1

**PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

**NodeView:
A profiler/debugger for parallel computers using message passing**

by

Craig Lindsey MacDonald

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
May 1996

copyright
1996, Craig Lindsey MacDonald

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa (Ontario)
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395 rue Wellington
Ottawa (Ontario)
K1A 0N4

Y 1214 . A 1981-1982

Y 1214 . A 1981-1982

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-19481-7

Canada

Name _____

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided

U.M.I.

SUBJECT TERM

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
 Architecture 0729
 Art History 0377
 Cinema 0900
 Dance 0378
 Fine Arts 0357
 Information Science 0723
 Journalism 0391
 Library Science 0399
 Mass Communications 0709
 Music 0413
 Speech Communication 0459
 Theater 0465

EDUCATION
 General 0515
 Administration 0514
 Adult and Continuing 0516
 Agricultural 0517
 Art 0273
 Bilingual and Multicultural 0282
 Business 0688
 Community College 0275
 Curriculum and Instruction 0277
 Early Childhood 0518
 Elementary 0524
 Finance 0277
 Guidance and Counseling 0519
 Health 0680
 Higher 0745
 History of 0520
 Home Economics 0278
 Industrial 0521
 Language and Literature 0279
 Mathematics 0280
 Music 0522
 Philosophy of 0998
 Physical 0523

Psychology 0525
 Reading 0535
 Religious 0527
 Sciences 0714
 Secondary 0533
 Social Sciences 0534
 Sociology of 0340
 Special 0529
 Teacher Training 0530
 Technology 0710
 Tests and Measurements 0288
 Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
 General 0289
 Ancient 0290
 Linguistics 0291
 Modern 0401
 Literature 0294
 General 0295
 Classical 0297
 Comparative 0298
 Medieval 0316
 Modern 0591
 African 0305
 American 0352
 Asian 0355
 Canadian (English) 0593
 Canadian (French) 0311
 English 0312
 Germanic 0315
 Latin American 0313
 Middle Eastern 0314
 Romance 0314
 Slavic and East European

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
 Religion 0318
 General 0321
 Biblical Studies 0319
 Clergy 0320
 History of 0322
 Philosophy of 0469
 Theology

SOCIAL SCIENCES

American Studies 0323
 Anthropology 0324
 Archaeology 0326
 Cultural 0327
 Physical 0310
 Business Administration 0272
 General 0770
 Accounting 0454
 Banking 0338
 Management 0385
 Marketing 0501
 Canadian Studies 0503
 Economics 0505
 General 0508
 Agricultural 0509
 Commerce-Business 0510
 Finance 0511
 History 0358
 Labor 0366
 Theory 0351
 Folklore 0358
 Geography 0366
 Gerontology 0351
 History 0578
 General

Ancient 0579
 Medieval 0581
 Modern 0582
 Black 0328
 African 0331
 Asia, Australia and Oceania 0332
 Canadian 0334
 European 0335
 Latin American 0336
 Middle Eastern 0333
 United States 0337
 History of Science 0585
 Law 0398
 Political Science 0615
 General 0616
 International Law and 0617
 Relations 0814
 Public Administration 0452
 Recreation 0626
 Social Work 0627
 Sociology 0938
 General 0631
 Criminology and Penology 0628
 Demography 0629
 Ethnic and Racial Studies 0630
 Individual and Family 0700
 Studies 0344
 Industrial and Labor 0709
 Relations 0999
 Public and Social Welfare 0453
 Social Structure and 0537
 Development 0538
 Theory and Methods 0539
 Transportation 0540
 Urban and Regional Planning 0541
 Women's Studies

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
 General 0285
 Agronomy 0475
 Animal Culture and 0476
 Nutrition 0359
 Animal Pathology 0478
 Food Science and 0479
 Technology 0480
 Forestry and Wildlife 0817
 Plant Culture 0777
 Plant Pathology 0746
 Plant Physiology 0306
 Range Management 0287
 Wood Technology 0308

Biology 0309
 General 0329
 Anatomy 0353
 Botany 0369
 Cell 0793
 Ecology 0410
 Entomology 0307
 Genetics 0317
 Limnology 0416
 Microbiology 0433
 Molecular 0821
 Neuroscience 0778
 Osteography 0472
 Physiology 0786
 Radiation 0760
 Veterinary Science 0786
 Zoology 0760

Biophysics 0425
 General 0996
 Medical

EARTH SCIENCES

Biogeochemistry 0425
 Geochemistry 0996

Geodesy 0370
 Geology 0372
 Geophysics 0373
 Hydrology 0388
 Mineralogy 0411
 Paleobotany 0345
 Paleocology 0426
 Paleontology 0418
 Paleozoology 0985
 Palynology 0427
 Physical Geography 0368
 Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
 Health Sciences 0566
 General 0300
 Audiology 0992
 Chemotherapy 0567
 Dentistry 0350
 Education 0769
 Hospital Management 0758
 Human Development 0982
 Immunology 0564
 Medicine and Surgery 0347
 Mental Health 0569
 Nursing 0570
 Obstetrics and Gynecology 0380
 Occupational Health and 0354
 Therapy 0381
 Ophthalmology 0571
 Pathology 0419
 Pharmacology 0572
 Pharmacy 0382
 Physical Therapy 0573
 Public Health 0574
 Radiology 0575
 Recreation

