a+b; }

tree() {
    CanTypeDecl(int,SumCan);

    CanGet(SumCan,TreePath,sumfun);
    CanPut(SumCan);
}
```

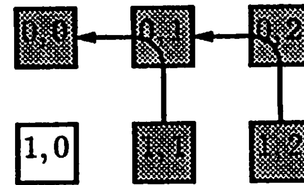
Since the code for the leaves and root perform logically different functions, they may require similar code bodies which create and destroy the canisters for the itinerary. If, however, the external I/O interface supports canister communication, it can perform this task, and all of the code can be specified with this single code body. Canisters thus allow the user to ignore irrelevant specifics of communication and focuses on the logic of merging the values and passing them up the tree. This same code would, in fact, work for unbalanced trees and for trees with nodes of varying degree.

Example 3. Many algorithms for the Wavefront Array Processor can be directly converted to run in the Simple Simon environment. Simon code bodies correspond to instances of Kung's ***local view*** of the WAP implementation. The monolithic code for the WAP processor considers computation in light of global communication ***wavefronts***, a communication abstraction not supported in other programming environments. However, in `LU-decomposition` describing communication patterns using itineraries makes the multiplexing of channels and the routing during the reduction process more straightforward. Implementing the paths depicted in Figure 4-7 significantly increases the chances of detecting communication errors in

these algorithms.

The `Shift` path illustrates the itinerary required to support the of the reduction phase of the WAP `LU-decomposition` algorithm.

```
        path Shift {
          type double;
          connection lud11 {access create, write;}
(1)                   to lud01 {alias Pass;};
          connection lud01 {alias Pass;}
                      to lud00 {access read, destroy;};
          connection lud12 {access create, write;}
                      to lud02 {alias Pass;};
          connection lud02 {alias Pass;}
(2)                   to lud01 {access read, destroy;};
            . . .
        };
```



Here, process `lud01` participates in the path as an intermediary under the alias `Pass` (1) and as a recipient the `Shift` path identifier (2). The process accesses these points of the itinerary with logically distinct operations.

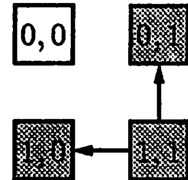
```
lud()
{
    CanTypeDecl(double,SCan);
     . . .
    /* Shift the matrix: */
    CanPut(CanGet(SCan,Pass));
    CanDestroy(CanEmpty(CanGet(SCan,Shift),a));
     . . .
}
```

A canister created and sent from process `lud12` along the `shift` itinerary is forwarded by `lud02` using the pass view and received by `lud01` as part of the `shift`. The alias encodes local logic while the itinerary encodes global flow.

Example 4. An iterative relaxation technique, such as the Jacobi approximation technique for solving PDE's, requires an itinerary for access to local neighborhoods. Each process first broadcasts a value to its local neighborhood and then averages

values received from that neighborhood. These are logically different functions, as is indicated by the path:

```
path Export {
    . . .
  connection Jacobi11 {access create, write;}
          to Jacobi10 {access read, destroy;
                       alias Import; };
  connection Jacobi11 {access create, write;}
          to Jacobi01 {access read, destroy;
                       alias Import; };
     . . .
};
```



Each process is a source for the Export path which has fan-out for broadcasting and is a sink for the Import path which has fan-in for combining values. Code for the approximation process consists of a loop based on:

```
CanPut(CanFill(CanCreate(jcan,Export),localvalue));
CanDestroy(CanEmpty(CanGet(jcan,Import,sumfun),sumvalue));
localvalue = sumvalue/FanIn(Import);
```

The sumfun function is similar to that of the tree summation example (canisters are merged pairwise). The sum is normalized with the knowledge of the fan-in of the Import itinerary. Similar code without canister communication performs logical broadcasts and merges using multiple puts and gets which makes the code confusing. Usually, such approximation techniques use rectangular communication structures. With canister communication, more complex topologies can be used to define the local neighborhoods without impacting the logic of the algorithm.

4.5 The Use of Canister Communication in Debugging

Parallel programs are difficult to debug both because of their inherent complexity and because they are not amenable to sequential debugging techniques. Appropriate treatment of communication can make debugging simpler. In particular, we have found that canister communication can help in at least two ways: it indicates the intended patterns of message traffic that are otherwise unknown, and it provides a vehicle for attaching special purpose tags to messages. We briefly describe these uses.

4.5.1 Recognizing Patterns in Communication

Pattern-oriented debuggers, such as those of Bates[18] and Hough[76], scan event-based trace files for communication patterns[17] that the programmer expects to find. Once a pattern is found, it can be displayed or used as a component of larger patterns. Canisters provide relationships between messages which can be used to build patterns.

Consider, for example, the `jacobi iteration` problem. Using a pattern-oriented debugger, it is possible to define the patterns for the correct association of broadcast values[75], but the definitions are quite complex. Using canister communication, the necessary pattern (the all-neighbors broadcast) would already be available as an itinerary. Thus canisters provide correlations between messages that can easily be exploited during programming as well as debugging; they provide a more abstract set of events from which the user can build further patterns to investigate.

4.5.2 Radioactive Tagging

Communication errors in parallel programs often result in computations on the wrong values. One possible method for detecting such errors is to *radioactively tag* values in order to trace their influence. In this scenario data values would be initially tagged. As execution proceeds, tagged values would *contaminate* any computations in which they were used and intermediate values would inherit the

tags of the values used to produce them. Suspicious results could then be inspected for appropriate tags.

Radioactive tagging would provide enormous amounts of information to the debugger, but would be prohibitively expensive to implement. Canisters provide the basis for an efficient approximation: the canister *itself* conveys tags which accumulate in participating processes. Both processes and canisters are labeled by sets of tags which obey the following rules:

1. Each canister is tagged at creation by the user, perhaps based on initial contents.
2. Each canister read or created at a process appends its tag set to the tag set of the process.
3. Each canister sent from a process is retagged with the tag set of the transmitting process.
4. Merged canisters receive the union of tag sets, while duplicated canisters receive copies of tag sets.

These rules enable us to make use of radioactive tagging in the dynamic programming example above. If values on the diagonal are sent in tagged canisters, then every process should show contamination from all of the processes in the triangular quadrant it dominates. Figure 4–14 depicts all processes which depend on the cost computed at $[3,3]$. If because of a timing error, a computed cost fails to arrive at a dependent process in time to be included in the pairing minimization, the associated tag will not be included in the tag list of the final computed minimum. A similar technique would enable us to determine, for example, whether all processes contribute in a `tree summation`, or whether appropriate terms are being computed in a `band matrix multiplication` or whether the generation of L-values is proceeding correctly in an `LU decomposition`.

4.6 Conclusion

We have introduced a communication abstraction based on the behaviors of message structures which persist along multihop paths. With this mechanism

in hand, programmers are able to describe an algorithm more lucidly, separating the logic of the algorithm from the details specific to communication. Canister communication also provides debugging abstractions which can be used directly by a pattern-oriented parallel debugger.

We have implemented canister communication in the Simple Simon environment and will be extending the implementation to support use by the Belvedere parallel debugger[75]. Our experience to date has suggested a number of extensions to the canister communication. Firstly, because messages may simultaneously support more than one purpose, the definition of itineraries is naturally hierarchical. Secondly, when communication structures are composed to support more complex computations, the ability to compose of itineraries is important. a pipeline, necessitates the composition of itineraries. Thirdly, while we have discussed nonshared implementations of canisters, previous work has suggested that message passing can be a useful abstraction in shared memory environments[112]. Where that is the case, canister communication may also be a useful extension. Finally, since our communication patterns are limited to those that are determinable at compile time, we are investigating methods for supporting dynamically initiated, statically structured communication patterns.

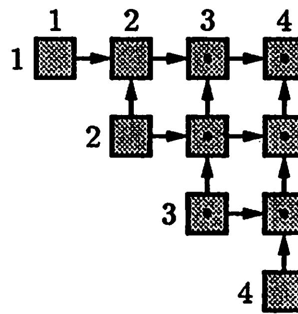


Figure 4–14: Contamination in **dynamic programming**: all processes which dominate $[3,3]$ become tagged by its radioactive cost message.

Chapter 5
GRAPH GRAMMARS

Graph grammars are a formalism that aid in the construction, analysis and parsing of graphs. Just as string grammars are a formalism for meaningful transformations on streams of string data, graph grammars provide powerful descriptions of the manipulation of graphs. Typical domains for graph grammars are the description of spatial relations in biological systems, associations between database relations, and transformations on abstract data types. Unfortunately, the graph grammars used in these domains[1,34,51,82,83,118,128] can not generate graphs with the regularity often found in highly parallel communication structures. Aggregate rewriting graph grammars can express this by transforming aggregates of similarly labeled nodes in precisely the same manner. In addition, with aggregate rewriting graph grammars we investigate a new "partitioning" mechanism which provides the ability to duplicate graphs of arbitrary topology and size.

5.1 Graph Rewriting Systems

Traditional string grammars are characterized by a single point of activity: a single substring of a sentential form is removed, transformed, and reintroduced. *L-systems*, or *Lindenmayer systems* are a parallel version of string grammars in which a production can be applied simultaneously in more than one context to effect the transformation. These systems are important because they model biological systems which are composed of many independent points of growth. A production application simulates a quantum of growth that occurs throughout the system in parallel. In general, *L-systems* are characterized by their *parallel rewriting* of sentential forms.

Strings are less satisfactory than graphs in representing structured systems or systems where interaction occurs between more than two neighbors. A cellular

system, for example, is more accurately simulated when considered as a multidimensional system. A cell mass might be represented as in Figure 5–1; each node of the graph represents cell and each edge a cell-cell interface. The growth process could then described using a *graph grammar*.

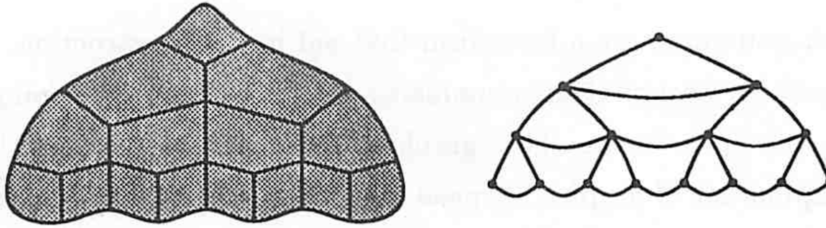


Figure 5–1: A cell mass and its representative graph.

The transformations of graph grammars, like string grammars, may be accomplished sequentially or in parallel. In either case, the concepts of graph rewriting are much the same as those of string rewriting. A *start graph* is transformed by a series of *productions* which generate the *sentential forms* of a *derivation*. The *language* of a graph grammar is the set of terminally labeled sentential forms. The biological motivation for each of these systems brings to light an important concept. Because each graph transformation is a complete biological transformation (*e.g.* each cell splits), *each sentential form corresponds to a meaningful configuration*. Likewise, if each transformation of a process graph is meaningful (*e.g.* conquer nodes divide), each sentential form describes a possible process structure. Our motivation in this chapter is to apply graph grammars more directly to the problem of describing communication structures.

Considerable efforts have been made to provide general mechanisms for the rewriting of graphs[138]. Some systems rewrite portions of graphs sequentially, while others rewrite the entire graph in parallel. In this chapter, we provide a new hybrid formalism which is motivated by the following goals:

- *Massive, uniform changes should be accomplished succinctly*. The uniformity of processes and the scaling nature of many communication structures requires

us to consider systems which grow quite quickly (*e.g.* cubes, shuffle graphs, De Bruijn graphs). The difficulty in manipulating graphs should not be proportional to the graph size. Our formalism is unique in that it utilizes parallelism of production application to aid in communication structure manipulation.

- *Transformations on graphs should be independent of context*, if possible. The programming of many parallel systems occurs in a localized context, and as a result, neighboring processes are often not considered. Our graph grammar formalism also can be made free of node context.

- *The grammar mechanism should support familiar transformations easily*. In constructing large systems engineers are fond of saying "begin by constructing two smaller networks ... ," or " ... tied together with crossbars ... ." These operations often can be expressed as primitive transformations in the underlying formalism.

In the next section, we present a graph grammar formalism for succinctly describing massively parallel communication structures.

5.2 Aggregate Rewriting Graph Grammars

In this section we introduce *aggregate rewriting (AR) graph grammars*. An *aggregate* is a set of logically related nodes in a graph; for example (see Figure 5–2), the leaves of a binary tree form an aggregate, as does its left subtree and a path from its root to a leaf. AR graph grammars rewrite entire aggregates in a single step (see Figure 5–3). Each rewriting rule is extrapolated from a production that transforms a small, fixed size subgraph; thus arbitrarily large aggregates can be manipulated, allowing concise descriptions of recursively definable graph families. This is a characteristic that distinguishes AR graph grammars from other formalisms previously forwarded.

Informally, a production of an aggregate rewriting graph grammar removes an aggregate from a *host graph*, transforms it, and reinserts the result by regenerating edges that provide the interface. Derivations in these grammars are similar to those

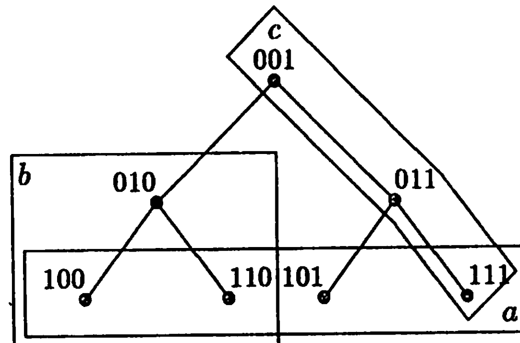


Figure 5–2: Graph aggregates. A complete binary tree with integer-labeled nodes suggests several aggregates: (a) the set of leaves, (b) the left subtree, and (c) a path from root to leaf. Each can be described as the set of nodes whose labels are related in some specific way.

found in both sequential and parallel graph rewriting systems: productions are parallel in their rewriting of nodes and sequential in their rewriting of aggregates.

This dual nature is especially useful in the manipulation of large communication graphs. The programmer usually views the programming of processes from an extremely localized context: "exchange this data with a neighboring process" or "compute and pass the result on for accumulation." He thinks about his communication structures in the same localized way because massive and complex changes to a network are not easily imagined. AR graph grammars provide a natural context to describe iterative modification of communication structures.

