Iowa State University
Digital Repository

Retrospective Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

1989

Visualization of program performance on concurrent computers

Diane Thiede Rover
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons, and the Electrical and Electronics Commons

Recommended Citation

Rover, Diane Thiede, "Visualization of program performance on concurrent computers " (1989). *Retrospective Theses and Dissertations*. 9173.
https://lib.dr.iastate.edu/rtd/9173

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

المنارة للاستشارات

www.manaraa.com

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9014948

Visualization of program performance on concurrent computers

Rover, Diane Thiede, Ph.D.

Iowa State University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Visualization of program performance on concurrent computers

by

Diane Thiede Rover

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

Members of the Committee:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1989

Copyright © Diane Thiede Rover, 1989. All rights reserved.

TABLE OF CONTENTS

Page

CHAPTER I. INTRODUCTION 1

- Concurrent Computation 2
- Motivation 9
- Complex Systems 11
- Visualization 14
- Scope of this Work 16

CHAPTER II. RELATED WORK 18

- The Mapping Problem 18
- Performance Analysis Tools 22

CHAPTER III. PERFORMANCE MONITORING 31

- Evaluation Methods 31
- Monitoring Complex Systems 37
- Data Management 42
- Perspectives 45

CHAPTER IV. REPRESENTING PERFORMANCE 50

- Data Presentation 50
- Categories of Concurrent Computers 52

CHAPTER V. PICTURES OF PERFORMANCE 59

- Methodology 59
- Observable Parameters 63
  - Basic metrics 64
  - Derived metrics 66
- Graphics 93
  - Plots 94
  - Profiles 98

CHAPTER VI. PROTOTYPE IMPLEMENTATION 99

- Simulation 99
- Graphics Software 105
- Case Studies in Visualization 111
  - System configuration 111
  - Broadcast communication program 115
  - Collect communication program 137
  - Shift communication program 155
  - Divide-and-conquer quicksort program 161
  - One-dimensional wave equation program 169

CHAPTER VII. DISCUSSION AND CONCLUSIONS 206

Future Work 206
A Question of Dimension 209
Research Contributions 210

BIBLIOGRAPHY 213

ACKNOWLEDGEMENTS 222

LIST OF TABLES

		Page
Table 4.1.	Complexity categories for performance data presentation formats	57
Table 6.1.	Simulation results for the case studies	118
Table 6.2.	Key for global statistics	119
Table 6.3.	Broadcast routine. Selected global statistics for snapshot number 6 taken at 0.0072 seconds	120
Table 6.4.	Broadcast routine. Selected global statistics for snapshot number 10 taken at 0.0144 seconds	121
Table 6.5.	Key for local statistics	122
Table 6.6.	Broadcast routine. Selected local statistics for Processor 0, (x,y) = (0,0)	123
Table 6.7.	Broadcast routine. Selected local statistics for Processor 100, (x,y) = (7,4)	124
Table 6.8.	Collect routine. Selected global statistics for snapshot number 8 taken at 0.00315 seconds	144
Table 6.9.	Collect routine. Selected global statistics for snapshot number 15 taken at 0.0168 seconds	145
Table 6.10.	Collect routine. Selected local statistics for Processor 0, (x,y) = (0,0)	146
Table 6.11.	Collect routine. Selected local statistics for Processor 100, (x,y) = (7,4)	147
Table 6.12.	Quicksort program. Selected global statistics for snapshot number 10 taken at 0.02273 seconds	164
Table 6.13.	Quicksort program. Selected global statistics for snapshot number 15 taken at 0.0279 seconds	165
Table 6.14.	Quicksort program. Selected local statistics for Processor 0, (x,y) = (0,0)	166

Table 6.15. Quicksort program. Selected local statistics for Processor 100, (x,y) = (7,4) 167

Table 6.16. 1-D Wave program. Selected global statistics for snapshot number 4 taken at 0.01 seconds 189

Table 6.17. 1-D Wave program. Selected global statistics for snapshot number 12 taken at 0.03 seconds 190

Table 6.18. 1-D Wave program. Selected global statistics for snapshot number 23 taken at 0.052 seconds 191

Table 6.19. 1-D Wave program. Selected global statistics for snapshot number 36 taken at 0.09 seconds 192

Table 6.20. 1-D Wave program. Selected local statistics for Processor 0, (x,y) = (0,0) 193

Table 6.21. 1-D Wave program. Selected local statistics for Processor 100, (x,y) = (7,4) 194

LIST OF FIGURES

		Page
Figure 1.1.	A classification of computer systems based on the organization of data and control	4
Figure 1.2.	A distributed memory concurrent computer	6
Figure 2.1.	A systems approach to solving problems concurrently	20
Figure 3.1.	Three perspectives on system performance: program, architecture, and machine	47
Figure 5.1.	Two geometric graphs, in template form, for presenting performance data from a machine perspective: a dot plot and a cell plot	96
Figure 6.1.	Event-driven simulation and generation of event records	103
Figure 6.2.	Post-processing of an event trace	106
Figure 6.3.	Graphical interface of the visual analysis tool	108
Figure 6.4.	Mapping a three-dimensional (eight-node) hypercube onto a two-dimensional grid (gray code mapping)	113
Figure 6.5.	Assignment of processor addresses for an eight-dimensional (256-node) hypercube to locations in a two-dimensional (16x16) grid (gray code mapping)	114
Figure 6.6.	Basic operation of Broadcast on an eight-node hypercube	117
Figure 6.7.	Picture of performance (dither plot): Broadcast, ss#2 at 0.9 msec., cumulative traffic (bytes)	126
Figure 6.8.	Picture of performance (dither plot): Broadcast, ss#3 at 1.8 msec., cumulative traffic (bytes)	127
Figure 6.9.	Picture of performance (dither plot): Broadcast, ss#4 at 3.6 msec., cumulative traffic (bytes)	128

Figure 6.10. Picture of performance (dither plot): Broadcast, ss#5 at 5.4 msec., cumulative traffic (bytes) 129

Figure 6.11. Picture of performance (dither plot): Broadcast, ss#6 at 7.2 msec., cumulative traffic (bytes) 130

Figure 6.12. Picture of performance (dither plot): Broadcast, ss#7 at 9 msec., cumulative traffic (bytes) 131

Figure 6.13. Picture of performance (dither plot): Broadcast, ss#8 at 10.8 msec., cumulative traffic (bytes) 132

Figure 6.14. Picture of performance (dither plot): Broadcast, ss#9 at 12.6 msec., cumulative traffic (bytes) 133

Figure 6.15. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., cumulative traffic (bytes) 134

Figure 6.16. Picture of performance (dither plot): Broadcast, ss#11 at 16.2 msec., cumulative traffic (bytes) 135

Figure 6.17. Picture of performance (dither plot): Broadcast, ss#12 at 18 msec., cumulative traffic (bytes) 136

Figure 6.18. Picture of performance (3D plot): Broadcast, ss#10 at 14.4 msec., cumulative traffic (bytes) 138

Figure 6.19. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., cumulative communication time 139

Figure 6.20. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., cumulative wait time 140

Figure 6.21. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., processor activity (black: computing; gray: communicating; white: none) 141

Figure 6.22. Basic operation of Collect on an eight-node hypercube 142

Figure 6.23. Picture of performance (dither plot): Collect, ss#2 at 0.35 msec., cumulative traffic (bytes) 148

Figure 6.24. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., cumulative traffic (bytes) 149

Figure 6.25. Picture of performance (3D plot): Collect, ss#8 at 3.15 msec., cumulative traffic (bytes) 150

Figure 6.26. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., cumulative communication time 151

Figure 6.27. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., cumulative wait time 152

Figure 6.28. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., processor activity (black: computing; gray: communicating; white: none) 153

Figure 6.29. Picture of performance (dither plot): Collect, ss#15 at 16.8 msec., cumulative traffic (bytes) 154

Figure 6.30. Basic operation of Shift on an eight-node hypercube 156

Figure 6.31. Picture of performance (dither plot): Shift, ss#4 at 0.7 msec., cumulative wait time 157

Figure 6.32. Picture of performance (dither plot): Shift, ss#12 at 4.2 msec., processor activity (black: computing; gray: communicating; white: none) 158

Figure 6.33. Picture of performance (dither plot): Shift, ss#12 at 4.2 msec., cumulative communication time 159

Figure 6.34. Picture of performance (dither plot): Shift, ss#12 at 4.2 msec., cumulative wait time 160

Figure 6.35. Basic operation of Quicksort on an eight-node hypercube 162

Figure 6.36. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., cumulative work (operations) 170

Figure 6.37. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., cumulative computation time 171

Figure 6.38. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., cumulative communication time 172

Figure 6.39. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., processor activity (black: computing; gray: communicating; white: none) 173

Figure 6.40. Picture of performance (dot plot): Quicksort, at 23 msec., cumulative activity 174

Figure 6.41. Picture of performance (dot plot): Quicksort, at 23 msec., instantaneous activity 175

Figure 6.42. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., cumulative work (operations) 176

Figure 6.43. Picture of performance (3D plot): Quicksort, ss#15 at 27.9 msec., cumulative work (operations) 177

Figure 6.44. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., cumulative computation time 178

Figure 6.45. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., cumulative communication time 179

Figure 6.46. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., processor activity (black: computing; gray: communicating; white: none) 180

Figure 6.47. Picture of performance (dot plot): Quicksort, at 28 msec., cumulative activity 181

Figure 6.48.	Picture of performance (dot plot): Quicksort, at 28 msec., instantaneous activity	182
Figure 6.49.	Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, • : event	183
Figure 6.50.	Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : activity event	184
Figure 6.51.	Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : compute event	185
Figure 6.52.	Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : send event	186
Figure 6.53.	Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : receive event	187
Figure 6.54.	Picture of performance (dither plot): 1D Wave, ss#4 at 10 msec., processor activity (black: computing; gray: communicating; white: none)	195
Figure 6.55.	Picture of performance (dither plot): 1D Wave, ss#4 at 10 msec., cumulative communication time	196
Figure 6.56.	Picture of performance (dither plot): 1D Wave, ss#12 at 30 msec., processor activity (black: computing; gray: communicating; white: none)	198
Figure 6.57.	Picture of performance (dither plot): 1D Wave, ss#12 at 30 msec., cumulative computation time	199
Figure 6.58.	Picture of performance (dither plot): 1D Wave, ss#12 at 30 msec., cumulative communication time	200
Figure 6.59.	Picture of performance (dither plot): 1D Wave, ss#23 at 52 msec., processor activity (black: computing; gray: communicating; white: none)	201

Figure 6.60. Picture of performance (dither plot): 1D Wave, ss#23 at 52 msec., cumulative computation time 202

Figure 6.61. Picture of performance (dither plot): 1D Wave, ss#23 at 52 msec., cumulative communication time 203

Figure 6.62. Picture of performance (dither plot): 1D Wave, ss#36 at 90 msec., cumulative computation time 204

Figure 6.63. Picture of performance (dither plot): 1D Wave, ss#36 at 90 msec., cumulative communication time 205

CHAPTER I.

INTRODUCTION

> However, if I had waited long enough I probably would never have written anything at all since there is a tendency when you really begin to learn something about a thing not to want to write about it but rather to keep on learning about it always and at no time, unless you are very egotistical, which, of course, accounts for many books, will you be able to say: now I know all about this and will write about it. Certainly I do not say that now; every year I know there is more to learn, but I know some things which may be interesting now ... and I might as well write what I know about them now.
>
> Ernest Hemingway, from Death in the Afternoon

Many basic ideas and problems concerning computers and programming have been around since the 1820s and 1830s when Charles Babbage designed his Analytical Engine and one of his colleagues, Lady Lovelace (born Ada Augusta Byron), developed her own programming language. The Analytical Engine and other early mechanical computing machines evolved into the electronic digital computers that were first introduced in the 1930s and became the basis for present computers. Throughout this history and especially over the past fifty years, cost and performance have been important design issues. For a fixed cost, the designer typically wants the fastest machine possible. There are several means of achieving this end, including choosing a simple organization with very fast parts or a more complex organization with slower parts [Kuck, 1978].

The first choice, using fast parts, has met with success so far. The dramatic progress in microelectronics over the

past twenty-five years has led to faster device technologies and yielded rapid growth in computer performance. However, the three basic functions of switching, storage, and communication that are required in computing systems are beginning to approach fundamental physical limits [Seitz and Matisoo, 1984]. Thus, the second choice, using many slower parts, is becoming more important. This chapter discusses some implications of that choice.

Concurrent Computation

How can a complex organization with many slow parts, or processors, result in a fast machine? It is not simply that computers with more parts should be able to solve larger problems in less time. Rather, it depends on the nature of the parts and how we structure and control the parts.

Using many slow parts is the premise of parallel or concurrent computing. A spectrum of designs identifies the possibilities for exploiting the parallelism or concurrency among the many parts. Though many spectrums can be defined based on different criteria, two are mentioned here and will be referenced in later chapters. Within one spectrum, there are three regions based on the number and complexity of the processors. At one extreme are simple, bit-serial processors. Although any one of these processors is of little value by itself, the aggregate computing power can be large when many are coupled together. This approach can be likened to a large colony of termites devouring a log. At the opposite extreme are machines that use a small number of powerful processors. Each processor is based on sophisticated pipelining and is

built using the fastest available circuit technology. Continuing our analogy, this approach is similar to a few woodsmen with chain saws. The third, intermediate approach combines a large number of microprocessors. This is analogous to a small army of hungry beavers [Reed and Fujimoto, 1987]. The third approach, as discussed below, is most relevant to later chapters.

Within another spectrum, there are four classes based on the organization of data and the organization of control (or instruction execution). In a centralized organization, data or control resides in only one part of the computer: data in a shared memory and control in a designated processor. In a distributed organization, data or control is local to each part: data in a local memory and control in each processor. Figure 1.1 illustrates the four possible combinations. Note that centralized data refers to the shared memory model, and distributed data, the message passing (or distributed memory) model. Also, centralized control refers to synchronous (or lockstep) execution, and distributed control, asynchronous execution. The four classes consist of the following organizations: (1) centralized control and centralized data; (2) centralized control and distributed data; (3) distributed control and centralized data; and (4) distributed control and distributed data. This is roughly similar to Flynn's classification based on instruction streams and data streams, corresponding, respectively, to SISD, SIMD, MISD, and MIMD [Flynn, 1966]. The fourth class, as discussed below, is most relevant to later chapters.

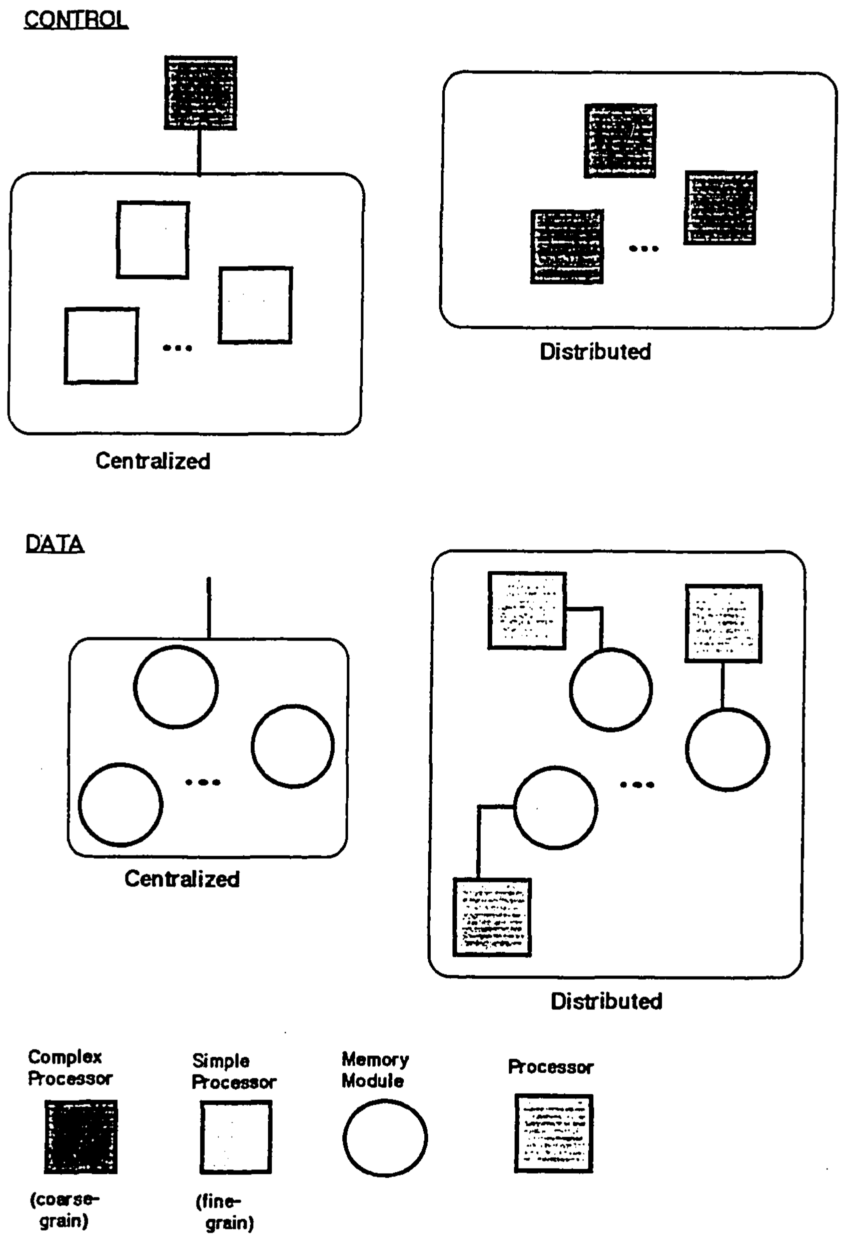


Figure 1.1. A classification of computer systems based on the organization of data and control

A note on terminology may be helpful at this point. The terms "concurrent" and "parallel" are often used interchangeably in the literature. Although a particular usage of the terms is evolving, there is no generally accepted distinction between the two terms. In a general context, we may interpret them to have the same meaning: a computer system is a concurrent or parallel one if it has more than one processing element, the processing elements are interconnected, and a collection of processing elements work together to solve a problem. However, the term "concurrent" sometimes denotes a computer system with more or less autonomous processing elements each having its own local memory and communicating via message passing. This is in contrast to systems with processing elements that operate in lockstep or that communicate using shared memory. Given this distinction, the term "concurrent" is more appropriate for our purposes. Thus, though we occasionally use both terms, parallel should be interpreted more generally and concurrent, more specifically.

A concurrent computer involves the collective and simultaneous interaction of many parts engaged in computation and communication activities. Figure 1.2 illustrates a computer system representative of the class of distributed memory concurrent computers. Coordination and cooperation are critical. Concurrent architectures and algorithms are the key to efficiently orchestrating the activities. The objective is to organize the interactions among the parts so that computations are performed concurrently and communication occurs locally within the concurrent computer. We will focus on a family of concurrent computers called multicomputers.

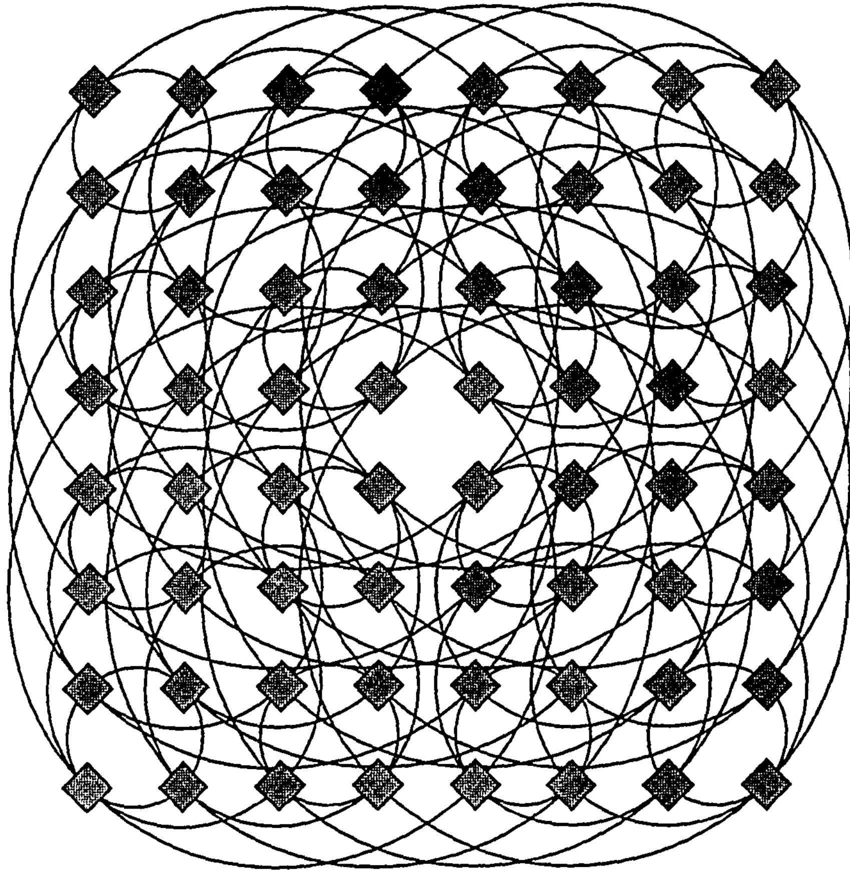


Figure 1.2. A distributed memory concurrent computer

Multicomputers consist of a large number, possibly hundreds or thousands, of nodes connected in some fixed topology or network. The nodes asynchronously cooperate via message passing to execute the tasks of parallel programs. Each network node, fabricated as a small number of VLSI (very large scale integration) chips, contains a processor, a local memory, a communication controller capable of routing messages without delaying the processor, and a small number of direct connections to other nodes. Specialized co-processors for floating-point, graphics, or secondary storage operations may also be included on a node. An important feature is that a multicomputer can be implemented using simple building blocks for the computation and communication components of a node [Reed and Fujimoto, 1987].

Application programs must be decomposed into concurrently executing tasks. The tasks may be small, medium, or large in size, and the multicomputer is termed a fine-grain, medium-grain, or large-grain (also coarse-grain) machine, respectively. Task size (for a program) may be measured as the amount of computation between task interactions, and grain size (for a multicomputer) loosely describes the node size or complexity. There are implementations of programming languages, models of computation, and architectures corresponding to each size. For example, the Actors model of concurrent computation [Agha, 1986] has been applied to both fine-grain and medium-grain configurations via the Concurrent Smalltalk [Dally, 1987] and the Cantor [Athas and Seitz, 1988] programming environments, respectively. The Large-Grain Data Flow (LGDF) model [Babb and DiNucci, 1987] has been applied to large-grain configurations, including the conventional

environment of Fortran on the Cray X/MP.

The idea of multicomputers is not new. Arthur Burks, an early pioneer in computing and colleague of John von Neumann, has suggested an architecture called "programmable computer structures:" a typical cell would hold a tiny computer which would store, process, and/or communicate information; and which would also control its own activities and regulate the passage of information through its own territory [Burks, 1981]. However, the interest in multicomputers has recently grown because improvements in technology have made them viable alternatives to other high performance computer systems. VLSI technology, namely powerful microprocessors and inexpensive memory, makes it both technically and economically feasible to construct multicomputers with many computing nodes. Although multicomputers have been the subject of numerous research projects since the 1970s, the idea remained unexploited until the construction and demonstration of the Cosmic Cube at Caltech in 1983 [Seitz, 1985]. The Cosmic Cube consists of a collection of nodes interconnected in a hypercube topology, one member of the multicomputer family of topologies. The computer system shown in Figure 1.2 is configured as a six-dimensional (64-processor) binary hypercube. Within the Cosmic Cube, each node includes a pair of Intel 8086/8087 processor chips, local memory, and a set of communication links. Following the success of the Cosmic Cube, four companies (namely, Intel, Ametek, Ncube, and Floating Point Systems) began producing commercial multicomputers configured as hypercubes.

Motivation

A research project often has its roots in some identifiable incident, observation, or thought. That origin may become the motivation for defining, clarifying, and attacking a problem. A chord is struck within the researcher that signals that a challenge awaits, that something interesting, exciting, and worthwhile needs investigating. This project has its roots in a couple of pieces of technical literature. In recollecting the origins, we choose to provide excerpts rather than merely summarize relevant passages. The ideas noted here were primary influences in this work, however we should mention that these were just a starting point. These ideas led to the discovery of many others, all of which influenced the direction of this work. Admittedly, this work could have taken several different directions depending on which ideas were emphasized.

The initial motivation for this work stems from a chapter in The Connection Machine, a book by Daniel Hillis, entitled "New Computer Architectures and Their Relationship to Physics or, Why Computer Science is No Good" (which was reprinted from [Hillis, 1982]). Hillis says that "there is beginning to be a forest to see through the trees." The phrase refers to the notion that computer systems are becoming large enough to exhibit the kind of simple, continuous behavior that we are accustomed to in physics, large enough that the behavior of the system can no longer be dominated by the behavior of any single component [Hillis, 1985]. Despite offering merely interesting insights, this chapter seems to open the door to a world of discovery, especially in the area of perceiving,

viewing, and understanding large computer systems.

Ivan Sutherland and Carver Mead also explore the relationship between computers and computer science in [Sutherland and Mead, 1977]. Excerpts from this paper include:

> Computer science has grown up in an era of computer technologies in which wires were cheap and switching elements were expensive. Integrated circuit technology reverses the cost situation ... . As we learn to understand the changed relative costs of logic and wiring and to take advantage of the possibilities inherent in large-scale integration we can expect a real revolution in computation, not only in the forms of computing machines but also in the theories on which their design and use are founded. ... Computer science as it is practiced today is based almost entirely on mathematical reasoning. It is concerned with the logical operations that take place in computing devices. It touches only lightly on the necessity to distribute logic devices in space, a necessity that forces one to provide communication paths between them. Computer science as it is practiced today has little to say about how the physical limitations to such communications bound the complexity of the computing tasks a physically realizable computer can accomplish. [Sutherland and Mead, 1977]

The paper proceeds to discuss the effects of communication, the importance of regularity in computing structures, the advent of distributed memory concurrent computers, and the goal of matching the complexities of problems to the simple patterns of communication in actual machines. Several later remarks summarize their thoughts:

> The challenge in designing or using a parallel processor ... lies in discovering ways in which simple patterns of communication within the processor can be made to match the communication tasks inherent in the problem being solved. ... We believe that just as an important part of today's computer science concerns itself with sequences of instructions in time, so an important aspect of computer science in the future will be the study of sets of communications distributed in space. [Sutherland and Mead, 1977]

Both of these treatises emphasize that computation and communication activities occur in time and space within the machine and that we need to start thinking about computing, especially large-scale parallel computing, with that in mind if we are to realize the potential power of future computer systems. These are challenging and stimulating ideas, and they are a premise for much of the work described in the chapters that follow. To facilitate thinking about the temporal and spatial behavior of parallel computation, we feel it is critical to have methods and tools that give us an appropriate view of system performance. Thus, we sought to create a performance "picture" that would illustrate program behavior within the time and space domains of a concurrent computer.

Complex Systems

Two concepts underlie the work described in this thesis. One is complex systems, and the other is visualization. We discuss complex systems in this section and then turn to visualization in the next section. However, we refer to the concepts again in later chapters, since both are common threads running throughout this work.

Complex systems come in many forms, including a colony of ants, a hive of bees, a society of people, a cluster of stars, the brain and its neurons, a chip with transistors, and a spreadsheet of cells, to name a few ([Fox et al., 1988] gives a longer, more detailed list). Each of these systems is characterized by a large collection of entities or members that are connected in some way. Because a concurrent computer

involves the collective and simultaneous interaction of many elements engaged in computation and communication activities across a network, it also is a complex system. The behavior and properties of other complex systems may enhance our understanding of concurrent computer systems. Several researchers have studied the relationship between complex systems and concurrent computers, and their writings include [Fox et al., 1988], [Kleinrock, 1985], [Wolfram, 1984], [Gelernter, 1987], and [Snodgrass, 1988]. We mention some contributions here and will refer to others in later chapters as well.

Gelernter compares honeybees and processes. He writes that like the bees maintaining a hive -- individually feeble agents working in concert -- a parallel program can bring large amounts of computing power to bear on a problem by establishing multiple processes or loci of activity. The bees coordinate their activities through visual and chemical signals; similarly, processes in a parallel program must communicate to work together. This example and others describe loosely-coupled systems that achieve a common goal with distributed control. Stated another way, each is a system in which loosely-coupled, self-organizing automatons demonstrate expedient behavior [Kleinrock, 1985].

We can identify several general parameters and properties of complex systems. These include:

- size
- structure (or topology)
- dimension
- granularity
- pattern of communication
- balance
- hierarchy of levels
- self-similarity (or scale invariance)

Size is the number of members in the system. If the system is extensible, then it can start at a given size and later be expanded to a larger size without adversely, or unreasonably, affecting the performance of the system. Structure is the connectivity (that is, nature of the connections) among the members. Topology can be static or dynamic over the life of the system. A dynamic structure is sometimes termed configurable (or reconfigurable). Dimension is the number of connections from a member to its neighboring members. Granularity reflects the amount of work to be done by a member. This can be a fixed or changing amount as the system progresses through time. Pattern of communication describes the spatial interaction among members. This may depend on the activities of the members and thus changes over time. Balance refers to a good, orderly mix of work and communication by all members. This is important to the performance of the system. A hierarchy of levels and self-similarity among levels means that members can be organized into, say, classes, and that the process by which work is done is more or less the same regardless of the level at which it happens. Only the scale is different. Hierarchy is useful to reduce the apparent complexity of a system since it supports selectively hiding or exposing the detailed workings of a system's members.

A fascinating account of complex systems is given in [Gleick, 1987]. In his book, James Gleick chronicles a set of beliefs about complexity that was once shared by scientists and an alternative set of ideas that is gaining acceptance. Three early beliefs were: (1) simple systems behave in simple ways; (2) complex behavior implies complex causes; and (3)

different systems behave differently. However, over the past twenty years, ideas have changed: (1) simple systems give rise to complex behavior; (2) complex systems give rise to simple behavior; and (3) the laws of complexity hold universally, regardless of the details of a system's constituent parts. So, understanding complexity in one system may lend insight into understanding complexity in another system. Also, there is reason to believe that simplicity exists at some level in the system.

Visualization

Visualization is an area of computer graphics that consists of techniques and tools that allow data to be observed and manipulated in a geometrical, rather than numerical or textual, format. The visualization field can be divided into three broad areas:

(1) visualization in scientific computing (ViSC),

(2) visual programming, and

(3) program visualization.

Visualization in scientific computing is the visualization of application program results (or output data). It refers to the animation of data such as that produced by supercomputer simulations, satellites, and measuring devices used in astronomy, meteorology, geology, and medicine. Visual programming, or graphical programming, is the specification of programs in a notation using two or more dimensions, as by flowcharts, graphs, diagrams, or icons (see [Shu, 1988]). Program visualization, also called algorithm animation, uses images to represent some aspect of a program's execution. The

work described in this paper falls predominantly in this latter area.

In general, the utility of visualization in scientific, engineering, and business applications is based on the ability of the human eye/brain combination to perceive and comprehend visual images orders of magnitude faster than numbers (or text) only. By using a computer to visualize data, we can absorb huge amounts of information. For instance, in a three-dimensional color representation on a higher-resolution graphics display, one displayed image can represent as many as ten million numbers. This global picture of the data gives researchers the ability to see simultaneously all of the information that otherwise might have to be printed on reams of paper. It allows researchers to discover relationships and invariants in collections of data. An important feature of many visualization systems is color, where typically the largest data values are represented by red and the smallest by blue. Color-coded data are useful to identify patterns and anomalies. Two additional important features include (1) interaction, exploring and manipulating the data during presentation, and (2) animation, displaying a series of images that illustrate relationships over time.

Graphics software tools that directly generate two- and three-dimensional pictures representing tables of data are becoming increasingly available, both commercially and in the public domain. Examples include MacSpin, DataScope, and Image, described in [Peltz, 1989] and [Schuster, 1989]. We discuss these tools and their relation to our work in Chapter VI. The tables of data to be analyzed by graphics tools can describe the behavior or state of any complex system. If the

system under study is a concurrent computer, then data collected to measure its performance (often thousands to millions of bytes) may be compiled into a tabular format. These tables can be transformed into pictures that offer insights into the development of algorithms, architectures, and machines.

Scope of this Work

The primary purpose of this work is to answer the following question with respect to Figure 1.2: How can we evaluate program performance on this computer system? The basis for our answer has its origins in the two concepts, complex systems and visualization, and in the application of these concepts to studying multicomputer systems. Of the many possible paths of study, three are pursued to varying extents: (1) monitoring, or measuring, program performance (i.e., data collection), (2) visualization of program performance (i.e., data presentation), and (3) development of performance models.

In answer to the stated question, we present a unique graphical approach to performance measurement of (possibly large) concurrent computer systems. Our approach attempts to present a performance "picture" that will offer insight into the development of concurrent algorithms, architectures, and machines. The key elements of the approach are listed.

(1) Observe and measure performance via instrumented execution of programs.

(2) Analyze and reduce performance data via appropriate techniques.

(3) Calculate aggregate measures of system behavior.

(4) Visually display program performance via a computer graphics format that illustrates computation and communication activities in time and space within the machine.

Chapter I has provided background and introductory information to establish a context for the remaining chapters. Chapter II describes related work in two areas: performance analysis tools and mapping algorithms onto architectures to achieve optimal performance. Chapter III discusses program monitoring (via instrumentation) as a method of performance evaluation, examines critical issues in measuring performance on concurrent computer systems, and presents perspectives for observing system performance. Chapter IV discusses several formats for presenting performance data and the appropriateness of particular formats for representing the performance of particular computer systems. Chapter V presents our approach to representing program performance, including a description of the method, definitions of measured parameters and calculated statistics, and specifications of the graphical formats. Chapter VI describes a prototype implementation of the approach and presents simulation results from several case studies. Finally, Chapter VII discusses yet unresolved issues relating to system dimension, outlines future work, and presents the contributions of this work.

CHAPTER II.

RELATED WORK

In the preceding chapter, some of the work being done in related areas has already been introduced. In this chapter, we review work being done in two other areas: the mapping problem and performance analysis tools. A discussion of the mapping problem is included because of its importance to the performance of concurrent computers and because of the potential contribution of this work toward solving the mapping problem. The majority of this chapter pertains to the latter area, performance analysis tools, and tools that implement visualization techniques are highlighted.

The Mapping Problem

A systems approach for developing effective concurrent computers emphasizes matching an algorithm, or class of algorithms, with an architecture. The essential points in such a design paradigm include: identifying parallel applications, developing concurrent algorithms, defining concurrent models of computation, specifying expressive concurrent programming languages, creating a concurrent architecture, developing efficient operating system and support software, and constructing an effective concurrent computer. These activities are illustrated in Figure 2.1. There should be a good match between each pair of levels, as

indicated by the adjoining arcs. Additionally, two objectives of this design process are expressiveness and efficiency. Expressiveness refers to the ease with which a program can be understood, and efficiency, the ease with which the actions implied in a program can be executed by the computer. Informally, the mapping problem involves devising a good match between levels to achieve optimal system performance.

One way to express the mapping problem is in terms of complex systems [Fox et al., 1988]. Concurrent computing can be viewed as a mapping between one complex system, the computer, and another complex system, the problem. An aim is to determine which complex computers are best applied to the various classes of complex problems. Fox suggests that we find general results of the form: "Complex computers with system parameters and properties of such and such values can be used to compute problems with this and that values for its respective defining parameters." Two fundamental hardware parameters for concurrent computers are the time to communicate a number between two nodes and the time to perform a calculation. Informally, the communication overhead reflects the amount of time a node spends conversing with its neighbors instead of doing productive work on its own. It is a function of the ratio of communication and calculation times and represents the fraction of the total run time spent on communication. A small ratio implies a better fit between problem and computer and is needed for good performance of a concurrent computer and algorithm. A similar analysis in terms of overhead incurred per unit of computation is presented by Stone [Stone, 1987]. Performance is shown to depend on the length of a runtime quantum relative to the

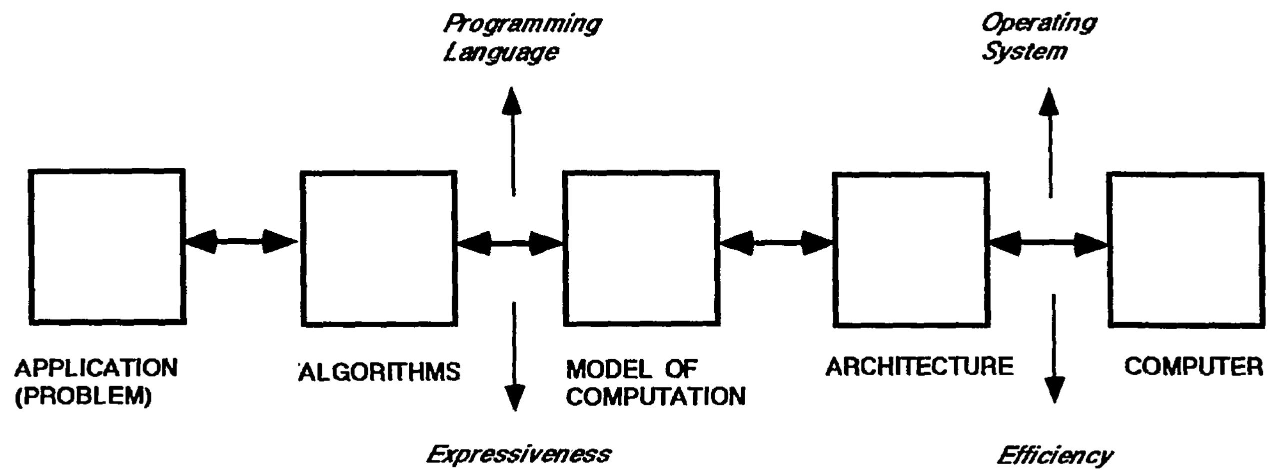


Figure 2.1. A systems approach to solving problems concurrently

length of communications overhead produced by that quantum. In each case, the ratio is used to balance concurrency and communication and thus achieve optimum performance. Seeking a balance between concurrency and communication is an approach to solving the mapping problem.

A graph-theoretic treatment of the mapping problem is given by Bokhari [Bokhari, 1987]. He calls it the assignment problem and defines a central problem and a variant. The central problem is that of assigning the modules of a program to the processors of a multicomputer. A module may contain either code or data and may communicate with other modules. The objective is to find an assignment that minimizes the total cost of executing the program. A variant of the central problem occurs when all processors execute the same program, but on different portions of a large domain, or data set. In this case, the domain is partitioned and each subdomain is assigned to a separate processor. So the first problem is that of assigning the nodes of a computation graph over the nodes of a given multicomputer system in order to minimize communication overhead. The second problem is that of partitioning the domain over the processors of a multicomputer system so that each processor has nearly the same computational load allocated to it.

A software system called Prep-P is being developed as a tool to help automate a solution to the mapping problem for multicomputers [Berman, 1987]. The problem is viewed in a context similar to that described by Bokhari (above), and it involves making an assignment of processes to processors. Prep-P is targeted at machines based on either a fixed or configurable communication network between processors and the

computation is modeled as a network of communicating processes. The topology of the process communication graph may not be a natural subgraph of the topology of the processor interconnection graph or, more commonly, the process communication graph may be much larger. The Prep-P system implements a particular mapping strategy that starts with a graph description of the algorithm and finishes with code that executes the algorithm on a parallel architecture simulator (the Poker simulator, described in the next section).

In most, if not all, formulations of the mapping problem, we can define metrics to evaluate, or measure, the quality of the mapping based on certain parameters of the system. Of course, we can define many types of metrics, as we will see in later chapters. To calculate the metrics, a mechanism is needed to extract values for the parameters of interest relating to system performance. This need has resulted in the development of numerous performance analysis tools.

Performance Analysis Tools

A number of projects have investigated, at least in part, the problem of representing parallel program performance. These projects have contributed to the general knowledge on multicomputers and analysis tools. This project has matured because of their contributions. Several projects have provided information about actual concurrent computer systems, which we are currently lacking. Two of these projects, Seecube and Hyperview, are discussed first. These are closely related to our work and were studied in depth. We then discuss PARET, PAW, Poker, B-Hive, PIE, Balsa, IPPM, PM,

LTRAMS, and Victor. Note that the names used may denote the project, the tool, or both. The projects are at various stages of development. A few were initiated only within the last couple of years, and none existed a decade ago. At the present time, there is a rapidly growing interest in tools to support performance analysis of multicomputers. Tools that use visualization techniques to represent program performance are especially relevant and are highlighted here.

Seecube, one of the first tools of its kind, allows the programmer of a hypercube computer, originally the Intel iPSC computer and currently the NCUBE computer, to visualize communications within a parallel program [Couch, 1988]. It uses post-processing of records of local events from each processor to reconstruct the global state of the computer at any time during a computation. There are several graphical representations of the state data, including: 3-cubes in space, 3-d Karnaugh map, linear plot, log butterfly plot, ordered circle, gray code circle, and Pascal triangle. These representations are different ways to organize n-dimensional plots in a plane, and they support up to about six dimensions (or sixty-four processors). There are three parts within Seecube: the Data Collector, the Resolver, and the Sequencer. The Data Collector is implemented as a library of communication routines on the hypercube that invisibly (as far as possible) store diagnostic event traces in local memory on each processor. At the end of computation, these traces are collected from each hypercube node processor and stored on the host processor. The Resolver cross-references these traces by matching sends with corresponding receives and sorts the traces into a single global trace for the entire hypercube.

Then the Sequencer graphically and dynamically displays the results of the Resolver.

Seecube is now part of a larger tool called Triplex, a collection of software tools which aid the programmer in implementing algorithms on the NCUBE multiprocessor. The tools address the problem of understanding the behavior of parallel programs in terms of both correctness and performance. Triplex has three components: the Simplex operating system for the NCUBE, the Commplex networking package for communication with the NCUBE from Sun workstations, and the Seeplex color graphics program for viewing depictions of program execution. Simplex supports the development of tools for real-time and offline debugging and performance monitoring. When Simplex is loaded, it synchronizes the local clocks on all nodes and maintains this synchrony. It provides out-of-band transmission (higher priority and reliability than other data transmissions) of system monitoring data of two kinds: (1) summary statistics, which summarize computational conditions at each instant in time, and (2) event statistics, which record histories of significant events.

An event happens locally within a processing node. The interaction of an outside monitor program (such as Seeplex) and the event statistics software embedded in Simplex involves two activities: (1) the selection of "collection points" to be enabled, and (2) the sending out of the stored "notes" upon request. The outside monitor program interacts with Simplex's logging capabilities through "parameters". A parameter corresponds to a set of collection points that are enabled as a group. Summary statistics are collected continuously and

reported only upon request. The interaction of the outside monitor program and the summary statistics monitor embedded in Simplex involves two activities: (1) initiation of reporting, including selection of parameters to be collected (done once only) and (2) polling for data (done repeatedly). More details are available in [Krumme et al., 1989], [Krumme, 1989], and [Couch, 1989].

Tapestry is a project at the University of Illinois (at Urbana-Champaign) that provides an experimental environment where different computer architectures can be matched to the computation requirements of an application's constituent algorithms [Campbell and Reed, 1988]. The research includes performance measurement, evaluation, and visualization. A collection of performance visualization tools called HyperView supports dynamic performance displays for viewing event traces. Included in the set of display views are: dials, bar charts, LEDs, Kiviat diagrams, matrix views, and general graphs. The inclusion of visualization tools is based on reasoning that is nearly identical to the motivation for much of the work described in this thesis:

> Parallel computer systems are among the most complex of [our] creations, making satisfactory performance characterization difficult. Despite this complexity, there is a strong tendency to quantify parallel system performance using a single metric. A complete characterization requires both static and dynamic characterizations. Static or average behavior analysis may mask transients that dramatically alter system performance. The importance of dynamic, visual scientific data presentation has only recently been recognized. Large, complex parallel systems pose equally vexing performance interpretation problems. Data from hardware and software performance monitors must be presented in ways that emphasize important events while suppressing irrelevant details. [Campbell and Reed, 1988]

Hyperview dynamically displays architectural and system activity via numerous system views. Detailed performance measurements also are provided via standard statistical displays. It was inspired by Seecube, and many displays were borrowed from Seecube. Whereas Seecube was built for the SunView window environment, Hyperview is based on the X window environment. Hyperview contains three cooperating modules: (1) data capture, (2) state analysis, and (3) visualization. A hardware monitor for the iPSC/2 hypercube is integrated with the performance visualization system (recall, Seecube uses a software monitor for the NCUBE). Tapestry researchers feel that the hardware support is crucial to the capture of detailed performance data. More information is available in [Rudolph and Reed, 1989], [Malony, 1989], and [Reed, 1989].