Speech Pathology 0460
 Toxicology 0383
 Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
 Chemistry 0749
 General 0486
 Agricultural 0487
 Analytical 0488
 Biochemistry 0738
 Inorganic 0490
 Nuclear 0491
 Organic 0494
 Pharmaceutical 0495
 Physical 0754
 Polymer 0405
 Radiation 0605
 Mathematics 0986
 Physics 0606
 General 0608
 Acoustics 0748
 Astronomy and 0607
 Astrophysics 0798
 Atmospheric Science 0759
 Atomic 0610
 Electronics and Electricity 0752
 Elementary Particles and 0756
 High Energy 0611
 Fluid and Plasma 0463
 Molecular 0346
 Nuclear 0984
 Optics 0346
 Radiation 0984
 Solid State 0346
 Statistics 0984
 Applied Sciences 0346
 Applied Mechanics 0984
 Computer Science

Engineering 0537
 General 0538
 Aerospace 0539
 Agricultural 0540
 Automotive 0541
 Biomedical 0542
 Chemical 0543
 Civil 0544
 Electronics and Electrical 0348
 Heat and Thermodynamics 0545
 Hydraulic 0546
 Industrial 0547
 Marine 0794
 Materials Science 0548
 Mechanical 0743
 Metallurgy 0551
 Mining 0552
 Nuclear 0549
 Packaging 0765
 Petroleum 0554
 Sanitary and Municipal 0790
 System Science 0478
 Geotechnology 0796
 Operations Research 0795
 Plastics Technology 0994
 Textile Technology

PSYCHOLOGY

General 0621
 Behavioral 0384
 Clinical 0622
 Developmental 0620
 Experimental 0623
 Industrial 0624
 Personality 0625
 Physiological 0989
 Psychological 0349
 Psychometrics 0632
 Social 0451

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of this thesis.

NodeView:

A profiler/debugger for parallel computers using message passing

submitted by

Craig Lindsey MacDonald

in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Thesis Supervisor

Director, School of Computer Science

Carleton University
May 1996

ABSTRACT

In general, parallel computers have more computational power than serial computers. However, there is an extra level of complexity for this advantage. Parallel programs require serial temporal information, as well as, spatial information.

The execution of a parallel program can produce a large volume of trace/state information. Examining such, in textual form, can provide insights into behavior, but the volume of data can be overwhelming. An alternative format casts these data in a graphical format. The main benefit of graphical visualization is that it handles spatial relationships in a more natural manner.

A tool created "NodeView" allows one to examine parallel program trace data. NodeView is simple to use and its functionality is provided through a graphical interface. NodeView provides information to help improve parallel application development.

FIGURES

Figure 1 - Paragraph's space-time diagram	22
Figure 2 - Paragraph's Kiviat diagram.....	23
Figure 3 - Paragraph's matrix view.....	23
Figure 4 - AVX trace.....	36
Figure 5 - 3D Kiviat tube	42
Figure 6 - Topology view	49
Figure 7 - Congestion map.....	50
Figure 8 - Classes and interactions.....	52
Figure 9 - Object creation order	54
Figure 10 - NodeView's communication architecture	55
Figure 11 - Node visual format	69
Figure 12 - Process view	73
Figure 13 - Buffer view	74
Figure 14 - AVX links collector.....	76
Figure 15 - AVX NInfo module	78
Figure 16 - Info format.....	80
Figure 17 - Info message receive.....	81
Figure 18 - Info download.....	81
Figure 19 - Interval timer event handler.....	83
Figure 20 - 4x4 routing map 1.....	89
Figure 21 - 4x4 routing map 2.....	90
Figure 22 - 8x8 congestion map.....	91
Figure 23 - I860 memory and process usage	93
Figure 24 - T805 memory and process usage.....	94
Figure 25 - Buffer queue usage	95
Figure 26 - Buffer worker usage	96
Figure 27 - File state usage.....	97
Figure 28 - Overloading host node.....	98
Figure 29 - NEMO system overview.....	101
Figure 30 - NEMO "1" file state data.....	103
Figure 31 - NEMO "1" buffer usage	104
Figure 32 - NEMO "2" buffer usage 1	105
Figure 33 - NEMO "2" buffer usage 2	106
Figure 34 - NEMO "2" buffer usage 3.....	107
Figure 35 - Cell driver.....	108
Figure 36 - RM1 live data	110
Figure 37 - RM2 live data	111
Figure 38 - RM2 stored data.....	113
Figure 39 - Separate link buffers.....	117
Figure 40 - Expanded Info Structure	118