We now provide a more formal definition.

5.2.1 The Formalism

We consider aggregate rewriting graph grammars in the context of undirected graphs without self-loops or multiple edges, although the definitions may be extended to include these special graphs. A ***graph*** is a 4-tuple $G = (V_G, E_G, L_G, \gamma_G)$, in which V_G is a finite set of nodes, E_G is a set of two element sets of V_G representing an edge set, and L_G is a set of node labels identified with nodes by a total labeling function $\gamma_G : V_G \rightarrow L_G$. Graphs G and H are ***isomorphic*** if there is a

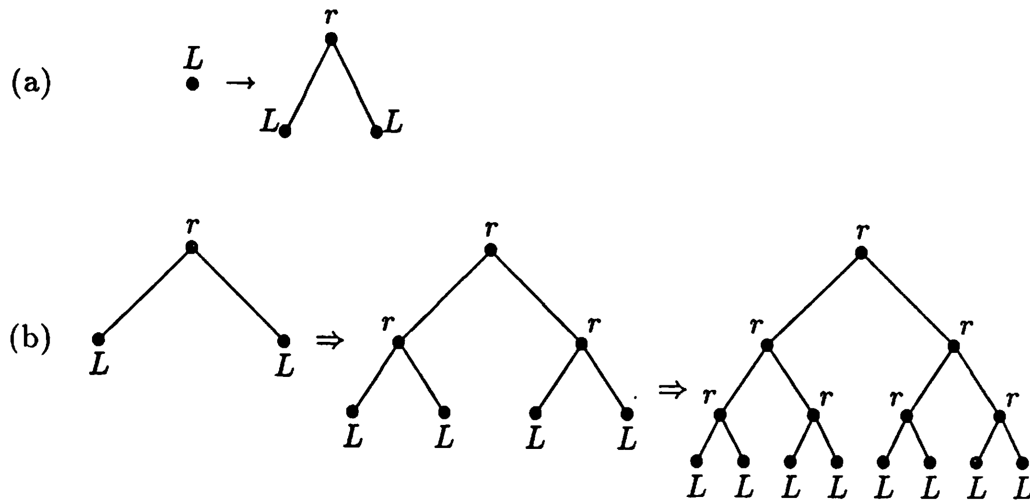


Figure 5–3: Aggregate manipulation. A production (a), which manipulates a single leaf of a binary tree describes the manipulation of all leaves. This production is used to grow successively larger, complete trees (b).

bijection $\iota : V_G \rightarrow V_H$ which induces the natural bijection between E_G and E_H. An *occurrence of G in H* is a subgraph G' of H which is isomorphic to G. While it is common to require that the isomorphism between G and G' be label-preserving, as we do here, it is not necessary. It is also possible to constrain the occurrence by other predicates as we discuss in a later section.

An *aggregate* of graph G in graph H is a graph consisting of the union of the occurrences of G in H (see Figures 5–4a–b). Since the aggregate consists of *all* occurrences of G in H, it is uniquely determined by graphs G and H. It is often useful to index the occurrences from a set $\mathcal{J} = \{1, 2, \ldots, n\}$ where $n = |\mathcal{J}|$. A *aggregate rewriting production* $\mathcal{P} = (M, D, \phi, \pi)$ rewrites occurrences of a *mother graph*, M, to copies of a *daughter graph*, D, under the direction of an *inheritance function* $\phi : V_D \rightarrow V_M$ and a partitioning function $\pi : \{1, \ldots, k\} \rightarrow 2^{\mathrm{dom}(\phi)}$, $k \geq 1$. The inheritance function ϕ is a partial, surjective function that indicates, for some nodes of the daughter graph, a node in the mother graph that will provide interface edges. The partitioning function defines a disjoint k-partitioning of the domain

of ϕ, such that for each $1 \leq i \leq k$, every bijective restriction of $\phi_{\pi(i)}$ is a graph isomorphism from D to M (*ie.* M occurs in each partition of D, perhaps several times). A production is applied to a ***host graph*** yielding a ***image graph***.

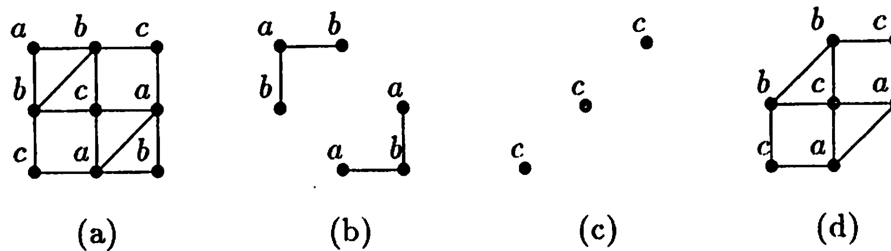


Figure 5–4: (a) A host graph with $a - b$ occurrences, (b) an aggregate of $a - b$ graphs, (c) the rest graph and (d) the interface. As is the case here, it is often useful to have graph occurrences overlap.

We now describe the mechanics of the parallel rewrite rule. All occurrences of the mother graph are removed from the host graph — yielding the ***rest graph*** (see Figure 5-4). The ***interface*** is defined by those edges which either are incident to both the rest graph and an occurrence of the mother graph, or are incident to two distinct occurrences of the mother graph. For each occurrence of the mother graph found in the host graph, a daughter graph is disjointly added to the rest graph. Each edge of the interface is rewritten using one of the following rules:

Rule 1 If the edge $\{u, v\}$ is incident to the rest graph at u and an occurrence of the mother graph at v, an edge is introduced between u and all instances of nodes $v' \in V_D$ for which $\phi(v') = v$.

Rule 2 If the edge $\{u, v\}$ is incident to two occurrences of a mother graph, an edge is introduced between copies of the daughter graph between instances of $u', v' \in \pi(i) \subseteq V_D$ for which $\phi(u') = u$ and $\phi(v') = v$.

Various applications of the inheritance function are depicted in Figure 5–5. In each example, the $a - b$ edges are inherited from interface edges between the rest graph

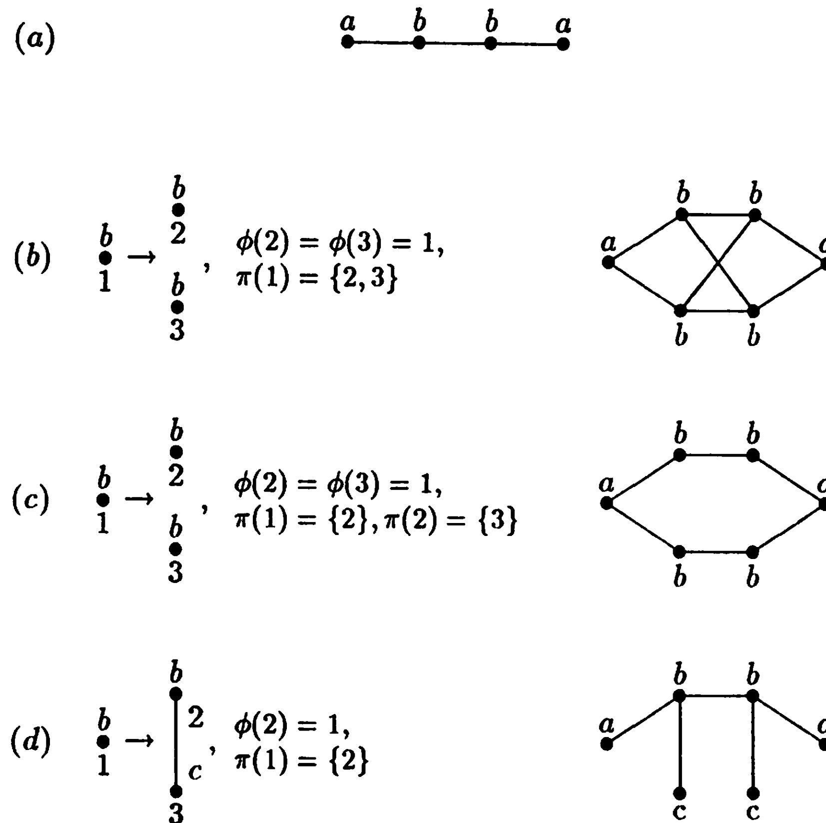


Figure 5–5: The various effects of inheritance functions on production application. The effect of rewriting the same host graph(a), using a total inheritance function without partitioning (b), a total inheritance function with partitioning (c), and a partial inheritance function (d).

and instances of the mother graph; the $b-b$ edges are inherited from interface edges between distinct instances of the mother graph. In (b), the inheritance function was not partitioned so all pairings of inheriting nodes from different daughter graphs inherited the host $b-b$ edge; in (c), the inheritance function was partitioned so only nodes in the same partition inherited the host $b-b$ edge; in (d), the inheritance function was not total and nodes labeled c did not inherit any edges.

An ***aggregate rewriting graph grammar*** is a system $G = (\Sigma, \Delta, P, S)$ where Σ is a finite, nonempty ***label set***, $\Delta \subseteq \Sigma$ is a ***terminal label set***, P is a set of aggregate rewriting productions, and S is a ***start graph***. A graph H ***directly derives*** a graph K,

written $H \Rightarrow K$, if there exists a production that transforms H into K as described previously. The reflexive, transitive closure of $\Rightarrow$ is written $\stackrel{*}{\Rightarrow}$, while the transitive closure is written $\stackrel{+}{\Rightarrow}$. A graph H ***derives*** K if $H \stackrel{*}{\Rightarrow} K$. A graph K is a ***sentential form*** of a grammar $G = (\Sigma, \Delta, P, S)$ if $S \stackrel{*}{\Rightarrow} K$. The ***language of*** G is the set of all sentential forms that are labeled only from Δ.

A ***node aggregate rewriting graph grammar*** consists only of productions whose mother graphs have exactly one node.

To understand the aggregate rewriting mechanism further, consider the following set of commutative diagrams where indexed $i \in \mathcal{J}$:

$$\begin{array}{ccc} M & \stackrel{\phi}{\longleftarrow} & D \\ \psi_i \downarrow & & \downarrow \psi_i \\ H \supseteq M_i & \stackrel{\phi_i}{\longleftarrow} & D_i \subseteq H' \end{array}$$

The number of such diagrams needed to describe a derivation step is, of course, determined by the number of occurrences of the mother graph in the host. The function ψ_i is the occurrence isomorphism restricted to the mother graph, and extends its domain to the daughter graph which is similarly labeled. We might expect, for example, the structure of ψ_i to directly suggest how these graphs are "similarly labeled." The inheritance function ϕ can be extended to an occurrence of the inheritance function $\phi_i = \psi_i \circ \phi$. This suggests how nodes of the image graph inherit from each occurrence in the source graph. Finally, we define a ***descendent function*** $\delta : H \rightarrow H'$ as

$$\delta(v) = \begin{cases} \{v\} & \text{if } v \notin V_{M_k} \text{ for all } k \in \mathcal{J} \\ \phi_k^{-1}(v) & \text{if } v \in V_{M_k} \end{cases} .$$

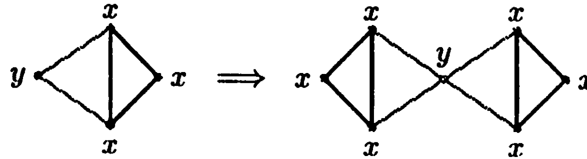
The descendent function indicates what subset of the nodes of H' inherit the edges incident to nodes of H.

We now examine a number of examples of aggregate rewriting graph grammars.

Example. As we have stated, an important distinction between aggregate rewriting graph grammars and other graph grammar formalisms is their ability to copy graphs of arbitrary size and structure. The following production, for example, rewrites an arbitrary x-labeled aggregate to two disjoint copies.

$$\overset{x}{\underset{1}{\bullet}} \rightarrow \overset{x}{\underset{2}{\bullet}} \quad \overset{x}{\underset{3}{\bullet}} \qquad \phi(2) = \phi(3) = 1,\ \pi(1) = \{2\}, \pi(2) = \{3\}$$

Since both daughter nodes inherit from the single mother node, every occurrence is reintroduced twice into the result graph. The edges of the subgraph induced by the aggregate are also copied twice, through the use of Rule 2. In the derivation step



each x-x edge is copied twice, producing the two triangles. The remaining edges are were copied using Rule 1.□

Example. The construction of vectors is performed with the following grammar.

Start graph: $\overset{V}{\bullet}$

Productions: (1) $\overset{V}{\underset{1}{\bullet}} \rightarrow \overset{v}{\underset{2}{\bullet}}\!\!-\!\!\!-\!\!\overset{V}{\underset{3}{\bullet}} \qquad \phi(2) = 1,\ \pi(1) = \{2\}$

(2) $\overset{V}{\underset{1}{\bullet}} \rightarrow \overset{v}{\underset{2}{\bullet}} \qquad \phi(2) = 1,\ \pi(1) = \{2\}$

Since the start graph contains only one node, V, the initial aggregate matched is simply the single node. The first production has the effect of terminally relabeling the aggregate and appending a nonterminally labeled extension to the graph. Successive application of this production extends the vector at the nonterminally labeled end:

$$\overset{V}{\bullet} \overset{(1)}{\Longrightarrow} \overset{v}{\bullet}\!-\!\!-\!\overset{V}{\bullet} \overset{(1)}{\Longrightarrow} \overset{v}{\bullet}\!-\!\!-\!\overset{v}{\bullet}\!-\!\!-\!\overset{V}{\bullet} \overset{(2)}{\Longrightarrow} \overset{v}{\bullet}\!-\!\!-\!\overset{v}{\bullet}\!-\!\!-\!\overset{v}{\bullet}$$

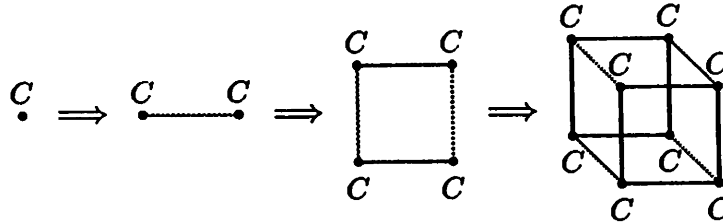
The final production terminally labels the graph.□

Example. Construction of binary cubes is achieved through the use of the following grammar.