PARET is the Parallel Architecture Research and Evaluation Tool [Nichols and Edmark, 1988]. It is a software package that provides a multicomputer system laboratory for studying: (1) the interaction of algorithms and architectures; (2) the effects of varying physical resources on system performance; and (3) alternate mapping, scheduling, and routing strategies, both static and dynamic. Through simulation, users exercise multicomputer models and study performance in an interactive and animated environment. Algorithms and architectures are displayed as directed flow graphs. It also provides both runtime and summary statistics.

PAW is the Performance Analysis Workstation for queueing networks [Melamed and Morris, 1985]. The network is animated during simulation, and the user can control simulation parameters. A simple graphical representation of a network of arbitrary topology shows message passing by moving symbols

from one box to another on a graphical display.

The Poker system was originally planned to emulate a very specific architecture, CHiP (the Configurable Highly Parallel Computer) [Snyder, 1982], although it has been extended to the Cosmic Cube [Snyder 1984]. Poker has separate windows that allow the user to focus on different multicomputer functions such as setting the switches to create a particular interconnection, assigning processes to processors, and writing the code for a particular process. Though it is not directly related to performance analysis, it provides a good view of the multicomputer.

The B-Hive project measures static properties of processor interconnections to select the best candidate topologies to execute an application program [Agrawal et al., 1986]. To select the best architecture for a particular application and initiate a simulation of the execution, the directed flow graphs representing parallel software are allocated to undirected graphs representing the interconnection. Simulation results are in summary form, consisting mainly of execution times, average utilizations, and average path measures.

PIE is the Programming and Instrumentation Environment for parallel processing [Segall and Rudolph, 1985]. It is specific to a particular shared memory system, but its designers expect it to be translatable to other systems. It supports the shared dataspace model of concurrent computation and tuple-based programming languages such as Linda. PIE provides an animated graphical representation of program objects and their relationships. During execution, several graphical displays show the status of the computation,

including a dynamic invocation tree, which shows utilization of processes and processors, and a bar graph, which shows cumulative statistics.

IPPM, the Interactive Parallel Program Monitor, was written as a debugging aid for the Intel iPSC [Brandis, 1986]. It monitors communication events on each processor by sending event debugging messages to the host. The host filters the events reported and stores an ordered event trace. Simultaneously the event trace is graphically displayed on a workstation.

Currently only for depicting sequential algorithms, Balsa, the Brown University Algorithm Simulator and Animator, creates an algorithm animation environment [Brown, 1988]. It is one of the most advanced and widely recognized program visualization tools. A user watches execution of an algorithm through various views, using graphical displays to explore a program in action. "Interesting events" play a key role in the animation of an algorithm. Typically, a general plan for visualizations of the algorithms is set forth, mainly to identify the interesting events in the algorithm which should lead to changes in the image being displayed. Then, interesting event signals are added to the algorithm. The intent of the research is to capture the entirety of an algorithm in a single, static picture.

PM is a parallel performance monitor that is one of the support tools packaged with EXPRESS, a parallel operating environment from Parasoft that runs on multicomputers such as transputers, NCUBE, Caltech Mark III, and Intel iPSC [Flower, 1989]. PM provides information about the execution or performance of a parallel program, including: communication

times, routines being called, activity on each processor at any point in a program, time spent in a routine, and so forth. Specifically, three tools are available: (1) the execution profiler, which monitors time spent in individual routines; (2) the communications profiler, which monitors time spent in communications and input/output; and (2) the event profiler, which shows the interactions between processors and allows user-specified events to be monitored.

The Victor project at IBM Research in Yorktown involves a transputer-based mesh of processor nodes and special hardware components associated with the nodes to support monitoring performance [Wilcke, 1989]. A color-coded display screen shows processor and link activity. Important issues being studied via this project include space-sharing (versus timesharing) and embedding logical topologies into physical topologies.

LTRAMS, the Loosely-Coupled Trace Measurement System, is an instrumentation tool being developed by the National Institute of Standards and Technology (NIST) [Roberts, 1989]. The tool supports a distributed hybrid (hardware/software) monitor measurement approach in which software triggers a measurement (or sampling operation) and hardware collects and stores the data. A global interrupt yields a snapshot of system performance. Important issues being studied via this project include grain size, perturbation or disruption effects of the monitor, and VLSI implementations of the instrumentation.

A project that has considerable merit but a slightly different emphasis than the ones we have already reviewed is underway at the University of North Carolina [Snodgrass,

1988]. The focus of the project is that a historical database, an extension of a conventional relational database, provides an effective way to manage information processed by the monitor of a complex system. The approach creates the conceptual view that the dynamic behavior of the monitored (subject) system is available as a collection of historical relations, each associated with a sensor in the subject system. It entails: specifying the low-level data collection, specifying the analysis of the collected data, performing the analysis, and displaying the results. The eventual goal is to couple the relational model with a suitable programming environment to form an integrated instrumentation environment. Thus far, the approach has been tested via two prototype implementations, one monitoring the Cm* multiprocessor system [Swan et al., 1977] and a second monitoring the Berkeley UNIX 4.2BSD operating system on a Sun workstation. We will return to several of the details of this project in the next chapter.

CHAPTER III.

PERFORMANCE MONITORING

Far better never to think of investigating truth at all than to do so without a method.

Rene Descartes

In Chapter II, several performance analysis tools are highlighted. The objective in developing any analysis tool is to use it to gain a better understanding of system performance within some context. In fact, tools become a necessity in order to properly investigate the basic principles associated with the behavior of complex computer systems. Even though each tool may use slightly different mechanisms to investigate the principles, it is significant that the tools apply a method to evaluate performance. An appropriate method and a good implementation of the method can help convert a disparate collection of results into a meaningful, coherent model. In this chapter, we focus on monitoring as a method of evaluating performance. Furthermore, this and the following chapter present a framework that creates an integrated environment for performance measurement and visualization.

Evaluation Methods

Performance measures can be obtained by applying the following evaluation methods:

- benchmarks
- monitoring (hardware or software)
- emulation
- simulation
- analytical modeling

For the projects reviewed in the preceding chapter, an assortment of methods is applied. PARET, PAW, B-Hive, and Balsa use simulation. Poker uses software emulation. HyperView, Victor, and LTRAMS are coupled with hardware monitors, while Seecube, IPPM, PM, PIE, and the relational approach interact with software monitors. Benchmarks and analytical modeling have been applied to multicomputers as well [Reed and Grunwald, 1987]. Each method has advantages and disadvantages when critiqued in areas such as accuracy, complexity, and flexibility. Unfortunately, no method achieves the best marks in all areas. So, we must choose a method that satisfies our particular needs.

The choice of a method is driven by these factors: our interest in studying the dynamic behavior of a complex multicomputer system, and our preference for a scheme the lets us capture the peculiarities of actual programs running on an actual computer. Given these requirements, monitoring is the method to be used. Though much has been written about monitoring uniprocessor systems, how monitoring should be done for multicomputer systems is under study. In the remainder of this chapter, we summarize the present approaches to monitoring multicomputer systems and consider several of the issues that influence implementations of monitors. Though we simulate a distributed monitor in the prototype system described later, developing and verifying an actual monitor are beyond the scope of the work reported here. However, we

should note that a monitor is truly the heart of a performance measurement system and much work remains to be done in the area of monitoring multicomputer systems.

Monitoring, or instrumented execution, is the extraction of dynamic information concerning a computation as the computation proceeds. It involves observing and recording information at particular points in the computational system. The points may be positions in the spatial domain of the system or moments in the temporal domain of the system. A monitor may be implemented in hardware, firmware, or software, or some combination of the three. A hardware monitor consists of probe-type circuitry physically built into the machine. A software or firmware monitor typically includes special routines (or sections of code) augmented for data generation, collection, and analysis. Some form of hardware support is being included in most present and future systems; for example, a hardware-assisted software monitor may consist of software that generates the monitoring data and hardware that captures (that is, collects and stores) the data. Ideally, a monitor should be transparent to the user, implemented at the system level rather than the application program level. Two possibilities include compiler instrumentation and operating system instrumentation. These approaches permit access to useful system level information and automatic compensation for monitoring artifact and measurement inaccuracies.

A note on terminology to ensure clarity may be useful at this point. We have used and will use the term "monitor" and its related forms in varying contexts. Interpretation within the specific context should avoid confusion. Because the adjective form "monitorial" is rather awkward to use, the noun

form "monitor" or verb form "monitoring" may be used as an adjective. For example, "monitor data" or "monitoring data" may refer to the data processed by the monitor. Other examples include "monitoring granularity" and "monitoring artifact". In these examples, the term "measurement" can be used interchangeably with "monitor" (or "monitoring"), for example, "measurement data". In addition to being used as an adjective, the term "monitoring" may be used as a noun.

Monitoring is a fundamental component of many computing activities and has two primary applications: (1) debugging of programs and (2) measuring (or tuning) performance. It is a first step in understanding a computation, for it provides an indication of what happened, thus serving as a prerequisite to determining why it happened. Though a monitor may support both debugging and measurement, the two activities have certain distinctions. Debugging is typically done from a programming viewpoint, while performance tuning may be from a programming or engineering viewpoint. Debugging places more stringent requirements on the role of the monitor. A monitor should be able to support user interaction in real-time during program execution; to suspend, single-step, and resume the program execution; and to symbolically access program information, such as code and variables. A monitor that supports performance measurement also requires feedback from program execution, but it may be able to use post-processing of data rather than real-time processing, which eases some requirements. But other obstacles remain. For example, data management, particularly data storage, becomes a greater concern. Further, a monitor needs to handle a potentially large number of statistical calculations and should provide an

interface to a graphical display. Finally, though the monitor inevitably affects the performance it attempts to measure, its perturbations should be minimal so that it remains a useful tool for evaluating performance. Present monitors are being developed for both applications; however, in accord with the objectives of this work, we focus on performance measurement.

Before proceeding with a discussion about monitoring multicomputers, a few general comments about instrumentation may be helpful. A subject system or target system is the program and machine being monitored. The additions to the subject system to accomplish performance measurement comprise the instrumentation. A collection point [Couch, 1989] or sensor [Snodgrass, 1988] is a mechanism (for example, a hardware probe or system routine) that captures performance data concerning an event within the subject system. An event is viewed as occurring instantaneously and reflects a change in the state of the system. Thus, a state has some time duration and is demarcated by the events that caused the transitions to it and from it. More specifically, an event is associated with a change in the values of one or more parameters of interest.

To observe the behavior of the system, we track the values of specified parameters during program execution and generate a list of changes in their values. In other words, we log occurrences of events. The list is called an event trace, and the elements of the list are called notes [Couch, 1989] or data packets [Snodgrass, 1988]. A data packet may be as simple as a bit that is complemented when the event occurs or as complex as a long record containing system data. Typically, a data packet encodes information that includes

event type, parameter name, parameter value, and a time stamp. If the event is detected and the information logged when the event occurs, data packets are called traced data packets, and their generation is synchronous with the event. Alternatively, sampled data packets are logged only via an external request, and thus their generation is asynchronous with the event. Enabling a sensor allows it to detect events and generate data packets when events occur. Sensors may be enabled and disabled via flags. A traced sensor, which generates traced data packets, is enabled in the above sense; a sampled sensor, which generates sampled data packets, is triggered at selected times. Filtering is the removal of irrelevant data packets before they are completely processed by the monitor.

In contemporary implementations, monitoring may be summarized as consisting of three phases: (1) data collection, (2) data analysis, and (3) data display. More specifically, monitoring consists of a series of steps [Snodgrass, 1988]:

(1) sensor configuration, which involves deciding what information each sensor will record and where the sensor will be invoked;

(2) sensor installation, which involves coding sensors (if in software) and defining temporary and permanent storage of collected data;

(3) enabling sensors, which permits some sensors to be permanently enabled, storing monitoring data whenever executed, and others to be individually or collectively enabled;

(4) data generation, which involves executing the subject program and storing the collected data;

(5) analysis specification, which involves deciding what statistics to compile, usually done via a menu of available statistics or a simple command language;

(6) display specification, which involves deciding how to view the data, usually done via a menu of formats, ranging from a list of data packets printed in a readable form to standard reports to simple graphics;

(7) data analysis, which usually occurs in batch mode after the data have been collected; and

(8) display generation, which usually occurs immediately after data analysis.

Steps one through four comprise the data collection phase; steps five and seven, data analysis; and steps six and eight, data display. Most monitoring systems include these eight steps, although the ordering and composition may differ slightly.

Monitoring Complex Systems

Thus far, we have considered monitors in general. However, monitors for complex systems demand special consideration. Two important distinctions relevant to monitoring include: (1) complex systems often exhibit a lack of central control, and (2) complex systems may consist of a very large number of components. In this section, we address several problems that result from these characteristics, including data storage, clock synchronization, and performance perturbations.

For distributed memory computer systems, a separation of the monitor into two components is required: (1) a remote

monitor, performing functions requiring close interaction with the user; and (2) a resident monitor, performing functions requiring close interaction with the subject system. The distributed resident monitor exists at each processor, sending collected data to the centralized remote monitor. Functionally, the resident monitor collects the data packets and (possibly) interacts with the operating system, and the remote monitor analyzes and displays the data. Data collection is divided between the sensor storing the data packet in a buffer and the resident monitor extracting the data packets from the buffer and assembling them into larger packets to be sent to the remote monitor.

Collected data is stored in a memory buffer on each processor node. If the program execution time is small enough, or the buffer large enough, this approach may be sufficient to handle data storage needs. Several options exist when the buffer size is insufficient, including terminating logging, using a circular buffer [Couch, 1989], using a partitioned buffer [Rudolph and Reed, 1989], using a disk buffer, data streaming, filtering, and distributing the analysis. Data storage requirements vary depending on the implementation. Sensor control is particularly important, since a complex system has a potentially large number of sensors. A brute-force enabling of all sensors is excessively inefficient, since more storage is needed if all data are first collected and then analyzed. Alternatively, less storage is needed if the desired information is specified before any actual data are collected. Hence, only the necessary sensors should be enabled, thereby filtering out unnecessary data packets. Filtering should occur early and

often, so that scarce storage, processing, and communication resources are not expended on data that are later discarded. In fact, in terms of the resources in a complex system, it is likely impossible to store data on every event when many sensors are present. Powerful filtering techniques could even enable and disable sensors based on previously received data. However, achieving high degrees of filtering requires additional storage and processing to determine if a sensor is indeed enabled. This is expensive in an environment supporting many entities. Optimally, we want to enable the minimum number of sensors and perform just the computations needed to derive the desired information. Substituting sampled sensors for traced sensors where feasible can also reduce data storage overhead.

The data analysis generally occurs at a central node that hosts the remote monitor. The data packets are sent to this node from buffers in the processors where the sensors were located that generated the packets. However, much of the analysis could occur locally, with only that analysis requiring more global information being performed remotely (i.e., at the host). This distribution of analysis reduces the amount of data stored and also the amount of data transmitted between resident and remote monitors. Clearly, it is beneficial to limit the data transmissions between resident and remote monitors, both to limit the possible effects on application program performance and to keep the transmission rate within the bandwidth of the communications network.

Another issue that surfaces is the lack of a global clock. On most distributed systems, each processor has an independent clock. The data packets generated by the sensors,

however, contain time stamps that are supposed to represent global times across the entire system. We cannot use unsynchronized local clocks for timestamping events if we expect to merge the event traces from all processors based on the time stamps. However, it is theoretically impossible to synchronize imprecise physical clocks over a distributed network with nondeterministic transmission times [Lamport, 1978]. But it is possible to implement a time-keeping algorithm that maintains a global clock with a bounded imprecision. The algorithm ensures that a message is received at a global time that is later than the global time at which the message was sent, and it preserves a partial ordering of local events. The distributed algorithm can be implemented in the operating system to effectively synchronize clocks on all processors. If the operating system provides a reliable and fault tolerant communication mechanism, supporting recovery from lost messages or crashed processors, then a global clock is probably already computed by this mechanism. Alternatively, it is possible to synchronize the clocks using a global, serial connection to all processors [Rudolph and Reed, 1989]. Because the clocks may still drift at a rate large enough to affect the merging, a correction for any drifting is typically included as the traces are merged. More sophisticated approaches to clock synchronization are still being developed.

A monitor inevitably affects the performance it attempts to measure due to the instrumentation and to the storage, processing, and communication overhead of event data in the system. The effects are sometimes called performance perturbations or monitoring artifact and may significantly

degrade application program performance. Though there exist ways to reduce the overhead, it cannot be eliminated. Hence, a minimal performance penalty is usually accepted, and perturbations are prescribed to be within certain allowable limits. Generally, the more measurements taken, the greater the perturbation of the system. One approach to limit the perturbation for many measurements is to make many separate measurements and combine the results. In any case, to present accurate performance measurements, the monitor should be designed to compensate for the effects.

Many of the problems encountered with complex systems are merely a result of the size of the system. Though systems continue to grow larger, the endless quest for a truly scalable machine reveals the difficulty of building real systems that have minimal scaling effects. Clearly, if it is difficult to build a system that scales well, it is not easy to build a monitor, overlooking this system, that scales well. We have already mentioned some of the problems of scale in collecting a large volume of performance data. The remaining chapters introduce a possible solution to problems of scale in representing a large volume of performance data. The solution may also help us to better understand scaling effects within the machine. There is a pressing need in the study of complex systems to effectively deal with scalability and problems of scale.

Data Management

In an abstract sense, monitoring is concerned with retrieving information and presenting this information in a derived form to the user. Hence, the monitor is an information processing agent, with the information describing time-varying relationships between entities involved in the computation. In fact, the sensors associated with the monitor consume information as input and then generate information as output in the form of event records. In simplistic terms, tables of event records are created at each processor, describing local behavior, and then the tables are merged into one or more tables describing global behavior. We might find this organization of data fairly manageable, at least conceptually, since tables are such a familiar construct. However, because the tables are possibly very large, we need efficient and effective ways to access and present the information contained in them.

It may be illustrative to comment on one way to access, and in general manage, the tabular information, before looking at ways to present the information. Snodgrass has developed formalizations of the information processed by a monitor and proposed that the information can be perceived as a relational database [Snodgrass, 1988]. He differentiates four types of databases by their ability to support temporal information: snapshot, rollback, historical, and temporal [Snodgrass and Ahn, 1986]. Furthermore, he suggests that the historical type is appropriate for monitoring because of its ability to model the dynamic state of a computation [Snodgrass, 1988]. At a glance, the historical type of database is a natural way to

view monitoring information. When information is stored to update the status of the system, event data is recorded along with a time stamp in an event record (or data packet). Instead of a new event record overwriting an old one, event records are preserved (at least until processing is done), and the time stamp is used to give a partial ordering of the event records in the event trace. So the state of the system at a particular moment in time can be reconstructed from the most recent event records with time stamps less than or equal to the desired time. In fact, the global event trace can be conceptually partitioned into any number of tables depending on the desired view of the data. Thus, the dynamic behavior of the system is available as a collection of tables, or, in database terms, relations. In practice, the tables are only conceptual and do not actually collectively exist in their entirety as data stored either in main memory or in secondary storage.

This fictional database stores primitive or basic information that is captured by the sensors. The analysis process then produces derived information. Derived information typically holds more meaning for the user. Flexibility in analysis extends the usability of the tool, since (derived) information not anticipated at the time the monitor was implemented may still be requested by the user, provided the basic information is available to the monitor. Some command language is typically provided to specify the derived information. The command language may be very sophisticated, such as TQuel, the general temporal query language for the relational monitor described by Snodgrass [Snodgrass, 1988]. However, it is usually simpler, such as

the checklist windows of Seeplex [Couch, 1989], or the softkeys of various workstation-based tools.

Each event record in the fictional database has some number of fields. One field identifies the time that the event occurred. Another field identifies the location where the event occurred. Position is denoted by processor number, process name, channel number, or similar attributes. The time and position attributes are essential if the measurements are to reflect the temporal and spatial behavior of the system. Other attributes identify and describe the event. Events may be placed in categories, including: message-related, process-related, system-related, and user-defined [Rudolph and Reed, 1989]. Several events may be associated with message transmission, including send request, start of transmission, end of transmission, receive request, and actual reception. Many of the message events are logged with the sequence number of the message. This number is generated by the sending processor node; thus, a message's source node and sequence number uniquely identify it. This allows event processing software to associate send events on the source node with receive events on the destination node. Events recording entry to and exit from system calls provide information about the time consumed by operating system activities (versus user activities). System time is typically partitioned according to type of system call, including message activity, input-output activity, system activity (e.g., load balancing), and idle (no activity). If multiple processes are supported, then process-related events may record context switches between processes and identify currently executing processes (or objects in an object-oriented environment). Finally, user

events are triggered by the application program, usually via special operating system calls. These may record sections of code passed or values of variables.

At a fundamental level, the fictional database is just a table or set of data. By definition, it is a collection of multivariate data, because it involves a group of entities about which we have several quantitative measurements. The entities are the processors and processes in the concurrent computer. Data on these entities have been called observations, events, records, notes, and packets (among other similar terms). The different measurements have been referred to as variables, attributes, fields, metrics, and parameters (again, among other similar terms). By viewing the data in the sense of multivariate data, we have at our disposal powerful techniques for analyzing and displaying the data. We will return to this idea in Chapter V.

Perspectives

The state of the computer system, and thus system performance, can be viewed from different perspectives. Here, perspective refers to observational viewpoint or frame of reference. Different perspectives provide different pieces of the performance puzzle.

Three types of perspective are possible:

(1) program (algorithm or software),

(2) architecture (logical network), and

(3) machine (physical network or hardware).

A program perspective shows the flow of control and data in terms of algorithm entities (e.g., processes or data

structures). A notation for this typically uses two or more dimensions, including flowcharts, graphs, diagrams, or icons. A similar notation may be used for an architecture perspective. An architecture perspective highlights the logical structure and interaction of the components in the computer system (e.g., processors and channels). A machine perspective focuses on the implementation of the architecture in two- or three-dimensional space. The machine perspective could extend down into a circuit-level description of the hardware.

Figure 3.1 contains an example illustrating the relationships among these three perspectives. The program is represented via a process communication graph. The architecture is assumed to be a three-dimensional (eight-node) binary hypercube and is represented by a node interconnection graph. Finally, the machine is a two-dimensional geometrical layout, where each cell in the layout denotes a node within the computer system. Observe that Figure 3.1 also describes the process that maps a program onto a machine to achieve optimal performance. As presented in Chapter II, a substantial amount of research work has focused on the first two perspectives and on the associated mapping between algorithm and architecture. The remaining chapters describe the work we have done to also focus on the last perspective and on the mapping onto the machine.

Continuing with our discussion of perspective, we can define two levels of perspective:

(1) microscopic (low-level) and

(2) macroscopic (high-level).

A microscopic perspective focuses on the individual components

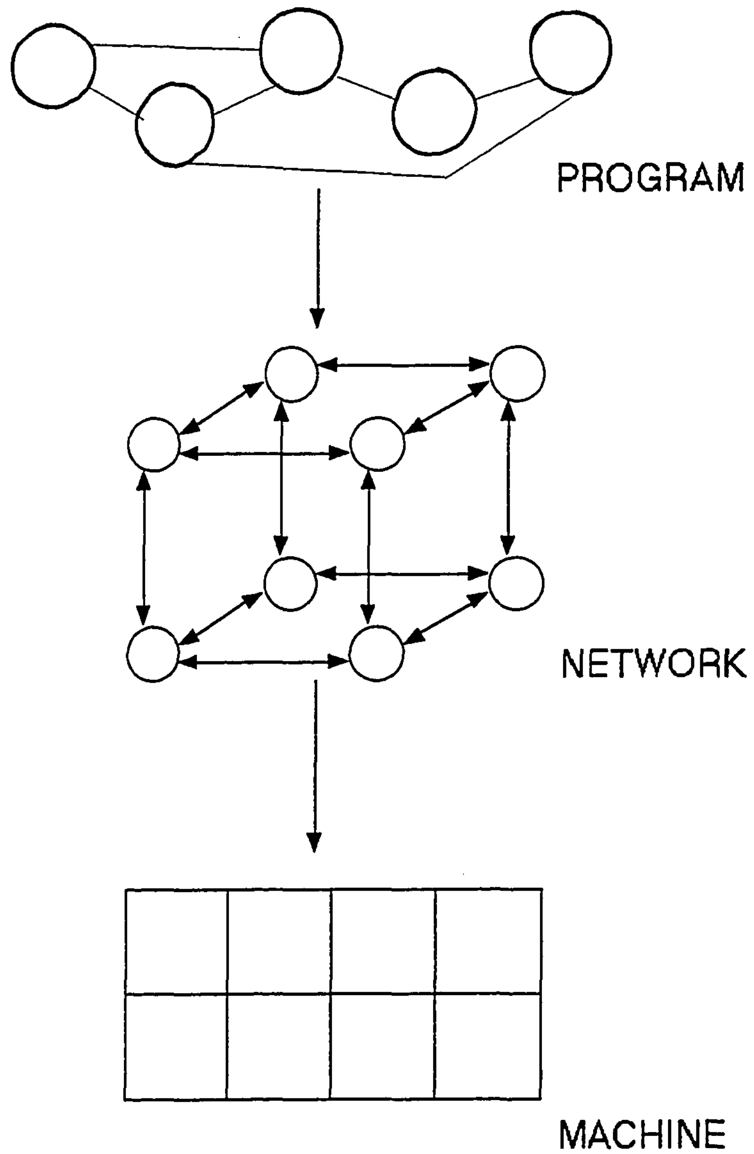


Figure 3.1. Three perspectives on system performance: program, architecture, and machine

of the system and evaluates each in isolation. Specific behavior can be inspected in detail. A macroscopic view reflects the overall behavior of the collection of components. Aggregrate (or global statistical) measures of performance can be obtained. The behavior of any component is analyzed in the context of all components.

The levels of perspective are associated with monitoring granularity. Monitoring granularity refers to the "size" of the entities or events that are observed by the monitor. The grain size influences the location and implementation of the sensors in the subject system. The grain size may range from less than one instruction to the whole program. Fine grain or low level monitoring involves tracing at the level of processor instructions, where new instructions change the state of processor registers, memory, and so forth. Fine grain monitoring requires hardware (or hardware-assisted) sensors. Medium grain or intermediate level monitoring involves tracing at the level of primitive operating system routines. The activities being monitored typically cause local state changes and may include interprocessor communication, input-output, and context switching. Hardware or software sensors are applicable to medium grain monitoring (with the obvious cost and performance tradeoffs). Coarse grain or high level monitoring involves tracing at the level of sophisticated operating system routines, such as collective communication routines or load balancing routines. The activities being monitored cause global, system-wide state changes. In essence, local state changes are lumped together and their effect as a whole is observed rather than the individual effects. Coarse grain monitoring results in the

smallest overhead on system resources and the least amount of detailed information. Software sensors are usually sufficient. Depending on the implementation of the monitor, information at particular levels in the subject system may be outside of the scope of the instrumentation and thus may be inaccessible for inspection.

In succeeding chapters, we use our knowledge about the nature of the monitor and the information that it processes to develop appropriate views of the information, views that will enhance our understanding of system performance.

CHAPTER IV.

REPRESENTING PERFORMANCE

Alice declared: "Dear, dear! How queer everything is today! And yesterday things went on just as usual. I wonder if I've been changed in the night? Let me think: was I the same when I got up this morning? I almost think I can remember feeling a little different. But if I'm not the same, the next question is 'Who in the world am I?' Ah, that's the great puzzle!"

Lewis Carroll, from Alice's Adventures in Wonderland

In this chapter, we move closer to the emphasis of this work, which is perceiving and understanding program performance on concurrent computers. The preceding chapter explored the monitoring of multicomputers in terms of complex systems. Given the abundance of monitoring information, we now turn to visualization. We examine ways to represent performance data, first presenting a spectrum of alternatives and then selecting appropriate representations for multicomputers.

Data Presentation

Performance data that are collected need to be analyzed and then presented in some meaningful format. Roughly, the analysis tells us what to look at, and the presentation tells us how to look at it. Data presentation should reflect the environment of a computation, because environmental conditions largely dictate system performance. That is, we typically

need to couple quantitative measurements of the system with qualitative observations.

As an example, Reed gives an illuminating analogy regarding peak versus actual (or achieved) performance [Reed, 1989]. When you need to drive someplace, say the grocery store, can you predict how much time it will take? Of course, it depends on how fast you can drive your car. Is it reasonable to expect peak performance? That is, if your car is capable of going, say, ninety miles per hour, is that the speed at which you will travel? Clearly, that is not very likely. It is more likely that your speed will vary along the way, depending on the streets traveled, the traffic, traffic lights, time of day, weather, accidents, and so forth. At best, you can estimate your average speed, and use it to predict your travel time. Although actual performance is more useful than peak performance, it does not tell us much about the actual trip. We may expand on Reed's analogy to expose the need and usefulness of more details. For example, if travel time is longer than might be expected, why is it? Traffic may be slowing to a halt at a heavily used bridge. In other words, we need to record specific times and places to fully explain performance. In terms of computer systems, single number performance measures may be of some use, but we may also need to be presented with a detailed account of when and where events occurred.

So, of the possible performance scenarios for complex systems, some are more informative than others. A single number such as peak performance or speedup is coarse and without insight. Another single number, the mean value of a distribution over all processors, has more information

content, but focuses on one particular moment in time. It isolates when events were observed. Alternatively, a timing profile focuses on one particular position in the system, say a processor. It isolates where events were observed.

A timestamped event trace, though selective in its information content, comes closest to providing a detailed account that includes both when and where events occurred. We can view an event trace in several forms, including textual, statistical, and graphical. Textual form comprises a raw listing of the sequence of individual events. The large number of events typically logged for a complex system makes this form impractical for manual performance evaluation. Statistical form is a compilation of statistics extracted from the event trace. The statistics are typically timing and counting metrics that give static insight about computation and communication activities. Graphical form involves visual display of information from the event trace and is especially powerful if coupled with animation. Animation is the process of stepping through the event trace and updating a display as time progresses. There are many possible types of data displays, including bar charts, line graphs, matrix diagrams, and general graphs, to name a few. Animated visual displays provide dynamic insight about computation and communication activities.

Categories of Concurrent Computers

As discussed in the previous section, measurement data should be presented in some meaningful format. Because formats have different capacities for conveying information,

the type of format to be used depends on the nature and volume of the data to be presented. The nature and volume of measurement data are determined by the computer system being observed. In this section, we suggest a categorization of computer systems that relates to the data generated during performance monitoring. The intent is to create a framework for the performance formats developed in the next chapter.

First, let us propose a metric that is proportional to the potential volume and variability of monitoring data; call it the complexity coefficient, CC. Let the complexity coefficient represent the information-bearing capacity of the computer system under study (including both processing-related and communication-related information). For our purposes, the complexity coefficient has the following definition:

$$CC = a \cdot n + b \cdot n$$

where n is the number of processors (computer size), a is the number of bits per processor (processor size), and b is the product of the number of channels, or neighbors, per processor (network dimension) and the number of bits per channel (channel width). So, roughly, CC is the sum of the processing and communication capacities of the system, measured in bits. A larger complexity coefficient means a potentially larger volume of monitoring data, and this places greater requirements on formats for presenting the data. The information-conveying capacity of the presentation format should meet or exceed the information-bearing capacity of the computer system under study. Of course, the actual features of the data set also depend on the application program, but a metric independent of the program is sufficient for this work. Via the complexity coefficient, complexity categories can be

established with each category having some range of CC. A computer system is then placed in a category depending on its calculated value of CC. This provides a basis for comparing computer systems.

Next, let us identify the types of data presentation that may be assigned to the complexity categories. Two definable features of data presentation are performance perspective and performance format. The types of performance perspectives include: program, network, machine, microscopic, and macroscopic. These were discussed in Chapter III. Each type of performance format fits into one of three representations: (1) single number, (2) table, or (3) graph. More specifically, the formats include:

- raw datum
- statistical datum
- table of data
- program flowchart
- program graph
- network graph
- basic chart
- ordered network graph
- multivariate (multidimensional) data plot

A raw datum is a single number performance indicator, such as execution time, network bandwidth, or processor throughput. A statistical datum is an aggregate measure, possibly a spatial or temporal average, such as average network latency. A table of data is any listing or collection of textual data, including the actual event trace and a compilation of raw or statistical datums. A basic chart, such as a two-dimensional line or bar chart, is the traditional

mechanism for displaying data. A program flowchart is a conventional icon-based control or data flow diagram illustrating program operation. A program graph is a general graph with nodes and edges that depicts the process or object structure of the program. Here, structure refers to the topology or connectivity of the constituent components, which are shown as nodes, and an edge or connection implies communication between the two adjoining nodes.

A network graph, on the other hand, is a general graph that depicts the processor structure, from an architectural or logical viewpoint. An ordered network graph is a network graph in which the nodes are placed in a special pattern, such as a gray code circle representation for a hypercube topology. Finally, a special kind of multivariate (or multidimensional) data plot that we refer to as a machine plot is a two- or three-dimensional geometrical layout. It illustrates processor configuration and information about the processors. Machine plots will be discussed in more detail in the next chapter.

Table 4.1 pairs complexity categories with appropriate types of performance perspectives and formats. Recall, the capacity of the data presentation mechanism to convey information should meet or exceed the processing and communication capacities (given as the complexity coefficient) of the computer system under study. In a rather coarse but elucidative partitioning of the full range of complexity coefficients, five overlapping complexity categories are marked out. Each category corresponds to a particular type of computer system, and the overlap results from similarities among the systems. Despite the overlap, the ranges of

complexity coefficients have distinguishable bounds and largely depend on system size. The five systems we use for comparison are: uniprocessor, bus multiprocessor (shared memory), small multicomputer, medium multicomputer, and large multicomputer. The multicomputer distinctions are consistent with a description given in [Reed and Fujimoto, 1987].

Further, no assumptions are made about the multicomputer network topology, and the topology can range from a ring to a mesh to any k-ary n-cube (cubes with n dimensions and k nodes in each dimension, of which the binary n-cube is a special case). Also, we should note that the values used to define the computer systems are realizable with present technology but may not exist in current systems; so the ranges are broad, and actual values would tend to cluster in fairly narrow regions. The upper bound for channel width results from a discussion in [Dally, 1987] about VLSI wiring density and pin count limitations.

Observe the following by examining Table 4.1. The larger computer systems require more global, hierarchical approaches to representing performance. Though we still benefit from access to low-level details, or a microscopic view, we first need to see the higher levels, or a macroscopic view, so that we are not overwhelmed by the details. Also, the increasing importance of accurately accounting for both time and space as systems scale up leads to visualization of performance data in the context of the machine, not just the program or the network. The more traditional formats, including numbers, tables, and flowcharts, break down under the additional complexity of large systems. Even general topological graphs are not sufficient for very large systems. These formats

Computer System					Complexity Coefficient	System Type	Data Presentation	
System Size (n)	Processor Size (a)	Network Dimension (b_1)	Channel Width (b_2)	Channel Size ($b=b_1*b_2$)	($CC=a*n+b*n$)		Performance Perspective	Performance Format
1	1-32	0	0	0	1-32	uniprocessor	P, Mi	N1, T, G1, G4
2-50	1-32	1	1-16	1-16	4-2400	bus multiprocessor	P, N, Mi	N1, T, G1, G2, G3, G4
10-100	1-32	1-6	1-16	1-96	20-12800	small multicomputer	P, N, Mi	N1, N2, T, G1, G2, G3, G4, G5
100-1000	1-32	2-9	1-16	2-144	300-176000	medium multicomputer	P, N, M, Mi, Ma	N2, G2, G3, G4, G6
1000-*I*	1-32	2-*I*	1-16	2-*I*	3000-*I*	large multicomputer	P, N, M, Ma, (Mi)	N2, G4, G6, (G2, G3)

Perspective:
P = program
N = network
M = machine
Mi = microscopic
Ma = macroscopic

Format:
N1 = raw datum
N2 = statistical datum
T = table of data
G1 = program graph: iconic
G2 = program graph: topological
G3 = network graph: topological
G4 = basic chart
G5 = network graph: ordered
G6 = machine plot: geometrical

() : indirectly used, via a hierarchical selection mechanism
I : infinity

Table 4.1. Complexity categories for performance data presentation formats

simply cannot convey enough information at a glance. Though the traditional formats are still very useful for viewing isolated parts of the system, new formats for viewing the system as a whole are essential. One of these formats, called a machine plot, has been developed as part of this work and is described in the next chapter. Finally, for an accurate and (sufficiently) complete understanding of system behavior, it is important to have a wide variety of views and interpretations of monitoring data.

CHAPTER V.

PICTURES OF PERFORMANCE

Nothing ever becomes real till it is experienced -- even a proverb is no proverb to you till your life has illustrated it.

John Keats

The previous two chapters presented a framework that supports the measurement and representation of performance data associated with (possibly) large concurrent computers. We have developed a methodology for pictorially displaying the performance of multicomputer systems that is consistent with this framework. This chapter describes the methodology and defines novel metrics and graphics.

Methodology

The method we have developed for "picturing" the performance of multicomputer systems creates a laboratory for observing, analyzing, and displaying performance. A prototype implementation that demonstrates the approach, including simulated results from several case studies, is presented in the next chapter. Four key elements of the approach are:

(1) Observation and measurement of performance via instrumented execution of actual and synthetic benchmark programs.

(2) Analysis and reduction of performance data via

appropriate techniques.

(3) Calculation of aggregate measures of system behavior.

(4) Visual display of program performance via a computer graphics format that illustrates computation and communication activities in time and space within the machine.

The fully-equipped laboratory configuration consists of these components:

- multicomputer (real or simulated system)
- distributed software monitor (possibly with hardware support)
- benchmark generator
- graphics workstation
- program database
- machine database
- event database
- statistical analysis tool
- visual analysis tool

Each of these components may be a highly capable subsystem by itself. Quite a bit of development activity still remains to be done in each area. Even more is needed to integrate the components into a functioning enterprise. Thus, while much work would be required to fully implement this laboratory, the prototype described in the next chapter demonstrates that the approach is both feasible and powerful.

Though an actual multicomputer is the intended target system, the method works fine in conjunction with a multicomputer simulator, which may be the only operational form of the system in the early stages of development. The software monitor is an instrumented version of the operating

system and, given an actual multicomputer, may be assisted by a hardware monitor. Special operating system routines log events of interest at each node, so each node maintains a trace that delineates its computation and communication activities. Upon program completion, the event traces from all nodes are combined into a global event trace, which forms the event database described in Chapter III. Depending on the implementation of the monitor, post-processing of event data may be coupled with real-time processing. Whether retrieved from the nodes during or after program execution, event data is processed to reconstruct program state information and extract other desired information.

Either instead of or in the absence of actual application or benchmark programs, synthetic benchmark programs may be executed. The benchmark generator creates user-specified synthetic benchmarks that drive the system according to predefined concurrent programming paradigms or network traffic patterns. The machine database contains machine-dependent information about network topology, routing, and physical implementation, and it is accessed to interpret trace data in the correct context. The program database contains application-dependent information about the process (or object) structure that can be inferred from the program text and the event database. It can be accessed along with the machine database to analyze the mapping of the program onto the machine.

The statistical analysis tool comprises the statistical software on the host system. It may also reside on the nodes of the multicomputer, if any distributed analysis is supported. The statistical software processes the monitoring

data, calculating local and global statistics. Some statistics are predefined and others may be user-defined. If analysis is closely linked with monitoring, desired statistics may be user-selected before runtime so that the monitor can filter out unnecessary data.

The visual analysis tool pictorially displays the dynamics of the system. It provides an interactive and animated environment for replaying the spatial and temporal behavior of the machine. Furthermore, via a hierarchical presentation of data, it attempts to display performance data at an appropriate level. Color-coded plots, described at the end of this chapter, show system activity, and time-series profiles can report statistical metrics and individual node activity. Finally, the graphics workstation hosts the multicomputer and maintains the performance databases and software tools. It has a window-based user interface that serves as a control panel for using the laboratory.

The following steps indicate how the method works:

(1) Configure the multicomputer for instrumented execution.

(2) Set up the statistical analysis tool.

(3) Generate a synthetic benchmark (if needed).

(4) Run the application or benchmark program.

(5) Collect traces and create the event database.

(6) Set up the program and machine databases.

(7) Invoke the statistical and visual analysis tools.

(8) Evaluate the performance via displayed metrics and graphics.

In summary, the methodology offers a combined prescription for:

(1) effective ways to specify, capture, and retrieve information;

(2) effective ways to process information; and

(3) effective ways to display information.

The first item is achieved via instrumentation. The second, via a database-like organization and multivariate cluster analysis techniques. And the third, via multidimensional graphics. We will consider the second and third items in more detail.

Observable Parameters

Several factors determine the data that can be observed and thus the measurements that can be made. Since the monitor acts as our eyes into the system and is inevitably a selective viewer, it significantly affects the observable data. The more closely the monitor, specifically the instrumentation, is integrated with the system, the more information that is available to it. Close ties with the hardware yield machine-level details, and close ties with the operating system provide system-level and application-level details. Clearly, if the system does not permit a certain level of integration (or intervention), then certain parameters cannot be measured.

Observations are made by the sensors. The type of sensor, where it is located, and when it is enabled determine the information content of an observation. The observable parameters that can be obtained directly via a sensor are called basic metrics. Alternatively, the observable parameters that are only partially defined by sensor measurements are called derived metrics. A derived metric

requires information from a database or calculations involving other quantities to fully specify it. We consider each in turn.

Basic metrics

A basic, or primitive, metric is an observable parameter that can be obtained directly via a sensor. That is, it is directly measurable. It is a local variable, describing an observation from a low-level or microscopic perspective. A minimal set of basic metrics consists of:

- time of occurrence
- position (processor number)
- process (or object) identification
- event identification
- values of event-specific variables

Time and position metrics are used to represent temporal and spatial behavior. More importantly, the time stamp and processor number uniquely identify the observation in a global context. The event identification denotes the state change associated with the observation. It is decoded to interpret the values of any event-specific variables. Event-specific variables pertain to the event categories specified in Chapter III. The choice of variables depends on the monitor implementation. For the purposes of this work, only several variables are required to represent the desired view of performance. For message-passing events relating to interprocessor communication, we are interested in source and destination processor numbers, message length, and channel number. Message length (or size) is typically stated as a byte count. For message-passing events relating to input-

output, similar variables are important, except for one distinction: either the source or destination is not a processor. For process-related events, process size is of interest. Process size is the load or amount of work done by the process, typically stated as an operation count. The granularity of an operation depends on the application program. An operation may range from an integer or floating-point operation to a module of code. Depending on the program and the monitor, the load may be an actual, known quantity, an approximate quantity, or possibly an expected quantity (if probabilities are used for nondeterministic loads).

For subsequent use in defining derived metrics, we name four of the basic metrics as follows:

- t : time
- p : processor number
- m : message length
- w : work

Time is expressed in seconds. The processor number is typically a nonnegative integer sometimes used as the address of the processor. At a fundamental level, both message length and work represent amounts of information. Message length is stated in bytes, where a byte is a group of eight bits, and work is stated in terms of operations, which involve operands and results that are also groups of bits. Hence, bits would be an appropriate unit of measure.

Derived metrics

A derived metric is an observable parameter that can be obtained via a combination of sensor measurements, database information, and calculations. It typically is a function of one or more basic metrics. If a derived metric uses only local information, say basic metrics from one spatial locality, then it offers a low-level or microscopic perspective on performance. Alternatively, if it includes global information, then it gives a high-level or macroscopic perspective. Global information involves metrics and other information from more than one spatial locality.

A metric may be a function of the independent variables time and position. It can be defined at a particular position or point in space, say a processor; it can also pertain to a range of positions. Further, a metric can be defined at a particular moment in time, possibly the end of program execution; it can also pertain to a period of time. Hence, a metric may be for a single value or a range of values of an independent variable. The types of metrics include timings, counts, and ratios. A ratio may be a time rate, a density, a percent, or other interesting comparison between values. In some cases, it is useful to perform operations on a set of values for a metric, including finding the average value, maximum value, minimum value, and summation value. For use later, general definitions for these operations follow. Let k be the metric of interest, P be the highest processor number, and T the latest time. Though these definitions cover the full range of time and position values, subranges could be specified.

$SUM_p(k)$: $\sum_{p=0}^{p=P} k$

$SUM_t(k)$: $\sum_{t=0}^{t=T} k$

$AVG_p(k)$: $\sum_{p=0}^{p=P} k \;/\; N_p$, where N_p is the number of processor values

$AVG_t(k)$: $\sum_{t=0}^{t=T} k \;/\; N_t$, where N_t is the number of time values

$MAX_p(k)$: Find largest value of k over all processors

$MAX_t(k)$: Find largest value of k over time

$MIN_p(k)$: Find smallest value of k over all processors

$MIN_t(k)$: Find smallest value of k over time

Many of the metrics used to evaluate program performance on concurrent computers in some way quantify computational and communication aspects of program execution. Some of these metrics are comparisons between amounts of computation and communication. However, there are no generally accepted units of measure for amounts of computation and amounts of communication. Several units of measure are meaningful. The amount of work done by a processor can be derived from either some number of granules of computation or computation quanta (such as elements in a list or points in a domain) or number of operations. The amount of message traffic through a processor can be stated as some number of communication quanta, such as bytes. Ultimately, for a pure comparison, we may want to specify work and traffic in the same units, such as bits. Expressing amounts of computation and communication

in terms of the time required per quantum also facilitates comparison.

