1994

Performance visualization for parallel programs: task-based, object-oriented approach

Jungsun Kim
Iowa State University

Performance visualization for parallel programs: Task-based, object-oriented approach

Kim, Jungsun, Ph.D.

Iowa State University, 1994

Performance visualization for parallel programs: Task-based, object-oriented approach

by

Jungsun Kim

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:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa

1994

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Advances in microcomputer and VLSI technologies have made it feasible to build a parallel processing system by interconnecting a large number of processors and/or memory modules. A parallel processing system can be organized as a distributed memory concurrent computer system as shown in Figure 1.1.

A distributed memory concurrent system consists of a large number, possibly hundreds or thousands, of processing elements (PEs) interconnected in some topologies. Each PE contains a processor, a local memory, and a network interface circuit for connecting to other PEs. Specialized co-processors for floating-point, communications, graphics, or second storage operations may also be included on a PE. This type of computer system is sometimes called a *multicomputer* system as compared to a *multiprocessor* system in which PEs share a global memory. Cooperation and synchronization among PEs are achieved via asynchronous message-passing mechanism in multicomputers. The hypercube, the Butterfly, and the Transputer are a few examples that belong to multicomputers.

Multicomputers are capable of solving large scale problems that cannot be solved efficiently in a conventional uniprocessor environment. However, developing and analyzing performance of concurrent programs on multicomputers is not an easy task because of the added complexity due to the collective and simultaneous interaction

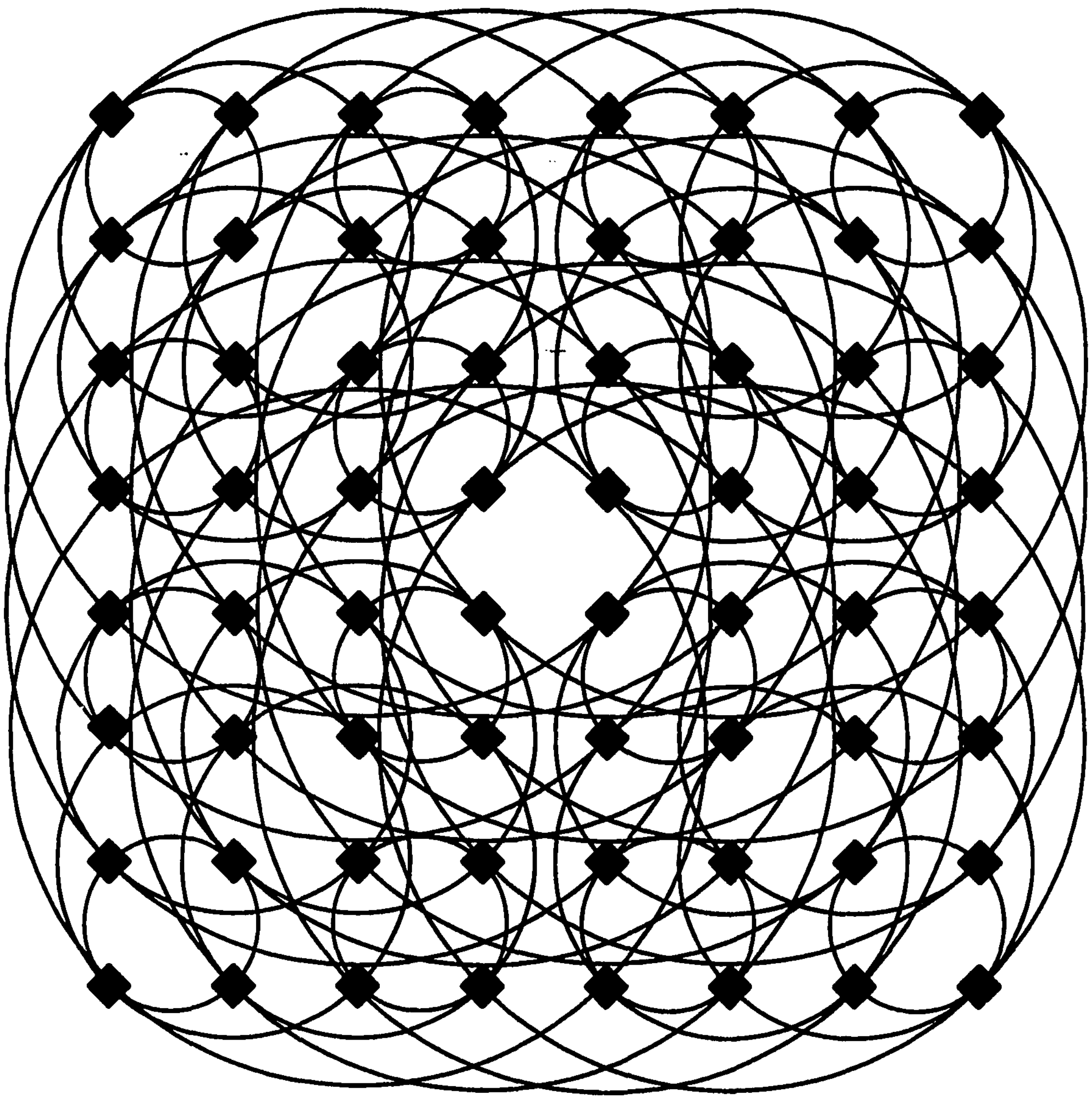


Figure 1.1: A Distributed Memory Concurrent Computer

of many PEs engaged in computation and communication activities. Therefore, it is critical for programmers to have appropriate methods and tools which would offer insight into the development of parallel programs.

Development of efficient parallel programs normally involves an iterative refinement framework with three phases — design, measurement, and modification of the performance of successive computation prototypes as shown in Figure 1.2. This iterative performance fine-tuning is called *performance debugging* [29]. One goal of performance debugging is to produce an ideal program-machine mapping for a target machine to achieve an optimal performance.

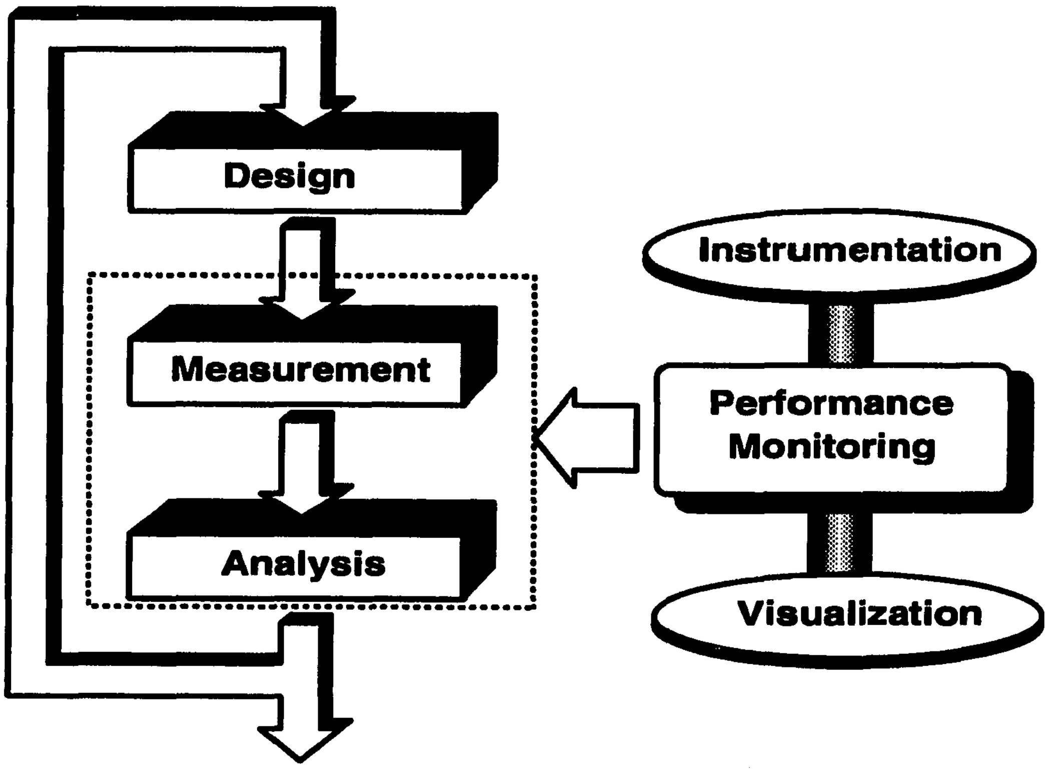


Figure 1.2: Performance Debugging

Performance debugging is usually accomplished by *performance monitoring* which refers to the combined activities of *program instrumentation* and *performance visualization.* Instrumentation answers the question, "What is observed?"; and visualization, "How is it viewed?" [46]. Answers to these questions facilitate conceptual understanding of dymanic activities and performance fine-tuning for parallel programs. There is a producer-consumer relationship between instrumentation and visualization. Instrumentation generates performance related data. Visualization tools process, reduce, transform, and present collected data in the graphical format. There are other ways of representing instrumented information such as numbers and tables. However, graphics have been an effective way of delivering information to humans. Another driving force to the visualization is the X window system and the advance of high-resolution graphics technology. A variety of tools have been developed for monitoring parallel systems. The reader is referred to an extensive list of references in [21, 41, 52]. We will focus our attention on performance visualization in this dissertation.

Performance visualization is an area of computer graphics that consists of techniques and tools that allow program or data to be specified, observed and manipulated in a pictorial, rather than numerical or textual format [42]. Graphical representation of performance information greatly increases the bandwidth of man-machine communication. It facilitates human understanding and effective use of computer programs. The behavior of parallel programs on advanced architectures is often extremely complex, and monitoring the performance of such programs can generate vast quantities of data. So it seems natural to use visualization to gain insight into the behavior of the parallel programs so we can better understand them and improve their performance. Without visualization, programmers must often go through the tedious task of conceptualizing the relationships among an enormous amount of primitive raw numerical data. Eliminating this tedium is essential to making performance debugging a productive activity [29]. Recently, there is a growing interest in performance visualization of parallel programs being executed on distributed memory concurrent computer systems as well as multiprocessor systems. Performance visualization is now considered to be an integral component of a programming environment for parallel systems.

To be effective and useful, a performance visualization tool must provide a rich set of views reflecting various aspects of performance data. Since the application domains for distributed memory concurrent computer system keep growing, the visualization tool should be flexible enough to accommodate unknown future demands of users (eg. new performance perspectives and application-specific views) with little effort.

Most of the performance visualization tools to date, developed with conventional procedural programming languages, are generally not easy to expand, even if they are well structured. This is largely because the program design using procedure languages are inherently based on the functional decompositions of a target application. And procedural languages normally lack capabilities to support expandable design. Moreover, trace data and performance views are semantically tightly coupled in most cases. In case the trace record formats are changed or new trace record formats are introduced, significant portions of source code must be rewritten. An existing visualization tool may not be reused on an application which requires a totally different set of trace record formats.

Also when a performance view is tightly coupled with the performance aspect of which it is responsible, we may need to rewrite a view from scratch even if it has very similar characteristics as one of the previously developed views. This is really a waste of time and effort.

Object-oriented programming provides a powerful mechanism to facilitate the design and implementation of extensible, resuable, and therefore general programs. An object-oriented programming also encourages a methodology for designing and creating a program as a set of autonomous components which can be developed independently. For the last decade, an object-oriented programming has been successfully applied to a number of applications, especially to the applications which require highly interactive graphical user interfaces [49]. In developing our visualization environment, called Concurrent Object-Oriented ParaGraph (COOPG), an object-oriented approach is adopted and as an implementation language, C++ is chosen.

In this dissertation, we describe an object-oriented approach to general purpose performance visualization for parallel programs running on distributed memory concurrent computer systems. Lots of effort has been directed toward building a tool which is resilient to environmental changes. Our work is primarily based on the ParaGraph [20], a widely used performance visualization tool for parallel programs.

Motivation

As an effort to develop a parallel program development environment, a number of tools for performance monitoring have appeared for the last decade. Visualization of parallel computers has been the subject of recent Ph.D thesis [41, 43, 32], technical articles [3, 23, 20, 27, 32, 31, 34, 48, 50, 44], and even one book [52]. Integrated environments that combine parallel programming, debugging and monitoring have also been developed [13, 51]. Instead of enumerating and describing the currently available combinations of tools for instrumentation and visualization, we will focus our attention on one of the successful pairs on which our research is based — PICL [16] and ParaGraph [20].

Currently, at the Scalable Computing Laboratory in Ames Lab, PICL and ParaGraph are adopted as our performance monitoring tools for the development of parallel programs on an nCUBE 2 system. Case studies of how these tools are applied to performance debugging and fine-tuning can be found in [46, 45]. We intend to enhance these tools for better performance and extensibility as a first step toward developing a fully integrated parallel program development environment.

Organization of the Dissertation

This dissertation is organized as follows. Introduction and motivation of the research is provided in this chapter. Chapter 2 describes the background information and terms for the performance monitoring including the instrumentation and the visualization. A standardization effort in the area of performance visualization and a proposed performance environment architecture by a standardization working group are also briefly described in Chapter 2. Related works are reviewed in Chapter 3. Chapter 4 describes the performance metrics which must be measured and provided by a performance visualization environment. Chapter 4 introduces the basic concepts and terminology of an object-oriented programming. Advantages and disadvantages of an object-oriented programming and concurreny issues are described as well. Chapter 6 addresses the design goal of our performance visualization environment. Chapter 7 describes the design and the implementation of two generations of our performance visualization environments — the Object-Oriented ParaGraph and the Concurrent Object-Oriented ParaGraph. Finally, discussion and conclusion are provided in Chapter 8.

CHAPTER 2. BACKGROUND

This chapter briefly explains the terms and background information about the performance visualization, along with the standardization effort for the performance visualization environment.

Performance Monitoring

Performance measures can be obtained by applying the following evaluation methods: benchmarks, monitoring (hardware or software), emulation, simulation, and analytical modeling. Benchmarks are intended to be used for the evaluation of a particular machine. Analytical modeling is too complicated to be applied to complex systems. Emulation and simulation are somewhat flexible but they cannot generally be used to analyze ordinary concurrent programs. Since we are interested in analyzing and debugging concurrent programs in complex multicomputer systems, monitoring is the method to be used.

The instrumentation and the visualization compose an integral part of performance monitoring tools. The instrumentation recognizes and records events of interest as they occur to support the analysis of parallel programs. The visualization then processes the collected information and displays it in a reasonable format (graphical format) that a human can easily understand.

Instrumentation

Analysis of a program can be driven by a record of the major state transitions (events) produced by each cooperating process. Instrumentation involves observing and recording information at particular points in the system. In this subsection, the instrumentation design problem is defined by examining the ways in which the collected data will be used, and the constraints that are placed on the eventual solution.

Reasons for Performance Measurement

Performance measurement can be used for both system analysts or designers, and application programmers for their own purposes [41].

- **System Analyst's View**
  1. Capacity planning, System tuning, Guiding policy decision, and Accounting
  2. Design decision guide for future products
  3. Insight into the behavior of the operating system
- **Application Programmer's View**
  1. Performance Debugging — the twin task of debugging for performance and debugging for correctness in a multiprocessor/multicomputer environment may require different approaches than have been employed in uniprocessor environments. Several research efforts have shown the utility of an event

trace, that is a time ordered record of interactions and flow-control, in analyzing a parallel application's behavior.

A Statement of the Instrumentation Problem

There are four major questions for the designer of an instrumentation tool as shown in Figure 2.1.

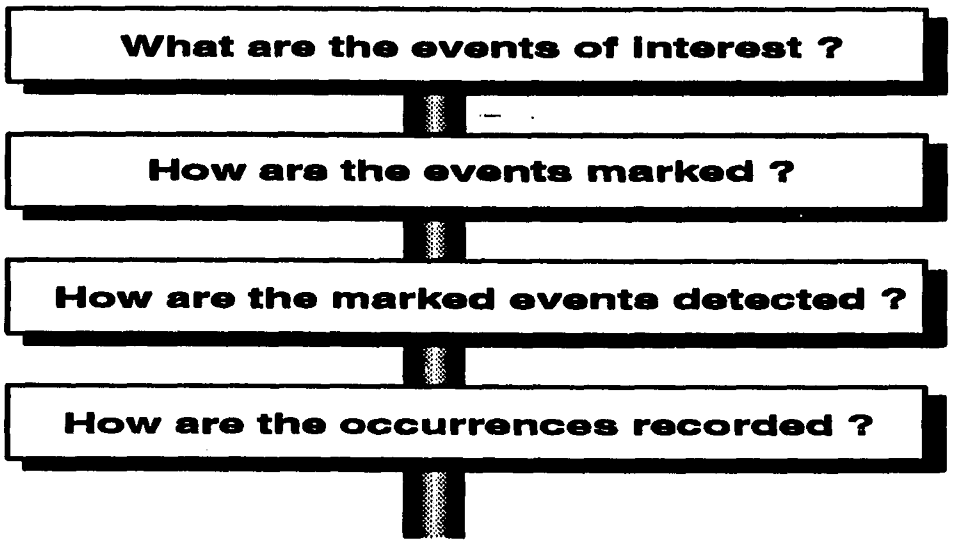

What are the events of interest ?
How are the events marked ?
How are the marked events detected ?
How are the occurrences recorded ?

Figure 2.1: Issues of Concern

Many techniques have been used to collect information from running systems for post experiment analysis, and even in some cases, for analysis in real time [41]. There are tradeoffs that must be made in constructing a solution to the problem of designing a device that recognizes and records events as they occur, as *event collector*, to support the analysis of parallel programs. It is assumed here that performance measurement is being conducted to aid in application performance debugging and, to a lesser extent, debugging for correctness. The following summaries the requirements of instrumentation tools [41].

1. Provide insight into the temporal and spatial relations between cooperating processes.

    - The data collection mechanism should produce a time ordered record of event occurrences.

    - The events of interest are related to the flow of control within the program and to the allocation of resources within the system.

2. Must not be peculiar to a particular programming style, idiom, or environment.

3. Time durations must be measurable to a constant resolution and reasonable accuracy.

4. Must have minimal effect on running applications.

    - The runtime cost and perturbation caused by instrumentation mechanism should be predictable or minimal.

5. Must be inexpensive.

The Design Space

Instrumentation techniques are classified into three categories according to the way in which events are marked, detected and recorded.

- **software instrumentation:** events are recognized and recorded by software executing within the same system as the measured application.

- **hardware instrumentation:** events are recognized and recorded by an agent external to the system executing the measured application.
- **hybrid instrumentation:** events are marked by software inserted into the measured application, event recording is performed by an external agent.

Software instrumentation is typically designed to collect and record high level application and system events, such as entry and exit to program modules, scheduling transitions, I/O requests, and message send/receive requests. A potentially large overhead is a major problem. Hardware instrumentation has been used to collect events at the processor level, such as initiation and completion of memory access, cache hits and changes in task priority. Event recognition and collection impose no overhead on the instrumented system. Hybrid instrumentation has been employed to collect the low level events associated with hardware instrumentation and the software level events associated with software instrumentation. Hybrid instrumentation takes advantage of both software and hardware instrumentations while overcoming the deficiencies of both.