Start graph: $\overset{C}{\bullet}$

Productions: $\underset{1}{\overset{C}{\bullet}} \rightarrow \underset{2}{\overset{C}{\bullet}}\!\!-\!\!\!-\!\!\!-\!\!\underset{3}{\overset{C}{\bullet}}$ $\quad \phi(2) = \phi(3) = 1$, $\pi(1) = \{2\}, \pi(2) = \{3\}$

Binary cubes of arbitrary size can be generated from successive application of this single production:



The characteristics of the production are similar to the copying production specified earlier: each daughter node generates a duplicate of the rewritten graph (a cube). To correctly form the cube, however, an edge is introduced between corresponding nodes of both graphs shown here by the gray edges.□

Example. To understand the power of aggregate rewriting productions to perform natural transformations, we demonstrate the use of the following productions in the generation of butterfly networks (SW-banyans with $f = s = 2$). Consider the productions

Productions: (1) $\underset{1}{\overset{A}{\bullet}} \rightarrow$ B node 2 joined by an edge labelled A to node 3 $\quad \phi(2) = 1,\ \pi(1) = \{2\}$

(2) $\underset{1}{\overset{B}{\bullet}} \rightarrow \underset{2}{\overset{B}{\bullet}} \quad \underset{3}{\overset{B}{\bullet}}$ $\quad \phi(2) = \phi(3) = 1$, $\pi(1) = \{2\}, \pi(2) = \{3\}$

(3) $\underset{1}{\overset{A}{\bullet}} \rightarrow \underset{2}{\overset{A}{\bullet}} \quad \underset{3}{\overset{A}{\bullet}}$ $\quad \phi(2) = \phi(3) = 1$, $\pi(1) = \{2\}, \pi(2) = \{3\}$

Suppose the sentential form to be rewritten is itself a butterfly (as is the singleton start graph) labeled A (see Figure 5–6). The first production extends the height of

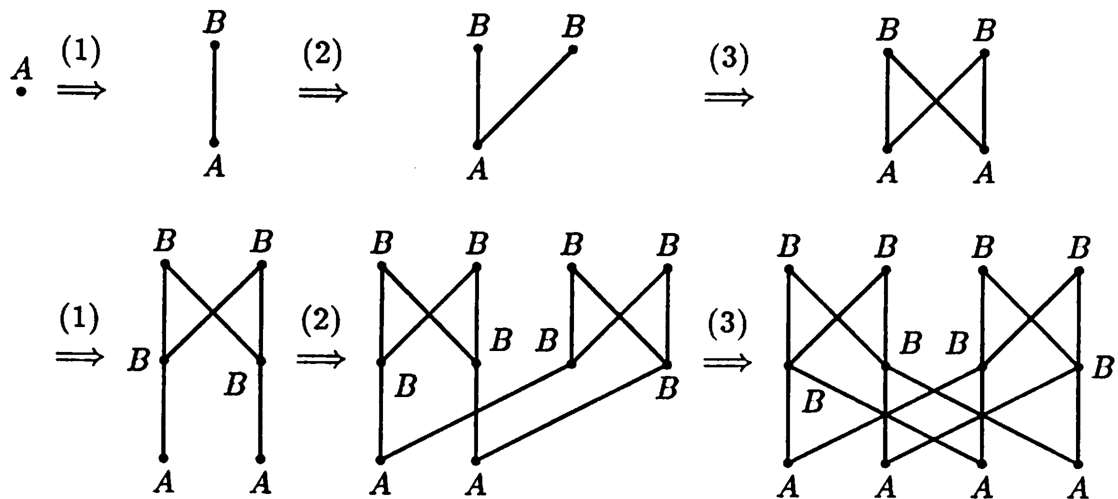


Figure 5–6: Derivation of a two stage butterfly network.

the current network by adding a new layer of "apex" nodes. The second production duplicates the base butterfly network twice, including the arcs to the new layer of apexes. To complete the network, the third production is applied, which creates two copies of the new switches, completing the introduction of the 2×2 crossbars between the new switches and corresponding nodes of the base butterfly. These three productions may be iteratively applied, generating the family of butterfly graphs. □

Example. Some distinction can be made between various graph grammars by their ability to describe string languages[85]. The following grammar generates the language of all vectors labeled with palindromes over $\{a, b\}$.

Start Graph: $\overset{A}{\bullet}$

Productions:

(1) $\underset{1}{\overset{A}{\bullet}} \longrightarrow \underset{2}{\overset{C}{\bullet}}$ $\phi(2) = 1,\ \pi(1) = \{2\}$

(2) $\underset{1}{\overset{A}{\bullet}} \longrightarrow \underset{2}{\overset{B}{\bullet}}\!\!-\!\!\underset{3}{\overset{B}{\bullet}}$ $\phi(2) = \phi(3) = 1,\ \pi(1) = \{2,3\}$

(3) $\underset{1}{\overset{A}{\bullet}} \longrightarrow \underset{2}{\overset{B}{\bullet}}\!\!-\!\!\underset{3}{\overset{C}{\bullet}}\!\!-\!\!\underset{4}{\overset{B}{\bullet}}$ $\phi(2) = \phi(4) = 1,\ \pi(1) = \{2,4\}$

(4) $\underset{1}{\overset{B}{\bullet}} \longrightarrow \underset{2}{\overset{C}{\bullet}}\!\!-\!\!\underset{3}{\overset{B}{\bullet}}$ $\phi(2) = 1,\ \pi(1) = \{2\}$

(5) $\underset{1}{\overset{B}{\bullet}} \longrightarrow \underset{2}{\overset{C}{\bullet}}$ $\phi(2) = 1,\ \pi(1) = \{2\}$

(6) $\underset{1}{\overset{C}{\bullet}} \longrightarrow \underset{2}{\overset{a}{\bullet}}$ $\phi(2) = 1,\ \pi(1) = \{2\}$

(7) $\underset{1}{\overset{C}{\bullet}} \longrightarrow \underset{2}{\overset{b}{\bullet}}$ $\phi(2) = 1,\ \pi(1) = \{2\}$

The first production generates the single character palindrome, the second production begins an even length string and the third production begins an odd length string. The fourth production extends vectors by adding a nonterminal to each of its ends. The remaining productions convert nonterminals to terminals, and can be applied several times throughout a derivation. In each sentential form, any pair of nodes equidistant from the center must have identical labels and must, therefore, be rewritten at the same time by a single production. Note that a string grammar generating palindromes is quite different — it rewrites a single nonterminal in the center of the string. All attempts to use the center as a generator in AR graph grammars result in nonlinear graphs with extra edges; an extension of AR graph grammars, discussed in Chapter 6, allows selective inheritance of edges, which is sufficient to generate these graphs. □

Remark. The palindrome grammar typifies the bias AR grammars have toward the generation of regular structures: it is easy to produce the set of palindromes

but it is relatively difficult to produce the set of nonpalindromes. In fact, if we limit ourselves to single-node rewriting productions as in this example, non-palindromes cannot be produced at all. This is because a single non-terminal cannot rewrite nontrivially in the middle of the vector and preserve the ordered linear structure: if $\mathcal{P}$ is a production rewriting a single node to a connected, nontrivial daughter graph, $\mathcal{P}$ increases the degree of each inheriting node. Clearly, single node replacements must occur at the ends of the vector. If the nonterminals at the end of the strings are similarly labeled, the string will have the same prefix and (reversed) suffix; if they are labeled differently, then there is no control on the relationship between the left and right sides. The result is that a grammar that is either too conservative (generating only graphs which have the form wvw) or too liberal (generating graphs whose ends are independent – a superset of the nonpalindromes). Using more complicated AR grammars that are not limited to single node replacements, it is possible to describe the set of nonpalindromes.

5.2.2 Labeling Extensions

Much of the power of the aggregate rewritting mechanism stems from the ability of the user to identify the portion of the graph to be affected by the production. Since the formalism depends on the notion of *occurrences*, it is important to understand the matching of the mother graph that forms each occurrence.

In the preceding grammars sentential forms were labeled from a small fixed alphabet. In the sentential forms, many nodes were labeled similarly — all nodes of each cube sentential form are labeled identically. This labeling can be undesirable, for two reasons:

1. Although nodes of a graph are logically equivalent, they are often labeled differently. For example, cube processes are often referenced by unique addresses, but our transformations have been total in their rewriting.

2. Often, different interpretations of the same labels generate distinct, but useful aggregates for manipulating these graphs. For example, the two aggregates

of the binary cube whose nodes are distinguished by the value of the most-significant bit in their address are disjoint, but logically important aggregates (each is a component $n-1$ cube).

We assume a more utilitarian labeling of graphs: nodes are labeled by finite strings from a small fixed alphabet. These labels may be interpreted as strings of characters, and representations of integers. We provide a number of examples of useful labels, as motivation:

Value	Motivation
PE12	a processor name (as in Poker)
01001	a process address (as in cubes)
{arg1,arg2,arg3}	local arguments (as in Simon)
COLOR=blue	an attribute/value pair (Simple Simon)
interior.c	process code location
(10,12)	a coordinate pair

To similarly extend the power of the occurrence-matching mechanisms, we use a pattern specification based on ***unification***. Each pattern is expressed in terms of variables which are initially unbound, or *free*. As patterns of the mother graph are compared to labels of the sentential form, values for free variables are deduced, and values of bound variables are compared. A complete label match occurs when the comparison can be made with consistent bindings for all variables (some may remain free). An occurrence of a graph is found when a graph isomorphism allows matching of labels and patterns where all variables can be consistently bound.

We use the following notation to specify label patterns.

Expression	Meaning
λ	the length 0 string
$0, 1, 2, A, B, C$	literal match
x, y, z	character variable
$\overline{0}, \overline{x}$	boolean negation
0...	digit repetition
α, β	string variable
i, j, k	integer variable
$x \cdot \alpha$	pattern catenation
$+, -, \times$	standard arithmetic operations

In this notation, the string 100001 matches $1 \cdot i$ or 10...1, and deduces the value 00001 for i. The patterns for the high and low valued nodes of a cube discussed earlier, are identified by $1 \cdot i$ and $0 \cdot i$ respectively (corresponding nodes would lead to consistent bindings for the variable i).

Once an occurrence has been found all variables have been bound, and the labels on the corresponding occurrence of the daughter graph can be interpolated from the pattern variable values. Each daughter graph, therefore, is reintroduced in a manner that is consistent with the mother graph from which it was generated.

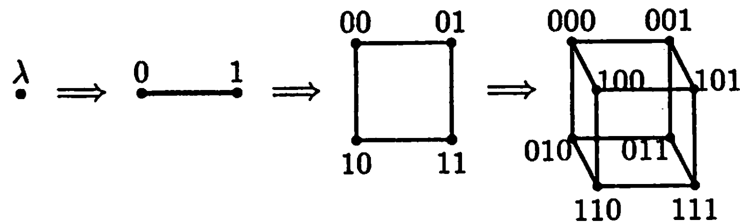
We now consider several examples using pattern-based occurrence recognition.

Example. We reconstruct the binary cube, generating a standard addressing scheme. In the following grammar, each process is labeled with a binary string, which is extended by one digit for each sentential form.

Start graph: $\overset{\lambda}{\bullet}$

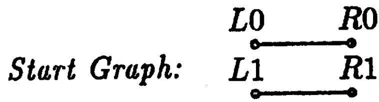
Production: $\underset{1}{\overset{i}{\bullet}} \rightarrow \underset{2}{\overset{0 \cdot i}{\bullet}} \!-\!\!-\!\!-\! \underset{3}{\overset{1 \cdot i}{\bullet}}$ $\quad \phi(2) = \phi(3) = 1$, $\pi(1) = \{2\}, \pi(2) = \{3\}$

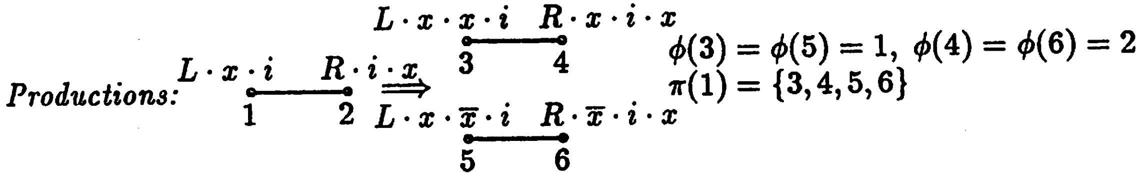
A derivation of the 3-cube would be

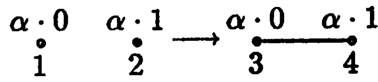


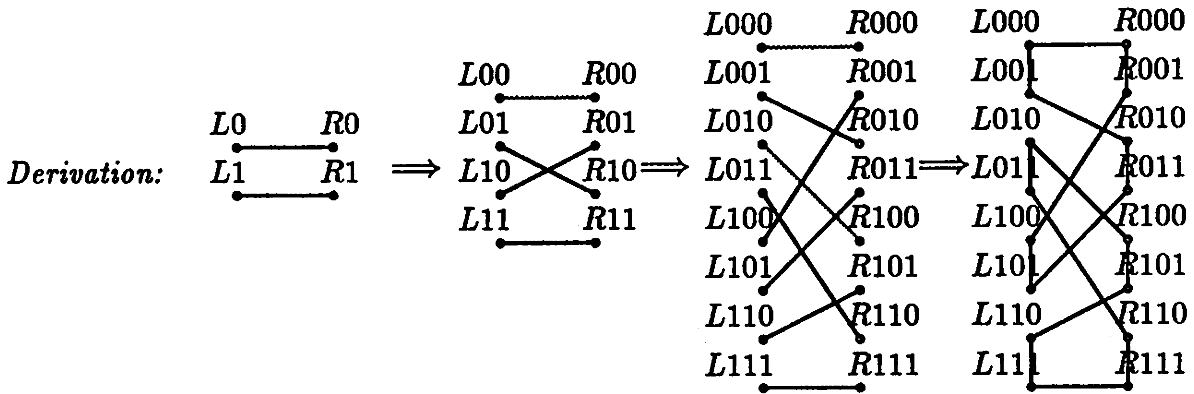
While each node of a sentential form is uniquely labeled, every node is an occurrence of the mother graph, causing the entire cube to be rewritten at each stage.□

Example. A common component of multistage switching networks is the shuffle-exchange[161], which is constructed using the following grammar:









Each rewriting of a shuffle graph generates the next larger shuffle graph, with each node and edge doubling. The mother graph depicted in these productions is a two-node graph which is matched to the sentential form in a manner that binds values for x and i consistently. The first production is applied interactively, until a shuffle graph of the appropriate size is generated. For example, the gray edge of the second sentential form generates the two gray edges of the third. The second production can then be used to generate exchange arcs.□

5.3 Structural Properties

In this section, we consider the structure of the graphs which are generated by grammars composed of productions whose mother graphs are single nodes — *node aggregate rewriting* or NAR graph grammars for short.

