TVIEW - a graphical representation of programs running on the transputer.

By
Hilde Anita Larsen
B.Sc., University of Bergen, Norway, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE STUDIES (DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming to the required standard

UNIVERSITY OF BRITISH COLUMBIA
March 1991
© Hilde Larsen, 1991
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date 23.03.91
Abstract

The primary motivation behind building multiprocessors is to cost-effectively improve system performance. Debugging and performance analysis of parallel programs, however, are complex tasks and the lack of tools to observe the behaviour of a program running on a multicomputer network, limits the programmers ability to efficiently debug and optimize parallel programs.

In this thesis we investigate the use of different graphical representations of parallel programs running on a network of transputers as a tool for performance analysis. Post-mortem traces collected from the program execution by the underlying monitor enables us to graphically reconstruct the states of the system that were true at run time. We show how the performance analyst can get an accurate view of the behaviour of the parallel program during execution by using a basic set of visualization tools.

The challenge is to determine the types of graphical displays that are most useful for presenting the behaviour and performance of a parallel program.

Problems in graphically visualization of parallel program executions are: efficiently managing of potentially large volumes of performance data, ensuring consistency among the tool-components, correctly responding to any combinations of user events, meeting the desirable system requirements of extensibility and maintainability.
# Contents

Abstract ii

Contents iii

List of Figures v

Acknowledgement vi

1 Introduction 1
   1.1 The Environment ................................................. 2
   1.2 Motivation and Objectives ........................................ 4
   1.3 Problems ...................................................... 7
   1.4 Thesis Outline .................................................. 8

2 Related Work 9
   2.1 Seecube ................................................................ 9
   2.2 HyperView ......................................................... 11
   2.3 IPS .................................................................... 12

3 TVIEW - The Users's Perspective. 14
   3.1 The Monitor - Underlying Mechanisms ............................. 15
      3.1.1 Event Trace .................................................. 16
      3.1.2 Utilization Trace ............................................ 17
   3.2 Performance - What is it? ........................................... 17
   3.3 A Display Overview .................................................. 19
      3.3.1 The Event Display ........................................... 21
      3.3.2 Weighted Critical Path Analysis ............................. 28
      3.3.3 Topology Display and Dynamic Sampling .................. 28
      3.3.4 Message Passing Display .................................... 33
      3.3.5 Summary ........................................................ 34
4 Design and Implementation

4.1 Object Oriented programming methodology ............................................. 36
  4.1.1 InterViews .................................................................................. 36

4.2 Decomposition Mechanisms ..................................................................... 37

4.3 Implementing the Dynamic Mechanisms .................................................. 40

4.4 Event Control Mechanisms ..................................................................... 41

4.5 Trace Control ......................................................................................... 43

4.6 Support for Adding new Analyzing and Display Components .................. 44

4.7 Experience with C++ and InterViews ..................................................... 44

5 Conclusion, and Future Work .................................................................... 48
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>General Flow</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>The Trace Entry Structure (@Jie Jiang)</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>The TVIEW Environment</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Event Display: EM - mode (Shows a series of broadcasts from several processors)</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Event Display: TTM - mode (Also shows a series of broadcast, but this is not clearly seen)</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>Event Display showing user defined events</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Event Display: TTM-mode versus EM-mode (Initialization phase)</td>
<td>26</td>
</tr>
<tr>
<td>3.8</td>
<td>Event Display: Overlapping of Event Icons in the TTM-mode (Below) versus non Overlapping of Event Icons in the EM-mode (Above)</td>
<td>27</td>
</tr>
<tr>
<td>3.9</td>
<td>Basic Topology Display</td>
<td>29</td>
</tr>
<tr>
<td>3.10</td>
<td>Mapping of Statistical Information</td>
<td>30</td>
</tr>
<tr>
<td>3.11</td>
<td>Message Passing Display along with an Event Display. The Message Passing Display reflect the communication patterns more clearly than the Event Display.</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Toolkit hierarchy</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>TVIEW's main components</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>The DynamicBaseView Class</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>The Viewlist</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>General Flow</td>
<td>42</td>
</tr>
<tr>
<td>4.6</td>
<td>Inheritance and Virtual Functions</td>
<td>45</td>
</tr>
</tbody>
</table>
Acknowledgements

I thank Dr. Alan Wagner for acting as thesis supervisor. This work benefited greatly from his expertise, and without his encouragement and support this work would hardly have made it through.

Thanks are extended to my project partner Jie Jiang, who as being the first experimental user of Tview brought valuable feedback and suggestions.

Norman Goldstein not only gave me several rides back home, he also generously introduced me to the craft of riding the transputer.

I would also like to thank the technical staff at the department of computer science for always helping me out whenever I came across some "unexplainable" technical problem. As it turned out, they were all very much explainable.

Thanks to Lawrence Chee for spending a whole sunny Sunday to proofread my thesis, facelifting the write-up to the very better.

Most of all I would like to thank Jonny Hesthammer for his emotional support, always putting me back on track whenever I lost sight of the global picture.

And finally, my thanks goes to all my fellow students at UBC who made my years in Canada so enjoyable.
Chapter 1

Introduction

A multicomputer network is a locally connected set of loosely coupled autonomous nodes interconnected in some topology. Each node is equipped with a microprocessor, local memory and hardware support for internode communication. Multicomputer networks have shown to be cost effective high performance systems, and they are becoming commonplace in scientific computing.

The programmer, however, has to deal with the added complexity that is involved when developing programs under a parallel processing environment. In addition, the lack of tools to observe the behaviour of a program running on a multicomputer system limits the programmers ability to effectively debug and optimize parallel programs. For parallel machines, in particular message-passing multicomputers with distributed memory, the correspondence between source code and execution is complex. This is a result of the communication delays, the lack of a global clock and the lack of global control.