Visualization

Recently, there has been a great deal of interest in systems that utilize graphics in human-computer communications, in programming, and in visualizing program, data and performance. By *visualization* we mean the use of visual representations (such as graphics, images, or animation sequences) to illustrate program, data, performance statistics, or the dynamic behavior of a complex system. There are three closely related but different areas of visualization as shown in Figure 2.2: scientific

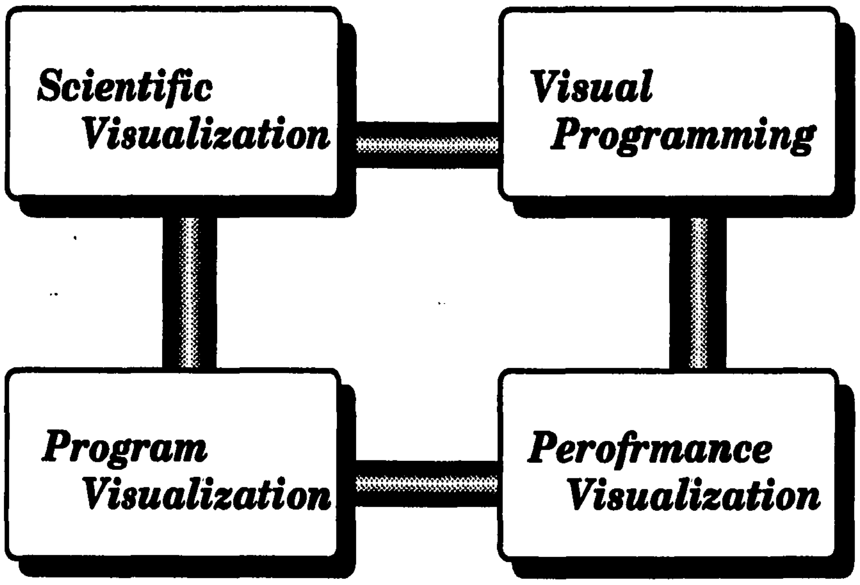


Figure 2.2: Related Areas of the Visualization

visualization, program visualization and visual programming.

Scientific visualization is the visualization of application program results. It involves complex image processing and animation of the output data produced by supercomputer simulations, satellites, and measuring devices used in astronomy, meteorology, and medicine. Visual programming is the specification of a computer program using graphics (icons). Program visualization is the use of graphics to enhance the understanding of a program that has already been written. The programs themselves are normally written in traditional languages. Program visualization normally relies on algorithm animation which illustrates the fundamental operations of the algorithm in the program, as compared to the program animation where the details of the code itself are displayed. Performance visualization seems to be a superset of program visualization because performance analysis should be based on the thorough understanding of the program behavior.

In this dissertation, we restrict our attention to performance visualization.

Performance Visualization

Graphical visualization aids human comprehension of complex phenomena and large volumes of data. The behavior of parallel programs on advanced architectures is often extremely complex, and monitoring the performance of such programs can generate vast quantities of data. So it seems natural to use visualization to gain insight into the behavior of the parallel programs so we can better understand them and improve their performance.

Graphical support for the visualization of programs and their run-time states and results gained momentum in recent years primarily because the falling cost of graphics-related hardware and software has made it feasible to use pictures as means of communicating with computers. The objective is to use high-resolution graphical displays to make the task of program development and testing easier.

Design, analysis, implementation, and maintenance of a program involve mental activities not only based on the appearance of a program but also observations of how the program works, why it works, how the components are put together, what effects they have on each other, and so on. To assist in the programming process, a tool that provides multiple views of a program and its execution states would be more effective than a tool that focuses only on the program text.

Standardization Effort For the Performance Visualization

Like many different areas of computer systems, there has been an effort to provide standards in the performance instrumentation and visualization. And as a result, standard work group was formed. Establishing standards has advantages of merging advanced technologies being developed independently and avoiding duplication of efforts by providing common foundations.

Malony and Nichols [52] indicates the performance instrumentation and visualization are fairly promising areas for standardization. Since most of visualization tools were implemented based on the X window systems, it would be desirable to have a standard set of X window based performance views which can be shared on different environments. As for the need for the standardization of performance visualization, Malony and Nichols [52] said:

> Although there are many possible alternatives to presenting performance data graphically, there can be an attempt to standardize on the methods used for graphics programming so that independently developed performance displays can be shared. ... However, it has been demonstrated clearly that color and graphics approach can be used effectively in presenting performance data. Thus, a standard graphics approach to performance visualization should allow basic performance displays to be constructed easily and shared, but should not be so simple that more exotic techniques cannot be explored.

Bargraph, meters, timelines, matrix, call graphs, kiviat, pie graph, led, contour plot, surface and scatter plot implemented on top of X window system are initial list of performance displays that the standardization group proposed (see Figure 2.3).

Performance Environment Architecture

Along with the standardization efforts, the working group proposed the architectural model of the HyperView environment as a design standard for the performance environment which integrates performance instrumentation, analysis and visualiza-

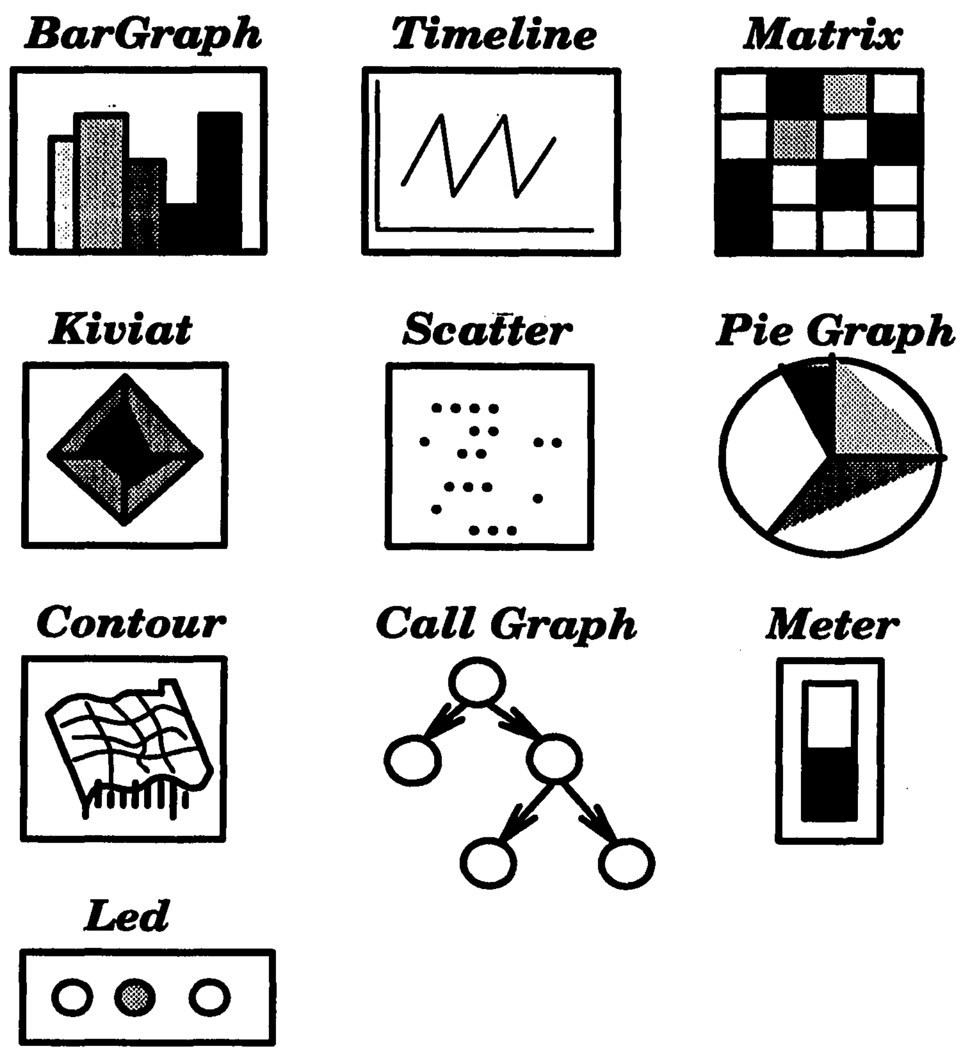


Figure 2.3: Proposed Performance Displays

tion. Figure 2.4 shows a proposed general purpose performance environment architecture.

The performance environment architecture supports a trace-driven, post-mortem performance analysis. The major components of it are Control, Filter, Strainer and Displays. Control is responsible for managing trace data and handling user's trace control inputs. Trace event dispatching to active filters and general configuration request are also performed by the Control. Filters are processing dispatched trace events to maintain internal performance-related information. There is a set of interfaces defined for filters so that new filters can be developed in modular fashion while encapsulating filter functionality. Diverse performance aspects are finally displayed through a set of displays which together defines the display capabilities of the environment. Each display provides user definable attributes of the display through resource interfaces. Modular development of displays are also supported by the environment architecture. User can define a subset of performance information of filters via strainers. In summary, the standard work group build a flexible and modular performance environment architecture by defining the standard interfaces between the filter, strainer and display.

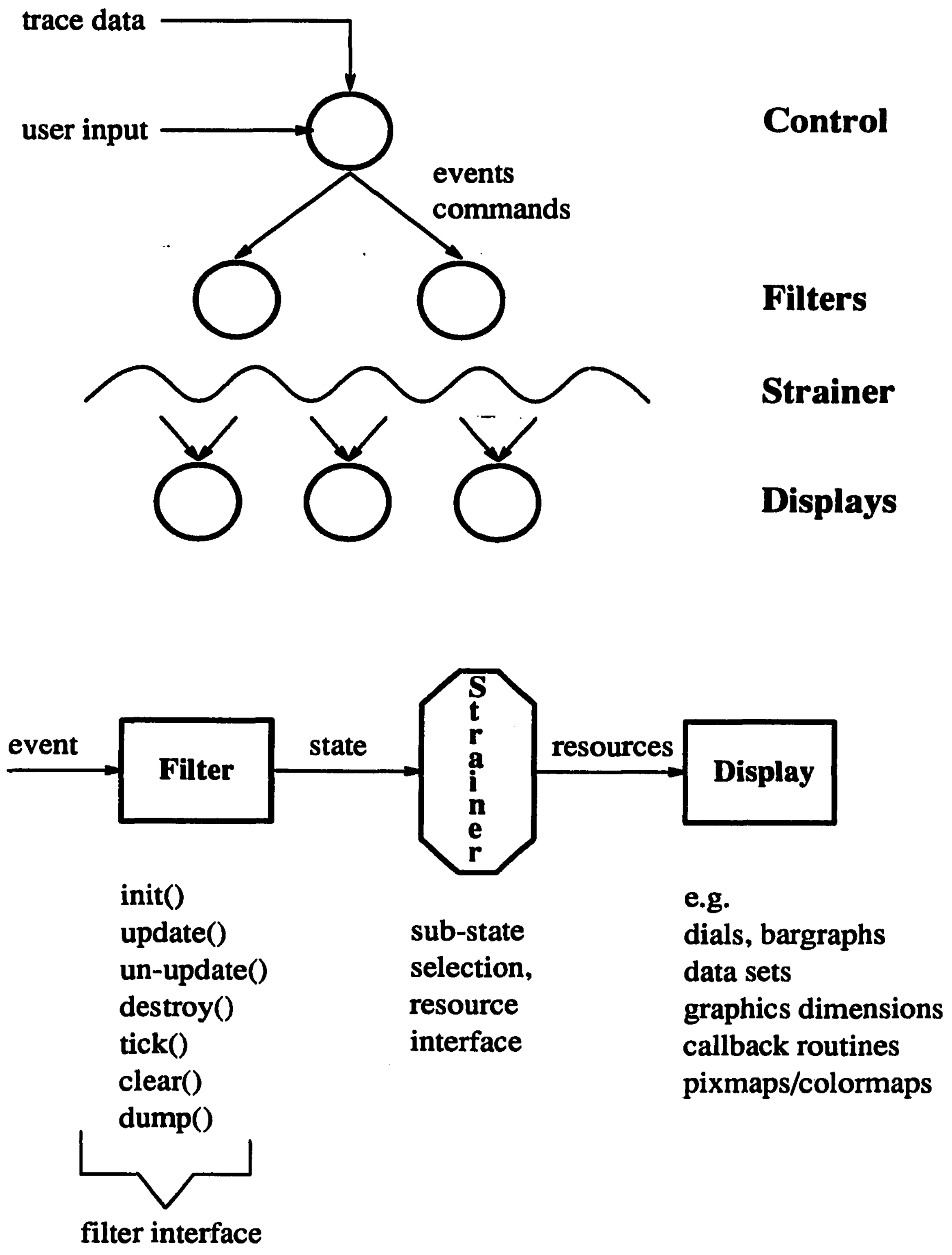


Figure 2.4: Performance Environment Architecture [52]

CHAPTER 3. RELATED WORK

Belvedere

Belvedere [22] is a pattern-oriented, trace-based, post-mortem debugger for message-passing multicomputers. It is one part of the Simple Simon Programming Environment which provides support for mutiprocessor simulation. Belvedere provides facilities for animation and manipulation of interprocess communication patterns resulting from both control and data flows. It treats the event traces as a relational database and allows users to select portions of the database for animation. Both the simulator generated primitive events and user-defined abstract events can be animated.

PIE

PIE [51] (Programming and Instrumentation Environment) is a software development environment for parallel processing that helps users to observe the execution behavior of application programs as well as operating system. It is designed for fast and correct performance debugging of parallel programs running on shared-memory multiprocessors. In addition, most of the support to parallelize the application is automatically generated by the PIE. Major components of the PIE includes the Modular Programming (MP) metalanguage, the program constructor, and the implementation assistant. 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. It is built on top of Mach operating system using X window system, but can be ported to other Unix-like operating systems.

Seeplex

Seeplex [10, 27] is a part of a larger tool called Triplex, a collection of software tools which aid the programmer in developing algorithms and monitoring executions on the NCUBE multicomputer. The tools address the problem of understanding the behavior of parallel programs in terms of both correctness and performance. Triplex consists of three components — Simplex, Seeplex, and Commplex. The Simplex is an operating system for the NCUBE. It measures and collects an enormous number of execution-related data. The Seeplex is a color graphics display program for viewing depictions of program execution collected by the Simplex in a large number of different ways. Simplex and Seeplex operate as tightly coupled programs to support real-time and offline debugging and performance monitoring. The Commplex is a communications package for communication with the NCUBE from Sun workstations.

Seeplex is extended from one of the first performance visualization tools called Seecube. It has an emphasis on the scalable monitoring, i.e, schemes which will work regardless of the number of processors or the sophistication of measurement desired.

HyperView

HyperView [32] is an interactive performance visualization environment system that integrates performance data analysis and visualization to help the performance analysts to browse through a trace system and application execution. The first generation HyperView provides a diversity of views reflecting architectural and system activity. One notable feature of it is the decoupling of data analysis and display components. This decoupling has several advantages — visualization independent of target machine, enhanced performance through distributed processing, and dynamic system reconfiguration without changing the interface between components when adding new data analyses and displays. Since, however, the first generation HyperView has a limitation of inextensibility to the display of application performance, the second generation HyperView was developed to address this problem. Flexibility and dynamic reconfigurability were primary design goals of the second generation HyperView.

User interface, an event preprocessor, a set of filters, a set of strainers, and a set of display views are components of the HyperView infrastructure. An trace event stream is converted into the standard format through the event preprocessor. Filters process converted trace events and maintain an internal event information for each node like message counts, volumes, size, source and destination node, node utilization, and program state. The internal event information can be displayed in a variety of ways using the semantically independent views. Each filter maintains a set of strainers to produce a view specific data representation from the internal filter state. User can configure and manage the filters, strainers, and views through the user interface when changing attributes of performance views or opening new views.

Design of HyperView was inspired by Seecube, and many views were borrowed from Seecube. While Seecube was built for the SunView window environment, HyperView's implementation was based on the X window system.

PICL and ParaGraph

Portable Instrumented Communication Library (PICL)

PICL [16] is an interface library that can be used to develop concurrent programs on several multicomputers. This package was developed at the Oak Ridge National Laboratory in 1990. PICL provides generic low-level communication primitives and high-level communication operations on a large platform of message-passing multicomputers. These operations are portable in the sense that communication routines can be specified independent of target machines. Users are provided with an efficient and uniform interface for programming a reasonably wide range of message-passing machines, building that interface on top of whatever tools are provided by the underlying operating system. Low-level communication and system interface routines provide a portable syntax for message-passing programs. The high level routines are designed to run on various network topologies so that user can take advantage of the physical interconnection network and algorithm characteristics.

Furthermore, the library provides an execution tracing facility that can be used to monitor the performance of parallel programs or to aid in performance debugging. Table 3.1 shows PICL routines for tracing.

The event tracing facility of PICL can be classified as a software instrumentation. The trace information is obtained by instrumenting the library without any modification of user code. A user can control the type and volume of trace data.

Table 3.1: PICL Routines for Tracing

traceenable:	Enable tracing and specify the name of the trace file.
tracehost:	Begin tracing on the host.
tracenode:	Begin tracing on a node processor.
tracelevel:	Specify the amount and type of trace information.
traceinfo:	Return the current **tracelevel** specification.
tracemark:	Generate a user-typed trace record.
tracemsg:	Write a line of text directly into the trace file.
traceflush:	Send trace information to the trace file *now.*
traceexit:	Stop tracing.

The trace information consists of a record of events (message sending or message receiving, etc.). A trace record is an integer set that specifies the event type, time stamp, processor number, message length, and other useful information. The record type is indicated in Table 3.2.

Considerable effort has been made to achieve clock synchronization and to minimize the effect on the running programs. As pointed out in Chapter 2, clock resolution and its accuracy is vitally important not to invalidate the presented information. PICL trys to synchronize the clock and adjust for potential clock drift, so the time stamps are as consistent and as meaningful as possible. PICL also minimizes the overhead of instrumentation by collecting trace data on the local memories while the program is running. After an instrumented program has finished execution, the trace data are transferred to the host. The added overhead is a function of the frequency and volume of communication traffic. In general, program perturbation is small enough that the behavior of the uninstrumented program is not changed fundamentally.

PICL is implemented on the assumption that interprocessor communication is

Table 3.2: Verbose and Compact Format Trace Record Type Identifiers.

compact format integer	verbose format keyword
1	`trace_start`
2	`open`
3	`load`
4	`send`
6	`recv`
7	`recv_blocking`
8	`recv_waking`
9	`message`
10	`sync`
11	`compstats`
12	`commstats`
13	`close`
14	`trace_level`
15	`trace_mark`
16	`trace_message`
17	`trace_stop`
18	`trace_flush`
19	`trace_exit`
20	`block_begin`
21	`block_end`

interrupt-driven and that messages can be routed between any pair of chosen PEs.

Currently at the Scalable computing laboratory in Ames Lab, research is under way for customizing tracing routines of the PICL on nCUBE 2 to further reduce instrumentation perturbation effects.

ParaGraph

ParaGraph [20] is a software tool that provides a detailed, dynamic, graphical animation of the behavior of message-passing parallel programs and graphical summaries of their performance.

ParaGraph displays the behavior and performance of real parallel programs running on real parallel computers to solve real problems. In effect, ParaGraph provides a visual replay of the events that actually occurred when a parallel program was run on a parallel machine.