We first prove some basic properties of NAR graph grammar derivations, and the relationships held among sentential forms. The next sections identify conditions that preserve two important properties of Chapter 3: connectedness and symmetry.

5.3.1 Basic Properties

One important distinction between edges is the manner by which they may be reinstantiated during a production. Edges in the rest graph are rewritten directly, while interface edges follow edge rewriting Rules 1 and 2. The nature of this distinction is encoded, therefore, by the number of points where an edge is incident to an aggregate.

Definition 5.1. *The edges of a sentential form in a AR graph grammar derivation are classified according to the number of aggregate nodes to which they are incident. Edges are*

- Type 0 *whenever an edge is contained by the rest graph (a* rest edge*),*
- Type 1 *whenever an edge is singly incident to the aggregate (an* interface edge*), or*
- Type 2 *whenever an edge is incident to two nodes of an aggregate (an* aggregate edge*).*

Our first theorem describes the "monotonicity" of graph grammar derivations. Because ϕ is surjective successive applications of NAR productions fail to remove arcs or nodes. This property is important to the understanding of the effects of production application.

Theorem 5.2. *Let $\mathcal{P} = (M, D, \phi, \pi)$ be a production, where M consists of a single node. If $\mathcal{P}$ rewrites H to H' there exists a subgraph isomorphism from H into H'.*

Proof. Pick an inheriting node, x, from D. Since the mother graph M is a single vertex, it is sufficient to show that the union of the set of occurrences of x in H' with the nodes of the rest graph induces a subgraph H'' of H' which is isomorphic to H (see Figure 5–7). Let $\iota : H \to H''$ be the bijection which maps rest graph nodes of H directly to H'' and occurrences of the node of M to respective occurrences of x in $H'' \subseteq H'$. Suppose $e = \{u, v\}$ is an edge in H. If e is a Type 0 edge, it is fully in the rest graph, and occurs in H'' trivially. If e is Type 1, then a corresponding edge is generated by Rule 1 is in H'. If e is Type 2, then the images of u and v reside in the same partition (the partition of x) and Rule 2 generates a corresponding edge in H''. Since all nodes and edges of H are inherited by H'' respecting ι, and ι is a bijection, then ι is a graph isomorphism. □

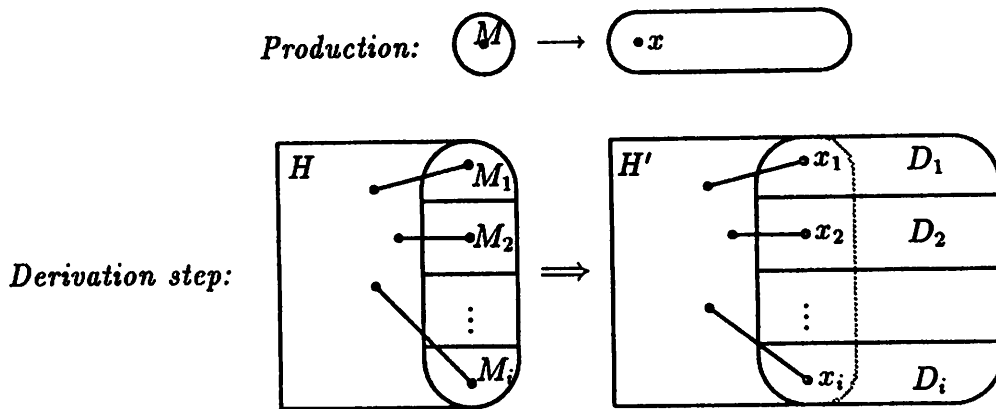


Figure 5–7: A production and derivation for Theorem 5.2.

Since NAR graph grammars are composed only of productions with singleton mother graphs, we make a special statement about the relationship of the graphs they derive.

Corollary 5.3. *The sentential forms of a NAR derivation form an ascending chain on graph-inclusion.*□

Theorem 5.2 is actually conservatively stated. Since the selection of x in the proof could have been *any* inheriting node of D, one may find an isomorphism from

H into H' that maps an arbitrary node v of H onto any of its descendents in $\delta(v)$. This allows us to make the following claim.

Theorem 5.4. *Let $\mathcal{P}$ be a production as in Theorem 5.2. If H is rewritten to H' by this production, then if u is a node of H and v descends from u,* $\deg(u) \leq \deg(v)$.

Proof. The node u is either a rest graph node or it is an aggregate node. In the latter case any descendent of u can be used as x in the proof of Theorem 5.2. The ascending chain condition guarantees the degree of descendent nodes is nondecreasing. □

Clearly this result holds true for each step of a NAR graph grammar derivation.

Now consider a grammar which generates graphs with *unbounded* degree. Edges may be introduced by a production or they may be inherited from the previous sentential form. The degree of graphs which define productions, however, is clearly bounded and thus most edges in a graph of high degree are accumulated through inheritance.

A "loop in inheritance of labels" occurs when a node inherits edges from a similarly labeled node in a previous sentential form. The grammar for the family of binary cubes is an example of such a grammar. When the productions which support such a loop add edges to the graph through inheritance, they may accumulate, generating graphs with arbitrary degree.

Lemma 5.5. *Suppose there is no loop in the inheritance of labels in a NAR graph grammar with production set P and label set Σ. Then each node of any sentential form has a bounded number of eventual descendants.*

Proof. The number of descendents of a node in any production is bounded by

$$\max_{\mathcal{P} \in P}(|dom(\phi_{\mathcal{P}})|)$$

Possibly all of these descendents are themselves rewritten by productions. If there are no loops in the inheritance of labels, this rewriting must have depth less than

$|\Sigma|$. We conclude the number of eventual descendents of a single nonterminal could be no more than

$$\left(\max_{\mathcal{P}\in P}(|dom(\phi_{\mathcal{P}})|\right)^{(|\Sigma|-1)}$$

in any sentential form. □

Once the number of eventual descendents is bounded, the number of edges that each descendent can introduce is likewise bounded.

Theorem 5.6. *Suppose G is a NAR grammar and there are no loops in the inheritance of nonterminals, then G generates only graphs of bounded degree.*

Proof. Nodes of a sentential form are of two types: those that descend from nodes in previous sentential forms and those that do not (*ie.* are newly introduced). We consider nodes of the last type — call these ***source nodes***. By bounding the degree on source nodes and any node inherited from such node, we place a bound on the degree of any node appearing in a sentential form.

Select a source node, u. This node has degree at most d — the maximum degree of any daughter graph or start graph node. Since u may be rewritten at most $|\Sigma|-1$ times, any descendent could accumulate at most $d(|\Sigma|-1)$ new edges (edges newly introduced by daughter graphs). Since no edge can be replicated more than a bounded number of times (Lemma 5.5), a bound on the degree of any descendent of u is

$$d(|\Sigma|)\left(\max_{\mathcal{P}\in P}(|dom(\phi_{\mathcal{P}})|\right)^{(|\Sigma|-1)}$$

As this bound is constant for any particular grammar, the theorem is proved. □

Because NAR graph grammars have these monotonic properties, in some sense productions which occur late in a derivation can not "undo" edges and nodes that were added early in the derivation. As we shall see in the next section, NAR graph grammars are incapable of rejoining descendents of nodes which have been separated.

5.3.2 Connectedness

While it not possible to remove edges from a connected component, it is possible to generate larger, disconnected graphs. Our next theorem demonstrates the conditions under which connectedness is preserved in sentential forms. As we shall see, this can be avoided by placing appropriate constraints on the inheritance and partitioning in productions.

Theorem 5.7. *Suppose H is a connected graph rewritten to H' by $\mathcal{P} = (M, D, \phi, \pi)$, a production whose mother graph consists of a single node. If any of following conditions hold, H' is a connected graph.*

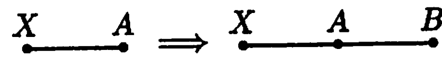
1. *the daughter graph is connected, or*
2. *H is nontrivial, and each connected component of the daughter graph mentions a node of each partition, or*
3. *H is not totally rewritten, and some node of each connected component of the daughter graph inherits from the mother graph.*

Proof. With the aid of Figure 5–8, we prove the theorem for each condition:

Case 1: Suppose the daughter graph, D, is connected. If H is a singleton, H' is isomorphic to D and is connected. If H is nontrivial, then the image of each node is a connected component — either the node itself or a connected graph isomorphic to D. It is clear that if the nodes of H are connected by the edges of H, then the images of these edges serve to connect the connected components.

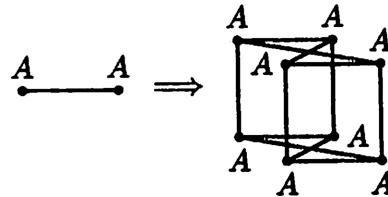
Case 2: Suppose H is nontrivial and that connected component in D mentions a member of each partition. Since H is nontrivial, each aggregate node is mentioned by a Type 2 or Type 1 edge. In the Type 1 case, each edge rewrites to many edges which are incident to all descendents of the mother node and all connected components are adjacent to the rest graph node. In the Type 2 case, each edge is rewritten to one or more crossbars which remain in partitions. However, each component of each image of the daughter graph mentions members of each partition. Thus Type 2 edges serve to connect all components of one daughter graph to all components of another. In either case, then, the image graph is thus connected.

Case 1: A (1) $\longrightarrow$ A (2) — B (3) $\quad \phi(2) = 1,\ \pi(1) = \{2\}$



Case 2: A (1) $\longrightarrow$ A (2) | A (4), A (3) | A (5) $\quad \phi(2) = \phi(3) = \phi(4) = \phi(5) = 1$

$\pi(1) = \{2,3\}, \pi(2) = \{4,5\}$



Case 3: A $\longrightarrow$ A (2) — B (3), A (4) — B (5) $\quad \phi(2) = \phi(4) = 1$

$\pi(1) = \{2\}, \pi(2) = \{4\}$

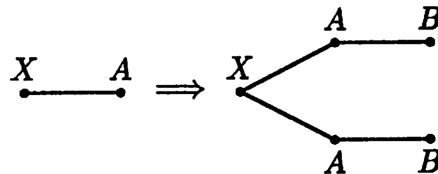


Figure 5–8: Examples of the three conditions for preserving connectedness.

Case 3: Suppose, now, that H is nontrivial and not completely rewritten by $\mathcal{P}$ and that some node in each connected component of D inherits from the mother graph. The rest graph is trivially rewritten. Type 2 edges rewrite to connected components of corresponding daughter graphs. Thus the image of the aggregate *can* be disconnected. However, since each node of the aggregate has a descendent in every connected component of D, Type 1 edges make each component adjacent to the same rest node. This is sufficient to make H' connected and the theorem is proved. □

The only mechanism for introducing new edges into a sentential form is within a single occurrence of a daughter graph. All other edges are inherited. Thus, if two nodes are not adjacent a sentential form, no node-rewriting production can make them adjacent. We may then conclude that two connected components of a sentential form likewise cannot be "reconnected".

Corollary 5.8. *The number of components in successive sentential forms in a NAR graph grammar derivation is nondecreasing.*□

Next, we more fully consider the preservation of relationships between nodes which are rewritten.

5.3.3 Symmetry Properties

Many parallel algorithms and hardware show some degree of symmetry. The effect of symmetry on a system is to make more of its components interchangeable. It is interesting, therefore, to determine under what conditions symmetry is preserved in the derivation of communication structures. We first turn to more technical definitions of symmetry.

Definition 5.9. *Two nodes u and v of a communication graph are* similar *if an automorphism of the graph maps u to v. A graph is node-symmetric (here, just* symmetric*) if every pair of nodes are similar. The graph H is* symmetric on subgraph *$S \subseteq H$ if every pair of points in S are similar and the respective automorphisms are automorphisms of S.*

The concern with point-symmetric graphs stems from the desire to generate communication structures with interchangeable processes. A process can *migrate* to similar processors. If a communication structure is symmetric with respect to some subset of nodes, each node has a similar view of resources and each can be used to locate similar processes.

Definition 5.10. *We say an automorphism α of a daughter graph* respects partitioning *(or is* π-symmetric*) if whenever u and v are in the same partition $\alpha(u)$ and $\alpha(v)$ are also. We say it* respects inheritance *if whenever u is a descendent of mother node m, so is $\alpha(u)$.*□

The need for π-symmetry is demonstrated by Figure 5–9. The daughter graph of production (a) is symmetric, but not π-symmetric (*e.g.* map node 3 to node 2), thus the image graph is not symmetric. If a graph is π-symmetric, then partitioning of nodes descending from a common parent is equal; otherwise, as this figure suggests, crossbars of different sizes occur between instances of the daughter graph.

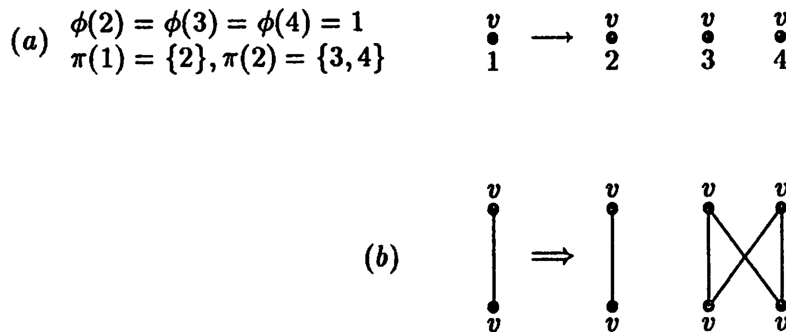


Figure 5–9: Necessity for π-symmetric functions.