Another point is worth mentioning. Communication over the network has two components, local traffic and through traffic. Local traffic consists of the messages sent or received by a processor; that is, the processor is the source or destination node. Through traffic consists of the messages traveling through a node, enroute to a destination node. Local traffic is directly observable. However, since message routing is often handled by a special communications processor on the node, through traffic is not visible to the monitor unless sensors are located within the communications processor. So, a value for through traffic may need to be interpolated from available global information, such as the routing strategy. Finally, half of the sum of local traffic over all nodes is the total system traffic, because local traffic is counted twice, at both source and destination.

Microscopic derived metrics that are of interest include:

- processor state
- operation count
- computation load (granule count)
- computation time
- computational energy (total work)
- computational power
- execution rate (throughput)
- program energy
- energy ratio
- message count
- communication load (byte count)
- communication time

- communication intensity
- communication density
- communication flow
- I/O traffic
- channel usage
- execution time
- percent computation time
- percent communication time
- granularity factor
- communication overhead

These metrics are defined at a particular position, the processor, typically at a particular moment in time. Definitions for each follow.

Processor state describes the current activity of the processor. The following types of activities may be encoded.

proc_state = (mode, status, activity, communication)

where

```
mode          ∈ {operating system, user}
status        ∈ {idle, active}
activity      ∈ {none, computation, communication}
communication ∈ {interprocessor send,
                 interprocessor receive,
                 input, output}
```

Operation count is the cumulative number of operations performed locally by tasks running on a processor. It is a function of the basic metric, work, w.

$$op_cnt = \sum_{t=0}^{t=T} w \quad \text{at } t=T,\ p=P$$

The summation implies adding all values of w recorded for processor P through time T.

Computation load, or granule count, is the cumulative number of computation quanta or granules operated on by tasks running on a processor. The size of a quantum depends on the application, and possible values range from a bit to a byte to a 64-bit word, or even larger. It is a function of the basic metric, work, w. The constant Q is the number of quanta per operation.

$$comp_ld = Q \cdot \sum_{t=0}^{t=T} w \quad \text{at } t=T,\ p=P$$

$$= Q \cdot op_cnt$$

Computation time is the cumulative time spent doing work, or local processing activities that contribute to the solution of the problem.

$$comp_tm = \sum_{t=0}^{t=T} \Delta t_W \quad \text{at } t=T,\ p=P$$

Here, Δt_W refers to a time period during which work is done.

Computational energy, or total work, is the cumulative amount of information processed by tasks running on a processor. Information is measured in bits. It is a function of the basic metric, work, w. The constant B is the number of bits per operation. Alternatively, the constant B_Q is the number of bits per quantum.

$$\text{comp_energy} = B \cdot \sum_{t=0}^{t=T} w \quad \text{at } t=T,\ p=P$$
$$= B \cdot \text{op_cnt}$$
$$= B_Q \cdot \text{comp_ld}$$

Computational power is the average temporal rate at which work is done or energy is expended, exclusive of any overhead.

$$\text{comp_power} = \text{comp_energy} \,/\, \sum_{t=0}^{t=T} \Delta t_W \quad \text{at } t=T,\ p=P$$

Here, Δt_W refers to a time period during which work is done.

Execution rate, or throughput, is the temporal rate at which work is done, or energy is expended, over the duration of program execution. This metric includes overhead effects.

$$\text{exec_rate} = \text{comp_energy} \,/\, T \quad \text{at } t=T,\ p=P$$

Program energy is the total amount of information expected to be processed by tasks running on a processor. It is an estimate derived from knowledge about the program. Information is measured in bits.

$$\text{prog_energy} = f(\text{program}) \quad \text{at } p=P$$

An amount can be determined automatically via proper analysis of the program database. The user can also input a value.

Energy ratio is the ratio of computational energy to program energy. It estimates the degree of completeness of information processing on a scale from zero to one.

$$\text{energy_r} = \frac{\text{comp_energy}}{\text{prog_energy}} \quad \text{at } t=T,\ p=P$$

Message count is the cumulative number of messages sent or received by the processor, including interprocessor communication and input-output.

$$\text{msg_cnt} = \sum_{t=0}^{t=T} i_m \qquad \text{at } t=T,\ p=P$$

Here, i_m is a binary variable. $i_m=1$ if an event recorded for processor P through time T is message-related; otherwise $i_m=0$.

Communication load, or byte count, is the cumulative number of communication quanta involved in message passing, including interprocessor communication and input-output. Here, we assume the size of a quantum is a byte. This metric is a function of the basic metric, message length, m. We include only local traffic in this definition, that is, messages sent and received by the processor. Note that through traffic could be included if sensors were available to record it in local storage.

$$\text{comm_ld} = \sum_{t=0}^{t=T} m \qquad \text{at } t=T,\ p=P$$

Recall, the summation implies adding all values of m recorded for processor P through time T.

Communication time is the cumulative time spent by the processor in message passing, including interprocessor communication and input-output.

$$\text{comm_tm} = \sum_{t=0}^{t=T} \Delta t_m \qquad \text{at } t=T,\ p=P$$

Here, Δt_m refers to a time period during which the (main) processor is busy with communication activities.

Communication intensity is the cumulative amount of information involved in message passing (for local traffic only). Information is measured in bits. It is a function of the basic metric, message length, m. The constant B_Q is the number of bits per quantum; a quantum is a byte, so B_Q=8.

$$\text{comm_int} = B_Q \cdot \sum_{t=0}^{t=T} m \quad \text{at } t=T,\ p=P$$
$$= B_Q \cdot \text{comm_ld}$$

Communication density is the amount of information involved in message passing that exists at the time of interest (for local traffic only). Information is measured in bits. This indicates the number of bits involved in communication at a particular time within the "space" of the processor node .

$$\text{comm_den} = B_Q \cdot \sum_{t=0}^{t=T} m_T \quad \text{at } t=T,\ p=P$$

Here, m_T refers to the lengths of messages that are currently being sent or received, including any buffered messages, at time T. The constant B_Q is the number of bits per quantum; a quantum is a byte, so B_Q=8.

Communication flow is the time rate at which information involved in message passing is processed and transmitted (for local traffic only). It indicates the rate at which bits "flow" into and out of the "space" of the processor node.

$$\text{comm_flow} = \text{comm_int} \,/\, \sum_{t=0}^{t=T} \Delta t_m \quad \text{at } t=T,\ p=P$$

Here, Δt_m refers to a time period during which the processor node is busy with communication activities. The endpoints of a time period depend on the data that can be logged by the monitor. Optimally, a message send time interval would range from initiation of message transmission by the program to completion of hardware transmission (at the source node). A message receive time interval would range from physical reception of the message to completion of message transmission and processing by the program (at the destination node).

I/O traffic is the portion of the communication load due to input-output activities (for local traffic only).

$$\text{io_traffic} = \sum_{t=0}^{t=T} m_{IO} \quad \text{at } t=T,\ p=P$$

Here, m_{IO} refers to the lengths of messages (in bytes) that are recorded as having a type of either input or output.

Channel usage is the number of communication channels currently being used for message passing at the time of interest (for local traffic only).

$$\text{chan_use} = \sum_{t=0}^{t=T} i_{ch,T} \quad \text{at } t=T,\ p=P$$

Here, $i_{ch,T}$ is a binary variable. $i_{ch,T}=1$ if a message-related event recorded for processor P through time T generates traffic on channel ch during time T; otherwise $i_{ch,T}=0$. A relative value may be stated as the ratio of channel usage to the number of channels per processor.

Execution time is the total amount of time the processor

node has been involved with program execution. It begins at time t=0, when the processor node is initialized. It should be equal to the sum of computation time and communication time (within reasonable measurement error).

$$\text{exec_tm} = T \text{ or } T_{done} \quad \text{at } t=T,\ p=P$$

T is the time of interest. T_{done} is the maximum time recorded for the processor. If T is greater than T_{done}, then T_{done} is used.

<u>Percent computation time</u> is the percent of the total time that is spent doing work. This is sometimes referred to as computational efficiency.

$$\%\text{comp_tm} = \frac{\text{comp_tm}}{\text{exec_tm}} \times 100\%$$

<u>Percent communication time</u> is the percent of the total time that is spent doing message passing.

$$\%\text{comm_tm} = \frac{\text{comm_tm}}{\text{exec_tm}} \times 100\%$$

<u>Communication overhead</u> is a measure of the time spent communicating per unit of time spent doing work. It is the ratio of communication time to computation time.

$$\text{comm_ovrhd} = \frac{\text{comm_tm}}{\text{comp_tm}} \quad \text{at } t=T,\ p=P$$

<u>Granularity factor</u> is a measure of the amount of information involved in processing activities relative to the amount of information involved in communication activities. It is the ratio of computational energy to communication intensity. It roughly indicates the number of bits of computation per bit of communication.

$$gran_fact = \frac{comp_energy}{comm_int} \qquad \text{at } t=T,\ p=P$$

If through traffic is not locally discernible via sensors at a processor node, it can be (roughly) reflected in the communication metrics by using global information. What is needed is a function that interpolates position and time along the route of the message. Hence, it depends on the message routing strategy of the operating system. In the simplest case, a specific route and a uniform rate along the route are assumed. Obviously, more complex cases are likely to occur, requiring more sophisticated interpolation functions. The function should generate event records similar to any message-related event record except with a type specified as through traffic. These event records would then be used to compile the desired statistics. We refer to this function by:

Interpolate_f (e_m, routing algorithm).

It takes as input a stream of message-related event records, e_m, and a routing algorithm. It matches send events with corresponding receive events and notes source and destination processor numbers and time stamps. Then, based on the routing algorithm, it determines (the most probable) intermediate nodes, if any, enroute from source to destination. Dividing the transmission time interval equally, it associates a time

with each visit to a processor node. This information is stored in event records along with the type and other message information. The output of the function is a stream of through traffic event records, e_{thru}. For clarity, we mark any metrics that use this function with a subscript "thru."

The communication metrics include message count, communication load, communication time, communication intensity, communication density, communication flow, channel usage, and I/O traffic. The previous definitions for microscopic metrics were for local traffic only. A value that reflects through traffic at processor P may be obtained by invoking Interpolate_f and performing some additional processing on the resulting through traffic event records. The following definitions specify the additional processing. Let t=T and p=P be the time and processor of interest, respectively. The subscript "tot" denotes a total value for the metric.

msg_cnt_{thru} = count of the number of event records in the result of Interpolate$f$

$$msg_cnt_{tot} = msg_cnt + msg_cnt_{thru}$$

$$comm_ld_{thru} = \sum_{t=0}^{t=T} m_{thru}$$

$$comm_ld_{tot} = comm_ld + comm_ld_{thru}$$

Here, m_{thru} refers to the lengths of messages that are recorded as being of type through traffic, which includes all event records generated by Interpolate_f.

$$\text{comm_int}_{thru} = B_Q \cdot \sum_{t=0}^{t=T} m_{thru}$$
$$= B_Q \cdot \text{comm_ld}_{thru}$$

$$\text{comm_int}_{tot} = \text{comm_int} + \text{comm_int}_{thru}$$

The constant B_Q is the number of bits per quantum; a quantum is a byte, so B_Q=8.

$$\text{comm_den}_{thru} = B_Q \cdot \sum_{t=0}^{t=T} m_{thru,T}$$

$$\text{comm_den}_{tot} = \text{comm_den} + \text{comm_den}_{thru}$$

Here, $m_{thru,T}$ refers to the lengths of messages that are of type through traffic and are currently being transmitted.

$$\text{comm_flow}_{thru} = \text{comm_int}_{thru} / \sum \Delta t_{thru}$$

$$\text{comm_flow}_{tot} = \text{comm_int}_{tot} / \sum (\Delta t_m + \Delta t_{thru})$$

Here, Δt_{thru} refers to a time interval during which through traffic visits a processor node, which can be determined from the event record list generated by Interpolate_f. Δt_m refers to a time period during which the processor node is busy with communication activities. The summation involves all time intervals for processor P (excluding any duplicate time periods, so that each time is counted only once).

$$\text{io_traffic}_{thru} = \sum_{t=0}^{t=T} m_{IO,thru}$$

$$\text{io_traffic}_{tot} = \text{io_traffic} + \text{io_traffic}_{thru}$$

Here, $m_{IO,thru}$ refers to the lengths of messages that are recorded as having a message type of either input or output and a type of through traffic. That is, event records for input-output communication activities are processed by Interpolate_f, and the resulting event records reflect I/O-related through traffic.

$chan_use_{thru}$ = count of the number of channels being used at time T in the result of Interpolate_f

$chan_use_{tot}$ = chan_use + $chan_use_{thru}$

Many of the macroscopic derived metrics are global measures based on corresponding microscopic metrics. These metrics involve some type of aggregate operation over all local values. Common aggregate operations include summation, averaging, and extrema-finding. A variable name used to denote a global measure will be prefixed with a "G_". A definition based solely on an aggregate operation has a standard form. Given the aggregrate operation "op" and the microscopic metric "k", the global metric is defined as follows: $G_k_{op} = op(k)$.

Macroscopic derived metrics that are of interest include:

- processor spatial coordinates
- state (or activity) occurrences
- operation count
- computation load
- computation time
- computational energy
- computational power
- program energy
- energy ratio

- message count
- communication load
- communication time
- communication intensity
- communication density
- communication flow
- I/O traffic
- execution time
- percent computation time
- percent communication time
- granularity factor
- communication overhead
- utilization
- channel utilization
- concurrency
- communication concurrency
- balance
- communication balance
- efficiency
- synergy

These metrics are defined over the space of the whole system, either at a particular moment in time or over the lifetime of the system. Definitions for the metrics follow.

Processor spatial coordinates give the position of the processor in a physical layout, such as a one-, two-, or three-dimensional grid. It is a function of the basic metric, processor number, p. The coordinates depend on the mapping function used to map or embed nodes of the network topology onto the physical topology. The result of this function (and the value of this metric) is a tuple with one, two, or three

components corresponding, respectively, to a one-, two-, or three-dimensional physical topology.

proc_coord = Mapping_f (p, N, topology)

Here, N is the total number of processors in the system, and topology refers to the network and physical topologies.

State occurrences, or activity occurrences, are absolute or relative indicators of the number of processors in each of the defined (and recorded) processor states. Possible states or activities include computing, sending, waiting, receiving, inputting, and outputting. An absolute value is a count of the number of processors in the specified state at a particular moment in time. A relative value is a ratio of the absolute value to the total number of processors in the system.

Operation count is based on the microscopic metric, op_cnt. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_op_cnt_{SUMp} = SUM_p (op_cnt)$$

$$G_op_cnt_{AVGp} = AVG_p (op_cnt)$$

$$G_op_cnt_{MAXp} = MAX_p (op_cnt)$$

$$G_op_cnt_{MINp} = MIN_p (op_cnt)$$

Computation load is based on the microscopic metric, comp_ld. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_comp_ld_{SUMp} = SUM_p (comp_ld)$$

$$G_comp_ld_{AVGp} = AVG_p (comp_ld)$$

$$G_comp_ld_{MAXp} = MAX_p (comp_ld)$$

$$G_comp_ld_{MINp} = MIN_p (comp_ld)$$

<u>Computation time</u> is based on the microscopic metric, comp_tm. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_comp_tm_{SUMp} = SUM_p (comp_tm)$$

$$G_comp_tm_{AVGp} = AVG_p (comp_tm)$$

$$G_comp_tm_{MAXp} = MAX_p (comp_tm)$$

$$G_comp_tm_{MINp} = MIN_p (comp_tm)$$

<u>Computational energy</u> is based on the microscopic metric, comp_energy. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_comp_energy_{SUMp} = SUM_p (comp_energy)$$

$$G_comp_energy_{AVGp} = AVG_p (comp_energy)$$

$$G_comp_energy_{MAXp} = MAX_p (comp_energy)$$

$$G_comp_energy_{MINp} = MIN_p (comp_energy)$$

<u>Computational power</u> is based on the microscopic metric, comp_power. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_comp_power_{AVGp} = AVG_p (comp_power)$$

$$G_comp_power_{MAXp} = MAX_p (comp_power)$$

$$G_comp_power_{MINp} = MIN_p (comp_power)$$

A useful definition that gives the overall rate of doing work (exclusive of overhead) is:

$$G_comp_power = G_comp_energy_{SUMp} / \Sigma \Delta t_w$$

Δt_w refers to a time period during which work is done. The summation involves all time intervals over all processors, excluding any duplicate time periods (so that each global time is counted only once).

Execution rate is based on the microscopic metric, exec_rate. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_exec_rate_{AVGp} = AVG_p (exec_rate)$$
$$G_exec_rate_{MAXp} = MAX_p (exec_rate)$$
$$G_exec_rate_{MINp} = MIN_p (exec_rate)$$

A useful defintion that gives the overall rate of doing work over the duration of program execution (including overhead) is:

$$G_exec_rate = G_comp_energy_{SUMp} / MAX_p(exec_tm)$$
$$= G_comp_energy_{SUMp} / G_exec_tm_{MAXp}$$

Program energy is based on the microscopic metric, prog_energy. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_prog_energy_{SUMp} = SUM_p (prog_energy)$$
$$G_prog_energy_{AVGp} = AVG_p (prog_energy)$$
$$G_prog_energy_{MAXp} = MAX_p (prog_energy)$$
$$G_prog_energy_{MINp} = MIN_p (prog_energy)$$

Energy ratio is based on the microscopic metric, energy_r. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_energy_r_{AVGp} = AVG_p\ (\ energy_r\ )$$

$$G_energy_r_{MAXp} = MAX_p\ (\ energy_r\ )$$

$$G_energy_r_{MINp} = MIN_p\ (\ energy_r\ )$$

A useful definition that gives the overall ratio is:

$$G_energy_r = \frac{SUM_p(\ comp_energy\ )}{SUM_p(\ prog_energy\ )} = \frac{G_comp_energy_{SUMp}}{G_prog_energy_{SUMp}}$$

<u>Message count</u> is based on the microscopic metric, msg_cnt. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_msg_cnt_{AVGp} = AVG_p\ (\ msg_cnt\ )$$

$$G_msg_cnt_{MAXp} = MAX_p\ (\ msg_cnt\ )$$

$$G_msg_cnt_{MINp} = MIN_p\ (\ msg_cnt\ )$$

Similar definitions hold using msg_cnt_{thru} and msg_cnt_{tot}. A definition for the cumulative number of messages in the system is:

$$G_msg_cnt = SUM_p(\ msg_cnt\ )\ /\ 2$$

<u>Communication load</u> is based on the microscopic metric, comm_ld. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_comm_ld_{AVGp} = AVG_p\ (\ comm_ld\ )$$

$$G_comm_ld_{MAXp} = MAX_p\ (\ comm_ld\ )$$

$$G_comm_ld_{MINp} = MIN_p\ (\ comm_ld\ )$$

Similar definitions hold using $comm_ld_{thru}$ and $comm_ld_{tot}$. A definition for the cumulative number of communication quanta, or message bytes, in the system is:

$$G_comm_ld = SUM_p(\ comm_ld\ )\ /\ 2$$

Communication time is based on the microscopic metric, comm_tm. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_comm_tm_{SUMp} = SUM_p (comm_tm)$$
$$G_comm_tm_{AVGp} = AVG_p (comm_tm)$$
$$G_comm_tm_{MAXp} = MAX_p (comm_tm)$$
$$G_comm_tm_{MINp} = MIN_p (comm_tm)$$

Communication intensity is based on the microscopic metric, comm_int. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_comm_int_{AVGp} = AVG_p (comm_int)$$
$$G_comm_int_{MAXp} = MAX_p (comm_int)$$
$$G_comm_int_{MINp} = MIN_p (comm_int)$$

Similar definitions hold using $comm_int_{thru}$ and $comm_int_{tot}$. A definition for the cumulative amount of information involved in message passing in the system is:

$$G_comm_int = SUM_p(comm_int) / 2$$

Communication density is based on the microscopic metric, comm_den. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_comm_den_{SUMp} = SUM_p (comm_den)$$
$$G_comm_den_{AVGp} = AVG_p (comm_den)$$
$$G_comm_den_{MAXp} = MAX_p (comm_den)$$
$$G_comm_den_{MINp} = MIN_p (comm_den)$$

Similar definitions hold using $comm_den_{thru}$ and $comm_den_{tot}$.

Communication flow is based on the microscopic metric, comm_flow. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_comm_flow_{AVGp} = AVG_p (comm_flow)$$

$$G_comm_flow_{MAXp} = MAX_p (comm_flow)$$

$$G_comm_flow_{MINp} = MIN_p (comm_flow)$$

Similar definitions hold using $comm_flow_{thru}$ and $comm_flow_{tot}$. A useful definition that gives the overall rate of processing and transmitting message information is:

$$G_comm_flow = G_comm_int / \Sigma \Delta t_m$$

Δt_m refers to a time period during which the processor node is busy with communication activities. The summation involves all time intervals over all processors, excluding any duplicate time periods (so that each global time is counted only once).

I/O traffic is based on the microscopic metric, io_traffic. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_io_traffic_{SUMp} = SUM_p (io_traffic)$$

$$G_io_traffic_{AVGp} = AVG_p (io_traffic)$$

$$G_io_traffic_{MAXp} = MAX_p (io_traffic)$$

$$G_io_traffic_{MINp} = MIN_p (io_traffic)$$

Similar definitions hold using $io_traffic_{thru}$ and $io_traffic_{tot}$.

Execution time is based on the microscopic metric, exec_tm. Aggregate operations include SUM_p, AVG_p, MAX_p, and MIN_p.

$$G_exec_tm_{SUMp} = SUM_p (exec_tm)$$

$$G_exec_tm_{AVGp} = AVG_p (exec_tm)$$

$$G_exec_tm_{MAXp} = MAX_p (exec_tm)$$

$$G_exec_tm_{MINp} = MIN_p (exec_tm)$$

Percent computation time is based on the microscopic metric %comp_tm. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_\%comp_tm_{AVGp} = AVG_p (\%comp_tm)$$

$$G_\%comp_tm_{MAXp} = MAX_p (\%comp_tm)$$

$$G_\%comp_tm_{MINp} = MIN_p (\%comp_tm)$$

Percent communication time is based on the microscopic metric %comm_tm. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_\%comm_tm_{AVGp} = AVG_p (\%comm_tm)$$

$$G_\%comm_tm_{MAXp} = MAX_p (\%comm_tm)$$

$$G_\%comm_tm_{MINp} = MIN_p (\%comm_tm)$$

Granularity factor is based on the microscopic metric gran_fact. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_gran_fact_{AVGp} = AVG_p (gran_fact)$$

$$G_gran_fact_{MAXp} = MAX_p (gran_fact)$$

$$G_gran_fact_{MINp} = MIN_p (gran_fact)$$

A useful definition that gives the overall ratio is:

$$G_gran_fact = \frac{SUM_p(comp_energy)}{SUM_p(comm_int)} = \frac{G_comp_energy_{SUMp}}{G_comm_int_{SUMp}}$$

Communication overhead is based on the microscopic metric comm_ovrhd. Aggregate operations include AVG_p, MAX_p, and MIN_p.

$$G_comm_ovrhd_{AVGp} = AVG_p (comm_ovrhd)$$
$$G_comm_ovrhd_{MAXp} = MAX_p (comm_ovrhd)$$
$$G_comm_ovrhd_{MINp} = MIN_p (comm_ovrhd)$$

A useful definition that gives the overall ratio is:

$$G_comm_ovrhd = \frac{SUM_p(comm_tm)}{SUM_p(comp_tm)} = \frac{G_comm_tm_{SUMp}}{G_comp_tm_{SUMp}}$$

Concurrency is the number of active processors involved in program execution.

$$\begin{aligned} concurrency &= \text{count of the number of active processors} \\ &\leq N \end{aligned}$$

where N is the number of processors in the system (or the part of the system allocated to this problem).

Communication concurrency is the number of active channels involved in message passing activities.

$$\begin{aligned} comm_concur &= \text{count of the number of active channels} \\ &\leq N \times d \end{aligned}$$

where N is the number of processors in the system (or the part of the system allocated to this problem), and d is the dimension of the system (i.e., the number of channels per processor).

Utilization is the percent of the total number of processors that are actively involved in program execution.

$$\text{utilization} = \frac{\text{concurrency}}{N} \times 100\%$$

where N is the number of processors in the system (or the part of the system allocated to this problem).

Channel utilization is the percent of the total number of channels that are actively involved in message passing activities.

$$\text{chan_util} = \frac{\text{comm_concur}}{N \times d} \times 100\%$$

where N is the number of processors in the system (or the part of the system allocated to this problem), and d is the dimension of the system (i.e., the number of channels per processor).

Balance is a measure that roughly indicates the amount of load imbalance in the system. It is the ratio of average computation time to the maximum of all processor computation times. Generally, as the ratio approaches one, the distribution of work becomes more uniform across the system. A spatial measure based on computational energy (in bits) may also be stated.

$$\text{balance} = \frac{AVG_p\ (\text{ comp_tm })}{MAX_p\ (\text{ comp_tm })} = \frac{\text{G_comp_tm}_{AVGp}}{\text{G_comp_tm}_{MAXp}} \leq 1$$

Communication balance is a measure that roughly indicates the extent to which traffic is evenly distributed in the system. It is the ratio of average communication time to the maximum of all processor commmunication times. Generally, as the ratio approaches one, the distribution of traffic becomes more uniform across the system. A spatial measure based on

communication intensity (in bits) may also be stated.

$$\text{comm_bal} = \frac{\text{AVG}_p\text{(comm_tm)}}{\text{MAX}_p\text{(comm_tm)}} = \frac{\text{G_comm_tm}_{\text{AVGp}}}{\text{G_comm_tm}_{\text{MAXp}}} \leq 1$$

Efficiency is a measure that indicates the quality of a parallel solution compared to a sequential solution. Two definitions are useful. Both are ratios of computation done on a sequential system to computation done on a parallel system. The first, eff_t, states computation in units of time; and the second, eff_{op}, in units of operations. As the ratio approaches one, the parallel system is spending more of its time doing useful work.

$$\text{eff}_t = \frac{T_s}{N \cdot T_p} = \frac{T_s}{N \cdot \text{MAX}_p\text{(exec_tm)}} = \frac{T_s}{N \cdot \text{G_exec_tm}_{\text{MAXp}}}$$

$$= \frac{T_s}{\text{SUM}_p\text{(exec_tm)}} = \frac{T_s}{\text{G_exec_tm}_{\text{SUMp}}}$$

$$\leq 1$$

where N is the number of processors in the system (or the part of the system allocated to this problem), T_s is the time that is (or would be) required to solve the problem on a sequential system, and T_p is the time required to solve the problem on a parallel system.

Synergy is a measure that indicates the quality of the mapping of the parallel program onto the parallel computer. We want a measure that parameterizes the properties of a good mapping: balanced load, high concurrency, and relatively low communication overhead. There is potential conflict among these parameters. The mapping problem is essentially an optimization problem. Several solutions to the problem have

been proposed (see the Fox and the Bokhari references), and the typical approach minimizes or maximizes some function. For our purposes, we choose a framework outlined in [Fox et al., 1988].

A problem is formulated in terms of a set of processes or objects, which are viewed as the vertices of a graph. Objects that communicate are connected by an edge of the graph. Each object α does work, w_α, and objects α and β need to communicate an amount of information $c_{\alpha\beta}$. The parallel computer can be described in the same form: a set of processors and an interconnection network. A subset of objects is allocated to each processor; let object α be at processor p and object β be at processor q. The total amount of work W_p for processor p and the total amount of communication C_{pq} along the path from p to q can be written

$$W_p \propto \sum_{\alpha} w_\alpha$$
$$C_{pq} \propto \sum_{\alpha,\beta} c_{\alpha\beta}$$

An objective function can be defined in terms of these variables. Though its exact form may vary depending on the application program or computer, it typically involves summation of work and communication values for all processors and paths in the system.

$$\text{Objective_}f = f\ (\ C_{pq},\ W_p\ )$$

We want to minimize the function over the whole system. Minimizing the objective function corresponds to maximizing the synergy, so we can use the reciprocal:

synergy = 1 / Objective_f

The notion of hot spots of activity in the system is supported by the defined metrics. Informally, a hot spot is a locale with high contention for its resources. At a hot spot, there exists a large amount of information being processed and a large amount of time being spent processing that information. Both computation hot spots and communication hot spots are possible. Using the above metrics, a hot spot occurs when the amount of computation (or communication) at a processor is greater than some percentage of the total amount in the system or greater than some constant amount. Hot spots become visually apparent via the graphics described in the next section.

One of the powerful aspects of this approach, as we have described it so far, is the expressiveness, flexibility, and simplicity of applying general multivariate statistical data analysis techniques to the performance data. Conventional cluster analysis techniques have been used to effectively reduce and order the data. The analysis presented in this section is just the beginning of the possibilities for exploring the data. For example, although the time-dependent and position-dependent parameters are clearly relevant to understanding system performance, dependencies with variables other than time and position may reveal important relationships as well. As we shall see in the following sections, the greatest power comes from applying graphical data analysis software for visualizing the performance data.

Graphics

Without graphics, vast amounts of diverse information cannot be easily assimilated. Visualization of data describing program performance is becoming especially important. Observations are more comprehensive and immediately clearer than any that can be drawn from the data in textual form. As discussed in Chapter IV, useful formats for representing data depend on the data and thus on the system that generates the data.

Of the possibilities for displaying data describing program performance on concurrent computers, we will focus on two formats. One format, timing profiles, is already in common use; and the second format, data plots, is introduced in this thesis. Both formats offer views of the basic and derived metrics defined in preceding sections. Timing profiles illustrate measures over time; and data plots, over space (and time if animation is used). Timing profiles are inherently sequential and (typically) one-dimensional in their presentation of information. Conversely, data plots are naturally parallel and multidimensional. Data plots encompass several different types of plots, from which we create *machine plots* developed through the course of this work. Although certain of the basic data in the event trace may be graphed directly, some preprocessing is typically required by the graphical software tools. The preprocessing may involve formatting the data and converting data into derived metrics.

Plots

A general class of multivariate data plots can be effectively tailored to display performance measurement data. The two types of data plots most relevant to this work are scatter plots and block plots. Scatter plots, or dot plots, are two- or three-dimensional pictures of data. Each dot represents an observation (or event) and denotes values for the variables associated with the coordinate axes. Further, the dots may be color-coded to denote the value of another variable. Up to four variables may be displayed. Within a dot plot, only dots representing observations of interest are visible. Subsets of dots may be selectively highlighted or hidden.

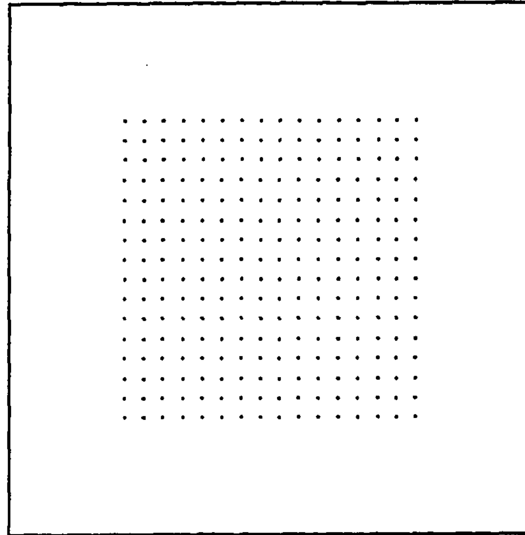
Block plots, or cell plots, are two-dimensional pictures of data. By definition, a block plot corresponds to an image. An observation is represented by a block of pixels, yielding a two-dimensional array of blocks or cells (i.e., an image) for a set of observations. A block denotes a value (or range of values) for the row and column variables, and it is shaded or color-coded to denote the value (or relative value) of a variable of interest. Three variables may be displayed. Within a block plot, all cells in the array are visible.

In the color-coded plots, data are often displayed with the highest values in red and the lowest in blue, with the display colors following the wavelengths of colors found in the visible-light spectrum. Color is an important feature, because it helps to visually identify trends and patterns in data sets.

Plots are especially useful for showing system activity. Though a data plot is completely general in the sense that any

subset of performance variables may be displayed, we define a special kind of plot. A machine plot is a special kind of plot (either a dot or cell plot) that displays spatial and temporal information extracted from performance measurement data. Spatial (space- or position-dependent) data are shown by labeling the coordinate axes with position variables. Temporal (time-dependent) data are reflected via animation by incrementally updating the display using a time variable. Hence, for displaying computer system performance, dots and cells represent processors. Processor numbers (or addresses) are mapped to grid coordinates. Adjacency in the plots (i.e., adjacent dots or cells) can correspond to physical proximity of processors within the actual machine. A dot or cell is color-coded to indicate the value of a parameter for the processor. The displayed parameter may be any of the microscopic metrics defined earlier in the chapter. Thus, a machine plot illustrates the spatial distribution of values of a parameter over all processors. An example of each type of plot, in template form, is shown in Figure 5.1.

This machine perspective distinguishes this approach to presenting performance measurement data. It achieves four essential objectives. First, in this format, a system with hundreds to thousands of processors can be displayed at once. A single plot is a snapshot of performance that captures the whole system at some point in time. A sequence of plots displays the progression of the system over time. On a high-resolution color display, a visually striking picture is produced. Patterns and anomalies in system performance can be visually detected. Hot spots of activity, typically identified as regions colored in red, are immediately



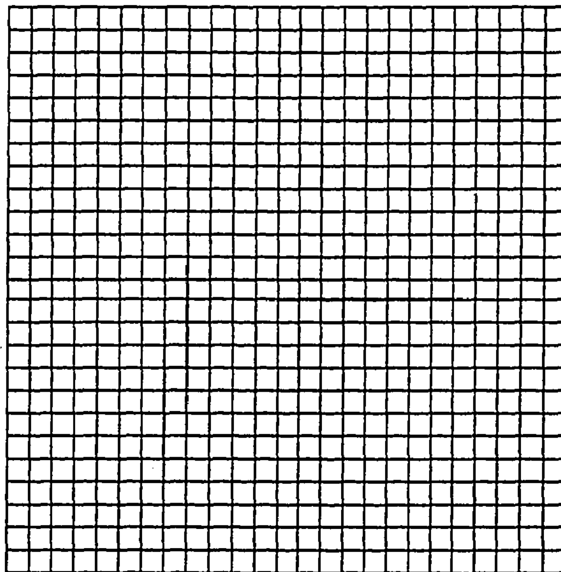


Figure 5.1. Two geometric graphs, in template form, for presenting performance data from a machine perspective: a dot plot and a cell plot

discernible.

Secondly, a two- or three-dimensional plot is appropriate to accurately account for the behavior of the computer system in both time and space. One reason that it is appropriate is that a network of any (logical) dimension must be implemented physically in two or three dimensions. A second reason, which we mention again in the last chapter, is that large, fine-grain multicomputers may require an architecture based on a two- or three-dimensional mesh [Athas and Seitz, 1988]. That is, a mesh will be used for both the logical network and physical network topologies. This development is partly due to VLSI wiring density constraints [Dally, 1987] and to communication latency contraints of higher speed clocks.

Thirdly, a machine perspective facilitates showing the flow or movement of granules of computation and communication throughout the system. Thus, we emphasize both computation and communication activities, and account not only for the time spent in these activities but also for the space used by these activities. Finally, a fourth point is that the format provides a "surface" upon which we can superimpose (or overlay) program and network graphs in order to analyze the mapping between levels.

Plots depict spatial characteristics of performance for the system as a whole. The macroscopic metrics, or summary statistics, defined earlier in the chapter associate single numerical measures with these performance pictures. Profiles, discussed next, emphasize temporal behavior (typically at the expense of spatial behavior).

Profiles

A time-series profile or timing profile (sometimes called a strip chart) depicts the value of a parameter as a function of time. It is a simple line graph with time on the x-axis and the parameter on the y-axis. We define two types of timing profiles: micro-charts and macro-charts, for displaying microscopic metrics and macroscopic metrics, respectively. Micro-charts are useful for closer inspections of individual processor nodes. Macro-charts show trends of summary statistics over time. Both types of timing profiles can be coupled with machine plots (above) to effectively display temporal as well as spatial behavior.

Any of the metrics described in the preceding sections can be graphed via a timing profile. A macro-chart is useful as a pop-up window in conjunction with a plot. A plot illustrates the value of a parameter at each processor at a particular time, and the profile tracks values for a corresponding global parameter over time. When animation is invoked for the plot, the profile provides a global context for the series of plots.

A micro-chart is useful when it is accessible via a hierarchical selection, or zoom-in, mechanism. If more detailed information is desired about a particular region or processor in a displayed plot, that region or processor can be highlighted. A profile window can then be opened for an individual processor. Additional details about the highlighted region can also be selected via a pull-down menu, including information about the channels, the node, the network, and the program.

CHAPTER VI.

PROTOTYPE IMPLEMENTATION

The Dodo said to Alice: "... the best way to explain it is to do it." First it marked out a race course, in a sort of circle, and then all the party were placed along the course, here and there. There was no "One, two, three, and away!", but they began running when they liked, and left off when they liked, so that it was not easy to know when the race was over.

Lewis Carroll, from Alice's Adventures in Wonderland

A simulation-based prototype implementation demonstrates the feasibility of the prescribed approach for representing performance measurement data. It is not specifically targeted to any particular architecture or machine. Rather than using actual instrumentation, a simulator was developed that generates the event traces for programs executing on a (possibly large) hypercube multicomputer. The event traces are processed and then graphically displayed in several formats. This chapter describes the simulation environment, the graphics software, and the synthetic application programs. Pictures of performance, merely frames of an animated story detailing program performance, are shown.

Simulation

The simulation environment is a simplified version of the fully equipped laboratory outlined in Chapter V. In most respects, it provides the essential features of the

laboratory. The simulator itself merely supports the primary objective which is to investigate data presentation. The choice of this objective is partly due to the need to focus on just one aspect of this expanding area of study and to the increasing importance of data visualization. Also, it is an obvious one because of our presently limited parallel computing resources. Hence, the simulator encompasses the hypercube, the monitor, and the application program components of the proposed laboratory. It is a minimal implementation that yields representative event traces describing program execution; note that we are not concerned at this time about verifying the correctness of the simulator and event traces with respect to an actual implementation. Our purpose is to generate event traces with data similar to the data that would be found in actual event traces from program execution on an instrumented hypercube multicomputer. To this end, we have been successful, based on comparisons with event traces from the Seecube software package [Couch, 1988].

The simulator consists of several modules of code. At the top-level, the user provides information about the multicomputer and the program. A hypercube architecture is assumed in the prototype, so all that is required as input is the dimension of the cube. The user is given a menu of synthetic application programs that are available in the program library. The program library includes collective communication routines and complete application programs. These synthetic programs drive the simulation according to some computation or communication paradigm. The collective communication routines include:

- 1-D Shift (or Rotate)
- 2-D Shift (or Exchange)
- Broadcast
- Collect
- Combine (or Global Exchange)

Because of the importance of efficient communication for multicomputers, these routines are becoming fairly standard communication algorithms. They are documented in [Fox et al., 1988], [Gustafson et al., 1988], and [Geist et al., 1989], among other sources. The applications currently in the program library include:

- Quicksort (Divide and Conquer)
- 1-D Wave Equation (Domain Decomposition)
- 2-D Laplace's Equation (Domain Decompostion)
- 1-D Potential Energy Problem (Domain Decomposition)

The concurrent computation paradigm is noted in parentheses. The load per processor changes logarithmically (with respect to time) in the Quicksort application program, and it remains constant (over time) in the other applications. In other words, these programs have a regular structure in terms of computation and communication. Programs with an irregular (less predictable) structure that may require dynamic load balancing and dynamic message routing strategies are beyond the scope of this prototype implementation. These programs are based on fairly standard algorithms and serve the purpose of illustrating changing amounts of computation and communication over time and space during program execution. They are documented in [Fox et al., 1988] and [Gustafson et al., 1988], among other sources.

Since the simulation is event-driven, the synthetic programs access more basic event-specific routines. The event-specific routines update the activities of the

processors in the system as dictated by the program being executed. The updates occur in sweeps across the system, with each sweep advancing the local clocks on each processor. The local clocks, though maintained individually, are synchronized when communication-related events occur.

Figure 6.1 depicts the event-driven simulation and the recording of events. Computation-related events change and record the amount of work being done by each processor, and communication-related events change and record the amount of traffic local to each processor. The event-specific routines also function as the monitor and include calls to logging routines that store monitoring data when events occur. The following activities (along with any supporting information) are recorded in the present version: (1) idle (or no activity), (2) compute, (3) interprocessor send, (4) interprocessor receive, (5) wait to receive, (6) input, and (7) output.

The event records are logged to a single trace file. In the interests of time and storage constraints in the context of the simulation environment of the prototype, the format of an event record is as basic as possible. It consists of the following fields: (1) time of event, (2) processor address, (3) new state (or activity) resulting from the event, (4) work (in operations), and (5) traffic (in bytes). Even with the relatively simple programs being monitored in this prototype, trace records are generated frequently and the trace file grows quickly. The 1-D Wave Equation program from the library creates a 200,000-byte file when run in its most limited form. Trace files that we have inspected from the Seecube software package are in excess of one million bytes [Couch, 1988].

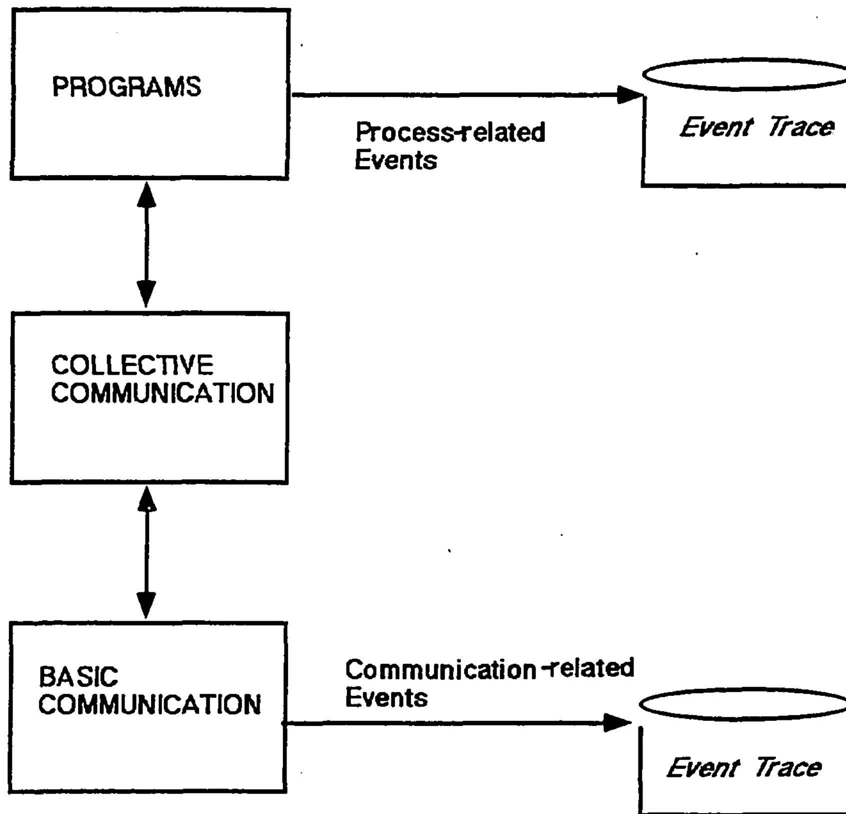


Figure 6.1. Event-driven simulation and generation of event records

Since time and storage are required during each phase of processing the trace file, including creation, analysis, and display, limiting the overhead becomes important.

Although it is important in any implementation, excessive overhead is especially noticeable in a prototype built with modest components. An actual implementation would require a high-performance workstation (for example, featuring a 32-bit processor, 25 megahertz clock, and 70 megabyte hard disk) with possible hardware support for graphics. The present setup for the prototype implementation consists of an IBM PC/AT personal computer (16-bit processor, 12 megahertz clock, and 20 megabyte hard disk), an Apple Macintosh-II personal computer (color graphics), and serial communications via a host computer (to upload and download data between the two personal computers). The PC/AT is used as the simulation and analysis engine, while the Mac-II is used as the graphics engine. Simulation and analysis software is written in Pascal and runs on the PC-AT. Preliminary versions of the graphics software were written in Pascal for the PC/AT. However, the current graphics software tools, described in the next section, are commercially-available packages that run on the Mac-II.

The following steps are performed via the PC/AT to generate monitoring data to be displayed by the graphics software tools:

(1) Invoke the simulator.

(2) Select a synthetic program.

(3) Wait for the simulation to complete (post-processing of measurement data).