To date, ParaGraph has been used only for post-processing trace files created during execution and saved for later study. But its design does not rule out the possibility that the data could arrive at the graphical workstation as the program executes.

However, there are major impediments to genuine real-time performance visualization. With the current generation of distributed-memory parallel architectures, it is difficult to extract performance data from the processors and send it to the outside world during execution without significantly perturbing the program being monitored. Also, the network bandwidth between the processors and the workstation and the drawing speed of the workstation are usually inadequate to handle the very high data-transmission rates that a real time display requires. Finally, humans would be hard pressed to digest a detailed graphical depiction unfolding in real time. One of ParaGraph's strengths is that it lets you replay the same execution trace data repeatedly.

ParaGraph adopts a dynamic approach whose conceptual basis is an algorithm animation. We see a trace file as a script to be played out, visually reenacting the original live action to provide insight into a program's dynamic behavior.

Distinguished features of ParaGraph are:

- **The number of displays it provides.** It is important to provide multiple views to users. ParaGraph provides a substantially greater variety of perspectives than other packages.

- **Its portability among architectures and displays.** Many previous packages for visualizing parallel algorithms have targeted a particular parallel architecture and/or have been on a proprietary graphical display system. ParaGraph is applicable to any parallel architecture having message passing as its programming paradigm, and ParaGraph itself is based on the X window system, which is widely available on workstations from many vendors.

- **The intuitive appeal and aesthetic quality of its displays.**

- **Its ease of use.** ParaGraph provides an interactive, graphical user interface and relies on PICL to provide requisite trace data without requiring the user to instrument explicitly the parallel program under study.

- **Its extensibility.** ParaGraph provides a mechanism for users to add new displays of their own design to facilitate incorporating special-purpose displays.

CHAPTER 4. PERFORMANCE METRICS

During the execution of a parallel program, multiple asynchronous processes interact and make internal state transitions as time progresses. Parallel program analysis, therefore, requires that performance metrics should at least reflect three dimensional spaces as shown n Figure 4.1 — ***process interactions***, ***process states***, and ***time***.

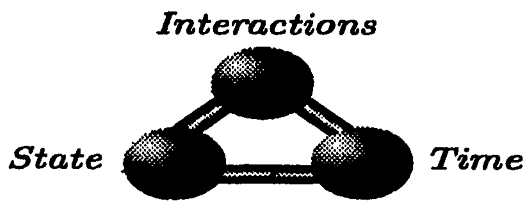


Figure 4.1: Three Dimensional Spaces for Performance Metrics

Other types of metrics that can be calculated from trace event information include counts, and ratios. A ratio may be a time rate, a density, a percent, or other relative comparison. Appropriate metrics for multicomputers include [41]:

- **Processor state** describes the current activity of a processor.
- **Operation count** or **Work** is the number of operations (calculations) performed locally by one or more tasks running on a processor.
- **Computation time** is the time spent being busy doing work, exclusive of idle time and communication time.

- **Computational power** is the average rate at which work is done, exclusive of any overhead.
- **Execution time** is the total amount of time a processor has spent executing a program, including computation and communication time.
- **Message volume** is the number or size of messages pending (i.e. sent but not yet received) by a processor.
- **Communication time** is the time spent by a processor in message passing, including overhead and transmit time.
- **Communication flow** is the average rate at which bytes are processed and transmitted at a processor.
- **Execution rate** is the average rate at which work is done over the duration of a time interval, including any overhead.
- **Percent computation time** is the percent of the total time that is spent doing work.
- **Percent communication time** is the percent of the total time that is spent doing message passing.
- **Communication Overhead** is a measure of the time spent communicating per unit of time spent doing work.
- **Concurrency**
- **Utilization**

CHAPTER 5. OBJECT-ORIENTED PROGRAMMING

This chapter introduces the basic concepts and terminology of object-oriented programming. Advantages and disadvantages of an object-oriented programming and concurreny issues are described as well.

Basic Concepts

Object-oriented design ensures encapsulation, inheritance, and polymorphism. It also supports development of reusable software. In object-oriented design, major building blocks for program construction are *objects.* An object is an encapsulated set of state data, together with a set of related functions (operations) that manipulate the shared state data. The key idea is that a collection of data and functions that normally operated on the data are closely related and should be treated as a single entity rather than as separate things [35]. In Smalltalk, functions defined by an object are often referred to as *methods*, and invoking a method is called sending a *message* to an object. In C++, methods are referred to as *member functions* and invoking a method is performed using a procedure-call interface. In general, the data of an object are hidden from all the other objects. Encapsulation is ensured by allowing the access and modification of the state data to be accomplished only through a set of publically accessible functions defined for the object. Public functions define an

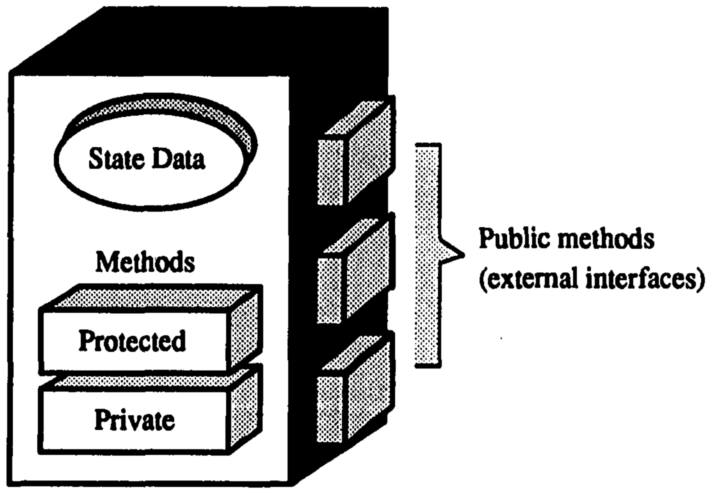


Figure 5.1: An Object Module

object's external interface, while private and protected functions define an object's behaviour (see Figure 5.1)

Objects provide a coarser granularity for program decomposition than is provided by using conventional functional decomposition. Writing programs in terms of objects and their interactions is more natural than to rely on artificially invented functions. Software designs in both functional decomposition and data decomposition usually lead programmers to come up with the solution structure radically different from the problem domains.

Many similar objects can be described by the same general description. The description of an object is called a *class* and it is basically an extension of abstract data type for similar objects defined through a generalization process. It is used as a template from which objects may be created. Every object is an instance of a class. Objects created from the same class share the structure and behaviour of the class. All instances of a class have the same number and types of state data, called *instance* *variables.* But the values of those instance variables are different among them. Some of an object's private variables are shared by all other instances of its class. These variables are called *class variables* and are part of a class. In C++, class variables are named as *static class members.* Class variables are useful when there is a need to maintain coherent information that must be available to all objects of its class.

Classes can form a hierarchy via *inheritance.* Inheritance is a powerful mechanism that facilitates code reusability and supports an incremental modification which is desirable is the software systems. A class derived from one or more *parent classes* or *superclasses* as a *subclass* inherits the attributes and behavioral characteristics of its parents. A subclass can modify the attributes and behavior of its parent class in one of the following ways — adding new instance variables, adding new methods not defined by the parent class, overriding the existing methods defined by the parent class, or adding new class variables. When a class has more than one parent classes, it uses *multiple inheritance*, otherwise it uses *single inheritance.* The potential for code sharing is greater in multiple inheritance, but the possibility of conflicts between multiple parent classes increases the complexity. There is also a derivation of the inheritance, called *delegation.* Delegation is a mechanism that permits an instance object to delegate responsibility for servicing an invocation request to another object. Unlike inheritance, delegation is class independent. Indivisual instance objects of the same class may have different objects servicing requests they are unable to service.

Another important feature of object-oriented languages is *polymorphism.* Polymorphism allows various types of objects to respond to the same message in different ways, without requiring the program to know the exact type of the object. Polymorphism provides a mechanism for building a general and extensible code. In C++, polymorphism is supported by *virtual* member functions.

A programming language is defined as being *object-based* if it supports objects as a language feature and *object-oriented* if it also supports the concept of inheritance and polymorphism [7]. C++ and Smalltalk are classified as object-oriented languages, whereas Ada and Modula-2 belong to object-based languages. Object-oriented languages can be considered either revolutionary or evolutionary, depending on the degree to which access to conventional programming techniques is retained [8].

> Pure object-oriented languages such as Smalltalk-80 represent the revolutionary approach and provide the advantage of conceptual simplicity. The programmer works in a computational environment that contains only objects, so the break with the past is clean and crisp.
>
> ... an evolutionary approach – adding object-oriented concepts on top of conventional languages. A number of these hybrid languages exist today, including Objective-C, OOPC, Flavors, and Clascal. These languages do not offer the conceptual consistency of Smalltalk-80, but they do have one considerable advantage: They can often be used for production programming, where pure languages like Smalltalk are usually unacceptable. ... Since they retain conventional languages as a substrate, efficiency can be outstanding.

Pros and Cons

Object-oriented languages have many advantages over traditional procedure-oriented languages. These advantages include:

- Increased reliability and decoupling of specification from implementation through information-hiding and data abstraction

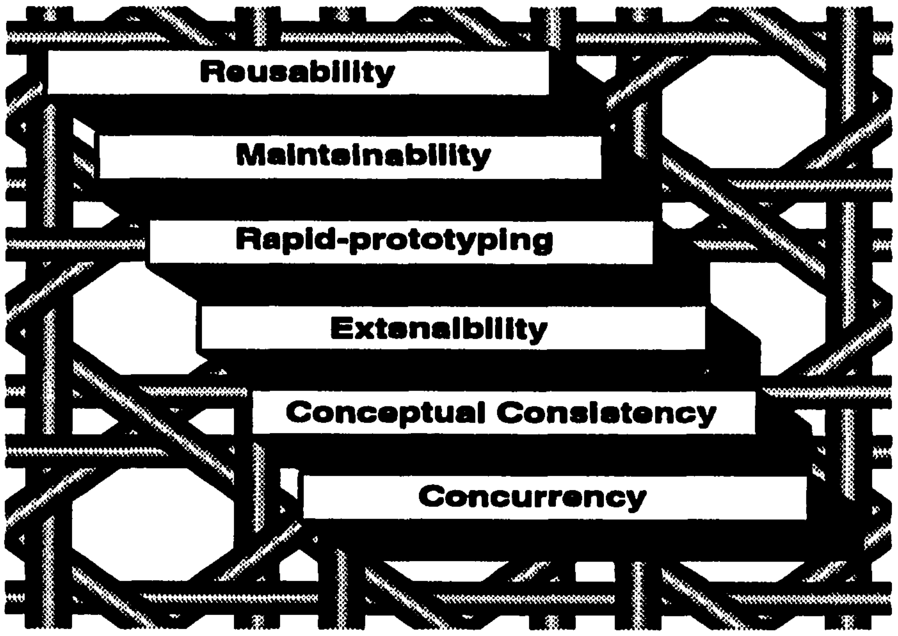


Figure 5.2: Benefits of the Object-Oriented Programming

- Flexibility of adding new classes of objects without having to modify existing code through dynamic binding
- Code reusability through inheritance combined with dynamic binding
- Reduced overall code and increased productivity

Object-oriented programming also provides major advantages in the production and maintenance of software: shorter development times, a high degree of code sharing, and malleability [35]. Moreover, a more natural representation of the real-world model can be realized in the code by using object-oriented programming. The benefits of an object-oriented programming are summarized in Figure 5.2.

On the other side of the coin, runtime cost of the dynamic binding mechanism is thought to be a major disadvantage by some. In general, however, object-oriented programming is considered to be a promising technology for constructing complex software systems for present and in the future.

Software ICs

Cox [9] introduced a concept of software ICs (integrated circuits). Software IC is a reusable software component and its concept came from a combination of aspects of subroutine libraries and Unix filters. The hardware IC chips revolutionalize the design of hardware systems due to their massive reusabilility and encapsulated operating functions. IC chips provide well-defined services on request without requiring to know the internal methods or data. When changes are necessary, modified IC chips can be used while inheriting most of the implementation without affecting interfaces or ICs not affected by the change. The software ICs are similar to the hardware counterparts. Object-oriented programming enables and encourages the use of software ICs. Through encapsulation, inheritance and dynamic binding features of object-oriented programming, the software IC concept is realizable and users can build a software system which satisfies evolving requirements.

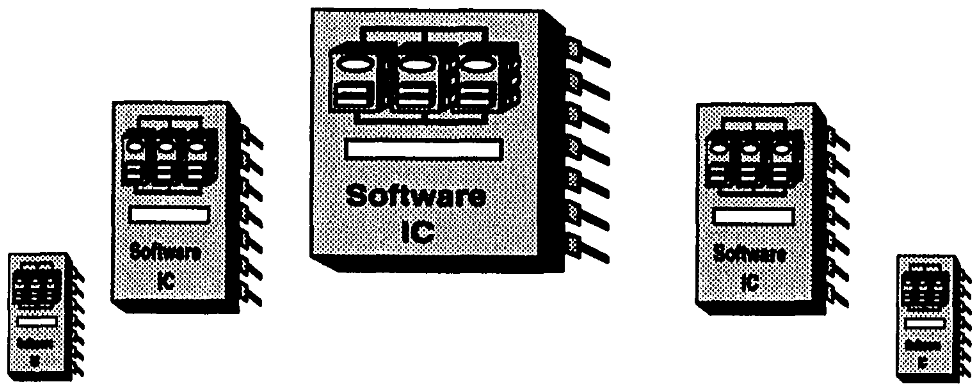


Figure 5.3: Software ICs

Concurrency

Concurrency involves having several, simulateous threads of computation within a system. The difference threads may communicate with each other using message passing or shared memory. Advantages of concurrent modeling includes:

- Improved modelling capabilities. In the real world, actions take place concurrently, realizing such interactions is more natural in a concurrent language.
- Concurrency allows the user interface, I/O processings, and local computation to take place independently.

Concurrency arises naturally from object-oriented design, rather than explicit consideration from the beginning. This happens because the analysis is done in terms of autonomous objects perhaps exchanging messages, and such objects are naturally concurrent. Autonomy implies concurrency. Consequently, one can design object systems on a uniprocessor and later convert it to multiprocessors object system with little or no change. But this does not guarantee a good parallel object-oriented design.

CHAPTER 6. DESIGN GOALS

This chapter describes the design goals of the proposed performance visualization environment and addresses the implementation contraints. As an effort to generate a general purpose and as yet efficient performance visualization tool, we combined the ideas of previous visualization environments described in Chapter 3. However, several of the ideas described in this paper are mainly based on results from the ParaGraph [20] and the HyperView [32]. We have also based our implementation on ParaGraph 2.0. Although most views of our current prototype object-oriented ParaGraph are borrowed from ParaGraph, the views and functions of ParaGraph are just a subset of those of ours. The main design goals are as follows:

- **Simplicity:** The overall program design should be clean and simple. This implies that each component object should be simple to perform only one well-defined major function. The interaction between objects should be accomplished purely by sending and replying messages, not by depending on the internal data structure of objects. This property provides an extra benefit of easy extension to actual parallelism.

- **Ease of Use:** Users of a visualization tool must feel comfortable in operating the tool. The tool should provide a consistent and unambiguous user interface which is highly interactive, mouse-, and menu-driven. If color coding scheme is used, then it must be consistent throughout the entire views so that users may not be confused.

- **Extensibility:** A visualization tool should provide a flexible mechanism for users to add new views of their own design to facilitate incorporating special-purpose views, (Even though the ParaGraph supports this mechanism, it is primitive and awkward in that users may end up with proliferation of ParaGraphs which differ only one view between them.) If a visualization tool supports only a predetermined set of performance aspects and a specific set of trace record formats, its functional lifetime will be limited. By defining standard interfaces for each component to encourage modular design, the visualization tool can be quite flexible to accommodate changing users' requirements. Extensibility is also desirable in that system evolution can be accomplished with little effort.

- **Reusability:** Isolating the semantics of performance view from its analysis module would greatly increase the possibility of module reuse. In this case, view modules can be reused. This idea can also be found in [39]. The module resue can also be obtained by isolating the semantics of trace data from its analysis module. In this case, analysis and/or reduction modules can be reused. The COOPG must be designed to be useful in analyzing the performance of different distributed memory concurrent systems with disparate trace record formats with little effort.

- **Efficiency:** The main purpose of using parallel computers is to reduce computation time as much as possible. It would be absurd if the analysis tool for the parallel programs becomes a bottleneck in the production cycle. Therefore efficiency should be considered throughout the design and implementation phase of the performance visualization environment. Poorly designed program with unproper abstraction may cause unacceptable efficiencies. Care should should be taken for a visualization environment to induce as little overhead as possible, preferably none at all.

- **Portability:** A visualization tool should be capable of running on diverse platform of workstations. Since the object-oriented performance visualization environment is written in C++ with the graphical displays based on X window library, it is highly portable. Although the current prototype environment does not support graphics system other than the X window system, the design principle does not preclude the possibility of implementing the tool on the other graphics systems to gain better response time. This is possible because all the graphics related functions are encapsulated from the rest of the tool. Therefore as far as the external interfaces are consistent, X window graphics functions can be safely replaced with a customized graphics functions. The other alternative is to develop an abstract window system superclass so that all the other window systems can be derived from that superclass.

In designing the performance visualization environment, the following requirements and restrictions must also be considered.

- Drawing of views must be synchronized to reflect the relationship among views representing various aspects of performance. Each snapshot of the display must deliver the current behaviour of the program seen by different angles of analysis.

- Trace records must be processed sequentially. Except some summary statistics views, most of the views require time lined-anaysis.
- In concurrent implementation, unnecessary synchronization overhead must be avoided. Granularity of concurrency must be carefully selected from the outset of the design process.

CHAPTER 7. THE VISUALIZATION ENVIRONMENT

We have designed and implemented a prototype performance visualization environment, called Concurrent Object-Oriented ParaGraph (COOPG), for the distributed memory concurrent memory compūters with the design goals in the previous chapter in mind. The COOPG is an evolutionary extension of our first generation performance visualization environment, Object-Oriented ParaGraph (OOPG) [25], for better efficiency and extensibility. This chapter describes the design and implementation of both OOPG and COOPG which are trace-driven, post-mortem performance visualization environments.