Graphical workstations along with their graphical user interfaces, have changed the way we work with computers. Windowing system supports the view of several concurrent
activities by providing the user with a collection of windows. The set of windows create a good environment for visualizing the performance of parallel programs. Different performance data and system characteristics can be displayed in an organized but flexible way, enabling the performance analyst to view the global state of the system from different perspectives, and to explore the recorded computation.

This thesis investigates the use of different graphical representations of program executions on a network of transputers as a tool for performance analysis. Traces collected during program execution by the monitor enables us to graphically reconstruct the states of the system that were true during the program run. We show that by providing the analyst with a basic set of visualization tools, the analyst can get an accurate view of the behaviour of the parallel program during execution. The challenge is to determine the types of visual displays that will be most useful for presenting the behaviour of a parallel program and its performance.

1.1 The Environment

TIPS (Transputer-based Interactive Parallelizing System) is the parallel programming development environment that is being developed at the University of British Columbia. The goal of TIPS is to provide the programmer with a convenient environment for the design, development, and execution of parallel programs targeted for transputer based multicomputers. It incorporates 4 major tools; TMON, TMAP, TRES and TVIEW.

TMON: This is the performance monitor that runs on the transputer. It measures resource utilization at regular intervals and traces process events transparently
CHAPTER 1. INTRODUCTION

during execution. Upon completion of the program this data is used by TVIEW to display the results to the user. TMON is further outlined in Chapter 3.

**TMAP**: TMAP is a system built on top of the operating system. It supports a topology independent mapping facility that hides the underlying architecture from the user. It is capable of automatically generating a mapping of a virtual process graph onto a given network topology. The user provides TMAP with a description of the program. This program is described as a collection of communicating processes, i.e., as a process graph. In addition, the user can specify additional information such as communication channel, channel weight (estimated usage of channel), number of transputers to use, and, lastly, the number of ports to use.

**TRES**: The focus of this tool is to develop performance models for specific paradigms in order to identify their resource requirements. In accordance with the paradigm used and the computational task to be run, TRES determines the optimal topology and number of processors to use. The user program is linked with the appropriate paradigm of the implementation. According to the hardware characteristics and the software overhead associated with the given implementation, a performance model is constructed. The performance model, along with application specific parameters, are used to determine the resources required by this program. These resource requirements include the number of nodes needed to maximize performance. This information is passed to TMAP, which then uses this resource information to map the application onto the network.

**TVIEW**: This is a tool that provides the analyst with a set of graphical visualization
tools to support the programmer in the analyzing and debugging process. TVIEW is the result of the work described in this thesis.

TIPS is currently targeted for the 74-node transputer based multicomputer at the Department of Computer Science at University of British Columbia. A host workstation, Sun4, along with 74 IMS T800 transputers, forms the multicomputer. Each transputer has a 32-bit 10 MIPS processor, 1 or 2 MBytes of on-chip memory, and four 20 Mbit/sec bidirectional serial links. The topology of the transputer network is statically reconfigured by using software on the host.

The Trollius Operating System is used. Trollius is a parallel operating system developed jointly at Cornell Theory Center and Ohio State University. It provides mechanisms for message passing between processes in addition to routines for process creation, process destruction and access to remote file systems.

1.2 Motivation and Objectives

Often, in a parallel programming environment, the only tool available for determining the runtime behaviour of the program, is by placing print statements at strategic locations in the program. There are several reasons why this solution is not well suited for a parallel message-passing environment. Print statements have to be routed to the host, and as they increase in number, they are likely to impose an unacceptable communication load on the system, which in turn degrades the performance. Due to the communication delays across the network, the print statements will arrive at the host in a random order,
rather than in a chronological order\textsuperscript{1}. As the number of processors increases, so does the number of print statements needed to capture events of interest, and the performance analyst will soon find himself overwhelmed with random sorted print statements from different processors, obscuring a clear view of the program execution.

To aid the analyst it is necessary to automate the analyzing process, including both data collection and analysis. The main objective of this thesis is to develop a set of visual displays that intuitively and correctly reflects the behaviour and performance of parallel programs running on a network of transputers. By displaying performance data graphically, we can take advantage of the ability of the human eye to identify and interpret patterns in the graphical images.

Generally, there are two major reasons to investigate the execution of a program, namely debugging and performance analysis. The main distinction between the two is that debugging can control the execution while performance monitoring stands by nonintrusively and observes. Even though the focus of this thesis is to visualize program behaviour and performance, it can also be used as a tool for detecting bugs in the program execution.

The design and implementation of the graphical user interface should meet the following criteria:

- Maintainability: The graphical tool environment should be easy to maintain. This can be partly realized by developing well defined modules that supports data hiding and encapsulation, along with simple and clear interfaces as a mean of interacting between them. By following good software engineering principles, it should be

\textsuperscript{1}This can, however, be solved by synchronizing the clocks and timestamping the events
possible to modify different parts of the system without affecting other parts.

- **Efficient**: The user should be able to use the visualization tool in a time-effective manner. If the tool is slow and awkward to use, the user may turn to simpler and more primitive means for his performance study.

- **Extensibility**: It should be easy to incorporate other analyzing tools into the environment. Different programs, possibly based on different programming paradigms, may require a different set of analyzing techniques. Or a programmer might want to approach the analyzing process using a different set of techniques than those already provided by the visualization tool. This could for instance be analyzing methods that are more data or algorithm oriented [Brown88], rather than focusing at the event activity.

From the user's perspective, the graphical tool environment should achieve the following goals:

- **User-Efficiency**: The environment should allow the user to efficiently develop, debug, and optimize parallel programs.