(4) Sort the trace file according to time of event.

(5) Map the processor addresses to physical grid

coordinates.

(6) Obtain any desired microscopic or macroscopic statistics (some statistical measures may be calculated later via the graphics software).

(7) Transform the trace file (monitoring data) and other measurement data into the data formats required by the graphics software tools (if needed).

(8) Transfer the data to the graphics engine.

For the mapping step, the hypercube network maps into physical two-dimensional and three-dimensional space via a standard gray code mapping of processor addresses to Cartesian coordinates [Fox et al., 1988]. Alternative mapping functions can be used, and constitute an avenue for future work. An example of a gray code mapping is given later in this chapter.

The following steps are performed via the Mac-II to graph the monitoring and measurement data:

(1) Capture the data from the simulation and analysis engine.

(2) Invoke the graphics software tools and select the desired data presentation formats.

The steps that comprise the post-processing of the event trace are summarized in Figure 6.2.

Graphics Software

As discussed in the previous chapter, the greatest power to evaluate program performance comes from applying graphical data analysis software for visualizing the measurement data. The graphics software of the prototype was selected to support the performance data presentation needs of complex computer

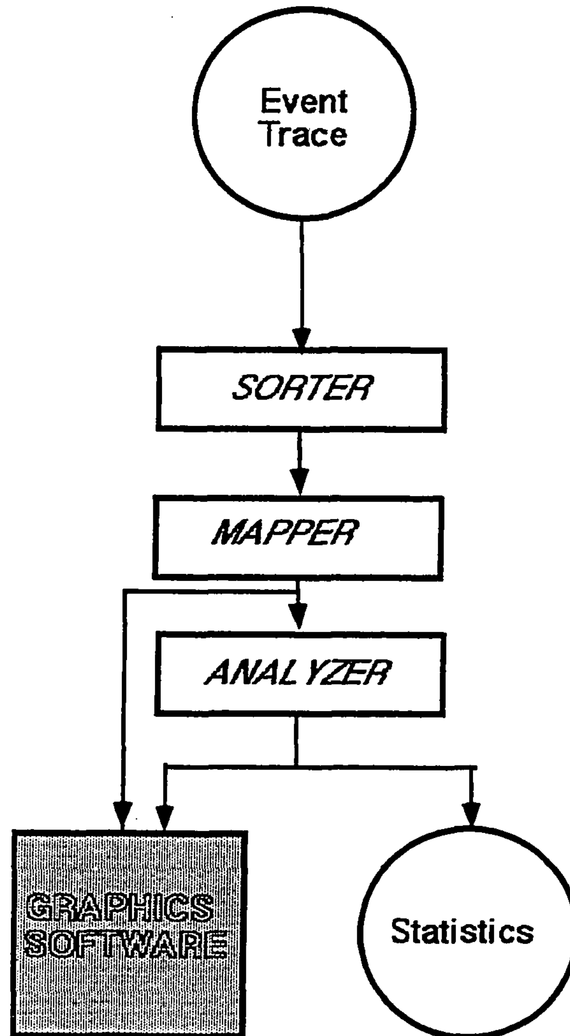


Figure 6.2. Post-processing of an event trace

systems. In particular, the two types of graphics prescribed for pictures of performance, profiles and data plots, are incorporated into a visual analysis tool. An example of the graphical interface of the visual analysis tool is illustrated in Figure 6.3.

The tool supports a hierarchical presentation of data, which reduces the apparent complexity of system performance. That is, the user has the opportunity to view the activity of the system as a whole and also to selectively view the detailed activities of individual processors. The image window provides a global view of the system. Via a selection mechanism, a particular processor can be highlighted. Data specific to the highlighted processor can then be accessed via a pull-down menu. A profile window, which plots the value of a local performance parameter over time via a time-series profile, may be opened for the highlighted processor. If available, information about the portion of the program or network associated with the processor can also be accessed.

With the current emphasis on visualizing scientific data, we have found several graphical software packages that, when used in combination, meet specific requirements of the visual analysis tool. Of course, these packages are separate from the monitoring system. In one sense that is good. It ensures an objective view of the monitoring and measurement data (at least beyond the collection and analysis stages) and indicates that performance monitoring data can be treated in as general a fashion as any other complex data set. Ideally, and ultimately for speed and simplicity, we want the graphics to be integrated in some way with the monitoring system. However, the graphical software packages are more than

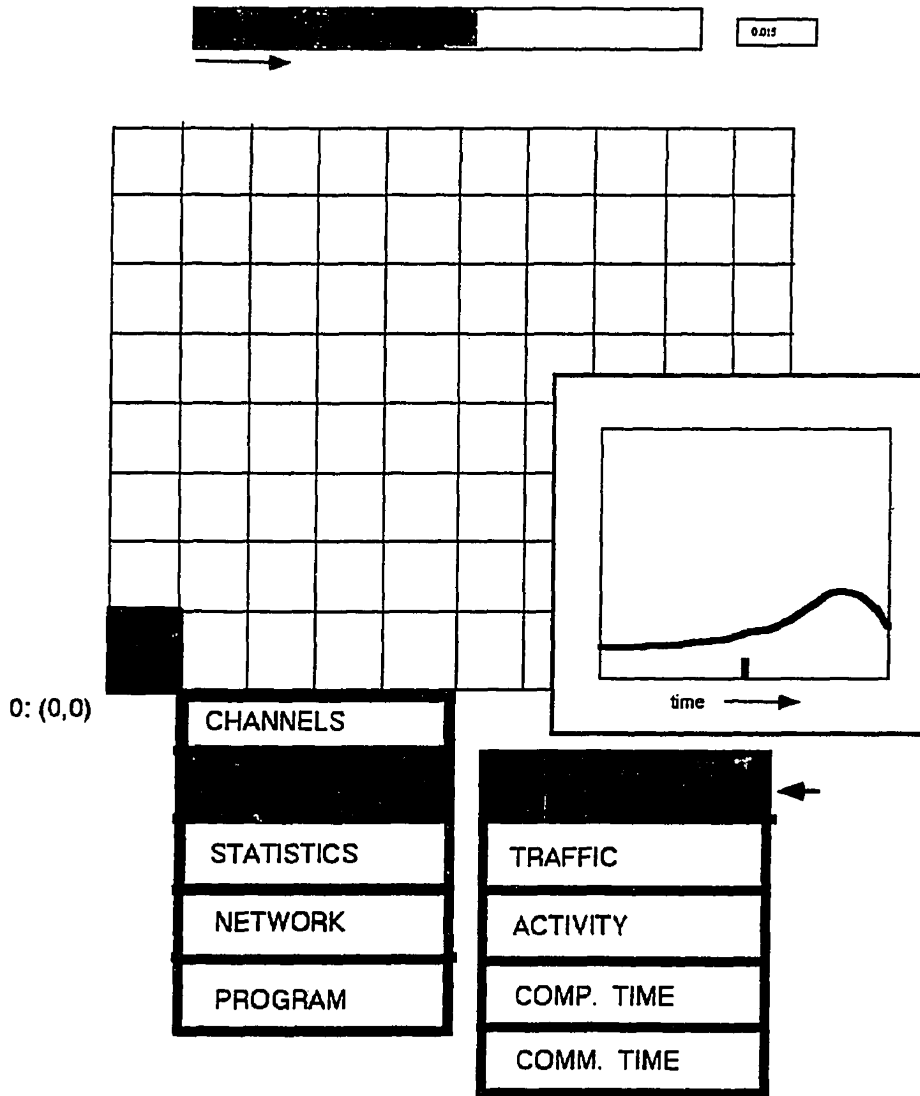


Figure 6.3. Graphical interface of the visual analysis tool

sufficient to illustrate the kinds of data that can be displayed, the utility of different types of data plots, and the power of this approach to visualizing program performance.

Four graphical data analysis software packages are used on the Mac-II. They are:

(1) StatView (Abacus Concepts Inc.)

(2) MacSpin (D^2 Software Inc.)

(3) NCSA DataScope

(4) NCSA Image

NCSA is the National Center for Supercomputing Applications at the University of Illinois, Urbana-Champaign.

These graphics tools display multivariate (numerical) data and provide different views and options for analysis. The first three packages accept text files of data as input. StatView and MacSpin only require a tabular format for input data where the table consists of labeled columns of data. DataScope requires additional header information describing the data and specifies a row-column spreadsheet format for the data. Though it offers several unique functions, DataScope primarily facilitates using Image, a graphics software package with enhanced plotting options. Via DataScope, we can store a data file in a specialized format called the Hierarchical Data Format (HDF, also a product of NCSA). Image reads HDF files but not text files.

StatView supports line and bar graphs. It is used to create profiles of individual processor performance or system performance over time. MacSpin is a powerful tool that supports the data plots referred to (in Chapter V) as dot or scatter plots. Data may be plotted in two or three dimensions, one variable (or parameter) may be selected and

used to color-code the dots, and one variable may be selected and used for animation. MacSpin is a natural choice for displaying the time-ordered event traces.

Image is also very powerful; it supports the data plots referred to (in Chapter V) as cell or block plots. Within Image, these data plots are called raster graphs. Data is plotted in two dimensions (two independent variables), and the value of the variable of interest (the dependent variable) is denoted by color. Rather than drawing discrete blocks of pixels on the screen (an option within DataScope), Image interpolates the data via a local averaging operation to create a smoother picture. Other plot options give alternative representations to replace the use of color, including contour plots, 3D plots, shaded plots, and dither plots. Image works well to display the performance of the system at a particular moment in time. It is especially suited to show intensity (relative value within a range of values) variations of some metric. The displayed variable may be based on processor state (or activity), amount of work, or amount of traffic, among others.

Examples of the graphs are shown in the next section when presenting the case studies. One point is worth reiterating. Color is often an important feature of the graphics for visualizing data. In some cases, animation is also important. Unfortunately, while both color and animation are easy to show on a computer's video display screen, these features are difficult to reproduce on paper. So, though some of the meaning inherent in color or animation may be represented via other forms on paper, certain qualities can only be perceived and appreciated on screen (or possibly via color photographs

of the screen).

Case Studies in Visualization

Post-processing of the event traces generated by the simulator transforms measurement data into snapshots of system performance, or system states, that can be statistically analyzed and graphically displayed.

System configuration

For the case studies presented in this section, the simulator is configured as an eight-dimensional (256-node) hypercube multicomputer. Although the approach to analyzing and visualizing program execution is independent of any specific architecture or machine, we selected the hypercube because of its popularity and commercial success thus far.

There is nothing magical about the number of processor nodes, 256; smaller or larger numbers may be used. Smaller numbers of processors result in less overhead and allow us to inspect the trace files and evaluate whether the simulator and synthetic programs are working as expected. However, larger numbers of processors are the intended target. Unfortunately, they simply result in too much overhead for the resources comprising the prototype. Consequently, 256 processor nodes is a good compromise. It is large enough to be interesting (and thus requiring more sophisticated data presentation) but not so large to be impractical.

The simulated hypercube is configured with the following hardware parameters:

- time per floating point operation = 15 microseconds

- time per integer operation = 1 microsecond
- time to initiate a message transfer = 400 microseconds
- time per byte in a message transfer = 2 microseconds

These times are taken from an actual NCUBE hypercube machine [Gustafson et al., 1988].

The logical network of the 256-node hypercube is mapped onto a two-dimensional grid (physical network) using a standard gray code mapping operation [Fox et al., 1988]. Figure 6.4 illustrates this mapping for the simplified case of an eight-processor (three-dimensional) hypercube. Processor numbers in the logical network are associated with grid coordinates in the physical network and are assigned to grid positions. For the 256-node hypercube, the assignment of logical processor numbers to locations in the grid is specified in Figure 6.5. Observe that Processor 0 is mapped to the lower left cell at location (0,0). Its hypercube neighbors reside either in row 0 or in column 0, and include processor numbers 1, 2, 4, 8, 16, 32, 64, and 128, as illustrated. Also observe that hypercube nearest neighbors may not be physically near neighbors.

The following five synthetic programs from the program library are studied in this section.

- Broadcast
- Collect
- 1-D Shift (or Rotate)
- Quicksort
- 1-D Wave Equation

The first three programs implement collective communication algorithms. Recall, collective communication is communication in which all processor nodes in the system concurrently

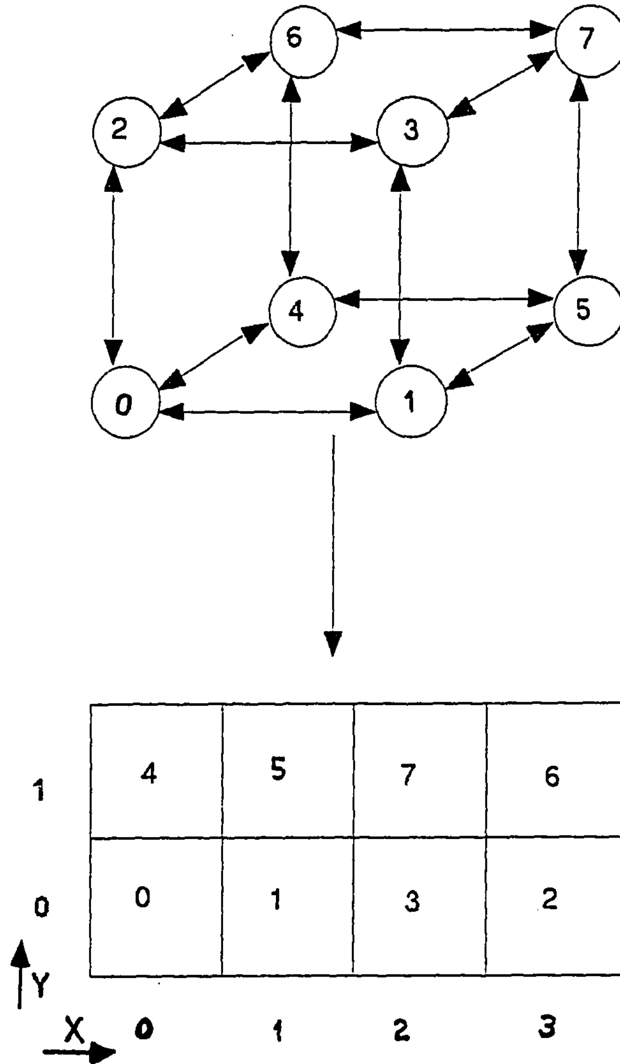


Figure 6.4. Mapping a three-dimensional (eight-node) hypercube onto a two-dimensional grid (gray code mapping)

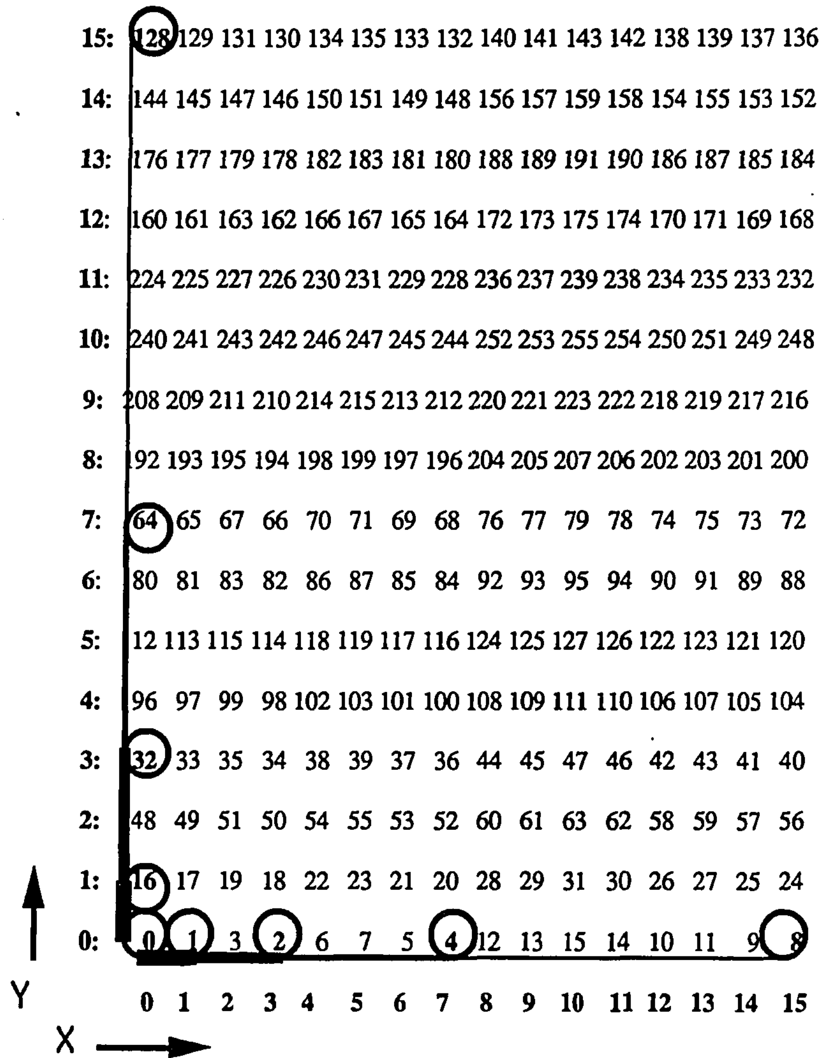


Figure 6.5. Assignment of processor addresses for an eight-dimensional (256-node) hypercube to locations in a two-dimensional (16x16) grid (gray code mapping)

interact in message-passing activities to achieve some degree of global exchange of information. The last two programs are applications, involving computational as well as communication activities. Each program was executed by the simulator to obtain an event trace file. For the case studies, we have primarily reconstructed a sequence of system states via snapshots of system performance at particular times.

Observations, or event records, resulting from each simulation were categorized according to a distribution of ten uniform time intervals spanning the execution time of the simulation. Snapshots of system performance were created at particular times during each time interval in direct proportion to the number of observations recorded for the time interval. The objective was to capture and illustrate the salient aspects of performance via representative snapshots while minimizing the time, storage, and computational effort involved. Selected statistical and graphical representations of performance are reported.

Broadcast communication program

Broadcast is a collective communication routine in which one node receives data from the host computer and initiates a distribution of this data to all other nodes. A common broadcast algorithm distributes the data using a tree-like processor communication graph. The basic operation of Broadcast is graphically depicted in Figure 6.6. For illustrative purposes, it is shown for the simplified case of an eight-node hypercube. Processor 0, the top node of the broadcast tree, initiates the broadcast by sending messages to its hypercube neighbors. Note that the simpler eight-

processor system is used to illustrate functionality, while the 256-processor system is the subject system in the simulations and is used to illustrate the visual analysis tool.

In the simulation, performed on a 256-node hypercube, a 100-byte message was broadcast among the processors. Table 6.1 categorizes the observations (event records) resulting from the simulation according to a distribution of ten uniform time intervals. The computation-related events are recorded when processors have completed their portion of the broadcast and indicate the availability of the processors to do work. The communication-related events include interprocessor send and receive operations and processor waits.

Selected global statistics for the system are documented in Tables 6.2 through 6.4. Table 6.2 is a key for the other tables, pairing global statistics with identifying alphabetic characters. Table 6.3 corresponds to snapshot number 6 (of 12) at time 0.0072 seconds. Table 6.4 corresponds to snapshot number 10 (of 12) at time 0.0144 seconds. Local statistics can be calculated for individual processors. Tables 6.5 through 6.7 document selected local statistics for two processors at particular times. Table 6.5 is a key for the other tables, pairing local statistics with identifying alphabetic characters. Processor 0 is detailed in Table 6.6, and Processor 100, in Table 6.7.

Images of program execution (generated by NCSA Image) are depicted in Figures 6.7 through 6.21. Figures 6.7 through 6.17 comprise an eleven-frame animated sequence of program states, corresponding, respectively, to the following snapshot times (in milliseconds): 0.9, 1.8, 3.6, 5.4, 7.2, 9, 10.8,

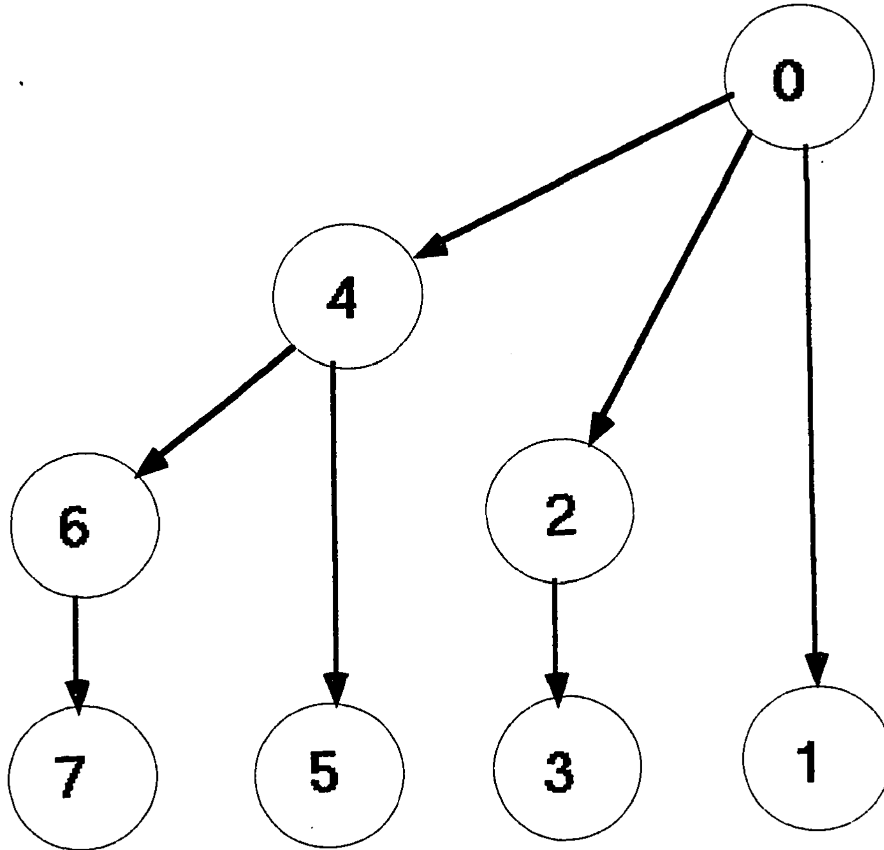


Figure 6.6. Basic operation of Broadcast on an eight-node hypercube

BROADCAST

Number of events: 1017
Total simulation time: 0.018 sec.

Distribution of events:

Period	A	B	C
0	0.0000	1	261
1	0.0018	4	20
2	0.0036	12	43
3	0.0054	26	71
4	0.0072	40	90
5	0.0090	43	84
6	0.0108	52	90
7	0.0126	42	65
8	0.0144	23	30
9	0.0162	10	10

COLLECT

Number of events: 1758
Total simulation time: 0.021 sec.

Distribution of events:

Period	A	B	C
0	0.0000	192	863
1	0.0021	32	242
2	0.0042	16	161
3	0.0063	8	80
4	0.0084	0	59
5	0.0105	4	33
6	0.0126	0	26
7	0.0147	2	15
8	0.0168	0	13
9	0.0189	1	11

QUICKSORT

Number of events: 1785
Total simulation time: 0.031 sec.

Distribution of events:

Period	A	B	C
0	0.0000	1	0
1	0.0031	0	0
2	0.0062	1	2
3	0.0093	0	0
4	0.0124	2	3
5	0.0155	9	13
6	0.0186	82	98
7	0.0217	162	164
8	0.0248	162	158
9	0.0279	91	72

1-D WAVE

Number of events: 6110
Total simulation time: 0.1 sec.

Distribution of events:

Period	A	B	C
0	0.00	51	424
1	0.01	181	324
2	0.02	98	734
3	0.03	170	808
4	0.04	105	859
5	0.05	157	703
6	0.06	6	691
7	0.07	0	632
8	0.08	0	130
9	0.09	0	37

A - Starting time for time period (seconds)
B - Number of computation-related events
C - Number of communication-related events

Table 6.1. Simulation results for the case studies

A : Cumulative work (operation count)
B : Number of computational periods
C : Current work (operations)
D : Cumulative computation time (seconds)
E : Computational power (operations per second)
F : Execution rate (operations per second)
G : Percent of total time spent computing
H : Cumulative traffic (byte count)
I : Message count
J : Current traffic (byte count)
K : Cumulative communication time (seconds)
L : Communication flow (bytes per second)
M : Percent of total time spent communicating
N : Cumulative wait time (seconds)
O : Percent of total time spent waiting
P : Fraction of communication time spent waiting
Q : Communication overhead (communication : computation)
R : Rularity factor (operations of work per byte of traffic)
S : Number of channels in use
T : Percent of channels used (channel utilization)

U : Processor activity distribution (#)
V : Processor activity distribution (%)
W : Processor concurrency (#), utilization (%)
X : Channel concurrency (#), utilization (%)
Y : Computational balance for time, operations
Z : Communication balance for time, bytes

-M, Mega or 10^6; -K, Kilo or 10^3

Table 6.2. Key for global statistics

Execution time = 0.0072 seconds

	Sum (Total)	Average	Maximum
H	7250	28	800
I	73	0	8
J	2000	8	100
K	0.2732	0.00107	0.0072
L	929 K	8.3 K	211 K
M	14.82%	19.14%	100%
N	-	-	0.0072
O	-	-	100%
S	35	0.14	4
T	0.8545%	0.8545%	25%

	NONE	COMPUTE	SND	WAIT	RCV	
U	0	49	9	187	11	
V	0.00	19.14	3.52	73.05	4.30	%

W	256	100.00%
X	35	0.85%
Z	0.1482	0.0354

Table 6.3. Broadcast routine. Selected global statistics for snapshot number 6 taken at 0.0072 seconds

Execution time = 0.014 seconds

	Sum (Total)	Average	Maximum
H	23400	91	800
I	234	1	8
J	4600	18	100
K	2.1272	0.00831	0.014
L	605 K	21 K	211 K
M	59.35%	85.94%	100%
N	2.0848	0.00814	0.014
O	58.17%	80.55%	100%
P	0.9801	0.7553	1.0
S	55	0.21	4
T	1.3428%	1.3428%	25%

	NONE	COMPUTE	SND	WAIT	RCV
U	0	187	25	23	21
V	0.00	73.05	9.77	8.98	8.20 %

W	256	100.00%
X	55	1.34%
Z	0.5935	0.1143

Table 6.4. Broadcast routine. Selected global statistics for snapshot number 10 taken at 0.0144 seconds

A : Snapshot sequence number
B : Time of snapshot (seconds)
C : Execution time of processor (seconds)
D : Action (0=NONE, 1=COMPUTE, 2=SEND, 3=WAIT, 4=RECEIVE, 5=INPUT, 6=OUTPUT)
E : Cumulative work (operation count)
F : Number of computational periods
G : Current work (operation count)
H : Cumulative computation time (seconds)
I : Computational power (rate of doing work, operations per second)
J : Throughput (execution rate, operations per second)
K : Percent of total time spent computing
L : Cumulative local traffic (byte count)
M : Number of messages for interprocessor communication
N : Current local traffic (byte count)
O : Cumulative communication time (seconds)
P : Communication flow (rate of transferring messages, bytes per second)
Q : Percent of total time spent communicating
R : Cumulative wait time (seconds)
S : Percent of total time spent waiting
T : Fraction of communication time spent waiting
U : Communication overhead (communication : computation)
V : Granularity factor (operations of work per byte of traffic)
W : Channel count (number of channels used)
X : Channel utilization (fraction of channels used)

Table 6.5. Key for local statistics

A	B	C	D	L	M	N	O	P	Q	R	S	T	W	X
1	0.0000	0.0000	2	0	0	0	0.0000	-	-	0.0	-	-	0	0.0000
2	0.0009	0.0006	2	100	1	100	0.0000	-	0.00	0.0	0.0	-	1	0.0625
3	0.0018	0.0018	2	400	4	100	0.0000	-	0.00	0.0	0.0	-	4	0.2500
4	0.0036	0.0034	1	800	8	100	0.0000	-	0.00	0.0	0.0	-	8	0.5000
12	0.0180	0.0038	1	800	8	0	0.0038	2.11E+05	100.00	0.0	0.0	0.0	-	-

Table 6.6. Broadcast routine. Selected local statistics for Processor 0, (x,y) = (0,0)

A	B	C	D	L	M	N	O	P	Q	R	S	T	W	X
6	0.0072	0.0000	4	0	0	0	0.0000	-	0.00	0.0000	0.00	-	0	0.0000
7	0.0090	0.0088	2	200	2	100	0.0000	-	0.00	0.0082	93.18	-	2	0.1250
12	0.0180	0.0096	1	300	3	0	0.0096	3.12E+04	100.00	0.0082	85.42	0.8542	2	0.1250

Table 6.7. Broadcast routine. Selected local statistics for Processor 100, (x,y) = (7,4)

12.6, 14.4, 16.2, and 18. The displayed parameter represents the cumulative amount of local traffic, in number of bytes, resulting from message passing activities (sends and receives) of the processor at the indicated time. In these dither plots, lightness denotes relatively low intensity (small) values of the displayed parameter and darkness, high intensity (large) values.

Observe how the traffic of the broadcasted message spreads throughout the system. It is apparent that Processor 0 and its hypercube neighbors, especially those neighbors at the higher levels of the broadcast tree, account for the largest amounts of traffic. The region surrounding Processor 255 is void of any traffic until the end of the simulation, since it is the last node to receive the broadcasted message.

Figure 6.18 shows an alternative type of image, a three-dimensional plot. The actual grid is shown and the third dimension portrays the displayed parameter. This image represents the same program state information as the image in Figure 6.15. Three additional images of program execution at 14.4 milliseconds are depicted in Figures 6.19 through 6.21. In Figure 6.19, the displayed parameter is the cumulative amount of time spent by the processor in communication activities (sends, receives, and waits). Similarities between Figure 6.19 and Figure 6.15 can be observed. The differences arise primarily because processor waits are reflected in Figure 6.19 and show up as higher intensity regions in the image. These regions consist of processors residing at the interior nodes in the broadcast tree. Figure 6.20 illustrates the cumulative amount of time spent in processor waits. Finally, processor activity at the snapshot time is displayed

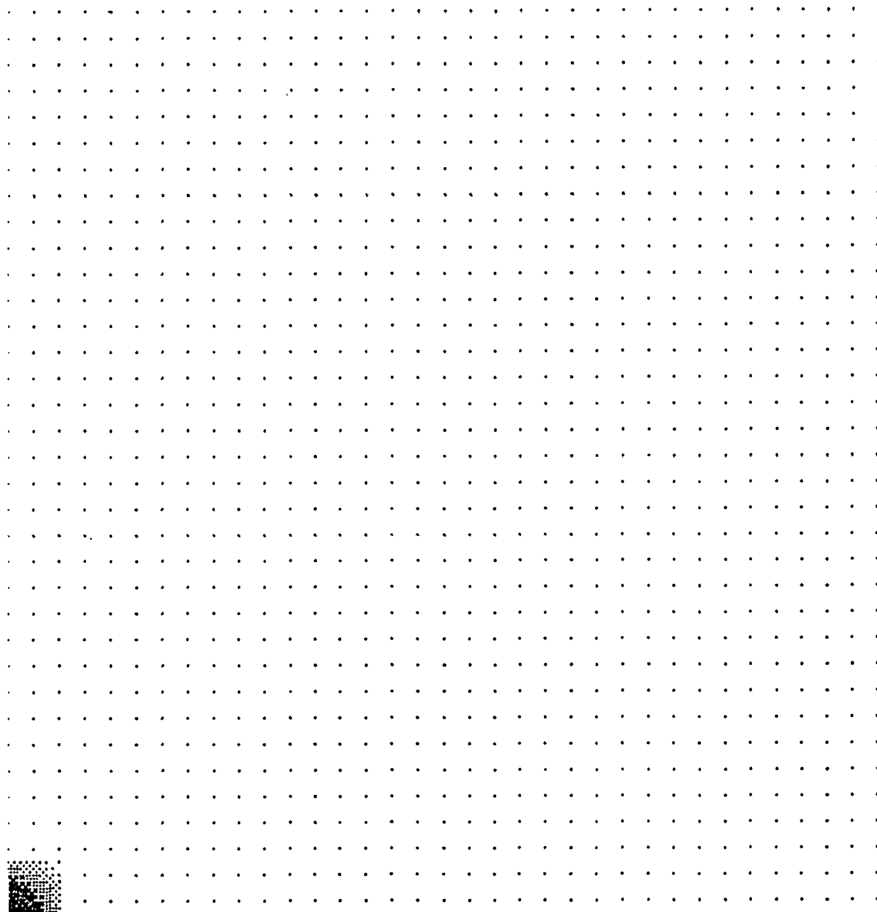


Figure 6.7. Picture of performance (dither plot): Broadcast, ss#2 at 0.9 msec., cumulative traffic (bytes)

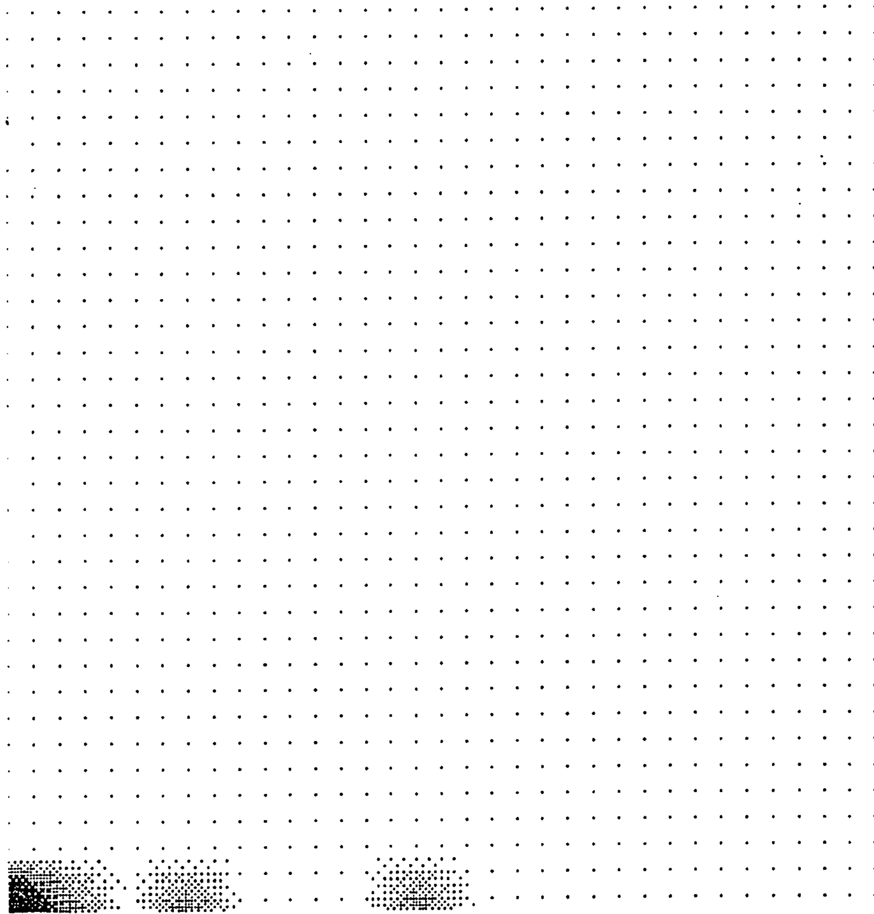


Figure 6.8. Picture of performance (dither plot): Broadcast, ss#3 at 1.8 msec., cumulative traffic (bytes)

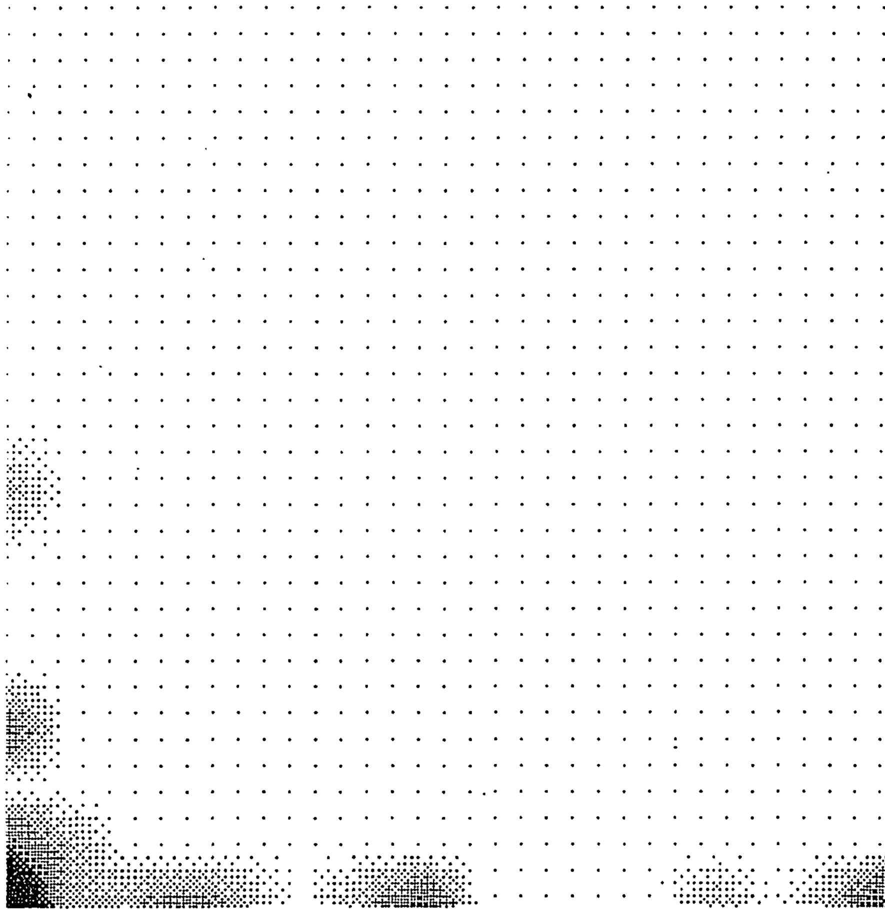


Figure 6.9. Picture of performance (dither plot): Broadcast, ss#4 at 3.6 msec., cumulative traffic (bytes)

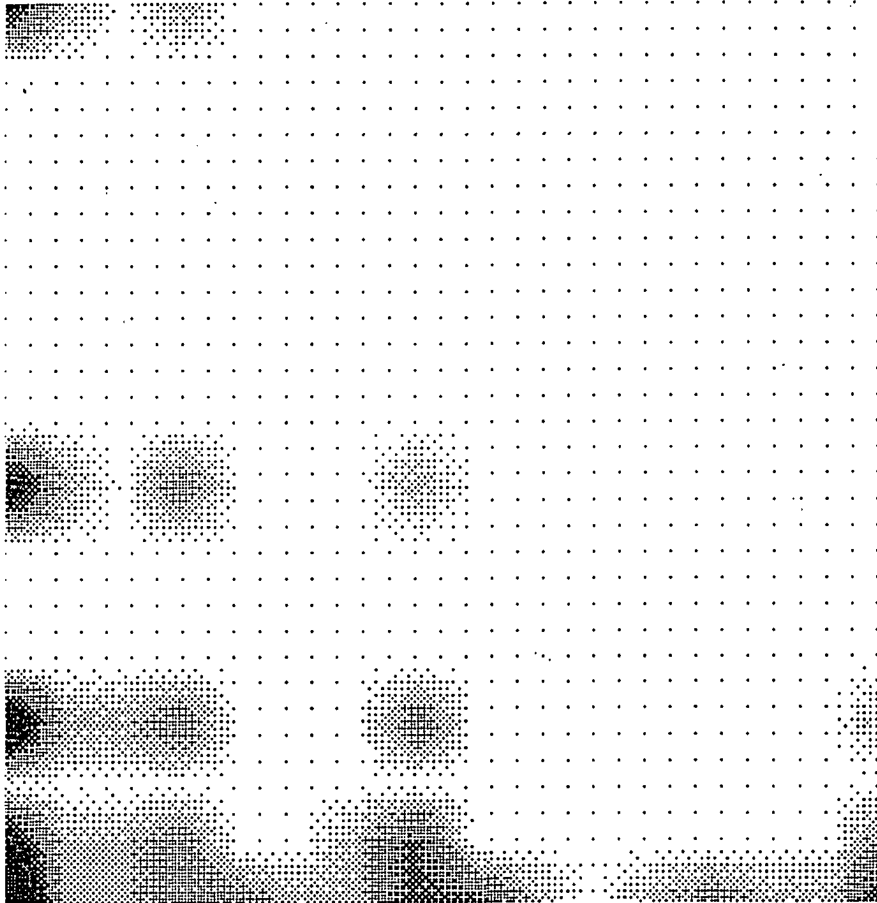


Figure 6.10. Picture of performance (dither plot): Broadcast, ss#5 at 5.4 msec., cumulative traffic (bytes)

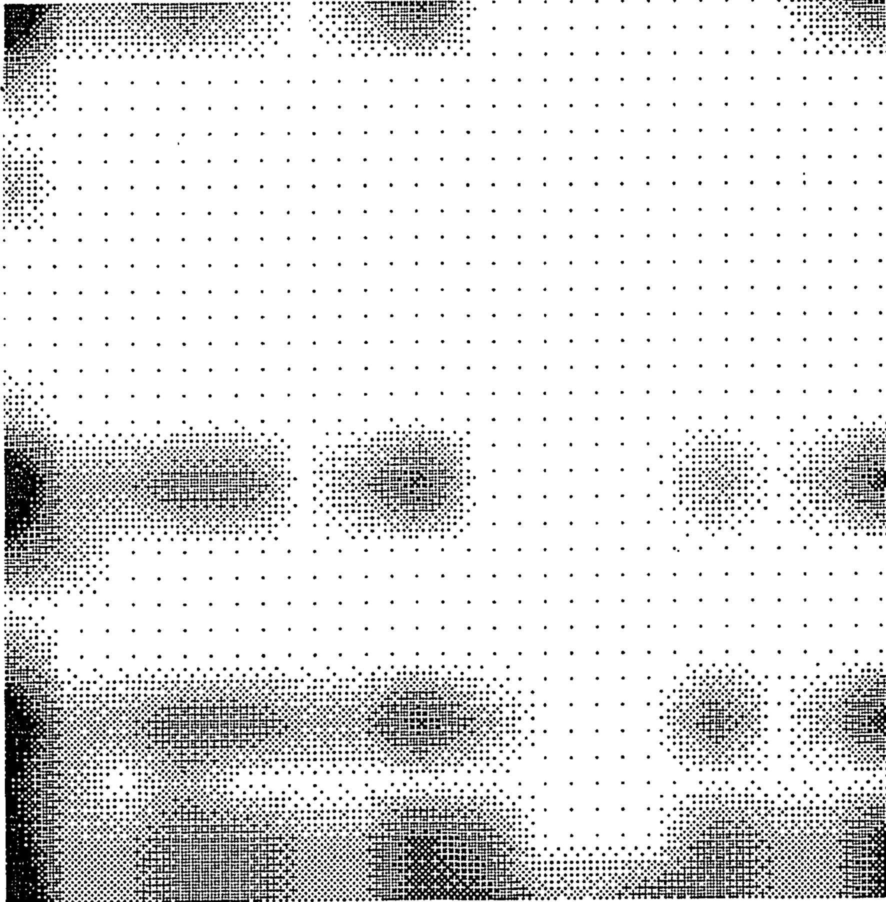


Figure 6.11. Picture of performance (dither plot): Broadcast, ss#6 at 7.2 msec., cumulative traffic (bytes)

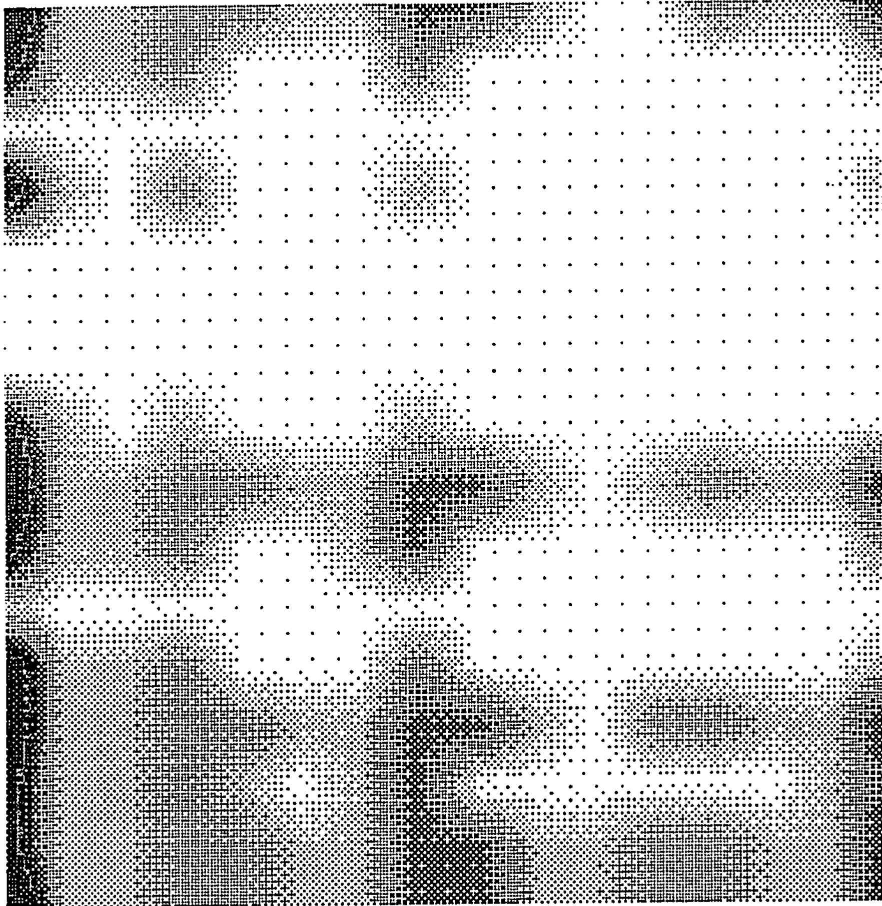


Figure 6.12. Picture of performance (dither plot): Broadcast, ss#7 at 9 msec., cumulative traffic (bytes)