The Object Oriented ParaGraph (OOPG)

The Object-Oriented ParaGraph (OOPG) was a stepping-stone implementation for our ongoing research work directed toward developing concurrent object-oriented ParaGraph. The OOPG consists of five major components as shown in Figure 7.1 — *Trace Manager, Filter Manager, Controller, Filters* and *Views.*

Each component in Figure 7.1 is implemented as a self-contained and autonomous object which accomplishes a well-defined function with few external dependencies. Trace Manager, Filter Manager, and Controller compose a backbone architecture of the OOPG. Filters and Views are components which can be freely plugged into the

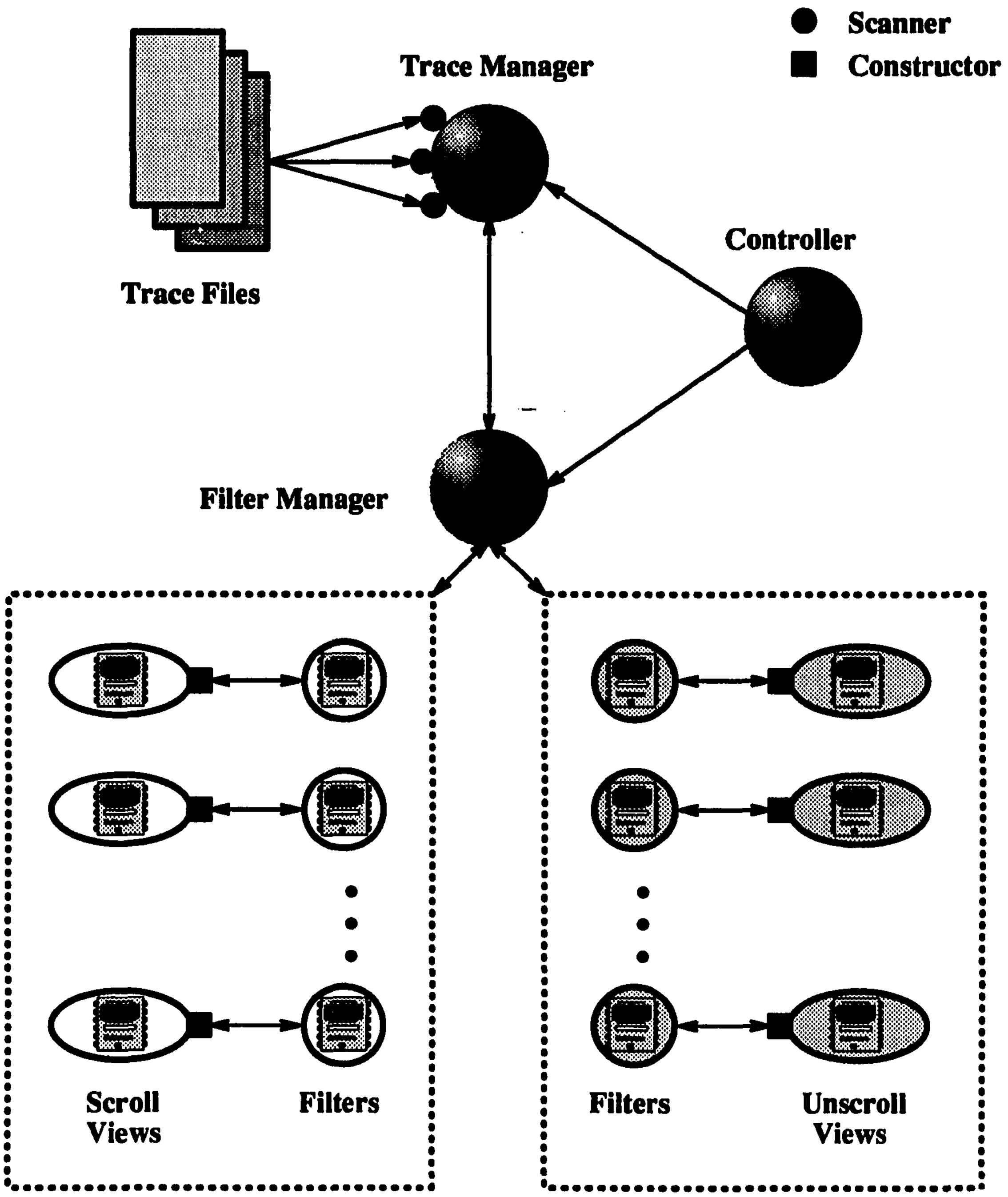


Figure 7.1: Infrastructure of the Object-Oriented ParaGraph

backbone architecture if necessary. Because Filters and Views are treated like integrated circuit chips which can be plugged into diverse Program Circuit Boards, they are referred to *Software-ICs* in Cox's [9] terminology. The overall organization of the OOPG thus makes it easy for users to extend it for their own specific performance perspectives with little programming effort. In this section, we will describe each component in turn.

Trace Manager

Trace Manager maintains a database of tracefiles and provides a graphical interface to the database so that users can select tracefiles with ease, as illustrated in Figure 7.2.

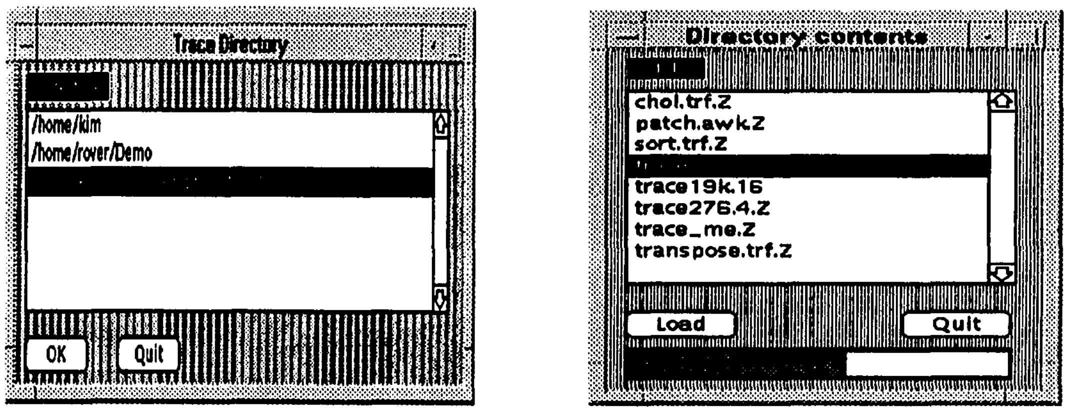


Figure 7.2: Graphical Interface for Tracefiles

It can handle both ASCII tracefiles and compressed binary tracefiles. Compressed binary files are uncompressed on-the-fly at run time. The Trace Manager also provides a portable mechanism for passing trace records to a set of filters. To be extensible and reusable across disparate sets of trace record types, the Trace Manager should be flexible. Instead of developing portable trace record formats, which is very unlikely, we decided to supply user definable mapping facilities, called *scanners*. Through scanners, disparate record formats can be mapped to our standard trace record formats at run time without affecting other components.

In the prototype implementation of the OOPG, we used trace record formats of the PICL [20] as our standard. In OOPG, each trace record should be associated with at least one of the trace record classes. Figure 7.3 shows the hierarchy of trace record classes.

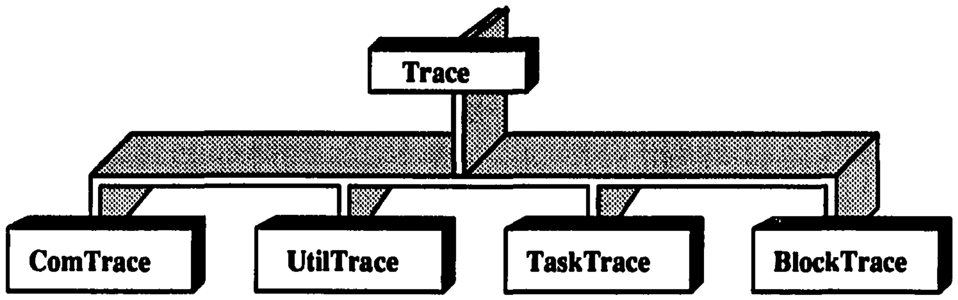


Figure 7.3: Hierarchy of Trace Record Classes

At the root of the hierarchy is a Trace class. Four other classes are derived from the Trace class after investigating how each of PICL's trace records is consumed in ParaGraph. Other classes can be easily derived from the extant hierarchy whenever necessary. Below are the specifications of Trace class and ComTrace class.

```
class Trace {
protected:
        int     type;                   // Trace Type
        int     sec;                            // Timestamp
        int     microSec;                       // Timestamp
        int     node;                           // Current Node
public:
        // Scanner Pointer
        friend  Trace   *defaultScan(char line[]);
};

class ComTrace : public Trace {
protected:
        int     sdNode;         // Source or Destination node
        int     msgType;        // Message Type
        int     msgLength;      // Message Length
public:
        // Scanner Pointer
        friend  Trace   *comScan(char line[]);
};
```

In the prototype implementation of our OOPG, we used a set of trace record types defined in PICL. Currently twenty three trace record types are defined in PICL. In OOPG, each trace record should be mapped to at least one of trace record classes. Figure 7.3 shows a hierarchy of trace record classes. At the root of the hierarchy is a Trace class. Four other classes are derived from the Trace class after investigating how each of PICL's trace records is consumed in ParaGraph. Other classes defining different trace record formats can be easily derived from the extant hierarchy. Below are the specifications of Trace class and ComTrace class.

Associated with each trace record class is a *scanner* which instantiates the class members from a given trace record. Scanners are user definable functions. The OOPG provides default stub functions for scanners using standard trace formats. Users need to modify only the scanners when disparate trace formats are required. Scanners should be registered before use through a method defined for the Trace

Manager as follow:

```
TRM.registerTracetype(RECV, comScan);
```

where TRM is a Trace Manager, RECV is a trace record type defined in PICL, and comScan is a scanner to be associated with RECV. The Trace Manager maintains a table of trace record types and corresponding scanners. For a given trace record type, it is allowed to register more than one scanners. For example, PICL allows BLOCK_BEGIN and BLOCK_END record types to be used for user's specific purposes. While ParaGraph uses those types for drawing task-oriented views, OOPG uses them for both task-oriented and speed-related views. Once scanners are registered, Trace Manager invokes appropriate scanner (or scanners) based on the trace type of a record and delivers created trace record objects to a set of filters through a Filter Manager. The operation of the Trace Manager is controlled via the Controller.

Filter Manager

Filter Manager maintains a list of filters and has a responsibility of requesting trace record objects to the Trace Manager and broadcasting them to a set of filters based on the operation selected by a user. Filters either process or discard the trace record objects. Filter Manager maintains two sets of filters depending on the scrolling capability of the views bounded to the filters. This division is simply to facilitate the control of the activation sequence for filters in a uniprocessor environment while allowing users to add their own filters with ease. Filters with scrolling views are activated before those with unscrolling views.

Controller

Users can interactively operate the OOPG through the Controller using mouse and keyboard. The Controller provides a hierarchy of menu systems as shown in Figure 7.16 and Figure 7.17. We developed an experimental class library (see Figure 7.8), called libPG.a, to facilitate the creation of windows, buttons, menus, menuitems, panels on top of X11 window system. For a fast prototype implementation, we borrowed menus from the ParaGraph. Since, however, each menu is constructed in an object-oriented design, creation of menus is more flexible and easier in OOPG. Menu systems with elegant and fancy look can be easily built separately from the rest of the OOPG components. This becomes possible because the menu system does not recognize the semantics of the menu items it displays.

Associated with each menu item, whether it be a button item or a text entry item, is an event handler which processes events and, if necessary, activates corresponding callback routines which are known to the Controller. Interpretation of those events is then delegated to the Controller. Figure 7.4 illustrates the event handling mechanism incorporated into the window objects in the class hierarchy of Figure 7.8.

Filters

Filters are self-contained objects which are responsible for data analysis, transformation, reduction and data presentation through views. Since filters are generally independent of each other, they can be freely added and/or removed without affecting the operations of others.

Each filter has a view displaying a particular performance aspect. A view is bounded to a filter through a registration method defined for a filter class and vice

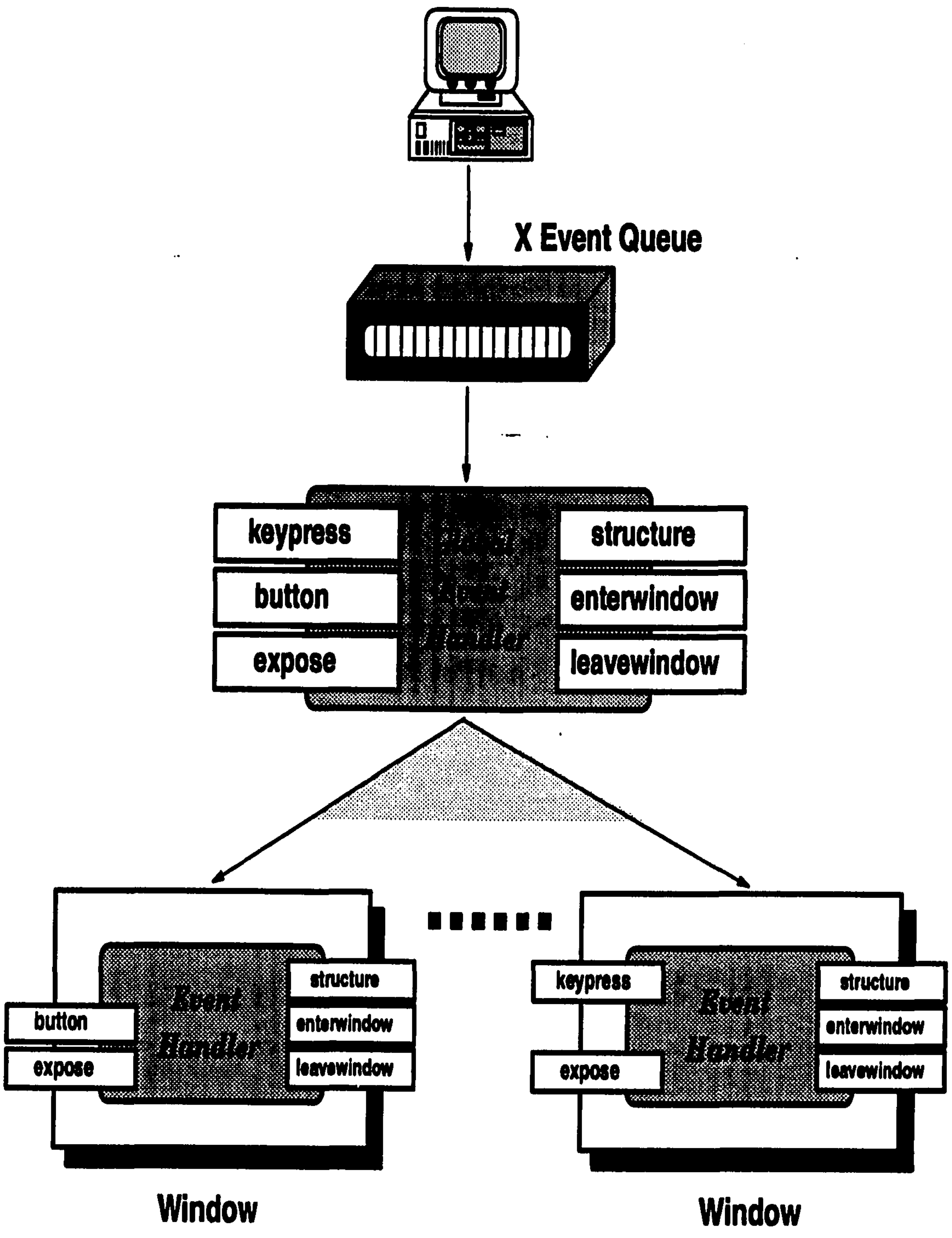


Figure 7.4: Event Handling for Window Objects

versa. The reason a filter should be registered to a view is that the view could be an active agent interacting with users and, if necessary, controlling the behavior of the filters they are bounded to. All filters should also be registered to the Filter Manager before use. Aside from the scrolling capability of the bounded views, filters are further classified into two groups: *transient* and *permanent* filters. Transient filters are normally simple and memoryless in that their operation is dependent purely on the current event like Figure 7.7 (a). However, permanent filters must keep track of previous event sequences for rather a complex analysis like Figure 7.7 (b). For reasons of performance, transient filters accept trace record objects only when the bounded views are displayed on the screen.

To be extremely general, filters must be totally independent of each other. However, an ultimate independency may cause an unacceptable delay because a lot of works might be duplicated among filters to obtain the same information. As a solution to this problem, we decide to create a hierarchy of filter classes, as shown in Figure 7.5, so that information sharing among filters can be performed naturally without causing delay by forcing filters to be derived from the extant hierarchy.

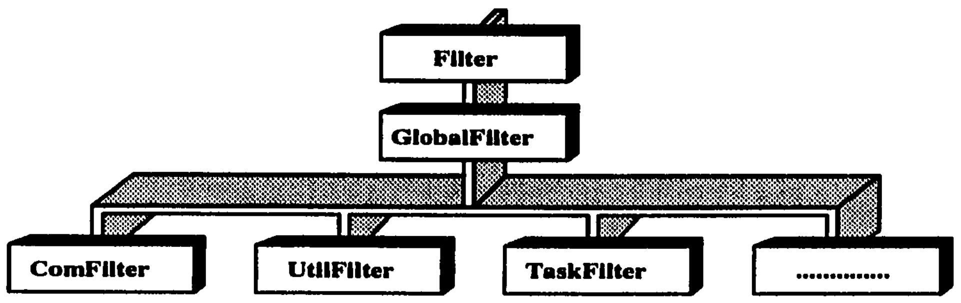


Figure 7.5: Class Hierarchy for Filters

Immediate subclasses of the GlobalFilter in fact extract only relevant information to their classes from trace record objects. The extracted information then becomes syntactically independent from even the standard trace formats. For example, ComFilter object generates information such as message volumes per node, queue lengths per link, message counts and etc. Since analysis filters derived from the ComFilter class only see these information, it is possible to develop independent reusable filters for performance aspects under consideration.

Views

A view is a window on the display which is responsible for drawing a particular aspect of performance. Originally, a view itself is not semantically tightly coupled with the displaying performance aspect. This means that each view is independent of the rest of the OOPG components, and only provides a set of configuration parameters and methods. We call this view as an *abstract* or *template* view. Configuration parameters are used to define a new view instance (*concrete view*), while the view methods define permissible operations to manipulate the view.

An abstract view become a concrete view after configuration parameters are assigned through a *constructor.* A concrete view is then tightly coupled with the displaying performance aspect. Concrete views can be created either directly from a corresponding abstract view (as shown in Figure 7.6), or indirectly from derived classes of an abstract class. This mechanism facilitates the creation of different views with similar characteristics and behavior from the same abstract views. Once created, concrete views should be registered to filters.