If the symmetry of the daughter graph does not respect the inheritance function, the symmetry can be destroyed by "nonsymmetric" interfaces with the rewritten aggregate. Consider Figure 5–10. The daughter graph of the production in (a) is symmetric with respect to nodes labeled v, but any automorphism used to

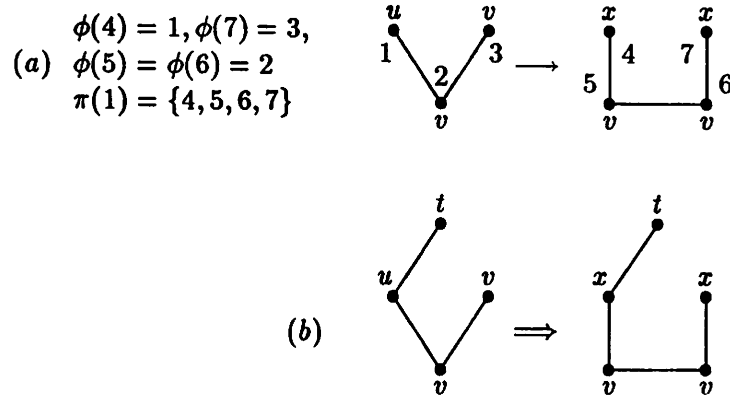


Figure 5-10: Necessity for preservation of inheritance.

indicate similarity between these nodes must exchange nodes labeled x, which does not preserve ϕ. The result is the production of nonsymmetric graphs, as in (b).

Our next lemma indicates when symmetries of a daughter graph are maintained as symmetries within each instance of the daughter graph in the image of a production.

Lemma 5.11. *Let $\mathcal{P}$ be a production which rewrites a graph H to H'. If the daughter graph D is symmetric on the discrete subgraph D_v, composed of descendents of mother graph node v, and the similarity automorphisms respect inheritance and partitioning, then H' is symmetric on each image of D_v.*

Proof. We construct appropriate automorphisms of H' from automorphisms of D. Suppose $u, v \in V_D$ and α is an automorphism of D which respects partitioning function π, preserves ϕ, and demonstrates the similarity of u and v. We may extend this automorphism to H' using

$$\alpha'(x) = \begin{cases} \psi_k \alpha \psi_k^{-1} & \text{for } x \in V_{D_k} \text{ for some } k \\ x & \text{for } x \text{ in the rest graph} \end{cases}.$$

Clearly, α' is an automorphism on the rest graph. For each $x \in V_{M_i}$ it also is an automorphism on the descendents of x, as this is an automorphism which preserves

the function inheritance ϕ_i. We need only show that α' is an automorphism on the interface edges. These edges are Type 1 and Type 2.

Suppose $e = \{u, v\} \in E_H$ is a Type 1 edge. Then, since α' respects inheritance and α' is an automorphism, the descendents of u inherit a copy of e; α' is an automorphism on these edges.

Suppose $\{u', v'\} \in E_{H'}$ is an image of a Type 2 edge, with $u' \in V_{D_i}$ and $v' \in V_{D_j}$. Since there is an edge between u' and v', each must be an image of daughter nodes in the same partition, *ie.* $\psi_i^{-1}(u')$ and $\psi_j^{-1}(v')$ occur in the same partition. Since α is π-symmetric, $\alpha(\psi_i^{-1}(u'))$ and $\alpha(\psi_j^{-1}(v'))$ are also in the same partition, and thus $\alpha'(u')$ and $\alpha'(v')$ must share an edge.

The fact that α' is a graph isomorphism stems directly from the fact that α' is a bijection on $V_{H'}$. □

Figure 5–11 demonstrates the use of this lemma. Since the daughter graph meets the conditions of the lemma, every instance of the daughter graph in the image shares the similarity property.

Unfortunately, the previous lemma is insufficient to show that nodes are similar *between occurrences* of the daughter graph — that requires the host graph to exhibit symmetry (for example, a cube). This would be an extremely powerful result, since it would allow the designer to specify homogeneous systems easily.

In order to assure *all* instances of descendents of a mother node are similar, we must impose some similarity between occurrences of the mother graph. The following theorem suggests that if the occurrences of the aggregate are "similar," the symmetry of daughter graph becomes symmetry in the transformed aggregate. We first define our notion of similarity of occurrences.

Definition 5.12. *A pair of occurrences M_i and M_j, $i, j \in \mathcal{J}$ in a graph H are* similar *if there exists an automorphism α of $\mathcal{J}$ which maps i to j and if α induces an automorphism of H which acts as $\psi_{\alpha(k)} \cdot \psi_k^{-1} : M_i \to M_{\alpha(k)}$ for each $k \in \mathcal{J}$. A graph is* symmetric with respect to the occurrences of M *if every pair of occurrences are similar.*□

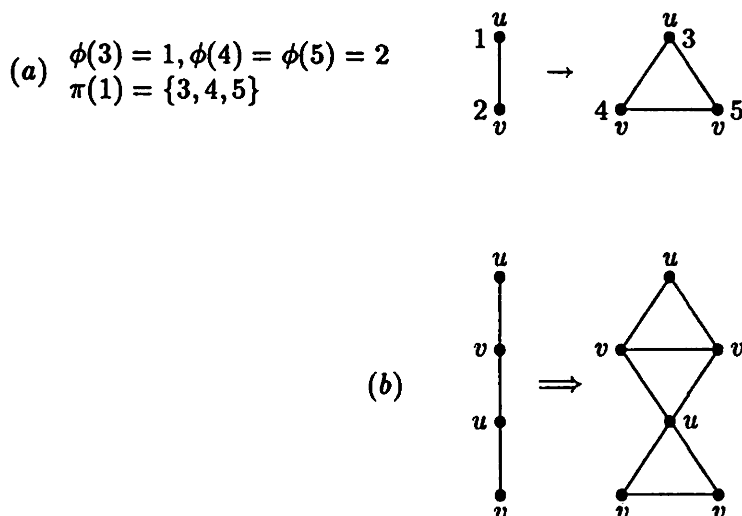


Figure 5–11: Daughter-induced symmetry. The symmetry of the daughter graph (a) is preserved within every occurrence found in the image graph (b). However, there is no symmetry *between* daughter graph occurrences because the host graph did exhibit have symmetry between mother graph occurrences.

This leads directly to our main result on symmetry.

Theorem 5.13. *Suppose P is a production which rewrites H to H' and the mother and daughter graphs meet the conditions of Lemma 5.11. If H is symmetric with respect to the occurrences of M, then for $v \in V_M$, H' is symmetric among all descendents of occurrences of v.*

Proof. From Lemma 5.11 we know that for each occurrence $M_i \subseteq H$, H' is symmetrical on the descendents of v for each $v \in V_{M_i}$. We need only show that descendents of two occurrences of $w \in M$ are similar.

Consider Figure 5–12. Select $w \in V_M$ and let w_i and w_j be occurrences of w in H. Let u_i and v_j inherit from w_i and w_j respectively. Since there exists an automorphism $\beta_{i,j}$ of H mapping M_i to M_j in the natural way, $\beta_{i,j}(w_i) = w_j$. We

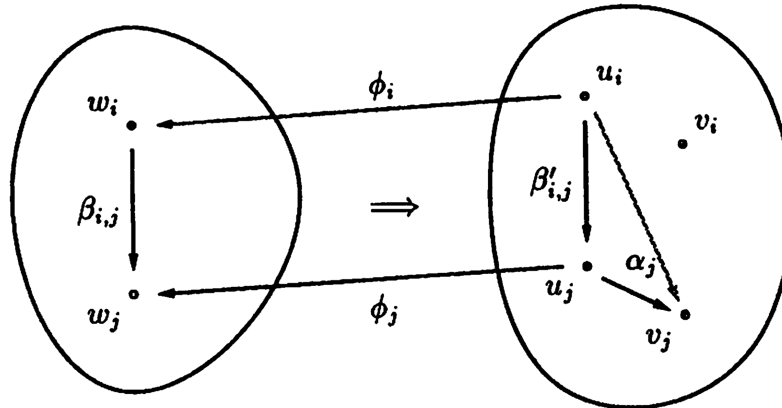


Figure 5–12: The functions relating descendents of different mother node occurrences.

may extend this to an automorphism of H' using

$$\beta'_{i,j}(x) = \begin{cases} \beta_{i,j}(x) & \text{if } x \text{ is in the rest graph} \\ \psi_{\beta(k)}(\psi_k^{-1}(x)) & x \in V_{D_k} \end{cases} .$$

$\beta_{i,j}(x)$ acts as $\psi_j\psi_i^{-1}$ on D_i (mapping D_i to D_j) and $\beta'_{i,j}$ is an automorphism on the descendents of occurrences of w, mapping u_i to a similar descendent of w_j, u_j.

We now construct α_j, the automorphism of Lemma 5.11, which maps u_j to v_j. It follows that $\alpha_j\beta_{i,j}$ is an automorphism on H and the descendents of occurrences of w.

Since this construction is independent of w, the theorem is proved. □

The designer of a network, therefore, can be assured that such a production, which rewrites a single occurrence has a similar effect throughout the graph. If an AR graph grammar's productions rewrite nodes, we have the following corollary.

Corollary 5.14. *Suppose H is symmetric and is totally rewritten to H' by a production with a singleton mother graph. If every node of the daughter graph D is a descendent and D is symmetric in a way that respects partitioning, then H' is symmetric.*□

The sole production of the grammar for binary cubes meets these constraints. Since the start graph for this grammar is symmetric, the same must be true of every member of the language.

5.4 Conclusion

We have shown AR graph grammars are extremely effective for specifying both families of graphs and recursive structures. With proper restrictions many properties can be preserved which are inherent to the structures associated with communication among thousands of processes. Both a bound on degree and symmetry, for example, are important characteristics of massively parallel communication structures and we have identified conditions that preserve these characteristics in NAR sentential forms. When describing such structures with a graph editor, constraints on productions similar to those presented here can provide feedback to the programmer. Such metrics aid the designer of programs for highly parallel machines.

Chapter 6
Graph Editors

In this chapter, we bring theory to reality through the design of a communication graph editor based on aggregate rewriting graph grammars. This work is currently being implemented as part of the Simple Simon Parallel Programming Environments Project. Thus this chapter not only describes application of the research in this dissertation, but it provides a basis for the design of new tools for the specification of massively parallel computation.

In an attempt to skip irrelevant details of implementation, we only develop those concepts that are interesting to the designers of parallel systems. In Section 6.1, we discuss some aspects of the Simple Simon environment that have affected the decisions we made in constructing this editor. In Section 6.2, we discuss the philosophy behind the editor, outlining the characteristics that we deem desirable in such a tool. In Section 6.3, we discuss the specifics of the graph building cycle supported by our editor.

6.1 The Simple Simon Environment

The Simple Simon environment is a testbed for tools that support massively parallel computation. The environment is not designed for a particular architecture, but instead runs in conjunction with the Simon multicomputer simulator[57]. Processors in the simulator are individually programmed, share no local memory, and use message passing for communication.

Central to Simple Simon is the concept of a parallel program database. Since each process is individually programmed, it would be difficult to manage the thousands of code bodies needed to support a single parallel program. Instead, we view the program as a collection of records describing processes, channels, and ports (the incidence of a channel and a process). The program records constitute the database

at time zero. During execution, Simon's instrumentation of the event queue provides execution trace information which is time stamped and added to the database as in a consistent format. The database is monitored *post facto* by the Belvedere animating debugger[75].

In order to make the system both extensible, each object and event in the Simple Simon environment can be labeled with an arbitrary set of *attributes*. An attribute is a name/type/value triple. Some attributes are directly interpreted by the Simple Simon environment, while others parameterize the user's code. The *graphical-rendition* attribute, for example, specifies a consistent graphical description of an object and is interpreted directly by a variety of tools in the environment. Process attributes parameterize the code and are passed to the process at run-time. We depend, in this chapter, heavily upon the use of attributes as labels on nodes, edges, and ports.

Communication structures are explicitly specified in Simple Simon and they provide a framework for the introduction of attributes and code bodies for processors. Canister communication has also been introduced in Simple Simon, making the construction of a graph editor an important step toward the integration of many portions of the environment.

6.2 The Graph Editor

The graph editor is, like any other editor, a mechanism for directing the user's efforts toward the construction of a well specified structure. The goals of the graph editor are as follows.

1. It must provide user-motivated graph annotations. A primary goal of the editor is to allow the specification of communication graphs in the "language" of the algorithm designer. The program should not be bound by constraints of the target architecture, but instead by constraints of standard parallel algorithm design techniques. Furthermore, it is important to provide support for implicit, system-defined annotations of graphs, especially those motivated by graph theoretic and graphical display considerations. It is also impor-

tant that transformations of the communication structure make appropriate transformations in its labeling.

2. It must provide support for manipulating large graphs. The user must be able to concisely specify transformations that can be applied to graphs with thousands of nodes. These transformations must preserve desirable graph characteristics (that, for large graphs, would otherwise be difficult to verify).

3. It must provide support for specifying graph families. Because parallel programmers tend to design and implement their algorithms in-the-small and then scale them for massive parallelism, the *scaling of the algorithm* should be "abstracted out" of the program specification. Thus communication structures should be specified as graph families.

With these goals in mind, we consider the process of constructing graphs with a graph editor.

6.3 The Graph Building Cycle

We use, as a model for the construction of graphs, a graph grammar derivation. The programmer initiates the construction of the graph with a *start graph*. This graph is often the smallest instance of the family of communication structures that he is specifying. The programmer then enters the *graph building cycle*. Each transformation of the graph occurs in one loop of the graph building cycle and corresponds to a step in a derivation. To specify a graph transformation, three distinct operations are necessary:

1. *Domain identification*. The domain for the transformation is identified by the programmer.

2. *Transformation specification*. The effect of the desired transformation is specified.

3. *Transformation application*. The specified production is applied to the graph in order to effect the desired transformation.

After the three operations have been applied, a logically complete transformation of the graph has been accomplished. The editor then analyzes the graph and provides feedback to the user. Experience suggests that most graphs can be specified with a small number of transformations. Applicability of these transformations is often independent of scale, thus iterative application of the transformations leads to the construction of a graph family.

In the following sections, we consider each step of the graph building cycle in detail.