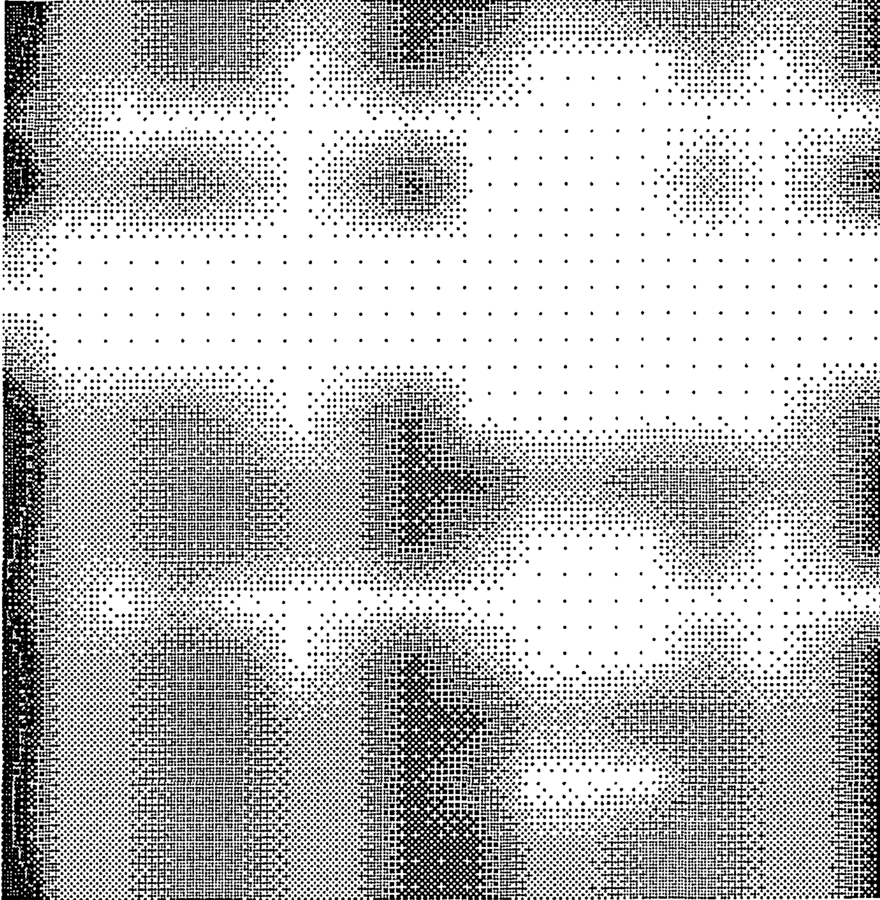


Figure 6.13. Picture of performance (dither plot): Broadcast, ss#8 at 10.8 msec., cumulative traffic (bytes)

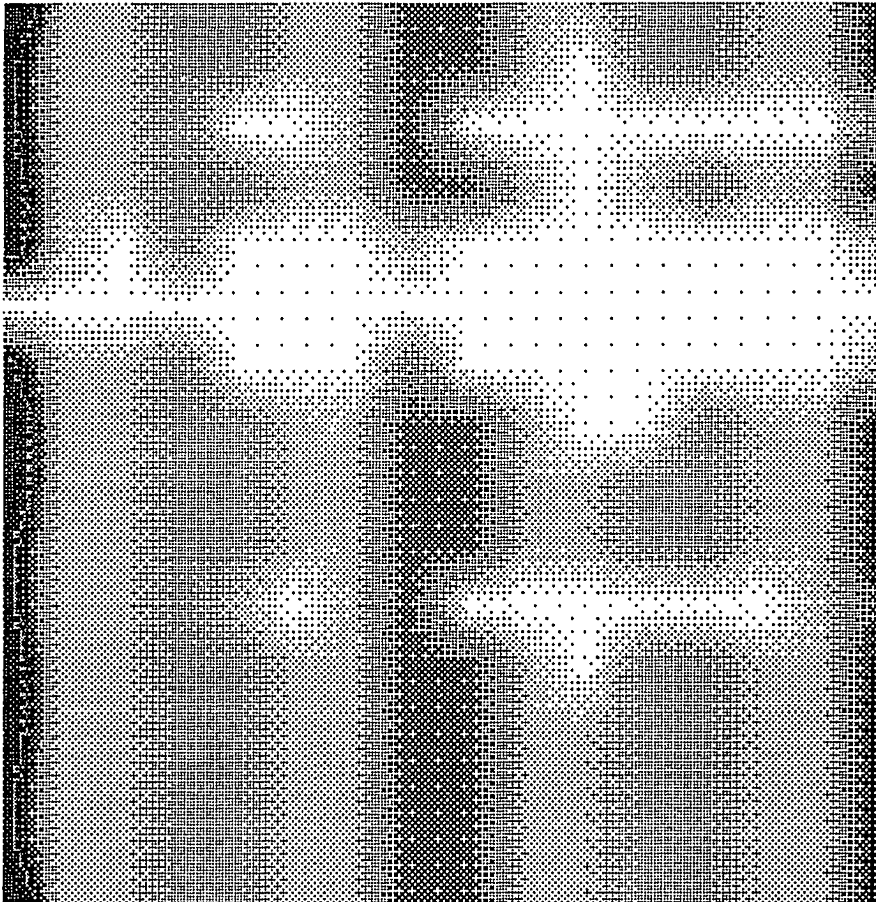


Figure 6.14. Picture of performance (dither plot): Broadcast, ss#9 at 12.6 msec., cumulative traffic (bytes)

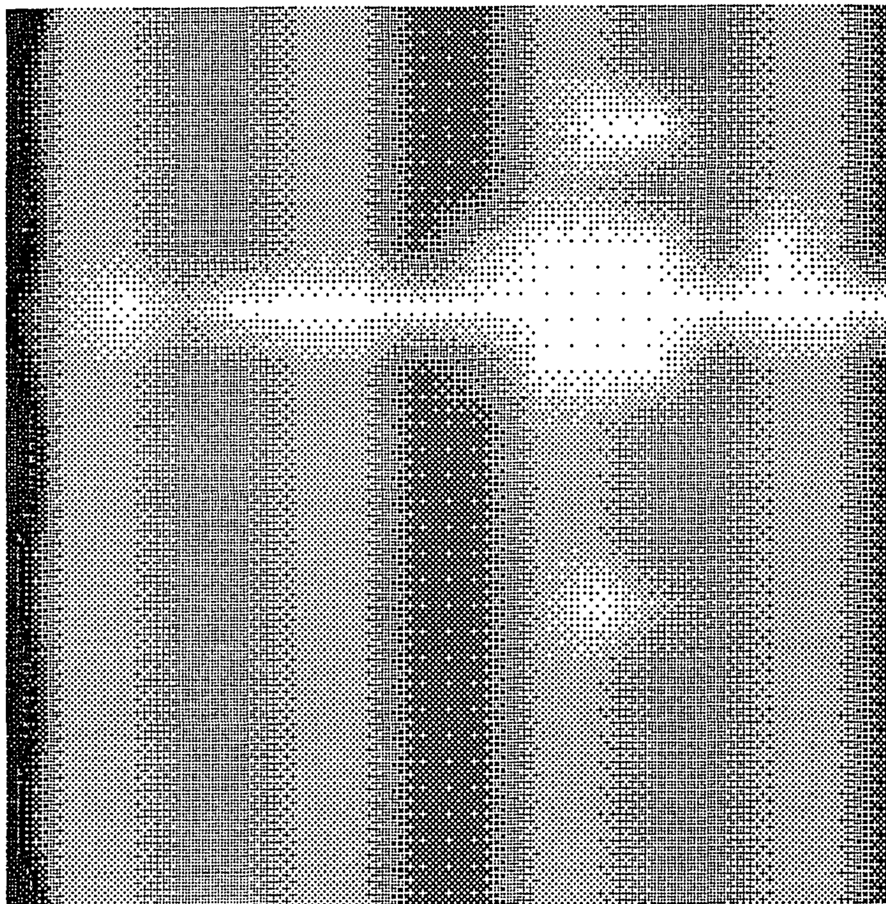


Figure 6.15. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., cumulative traffic (bytes)

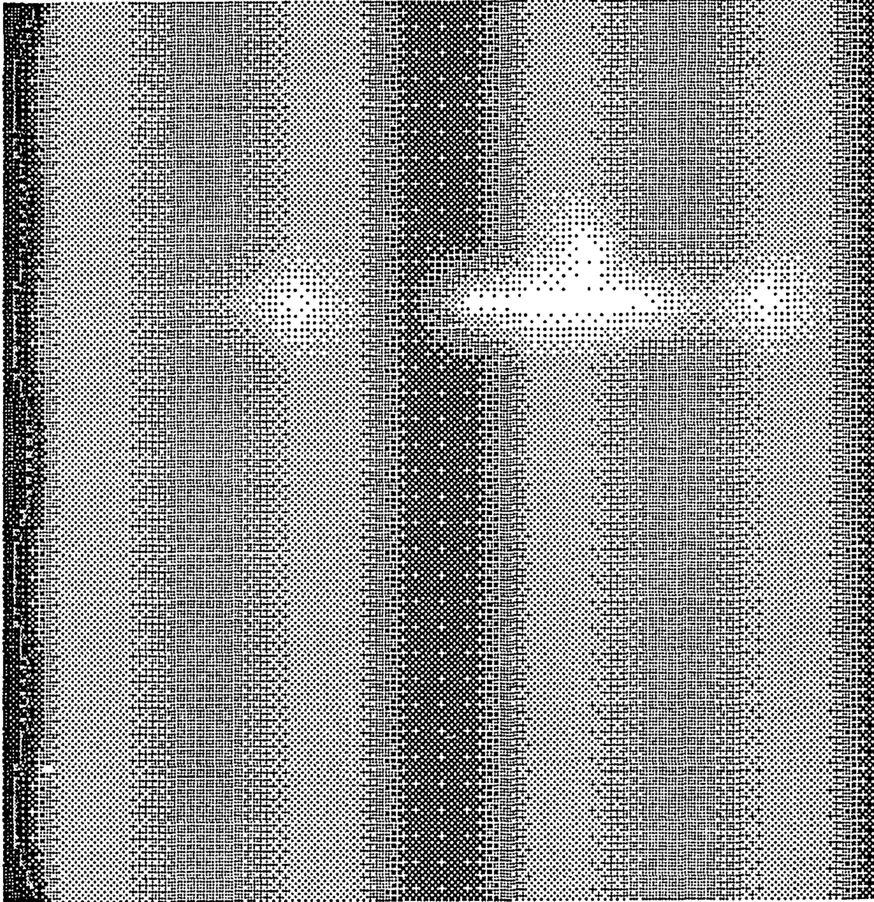


Figure 6.16. Picture of performance (dither plot): Broadcast, ss#11 at 16.2 msec., cumulative traffic (bytes)

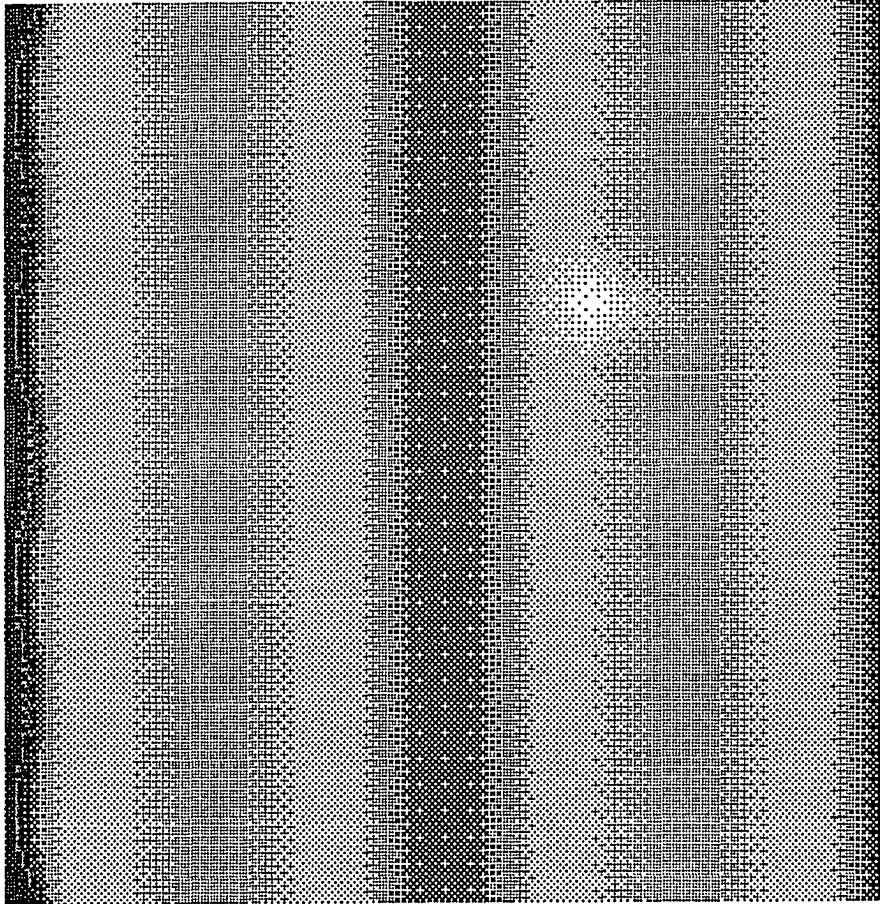


Figure 6.17. Picture of performance (dither plot): Broadcast, ss#12 at 18 msec., cumulative traffic (bytes)

in Figure 6.21. Black denotes computational activity; gray, communication activity; and white, no activity.

Collect communication program

Collect is a collective communication routine in which one node receives data from all other nodes and sends the data to the host computer. A common collect algorithm transfers the data using a tree-like processor communication graph. The basic operation of Collect is graphically depicted in Figure 6.22. For illustrative purposes, it is shown for the simplified case of an eight-node hypercube. Processor 0, the top node of the collect tree, receives messages from all other nodes via its hypercube neighbors.

In the simulation, performed on a 256-node hypercube, a 100-byte message was collected from the processors. Table 6.1 categorizes the observations (event records) resulting from the simulation according to a distribution of ten uniform time intervals. The computation-related events are recorded when processors have completed their portion of the collect and indicate the availability of the processors to do work. The communication-related events include interprocessor send and receive operations and processor waits.

Selected global statistics for the system are documented in Tables 6.2, 6.8, and 6.9. Table 6.2 is a key for the other tables. Table 6.8 corresponds to snapshot number 8 (of 17) at time 0.00315 seconds. Table 6.9 corresponds to snapshot number 15 (of 17) at time 0.0168 seconds. Local statistics can be calculated for individual processors. Tables 6.5, 6.10, and 6.11 document selected local statistics for two processors at particular times. Table 6.5 is a key for the

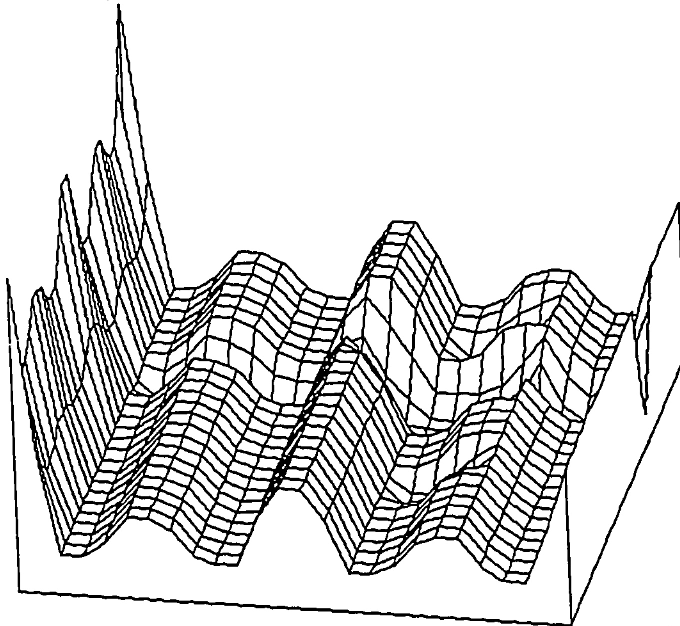


Figure 6.18. Picture of performance (3D plot): Broadcast, ss#10 at 14.4 msec., cumulative traffic (bytes)

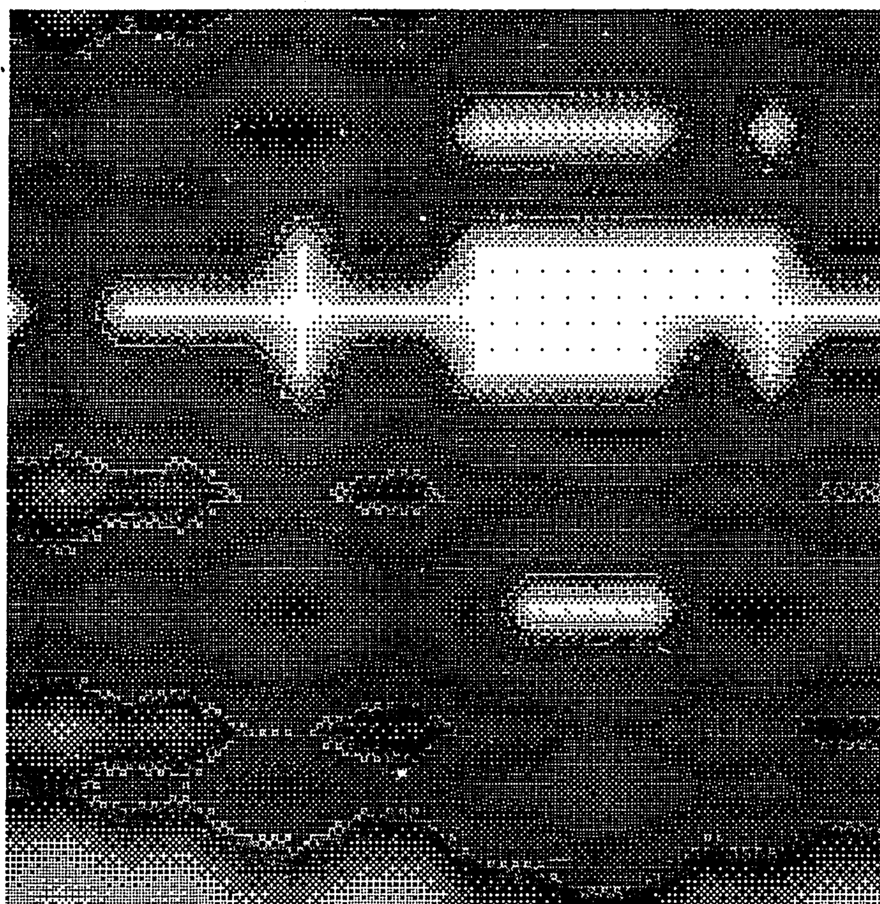


Figure 6.19. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., cumulative communication time

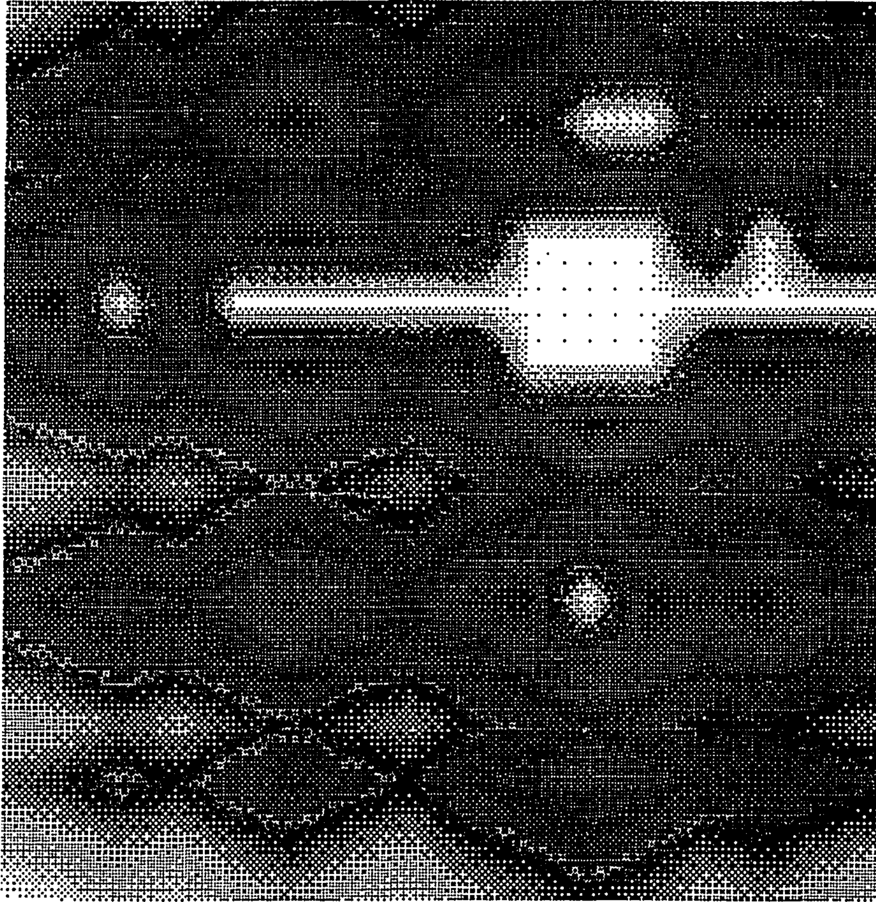


Figure 6.20. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., cumulative wait time

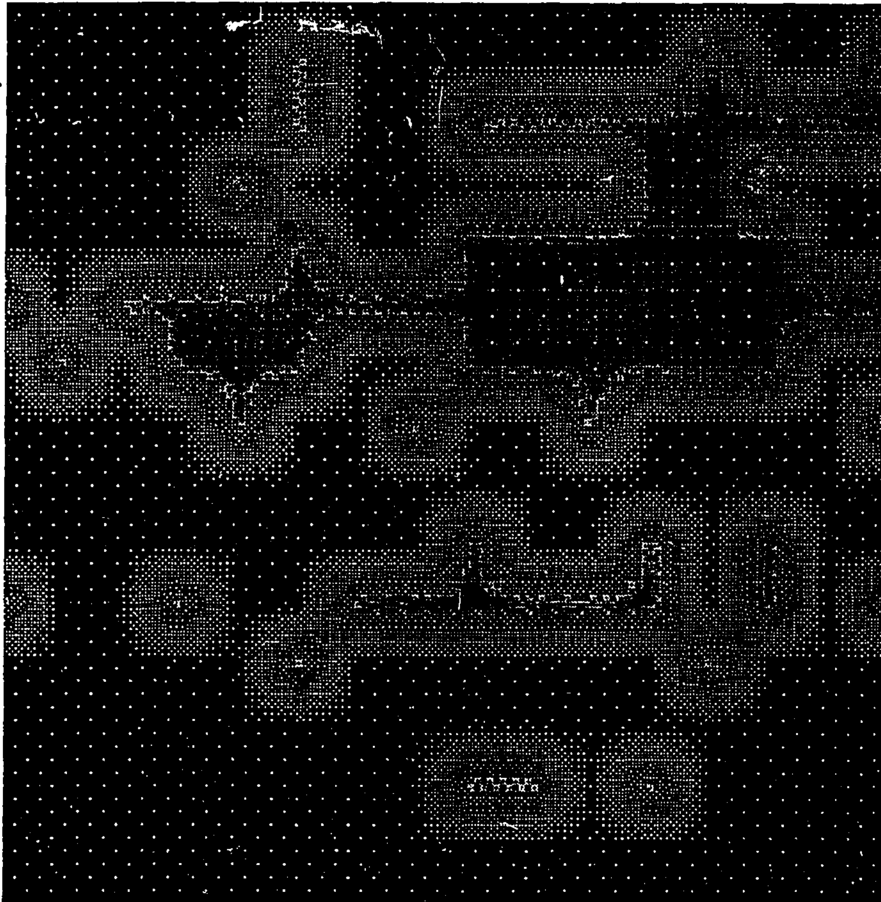


Figure 6.21. Picture of performance (dither plot): Broadcast, ss#10 at 14.4 msec., processor activity (black: computing; gray: communicating; white: none)

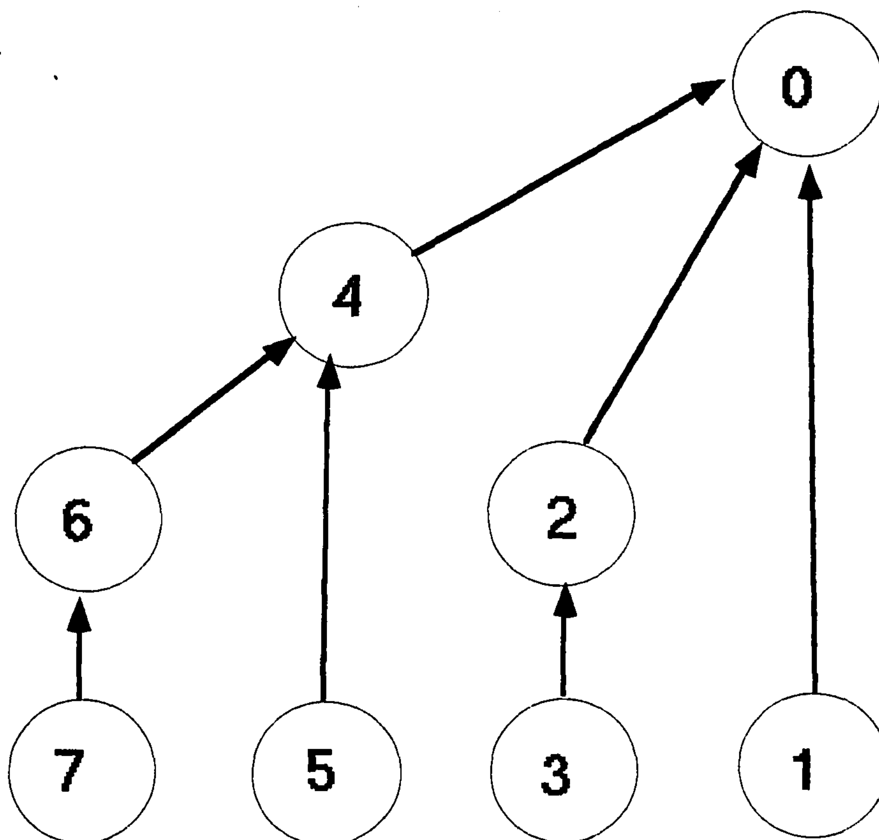


Figure 6.22. Basic operation of Collect on an eight-node hypercube

other tables. Processor 0 is detailed in Table 6.10, and Processor 100, in Table 6.11.

Images of program execution (generated by NCSA Image) are depicted in Figures 6.23 through 6.29. Figures 6.23, 6.24, and 6.29 display the cumulative amount of local traffic, in number of bytes, resulting from message passing activities (sends and receives) of the processor at the indicated time. The first of these three images corresponds to a state of the system near the beginning of the simulation; the second image, near the middle; and the third image, near the end. Very definite patterns can be observed. The vertical shaded regions roughly correspond to levels in the collect tree. It is apparent that nodes at the higher levels account for the largest amounts of traffic. By the end of Collect, hot spots of traffic are found at Processor 0 and its hypercube neighbors at the higher levels of the collect tree.

Figure 6.25 shows a three-dimensional image that represents the same program state information as the image in Figure 6.24. Three additional images of program execution at a snapshot time of 3.15 milliseconds are depicted in Figures 6.26 through 6.28. In Figure 6.26, the displayed parameter is the cumulative amount of time spent by the processor in communication activities (sends, receives, and waits). Similarities between Figure 6.26 and Figure 6.24 can be observed. Figure 6.27 illustrates the cumulative amount of time spent in processor waits. Finally, processor activity at the snapshot time is displayed in Figure 6.28. Black denotes computational activity; gray, communication activity; and white, no activity.

Execution time = 0.003 seconds

	Sum (Total)	Average	Maximum
H	35150	137	600
I	352	1	6
J	3200	13	100
K	0.2368	0.00093	0.003
L	28.4 M	204 K	250 K
M	30.83%	87.5%	100%
N	0.0778	0.0003	0.0024
O	10.13%	13.06%	85.71%
P	0.3285	0.0844	0.6
S	190	0.75	6
T	4.6631%	4.6631%	37.5%

	NONE	COMPUTE	SND	WAIT	RCV
U	0	224	8	15	9
V	0.00	87.5	3.13	5.86	3.52

W	256	100.00%
X	191	4.66%
Z	0.3083	0.2288

Table 6.8. Collect routine. Selected global statistics for snapshot number 8 taken at 0.00315 seconds

Execution time = 0.016 seconds

	Sum (Total)	Average	Maximum
H	49450	193	3000
I	495	2	30
J	600	2	100
K	0.458	0.00179	0.015
L	6.59 M	226 K	250 K
M	11.18%	99.22%	100%
N	0.1424	0.00056	0.008
O	3.48%	12.95%	60%
P	0.3109	0.1257	0.6
S	61	0.24	3
T	1.4893%	1.4893%	18.75%

	NONE	COMPUTE	SND	WAIT	RCV
U	0	250	4	1	1
V	0.00	97.66	1.56	0.39	0.39 %

W	256	100.00%
X	61	1.49%
Z	0.1193	0.0644

Table 6.9. Collect routine. Selected global statistics for snapshot number 15 taken at 0.0168 seconds

A	B	C	D	L	M	N	R	S
2	0.00035	0.0000	4	0	0	0	0.00000	-
3	0.00070	0.0004	3	100	1	100	0.00040	100.00
5	0.00140	0.0010	4	200	2	100	0.00040	40.00
7	0.00210	0.0016	4	300	3	100	0.00040	25.00
8	0.00315	0.0028	4	500	5	100	0.00160	57.14
9	0.00420	0.0040	4	700	7	100	0.00280	70.00
10	0.00630	0.0058	4	1000	10	100	0.00340	58.62
11	0.00840	0.0082	4	1400	14	100	0.00460	56.10
12	0.01050	0.0100	4	1700	17	100	0.00580	58.00
13	0.01260	0.0120	3	2100	21	100	0.00580	48.33
14	0.01470	0.0140	4	2400	24	100	0.00680	48.57
15	0.01680	0.0160	4	2700	27	100	0.00780	48.75
16	0.01890	0.0180	4	3100	31	100	0.00780	43.33
17	0.02100	0.0210	3	3600	36	100	0.00980	46.67

Table 6.10. Collect routine. Selected local statistics for Processor 0, (x,y) = (0,0)

A	B	C	D	L	M	N	O	P	Q	R	S	T
2	0.00035	0.0000	4	100	1	100	-	-	-	0.0000	-	-
3	0.00070	0.0004	3	200	2	100	-	-	-	0.0000	0.0	-
5	0.00140	0.0010	2	300	3	100	-	-	-	0.0006	60.0	-
6	0.00175	0.0016	3	400	4	100	-	-	-	0.0006	37.5	-
7	0.00210	0.0020	2	500	5	100	-	-	-	0.0006	30.0	-
17	0.02100	0.0030	1	600	6	0	0.003	2.00E+05	100.00	0.0006	20.0	0.2

Table 6.11. Collect routine. Selected local statistics for Processor 100, (x,y) = (7,4)

Figure 6.23. Picture of performance (dither plot): Collect, ss#2 at 0.35 msec., cumulative traffic (bytes)

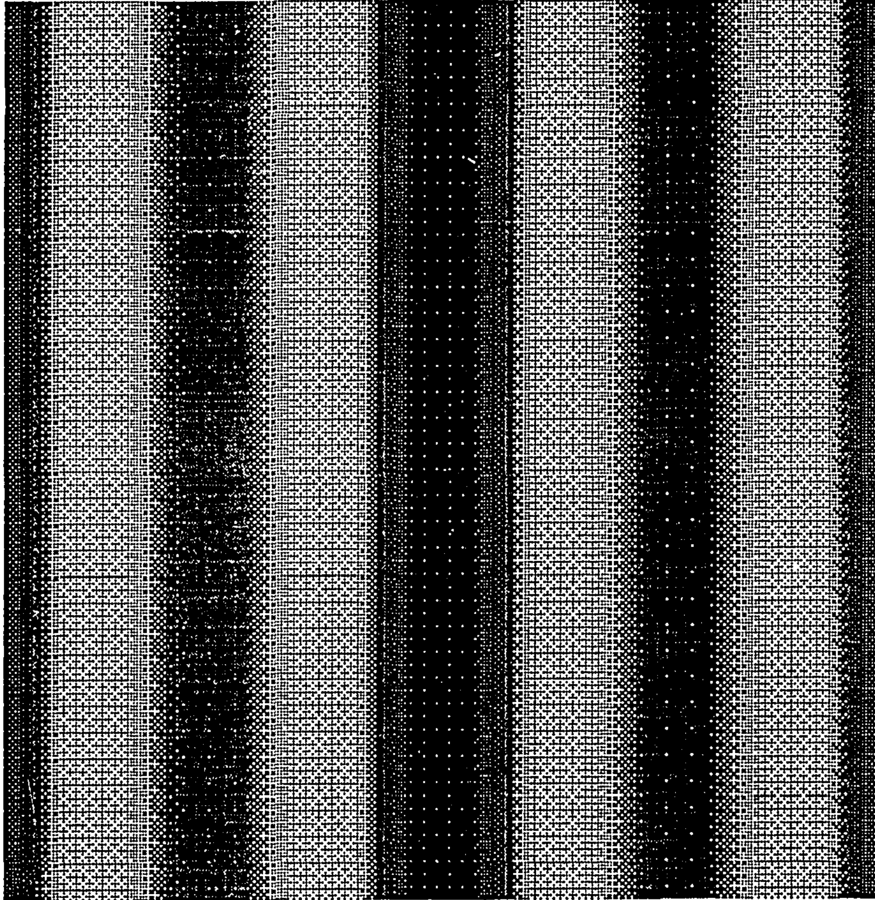


Figure 6.24. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., cumulative traffic (bytes)

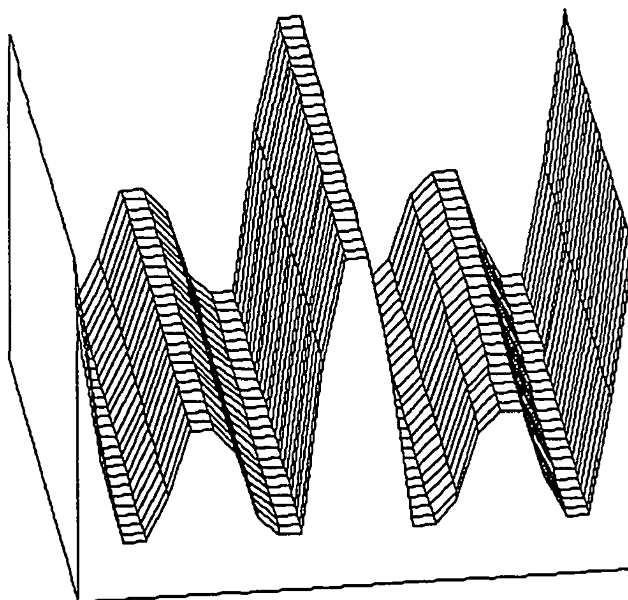


Figure 6.25. Picture of performance (3D plot): Collect, ss#8 at 3.15 msec., cumulative traffic (bytes)

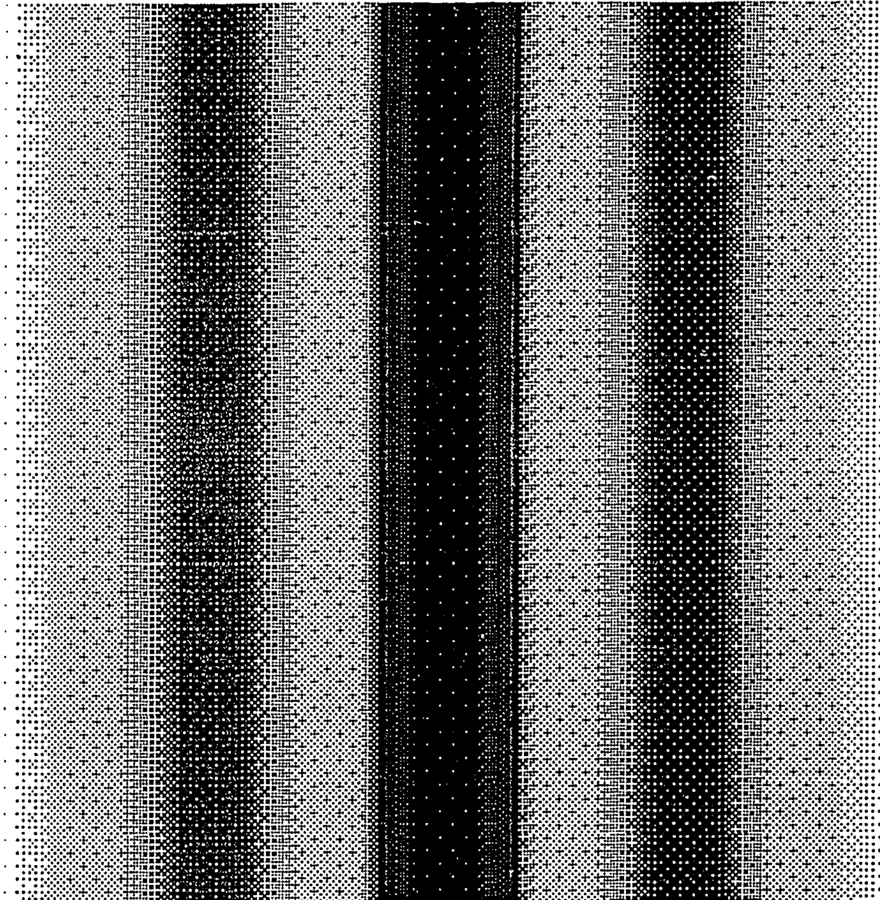


Figure 6.26. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., cumulative communication time

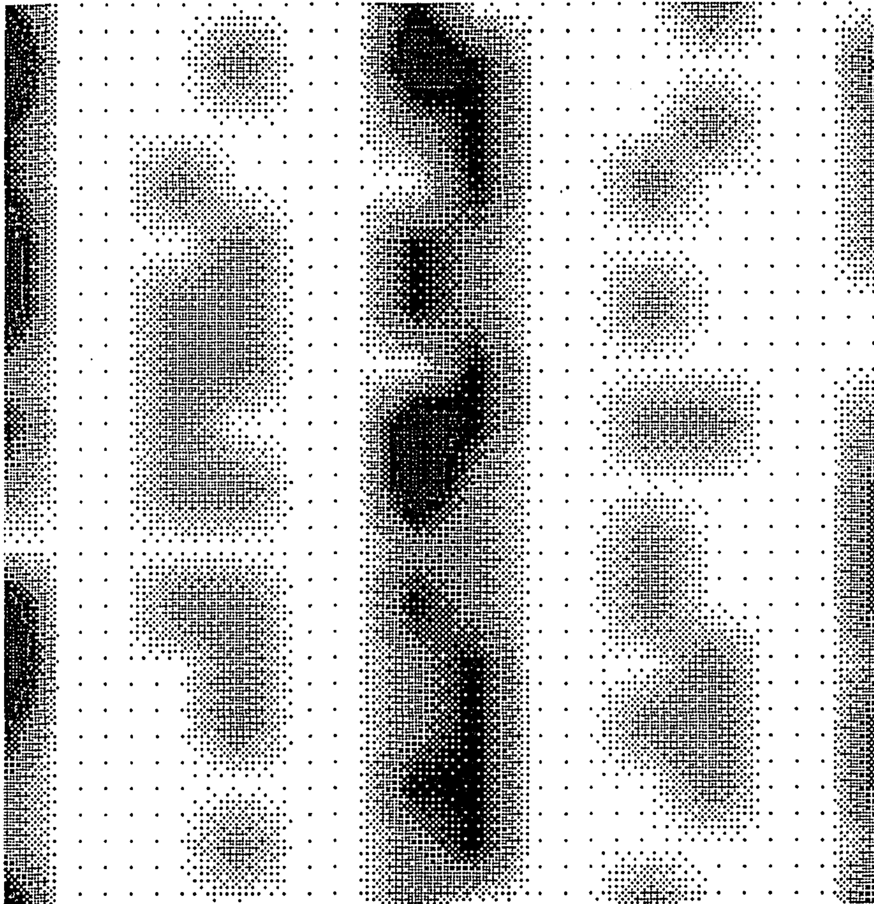


Figure 6.27. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., cumulative wait time

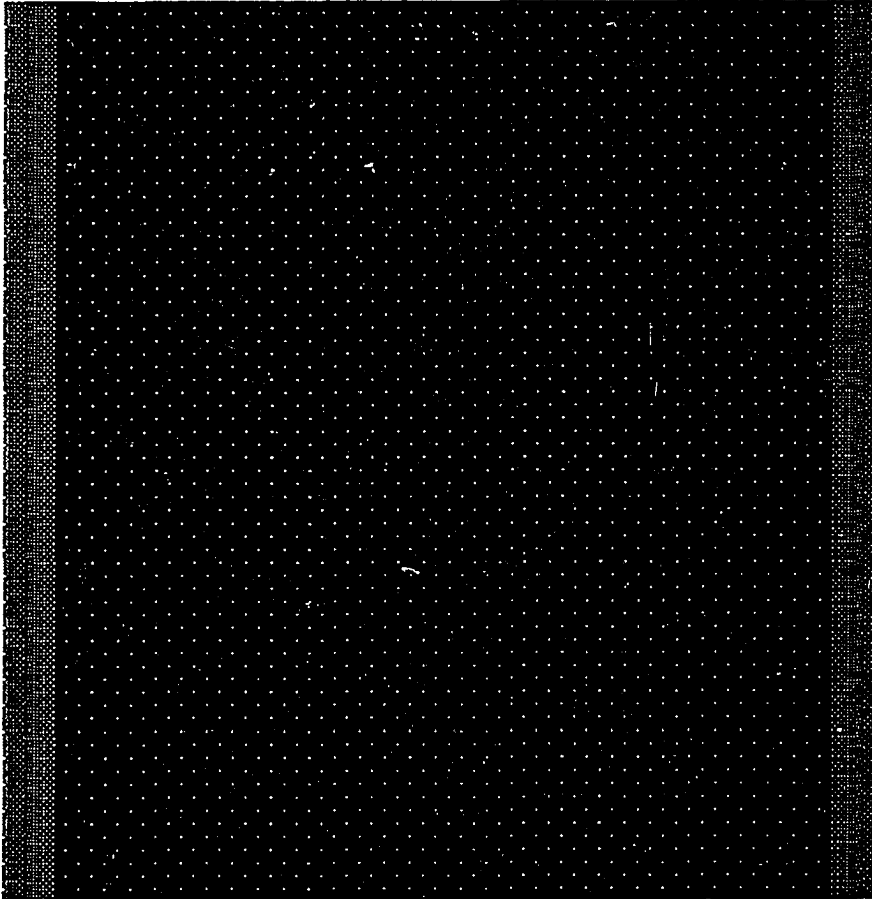


Figure 6.28. Picture of performance (dither plot): Collect, ss#8 at 3.15 msec., processor activity (black: computing; gray: communicating; white: none)

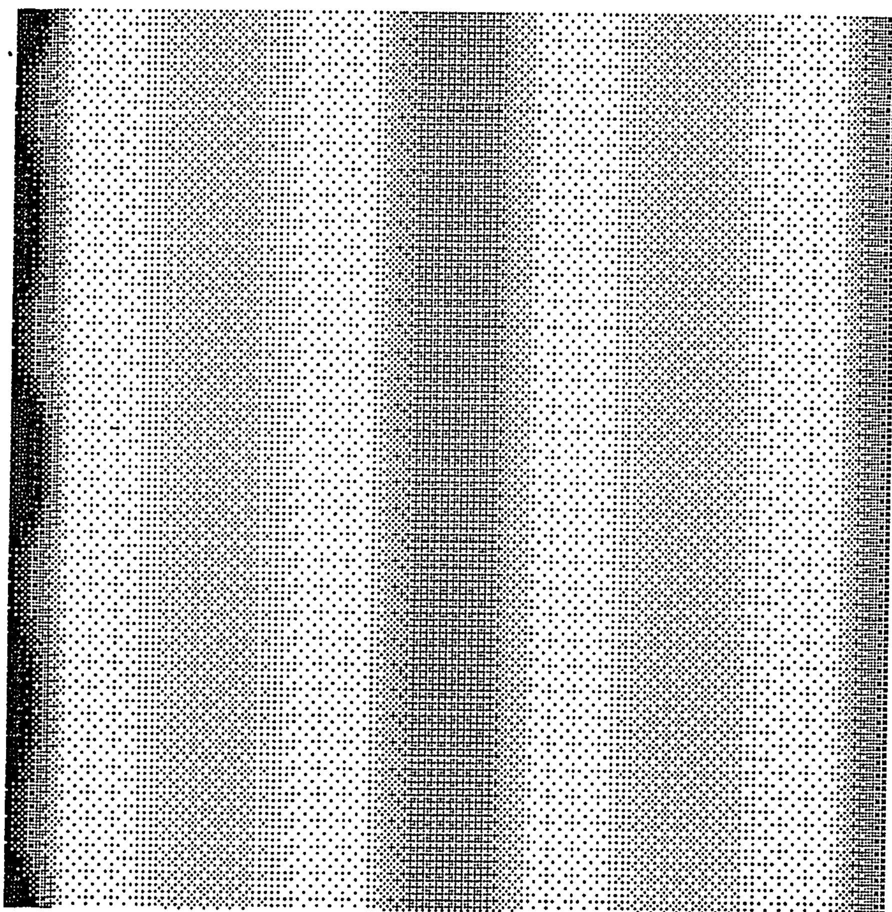


Figure 6.29. Picture of performance (dither plot): Collect, ss#15 at 16.8 msec., cumulative traffic (bytes)

<u>Shift communication program</u>

Shift is a collective communication routine in which processors transfer data around a ring of processors. For a one-dimensional shift right, each processor sends data to its right neighbor and receives data from its left neighbor. The basic operation of Shift is graphically depicted in Figure 6.30. For illustrative purposes, it is shown for the simplified case of an eight-node hypercube. In the simulation, performed on a 256-node hypercube, a 100-byte message was shifted by the processors.