Figure 7.7 (a) is an example of a concrete view created from an abstract view,

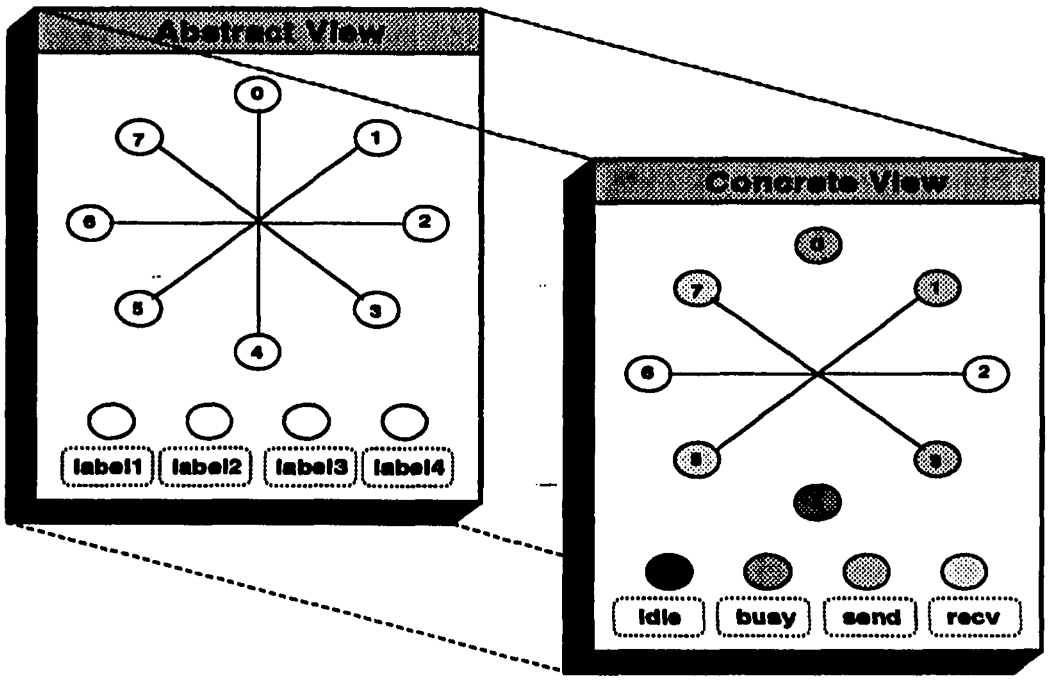
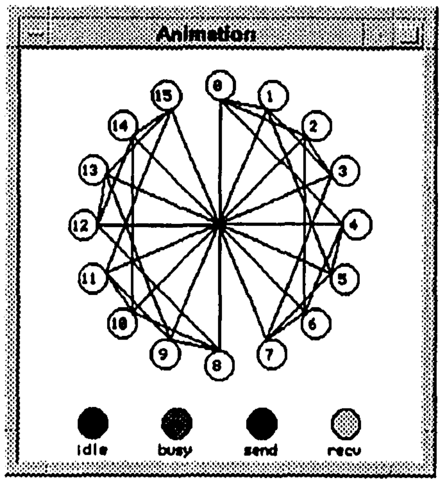
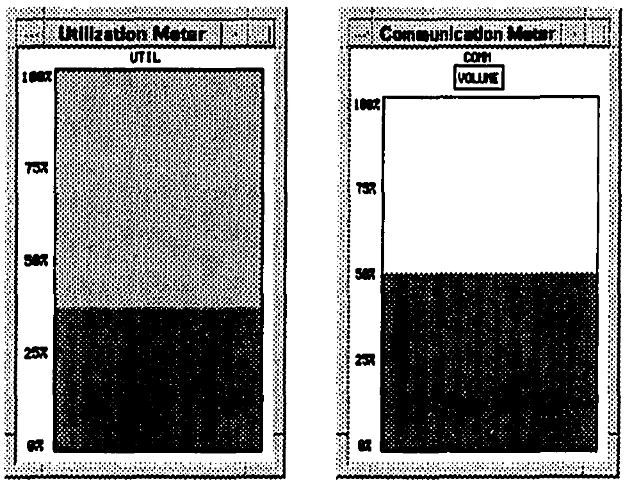


Figure 7.6: Creation of a Concrete View from an Abstract View





(a) AnimView instance

(b) MeterView instances

Figure 7.7: Concrete View Instances

called *AnimView*, to animate the message sending and receiving activities among processors. A C++ class definition for the configuration parameters and methods for the *AnimView* is:

```
class AnimView : public View {
    ...
    gc_t    stateNode[MAX_STATE];
    char    *legend[MAX_STATE];
public:
    void    setState(int state, gc_t gc, char *label);
    void    chgState(int node,int state);
    void    connect(int from, int to);
    void    disconnect(int from, int to);
    ...
};
```

Figure 7.7 (b) shows two different views derived indirectly from a common abstract view, called *MeterView.*

In our experimental class library, libPG.a, a set of abstract view classes are also defined together with the classes described earlier in Controller section. We examined many extant visualization tools to find out useful abstract classes to further promote the creation of new concrete views. A subset of the class hierarchy defined in the libPG.a is shown in Figure 7.8.

The Concurrent Object-Oriented ParaGraph (COOPG)

The implementation of the Concurrent Object-Oriented ParaGraph (COOPG) is evolved from the architecture of the Object-Oriented ParaGraph (OOPG). The main focuses of the COOPG over the OOPG is to achieve better efficiency, extensibility and reusability.

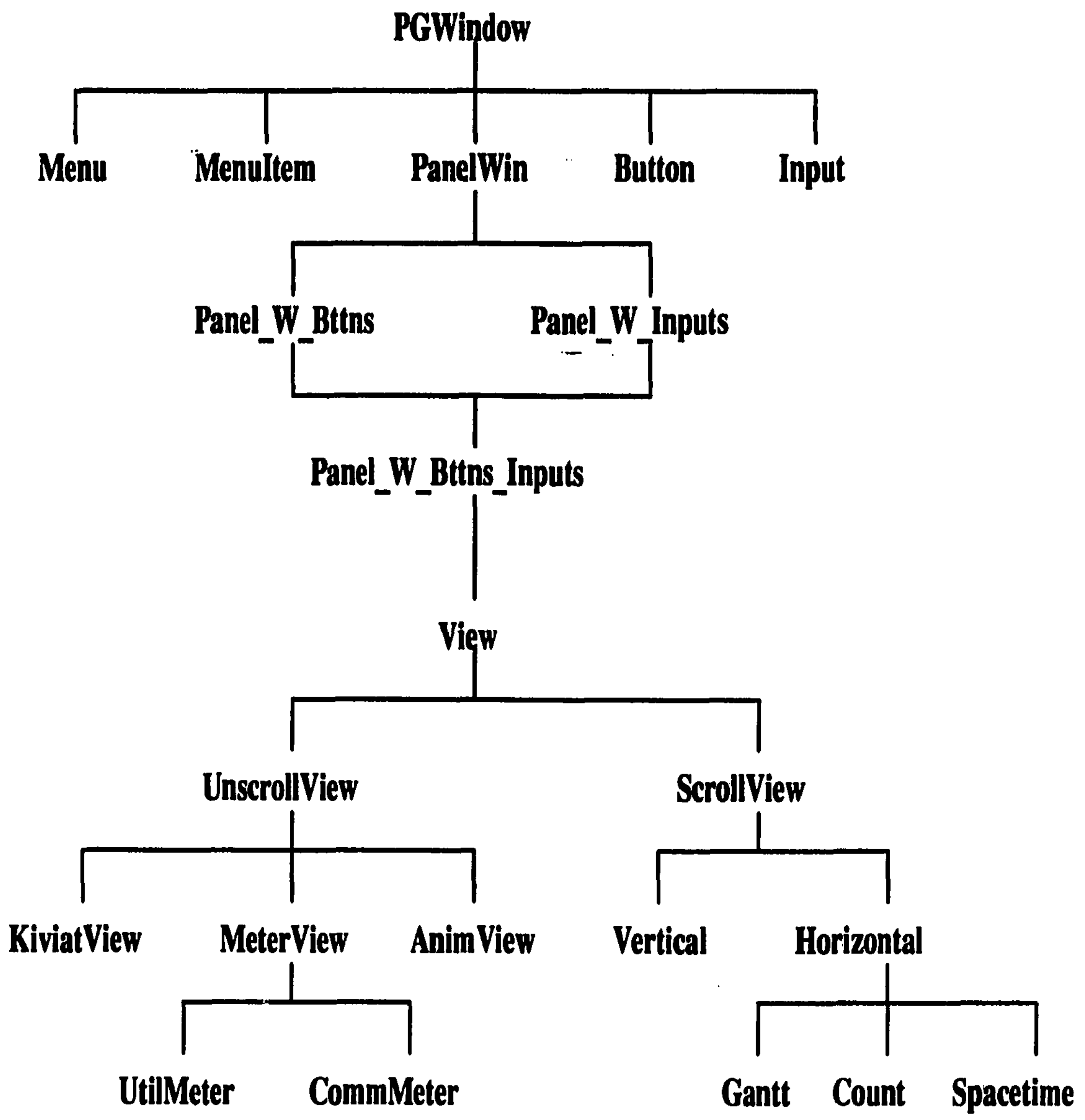


Figure 7.8: Class Hierarchy in libPG.a

Concurrency Models

As illustrated in Figure 7.9, the mapping of objects to processes can be realized in several ways. At one extreme, designers can assign one object per process. In this case, each object is an active object with its own computation and communication capabilities with other active objects. This framework provides a maximum concurrency and has an advantage of conceptual cleanness because the logical boundary and the physical boundary of an object does well coincide. However, it may suffer from a poor efficiency unless active objects are relatively independent because of the heavy synchronization overhead. At the other extreme, all objects in the system can be assigned to one process. In this framework, all the objects are treated as passive objects, and coordination of the objects are handled by a super agent.

In reality, most system fall in between, as collections of relatively coarse-grained processes, each housing a relatively large number of passive objects. The resulting architecture may be designed by clustering objects within processes. In this framework, each process has the same overall structure and must provide explicit interfaces allowing communication among objects, as well as mechanisms (especially *proxies*) that ship remote requests to other clusters [5]. Clustering can be done by considering performance (e.g. clustering objects with shared resources and clustering objects that heavily intercommunicate) or by emphasizing semantical closeness (e.g. subsystems).

In the implementation of our COOPG, a variation of the last scheme (see Figure 7.10) is adopted to compromise efficiency and conceptual cleanness. After investigating the operations and relationship of each object in the OOPG, objects are grouped together to form clusters based on their semantical closeness while considering the efficiency. However, all objects in each cluster are not passive ones. Each

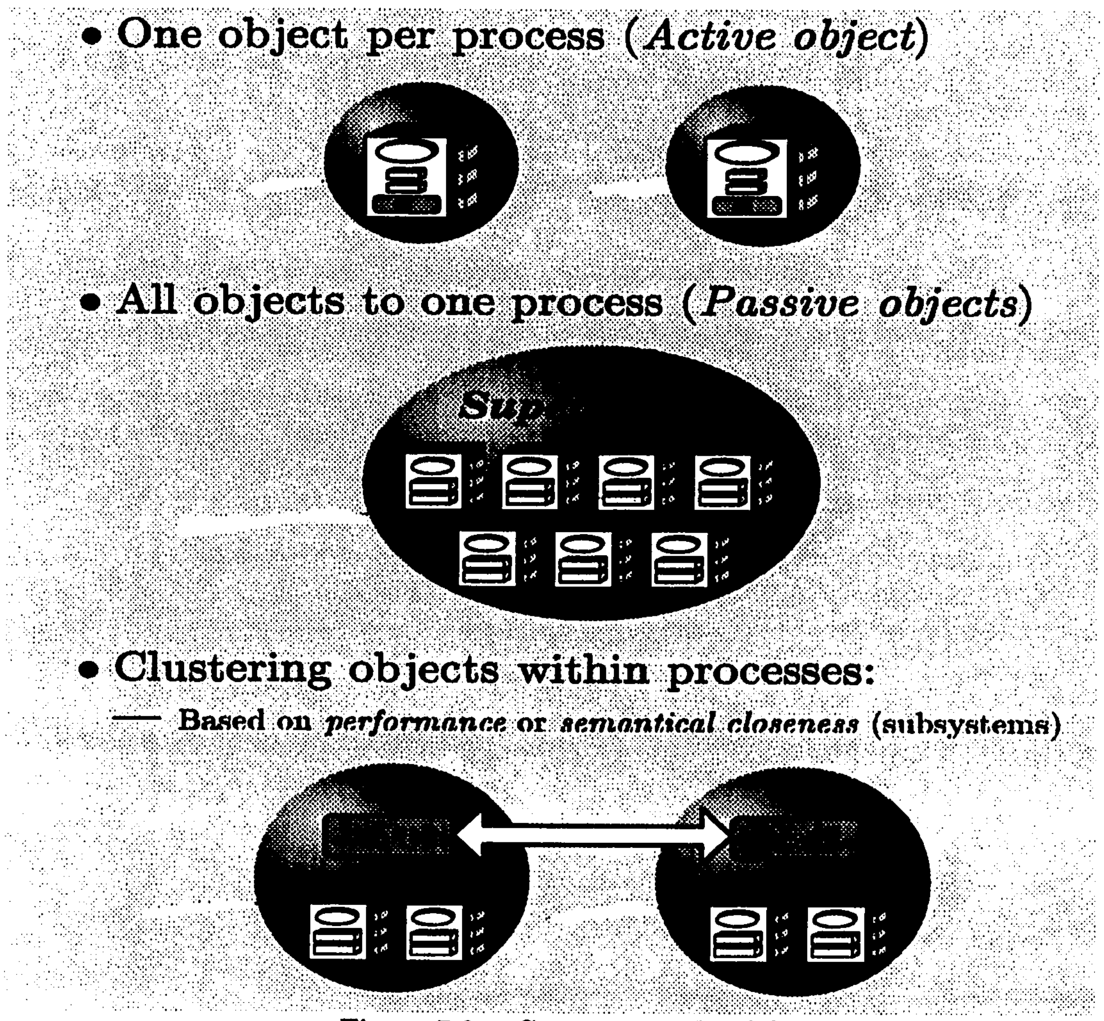

• One object per process (*Active object*)
• All objects to one process (*Passive objects*)
Sup
• Clustering objects within processes:
— Based on *performance* or *semantical closeness* (subsystems)

Figure 7.9: Concurrency Models

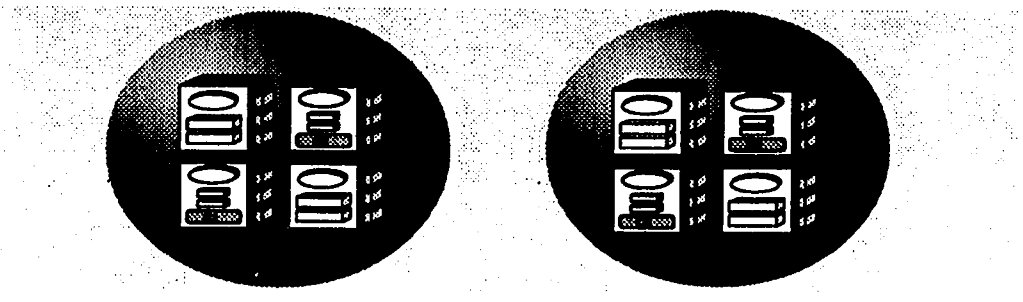


Figure 7.10: Concurrency Model in the COOPG

cluster contains a set of active objects and passive objects. An active objects are implemented via threads.

An Active Object: Task

A thread, called a *task*, has its own locus of control, and a computation to perform, and its own private data. A task can communicate by explicit sharing of data, or by messages. Basically, a task is an abstraction of an *activity*. Facilities for multi-thread computation can be provided by a language itself like Concurrent-Pascal and Mesa, or special run-time support systems and library functions can be used to augment the languages without those facilities as in [53, 33].

In order to provide the multi-thread facilities, as a library implementation, to the C++ language, we developed a C++ wrapper around the POSIX thread packages.[1]

The class specification of the base class *Task* is defined as:

[1]POSIX is an proposed IEEE standard for portable operating systems for open systems. Especially, POSIX 1003.4a is a threads extension, called pthreads, which describes the interface for light-weight threads. Light-weight threads, sometimes called tasks, are independent threads of control within a heavy-weight process that share global address spaces. The cost of context switches between threads is cheaper than the cost of context switches between processes because the context of a thread is smaller than that of a process. The Pthread standard provides a uniform base for multiprocessor shared-memory applications, real-time system environments, and a cheap model for multi-threaded programs on a single processor.

Based on the Pthreads standard, a library implementation of Pthreads package on the Sun SPARC station has been developed at Florida State University [33]. The main purpose of Pthreads package was to implement Ada tasks, but it can express parallelism within applications at the level of programming languages as well. The main features of the Pthreads includes task management (creation, join and destroy), preemptability, fast context switches, small critical sections, support of synchronization and signal handling, and few operating system calls. A language-independent interface is also provided.

```
class Task {

  public:

    virtual     ~Task();
    pthread_t   Pthread_getid() const { return pthreadID; }
    void        Pthread_yield (any_t arg);
    void        Pthread_exit  (any_t status);
    void        Pthread_detach ();

    static      int  Pthread_join   (pthread_t thread, any_t *__status);
    virtual     void Body() = 0;
    ...

  protected:

    Task()      { Pthread_create(); }
    static      void    Execute(Task *);

  private:

    int         Pthread_create ();
    pthread_t   pthreadID;
};
```

Since the constructor of the class *Task* is a *protected* member, users cannot create a thread directly. Instead, all classes which provides the abstraction of threads must be derived from the *Task* base class.

```
    class ActiveObject : public Taskl  { ... };
```

The base class provides the definitions of the methods that must at least be provided to the deriving classes.

The *Body* is a pure virtual function which must be defined by the derived class and it represents the controlling code for each object, i.e. the scope within which the controlling thread will execute.

A task can be in one of the following states:

Running The task is currently executing instructions.

Idle The task is waiting for some condition. Currently, the task is not executing, but it can be transferred to *Running* state if wait-for condition occurs.

Terminated The task has completed its computation and cannot be resumed. If detached, any memory associated with it may be recovered once the thread terminates.

Initialization The header file *Task.h* includes *pthreads.h* which defines the Pthreads interface, and contains the definition of the class *Task.* All modules using threads must include this file. *::pthread_init* initialize the Pthreads routines. A program using threads must explicitly call *::pthread_init* before using any of the other Pthreads functions.

Threads Control When a C++ program starts, it initiates a single thread of control, the *main* thread executing *main* function. New threads are created at the point where instances of the derived classes of the *Task* class are defined. Once new threads are created, they continue to execute in parallel with other threads.

A thread terminates when it returns from the *Body* it was executing. Unless *Pthread_exit* is explicitly called, the terminated thread is in limbo state.

Pthread_join forces the calling thread to wait for the specified thread to terminate. The return value of the specified thread is provided in *__status.*

A calling thread is detached by *Pthread_detach.* This operation indicates that the calling thread will never be joined. Once a thread is detached, any memory associated with it may be recovered once the thread terminates.

Pthread_yield causes the current thread to suspend execution and requeue itself at the tail of the its priority level in the ready queue. The argument is always NULL. This function is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor.

Pthread_getID returns the thread identifier of the calling thread. The thread identifier uniquely identifies the thread.

Synchronization Mutual exclusion and synchronization primitives, called *mutexes* and *condition variables*, are not encapsulated in the *Task* class definition. User can directly access those primitives supported by the Pthreads library using the "::" notation which allows access to global scope identifiers.

mutexes are used whenever necessary to prevent corruption of shared data by enforcing mutually exclusive access to them. Condition variables is a boolean function of the shared data that the mutex protects. They are used when a thread need to wait for some condition to occur. There are two extremes in using mutexes: one for protecting all shared data with one mutex, and the other for protecting every byte of the shared data with that many mutexes. Fine granularity normally increases the degree of possible parallelism, but it can lead to unacceptable overhead due to excessive locking and unlocking of mutexes.

Mutexes can be initialized by *::pthread_mutex_init* function and destroyed by *::pthread_mutex_destroy* function. Mutexes shared across processes are not currently supported by the Pthreads library.

The *::pthread_mutex_lock* attempts to lock a mutex and blocks until it succeeds.