- **Functionality**: It should give feedback to the analyst of the program behaviour and performance in a correct and intuitive manner, without requiring any extended effort by the analyst.

- **User-friendly**: It should be easy to use, and the user should be able to easily access the information in which the user is interested.
CHAPTER 1. INTRODUCTION

1.3 Problems

Performance visualization seeks to improve the analyst’s ability to understand the performance data. The challenge is to determine how to best visualize the information available, i.e. what types of displays are most beneficial. It is necessary to decide on what performance information is most valuable, or rather, what system characteristics should be focussed on in order to catch factors that have generally most impact on the performance. As communication plays an important role in parallel computation, it is natural to focus the performance analysis on the communication activity. However, other aspects are also of importance, and should accordingly be addressed.

From the application's point of view, it is necessary to develop ways of managing the set of displays. For example, it is important to keep the displays time consistent with each other, i.e., the time interval reflected by each view, must be the same. As a user manipulates an individual display entity, possibly changing the time domain, this change must be propagated throughout the system.

To meet the requirements of extensibility and maintainability, careful attention must be paid to the design of the tool. If there is no support to add new components to the tool environment, the program structure will become difficult to maintain and expand. The goal is to be able to add new analyzing and visualization components without requiring any modification to the existing software. This problem is partly solved by using an object-oriented design and programming methodology.
1.4 Thesis Outline

This section briefly describes the contents and structure of the thesis. In Chapter 2, we describe some approaches to performance visualization that have already been developed and used in other systems. Ideas that have contributed to this work are highlighted.

Chapter 3 presents TVIEW from the analyst's point of view. The concept of performance observability is introduced to show how TVIEW can focus the user's attention on potential performance problems. We discuss the tool components capabilities, and relate them to the previous work in this area.

Chapter 4 outlines the design and implementation strategies. As TVIEW is based on an object oriented design methodology, we discuss the design and implementation in terms of object decomposition, composition and their interaction mechanisms. In section 4.7 we present our experiences with using an object oriented approach, including our experience with the object oriented programming language C++\(^2\) and the object oriented window toolkit InterViews\(^3\).

Finally, Chapter 5 concludes this thesis by stating what we have learned and possible directions for future work.

\(^2\)C++ is a trademark of AT&T
\(^3\)InterViews is a trademark of Stanford
Chapter 2
Related Work

In the past few years there has been considerable interest in graphical user interfaces for parallel machines. There are several existing projects, similar to this one, that allow users to display and analyze their parallel programs. In this section a survey of previous work in this area that have contributed to this thesis is given.

The tools that are presented have several features in common. Some of the major commonalities are: the use of a graph to represent the underlying system, where nodes denote computing engines and edges denote communication links; the focus on the communication characteristics since it is the "cost" of parallelism; the use of dynamic displays rather than static characteristics since they better show how system dynamics evolve over time; the support of higher level views of the raw performance data.

2.1 Seecube

Seecube, developed at Tuft's University [Couch88], is an analysis tool for parallel programs executing on hypercube configured distributed memory machines. Seecube
uses port-mortem event traces from each processor to reconstruct the global state of the network during the computation. This global state is viewed through a variety of graphical representations. Seecube consists of three main parts: the Data Collector, the Resolver and the Sequencer.

**Data Collector**: The Data Collector is a library of communication subroutines on the parallel machine that store diagnostic event traces in the local memories of each processor. Basically, it is only concerned with message sending and receiving events. Each event is given a timestamp, a unique sequence number, the process id, the destination node, the destination process id, and the message type are stored.

**Resolver**  The resolver processes the event traces given by the data collector, and puts it in a format that is usable by the sequencer. Traces are cross-referenced by matching the message transmission events and the message reception events. Eventually, the traces are sorted into a single trace file.

**Sequencer**  The Sequencer displays the traces output by the resolver. It is an interactive environment which provides the analyst with a set of graphical displays reflecting the computation on the network. A single control panel frame controls the selection of a time domain in which the analyst wants to focus the analysis. The real power of the sequencer lies in its ability to create different views of the same states on the graphics screen.

Seecube focus on the representation of variable sized hypercubes in different ways to graphically represent the topology. The variety of topology displays enables the user to focus on different characteristics of the computation on the hypercube.
CHAPTER 2. RELATED WORK

Seecube is mainly concerned with reconstructing the states, rather than displaying the events. Each node and channel in the topology view, reflects some state in accordance to the global time, i.e. sending message, receiving message, etc. A nice feature of Seecube is its use of colour to display different states. For example, to show communication activity, warm or hot colours, like red, are used to express heavy communication, while cold colours like blue are used to express the lack of communication. Likewise, a busy processor can be assigned a red colour, and blue can be used to express inactivity.

Some of the techniques first developed in Seecube have been adopted and adjusted to fit into our environment. In particular, we found the idea of mapping different system characteristics onto a topology display, such as cpu-utilization and link activities, very attractive. In addition, we found the activity matrix view useful to represent communication relationship between nodes. Unfortunately, as TVIEW is currently targeted at monochrome workstations, we had to use something other than colours to present different system characteristics.

2.2 HyperView

The HyperView performance analysis and visualization system was targeted for hypercubes. Hyperview is a hypercube visualization system based on the Sun window environment and it was developed at the University of Illinois. As this work was also inspired by Seecube [Malony90], there are many similarities between the two systems.

The HyperView user interface permits simultaneous displays of the dynamic system state via a set of different views. From each view the user can extract differing information
which in conjunction to each other enable the user to get an overall feeling of the system
dynamics.