Four images of program execution (generated by NCSA Image) are depicted in Figures 6.31 through 6.34. The displayed parameter in Figure 6.31, corresponding to snapshot number 4 (of 16) at time 0.7 (of 7) milliseconds, is cumulative time spent waiting by a processor. Processors that are highlighted in this image are in the receive phase of the shift algorithm and have spent time waiting, while the other processors are in the send phase of the algorithm. The images in the next three figures correspond to snapshot number 12 (of 16) at 4.2 (of 7) milliseconds. Figure 6.32 illustrates processor activity. Black denotes computational activity; gray, communication activity; and white, no activity. Recall that computational activity is recorded when processors have completed their portion of the shift and indicate the availability of the processors to do work. The displayed parameters in Figures 6.33 and 6.34 are, respectively, cumulative amount of time spent by the processor in communication activities (sends, receives, and waits); and cumulative amount of time spent waiting by a processor.

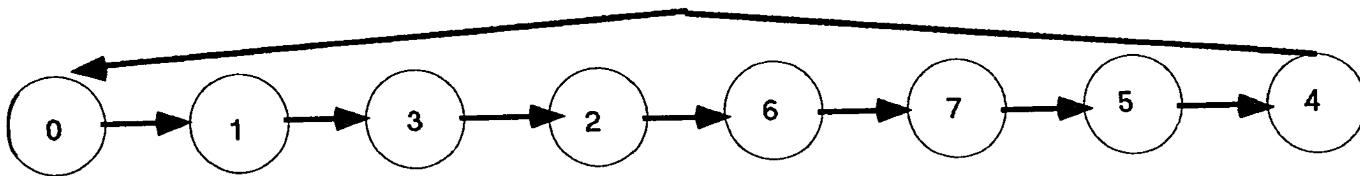


Figure 6.30. Basic operation of Shift on an eight-node hypercube

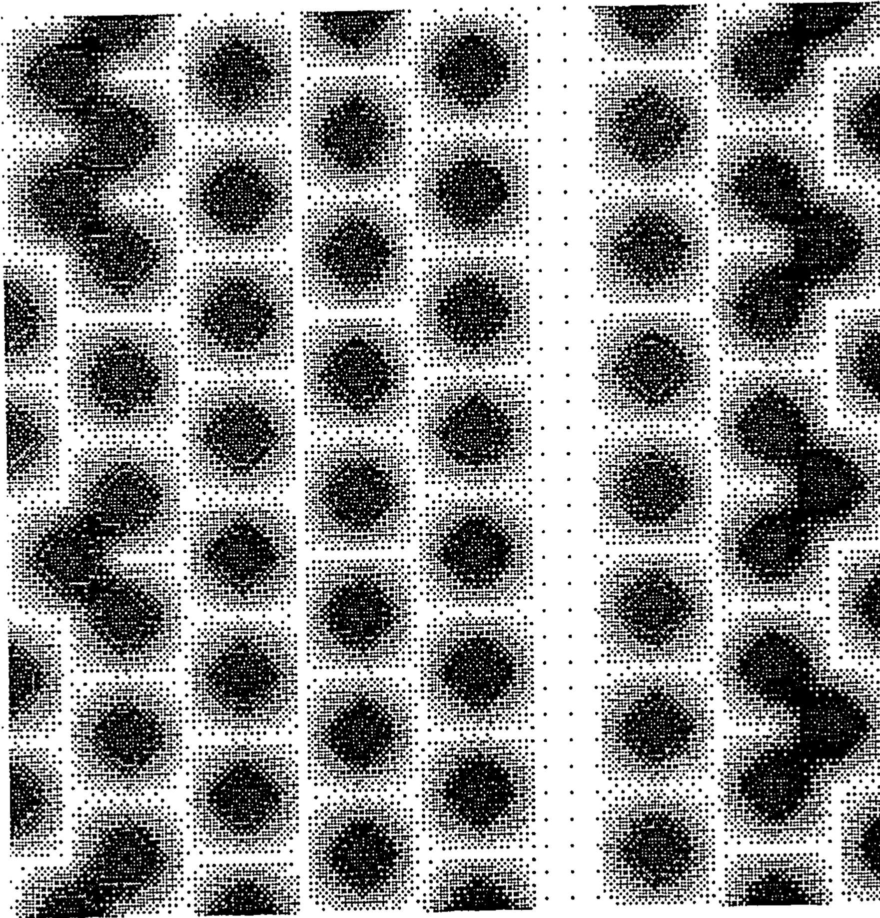


Figure 6.31. Picture of performance (dither plot): Shift, ss#4 at 0.7 msec., cumulative wait time

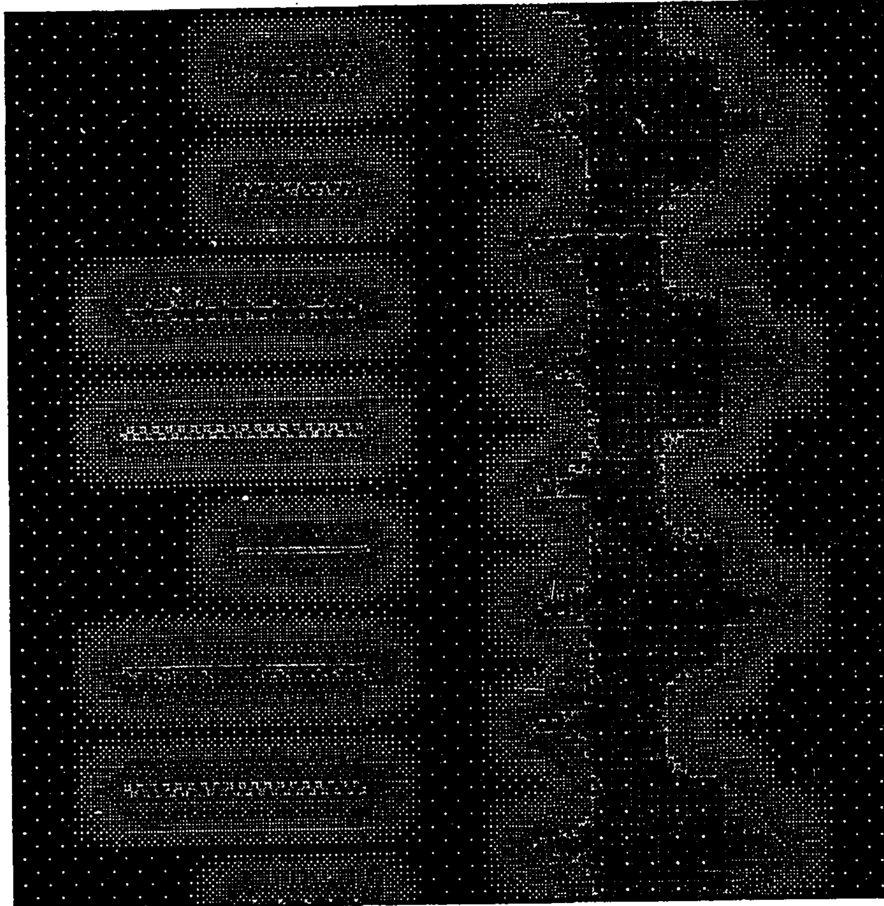


Figure 6.32. Picture of performance (dither plot): Shift, ss#12 at 4.2 msec., processor activity (black: computing; gray: communicating; white: none)

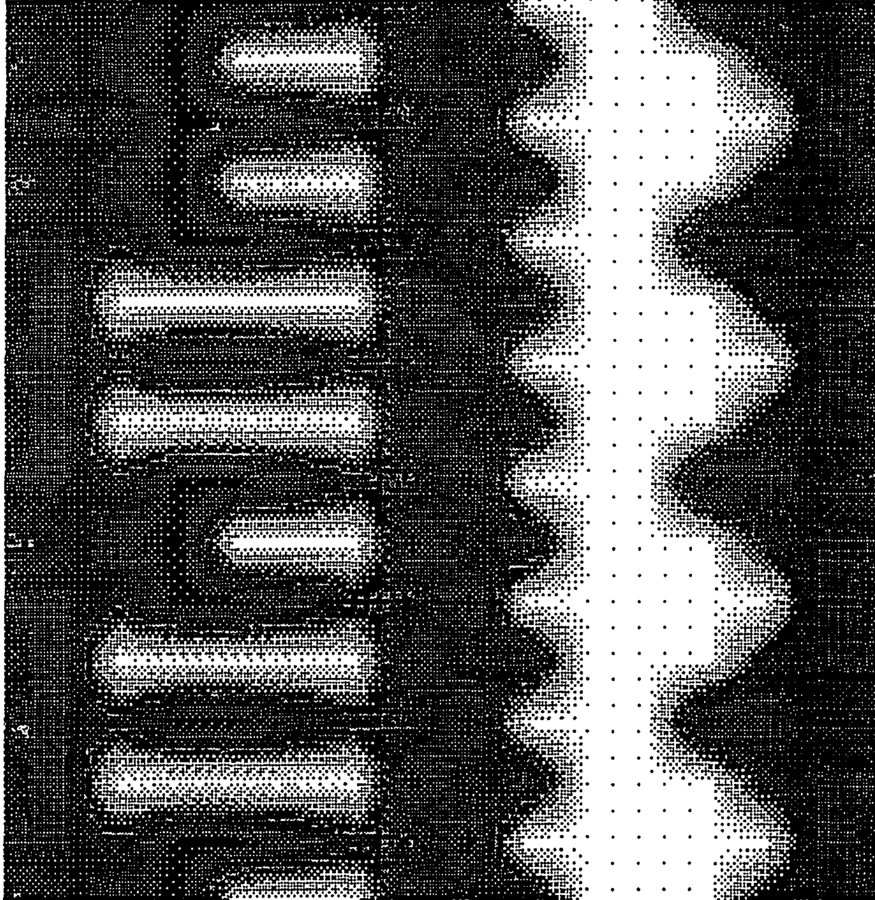


Figure 6.33. Picture of performance (dither plot): Shift, ss#12 at 4.2 msec., cumulative communication time

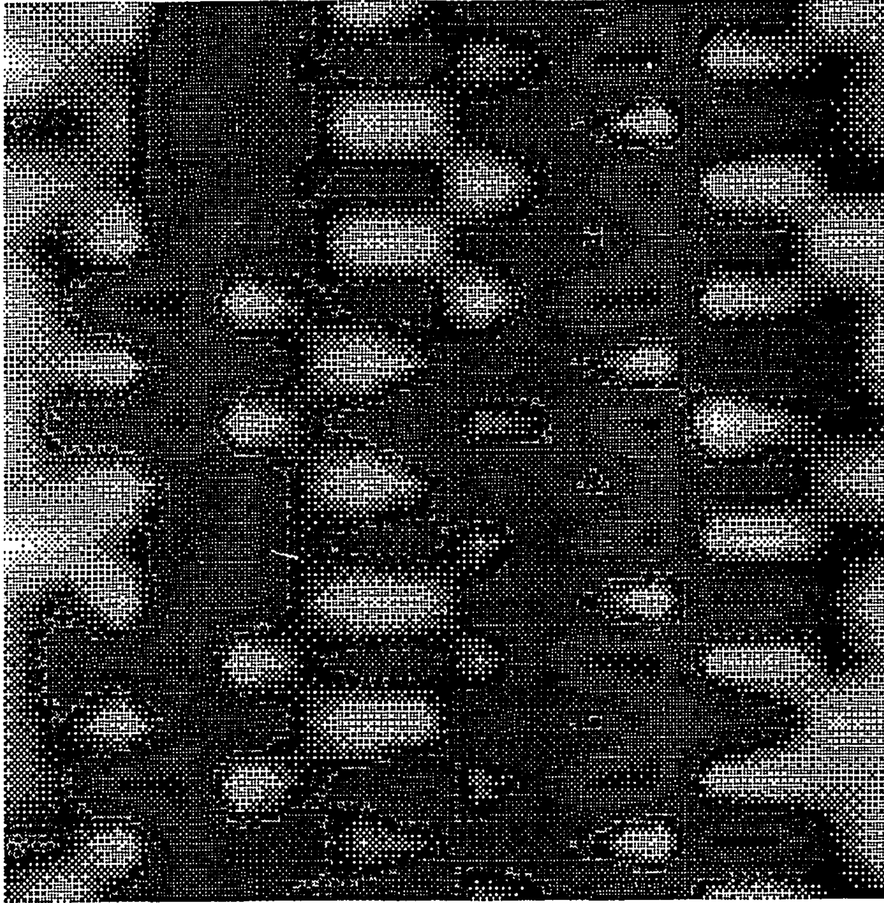


Figure 6.34. Picture of performance (dither plot): Shift, ss#12 at 4.2 msec., cumulative wait time

The collective communication programs spawn communication activities and generate message traffic in the system in some pattern. We may observe the flow or movement of communication quanta, as evidenced by images of processor activity, and we may also observe amounts of communication. These algorithms are typically used within larger application programs. Then, in addition to the traffic from the communication routines, there is work being done by the computational kernel of the application program. Thus, we may observe the flow or movement of computation quanta (if any occurs), and we may also observe amounts of work being done. This is illustrated via the following two case studies.

Divide-and-conquer quicksort program

Quicksort is an application program that uses a divide-and-conquer approach to sorting a list of numbers. One processor begins with the original list, divides the list in half, keeps half of the list, and sends half of the list to a neighboring processor. This continues recursively using a tree-like processor communication graph until all processors have a list. Each processor locally sorts its list. The basic operation of Quicksort is graphically depicted in Figure 6.35. For illustrative purposes, it is shown for the simplified case of an eight-node hypercube. Processor 0, the top node of the quicksort tree, initially has the original list.

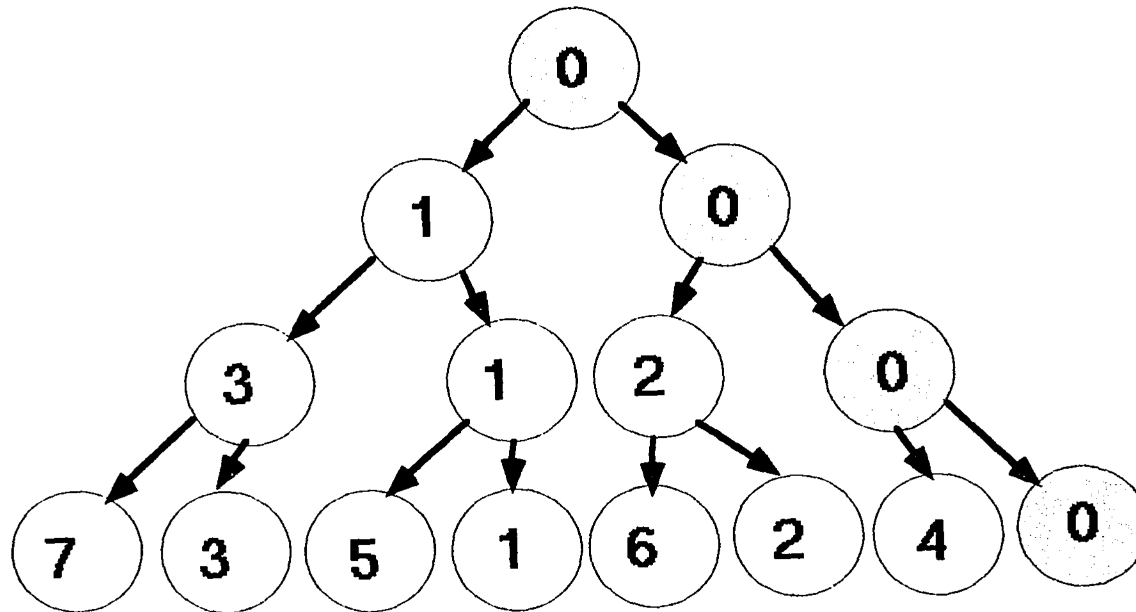


Figure 6.35. Basic operation of Quicksort on an eight-node hypercube

In the simulation, performed on a 256-node hypercube, a 4096-byte list was sorted on the processor ensemble. Table 6.1 categorizes the observations (event records) resulting from the simulation according to a distribution of ten uniform time intervals.

Selected global statistics for the system are documented in Tables 6.2, 6.12, and 6.13. Table 6.2 is a key for the other tables. Table 6.12 corresponds to snapshot number 10 (of 17) at time 0.0227 seconds. Table 6.13 corresponds to snapshot number 15 (of 17) at time 0.0279 seconds. Local statistics can be calculated for individual processors. Tables 6.5, 6.14, and 6.15 document selected local statistics for two processors at particular times. Table 6.5 is a key for the other tables. Processor 0 is detailed in Table 6.14, and Processor 100, in Table 6.15.

Images of program execution (generated by NCSA Image and MacSpin) are depicted in Figures 6.36 through 6.48. Figures 6.36 through 6.39 are images of program states at 0.0227 seconds (snapshot number 10). The displayed parameters are, respectively, cumulative amount of work, in number of operations, done by the processor at the indicated time; cumulative amount of time spent by the processor in computation activities (i.e., doing work); cumulative amount of time spent by the processor in communication activities (sends, receives, and waits); and processor activity. In Figure 6.39, black denotes computational activity; gray, communication activity; and white, no activity.

Figures 6.42 through 6.46 are images of program states at 0.0279 seconds (snapshot number 15). The displayed parameters are identical to those presented for snapshot number 10.

Execution time = 0.022 seconds

	Sum (Total)	Average	Maximum
A	23728	93	8176
B	147	1	9
C	368	1	64
D	0.0416	0.00016	0.0136
E	8.56 M	20.6 K	1.02 M
F	1.08 M	4.38 K	409 K
G	0.74%	0.77%	68%
H	10576	41	4080
I	85	0	8
J	2016	8	512
K	0.0658	0.00026	0.006
L	4.02 M	34.4 K	1.2 M
M	1.17%	1.19%	27.27%
Q	1.5817	0.0522	3.0
R	2.2436	0.2559	2.0039
S	25	0.098	1
T	0.61%	0.61%	6.25%

	NONE	COMPUTE	SND	WAIT	RCV
U	216	15	7	0	18
V	84.38	5.86	2.73	0.00	7.03 %

W	40	15.63%
X	25	0.61%
Y	0.0119	0.0113
Z	0.0428	0.0101

Table 6.12. Quicksort program. Selected global statistics for snapshot number 10 taken at 0.02273 seconds

Execution time = 0.027 seconds

	Sum (Total)	Average	Maximum
A	34592	135	8176
B	419	2	9
C	960	4	64
D	0.0726	0.00028	0.0136
E	20 M	42.2 K	2.03 M
F	1.28 M	5.86 K	409 K
G	1.05%	1.22%	68%
H	15576	61	4080
I	219	1	8
J	704	3	128
K	0.1728	0.00068	0.006
L	2.25 M	54.1 K	1.2 M
M	2.5%	2.82%	27.27%
Q	2.3802	0.1283	5.0
R	2.2209	0.7946	2.0039
S	25	0.098	2
T	0.61%	0.61%	12.5%

	NONE	COMPUTE	SND	WAIT	RCV
U	190	42	6	0	18
V	74.22	16.41	2.34	0.00	7.03 %

W	66	25.78%
X	25	0.61%
Y	0.0208	0.0165
Z	0.1125	0.0149

Table 6.13. Quicksort program. Selected global statistics for snapshot number 15 taken at 0.0279 seconds

A	B	C	D	E	F	G	H	I	J	K
3	0.0062	0.0000	2	4096	1	4096	0.0000	-	-	-
5	0.0124	0.0086	2	6144	2	2048	0.0082	7.49E+05	7.14E+05	95.35
6	0.0155	0.0150	1	7168	3	0	0.0126	5.69E+05	4.78E+05	84.00
7	0.0186	0.0180	2	8064	6	0	0.0136	5.93E+05	4.48E+05	75.56
17	0.0310	0.0200	0	8176	9	0	0.0136	6.01E+05	4.09E+05	68.00

L	M	N	O	P	Q	R	S	T	U	V	W	X
0	0	0	0.0000	-	-	0.0	-	-	-	-	0	0.0000
2048	1	0	0.0004	5.12E+06	4.65	0.0	0.0	0.0	0.0488	3.0	0	0.0000
3584	3	0	0.0004	8.96E+06	2.67	0.0	0.0	0.0	0.0317	2.0	0	0.0000
4032	6	64	0.0014	2.88E+06	7.78	0.0	0.0	0.0	0.1029	2.0	1	0.0625
4080	8	0	0.0034	1.20E+06	17.00	0.0	0.0	0.0	0.2500	2.0	1	0.0625

Table 6.14. Quicksort program. Selected local statistics for Processor 0, (x,y) = (0,0)

A	B	C	D	E	F	G	H	I	J	K
8	0.02015	0.000	4	0	0	0	0.000	-	-	-
9	0.02170	0.021	0	32	1	32	0.000	-	1.52E+03	0.00
17	0.03100	0.022	0	48	2	0	0.001	4.80E+04	2.18E+03	4.55

L	M	N	O	P	Q	R	S	T	U	V	W	X
0	0	0	0.0	-	-	0.0	-	-	-	-	0	0.0
32	1	0	0.0	-	0.0	0.0	0.0	-	-	1.0	0	0.0
48	2	0	0.0	-	0.0	0.0	0.0	-	0.0	1.0	0	0.0

Table 6.15. Quicksort program. Selected local statistics for Processor 100, (x,y) = (7,4)

Figure 6.43 shows a three-dimensional image that represents the same program state information as the image in Figure 6.42.

By comparing snapshot number 10 with snapshot number 15, observe how the work emanates from Processor 0 into the rest of the system. Most of the work is localized around Processor 0, as might be expected. In fact, because the length of the original list is small compared to the size of the system (only sixteen numbers remain to be sorted on each processor), we see that the bulk of the system has relatively little work to do. Viewing the snapshots of processor activity, we can observe the spread of activity in the system (as Processor 0 distributes work to its hypercube neighbors) and also the mix of computational and communication activities.

Figures 6.40, 6.41, 6.47, and 6.48 are dot plots of program activity generated by MacSpin. As in the images, a two-dimensional, 16x16 grid of processors is created. Visible dots denote active processors, and thus this format presents a visual display of concurrency and system utilization. Figures 6.40 and 6.41 correspond to snapshot number 10, and Figures 6.47 and 6.48, to snapshot number 15. Two different modes of observability are used. In Figures 6.40 and 6.47, all processors recording activity in the window of time spanning the initial time through the snapshot time are visible. Alternatively, in Figures 6.41 and 6.48, only those processors recording activity in the window of time spanning one percent below the snapshot time to one percent above the snapshot time are visible. In the former case, cumulative activity is presented; and in the latter case, (nearly) instantaneous activity is presented. Observe the similarities between

Figures 6.39 and 6.40 and between Figures 6.46 and 6.47.

Finally, another type of data plot can be generated by MacSpin that shows both temporal and spatial activity of the system via a single plot. An event space-time profile displays the distribution of events over time across all processors. Event profiles are depicted in Figures 6.49 through 6.53. Time is displayed on the horizontal axis, ranging from 0 seconds through the total execution time (here, 0.031 seconds). Processor addresses are displayed on the vertical axis, ranging from address 0 through address 255. A dot denotes the occurrence of an event at the indicated time on the indicated processor. In Figure 6.49, all types of events are shown as dots. In Figure 6.50, activity-related events are marked with "x". In Figures 6.51 through 6.53, computing, sending, and receiving events, respectively, are marked with "x". Note in Figure 6.52 how easily we can observe that only half of the processors perform send operations in Quicksort.

One-dimensional wave equation program

1-D Wave is an application program that uses a domain decomposition approach to solving the wave equation in one dimension using a finite difference method [Fox et al., 1988]. The problem domain (here, linear) is divided equally among all processors; that is, each processor is allocated the same number of points in the discretized domain. All processors are broadcast an initial set of data and then iteratively converge to a solution. Each iteration consists of a communication step followed by a computation step. The communication step involves an exchange of endpoint data

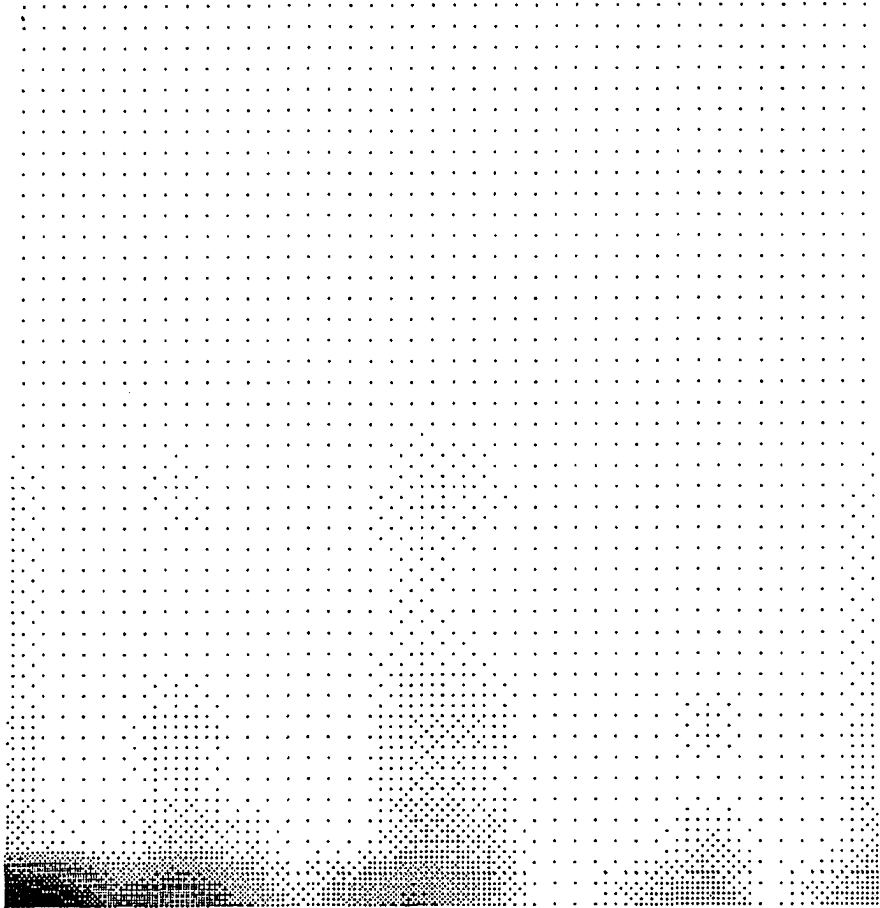


Figure 6.36. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., cumulative work (operations)

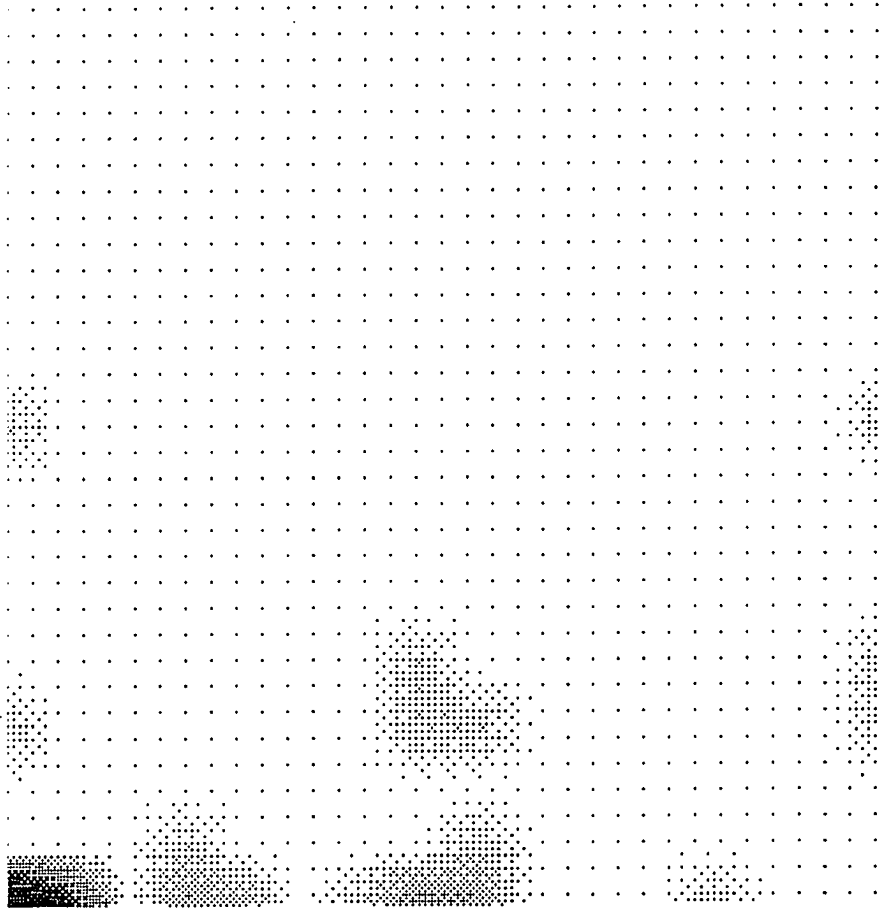


Figure 6.37. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., cumulative computation time

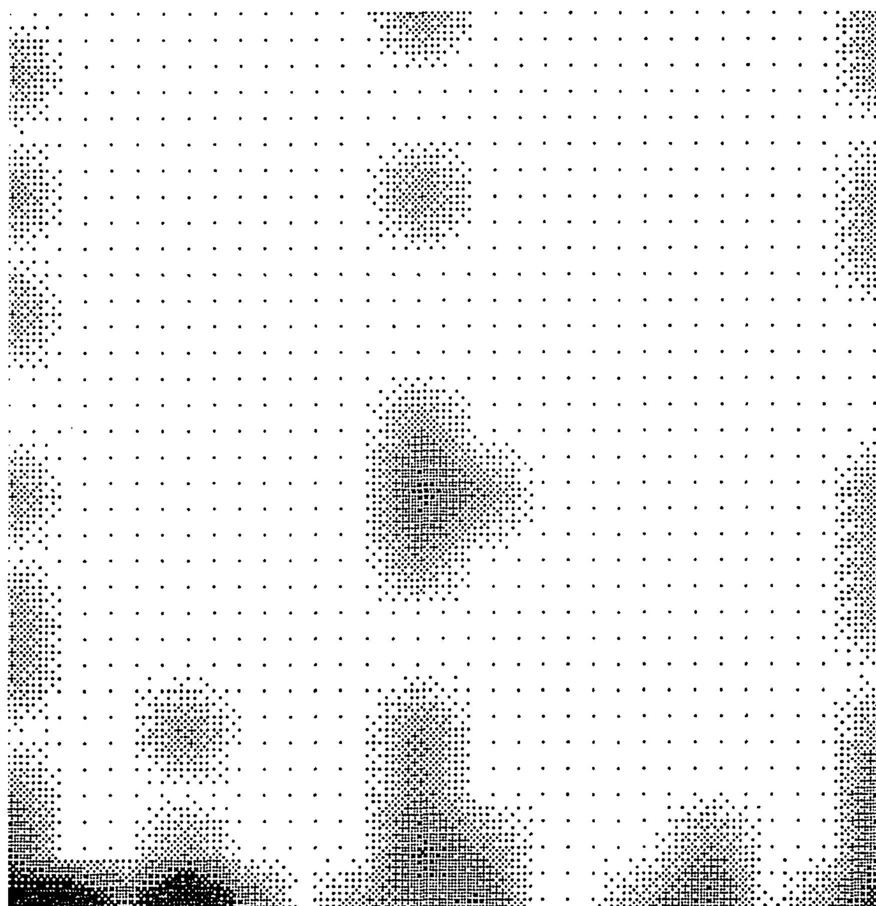


Figure 6.38. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., cumulative communication time

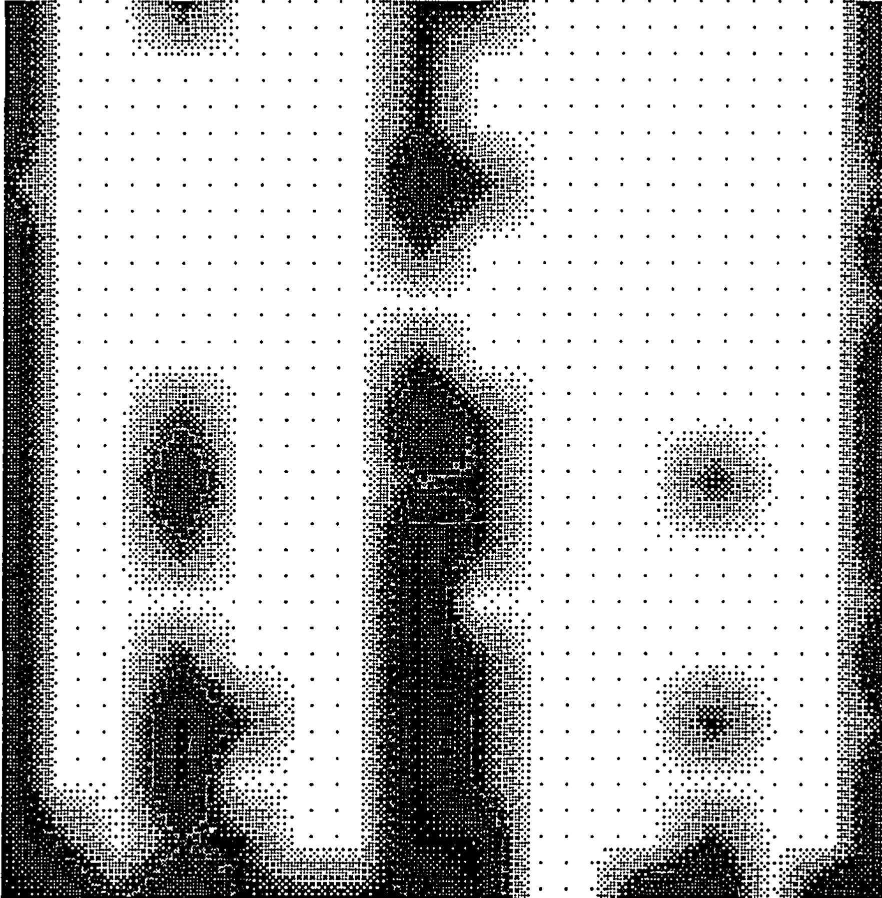


Figure 6.39. Picture of performance (dither plot): Quicksort, ss#10 at 22.7 msec., processor activity (black: computing; gray: communicating; white: none)

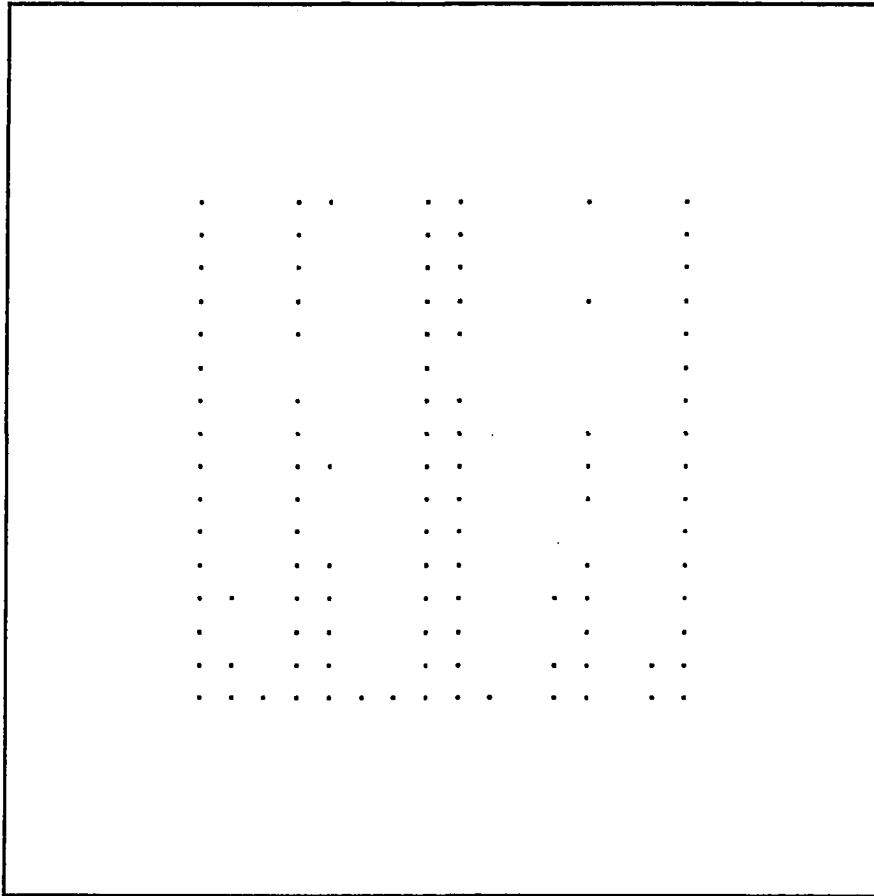


Figure 6.40. Picture of performance (dot plot): Quicksort, at 23 msec., cumulative activity

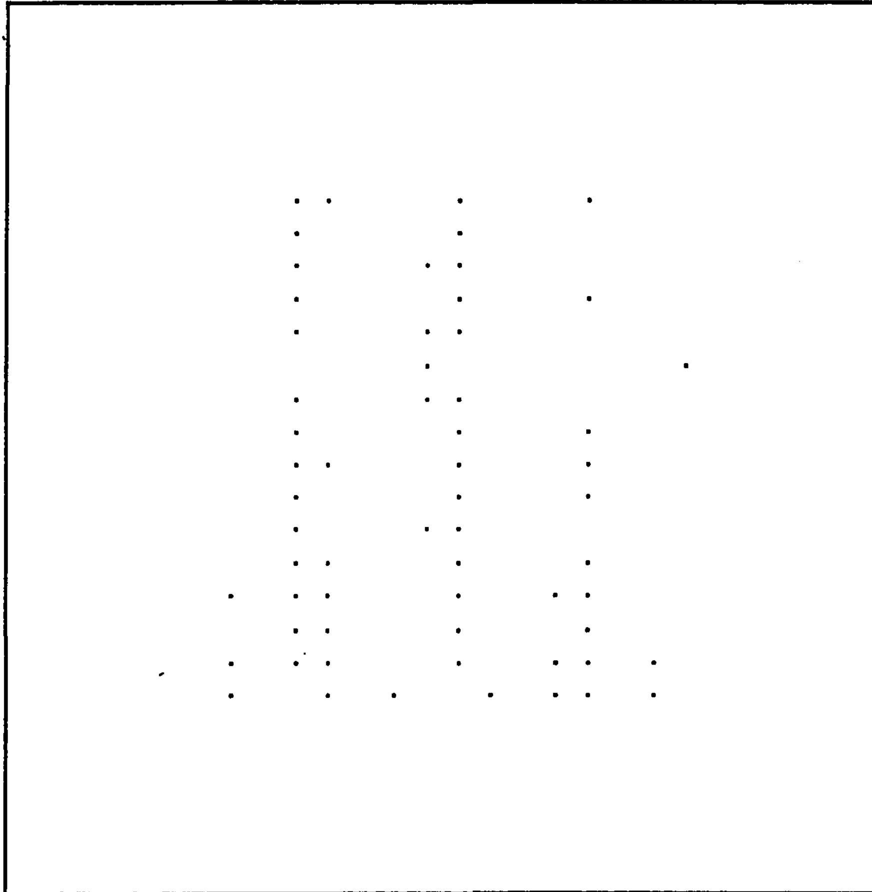


Figure 6.41. Picture of performance (dot plot): Quicksort, at 23 msec., instantaneous activity

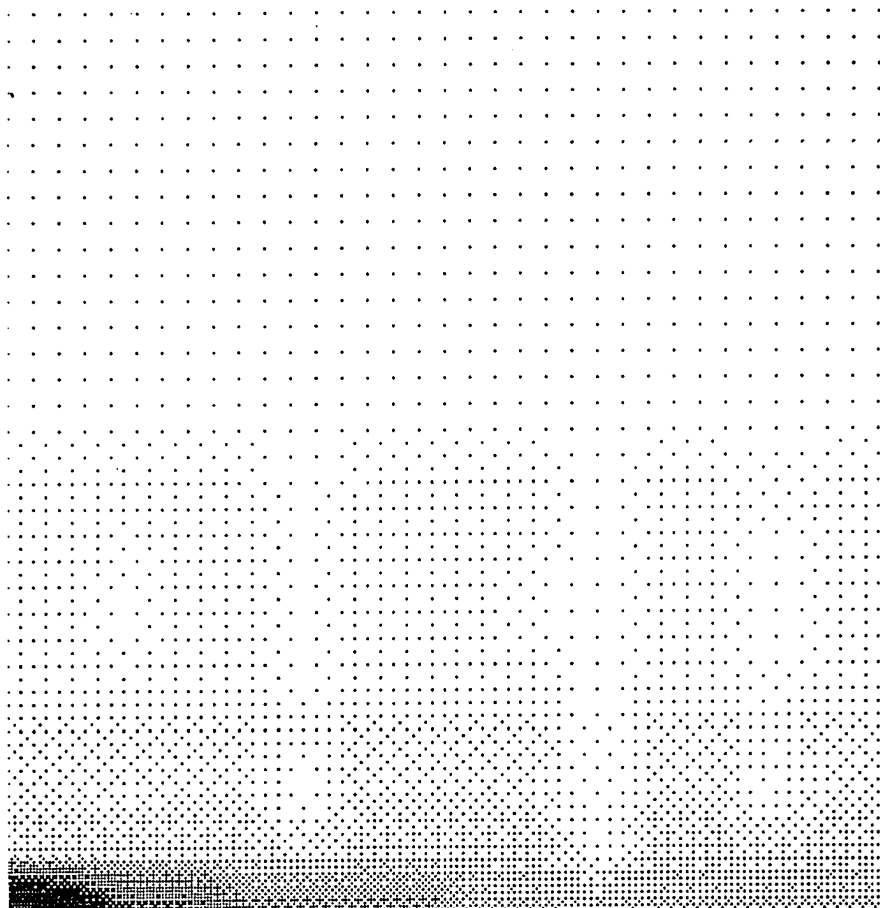


Figure 6.42. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., cumulative work (operations)