A mutex can be locked by only one thread at a time. Any subsequent attempt to lock the same mutex by other threads will make them block until the mutex is unlocked. The *::pthread_mutex_trylock* is the same as the *::pthread_mutex_lock* except that when locking is failed, it returns -1 and set the errno to EBUSY immediately rather than blocking the calling thread. If locking succeeds, it returns 0. A locked mutex can be unlocked by *::pthread_mutex_unlock*. Then a thread at the head of the waiting queue may resume execution and lock the mutex.

A thread issues *::pthread_cond_signal* when a condition associated with the condition variable becomes true so that a thread at the head of the wait queue may be woke up. *::pthread_cond_broadcast* is the same as the *::pthread_cond_signal* except that it wakes up all the waiting threads, not just one. When no threads are currently waiting, then nothing happens. This means that signal is memoryless.

The *pthread_cond_wait* suspend the current thread on the condition variable associated with a mutex until signaled by either of the above signaling functions. Since there is no guarantee that the condition is true when a return from a conditional wait, a global variable should be checked to determine the condition is true. A mutex must be locked before calling a conditional wait primitive. When signaled, the thread first automatically wakes up and reaquires the mutex, and return from the conditional wait. The proper usage of these primitives should be as follows:

```
Thread 1:

...
::pthread_mutex_lock(&mutex);
while (!ready)
    ::pthread_cond_wait(&cond,&mutex);
::pthread_mutex_unlock(&mutex);
```

```
...

-------------------------------------------------------------

Thread 2:

...
::pthread_mutex_lock(&mutex);
ready = TRUE;
::pthread_mutex_unlock(&mutex);
::pthread_cond_signal(&cond);
...
```

The *Task.h* header file defines some useful macros to allow users to write more concise and error-free programs.

```
#define SYNC_VARS(x) \
                pthread_mutex_t         x##Lock; \
                pthread_cond_t          x##Cond; \
                int                     x##Ready;

#define EXTERN_SYNC_VARS(x) \
        extern  pthread_mutex_t         x##Lock; \
        extern  pthread_cond_t          x##Cond; \
        extern  int                     x##Ready;

#define WAIT(x) \
                pthread_mutex_lock(&x##Lock);\
                while (!x##Ready)\
                        pthread_cond_wait(&x##Cond,&x##Lock);\
                pthread_mutex_unlock(&x##Lock);\
                x##Ready = 0;

#define SIGNAL(x) \
                pthread_mutex_lock(&x##Lock);\
                x##Ready = 1;\
                pthread_mutex_unlock(&x##Lock);\
                pthread_cond_signal(&x##Cond);
```

Software Design

The Concurrent Object-Oriented ParaGraph (COOPG) consists of three major component clusters — *Trace* cluster, *Filter* cluster and *View* cluster. The software architecture of the COOPG is shown in Figure 7.11. Clusters are normal heavy-weight processes working independently and collectively to construct an extensible and efficient performance visualization environment. In current implementation, communication among clusters are achieved by either a shared memory or message queues. In Figure 7.11 active objects (tasks) are drawn in white ovals, and passive objects are shown in white rectangles. Small dark gray circles represent *proxies* to enable remote procedure call semantics between filters and views, and they will be explained in detail later. Solid arrows indicate control flows (commands and status information) and dashed arrows denote data flows (trace events and graphics requests).

Trace Cluster The Trace cluster is responsible for maintaining a database of tracefiles. It consists of *Trace Manager* and *Writer* tasks. Most of the functions and features of the Trace Manager in the Object-Oriented ParaGraph are retained in the Trace Cluster. Therefore we only describes the differences here.

In the OOPG, trace events were retrieved from the database of trace files and passed to the Filter Manager one at a time on request. But in the COOPG, trace events are written to the Double Buffer in the shared memory at full speed by the *Writer* once a trace file name is provided to the *Trace Manager.* Double buffering allows overlapped execution of input processing and internal computations. The operation of the *Writer* is controlled by the *Trace Manager.* Even if the current implementation does not support multiple *Writer* tasks, the Trace Cluster can be

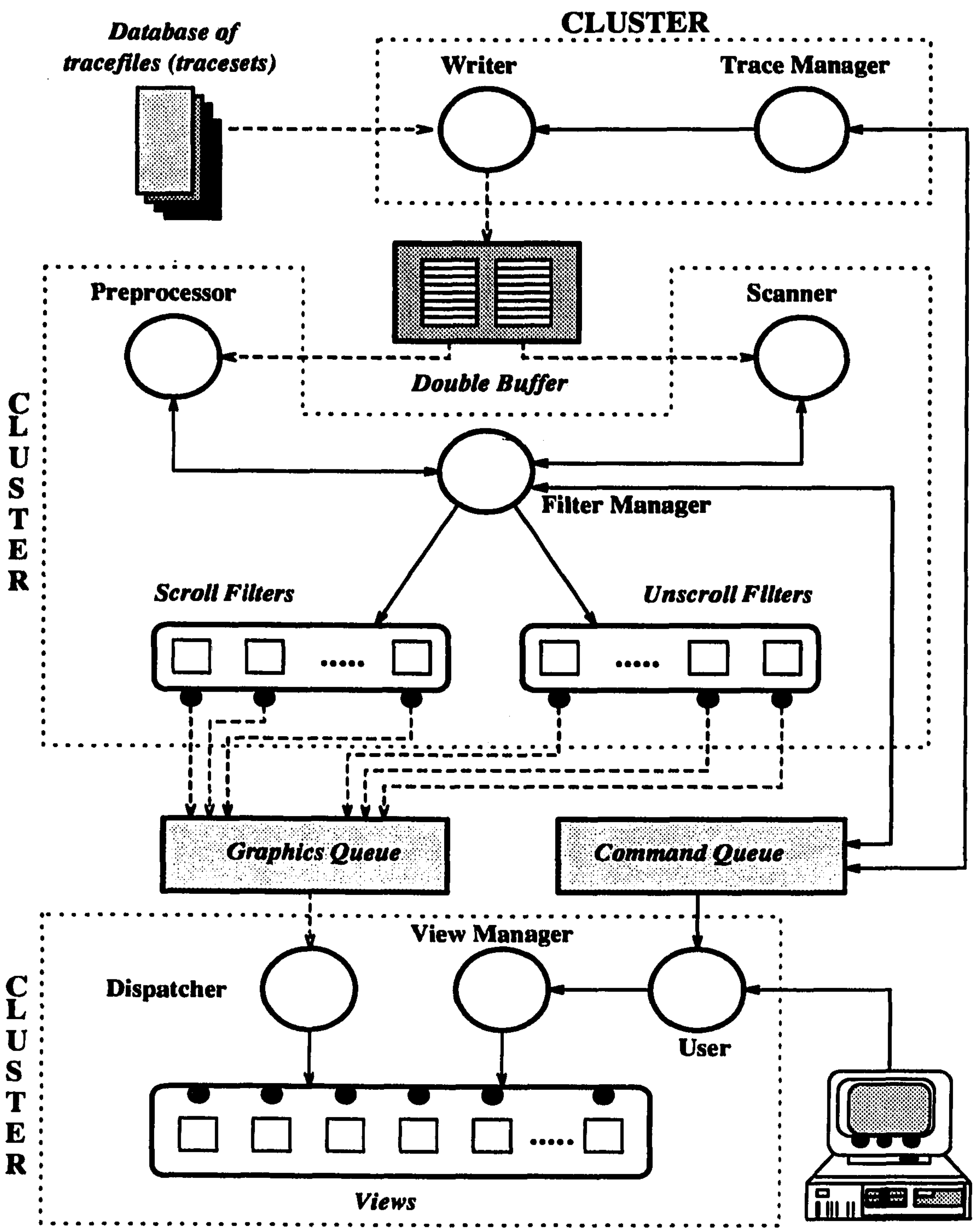


Figure 7.11: Software Architecture of the COOPG

extended to incorporate multiple *Writers* to speed up the preprocessing phase of the Filter cluster with little effort.

If multiple *Writers* are to be employed, the *Trace Manager* must provide an efficient mechanism to manage a database of trace files in which a trace file consists of disjoint partitions of trace events. It should also coordinate and schedule *Writers* so that *Writers* can handle a set of partitions in parallel. To reflect this possible extension in implementation, we will use the term *traceset* to denote a trace file. That is, a *traceset* is a collective terminology representing a trace file which consists of one or more partitions of trace events.

Filter Cluster The Filter cluster processes trace events read from the Double Buffer in the shared memory, collects performance related information and generates graphics requests to draw the information on the display in various forms. *Filter Manager*, *Preprocessor* and *Scanner* tasks are main components of the Filter cluster.

When we designed the Filter cluster, there were two design alternatives. The first alternative was to exploit higher degree of concurrency by making each filter as an active object. According to the class hierarchy in Figure 7.5, all the instances of the classes in that hierarchy and the instances of filter classes for particular views might be implemented as active objects. This scheme is a naive attempt to achieve concurrency based on the software architecture of the OOPG in Figure 7.1. It has an advantage of conceptual consistency between real-world objects and their physical implementations. But the first alternative experienced major performance degradation due to two main reasons that will be explained below: *view synchronization* requirement and *data dependency* among filters along the hierarchy in Figure 7.5.

Filters are in one of the two possible states as shown in Figure 7.12 — *preprocessing* state and *running* state.

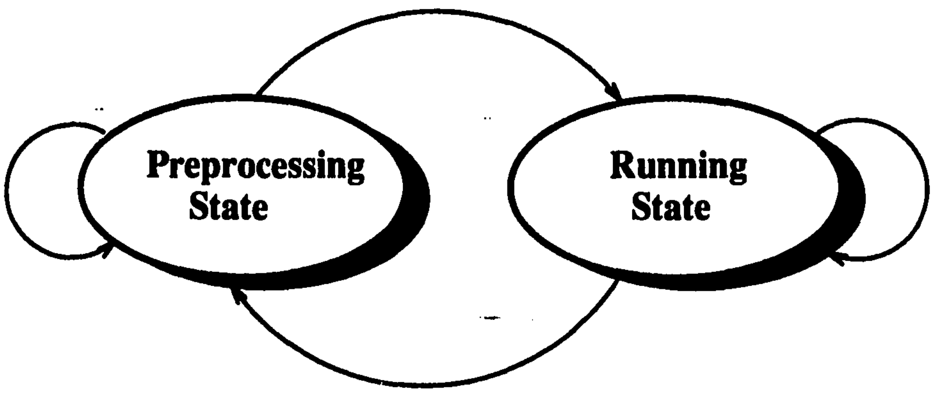


Figure 7.12: States of a Filter

In the *preprocessing* phase, all the trace events in a given traceset is processed once to obtain class specific global, performance related information to configure filters and views. The global information includes the number of processors involved, maximums and minimums of some metrics, start and stop times of the instrumentation, and other information necessary to instantiate data structures needed for particular filters and views. In the *running* state, trace events are processed sequentially one at a time to analyze the current behavior and to adjust performance metrics, and to draw performance views accordingly. Note, however, that all filters cannot proceed to process trace events at their own full speed at least in the *running* phase because the pictures drawn in the display must be synchronized at each point in time among all the opened views. Otherwise, the snapshot of the entire pictures is totally out of synchronization and cannot provide the clues for the relationship among views which may be the most valuable information in performance analysis. This makes the operation of each filter be synchronized for each trace event, and, as a result, incurs a severe performance penalty due to the heavy synchronization overhead. The view synchronization point among filters is shown at level 2 in Figure 7.13. Note that in the OOPG, a view object is a member of a filter. Thus view synchronization must be considered among filters instead of views.

Note also that while filters in the lower lever classes in the hierarchy are executing, filters in the higher lever classes must wait. This is because the current information maintained by the high level filters must be accessed by the lower level filters to ensure that all filters see one consistent state at a point in time. Thus another synchronization is required for data integrity at level 1 as well as level 2 in Figure 7.13. This observation of data dependency resulting from data sharing makes it infeasible to implement a plausible *pipelining* in the execution of filters.

In order to overcome the problems in the first alternative, we came up with a second design alternative which is more efficient while preserving the conceptual cleanness. In the second alternative, filters become passive objects with standard interfaces for preprocessing, running and configuration. Instead, artificial, functional objects, called *Preprocessor* and *Scanner* are created to handle the preprocessing and running phases of filters respectively as shown in Figure 7.14. Because of the sequential processing nature of the performance visualization environment, only single instance of the *Scanner* is created. But our design does not exclude the possibility of multiple *Preprocessor* instances. However, the current prototype implementation employs only one *Preprocessor*.

The *Filter Manager* maintains a set of filters and controls the operation of the *Preprocessor* and the *Scanner*. Like the OOPG, it maintains two sets of filters based

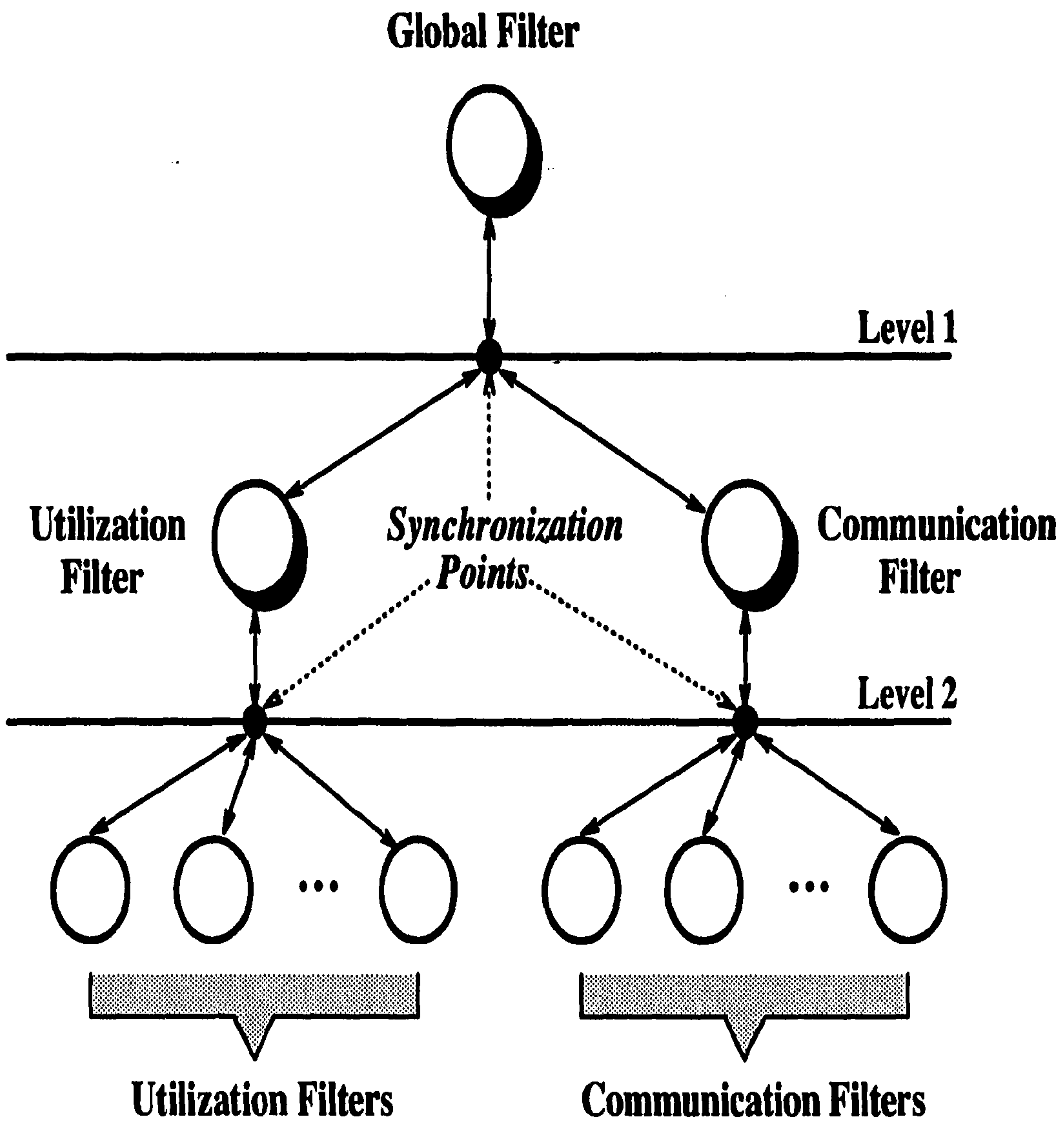


Figure 7.13: Synchronization Points of the Filters

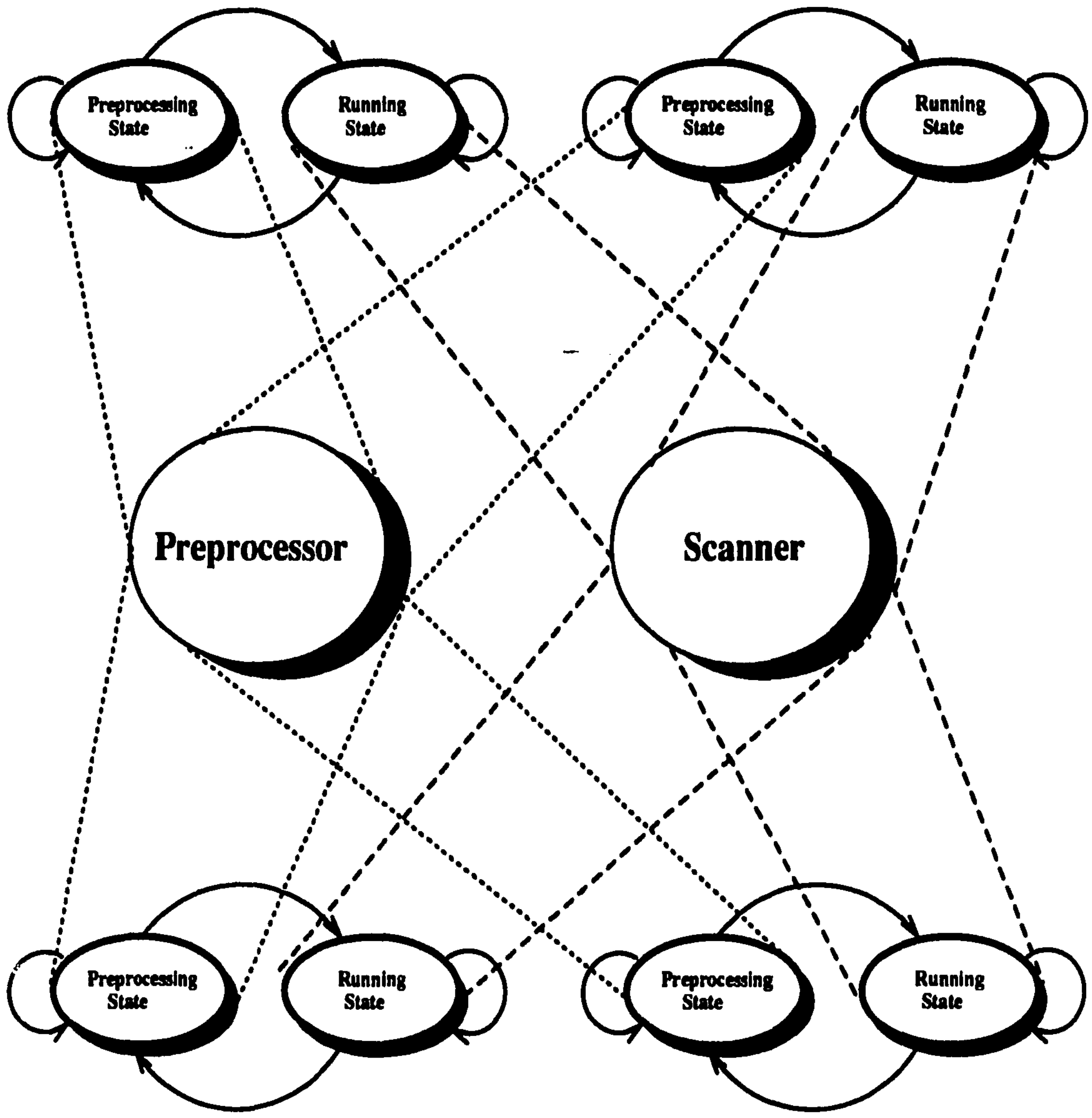


Figure 7.14: Preprocessor and Scanner Objects

on their scrolling capability of the views bounded to them. At the beginning of the preprocessing phase, the *Filter Manager* wakes up the *Preprocessor.* Then the *Preprocessor* retrieves trace event one at a time from the Double Buffer in the shared memory and send a message to each filter for preprocessing (invoke the *rcvPreprocessTrace* method) from the higher level filters to lower level filters in turn. At the end of the preprocessing phase, *preprocess* method is called for each filter to wrap up preprocessing. After preprocessing is done, the *Preprocessor* blocks itself until signaled by the *Filter Manager* later on user's request. In the running phase, the *Scanner* is activated by the *Filter Manager.* The operation of the *Scanner* is almost identical as the *Preprocessor* except that it invokes the *rcvScanTrace* method of each filter.