6.3.1 Domain Identification

Before any transformation can be performed on a graph, the domain of rewriting must be specified. Two pieces of information are needed: the aggregate and the binding of label variables. Both the domain and label context are identified with the specification of a single aggregate pattern which matches labels on the graph and forces bindings for label variables. In Simple Simon this specification can be either textual or graphical.

Textual specification of aggregates and details of binding labels were the subject of the discussion of Section 5.2.2. More often, however, aggregates are specified through a graphical interface by selecting nodes from an instance of the graph family under construction. A predicate defining the aggregate is then automatically generated by an analysis of node attributes (labels). Once this ***predicate deduction*** is performed, the user is presented with a ranked set of choices for his aggregate specification. Thus if he selects all nodes in the bottom row of a 4 $\times$ 4 mesh labeled with row and column indices, his choices will include the predicates `row=max` and `row=4`. `row=max` is probably the appropriate choice because it is the more general predicate and would, therefore, be useful in identifying the domain for transformations of larger graphs in the same family.

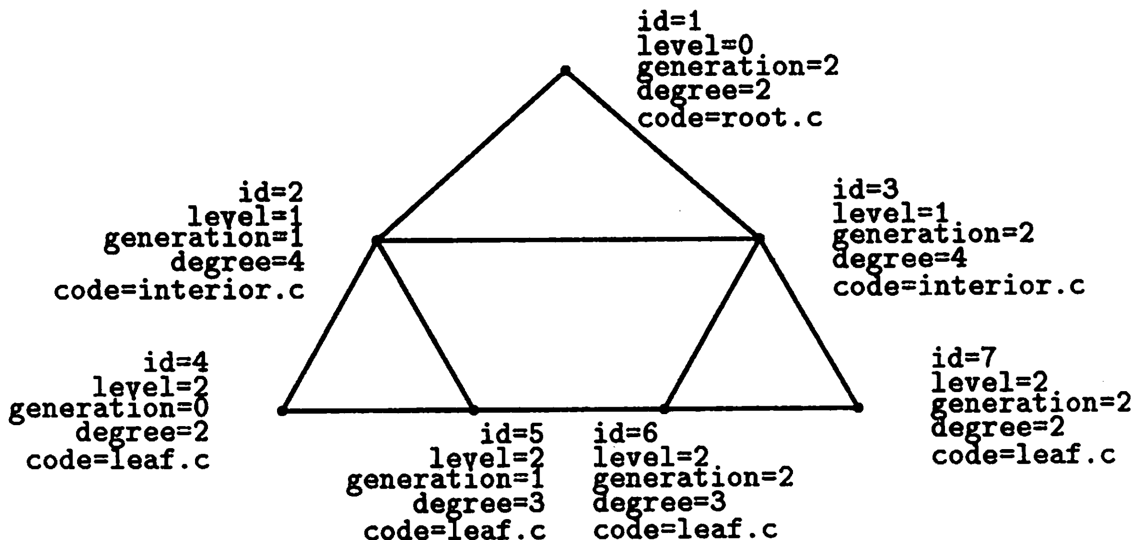
Our experience with ***predicate deduction*** in lightly annotated graphs indicates that most useful aggregates can be defined with little intervention from the programmer[171]. The process is as follows.

First, programmer-supplied node labels are automatically augmented with

graph-theoretic attributes including the creation index of a node, the index of the graph which generated this node, the graphical location of the node, and the index of the containing connected component.

Next, each attribute is *inverted*, providing an associated predicate of the form `name=value` for the node it labels. These *base predicates* are then used to build conjunctive or disjunctive compositions which precisely describe the desired aggregate. As a predicate is identified, it is appended to a list of usable descriptions. Once a list of descriptions is compiled, generalization of predicates is attempted.

Example. Consider the following binary x-tree. The user supplied only the `code` attribute during its construction, all other attributes were supplied by the system. Here, the degree of the root node is 2, interior nodes have degree 4, and leaves have degree 2 or 3. The nodes have various generation counts because two other x-trees were specified in the construction of this instance, thus some parts of the tree are "older" than others.



Inversions on node labels include, for example, the predicate `code=interior.c` which describes the node set $\{2,3\}$ and the predicate `degree=2` which describes the three corners of tree, nodes $\{1,4,7\}$. If the leaf nodes are selected by the user

as the desired aggregate, several predicates are identified, including `code=leaf.c`, `level=2` and the generalized predicate, `level=max`. If the left subtree is selected, the predicate identified is `(generation=0 or generation=1)`.□

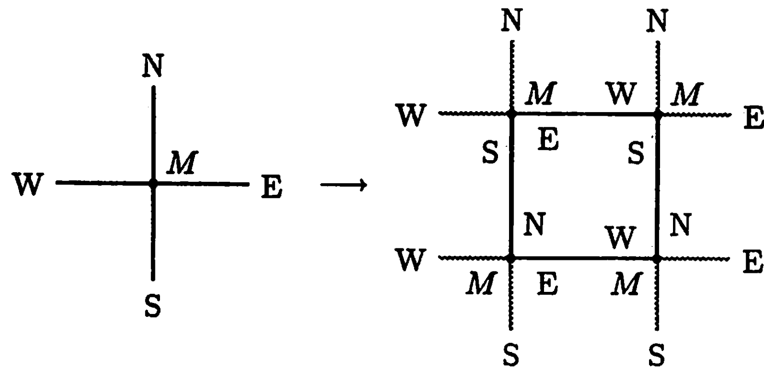
6.3.2 Transformation Specification

Once the domain of application has been determined and pattern variables have been bound, an appropriate transformation must be specified. We model these transformations in our graph editor as a modified form of the node aggregate rewriting graph grammar discussed in Chapter 5.

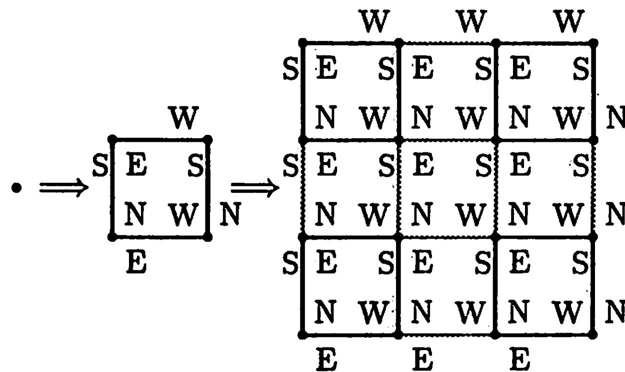
Unlike the graph formalism presented earlier, we explicitly manipulate ***ports*** — the labeled incidence of an edge and a node. There are two major motivations behind this decision. First, the use of ports in many environments, including Simple Simon, necessitates a mechanism for incorporating ports in the grammar formalism. Second, aggregates of nodes alone can not be used to construct certain necessary communication structures, including meshes.

The important distinction between the formalism of Chapter 5 and the specification of productions in the graph editor is a modification of inheritance of edges. Instead of inheriting ***all*** edges from antecedent nodes, each node inherits a subset of the edges of an antecedent node. Edges incident at a single node are distinguished by ***port labels***. When no distinction is made among port labels, each node inherits all edges of its antecedent and the grammar mechanism is identical to that of NAR graph grammars.

Example. The following port-labeled NAR graph grammar production specifies the sole production of a grammar which describes the family of square meshes. (Node labels are in italics, port labels are Roman.)



Each gray port on the right inherits from the similarly labeled port on the left, and gray ports in the same row (or column) determine the partition. Occurrences are based on node labeling alone; missing ports in a rewritten sentential form fail to produce respective ports in the sentential form. We thus have the following derivation from an appropriately labeled point:



Edges in gray are edges inherited from previous sentential forms.□

The specification of productions in the Simple Simon graph editor are modeled after node aggregate productions: the transformation always rewrites a single node to a user specified graph. Incident to both the node and graph are edges which are rewritten only if corresponding edges exist in the context of the rewriting. Inheritance is specified by associating edges on the right with edges on the left. The edges of each partition are displayed in a manner that indicates their similarity (for example, similar edges might be similarly colored).

6.3.3 Transformation Application

Once the aggregate has been identified and the production specified, the process of transformation is straightforward. Each occurrence of the rewritten portion of the production is removed and each replacement graph is reintroduced with appropriate treatment of the interface. In the Simple Simon environment, the rewriting of each node acts as a rewriting of the static portion of the database.

The generation of labels in the rewritten graph is based on the labels and variables which annotation the antecedent node. When user-supplied labels are not explicitly rewritten, they are inherited from the antecedent node without change. System-supplied labels are not rewritten but simply regenerated.

6.3.4 Graph Analysis

The process of describing a graph family is highly interactive. In particular, the graph editor provides information which aids preserving graph theoretic properties, and the user informs the system of progress toward constructing the desired graph.

A benefit of modeling the graph editor after aggregate rewriting graph grammars is the ability to glean information about the current graph by analyzing the transformations from which it was derived. Theorems of Chapter 5 aid in identifying transformations of the graph which potentially do not preserve desired properties.

Similarly, the user must inform the editor of successes in constructing the desired graph family. The generation of a member of the desired graph family determines the productions which can be iteratively applied to generate successive family members.

6.4 Conclusion

The generation of graphs in the Simple Simon environment is central to the programming process. The decision to construct a graph editor has lead to a parallel-specific solution based on a modified form of node aggregate rewriting graph grammars. This mechanism can efficiently generate many of the communi-

cation structures which are necessary in massively parallel systems.

The shaded portion of Figure 6–1 describes the components of the proposed editor for communication structures in the Simple Simon environment. User re-

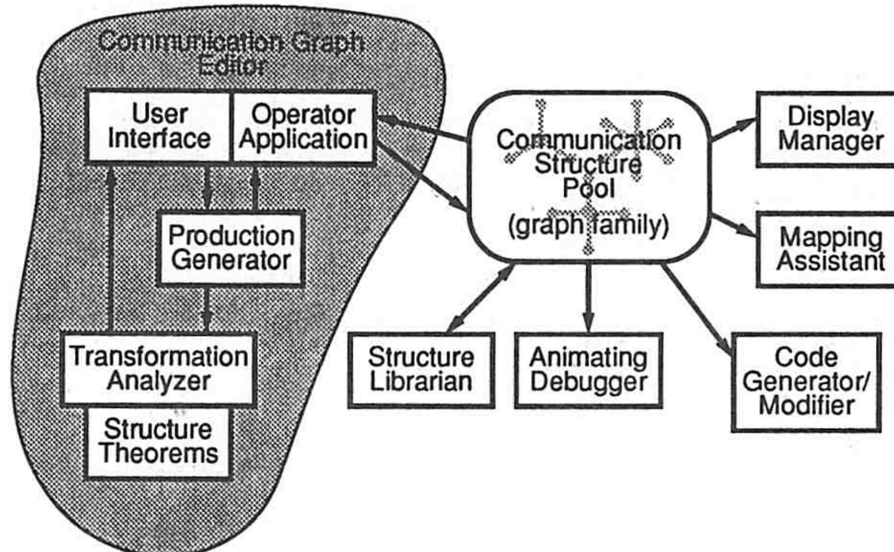


Figure 6–1: Components of the proposed graph editor.

quests become graph editor productions which act on a family of structurally related graphs maintained in the Simple Simon database. An analysis of the productions based on our structural theorems from Chapter 5 is then performed, providing feedback to the user.

Chapter 7
Conclusions and Evaluation

In this chapter we review methods for more effective description of communication in massively parallel architectures, as proposed by this dissertation. We then summarize direct research contributions of this thesis and, in the last section, we consider the limitations of the work and proposals for further work.

7.1 Overview

Synchronization of parallel processes requires some form of interprocess communication. In massively parallel algorithms communication is a mechanism which controls and structures the computation. It is fundamental to our understanding of each stage of the software life cycle. Unfortunately, even though we can not draw on our sequential programming experience for aid in specifying this aspect of parallelism, many environments are little more than "parallelized" sequential environments. To remedy this situation, we concern ourselves with concise, parallel-specific abstractions and tools for the specification of parallel communication.

The philosophy of software engineering is to make the software specification as close to the algorithm description as possible. Our approach has been motivated by the descriptions of communication as they appear in the literature. We believe that this is an important step toward making the process of programming massively parallel architectures more manageable.

7.2 Contributions of This Research

This dissertation sheds light on an aspect of parallel programming which has largely been overlooked: ***specification of massively parallel ensemble communication.*** Specifically, this dissertation makes the following important contributions:

Parallel algorithm design paradigms. Paradigms for designing sequential algorithms are important to our understanding of the sequential programming process. Their use suggests basic abstractions which aid in implementation. In this dissertation, we have identified characteristics of communication, such as connectivity, symmetry and low diameter, that are fundamental to parallel programming. These characteristics are seen as metrics for evaluating proposed methods for specification of communication.

A communication abstraction. Communication should be considered at a more abstract level than the message passing provided in many programming environments. The various techniques of constructing parallel algorithms require important abstractions of communication. These abstractions often consist of patterns of message exchange whose coordination is more global than point-to-point message passing. We proposed *canister communication*, which supports correct access to reusable message carriers traveling among processes on specified paths.

A model of communication specification. Communication is poorly specified in most environments because parallelism is provided by an extension of sequential data structuring techniques. We developed a new specification mechanism based on *graph grammars*. This mechanism is unique in its uniform treatment of *aggregates* of logically related nodes, which describes the structure of communication among processes more accurately than parallelized declarative mechanisms. Furthermore, it supports the implementation of algorithm *families*, which removes the tendency to respecify algorithms for minor variations in host architectures. Finally, we demonstrated the use of graph grammars as theoretical support for a graph editor.

7.3 Limitations and Future Work

As parallel programming environments become more responsive to the needs of programmers we expect the research presented in this dissertation will evolve.

New design paradigms will become important. Our treatment of algorithm design was limited to major paradigms currently in use. Future analysis will consider

paradigms that require the composition of communication structures, especially as (1) machines become more supportive of programs with massive parallelism and (2) reuse of parallel code (for example, through libraries) is better understood.

More algorithms will exhibit dynamic behavior. A large class of algorithms have statically determinable communication patterns. Another growing class of algorithms — including algorithms of artificial intelligence — have more dynamic communication requirements. A natural extension to the work presented here is the description of statically limited, dynamic growth of process structures.