Compared to SeeCube, Hyperview offers the performance analyst a richer set of display
mechanisms and analyzing methods. In particular, Hyperview focussed more on the
analysis of system performance. In addition, Hyperview provides a limited real-time
performance visualization that is able to display system statistics such as cpu-utilization
and link-utilization. This capability also exists in our system, but it has not yet been
exploited. In contrast, SeeCube is strictly post-morten.

HyperView contains three cooperating modules. These include data capture, state
analysis and visualization. HyperView does not provide the user with an event trace
facility. To explore the event trace, the user must use a separate tool, JED [Malony89],
(J)ust an (E)vent (D)isplay, a tool specifically designed to graphically display multipro­
cessor events. Events are represented as graphic icons. Users can control how events are
to be displayed using a special image bitmap facility.

2.3 IPS

IPS is a performance measurement system for parallel and distributed processes de­
developed at the University of Wisconsin-Madison [Miller90]. IPS decomposes a parallel
computation into several levels of abstraction to provide a hierarchical view of the per­
formance. The computation hierarchy is divided into five levels of abstractions: the
program level, the machine level, the process level, the procedure level and the prim­
itive activity level. At each level, different performance metrics are used to describe
the program’s execution. Earlier versions of IPS only provided a simple textual user interface [MiYa87], while the newest version, IPS-2, supplies a graphical user interface. Two of the major tools that IPS provides are the Phase Behaviour Analysis (PBA), and the Critical Path Analysis (CPA). The PBA tries to identify the different phases of the execution, while the CPA finds the path through the execution history that consumed the most amount of time. It also locates the most frequent sequences of events along the critical path.

TMON uses a weighted critical path analysis (WCPA) that extends the CPA to incorporate the notion of parallelism [Jiang90].

The way in which IPS presents some of the performance analysis is not well suited for programs that are distributed over a large network. Cpu-utilization plots are mapped on top of each other in the same display. Given a program running on 8 or more nodes, the plots of cpu-utilization will not be very readable. There are in addition no facilities such as JED, in the Hyperview environment, that would allow the user to browse through the raw event trace data. We have supported a event browsing facility similar to JED. It will be described in section 3.3.3. The lack of the ability to browse through the event trace, prevents the user from having a detailed analysis of the program execution.
Chapter 3

TVIEW - The Users’s Perspective.

TVIEW is an analyzing tool for parallel programs that uses post-mortem event traces from each processor. The event traces are collected by TMON [Jiang90], which is running on each node in the transputer-based multicomputer. From the event traces it is possible to reconstruct the global state of the network during the computation. Different analyzing methods can be applied to the performance data and different displays can be used to present the analysis results (See Figure 3.1).

Figure 3.1: General Flow
CHAPTER 3. TVIEW - THE USER'S PERSPECTIVE.

A variety of displays has been developed, so that they in conjunction or separately, will give the analyst an intuitive and correct picture of the program behaviour and its performance.

Interprocess communication plays an important role in a parallel processing environment, and many of the views concentrates on the network activity. These include communication activity between nodes and processes, concurrency and synchronization graphs, and link utilization statistics. In addition, a display presenting the event graph, reflects the program execution in terms of what, when, and where. That is what event, when it occurred, and where it occurred in terms of the processor id. A network topology view is used to display statistical information.

Each of the displays that is provided in TVIEW will be described in the next sections. The descriptions are presented with respect to the user's perspective. Later in Chapter 4 we focus on the different development and implementation strategies that were used. But first a more detailed outline of TMON, the monitor, is given. The following section also briefly addresses the concept of multiprocessor performance.

3.1 The Monitor - Underlying Mechanisms

The lower level monitoring system, TMON [Jiang90] provides the data used by TVIEW. TMON runs in the background on the transputer nodes and collects the data needed by the analyzing tools. There are two distinct strategies for collecting data from a network of processors, event tracing and event sampling. The first strategy records specific events as they occur. From the event record it is possible to reconstruct the states
of the network. States that are recognizable are the interaction activity at some specific
time, i.e. who is busy sending a message, and who is blocked waiting for a message. The
second strategy, event sampling records system statistics such as cpu-utilization, link
utilizations, etc, at specific time intervals during the computation. The event tracing
method can be expensive, both in terms of memory and communication costs. The local
memory at each transputer node is limited, and the trace data can only be temporarily
stored at the node. Tmon uses an adaptive strategy to report the trace data back to the
host. Trace data is sent at regular intervals only when the traffic across the network is
lightly loaded. However, even in the case of adaptive reporting, as a result as the number
of traced events increases, the monitoring process is likely to introduce an impact on the
performance being measured. Another problem that has to be dealt with is to define a
total ordering of the events across the transputer network. This was accomplished by
keeping the clocks synchronized and timestamping the events.

TMON integrates both the event sampling and the event tracing into the monitoring
process. The event sampling requires less processing time, memory and communication
overhead, as long as the event sampling intervals are sufficiently large(5msec). It does
not, however, provide the performance analyst with a set of detailed information about
the program execution. During program execution, TMON produces two files of trace
data, the Event Trace, and the Utilization Trace.

3.1.1 Event Trace

TMON automatically traces five standard event types: process creation, process exit,
message transmission, message reception, and receive call. Users can also define their own events by inserting probe-statements into the code. Each user and system event is timestamped. Along with the timestamp, additional information such as place of occurrence and process id is stored along with other event specific information.

The trace file is simply a file of event-records stored consecutively (For the structure of the trace entries, see Figure 4.2). To access the trace file the Unix command `lseek()` along with `read()` are used. A higher level interface is built on top of these routines; namely `NextEvent(nodeno)` and `GetEvent(nodeno, eventid)`, which returns the next event not yet read and a specific event given its event number respectively. More details on trace management are given in section 4.5.