In order to achieve a better efficiency, the functions[2] of filters are decoupled with operations of the views bounded to each filter. And the view synchronization point is shifted from the Filter cluster to the View cluster. Then how each filter send graphics requests to the corresponding view to display performance pictures? As a solution, we introduced the concept of *proxy* to handle communications among filters and views as shown in Figure 7.15.

When a filter in the Filter cluster needs to send a message to a remote object (view object) in the View cluster, it send a message to a local proxy view object for accessing remote object. Proxies are local, passive view objects in the Filter cluster environment with the same interfaces (public methods) as the remote view objects. Proxies act as local stubs for remote view objects, performing argument packing and sending graphics requests via undering communication mechanism. In our prototype

[2]Actually, the functions of the filters are executed via *Preprocessor* and the *Scanner.*

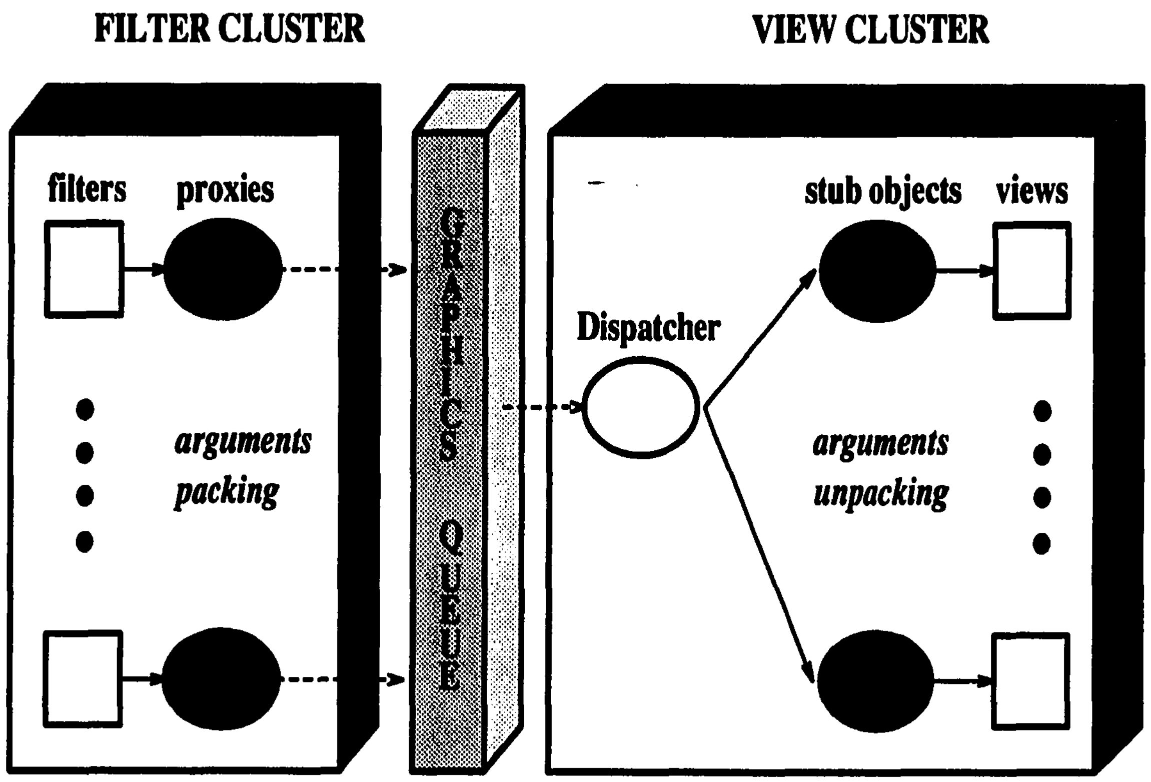


Figure 7.15: Proxies

implementation, the UNIX message queues are used as a communication mechanism as seen in Figure 7.11. In the *running* phase, filters can proceed to process trace events and generate graphics requests at full through the *Scanner* speed while providing a synchronization hint to the views in the View cluster. Synchronization hint is a predetermined message following the graphics requests of all the filters for each trace event. The sum of the graphics requests of all the filters per trace event is called a *frame*. The synchronization hint is necessary to support user's control over the behavior of the visualization environment like pause, resume and stepping. Under no circumstances, a subset of opened views should be suspended in the middle of drawing a *frame*.

Like the OOPG, filters are components (Software-ICs) which can be freely plugged into or out of the Filter cluster. This becomes possible due to the *polymorphism* and *dynamic binding* features of the object-oriented programming. As long as a filter is derived from the extant class hierarchy and defines pure virtual functions of the super classes, it can be added safely and elegantly to the performance visualization environment without affecting any other components. The public member functions of the root class *Filter* is listed below.

```
class Filter {
        ...
public:
        Filter();
        virtual ~Filter();
        virtual void    rcvPreprocessTrace(Trace *) = 0;
        virtual void    rcvScanTrace(Trace *) = 0;
        virtual void    preprocess()  = 0;
        virtual void    showPreprocessStatus() = 0;
        virtual int     HasView();
```

```
            virtual void     registerView(P_View *viewp);
            virtual void     drawView();
            virtual void     reset();
    };
```

View Cluster In our performance visualization environment, all the graphics functions are completely separated and isolated physically as well as logically and incorporated into the View cluster. This improves the portability and maintainability of the performance visualization environment.

The View cluster handles the user interface and is responsible for displaying user requested performance information graphically in an efficient and pleasing way. *View Manager*, *Controller* and *Dispatcher* tasks are active components of the View Cluster.

The user can interactively operate the COOPG through the *Controller* using a mouse and a keyboard. The *Controller* provides a main *Control Panel* through which the user can select desired views and control the visualization. The *Control Panel* is shown in Figure 7.16. It provides selection buttons for submenus for utilization, communication, task and other miscellaneous views. The available submenus are shown in Figure 7.17. An option menu for specifying various options and parameters is also included in the *Control Panel.* After selecting the desired views, the user presses *Start* button to begin the visualization for a given traceset. Then visualization then proceeds straight through to the end of the traceset unless the user presses *Pause/Resume* or *Step* buttons for detailed analysis. The *Step* button enables the user to process one trace event at a time in a single-step mode to study a *frame* or a predetermined number of *frames.*

The *Controller* also controls the behaviour of the Trace cluster and the Filter

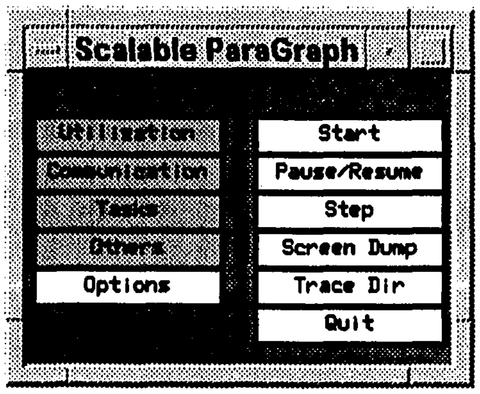
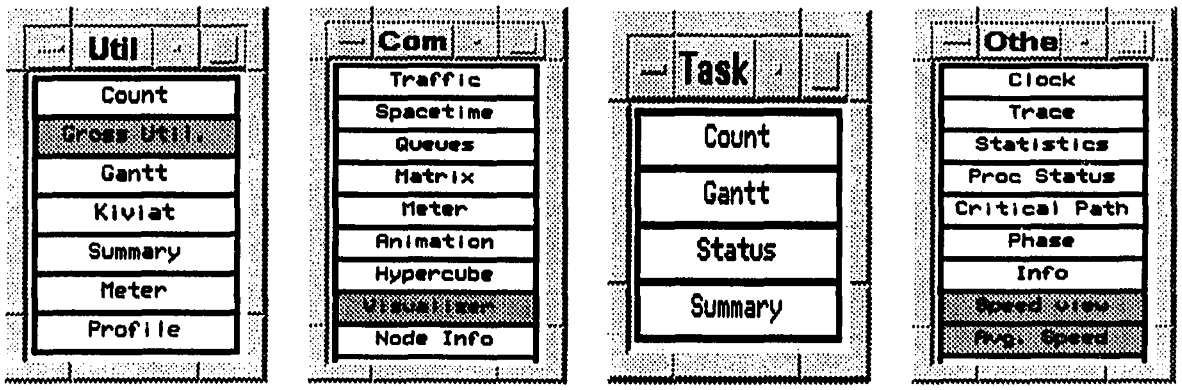


Figure 7.16: The COOPG Control Panel



(a) Utilization (b) Communication (c) Task (d) Other

Figure 7.17: The Submenus for Views

cluster by sending commands to them through the Command Queue, and/or controls the *View Manager* directly as shown in Figure 7.11. Status information is delivered to the View cluster from the Trace cluster and the Filter cluster through the Command Queue as well.

The *View Manager* maintains a set of views and configures those views accordingly after preprocessing of a given set of trace events. Like the *Filter Manager* in the Filter Cluster, the *View Manager* maintains two sets of views depending on the scrolling capability of each view. A pair of a filter and a view occupies the same position in the tables of the *Filter Manager* and *the View Manager,* respectively. And this positional index is used as a key to link that pair while the running phase is in progress. The actual table entries in the *View Manager* are not real view objects. Instead stub objects for each view are maintained in the table.

The *Dispatcher* monitors the Graphic Queue to check if there is any pending graphics request. Graphics requests in the Graphics Queue are fetched one at a time by the *Dispatcher* and the *Dispatcher* identifies an intended view by looking at a key in a request message. Then a corresponding view stub object is activated to perform parameter unpacking and it sends a drawing request message to a real view object (see Figure 7.15).

Detailed explanation of views and their relationship is already described earlier in this chapter while describing the OOPG. And like filters in the Filter Cluster, views are treated as *Software-ICs* which can be added/removed without affecting any other component.

Performance Views

The prototype performance visualization environment classifies views into four categories — utilization, communication, task and other miscellaneous views. Utilization views are helpful in determining the effectiveness with which the processors are used and how evenly the computational load is distributed and balanced across the processors. Communication views primarily deal with interprocessor communication. They are helpful in determining the frequency, volume, and overall communication pattern, and message congestion in the queues. A task[3] is a user-defined portion of the source code. And task views help in identifying the bottlenecks and locating corresponding tasks in the source code. Other views refer to some miscellaneous views and the application specific views of users' own design.

Most of the views supported currently or in the future is borrowed from the ParaGraph. Application specific views can be developed independently and added later to the View cluster as long as they conform to the standard interface. The user can open as many views as possible as long as they can fit into the physical display. Even if it is hard to pay attention to many views at the same time, the availability of multiple views greatly help the user to understand the behavior of parallel programs. The list of views supported presently and in the future are shown in Figure 7.18. Readers are strongly recommended to refer to [20] for the detailed description of each view borrowed from the ParaGraph. The description of newly incorporated views[4] are described below.

[3]Do not confuse this task with an active task representing a thread in the concurrent processing.

[4]Newly designed and incorporated views are marked with an asterisk (*).

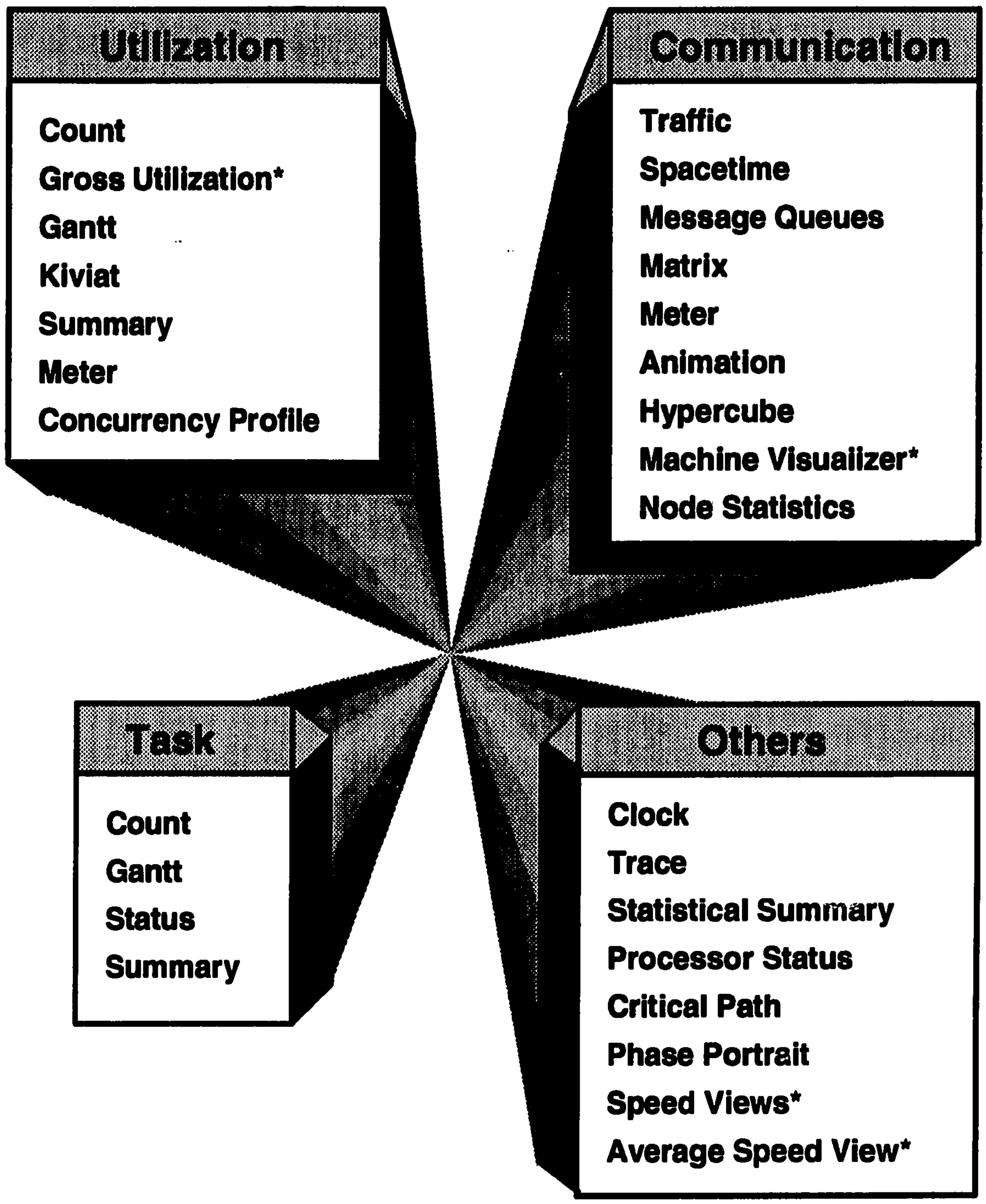


Figure 7.18: Performance Views in the COOPG

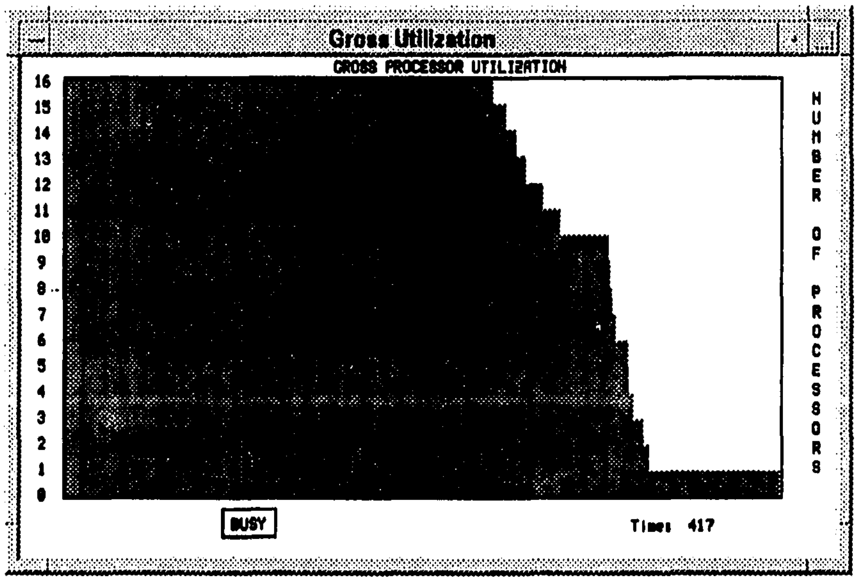
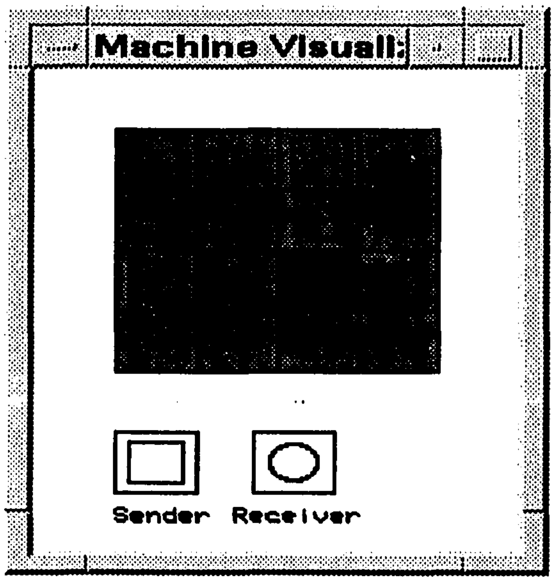


Figure 7.19: Gross Utilization

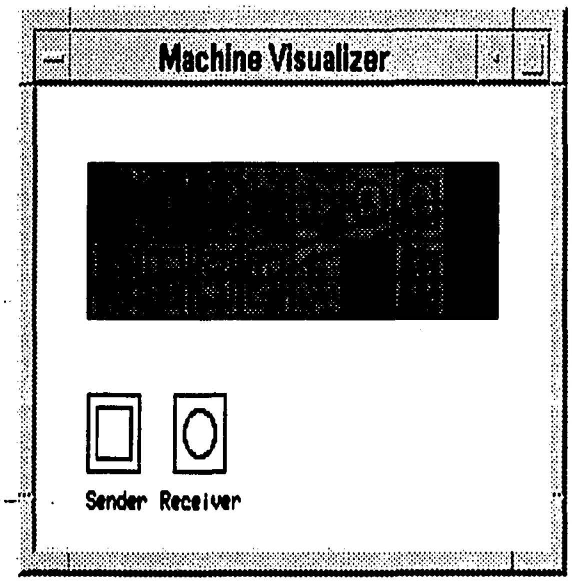
Gross Utilization The Gross Utilization view shows the percentage of time a total number of processors that are currently in each of the three busy/overhead/idle states. The number of processors is on the vertical axis and the current time is on the horizontal axis. The user can select the desired state by pressing a state selection button at the bottom of the view. See Figure 7.19.