Canister-based communication will be extended. We have observed that many algorithms transmit messages along well defined paths. As parallel algorithms become more complex, so will communication. Manipulation of communication paths should be easily reflected in the specification of canister-based communication, including nesting and concatenation of itineraries.

Graph grammar formalisms will be more approachable. The formalism of aggregate rewriting graph grammars is motivated by the need to succinctly specify the topologies of regular communication graphs. As our understanding of the characteristics of parallel communication becomes more complete, graph grammar formalisms can be made more suitable for specifying communication structures.

Graph editors will become a central component of other tools. Graph grammars lead to concise descriptions of graph families. The information provided by this grammatical description may provide useful information in other stages of the programming process, for example, mapping, debugging, and animation. In addition, the graph editor concept may have applications in other areas, such as the description of neural networks, or the specification of dynamic changes in distributed systems.

References

[1] ABE, N., MIZUMOTO, M., TOYODA, J., AND TANAKA, K. Web grammars and several graphs. *Journal of Computer and Systems Sciences*, 7:37–65, 1973.

[2] AHO, A., HOPCROFT, J., AND ULLMAN, J. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[3] AKL, S. G. *Parallel Sorting Algorithms*. Academic Press, Orlando, Florida, 1985.

[4] ALLEN, R., BÄUMGARTNER, D., KENNEDY, K., AND PORTERFIELD, A. PTOOL: a semi-automatic parallel programming assistant. In *1986 International Conference on Parallel Processing*, pages 164–169, August 1986.

[5] ANNARATONE, M., ARNOULD, E., GROSS, T., KUNG, H. T., LAM, M., MENZILCIOGLU, O., AND WEBB, J. A. The Warp computer: architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, 1986.

[6] ATHAS, W. C. AND SEITZ, C. L. Multicomputers: message-passing concurrent computers. *Computer*, 9–24, August 1988.

[7] BAILEY, D. A. AND CUNY, J. E. Graph grammar based specification of interconnection structures for massively parallel coumptation. In *Proceedings of the Third International Workshop on Graph Grammars*, pages 73–85, Springer-Verlag, Berlin, December 1987.

[8] BAILEY, D. A. AND CUNY, J. E. Structural properties of communication structures generated by aggregate rewriting graph grammars. In *Allerton Conference on Communication, Control, and Computing*, pages 1234–1240, University of Illinois at Urbana-Champaign, September 1987.

[9] BAILEY, D. A. AND CUNY, J. E. An approach to programming process interconnection structures. In *Proceedings of Parallel Architectures and Languages Europe, vol. 2*, pages 112–123, Springer-Verlag, Berlin, June 1987.

[10] BAILEY, D. A. AND CUNY, J. E. *Graph Grammar Based Specification of Interconnection Structures*. Technical Report 87-23, University of Massachusetts at Amherst, March 1987.

[11] BAILEY, D. A. AND CUNY, J. E. Graph grammar based specification of interconnection structures for massively parallel computation. In *Graph-Grammars and Their Application to Computer Science, Third International Workshop*, pages 73–85, Springer-Verlag, Berlin, December 1986.

[12] BAILEY, D. A. AND CUNY, J. E. An efficient embedding of large trees in processor grids. In *1986 International Conference on Parallel Processing*, pages 819–822, August 1986.

[13] BAILEY, D. A. AND CUNY, J. E. *The Use of Shape Grammars in Processor Embeddings.* Technical Report A-86-23, University of Massachusetts at Amherst, July 1986.

[14] BARNES, G., BROWN, R., KATO, M., KUCK, D. J., SLOTNICK, D., AND STOKES, R. The ILLIAC IV computer. *IEEE Transactions on Computers*, 17(8):746–757, August 1968.

[15] BASKETT, F., HOWARD, J. H., AND MONTAGUE, J. T. Task communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23–31, November 1977.

[16] BATCHER, K. E. Design of a massively parallel processor. *IEEE Transactions on Computers*, C-29(9):1–9, 1980.

[17] BATES, P. C. *EBBA Modelling Tool a.k.a Event Definition Language.* Technical Report 87-35, University of Massachusetts at Amherst, April 1987.

[18] BATES, P. C. *Debugging Programs in a Distributed System.* Technical Report 86-05, University of Massachusetts at Amherst, January 1986.

[19] BENTLEY, J. L. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, April 1980.

[20] BENTLEY, J. L. AND KUNG, H. T. A tree machine for searching problems. In *1979 International Conference on Parallel Processing*, pages 257–266, August 1979.

[21] BERMAN, F. Experience with an automatic solution to the mapping problem. In *The Characteristics of Parallel Algorithms*, pages 307–334, MIT Press, Cambridge, Massachusetts, 1987.

[22] BERMAN, F. *Prep-P Guide.*

[23] BERMAN, F., CUNY, J., AND SNYDER, L. *Unifying Programming Support for Parallel Computers*, chapter 20. Pelnum Press, New York, 1988.

[24] BERMAN, F., GOODRICH, M., KOELBEL, C., W. J. ROBISON, I., AND SHOWELL, K. Prep-P: a mapping preprocessor for CHiP architectures. In *1985 International Conference on Parallel Processing*, pages 731–733, August 1985.

[25] BERMAN, F. AND SYNDER, L. On mapping parallel algorithms into parallel architectures. In *1984 International Conference on Parallel Processing*, pages 307–309, August 1984.

[26] BHATT, S. N., CHUNG, F. R. K., HONG, J., LEIGHTON, F. T., AND ROSENBERG, A. L. Optimal simulations of butterfly networks. In *Symposium on the Theory of Computing*, 1988.

[27] BOKAHRI, S. H. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic, Boston, 1987.

[28] BBN. *Butterfly Parallel Processor Overview.* Technical Report Report 6148, Version 1, BBN Laboratories, Inc., Cambridge, Massachusetts, March 1986.

[29] BROWN, M. H. *Algorithm Animation.* PhD thesis, Brown University, Providence, Rhode Island, 1987.

[30] BROWNE, J. C. Framework for formulation and analysis of parallel computation structures. *Parallel Computing*, 3(1):1–9, March 1986.

[31] BROWNE, J. C., TRIPATHI, A., FEDAK, S., ADIGA, A., AND KAPUR, R. A language for specification and programming of reconfigurable parallel computation structures. In *1982 International Conference on Parallel Processing*, pages 142–149, August 1982.

[32] BROWNING, S. *Hierarchically Organized Machines*, chapter 8.4. Addison-Wesley, Reading, Massachusetts, 1980.

[33] CAPPELLO, P. R. AND STEIGLITZ, K. Unifying VLSI array designs with geometric transformations. In *1984 International Conference on Parallel Processing*, pages 1–6, August 1984.

[34] CASTELLANI, I. AND MONTANARI, U. Graph grammars for distributed systems. In CLAUS, V., EHRIG, H., AND ROZENBERG, G., editors, *Graph-Grammars and Their Application to Computer Science and Biology*, pages 20–38, Springer-Verlag, Berlin, 1982.

[35] CHEN, M. C. *A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI.* Technical Report YALEU/DCS/RR-432, Yale University, Department of Computer Science, October 1985.

[36] CHEN, M. C. *The Generation of a Class of Multipliers: A Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI.* Technical Report YALEU/DCS/RR-442, Yale University, Department of Computer Science, December 1985.

[37] CLAUS, V., EHRIG, H., AND ROZENBERG, G. *Graph-Grammars and Their Application to Computer Science and Biology. Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1979.

[38] COUCH, A. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* PhD thesis, Tufts University, Medford, Massachusetts, 1988.

[39] CUNY, J. E., BAILEY, D. A., HAGERMAN, J. W., AND HOUGH, A. A. Simple simon programming environment: a status report. In *Allerton Conference on Communication, Control, and Computing*, pages 238–247, University of Illinois at Urbana-Champaign, September 1987.

[40] CUNY, J. E., BAILEY, D. A., HAGERMAN, J. W., AND HOUGH, A. A. *Simple Simon Programming Environment: A Status Report.* Technical Report 87-22, University of Massachusetts at Amherst, March 1987.

[41] CUNY, J. AND SNYDER, L. Compilation of data-driven programs for synchronous execution. In *Proceedings of the 1983 Annual Symposium on Principles of Programming Languages*, pages 197–202, 1983.

[42] CULIK II, K. AND LINDENMAYER, A. Parallel graph generating and graph recurrence systems for multicellular development. *Internation Journal of General Systems*, 3:53-66, 1976.

[43] CULIK II, K. AND WOOD, D. A mathematical investigation of propagating graph 0L systems. *Information and Control*, 43:50–82, 1979.

[44] DEGROOT, D. Expanding and contracting SW-banyan networks. In *1983 International Conference on Parallel Processing*, pages 20–24, August 1983.

[45] DEGROOT, D. Partitioning job structures for SW-banyan networks. In *1983 International Conference on Parallel Processing*, pages 106–113, August 1983.

[46] DEGROOT, D. *Mapping Computation Structures Onto SW-Banyan Networks.* PhD thesis, Department of Computer Science, University of Texas, Austin, 1981.

[47] DESHPANDE, S. R. AND JENEVEIN, R. M. Scalability of a binary tree on a hypercube. In *1986 International Conference on Parallel Processing*, pages 661–668, August 1986.

[48] DEWEY, D. AND PATERA, A. T. Geometry-defining processors for partial differential equations. 1986.

[49] DIETZ, H. AND KLAPPHOLZ, D. Refined C: a sequential language for parallel programming. In *1985 International Conference on Parallel Processing*, pages 442–449, August 1985.

[50] DOSHI, K. A. AND VARMAN, P. J. Optimal graph algorithms on a fixed-size linear array. *IEEE Transactions on Computers*, C-36(4), January 1987.

[51] EHRIG, H., PFENDER, M., AND SCHNEIDER, H. J. Graph grammars: an algebraic approach. In *14th Conference on Switching and Automata Theory*, pages 167–179, 1973.

[52] FALMANAND, S. E. AND HINTON, G. E. Connectionist architectures for artificial intelligence. *Computer*, 20(1):100–109, January 1987.

[53] FENG, T. A survey of interconnection networks. *Computer*, 14(12):12–27, December 1981.

[54] FINKEL, R. A. *Large-grain parallelism — Three case studies*, pages 21–63. *Scientific Computation Series*, MIT Press, Cambridge, Massachusetts, 1987.

[55] FISHBURN, J. P. AND FINKEL, R. A. Quotient networks. *IEEE Transactions on Computers*, C-31(4):288–295, April 1982.

[56] FISHER, J. A. The VLIW machine: a multiprocessor for compiling scientific code. *Computer*, 17(7):45–53, July 1984.

[57] FUJIMOTO, R. M. *SIMON: Simulator of Multicomputer Networks.* Technical Report UCB/CSD 83/140, UC Berkeley, Berkeley, August 1983.

[58] GAJSKI, D., KUCK, D., LAWRIE, D., AND SAMEH, A. *Cedar.* Technical Report UIUCDCS-R-83-1123, University of Illinois, Urbana, February 1983.

[59] GANNON, D. A note on pipelining a mesh connected multiprocessor for finite element problems by nested dissection. In *1980 International Conference on Parallel Processing*, pages 284–286, August 1980.

[60] GANNON, D. AND ROSENDALE, J. V. On the structure of parallelism in a highly concurrent PDE solver. *Journal of Parallel and Distributed Computing*, 3(1):106–135, March 1986.

[61] GANNON, D., SNYDER, L., AND ROSENDALE, J. V. Programming substructure computations for elliptic problems on a CHiP system. In *Proceedings of the Symposium on the Impact of New Computing Systems on Computational Mechanics, ACME*, pages 65–80, 1983.

[62] GEHANI, N. AND MCGETTRICK, A. D. *Concurrent Programming.* Addison-Wesley, Reading, Massachusetts, 1988.

[63] GELERNTER, D., CARRIERO, N., CHANDRAN, S., AND CHANG, S. Parallel programming in Linda. In *1985 International Conference on Parallel Processing*, pages 255–283, August 1985.

[64] GOKE, R. L. *Banyan Networks for Partitioning Multiprocessor Systems.* PhD thesis, University of Florida, 1976.

[65] GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The NYU Ultracomputer – designing and MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, August 1983.

[66] GOTTLIEB, A. AND SCHWARTZ, J. T. Networks and algorithms for very-large-scale parallel computation. *Computer*, 15(1):27–36, August 1982.

[67] GUIBAS, L. J., KUNG, H. T., AND THOMPSON, C. D. Direct VLSI implementation of combinatorial algorithms. In *Caltech Conference on VLSI*, pages 510–525, January 1979.

[68] HARARY, F. *Graph Theory.* Addison-Wesley, Reading, Massachusetts, 1969.

[69] HAYES, L. S., LAU, R. L., SIEWIOREK, D. P., AND MIZELL, D. W. A survey of highly parallel computing. *Computer*, 15(1):9–24, January 1982.

[70] HELLER, D. E. *Multiprocessor Simulation Program SIMON.* Technical Report, Shell Development Company, November 1984.

[71] HILLIS, W. D. *The Connection Machine.* MIT Press, 1985.

[72] HILLIS, W. D. AND GUY L. STEELE, J. Data parallel algorithms. *Communications of the ACM*, 29(12), December 1986.

[73] HO, C. AND JOHNSSON, S. L. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *1986 International Conference on Parallel Processing*, pages 640–648, August 1986.

[74] HOROWITZ, E. AND ZORAT, A. Divide-and-conquer for parallel processing. *IEEE Transactions on Computers*, C-32(6):582–585, June 1983.

[75] HOUGH, A. A. AND CUNY, J. E. Initial experiences with a pattern-oriented parallel debugger. In *ACM SIGPLAN and SGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[76] HOUGH, A. A. AND CUNY, J. E. Belvedere: prototype of a pattern-oriented debugger for highly parallel computation. In *1987 International Conference on Parallel Processing*, pages 735–738, 1987.

[77] IBARRA, O. H., PALIS, M., AND KIM, S. M. Designing systolic algorithms using sequential machines. In *Conference on Foundations of Computer Science*, pages 46–55, 1984.

[78] INMOS. *Occam Programming Manual.* INMOS, Ltd., Bristol, England, 1983.

[79] JAMIESON, L. H., GANNON, D. B., AND DOUGLASS, R. J. *The Characteristics of Parallel Algorithms. Scientific Computation Series*, MIT Press, Cambridge, Massachusetts, 1987.