3.1.2 Utilization Trace

At regular intervals, defined by the monitor program, each processor node measures the utilization of its cpu and its four links. This information is used to show the resource utilization of the system. The utilization trace is similar to the event trace, including the interface built on top of `lseek()` and `read()`.

3.2 Performance - What is it?

Performance can be measured in terms of different metrics. The most important one is time. What is the execution time? How fast is it? A second metric is: How efficient is it? How well does it utilize the given resources?
Figure 3.2: The Trace Entry Structure (©Jie Jiang).
Studies have shown that a multiprocessor is running at optimal performance when all the processors are kept busy doing useful work.

In parallel processing, the speed-up obtained is largely dependent on the task granularity. The granularity can be defined as the amount of time spend on communication compared to computation. Coarse-grain parallelism produces a relatively small amount of communication compared to the amount of computation, while in fine-grained parallelism, there is a relatively large amount of communication overhead. Generally, an increased amount of parallelism incur an increased amount of communication overhead. To maximize performance, it is important to balance the degree of parallelism with the communication overhead.

3.3 A Display Overview

TVIEW was developed to display performance results to the analyst in a user friendly and intuitive way. Most of the displays are dynamic. Dynamic performance analysis captures changes in performance behaviour over some time domain. As the user "replays" the program execution, new states and utilization information is displayed, and the changing graphical attributes of the displays allows the performance analyst to observe time-based performance characteristics of the execution.

Figure 4.3 shows a set of currently working displays. TVIEW's main control panel consists of two parts; a main menu and a global clock. The global clock shows the current time, and it also includes a control panel for "replaying" the program computation. The user can start, stop or restart the program trace. The main menu allows the user to add
new displays to the interactive visualization environment. The types of displays available are the event display, a topology display onto which several different information statistics are mapped, a message-passing display, and a display that lists various system statistics. In the next few sections each display is described in detail.

Figure 3.3: The TVIEW Environment
3.3.1 The Event Display

Event tracing has proven to be an effective to gather performance data on multi-computer systems [Malony89]. A natural way to understand the runtime behaviour of parallel programs is to analyze the history of the program’s execution. Textual review of event traces, however, is all but impossible, since the trace files can be large. Therefore, the graphical event display provides the user with a convenient way to investigate and to interact with the event traces.

A parallel program consists of several concurrent processes running on multicomputer nodes that interact with each other by message passing. From the user’s perspective, the events of interest include: process creation, process destruction, message send, message receive, call receive, and possibly user-defined events. By design, these are the events captured by the monitor.

The event display is a graph of processor events versus time. Time is along the horizontal axis and the processors are organized vertically (See Figure 4.4). Each event is represented by a specific event icon.

In the first phase of performance measurement, a user is often interested in the relative sequence of events in different execution threads, the time of selected events and where they occurred. The motivation of the event display is to focus the user’s attention on the interactions between the processors. Communication arcs have been included in the display, in order to visually connect matching pairs of send events and receive events.

By studying the message passing characteristics of the system at this level, the analyst can better understand temporal relationships among the processes. This display
mechanism can be compared to the synchronization graph as used in Movolia [Fowler88].

Figure 3.4: Event Display : EM - mode (Shows a series of broadcasts from several processors)

In addition to the graphical representation of the events, textual information is also valuable. By clicking on an event icon, a small window with a set of associated event information pops up.

Normally, the event display is graphed with a linear time base called the True Time Mode or TTM-mode (See Figure 3.5). In a situation where there is a dense cluster of events, it is very difficult to view the graph as a series of individual events, because the events tend to overlap. To accommodate this problem, there exists another mode, namely the Expanded Mode or EM-mode (See Figure 3.4). In this mode, the time base is non-linear, but it still reflects a proper global ordering of the events. The events
Figure 3.5: Event Display: TTM - mode (Also shows a series of broadcast, but this is not clearly seen)
graphed in EM-mode are non-overlapping, allowing the analyst to study the execution history in more detail and communication anomalies such as unmatched send/receive communication is more likely to be caught.

In the bottom graphs of figure 4.7 and figure 3.8, the TTM-scrollbar at the bottom of each of the event displays represents the entire duration of the recorded program execution. The scroll box inside the TTM scroll bar indicates both the time position and time portion that is currently shown. In an EM graph, the scroll box and the scroll bar are strictly used as a mean for scrolling, representing neither time position nor time portion (See top of figure 4.7 and figure 3.8).

The icons chosen to represent process events are simple, but intuitive. The user-specified events are simply drawn as a square with a character tag (See Figure 4.6). (The tag is specified by the user when the probe is inserted into the source code(e.g. probe('s', "some user specified event")). In this way, it is not necessary to specify an icon for each user-defined event. The icons are easily associated with the event that it represents. This event representation is in contrast to the representations used in the event display tool, JED, developed at the University of Illinois. In JED, each event is represented as a sophisticated bitmap.
Figure 3.6: Event Display showing user defined events.
Figure 3.7: Event Display: TTM-mode versus EM-mode (Initialization phase).
Figure 3.8: Event Display: Overlapping of Event Icons in the TTM-mode (Below) versus non Overlapping of Event Icons in the EM-mode (Above)
3.3.2 Weighted Critical Path Analysis

When trying to improve the performance of a program, it is helpful to identify the path through the program that consumed the most time. A performance analysis method called weighted critical path analysis (WCPA) was developed and implemented in TMON [Jiang90]. The critical path identifies the parts of the program responsible for its length of execution. Reducing the length of the critical path will result in faster execution.

The weighted critical path analysis is incorporated into the event display, allowing the analyst to conveniently study the critical path. Events and communication cross edges that are found along the critical path are highlighted (See Figure 4.4).