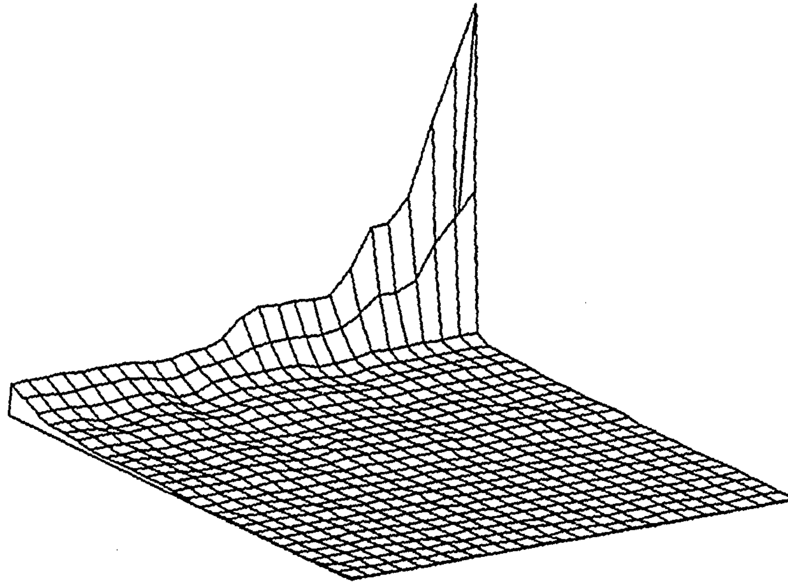


Figure 6.43. Picture of performance (3D plot): Quicksort, ss#15 at 27.9 msec., cumulative work (operations)

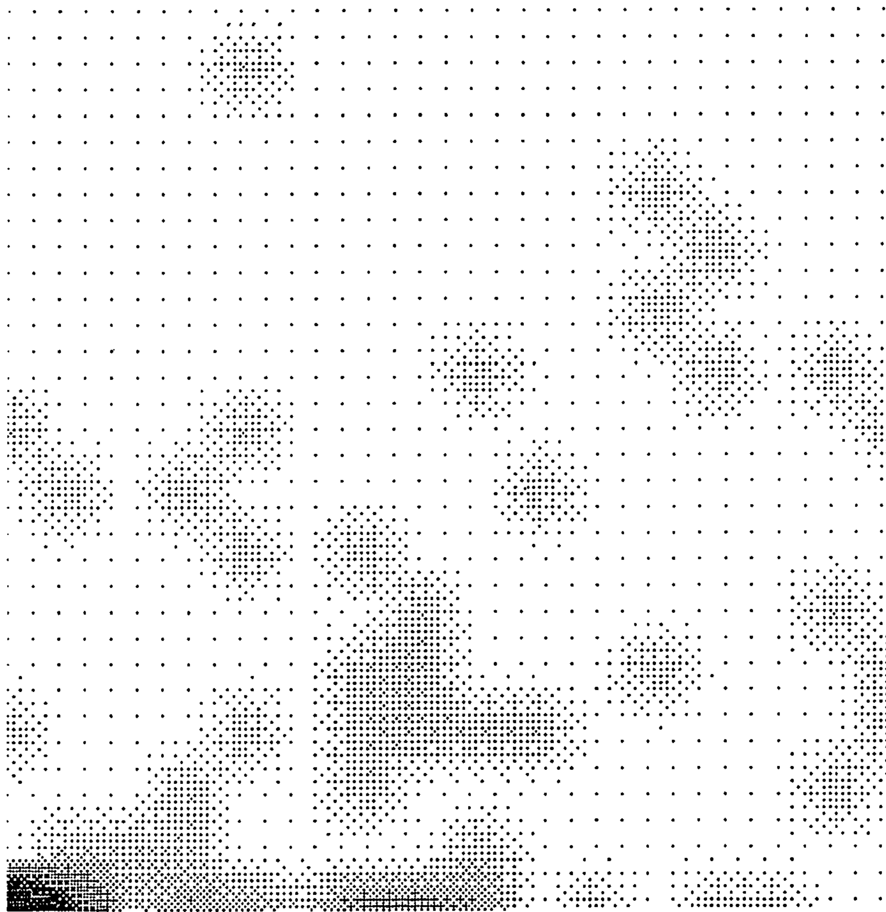


Figure 6.44. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., cumulative computation time

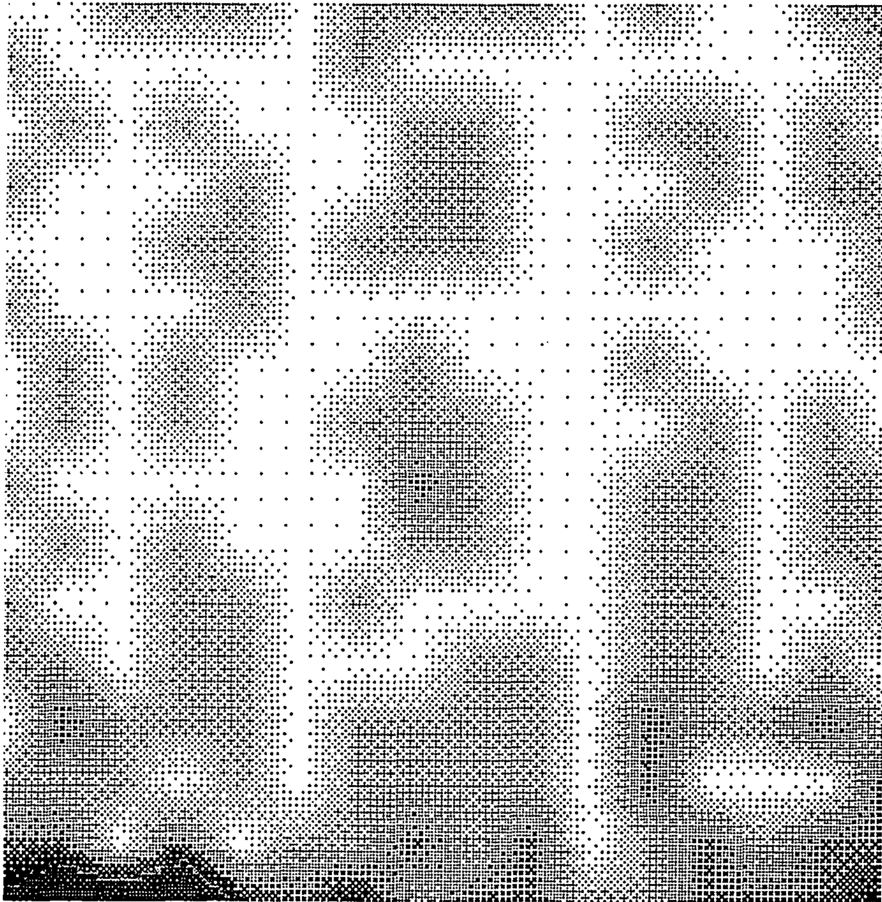


Figure 6.45. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., cumulative communication time

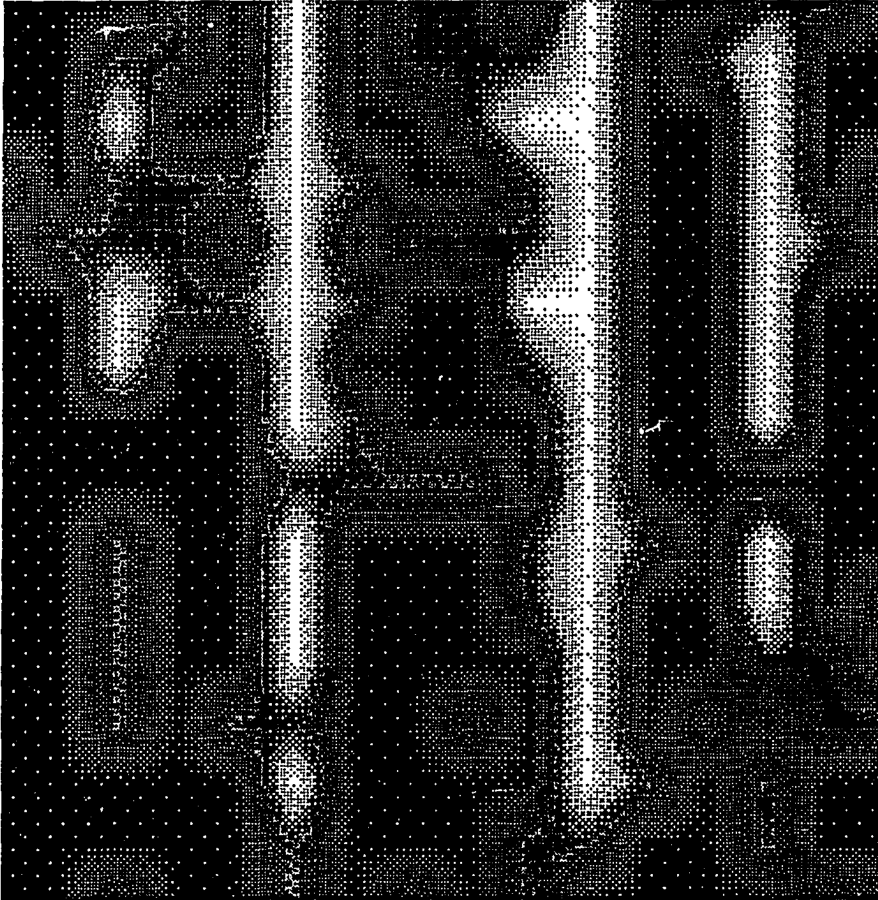


Figure 6.46. Picture of performance (dither plot): Quicksort, ss#15 at 27.9 msec., processor activity (black: computing; gray: communicating; white: none)

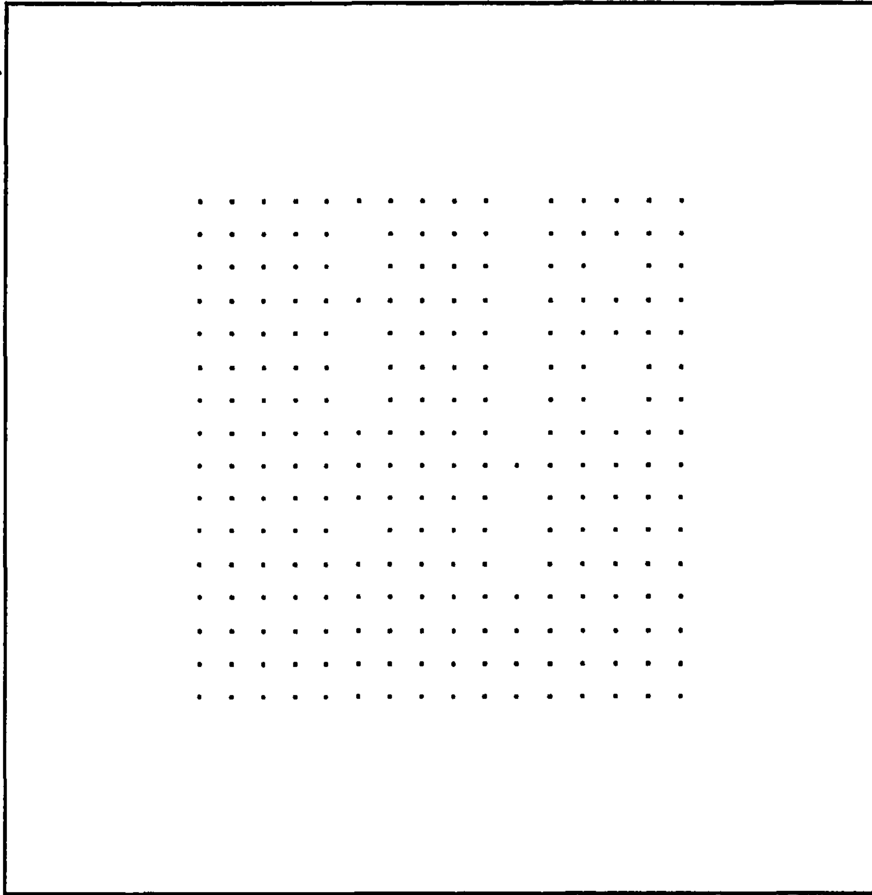


Figure 6.47. Picture of performance (dot plot): Quicksort, at 28 msec., cumulative activity

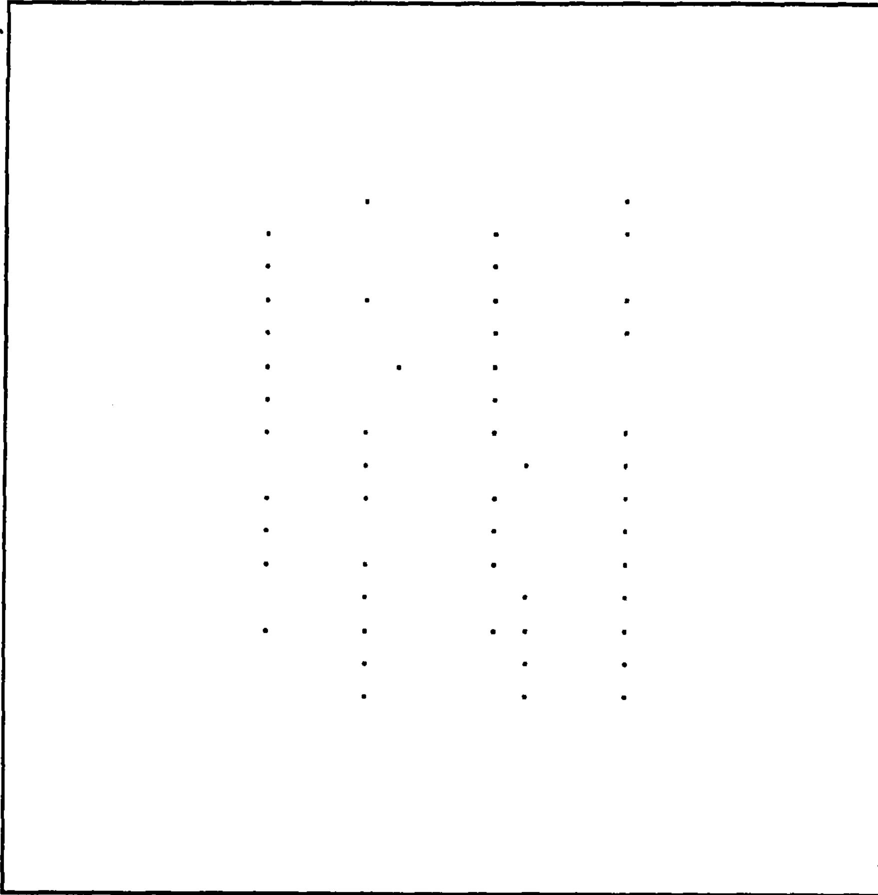


Figure 6.48. Picture of performance (dot plot): Quicksort, at 28 msec., instantaneous activity

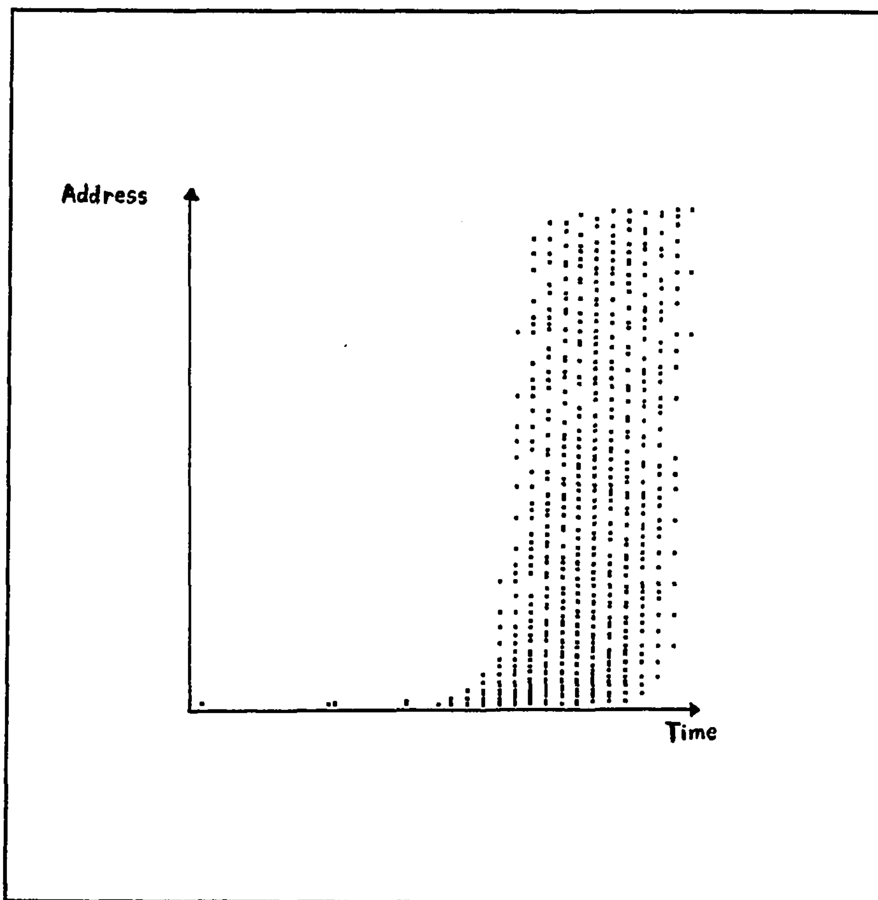


Figure 6.49. Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, • : event

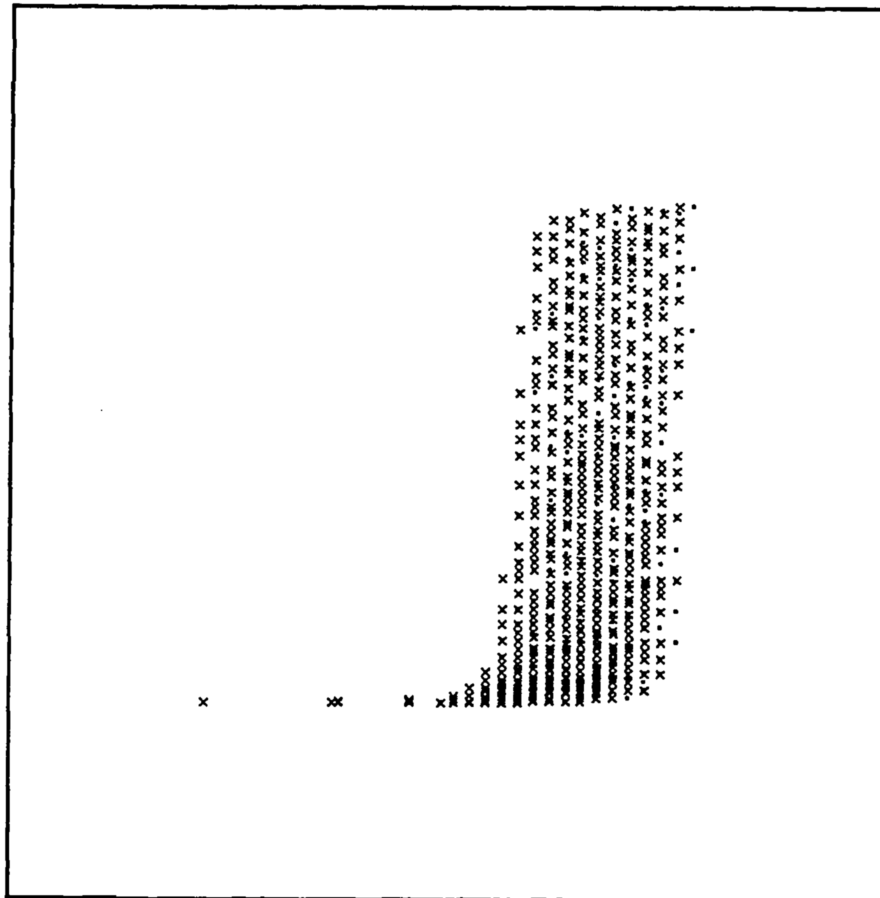


Figure 6.50. Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : activity event

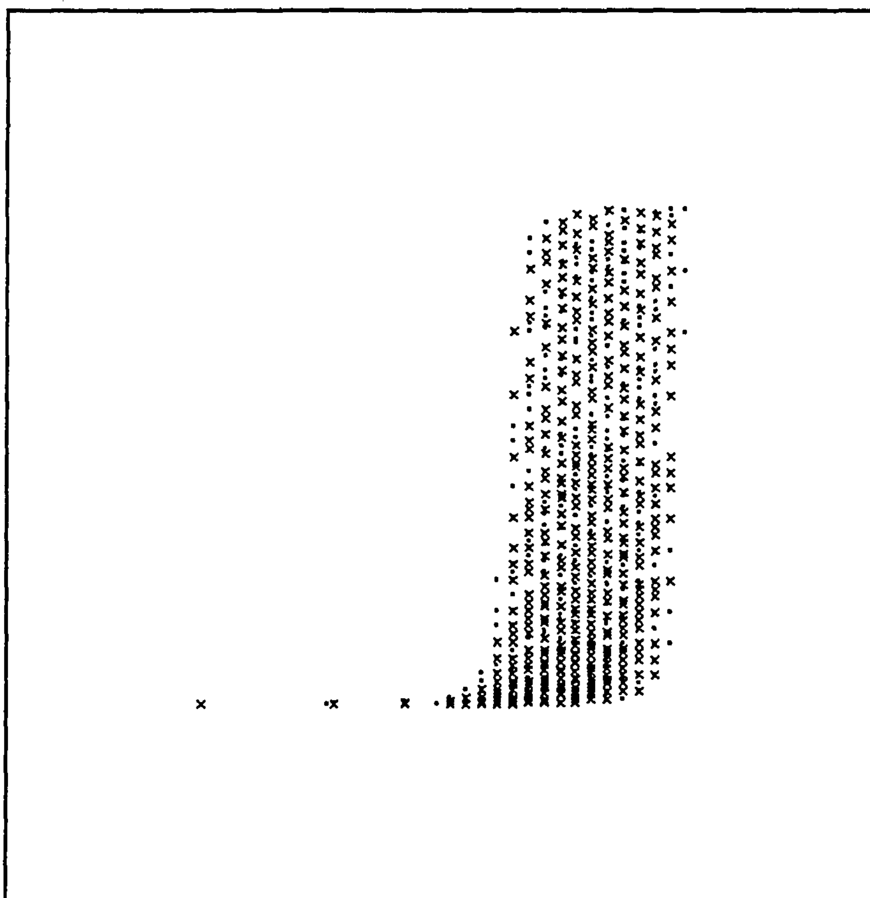


Figure 6.51. Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : compute event

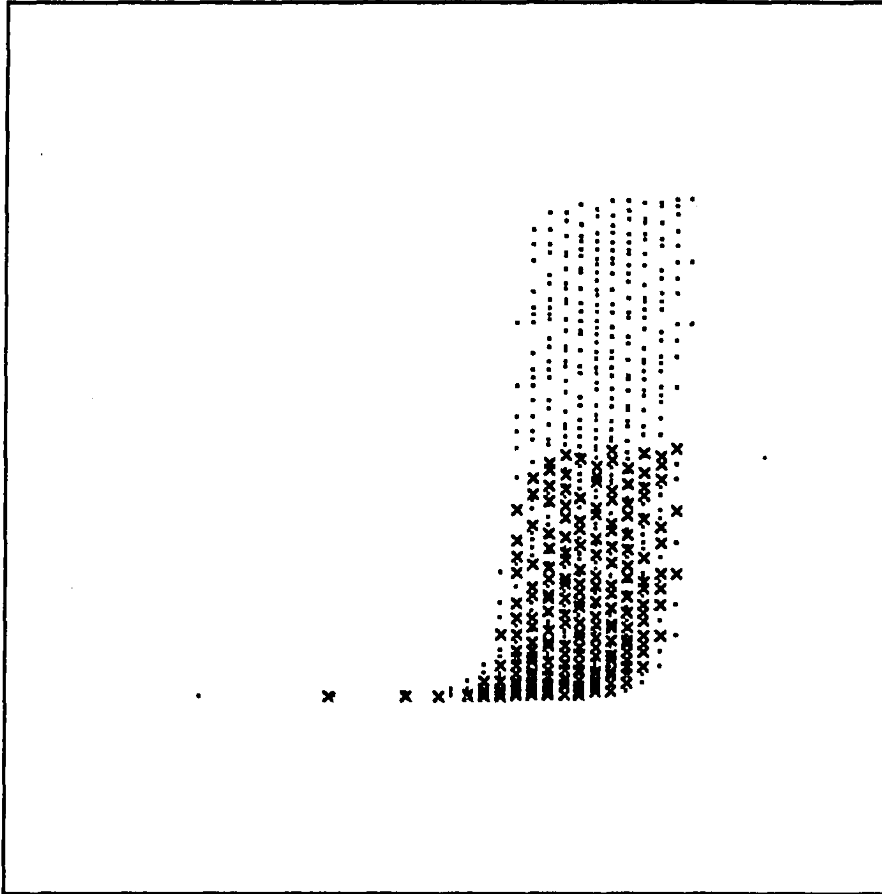


Figure 6.52. Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : send event

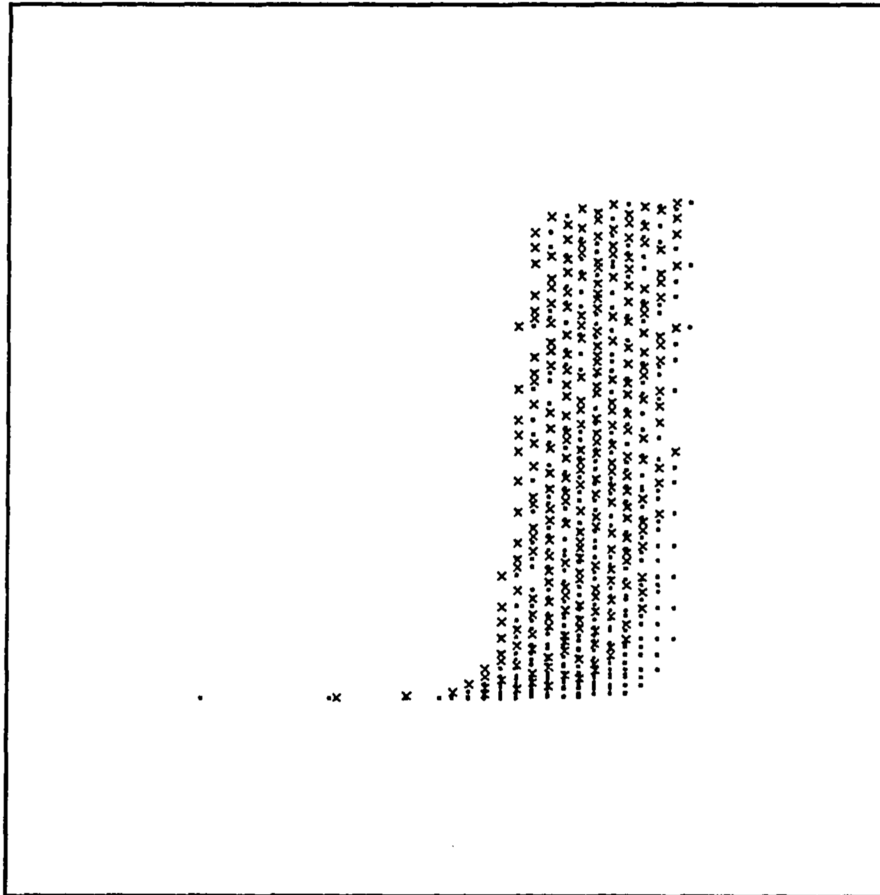


Figure 6.53. Event space-time profile: Quicksort. Time: 0 - 31 msec., Addresses: 0 - 255, x : receive event

values between processors via shift left and shift right collective communication routines. The final results from all processors are collected and sent to the host.

In the simulation, performed on a 256-node hypercube, each processor was assigned 100 points of the problem domain. Table 6.1 categorizes the observations (event records) resulting from the simulation according to a distribution of ten uniform time intervals.

Selected global statistics for the system are documented in Tables 6.2 and 6.16 through 6.19. Table 6.2 is a key for the other tables. Table 6.16 corresponds to snapshot number 4 (of 37) at time 0.01 seconds; Table 6.17, snapshot number 12 at time 0.03 seconds; Table 6.18, snapshot number 23 at time 0.052 seconds; and Table 6.19, snapshot number 36 at 0.09 seconds. Local statistics can be calculated for individual processors. Tables 6.5, 6.20, and 6.21 document selected local statistics for two processors at particular times. Table 6.5 is a key for the other tables. Processor 0 is detailed in Table 6.20, and Processor 100, in Table 6.21.

Images of program execution (generated by NCSA Image) are depicted in Figures 6.54 through 6.63. Figures 6.54 and 6.55 correspond to snapshot number 4 at 0.01 seconds. The displayed parameters are, respectively, processor activity and cumulative amount of time spent by the processor in communication activities (sends, receives, and waits). By comparing these images with those for Broadcast, we can observe that broadcasting activities are dominating program execution early in the simulation.

Execution time = 0.01 seconds

	Sum (Total)	Average	Maximum
A	65000	254	1000
B	65	0	1
C	63000	246	1000
F	6.5 M	33.8 K	357 K
H	40400	158	3200
I	101	0	8
J	14000	55	400
K	0.5264	0.00206	0.01
L	4.07 M	31.8 K	727 K
M	20.56%	25.39%	100.00%
N	-	0.00286	0.01
O	28.64%	32.08%	100.00%
P	-	0.1959	0.92
R	1.6089	0.4096	2.5
S	53	0.21	6
T	1.29%	1.29%	37.5%

	NONE	COMPUTE	SND	WAIT	RCV
U	0	63	10	158	25
V	0.00	24.61	3.91	61.72	9.77 %

W	256	100.00%
X	53	1.29%
Y	-	0.25
Z	0.2055	0.0493

Table 6.16. 1-D Wave program. Selected global statistics for snapshot number 4 taken at 0.01 seconds

Execution time = 0.03 seconds

	Sum (Total)	Average	Maximum
A	434000	1695	3000
B	345	1	2
C	211000	824	2000
D	2.1616	0.00844	0.0154
E	28.5 M	80.1 K	205 K
F	14.5 M	67.2 K	130 K
G	28.15%	29.76%	66.96%
H	103048	403	3216
I	517	2	12
J	6396	25	400
K	3.9114	0.01528	0.03
L	6.01 M	56.6 K	309 K
M	50.93%	65.69%	100.00%
N	3.2512	0.0127	0.024
O	42.33%	55.52%	96%
P	0.8312	0.8313	-
Q	1.8095	0.4894	1.0
R	4.2116	2.8185	7.2115
S	230	0.89	5
T	5.566%	5.566%	31.25%

	NONE	COMPUTE	SND	WAIT	RCV	
U	0	142	50	33	31	
V	0.00	55.47	19.53	12.89	12.11	%

W	256	100.00%
X	228	5.57%
Y	0.5483	0.5651
Z	0.5093	0.1252

Table 6.17. 1-D Wave program. Selected global statistics for snapshot number 12 taken at 0.03 seconds

Execution time = 0.052 seconds

	Sum (Total)	Average	Maximum
A	1251000	4887	6000
B	673	3	3
C	353000	1379	3000
D	5.6266	0.02198	0.0304
E	27.3 M	216 K	411 K
F	24.1 M	101 K	146 K
G	42.27%	45.4%	72.2%
H	105642	413	3232
I	1166	5	16
J	520	2	4
K	6.338	0.02476	0.052
L	4.22 M	36.3 K	182 K
M	47.61%	51.12%	100%
N	3.838	0.01499	0.033
O	28.83%	31.03%	63.46%
P	0.6055	0.6482	-
Q	1.1265	1.138	2.4667
R	11.8419	7.9323	13.8889
S	250	0.99	5
T	6.18%	6.18%	31.25%

	NONE	COMPUTE	SND	WAIT	RCV	
U	0	123	52	32	49	
V	0.00	48.05	20.31	12.50	19.14	%

W	256	100.00%
X	253	6.18%
Y	0.7230	0.8145
Z	0.4761	0.1277

Table 6.18. 1-D Wave program. Selected global statistics for snapshot number 23 taken at 0.052 seconds

Execution time = 0.09 seconds

	Sum (Total)	Average	Maximum
A	1536000	6000	6000
B	768	3	3
C	0	0	0
D	8.2966	0.03241	0.0454
E	-	209 K	411 K
F	17.1 M	87 K	103 K
G	36.01%	47.35%	78.28%
H	302896	1183	14832
I	1771	7	45
J	102400	400	400
K	7.2294	0.02824	0.057
L	45.6 M	95.7 K	899 K
M	31.38%	40.46%	78.08%
N	4.3962	0.01717	0.053
O	19.08%	24.41%	72.6%
P	0.6081	0.6535	-
Q	0.8714	1.0778	3.8
R	5.071	4.6558	7.2115
S	1100	4.3	-
T	26.56%	26.56%	-

	NONE	COMPUTE	SND	WAIT	RCV	
U	0	0	253	2	1	
V	0.00	0.00	98.83	0.78	0.39	%

W	256	100.00%
X	1088	26.56%
Y	0.7138	1.0
Z	0.4954	0.0798

Table 6.19. 1-D Wave program. Selected global statistics for snapshot number 36 taken at 0.09 seconds

A	B	C	D	E	F	G	H	I	J	K
1	0.00000	0.0000	2	0	0	0	0.0000	-	-	-
2	0.00333	0.0032	2	0	0	0	0.0000	-	0.00E+00	0.00
6	0.01667	0.0044	2	1000	1	1000	0.0000	-	2.27E+05	0.00
8	0.02200	0.0200	2	1000	1	0	0.0146	6.85E+04	5.00E+04	73.00
9	0.02400	0.0240	1	1000	1	0	0.0146	6.85E+04	4.17E+04	60.83
16	0.03800	0.0250	2	3000	2	0	0.0146	2.05E+05	1.20E+05	58.40
17	0.04000	0.0400	2	3000	2	0	0.0146	2.05E+05	7.50E+04	36.50
25	0.05600	0.0420	3	6000	3	0	0.0146	4.11E+05	1.43E+05	34.76
26	0.05800	0.0570	4	6000	3	0	0.0146	4.11E+05	1.05E+05	25.61
27	0.06000	0.0590	4	6000	3	0	0.0146	4.11E+05	1.02E+05	24.75
28	0.06250	0.0620	3	6000	3	0	0.0146	4.11E+05	9.68E+04	23.55
29	0.06500	0.0650	4	6000	3	0	0.0146	4.11E+05	9.23E+04	22.46
30	0.06750	0.0670	3	6000	3	0	0.0146	4.11E+05	8.96E+04	21.79
31	0.07000	0.0700	3	6000	3	0	0.0146	4.11E+05	8.57E+04	20.86
32	0.07250	0.0710	4	6000	3	0	0.0146	4.11E+05	8.45E+04	20.56
33	0.07500	0.0750	3	6000	3	0	0.0146	4.11E+05	8.00E+04	19.47
34	0.07750	0.0770	4	6000	3	0	0.0146	4.11E+05	7.79E+04	18.96
35	0.08000	0.0800	3	6000	3	0	0.0146	4.11E+05	7.50E+04	18.25
36	0.09000	0.0890	3	6000	3	0	0.0146	4.11E+05	6.74E+04	16.40
37	0.10000	0.0990	3	6000	3	0	0.0146	4.11E+05	6.06E+04	14.75

L	M	N	O	P	Q	R	S	T	U	V
0	0	0	0.0000	-	-	0.000	-	-	-	-
2400	6	400	0.0000	-	0.00	0.000	0.00	-	-	0.0000
3200	8	0	0.0044	7.27E+05	100.00	0.000	0.00	0.0000	-	0.3125
3204	9	4	0.0044	7.28E+05	22.00	0.000	0.00	0.0000	0.3014	0.3121
3212	11	4	0.0044	7.30E+05	18.33	0.004	16.67	0.9091	0.3014	0.3113
3216	12	4	0.0104	3.09E+05	41.60	0.004	16.00	0.3846	0.7123	0.9328
3220	13	4	0.0104	3.10E+05	26.00	0.004	10.00	0.3846	0.7123	0.9317
3232	16	4	0.0274	1.18E+05	65.24	0.005	11.90	0.1825	1.8767	1.8564
3232	16	4	0.0274	1.18E+05	48.07	0.005	8.77	0.1825	1.8767	1.8564
3632	17	400	0.0274	1.33E+05	46.44	0.022	37.29	0.8029	1.8767	1.6520
4432	19	400	0.0274	1.62E+05	44.19	0.023	37.10	0.8394	1.8767	1.3538
5632	22	400	0.0274	2.06E+05	42.15	0.025	38.46	0.9124	1.8767	1.0653
6032	23	400	0.0274	2.20E+05	40.90	0.027	40.30	0.9854	1.8767	0.9947
7232	26	400	0.0274	2.64E+05	39.14	0.029	41.43	1.0584	1.8767	0.8296
7632	27	400	0.0274	2.79E+05	38.59	0.030	42.25	1.0949	1.8767	0.7862
8832	30	400	0.0274	3.22E+05	36.53	0.031	41.33	1.1314	1.8767	0.6793
9632	32	400	0.0274	3.52E+05	35.58	0.031	40.26	1.1314	1.8767	0.6229
10432	34	400	0.0274	3.81E+05	34.25	0.034	42.50	1.2409	1.8767	0.5752
13632	42	400	0.0274	4.98E+05	30.79	0.038	42.70	1.3869	1.8767	0.4401
16832	50	400	0.0274	6.14E+05	27.68	0.044	44.44	1.6058	1.8767	0.3565

Table 6.20. 1-D Wave program. Selected local statistics for Processor 0, (x,y) = (0,0)

A	B	C	D	E	F	G	H	I	J	K	L
3	0.00667	0.000	4	0	0	0	0.000	-	-	-	0
4	0.01000	0.010	2	0	0	0	0.000	-	0.00E+00	0.00	400
10	0.02600	0.012	3	1000	1	0	0.000	-	8.33E+04	0.00	1200
11	0.02800	0.028	2	1000	1	0	0.000	-	3.57E+04	0.00	1208
12	0.03000	0.029	1	1000	1	0	0.000	-	3.45E+04	0.00	1212
19	0.04400	0.031	2	3000	2	0	0.000	-	9.68E+04	0.00	1216
20	0.04600	0.046	4	3000	2	0	0.000	-	6.52E+04	0.00	1220
21	0.04800	0.048	1	3000	2	0	0.000	-	6.25E+04	0.00	1232
28	0.06250	0.049	2	6000	3	3000	0.000	-	1.22E+05	0.00	1232
29	0.06500	0.064	4	6000	3	0	0.015	4.00E+05	9.37E+04	23.44	1632
30	0.06750	0.067	4	6000	3	0	0.015	4.00E+05	8.96E+04	22.39	2032
37	0.10000	0.070	2	6000	3	0	0.015	4.00E+05	8.57E+04	21.43	3632

M	N	O	P	Q	R	S	T	U	V
0	0	0.000	-	-	0.000	-	-	-	-
1	400	0.000	-	0.00	0.010	100.00	-	-	0.0000
3	400	0.012	1.00E+05	100.00	0.010	83.33	0.8333	-	0.8333
5	4	0.012	1.01E+05	42.86	0.010	35.71	0.8333	-	0.8278
6	4	0.012	1.01E+05	41.38	0.010	34.48	0.8333	-	0.8251
7	4	0.031	3.92E+04	100.00	0.012	38.71	0.3871	-	2.4671
8	4	0.031	3.94E+04	67.39	0.012	26.09	0.3871	-	2.4590
11	4	0.031	3.97E+04	64.58	0.013	27.08	0.4194	-	2.4351
11	0	0.049	2.51E+04	100.00	0.014	28.57	0.2857	-	4.8701
12	400	0.049	3.33E+04	76.56	0.014	21.88	0.2857	3.2667	3.6765
13	400	0.049	4.15E+04	73.13	0.016	23.88	0.3265	3.2667	2.9528
17	400	0.049	7.41E+04	70.00	0.018	25.71	0.3673	3.2667	1.6520

Table 6.21. 1-D Wave program. Selected local statistics for Processor 100, (x,y) = (7,4)

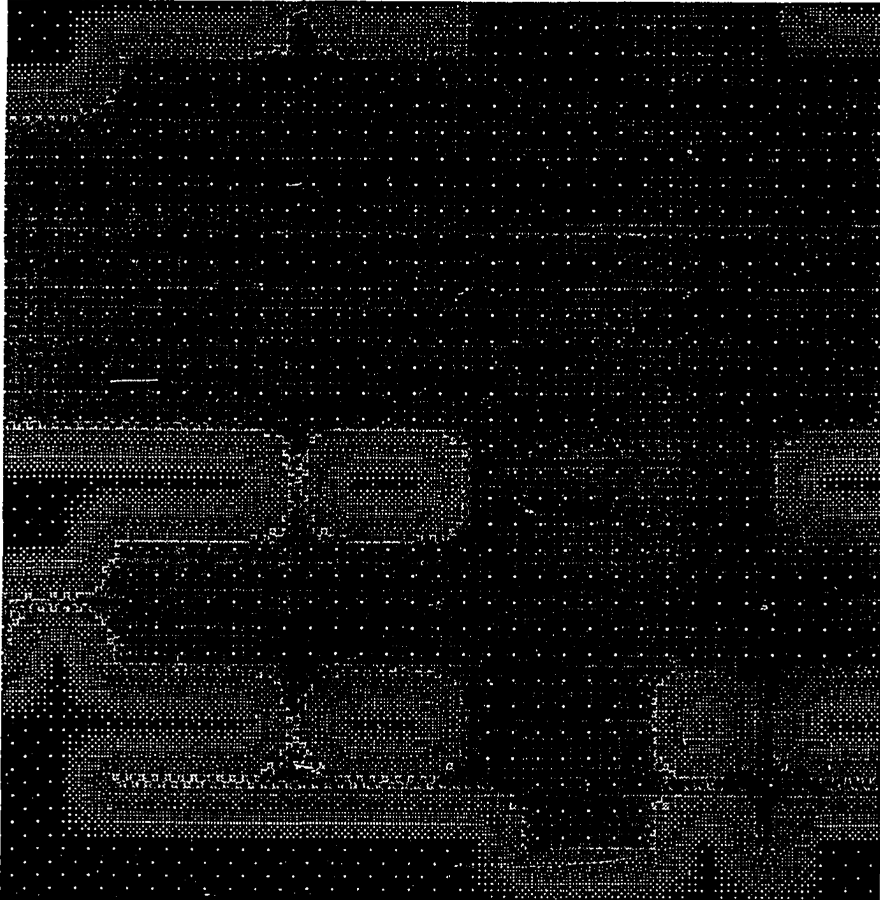


Figure 6.54. Picture of performance (dither plot): 1D Wave, ss#4 at 10 msec., processor activity (black: computing; gray: communicating; white: none)

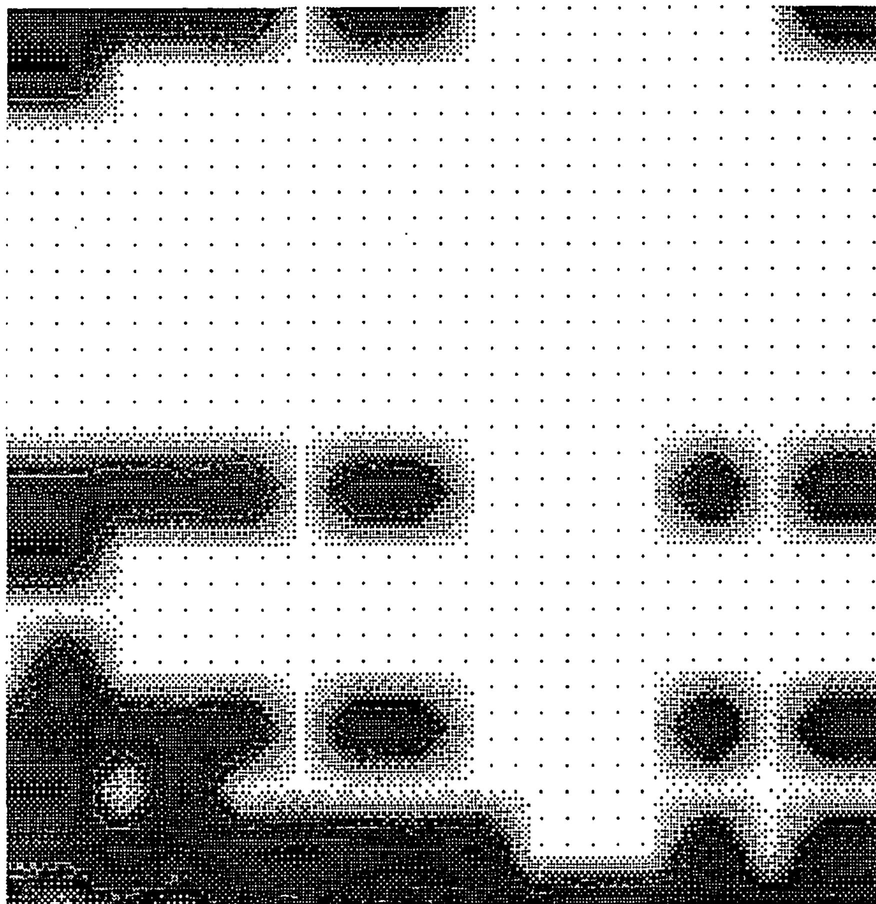


Figure 6.55. Picture of performance (dither plot): 1D Wave, ss#4 at 10 msec., cumulative communication time

Figures 6.56 through 6.58 correspond to states of the system at 0.03 seconds (snapshot number 12). The displayed parameters are, respectively, processor activity, cumulative amount of time spent by the processor in computation activities, and cumulative amount of time spent by the processor in communication activities (sends, receives, and waits). The displayed parameters in Figures 6.59 through 6.61, at 0.052 seconds (snapshot number 23), are identical to those presented for snapshot number 12.