[80] JAMIESON, L. H. Characterizing parallel algorithms. In *The Characteristics of Parallel Algorithms*, pages 65–100, MIT Press, Cambridge, Massachusetts, 1987.

[81] JANSSENS, D., ROZENBERG, G., AND VERRAEDT, R. On sequential and parallel node-rewriting graph grammars, II. *Computer Vision, Graphics, and Image Processing*, 23:295–312, 1983.

[82] JANSSENS, D., ROZENBERG, G., AND VERRAEDT, R. On sequential and parallel node-rewriting graph grammars. *Computer Graphics and Image Processing*, 18:279–304, 1982.

[83] JANSSENS, D. AND ROZENBERG, G. Graph grammars with neighbourhood-controlled embedding. *Theoretical Computer Science*, 21:55–74, 1982.

[84] JANSSENS, D. AND ROZENBERG, G. Graph grammars with node-label controlled rewriting and embedding. In CLAUS, V., EHRIG, H., AND ROZENBERG, G., editors, *Graph-Grammars and Their Application to Computer Science and Biology*, pages 186–205, Springer-Verlag, Berlin, 1982.

[85] JANSSENS, D. AND ROZENBERG, G. A characterization of context-free string languages by directed node-label controlled graph grammars. *Acta Informatica*, 16:63–85, 1981.

[86] JANSSENS, D. AND ROZENBERG, G. Decision problems for node label controlled graph grammars. *Journal of Computer and Systems Sciences*, 22:144–177, 1981.

[87] JANSSENS, D. AND ROZENBERG, G. On the structure of node-label-controlled graph languages. *Information Sciences*, 20:191–216, 1980.

[88] JANSSENS, D. AND ROZENBERG, G. Restrictions, extensions, and variations of NLC grammars. *Information Sciences*, 20:217–244, 1980.

[89] JORDAN, H. F. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3(2):93–110, May 1986.

[90] KAPAUAN, A., WANG, K., GANNON, D., CUNY, J., AND SNYDER, L. The Pringle: an experimental system for parallel algorithm and software testing. In *1984 International Conference on Parallel Processing*, pages 1–6, August 1984.

[91] KENNEDY, K. *Automatic Translation of Fortran Programs to Vector Form.* Technical Report 476-029-4, Rice University, Department of Mathematical Sciences, Houston, 1980.

[92] KEUHN, J. T. AND SIEGEL, H. J. Extensions to the C programming language for SIMD/MIMD parallelism. In *1985 International Conference on Parallel Processing*, pages 232–235, August 1985.

[93] KHOSHAFIAN, S. *Parallel Container Model for Data Intensive Applications.* Technical Report ACA-ST-030-88, MCC, Austin, January 1988.

[94] KORNFELD, W. A. Combinatorially implosive algorithms. *Communications of the ACM*, 25(10):734–8, October 1982.

[95] KUCK, D. ILLIAC IV software and application programming. *IEEE Transactions on Computers*, C-17(8):758–770, August 1968.

[96] KUCK, D., LAWRIE, D., CYTRON, R., SAMEH, A., AND GAJSKI, D. *The Architecture and Programming of the Cedar System.* Technical Report 21, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1983.

[97] KUNG, H. T. Let's design algorithms for VLSI systems. In *Caltech Conference on VLSI*, pages 510–525, January 1979.

[98] KUNG, H. T. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.

[99] KUNG, H. T. AND LEISERSON, C. Systolic arrays (for VLSI). In MEAD, C. AND CONWAY, L., editors, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

[100] KUNG, S. *VLSI Array Processors.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[101] KUNG, S., ARUN, K. S., GAL-EZER, R. J., AND RAO, B. Wavefront Array Processor: language, architecture, and applications. *IEEE Transactions on Computers*, C-31(11):1054–1066, November 1982.

[102] LAWRIE, D. H. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):99–108, December 1975.

[103] LAWRIE, D. H., LAYMAN, T., BAER, D., AND RANDAL, J. M. Glypnir – a programming language for ILLIAC IV. *Communications of the ACM*, 18(3):157–164, March 1975.

[104] LEBLANC, T. J. AND MELLOR-CRUMMEY, J. M. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[105] LEISERSON, C. E. Fat-trees: universal networks for hardware efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[106] LEVITAN, S. *Parallel Algorithms and Architectures: A Programmer's Perspective.* PhD thesis, University of Massachusetts, COINS, Amherst, Massachusetts, 1984.

[107] LI, G. AND WAH, B. W. The design of optimal systolic arrays. *IEEE Transactions on Computers*, C-34(1):66–77, January 1985.

[108] LI, H., WANG, C., AND LAVIN, M. Structured process: a new language attribute for better interaction of parallel architecture and algorithm. In *1985 International Conference on Parallel Processing*, pages 247–254, August 1985.

[109] LINT, B. AND AGERWALA, T. Communication issues in the design and analysis of parallel algorithms. *IEEE Transactions on Software Engineering*, SE-7(2):174–188, March 1981.

[110] LIPOVSKI, G. J. AND MALEK, M. *Parallel Computing.* Wiley-Interscience, New York, 1987.

[111] LISKOV, B. AND GUTTAG, J. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Massachusetts, 1986.

[112] MALONY, A. D., REED, D. A., AND MCGUIRE, P. MPF: a portable message passing facility for shared memory multiprocessors. In *1987 International Conference on Parallel Processing*, pages 739–741, August 1987.

[113] MEHROTRA, P. AND ROSENDALE, J. V. *The Blaze Language: A Parallel Language for Scientific Programming.* Technical Report 85-29, NASA, May 1985.

[114] MEAD, C. AND CONWAY, L. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Massachusetts, 1980.

[115] MEHROTRA, P. AND PRATT, T. Language concepts for distributed processing of large arrays. In *ACM Symposium on Principles of Distributed Computing*, pages 19–28, 1982.

[116] MILLER, R. AND STOUT, Q. F. Data movement techniques for the pyramid computer. *SIAM Journal of Computing*, 16(1):38–60, February 1987.

[117] MONTANARI, U. G. Separable graphs, planar graphs and web grammars. *Information and Control*, 16:243–267, 1970.

[118] NAGL, M. *On the Relation Between Graph Grammars and Graph L-systems*, pages 142–151. *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1977.

[119] NELSON, P. A. *Parallel Programming Paradigms.* PhD thesis, University of Washington, Seattle, 1987.

[120] NELSON, P. A. AND SNYDER, L. *Programming Paradigms for Non-shared Memory Parallel Computers*, pages 3–20. Scientific Computation Series, MIT Press, Cambridge, Massachusetts, 1987.

[121] NELSON, P. AND SNYDER, L. Programming solutions to the algorithm contraction problem. In *1986 International Conference on Parallel Processing*, pages 258–261, August 1986.

[122] NIX, R. Editing by example. In *Proceedings of Principles of Programming Languages*, pages 186–195, 1983.

[123] PADUA, D. A., KUCK, D. J., AND LAWRIE, D. H. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, 1980.

[124] PARKER, D. S. Notes on shuffle/exchange-type switching networks. *IEEE Transactions on Computers*, C-29(3):213–223, March 1980.

[125] PFISTER, G. F., BRANTLEY, W. C., GEORGE, D. A., HARVEY, S. L., KLEINFELDER, W. J., MCAULIFFE, K. P., MELTON, E. A., NORTON, V. A., AND WEISS, J. The IBM Research Parallel Processor Prototype (RP3): introduction and architecture. In *1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[126] PFISTER, G. F. AND NORTON, V. A. 'Hot spot' contention and combining in multistage interconnection networks. In *1985 International Conference on Parallel Processing*, pages 790–797, IEEE, August 1985.

[127] POWELL, M. L. AND MILLER, B. P. Process migration in DEMOS/MP. In *Proceedings of the Ninth Symposium on Operating System Principles*, pages 110–119, October 1983.

[128] PRATT, T. W. Pair grammars. *Journal of Computer and Systems Sciences*, 5:560–595, 1971.

[129] PREMKUMAR, U. V., KUPAR, R., MALEK, M., LIPOVSKI, G. J., AND HORNE, P. Design and implementation of the banyan network in TRAC. In *National Computer Conference*, pages 643–653, 1980.

[130] PREPARATA, F. P. AND VUILLEMIN, J. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 300–309, May 1981.

[131] QUINN, M. J. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, 1987.

[132] QUINTON, P. Automatic synthesis of systolic arrays from uniform recurrent equations. In *1984 International Symposium on Computer Architecture*, pages 208–214, 1984.

[133] RAMAKRISHNAN, I. V. AND BROWNE, J. C. A paradigm for the design of parallel algorithms with applications. *IEEE Transactions on Software Engineering*, SE-9(4):411–415, July 1983.

[134] RANADE, A. G., BHATT, S. N., AND JOHNSSON, S. L. *The Fluent Abstract Machine*. Technical Report YALEU/DCS/TR-573, Department of Computer Science, Yale University, January 1988.

[135] REED, D. A. AND FUJIMOTO, R. M. *Multicomputer Networks: Message-Based Parallel Processing*. *Scientific Computation Series*, MIT Press, Cambridge, Massachusetts, 1987.

[136] ROSEN, B. K. Deriving graphs by applying a production. *Acta Informatica*, 4:337–357, 1975.

[137] ROSENBERG, A. L., STOCKMEYER, L. J., AND SNYDER, L. Uniform data encodings. *Theoretical Computer Science*, 11:145–165, 1980.

[138] ROZENBERG, G. Dependence graphs. In *Proceedings of the Third International Workshop on Graph Grammars (to appear)*, December 1986.

[139] SCHNEIDER, H. J. AND EHRIG, H. Grammars on partial graphs. *Acta Informatica*, 6:297–316, 1976.

[140] SCHNEIDER, H. J. *Graph Grammars*, pages 314–331. *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, September 1977.

[141] SCHWAN, K. AND MATTHEWS, J. Graphical views of parallel programs. *SIGSOFT Software Engineering Notices*, 11(3):51–64, July 1986.

[142] SCHWARTZ, J. T. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):482–521, October 1980.

[143] SCOTT, M. L. Language support for loosely coupled distributed programs. *IEEE Transactions on Computers*, SE-13(1):88–103, January 1987.

[144] SEDGEWICK, R. *Algorithms*. Addison-Wesley, Reading, Massachusetts, 1984.

[145] SEITZ, C. L. The Cosmic Cube. *Communications of the ACM*, 28(1):22–32, January 1985.

[146] SEJNOWSKI, M. C., UPCHURCH, E. T., KAPUR, R. N., CHARLU, D. P. S., AND LIPOVSKI, G. J. An overview of the Texas Reconfigurable Array Computer. In *National Computer Conference*, pages 631–641, 1980.

[147] SIEGEL, H. J., MCMILLEN, R. J., AND JR., P. T. M. A survey of interconnection methods for reconfigurable processing systems. In *AFIPS*, pages 529–542, 1979.

[148] SIEGEL, H. J., SCHWEDERSKI, T., IV, N. J. D., AND KUEHN, J. T. PASM: a reconfigurable parallel system for image processing. *Computer Architecture News*, 4(12):7–19, September 1984.

[149] SLOTNICK, D., BORCK, W. C., AND MCREYNOLDS, R. C. The Solomon computer. In *National Computer Conference*, pages 97–107, 1962.

[150] SNYDER, L. An inquiry into the benefits of multigauge parallel computation. In *1985 International Conference on Parallel Processing*, pages 488–492, August 1985.

[151] SNYDER, L. A taxonomy of synchronous parallel machines. In *1988 International Conference on Parallel Processing*, August 1988.

[152] SNYDER, L. Parallel processing and the demise of the p-ram. February 1988.

[153] SNYDER, L. Parallel programming and the Poker programming environment. *Computer*, 17(7):27–37, July 1984.

[154] SNYDER, L. Introduction to the configurable highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[155] SNYDER, L., JAMISON, L. H., GANNON, D. B., AND SIEGEL, H. J. *Algorithmically Specialized Parallel Computers*. Academic Press, Orlando, Florida, 1985.

[156] SNYDER, L. AND SOCHA, D. Poker on the Cosmic Cube: the first retargetable parallel programming language and environment. In *1986 International Conference on Parallel Processing*, pages 628–635, August 1986.

[157] SOCHA, D., BAILEY, M. L., AND NOTKIN, D. Voyeur: graphical views of parallel programs. In *SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 206–215, May 1988.

[158] SONG, S. W. A highly concurrent tree machine for database applications. In *1980 International Conference on Parallel Processing*, pages 259-268, August 1980.

[159] STANFILL, C. AND KAHLE, B. Parallel free-text search on the connection machine system. *Communications of the ACM*, 29(12):1229–1239, December 1987.

[160] STINY, G. *Pictorial and Formal Aspects of Shape and Shape Grammars.* Birkhauser Verlag, Basel und Studgart, 1975.

[161] STONE, H. S. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, February 1971.

[162] STOUT, Q. F. Supporting divide-and-conquer algorithms for image processing. *Journal of Parallel and Distributed Computing*, 4(1):95–115, January 1987.

[163] THOMPSON, C. D. AND KUNG, H. T. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–71, April 1977.

[164] UHR, L. *Algorithm-Structured Computer Arrays and Networks.* Academic Press, Orlando, Florida, 1984.

[165] ULLMAN, J. D. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Maryland, 1984.

[166] ULLMAN, J. D. *Some Thoughts about Supercomputer Organization.* Technical Report STAN-CS-83-987, Stanford University, October 1983.

[167] VALIANT, L. AND BREBNER, G. Universal schemes for parallel computation. In *Symposium on the Theory of Computing*, pages 263–277, May 1981.

[168] WALTZ, D. L. Applications of the connection machine. *Computer*, 20(1):85–97, January 1987.

[169] WEEMS, C. C. *Image Processings with a Content Addressable Array Parallel Processor.* PhD thesis, University of Massachusetts, COINS, Amherst, Massachusetts, 1984.

[170] YOUN, H. Y. AND SINGH, A. D. On area efficient and fault tolerant tree embedding in VLSI. In *1987 International Conference on Parallel Processing*, pages 170–177, August 1987.