Our method differs from the technique used in IPS [MiYa87]. IPS uses a program/procedure/function call graph to represent the critical path of a program execution. The critical path is presented textually, i.e., as a list of program/procedure/function calls. This approach does not allow the analyst to approach the critical path analysis as an incorporated part of the program execution as it does not show the critical path in context to the rest of the execution.

3.3.3 Topology Display and Dynamic Sampling

The topology display is another major component of the TVIEW visualization environment. Initially, the topology display reflects the network topology of the transputer, i.e. how the transputer nodes are interconnected (See Figure 4.9). Each node is associated
Figure 3.9: Basic Topology Display
with an information window that will pop up when a node is chosen. This information includes the name of the current process along with its process id, the current utilization of its four links and its cpu-utilization.

![Figure 3.10: Mapping of Statistical Information](image)

On top of the topology display, there is the dynamic statistical analyzing facility (See Figure 4.10). Displays of processor utilizations, network utilizations, load balancing and other system characteristics, are more beneficial when mapped onto a topology display, compared to showing detailed performance data for each individual node. As the user replays the recorded computation, different statistical information is dynamically shown for each node. Each nodal entity in the display contains a series of evolving vertical lines. When TVIEW is running, a new vertical line is drawn at regular intervals, its height representing some specific statistical result. The result can either be a percentage or count value. In this way, each node in the topology display maintains a bar chart, showing how different system characteristics evolves over time.
The nodal entities use a scale to graph the statistics by drawing a number of horizontal lines. Each line represents a new increment. This is a similar approach as found in the xmeter tool that exists under the X-window system. Each time a graph is updated, the current value of the statistics and the scale are examined. When the current value of the statistics exceeds the scale, a horizontal line is drawn, and the nodal scale is incremented. The possible scales available for percentage values are 25, 50 and 100, indicated by 0, 1, and 2 lines respectively. As seen in figure 4.10, a horizontal line is drawn across node 0, indicating that the current scale of node 0 is 50%. In the case of count statistics, the scale has to be set according to their expected values.

The dynamic display of statistical information can be used to investigate how different factors affect the behavior and performance of a program execution over time. Statistical data that are useful in this context include:

- utilization statistics (cpu and link utilization).
- event statistics (frequencies of specific events).
- memory load statistics.
- I/O load statistics.

Utilization Statistics Displays the computational activity of the multicomputer, where the statistics graphed is the percentage of non idle time for the specific node. This display is particularly useful for identifying a balance or imbalance of activity within
the system. In particular, it is possible to detect serial phases in the execution, i.e. when only a few nodes are active while others are inactive. A serial phase may indicate a potential sequential bottleneck in the parallel application, degrading the performance of the program. By detecting the serial phases, the corresponding program code can be optimized to reduce the impact of the sequential bottleneck and thus improve the overall performance.

In the display showing utilization statistics, rectangles along the communication channels are added, and the channel utilization is displayed in the same way as the display of statistic information of the nodes. As the user replays the program, a bar chart appears within each node and channel entity, reflecting cpu-utilization and channel utilization respectively.

**Event Statistics:** As mentioned, to decide on which sections of the program that need to be more efficient, it is of importance to identify the sections that are frequently being executed. In the event statistics display, the graph maintained by each node reflects the frequency of a particular event during the execution (See right side of figure 4.10). That means that the height of the vertical lines inside the nodal entities is directly proportional to the number of events occurred in a predefined interval. Event statistics that may be of particular interest, is the I/O activity. I/O activity is slow and may seriously degrade the performance if not managed carefully.

By studying the set of dynamic statistics display, the performance analyst will get a better understanding of how different transputer nodes spend their execution runtime,
and thus be able to spot bottlenecks or unnecessary overhead.

### 3.3.4 Message Passing Display

![Message Passing Display](image)

Figure 3.11: Message Passing Display along with an Event Display. The Message Passing Display reflect the communication patterns more clearly than the Event Display.

The message passing display, which is inspired from Seecube [Couch88], allows the user to look at message passing between nodes. It ignores routing. This can be helpful when deciding on an optimal mapping of the processes onto the processors in the network. Ultimately, one want processes that communicate frequently with each other to be placed close to each other, and processes that do not exchange many messages to be placed
further apart.

The display is organized as a two-dimensional matrix, where the source of each pair of nodes is indicated by row, and the destination by column (See figure 4.11).

In the two-dimensional matrix, a set of NxN bar charts are dynamically maintained, where N is the number of transputer nodes used by the application. For example, the bar chart maintained in the cell at row 3 and column 2 represents the number of messages sent from node 3 to node 2 over a period of time. Each bar represents some number of messages sent within some predefined interval defined by the system. While the clock is running, the bar charts in the message passing display, reflects the message activity between the pairs of nodes.

3.3.5 Summary

A multiprocessor is running at optimal performance if all the processors are kept busy doing useful processing. However, it is very difficult to achieve optimal performance throughout the program’s run-time. Factors that degrade performance are:

- interprocessor communication overhead,

- lost efficiency when one or more processors are idle over some time domain, and

- I/O - overhead.

The displays developed in TVIEW, allow the performance analyst to focus on several system characteristics that may affect the performance.
Chapter 4

Design and Implementation

TVIEW was developed using the object oriented programming language C++ [Strou87] along with the InterViews Toolkit [LinVliCal89]. InterViews is an object-oriented toolkit built on top of the X-window system (See Figure 5.1). TVIEW runs on a Sun 4 under Unix and consists of about 10,000 lines of C++ code.

When developing TVIEW it was important to design it so that it facilitated maintenance, updating, and expansion. Fortunately, using an object oriented programming paradigm led to a well defined modular design. In this chapter we outline the design and implementation strategies used to accomplish our goals as stated in Chapter 1. First a
CHAPTER 4. DESIGN AND IMPLEMENTATION

small introduction to the concept of object oriented programming is given, along with a brief presentation of the InterViews toolkit.