Machine Visualizer The Machine Visualizer shows the edges for communication between processors on a two-demensional grid and is configurable for the topology. It displays the current state of each processor as well. See Figure 7.20.

Speed View The Speed View displays the speed of individual processors by a horizontal bar chart as a function of time. Processor number is on the vertical axis, and time is on the horizontal axis, which scrolls as necessary as the simulation



(a) 4 × 4 View



(b) 2 × 8 View

Figure 7.20: Machine Visualizer

proceeds. Color coding is used to denote the speed distribution, and it is displayed as a speed legend.. See Figure 7.21. The user can specify the blocks of code to monitor the speed of each processor. At the end of the simulation, the user can see the speed view normalized to the entire simulation time by pressing *Overview* button.

Average Speed View The Average Speed View depicts the average speed of all the processors by a horizontal bar chart as a function of time. Speed scale is on the vertical axis, and time is on the horizontal axis, which scrolls as necessary as the simulation proceeds. The color of each bar indicates the average speed (in MFLOPS) of the corresponding processors during each user-defined phase. See Figure 7.22.

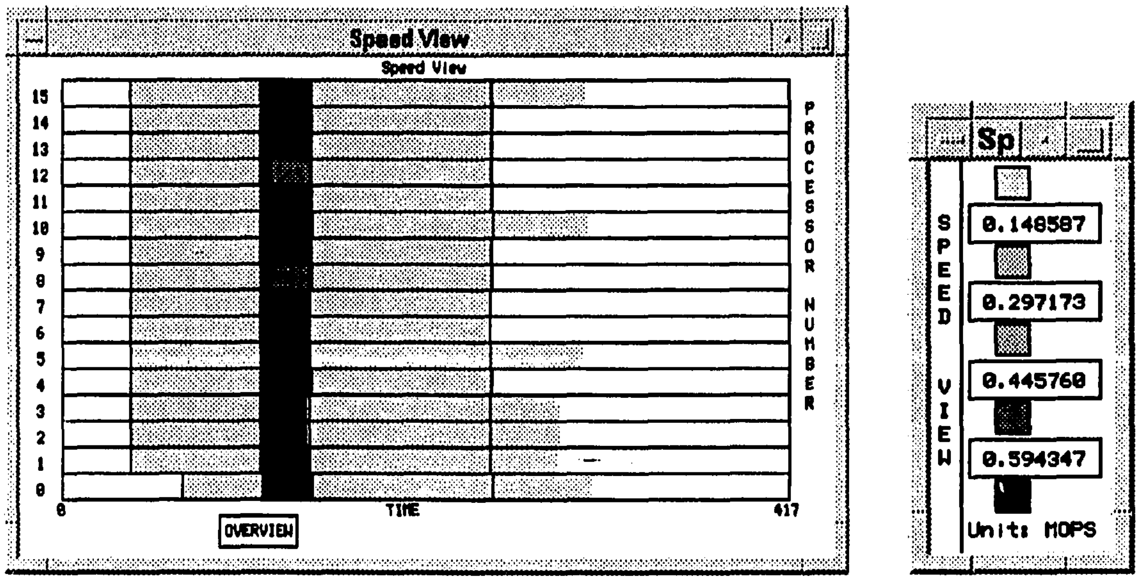


Figure 7.21: Speed View

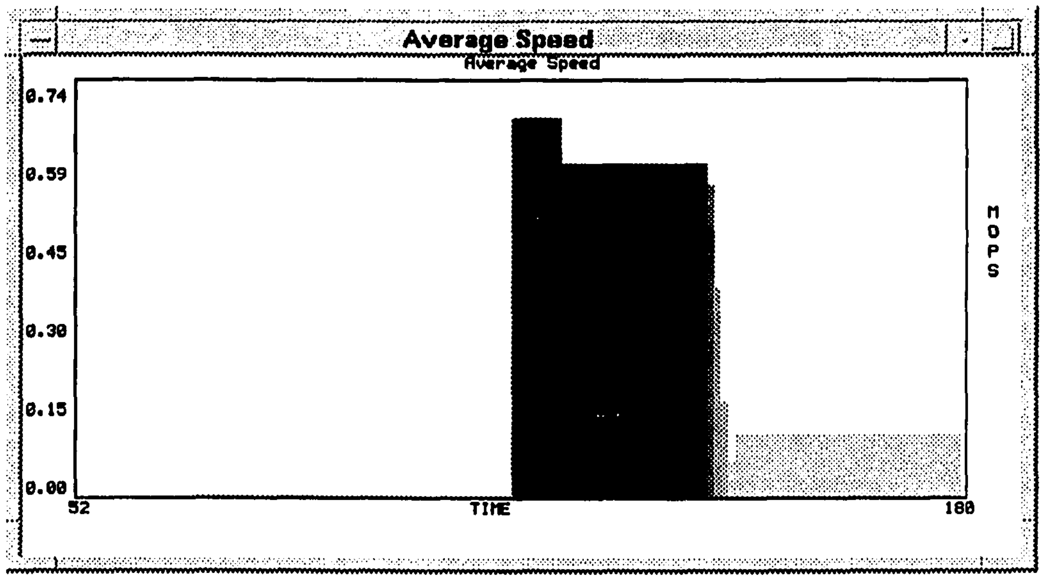


Figure 7.22: Average Speed View

CHAPTER 8. CONCLUSIONS

This chapter provides a summary and discussions of our work, and concludes with the remarks on the future work.

Summary and Discussions

The complexity of distributed concurrent computer systems makes *a priori* performance prediction and evaluation difficult and experimental performance analysis crucial. A performance environment including data collection, analysis, and visualization is needed to access the the logical and physical states of parallel programs. By translating the performance metrics into pictures, the performance visualization environment helps the user to assimilate those information quite an effective way. But designing an efficient, general-purpose performance visualization environment for parallel programs is equally a complex and difficult task. It is almost impossible to satisfy all users' needs and wants. Therefore it is necessary to develop an environment within which user's requirements can be easily added and/or removed. That is, an efficient and extensible environment is strongly desired. Fortunately, an object-oriented approach, both in design and implementation, provides a mechanism to build a simple, flexible, effective, and extensible performance visualization environment.

In this dissertation, we described the implementation of a general-purpose performance visualization framework, called the Concurrent Object-Oriented ParaGraph

(COOPG), based on autonomous objects, either active or passive, using an object-oriented approach. The COOPG is a trace-driven, post-mortem performance visualization environment concentrating on managing and displaying event traces produced from parallel programs running on distributed memory concurrent systems.

Main features of the COOPG is summarized as follows:

- Based on object-oriented design and programming
- Conceptual model is well mapped to the physical implementation .
- Mixed implementation of Cluster/Task Model
- Use of proxy to handle communications between filters and views
- Use of double buffering for overlapped execution of reading trace events and processing those events
- Use of two-stage pipelining for overlapping trace event processing and graphics display
- Easy to maintain and straight-forward to expand to incorporate user's specific views
- Filter-View pairs can be developed independent of other components

The performance visualization environment described here is an ongoing effort. The prototype implementation is as yet completed and further refinement work needs to be done, especially on the class hierarchies for filters and views.

As far as efficiency is concerned, it is yet premature to compare the performance of the COOPG with other visualization tools including ParaGraph. However, since most of the methods defined in classes are implemented as macros using C++ *inline* functions, we expect little performance penalty.

The COOPG must not be considered as a reimplementation of the ParaGraph in another form. Even if it borrowed menus and views from the ParaGraph for the prototype implementation, the underlying mechanism is significantly different. The COOPG provides a framework to create extremely flexible and extensible visualization tools. Functions of the ParaGraph may safely be considered as a subset of those which are provided by the COOPG.

We learned from the prototype implementation that performance visualization is an another promising area which is well suited for an object-oriented design.

Future Work

The Concurrent Object-Oriented ParaGraph is an efficient and flexible performance visualization environment providing an insight into the development of parallel programs. But there are still rooms for further improvement. First, mechanims must be developed to visualize massively parallel programs involving a very large number of processors efficiently and reasonably on the graphics terminals. Second, a facility need to be provided to associate performance pictures with the corresponding code segments in the source files. Third, more complex analysis filters must be devised to guide the user instead of just providing performance information. Finally, sound as well as pictures can be incorporated into the visualization environment. Auralization combined with visualization may aid the user in analysing the behavior of parallel programs more effectively [15].

BIBLIOGRAPHY

[1] Athas, W., and Seitz, C., "Multicomputers: message-passing concurrent computers," *Computer*, August 1988, pp. 9–23.

[2] Bates, P., "Debugging heterogeneous distributed systems using event-based models of behavior," *SIGPLAN Notices*, 24, 1989, pp. 11–22.

[3] Bernstein, D., and So, K., "Performance visualization of parallel programs on a shared memory multiprocessor system," *Proceedings of the 1989 International Conference on Parallel Processing*, vol. II, August 1989, pp. 1–10.

[4] Brown, M., "Exploring algorithms using Balsa-II," *Computer*, May 1988, pp. 14–36.

[5] Champeaux, D., Lea, D., and Faure, P., "The process of object-oriented design," *OOPSALA'92*, 1992, pp. 45-62.

[6] Chang, S., *Visual languages and visual programming*, New York: Plenum Press, 1990.

[7] Chin, R. S., and Chanson, S. T., "Distributed object-based programming systems," *ACM Computing Surveys*, Vol. 23, No. 1, March 1991, pp. 91–123.

[8] Cox, B. J., "Message/object programming: an evolutionary change in programming technology," *IEEE Software*, Jan. 1984, pp. 50-61.

[9] Cox, B. J., and Novobilski, A. J., *Object-oriented programming an evolutionary approach*, Second edition, Massachusetts: Addison-Wesley Publishing Company, Inc. 1991.

[10] Couch, A. L, "Problems of scale in displaying performance data for loosely coupled multiprocessors," *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, Mar. 1989, pp. 209–212.

[11] Couch, A. L, "Categories and context in scalable execution visualization," *Journal of Parallel and Distributed Computing*, 18, 1993, pp. 195–204.

[12] Cunningham, S., Craighill, N. K., Fong, M. W., and Brown, J. R., *Computer graphics using object-oriented programming*, New York: John Wiley & Sons, Inc. 1992.

[13] Fowler, R., Leblanc, T., and Mellor-Crummey, J., "An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors," *SIGPLAN Notices*, 24, 1989, pp. 163–173.

[14] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D., *Solving problems on concurrent processors*, Englewood Cliffs: Prentice-Hall, 1988.

[15] Francioni J. M., and Jackson, J. A., "Breaking the silence: auralization of parallel program behavior," *Journal of Parallel and Distributed Computing*, 18, 1993, pp. 181–194.

[16] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H., "A users' guide to PICL: a portable instrumented communication library," *Tech. Report ORNL/TM-11616*, Oak Ridge National Laboratory, Oak Ridge, TN, August 1990.

[17] Glenn, R. R., and Pryor, D. V., "Instrumentation for a massively parallel MIMD application," *Journal of Parallel and Distributed Computing*, 12, 1991, pp. 223–236.

[18] Guarna, V., Gannon, D., Jablonowski, D, Malony, A., and Gaur, Y., "Faust: an integrated environment for parallel programming," *IEEE Software*, 6, 1989, pp. 20–27.

[19] Haban, D., and Wybranietz, D., "A hybrid monitor for behavior and performance analysis of distributed systems," *IEEE Trans. on Software Engineering*, 16, 1990, pp. 197–211.

[20] Heath, M. T., and Etherridge, J. A., "Visualizing the performance of parallel programs," *IEEE Software*, Sep. 1990, pp 29–39.

[21] Heath, M. T., and Etherridge, J. A., "Visualizing performance of parallel programs," *Tech. Report ORNL/TM-11813*, Oak Ridge National Laboratory, Oak Ridge, TN, 1991.

[22] Hough, A. A., and Cuny, J. E., "Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation," *Proc. 1987 International Conference on Parallel Processing*, Aug. 1987, pp. 735–738.

[23] Joyce, J., Lomow, G., Slind, K., and Unger, B., "Monitoring distributed systems," *ACM Transactions on Computer Systems*, Vol. 5, No. 2, May 1987, pp. 121–150.

[24] Jones, O., *Introduction to the X window system*, Englewood Cliffs: Prentice Hall, 1988.

[25] Kim, J., and Wright, C. T., "An object-oriented approach towards a general-purpose performance visualization for parallel programs," *Second Annual Midwest Electro-Technology Conference*, April, 1993, pp. 6–9.

[26] Kraemer, E., and Stasko, J. T., "The visualization of parallel systems: an overview," *Journal of Parallel and Distributed Computing*, 18, 1993, pp. 105–117.

[27] Krumme, D. W., and House, B. R., "Collecting performance data on loosely coupled multiprocessors," *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, Mar. 1989, pp. 225–227.

[28] Leblanc, T., Mellor-Crummey, J., and Fowler, R., "Analyzing parallel program execution using multiple views," *Journal of Parallel and Distributed Computing*, 9, 1990, pp. 203–217.

[29] Lehr, T., Segall, Z., Vrsalovic, D. F., Caplan, E., Chung, A., and Fineman, C. E., "Visualizing performance debugging," *Computer*, October, 1989, pp. 38–51.

[30] Lippman, S. B., *C++ Primer*, Massachusetts: Addison-Wesley Publishing Company, Inc. 1991.

[31] Malony, A. D., Reed, D. A., Arendt, J. W., Aydt, R. A., Grabas, D. G., and Totty, B. K., "An integrated performance data collection, analysis, and visualization system," *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, Mar. 1989, pp. 229–236.

[32] Malony, A. D., *Performance observability*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1990.

[33] Mueller, F., "A library implementation of POSIX threads under UNIX," *1993 Winter USENIX*, Jan. 1993, pp. 29–42.

[34] Nichols , K., and Edmark, J., "Modeling multicomputer systems with PARET," *Computer*, May 1988, pp. 39–48. (Parallel Architecture Research and Evaluation Tool).

[35] Peterson, G. E., *Tutorial: object-oriented computing*, Vol. 1, Washington, D.C.: Computer Society Press, 1987.

[36] Pinson, L. J., and Wiener, R. S., *Applications of object-oriented programming*, Massachusetts: Addison-Wesley Publishing Company, Inc. 1990.

[37] Price, B. A., and Baecker, R. M., "The automatic animation of concurrent programs," *Proceedings of the first Moscow International HCI Workshop*, Aug. 1991, pp. 128–137.

[38] Price, B. A., Small, I. S., and Baecker, R. M., "A taxonomy of software visualization," *Proceedings of the 25th Hawaii International Conference on System Sciences*, Jan. 1992.

[39] Reed, D. A., Olson, R. D., Aydt, R. A., Madhyastha, T. M., Birkett, T., Jensen, D. W., Nazief, B. A. A., and Totty, B. K., "Scalable performance environments for parallel systems," *Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[40] Reed, D. A., Aydt, R. A., Madhyastha, T. M., Noe, R. J., Shields, K. A., and Schwartz, B. W., "The Pablo performance Analysis Environment," *Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[41] Reilly, M. H., *A performance monitor for parallel programs*, Boston: Academic Press, Inc., 1990.

[42] 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 Hypercube, Concurrent Computers, and Applications*, New York: ACM 1989.

[43] Rover, D. T., *Visualization of program performance on concurrent computers*, Ph.D. Thesis, Iowa State University, Ames, IA, 1989.

[44] Rover, D. T, and Wright, C. T., "Pictures of performance: highlighting program activity in time and space," *Proceedings of the Fifth Distributed Memory Computing Conference*, D. Walker and Q. Stout, eds., vol. II, Los Alamitos, CA, April 1990, IEEE Computer Society Press, pp. 1228–1233.

[45] Rover, D. T., Carter, M. B., and Gustafson, J. L., "Performance visualization of SLALOM," *Proceedings of the Sixth Distributed Memory Computing Conference*, New York: IEEE Computer Society, 1991.

[46] Rover, D. T., "A performance visualization paradigm for data parallel computing," *Proceedings of the 25th Hawaii International Conference on System Sciences*, Jan. 1992.

[47] Rover, D. T, and Wright, C. T., "Visualizing the performance of SPMD and data-parallel programs," *Journal of Parallel and Distributed Computing*, 18, 1993, pp. 129–146.

[48] Rudolph, D. C., and Reed, D. A., "CRYSTAL: Intel iPSC/2 operating system instrumentation," *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, Mar. 1989, pp. 249–252.

[49] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-oriented modeling and design*, Englewood Cliffs: Prentice Hall, 1991.

[50] Sarukkai, S. R., and Gannon D., "SIEVE: a performance debugging environment for parallel programs," *Journal of Parallel and Distributed Computing*, 18, 1993, pp. 147–168.

[51] Segall, Z., and Rudolph, L., "PIE: a programming and instrumentation environment for parallel processing," *IEEE Software*, 2, 1985, pp. 22–37.

[52] Simmons, Margaret, and Koskela, R., editors, *Performance instrumentation and visualization*, New York: ACM & Addison-Wesley, 1990.

[53] Stroustrup, B., and Shopiro, J. E., "A set of C++ classes for co-routine style programming," *1987 USENIX C++ Papers*, 1987, pp. 417–439.

[54] Stroustrup, B., *The C++ programming language*, Second edition, Massachusetts: Addison-Wesley Publishing Company, Inc. 1991.

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my major professor, Dr. Charles. T. Wright, for supervising my degree program and for invaluable advice and time guiding this work and correcting the dissertation.

I wish to express my sincere thanks to Dr. Diane T. Rover in Michigan State University for suggesting this research topic. Thanks are also due to my committee members, Dr. Prabhu, Dr. J. Gustabson, Dr. J. Davis, and Dr. T. Smay, for taking time to help with my research.

I appreciate the assistance of J. Metzger and M. Carter in the Ames Laboratory to help me use the facilities there.

Finally, but most importantly, I would like to thank all my family. Special thanks goes to my parents for their continuous, enduring support and encouragement during the entire degree program. Also, thanks goes to my wife, Sun Ae. Without her love and sacrifice, I couldn't finish this work. I dedicate this work to my son, In Young. He has been always my dream and inspiration.