We can observe some recurring patterns in these images, however many of the specific features are yet to be explored. Indeed, these are complex states of the system! We are currently speculating on the application of fractal methods to characterize the complexity of such images. Observe the complementary relationship between Figures 6.57 and 6.58 and between Figures 6.60 and 6.61: dark regions in one correspond to light regions in the other. This is a result of the regular, symmetric properties of 1-D Wave. At a particular instant in time, processors that have spent a relatively large amount of time communicating have necessarily spent a relatively small amount of time computing.

Finally, Figures 6.62 and 6.63 depict cumulative computation time and cumulative communication time, respectively, near the end of program execution (snapshot number 36 at 0.09 seconds). An interesting feature to note here is that the variations among the processors have diminished. That is, there appears to be greater balance in the system; this observation is supported by the global statistics reported in Table 6.19.

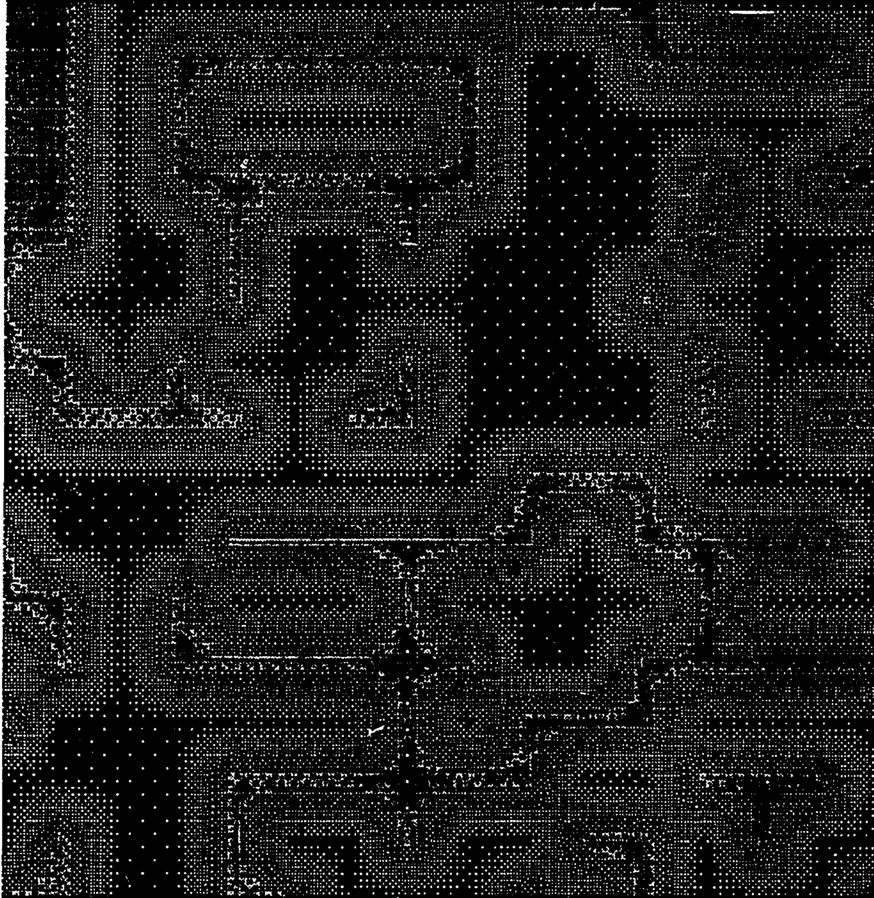


Figure 6.56. Picture of performance (dither plot): 1D Wave, ss#12 at 30 msec., processor activity (black: computing; gray: communicating; white: none)

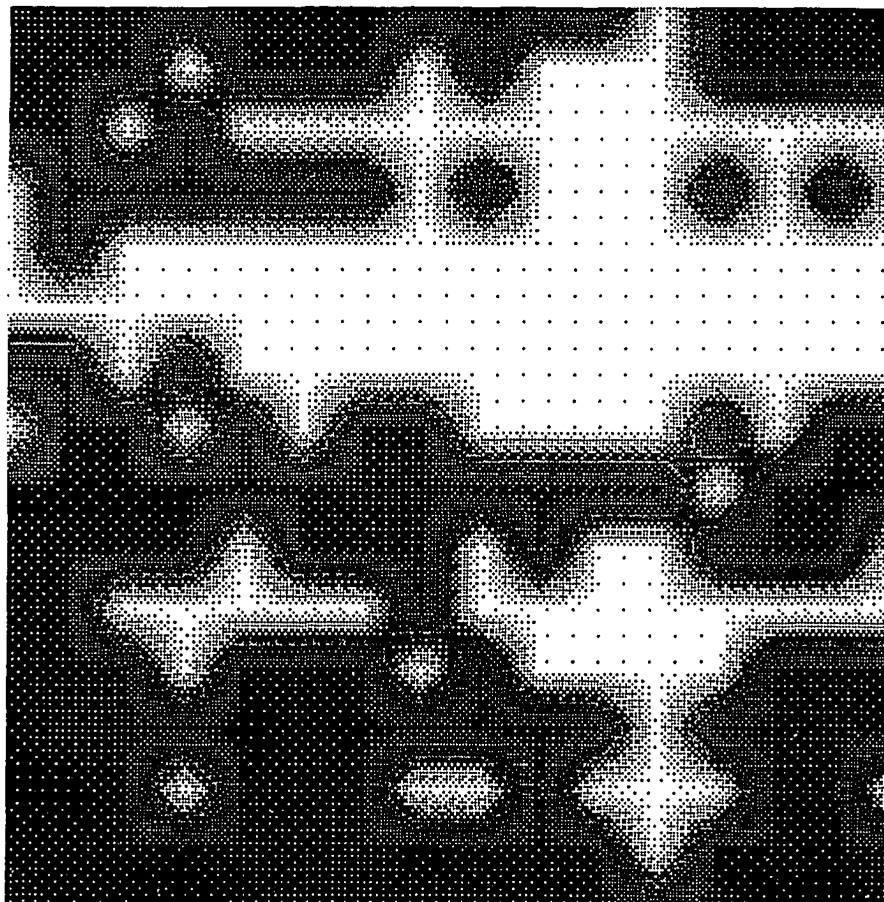


Figure 6.57. Picture of performance (dither plot): 1D Wave, ss#12 at 30 msec., cumulative computation time

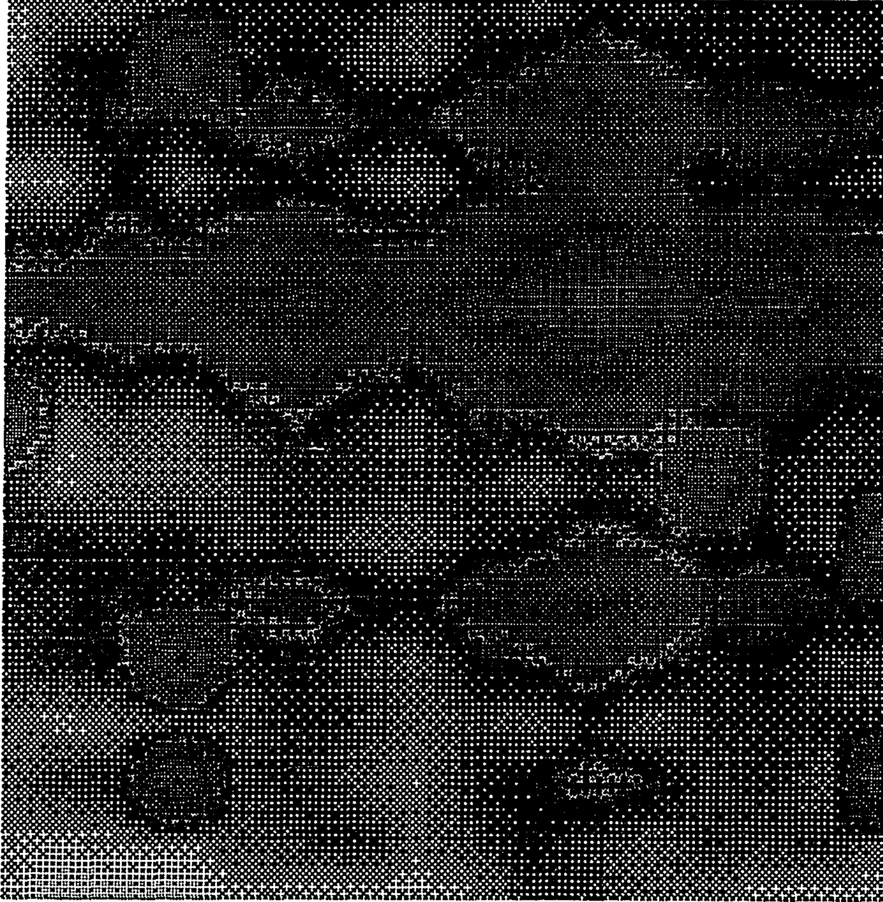


Figure 6.58. Picture of performance (dither plot): 1D Wave, ss#12 at 30 msec., cumulative communication time

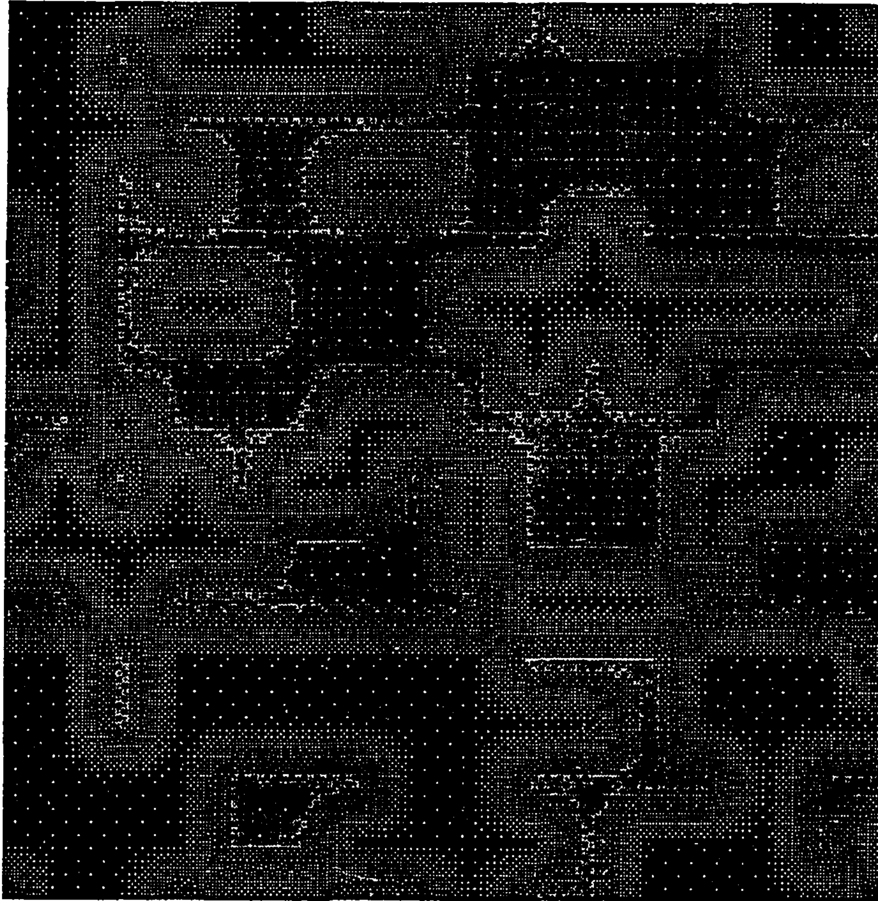


Figure 6.59. Picture of performance (dither plot): 1D Wave, ss#23 at 52 msec., processor activity (black: computing; gray: communicating; white: none)

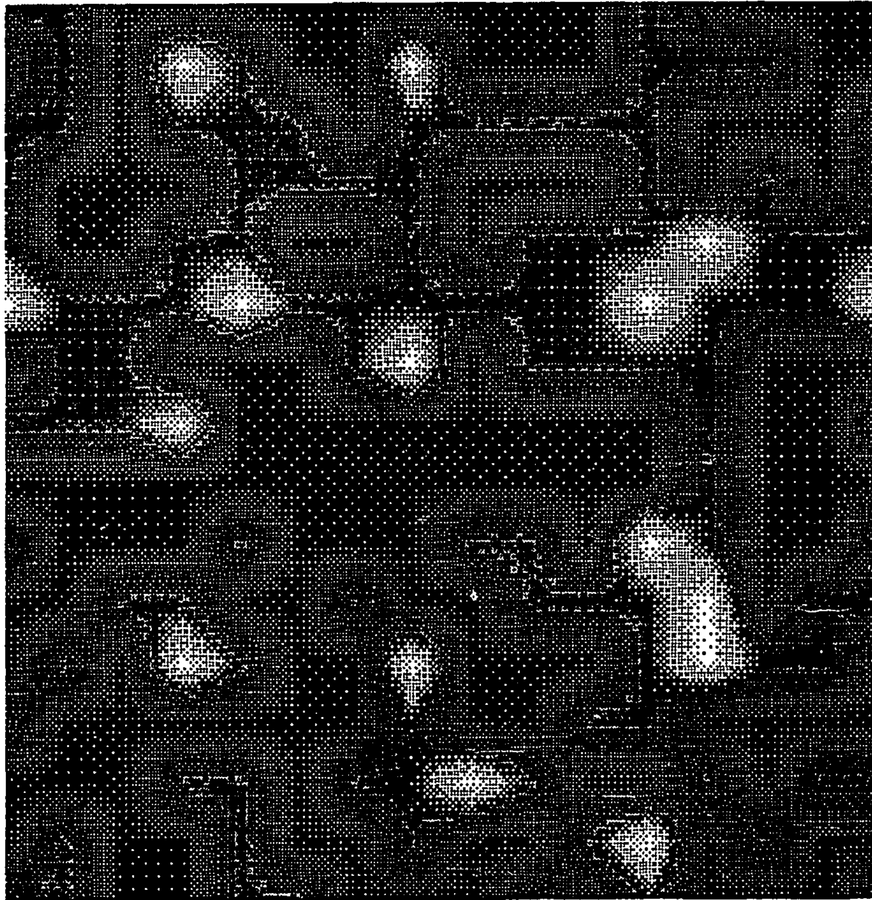


Figure 6.60. Picture of performance (dither plot): 1D Wave, ss#23 at 52 msec., cumulative computation time

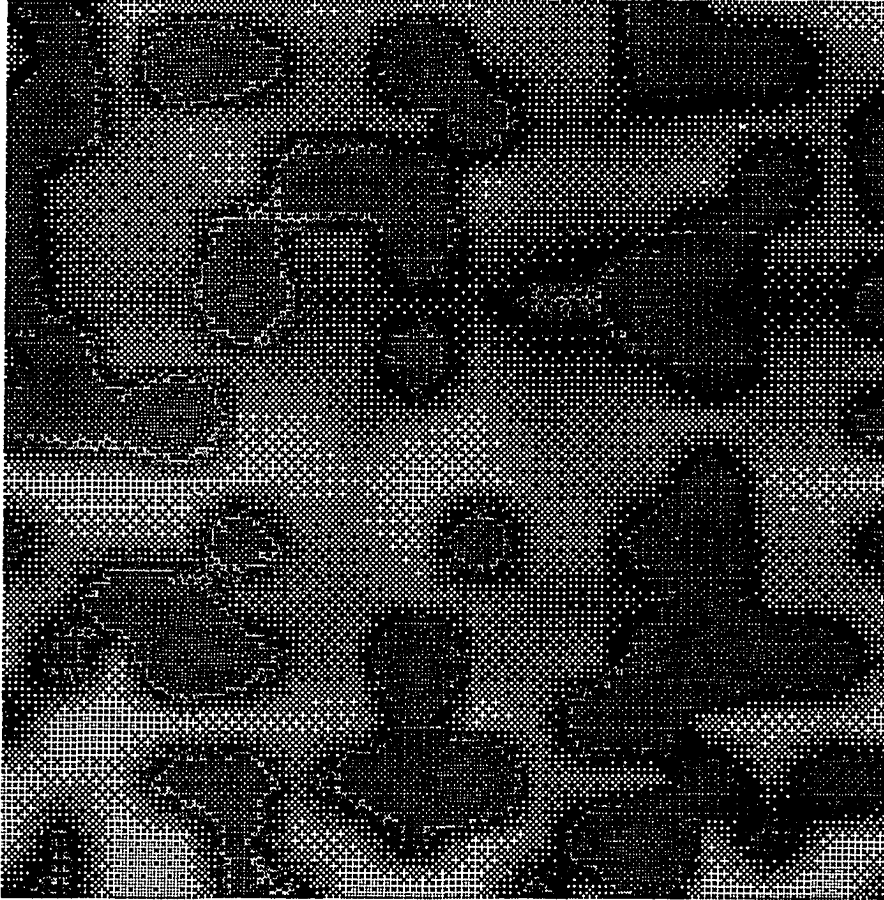


Figure 6.61. Picture of performance (dither plot): 1D Wave, ss#23 at 52 msec., cumulative communication time

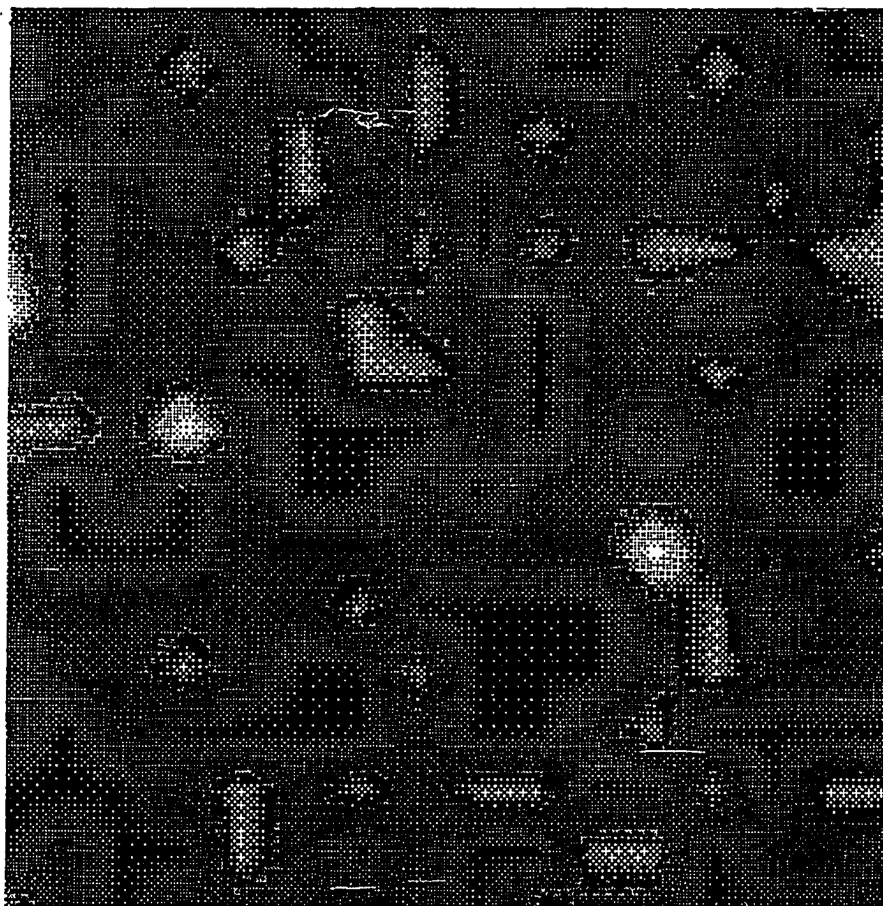


Figure 6.62. Picture of performance (dither plot): 1D Wave, ss#36 at 90 msec., cumulative computation time

Figure 6.63. Picture of performance (dither plot): 1D Wave, ss#36 at 90 msec., cumulative communication time

CHAPTER VII.

DISCUSSION AND CONCLUSIONS

> I think of a computer display as a window on Alice's Wonderland in which a programmer can depict either objects that obey well-known natural laws or purely imaginary objects that follow laws he has written into his program. Through computer displays I have landed an airplane on the deck of a moving carrier, observed a nuclear particle hit a potential well, flown in a rocket at nearly the speed of light and watched a computer reveal its innermost workings.
>
> Ivan Sutherland [Sutherland, 1970]

Future Work

The most obvious direction for future work is to go beyond a prototype. While much work would be required to transform the prototype into a fully configured laboratory, the implementation is sufficient to indicate that there are no insurmountable problems. Equally important, it points to the potential of the approach, particularly the general treatment of measurement data and the use of the data plots, for visualizing program performance on concurrent computers. Based on our experiences with the prototype, we can prescribe with greater accuracy the ideal features of the tools and laboratory.

Another fairly obvious direction is the extension of this approach with the appropriate features so that it may be integrated into a complete parallel programming environment,

possibly in support of visual programming [Shu, 1988]. A shorter term goal would be the inclusion of mechanisms for real-time processing of event data, which is important for program debugging.

The approach may offer new insights into the problem of mapping a parallel algorithm onto an underlying parallel machine. It provides a framework to investigate the effects of using different topologies (particularly different dimensions) at the program, network, and machine levels of concurrent computing. For example, we can compare performance measurements resulting from executing programs on different architectures. This topic is discussed further in the next section.

A few less obvious, but potentially fruitful, directions for future work relate to the following areas:

(1) image algebra,

(2) hypergraphics [Cluff, 1988],

(3) cellular automata models [Wolfram, 1984] and [Toffoli and Margolus, 1987], and

(4) chaos and fractals [Gleick, 1988] and [Zorpette, 1988].

Within the context of our approach, image algebra operations may be an alternative to conventional techniques for calculating summary statistics. The computational methods for image algebra can then be applied to the analysis of performance measurement data. For example, recall that the summary statistic for the average operation count of the system at time T may be defined as:

$$op_cnt_{AVGp} = SUM_p\ (op_cnt)\ /\ N_p \quad \text{at } t=T$$

The SUM_p function adds values of op_cnt from all processors at

time T. N_p is the number of processors. Alternatively, let A be an image at time T, in which the cells are assigned values for the parameter op_cnt. Let I be the identity image (i.e., all ones). Then we may use the dot product operation (•) to define the average operation count at time T:

$$op_cnt_{AVGp} = A \cdot I \;/\; N_p \quad \text{at } t=T$$

Extensions to hypergraphics presentation techniques and tools that explicitly support the display of performance measurement data from event traces may be appropriate. The dot plots, or scatter plots, that were used (generated by MacSpin) are a type of hypergraph. Although cumulative activity and instantaneous activity could be displayed, current activity (i.e., the most recent event occurrences) was not easily displayed.

Cellular automata models are similar in form to the cell plots. If they are also similar in function, they may prove to be useful for modeling concurrent computation at an abstract level.

Finally, fractals (or fractal geometry) may offer a way to describe the (possibly) irregular shapes apparent in the data plots. There is a recent trend toward modeling complex systems using fractals. A distinctive feature of most fractals is self-similarity, that is, similar patterns on different scales or levels. At a low level, we may not be concerned with patterns; however, at a high level, we are concerned with patterns. Typically, there is some sort of boundary between order at the top level and chaos at the bottom level. In complex systems, that boundary between order and chaos tends to be a fractal. So fractals offer a kind of measurement as to where chaos may end and order (or control)

may begin.

A Question of Dimension

In several instances throughout the course of this work, issues regarding dimension have been raised. Dimension was introduced in Chapter I as a property of complex systems and defined as the number of connections from a member of a complex system to its neighboring members. We are faced with questions of dimension when we are investigating the logical and physical networks of nodes in a computer system.

The dimension of the physical network is constrained within the three dimensions of physical space. In addition to topology, the geometry of the interconnections becomes a consideration. However, familiar Euclidean metrics may not be applicable to performance measurements if communication is restricted to orthogonal paths in the system. Performance measurements may need to be stated in terms of "taxicab metrics" [Hillis, 1985].

The mapping of the logical network onto the physical network necessarily places limitations on the topology and dimension of the logical network. For the logical network alone (i.e., considered in isolation), the greater its dimension, the greater its ability to support communication among the nodes. Unfortunately, greater dimensions result in wiring and timing problems for the physical network. Thus, at one extreme, we have large dimension hypercubes, which have nice logical properties. And at the other extreme, we have small dimension meshes, which have nice physical properties. The best logical network is still to be determined. It is

quite possible that a "compound hypercube" network (composed of moderate dimension hypercubes of hypercubes, and so forth) would have the best combination of logical and physical properties. That is, it would avoid the wiring problems of large dimension hypercubes, yet be more effective for communications than a strictly nearest-neighbor mesh [Basore, personal correspondence, 1989]. Interestingly, the fractal nature of a compound hypercube network may be an important aspect of its performance.

Finally, although a gray code mapping scheme was used to assign processors in the logical network to locations in the physical network, an optimal scheme is yet to be determined. Optimal may mean minimizing the length or the density of wires, among other criteria. The tradeoffs between different mapping schemes need to be investigated in order to identify the most important criteria. Our work should facilitate an evaluation of different mapping schemes. The criteria can then be used (by either hardware or algorithm designers) to configure a system for effective and efficient operation.

Research Contributions

This work has resulted in several important contributions to research relating to program performance on multicomputers. We emphasize visualization of performance measurement data. More importantly, we recognize that different computer systems may require different formats for representing performance data. Also, we propose to treat performance data in the sense of general multivariate data and apply the techniques and tools of multivariate data analysis for analyzing and

displaying the data. However, we can customize our approach since performance data is specialized because of its temporal and spatial characteristics.

The idea of a machine perspective, outlined in Chapters III through V and illustrated in Chapter VI, distinguishes our approach to presenting measurement data. The two- and three-dimensional data plots introduced to graphically represent program performance are unique in this area. Using this format, a system with hundreds or thousands of processors can be displayed at once. Other graphical representations currently in use do not easily support a system having a large number of processors, particularly a global or macroscopic view of the system. In particular, the development of images of program states is a novel contribution and presents numerous opportunities for future study. Also, a two- or three-dimensional plot is appropriate to accurately account for the behavior of the computer system in both time and space. It facilitates showing the flow or movement of granules of computation and communication throughout the system. Thus, we emphasize both computation and communication activities, and account not only for the time spent in these activities but also for the space used by these activities.

The metrics that we defined capture the computation and communication information in meaningful ways. An objective of this work was to effectively couple qualitative observations and quantitative measurements of the system into a coherent representation of performance.

Finally, we have developed a framework in which to study patterns in program execution. Patterns are visual and offer insight into the behavior of concurrent algorithms and the

systems that execute them. Using appropriate tools, we can view the system as a whole as well as focus our attention on particular parts of the system, as dictated by an interesting or unusual feature within an image. Closer inspections that include details about the program and machine can reveal the innermost workings of the system. For example, we might find that an algorithm behaves poorly because it generates too much traffic at a particular location in the system at a particular time. Or we might find that a faulty processor is causing inefficiences.

Furthermore, we are only beginning to understand the importance of structure in concurrent computing: the structure of the problem, the program, the network, and the machine, and the relationship among these structures. The view of performance that we have developed should be a useful tool for investigating these structures.

BIBLIOGRAPHY

Abraham, S., Gottlieb, A., and Kruskal, C. "Simulating Shared-Memory Parallel Computers." Ultracomputer Note No. 70, New York University, April 1984.

Agha, G. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA: MIT Press, 1986.

Agrawal, D., Janakiram, V., and Pathak, G. "Evaluating the Performance of Multicomputer Configurations." IEEE Computer, 19, No. 5 (May 1986): 23-37.

Athas, W. and Seitz, C. "Multicomputers: Message-Passing Concurrent Computers." IEEE Computer, 21, No. 8 (August 1988): 9-23.

Babb, R. G. and Di Nucci, D. C. "Design and Implementation of Parallel Programs with Large-Grain Dataflow." In The Characteristics of Parallel Algorithms, edited by Jamieson, Gannon, and Douglass. Cambridge, MA: MIT Press, 1987.

Backus, John. "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs." Communications of the ACM, 21, No. 8 (1978): 613-641.

Barasch, L., Lakshmivarahan, S., and Dhall, S. "Generalized Gray Codes and their Properties." Proceedings of the Third International Conference on Supercomputing, Vol. III. New York: IEEE, 1988.

Barnes, G., Brown, R., Katz, M., Kuck, D., Slotnick, D., and Stoker, R. "The Illiac-IV Computer." IEEE Transactions on Computers C-17 (1968): 746-757.

Barnsley, M. Fractals Everywhere. Boston: Academic Press, 1988.

Batcher, K. E. "Design of a Massively Parallel Processor." IEEE Transactions on Computers, C-29, No. 9 (1980): 100-115.

Beetem, J., Denneau, M., and Weingarten, D. "The GF11 Supercomputer." Proceedings of the Twelfth Annual Symposium on Computer Architecture. New York: IEEE, 1985.

Bell, C. G. and Newell, A. Computer Structures: Readings and Examples. New York: McGraw Hill, 1971.

Berman, F. "Experience with an Automatic Solution to the Mapping Problem." In The Characteristics of Parallel Algorithms, edited by Jamieson, Gannon, and Douglass. Cambridge, MA: MIT Press, 1987.

Bokhari, S. H. Assignment Problems in Parallel and Distributed Computing. Boston: Kluwer Academic Publishers, 1987.

Borodin, A. "On Relating Time and Space to Size and Depth." SIAM J. Computing, 6, No. 4 (1977): 733-744.

Brandis, R. C. "IPPM: Interactive Parallel Program Monitor." M.S. thesis, Oregon Graduate Center, August 1986.

Brantley, W., McAuliffe, K., and Weiss, J. "RP3 Processor-Memory Element." Proceedings of the 1985 International Conference on Parallel Processing. Washington, D.C.: IEEE Computer Society Press, 1985.

Brown, Marc H. "Exploring Algorithms Using Balsa-II." IEEE Computer, 21, No. 5 (May 1988): 14-36.

Burks, A. W. "Programming and Structure Changes in Parallel Computers." Lecture Notes in Computer Science 111 (1981): 1-23.

Buzen, J. P. "Fundamental Operational Laws of Computer System Performance." Acta Informatica, 7, No. 2 (1976): 167-182.

Campbell, R. H. and Reed, D. A. "Tapestry: Unifying Shared and Distributed Memory Parallel Systems." Technical Report No. TTR88-1. Dept. of Computer Science, Univ. of Illinois, August 1988.

Carroll, Lewis. Alice's Adventures in Wonderland. Stamford, CT: Longmeadow Press, 1988. (Original publication, 1865)

Chandy, K. and Lamport, L. "Distributed Snapshots: Determining Global States of Distributed Systems." ACM Transactions on Computer Systems 3 (February 1985): 63-75.

Cluff, E. "A Characterization and Categorization of Higher Dimensional Presentation Techniques." M.S. thesis, Dept. of Computer Science, Brigham Young University, 1988.

Couch, A. "Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors." Ph.D. dissertation, Department of Computer Science, Tufts University, April 1988.

Couch, A. "Problems of Scale in Displaying Performance Data for Loosely-Coupled Multiprocessors." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Crowther, W. "Performance Measurements on a 128-Node Butterfly Parallel Processor." Proceedings of the 1985 International Conference on Parallel Processing, 531-540. Washington, D.C.: IEEE Computer Society Press, 1985.

Dally, William J. A VLSI Architecture for Concurrent Data Structures. Boston: Kluwer Academic Publishers, 1987.

Deshpande, S. R., Jenevein, R. M., and Lipovski, G. J. "TRAC: An Experience with a Novel Architectural Prototype." TRAC Report. University of Texas, Austin, TX, 1985.

Dijkstra, E. W. "Solution of a Problem in Concurrent Programming." Communications of the ACM 8 (1965): 569-570.

Einstein, E. "A Tool to Aid in Mapping Computational Tasks to a Hypercube." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Enslow, P. "Multiprocessing Organization - A Survey." Computing Surveys, 9, No. 1 (1977): 103-129.

Flanders, P. M. "Efficient High Speed Computing with the Distributed Array Processor." In High Speed Computer and Algorithm Organization, edited by Kuck, Lawrie, and Sameh, 113-127. New York: Academic Press, 1977.

Flower, J. "NDB and PM: Debugging and Performance Tools for the Parallel Programmer." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Flynn, M. J. "Very High-Speed Computing Systems." Proceedings of the IEEE 54 (1966): 1901-1909.

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. Solving Problems on Concurrent Processors. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Frenkel, K. "Evaluating Two Massively Parallel Machines." Communications of the ACM, 29, No. 8 (1986): 752-758.

Fujimoto, R. "Simon: A Simulator of Multicomputer Networks." Report No. UCB/CSD 83/136. Electronics Research Lab, University of California, Berkeley, 1983.

Gajski, Daniel D., and Peir, Jih-Kwon. "Essential Issues in Multiprocessor Systems." IEEE Computer, 18, No. 6 (June 1985): 9-27.

Geist, G. "A Machine-Independent Communication Library." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Gelernter, D. "Programming for Advanced Computing." Scientific American, 257, No. 4 (October 1987): 91-98.

Gleick, J. Chaos: Making a New Science. New York: Viking, 1987.

Gottlieb, A. and Schwartz, J. T. "Networks and Algorithms for Very Large Scale Computation." IEEE Computer, 15, No. 1 (January 1982): 27-36.

Gottlieb, A., Grishman, R., Kruskal, C., McAuliffe, K., Rudolph, L., and Snir, M. "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer." IEEE Transactions on Computers C-32 (1983): 175-189.

Gustafson, J., Montry, G., and Benner, R. "Development of Parallel Methods for a 1024-Processor Hypercube." SIAM Journal on Scientific and Statistical Computing, 9, No. 4 (July 1988): 609-638.

Hayes, J., Mudge, T., Stout, Q., Colley, S., and Palmer, J. "A Microprocessor-based Hypercube Supercomputer." IEEE Micro, 6, No. 5 (October 1986): 6-17.

Haynes, L. S., Lau, R. L., Siewiorek, D. P., and Mizell, D. W. "A Survey of Highly Parallel Computing." IEEE Computer, 15, No. 1 (January 1982): 9-24.

Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages." Artificial Intelligence, 8, No. 3 (1977): 323-364.

Hillis, W. D. "New Computer Architectures and Their Relationship to Physics, or Why Computer Science is No Good." International Journal of Theoretical Physics, 21, No. 3/4 (1982): 255-262.

Hillis, W. D. "The Connection Machine: A Computer Architecture Based on Cellular Automata." Physica 10D (1984): 213-228.

Hillis, W. D. The Connection Machine. Cambridge, MA: MIT Press, 1985.

Hillis, W. D., and Steele, G. L. "Data Parallel Algorithms." Communications of the ACM, 29, No. 12 (1986): 1170-1183.

Holland, J. H. "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously." Proceedings of the 1959 E.J.C.C. (1959): 108-113.

Hopfield, J. and Tank, D. "Computing with Neural Circuits: A Model." Science 233 (August 8, 1986): 625-632.

Hwang, K., and Briggs, F. A. Computer Architecture and Parallel Processing. New York: McGraw-Hill, 1984.

Jamieson, L., Gannon, D., and Douglass, R., eds. The Characteristics of Parallel Algorithms. Cambridge, MA: MIT Press, 1987.

Jaynes, E. T. "Information Theory and Statistical Mechanics." Physical Review 106 (1957): 620-630.

Keyes, R. W. "Physical Problems and Limits in Computer Logic." IEEE Spectrum 6 (1969): 36-45.

Keyes, R. W. "Physical Limits in Digital Electronics." Proceedings of the IEEE, 63, No. 5 (1975): 740-767.

Keyes, R. W. "Fundamental Limits in Digital Information Processing." Proceedings of the IEEE, 69, No. 3 (1981): 267-278.

Keyes, R. W. and Landauer, R. "Minimum Energy Dissipation in Logic." IBM Journal of Research and Development 14 (1970): 152-157.

Kleinrock, L. "Distributed Systems." Communications of the ACM, 28, No. 11 (November 1985): 1200-1213.

Knox, D. "Investigation of a Microcomputer-based Multiprocessing System." M. S. thesis, Iowa State University, Ames, IA, 1983.

Krumme, D. "Problems of Scale in Collecting Performance Data on Loosely-Coupled Multiprocessors." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Krumme, D., Couch, A., House, B., and Cox, J. "The Triplex Tool Set for the NCUBE Multiprocessor." Report. Dept. of Computer Science, Tufts University, March 1989.

Kruskal, C. P. "Upper and Lower Bounds on the Performance of Parallel Algorithms." Ph.D. dissertation, New York University, October 1981.

Kuck, D. The Structure of Computers and Computations. New York: John Wiley and Sons, 1978.

Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." Communications of the ACM 21 (July 1978): 558-565.

Landauer, R. "Wanted: A Physically Possible Theory of Physics." IEEE Spectrum, 4, No. 9 (1967): 105-109.

LeBlanc, T. "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study." Butterfly Project Report 3. Computer Science Dept., Univ. of Rochester, January 1986.

Lee, R. "Empirical Results on the Speed, Efficiency, Redundancy, and Quality of Parallel Computations." Proceedings of the 1980 International Conference on Parallel Processing, 91-96. Washington, D.C.: IEEE Computer Society Press, 1980.

Levin, L. "Do Chips Need Wires?" Personal correspondence. Laboratory for Computer Science, MIT, Cambridge, MA, 1988.

Lipovski, G. J. and Malek, M. Parallel Computing: Theory and Comparisons. New York: John Wiley and Sons, 1987.

Malony, A. "An Integrated Performance Data Collection, Analysis, and Visualization System." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Mead, C. and Conway, L. Introduction to VLSI Systems. Reading, MA: Addison-Wesley, 1980.

Melamed, B. and Morris, R. "Visual Simulation: The Performance Analysis Workstation." IEEE Computer, 18, No. 8 (August 1985): 87-94.

Mundie, C. "Interacting with the Tiny and the Immense." BYTE, 14, No. 4 (April 1989): 279-288.

Nichols, K. and Edmark, J. "Modeling Multicomputer Systems with PARET." IEEE Computer, 21, No. 5 (May 1988): 39-48.

Patton, Peter C. "Multiprocessors: Architecture and Applications." IEEE Computer, 18, No. 6 (1985): 29-40.

Peltz, D. "Visualization of Data." MIPS, 1, (February 1989): 35-38.

Reed, D. A. "Instrumenting Distributed Memory Parallel Systems: A Report." In Instrumentation for Future Parallel Computer Systems, edited by Bucher, Simmons, and Koskela. Reading, MA: Addison-Wesley, 1989.

Reed, D. A. and Fujimoto, R. M. Multicomputer Networks: Message-Based Parallel Processing. Cambridge, MA: MIT Press, 1987.

Reed, D. A. and Grunwald, D. "The Performance of Multicomputer Interconnection Networks." IEEE Computer, 20, No. 6 (June 1987): 63-73.

Roberts, R. "Hybrid Performance Measurement Instrumentation for Loosely-Coupled MIMD Systems." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Rover, D. T. "Implementation of a Multiprocessor Architecture for Boundary Value Problems." M.S. thesis, Iowa State University, Ames, IA, 1986.

Rover, D. T., Prabhu, G. M., and Wright, C. T. "Characterizing the Performance of Concurrent Computers: A Picture is Worth a Thousand Numbers." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Rudolph, D. and Reed, D. "A Performance Evaluation Tool for the Intel iPSC/2." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Rummelhart, D. and McClelland, J. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vols. 1 and 2. Cambridge, MA: MIT Press, 1986.

Sanguinetti, J. "Performance of a Message-Based Multiprocessor." IEEE Computer, 19, No. 9 (1986): 47-55.

Savage, J. "Computational Work and Time in Finite Machines." Journal of the ACM 19 (1972): 660-674.

Schuster, D. "Visualization with DataScope and ImageTool." MIPS, 1, (March 1989): 33-35.

Schwartz, J. T. "Ultracomputers." ACM Transactions on Programming Languages and Systems, 2, No. 4 (1980): 484-521.

Segall, Z. and Rudolph, L. "PIE: A Programming and Instrumentation Environment for Parallel Processing." IEEE Software, 2, No. 11 (November 1985): 22-37.

Seitz, Charles L. "The Cosmic Cube." Communications of the ACM, 28, No. 1 (1985): 22-33.

Seitz, C. and Matisoo, J. "Engineering Limits on Computer Performance." Physics Today, 31, (May 1984): 38-45.

Shaw, D. E. "The NON-VON Supercomputer." Technical Report. Dept. of Computer Science, Columbia Univ., New York, August 1982.

Shu, N. C. Visual Programming. New York: Van Nostrand Reinhold Company, 1988.

Siegel, H., Siegel, L., Kemmerer, F., Mueller, P., Smalley, H., and Smith, D. "Passm: A Partitionable Multimicrocomputer SIMD/MIMD System for Image Processing and Pattern Recognition." Tech. Rept. TR-EE 79-40. Purdue University, 1979.

Snodgrass, R. "A Relational Approach to Monitoring Complex Systems." ACM Transactions on Computer Systems, 6, No. 2 (May 1988): 157-196.

Snodgrass, R. and Ahn, I. "Temporal Databases." IEEE Computer, 19, No. 9 (September 1986): 35-42.

Snyder, L. "Introduction to the Configurable, Highly Parallel Computer." IEEE Computer, 15, No. 1 (January 1982): 47-56.

Snyder, L. "Parallel Programming and the Poker Programming Environment." IEEE Computer, 17, No. 7 (July 1984): 27-36.

Stone, H. S. High-Performance Computer Architecture. Reading, MA: Addison-Wesley, 1987.

Sutherland, I. E. "Computer Displays." Scientific American, 223, (June 1970): 57-81.

Sutherland, I. and Mead, C. "Microelectronics and Computer Science." Scientific American, 237, No. 9 (1977): 210-228.

Swan, R. J., Fuller, S. H., and Siewiorek, D. P. "Cm* - A Modular, Multi-Microprocessor." Proceedings AFIPS Conference 46 (1977): 637-643.

Toffoli, T. and Margolus, N. Cellular Automata Machines: A New Environment for Modeling. Cambridge, MA: MIT Press, 1987.

Unger, S. H. "A Computer Oriented Towards Spatial Problems." Proceedings of IRE, 46, No. 10 (1958): 1744-1750.

Wilcke, W. "The IBM Victor Multi-Processor Project." Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications. New York: ACM, 1989.

Wolfram, S. "Cellular Automata as Models of Complexity." Nature, 311, No. 4 (1984): 419-424.

Yao, A. "The Entropic Limitations on VLSI Computations." Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing. New York: ACM, 1981.

Zorpette, G. "Fractals: Not Just Another Pretty Picture." IEEE Spectrum, 25, No. 10 (October 1988): 29-31.

ACKNOWLEDGEMENTS

I would like to acknowledge the contributions of the following to my academic and personal pursuits. First, let me thank Dr. Charles Wright and Dr. Paul Basore, major professors for my doctoral and masters work, respectively. I have benefited from their involvement throughout my graduate work. They offered guidance, yet the freedom to pursue my own interests. Their beliefs and methods have been stimulating and motivating forces in my work; each in his own way has offered something invaluable to my growth as a professional and as a person.

I would also like to thank the other members of my Program of Study committee, for their willing discussion of my efforts and participation in my degree program: Dr. Terry Smay, Dr. Gurpur Prabhu, Dr. Arthur Pohm, and Dr. Jim Davis. Dr. Chip Comstock attended the final oral examination as a substitute and deserves thanks too. I appreciate the assistance of Gary Bridges, Scott Irwin, and Paul Dorweiler in the Digital Systems Laboratory; when I needed something in the lab, they provided it. Several other colleagues and friends have listened to my ideas and given constructive comments at various stages of this research, including Roy Zingg, Robert Burton, Huey Ling, Jan Stone, and Deb Knox.

Much of this work was completed under an IBM Graduate Fellowship. I appreciate the funding provided by the Graduate Fellowship Program and in particular the efforts on my behalf of Dr. Jerry Balm and Dr. V. Sadagopan of IBM.

Also, thanks goes to my family for their love and support. To my parents, for enduring and endearing guidance; to my brothers, one for clearing a studious path and the other for telling me I was not always right; to my sister, for being a friend and a great second mom to our nine-month-old daughter, Brittany; and to my husband, Craig, for caring about me and my work. Finally, I dedicate this to Brittany. She is a wonderful distraction and has enriched this work in ways that no book could.