4.1 Object Oriented programming methodology

If using an object oriented programming methodology, the programmer will focus the program development around objects and their behaviour. An object is defined in terms of some private data structures and the private and public manipulation of these data structures. Objects provide a good abstraction mechanism that facilitates both data hiding and encapsulation.

An important feature of an object oriented language is inheritance. Inheritance allows properties and methods of a base class to be automatically propagated up through sub-types and instances of that base class. By constructing an appropriate set of base classes, a lot of source code duplication can be avoided. The inheritance mechanism supported by C++ also allows subclasses to redefine methods already defined in the parent class. This is done through the use of virtual functions. This ability to override the definition of a function was used often in TVIEW.

4.1.1 InterViews

The use of a window toolkit eases the development of a graphical user interface, as it allows the programmer to use the window system at a higher and more abstract level. Toolkits have built in user configurability and built in code for interaction with the
window manager and the operating system. It automatically handles window resizing and window movements. System events are filtered, and only higher level events are passed on to the application.

InterViews is an object oriented toolkit on top of the X-window system. It offers a rich set of composition strategies along with a variety of predefined components that allow the programmer to easily develop and implement complex user interfaces. Three different class categories are supported by InterViews; the interactor base class, the graphic base class and the text base class.

An interactor derived from the interactor class manages some area of potential input and output on the workstation display. Examples of interactors are buttons, scrollbars, and menus. Interactors can also be composed into what is called a scene. A scene provides mechanisms to compose interactors, i.e., when composing a control panel that consists of several buttons, the scene class supports operations to organize the buttons within the panel. Because a scene is itself an interactor it also handles events, along with distributing its input and output among its components.

The graphic base class provides the programmer with mechanisms to handle a graphical output. Functions to draw objects, erase objects, scale objects, etc. are available.

4.2 Decomposition Mechanisms

As shown in figure 5.2 TVIEW consists of three higher level objects. Two of them are composed of interactors provided by the InterViews package.
The three categories into which they are divided are:

- **Display Views**: The display views take a set of raw data (provided by the associated display manager) and display them as specified.

- **Display Managers**: The display managers are the tools that analyze the performance data. The display manager ensures that its specific display view is supplied with the data it needs, in the format that it requires.

- **Data Managers**: The data managers administer different sets of data, including manipulating the trace files. As the trace files are usually large, it is impossible to store all the data in local memory. As a result, the Data Managers cache this data.

The objects in the system are defined by their functions, i.e. by their public interface. The graphical display objects all provide a set of functions that are uniquely defined in the
system. This set of functions includes \texttt{Reset()}, \texttt{PopUp()}, \texttt{Update()}, \texttt{ClockUpdate()} and \texttt{Handle()}. The functions defined on the objects form the characteristics of the system.

The functions mentioned above apply mostly to the display managers, whose actions are the most visible to the user.

- \texttt{PopUp()}, this function initializes the graphical display manager and inserts it into the window management.

- \texttt{Update()}, if a window has been obstructed by another window or resized by the user, the display view has to be redrawn. \texttt{Update()} is automatically called when the window has been exposed to external damage or change.

- \texttt{ClockUpdate()}, this function implements the dynamic behaviour of the display. As the time passes, this display is periodically updated to reflect the change of the computation characteristics during the given time interval.

- \texttt{Reset()}, resets the display so that the user may replay the program execution, possibly with other display constraints.

- \texttt{Handle(Event e)}, the \texttt{Handle()} function implements the specific actions associated with a user event. Different events may trigger different actions. Typical actions are to change the state of the display or to pop up additional information upon user request.
4.3 Implementing the Dynamic Mechanisms

The specific dynamic behaviour of a display is defined by the mechanisms it uses to reflect the change of the system states at each update interval. The clock ticks occur at predefined intervals, and for each tick, new data and system states and characteristics are displayed.

Each of the dynamic views are derived from the class DynamicBaseView. This class implements the dynamic mechanisms of a dynamic view (See Figure 5.3). This is realized by the function ClockUpdate() defined in the base class as a virtual function. The function ClockUpdate() is redefined in each of its derived classes, so as to implement the specific dynamic behaviour of the derived class.
Upon creation, the view is automatically added to a viewlist, and for each clock tick, the list is traversed to 'ClockUpdate()' all views on the list. Viewlist is a pointer of class DynamicBaseView. The 'ClockUpdate()' function is uniquely defined by each view, and implements the dynamic behaviour of the corresponding view.

The class DynamicBaseView also maintains a Reset() function that resets the system display state.

4.4 Event Control Mechanisms

TVIEW is an event driven application that must handle a variety of different events. The user may choose item(s) from a menu, click on buttons or display specific targets, resize or move a window, etc. At the top level in the main program there is a standard event loop that is used to manage the event stream. This can be implemented as:
while ( alive )
{
    Read(event);
    if (event.type == DownEvent)
        event.target->Handle(event);

    if (quitButton->GetValue())
        alive = false;
}

Figure 4.5: General Flow

InterViews handles the lower level event management. Events are captured by TVIEW by the Read(event) function. The event is automatically associated with the target object, and the application passes the event on to it (See Figure 5.5).
4.5 Trace Control

The trace control manages the trace file. All the events from all the nodes are merged into a single trace file. A set of trace pointers are maintained within the trace file so that events from each node can be quickly accessed. Functions to read the trace file, to search for particular events and to match message transmission-events with a message-reception events are implemented. However, returning to the trace file whenever we need to search for some event is time consuming as the trace file is large and disk access is slow. Therefore, events for each process running on the different transputer nodes are cached. For each transputer, a CACHE-SIZE number of events are read into a data structure called the event cache. This avoids accessing the disk in the majority of situations. This is especially true when scrolling backwards in time.

The cache is implemented as a a two-dimensional array of trace entries(See Figure 3.2).

\[
\text{TraceEntry Cache}[\text{MAX\_NO\_NODES}][\text{CACHESIZE}];
\]

The cache is controlled by an \texttt{Update()} function, and whenever the current time exceeds the time domain maintained by the cache, a new set of events are automatically read into the cache.
4.6 Support for Adding new Analyzing and Display Components

In order for the tool environment to be extensible there must be support for adding new components without having to modify existing software. This problem is partly solved by using an object oriented design and programming methodology. By taking advantage of the mechanisms supported by object oriented environments, such as encapsulation, dynamic typing and inheritance, addition of new analyzing and display components to the tool environment is supported.

To add a new tool component, the programmer must create a new object that implements the desired behaviour of the component. The object must be derived from the Interactor base class provided by InterViews so that it can be incorporated into the interactive environment. When a new component is incorporated, the standard event loop will automatically acknowledge its presence so that any events that target the new component will be directed to it without modification to the source code.

If the tool component is dynamic, the object component must be derived from the class DynamicBaseView as described in section 4.3. This class ensures that the dynamic display is time consistent with the rest of the system, and it automatically implements the dynamic behaviour of the tool component.

4.7 Experience with C++ and InterViews
Both InterViews and C++ proved valuable in the development and the implementation of TVIEW. InterViews allowed the application to be abstracted from the window system. The user interface is entirely defined in terms of InterViews objects.

We used the inheritance mechanisms of C++. It made it possible to derive new classes from a set of base-classes, and create complex objects from simpler ones.

```
BaseView :

Class BaseView : public Interactor
{
    BaseView();
    ~BaseView();
    virtual int GetStatisticsData();
    public :
    virtual void Redraw();

Derived View

Class DerivedView : public BaseView
{
    DerivedView();
    ~Derived View();
    int StatisticsFunction();
    public :
    GetStatisticsData() {
    data = StatisticsFunction();
    return data;
    }
```

Figure 4.6: Inheritance and Virtual Functions.

Another useful feature in C++ is the ability to determine an object's behaviour at run-time using the virtual function declaration. This is called dynamic binding. Virtual functions can be used to define a set of operations for the most common version of a base
class. When necessary, the interpretation of these operations can be refined for particular derived classes. Virtual functions were used to implement the topology display discussed in Chapter 3. In the topology display a fixed type of view is used to display a variety of statistical information. Given the base class topology-view, which defines the display behaviour of the topology-display, it defines the virtual function `GetStatisticsData()` (See Figure 5.6). Each derived object of the class topology-view, refines the virtual function `GetStatisticsData()`, so that it implements the correct function. In this way, the function `GetStatisticsData()` is uniquely defined for each derived object from the topology-view class, while the rest of its functions are implemented by the topology-view base class.

Similarly, the class topology-view is derived from the InterViews interactor base class. Among other functions, the interactor class defines a virtual function called `Redraw()`. This function is refined in all of its derived classes, specifying what is to be drawn and how. Whenever a display view is obstructed, either by an overlapping window, or resizing, the underlying mechanisms of InterViews automatically calls `Redraw()` on the target interactor base class. This greatly simplifies event-management as the application programmer is not responsible for updating the display in response to window movement and resizing.

The syntax of C++ closely resembles C and it was not difficult to initially start programming. The most challenging part of using C++ was learning how to think in an object-oriented way. As a result, the code that was initially written essentially looked like "old" C programs, with only the major data structures represented as classes. As the object-oriented methodology grew more familiar, however, it was much easier to take
advantage of its mechanisms.
Chapter 5
Conclusion, and Future Work

The increasing complexity of programming parallel computer systems makes it necessary to provide the programmer with a set of tools that enables him to efficiently work in a parallel environment.

This thesis designed and implemented a set of basic visualization tools for analyzing the behaviour and performance of programs that execute on transputer-based multicomputers. It is our belief that TVIEW gives a useful representation of parallel program's behaviour and its performance.

TVIEW provides the performance analyst with displays that are useful for observing a load balance or imbalance within the system. The display that has proven to be the most useful is the event display. The event display presents the processor events along a global timeline. The eye can extract computing anomalies, such as message sends that are not matched with a message receive. Users can easily detect areas of special performance importance by studying the event display and its incorporated critical path analysis.
CHAPTER 5. CONCLUSION, AND FUTURE WORK

The topology display along with its dynamic display of statistical information helps the analyst to focus on possible factors that degrade program performance. Among the degrading factors on which the display can particularly focus are serial phases or imbalance in multicomputer and overloaded communication channels.

The message-passing display reflects the message passing activity across the multicomputer network ignoring routing. This permits the performance analyst to observe global relationships between nodes and processes.

One of the limitations of TVIEW, is that it only supports post-mortem analysis. This is partly true because of the limitations of the underlying hardware. There is an insufficient communication bandwidth to transmit the performance data to an external host without impacting on the performance being measured. Also, the analysis and visualization system must be able to process the performance data at the rate it is received. A solution is to limit the real time performance analysis to only concentrate on a few system characteristics such as cpu and link utilizations. Real time performance visualization is an active research area.

TVIEW is mainly targeted at moderate scale parallel systems. As the size of the multicomputer grows, the amount of performance data grows linearly, and a performance tool targeted at moderate scale parallel systems would not be prepared to practically handle all the performance data involved. Even though zooming and scrolling enables the visualization tool to present a lot of performance data, this is not a practical solution when the size of the system being measured exceeds hundreds of processors.
Bibliography