TABLE OF CONTENTS

ABSTRACT	III
ACKNOWLEDGMENTS	IV
TABLE OF CONTENTS.....	V
FIGURES	VIII
1 - INTRODUCTION.....	9
1.1 - RESEARCH AREA.....	9
1.2 - SURVEY OF PREVIOUS WORK.....	10
1.2.1 - <i>Parallel Visualization Systems</i>	11
1.2.1.1 - Data Collection.....	11
1.2.1.2 - Storage.....	15
1.2.1.3 - Data Analysis.....	16
1.2.1.4 - Displays.....	20
1.3 - AIMS OF RESEARCH.....	25
1.4 - THESIS CHAPTER STRUCTURE	26
2 - BACKGROUND	27
2.1 - CONCEPTS AND RESEARCH ENVIRONMENT	27
2.1.1 - <i>Parallel Computers</i>	27
2.1.1.1 - The AVX Series 2 System.....	32
2.1.1.2 - The AVX Series 2 Tracing System.....	34
2.1.2 - <i>Parallel Computer Performance Issues</i>	37
2.1.3 - <i>Parallel Program Visualization</i>	38
2.1.4 - <i>Effective Graphical Displays</i>	42
3 - NODEVIEW DETAILS	45
3.1 - CHOSEN METHODOLOGIES.....	45
3.2 - AIMS FOR RESEARCH ENVIRONMENT.....	46
3.3 - NODEVIEW DESCRIPTION	46
3.4 - NODEVIEW USES	48
3.5 - NODEVIEW DESIGN.....	51
3.5.1 - <i>NodeView High Level Design</i>	52
3.5.2 - <i>NodeView Host Components</i>	56
3.5.2.1 - Main Module.....	56
3.5.2.2 - Stubs Functions.....	57
3.5.2.3 - NodeViews Class.....	59
3.5.2.4 - Nodes Class.....	61
3.5.2.5 - Info Class	63
3.5.2.6 - NodeInfo Class.....	64

3.5.2.7 - FileInfo Class	66
3.5.2.8 - Node Class	68
3.5.2.9 - NodeView Class	69
3.5.2.10 - ProcessView Class	73
3.5.2.11 - BufferView Class	74
3.5.3 - NodeView AVX Components	76
3.5.3.1 - AVX Links Module	76
3.5.3.2 - AVX NInfo Daemon	78
3.5.4 - NodeView Info Message Structure	80
3.5.5 - Message Transfers	81
3.5.5.1 - Info Message Receive (real-time profiling)	81
3.5.5.2 - Info Downloading (post-mortem profiling)	81
3.5.6 - Interval Timer Event Handler	83
3.6 - DEVELOPMENT IMPLEMENTATION CONSIDERATIONS	84
3.6.1 - UI Graphical Library	84
3.6.2 - System Design Considerations	86
4 - NODEVIEW EXPERIMENTS	88
4.1 - EXPERIMENT DESIGNS	88
4.1.1 - Functional Experiments	89
4.1.1.1 - 4x4 Routing Map 1	89
4.1.1.2 - 4x4 Routing Map 2	90
4.1.1.3 - 8x8 Congestion Map	91
4.1.1.4 - Intel 860 Memory and Process Usage	93
4.1.1.5 - INMOS 805 Memory and Process Usage	94
4.1.1.6 - Buffer Queue Usage	95
4.1.1.7 - Buffer Worker Usage	96
4.1.1.8 - File State Usage	97
4.1.1.9 - Overloading the Host Node	98
4.1.2 - Production Experiments	100
4.1.2.1 - NEMO	100
4.1.2.2 - Cell Driver	108
4.1.2.3 - RM1 Live Data	110
4.1.2.4 - RM2 Live Data	111
4.1.2.5 - RM2 Stored Data	113
4.2 - INTERPRETATION AND SUMMARY	114
5 - CONCLUDING REMARKS	116
5.1 - SUMMARY OF RESULTS	116
5.2 - EVALUATION OF WORK	116
5.3 - POTENTIAL FUTURE WORK	117
6 - APPENDIX	121
6.1 - NODEVIEW STARTERS GUIDE	121
6.2 - NODEVIEW SOURCE CODE	121

REFERENCES	122
-------------------------	------------

FIGURES

Figure 1 - Paragraph's space-time diagram	22
Figure 2 - Paragraph's Kiviat diagram.....	23
Figure 3 - Paragraph's matrix view.....	23
Figure 4 - AVX trace.....	36
Figure 5 - 3D Kiviat tube	42
Figure 6 - Topology view.....	49
Figure 7 - Congestion map.....	50
Figure 8 - Classes and interactions.....	52
Figure 9 - Object creation order	54
Figure 10 - NodeView's communication architecture	55
Figure 11 - Node visual format	69
Figure 12 - Process view	73
Figure 13 - Buffer view	74
Figure 14 - AVX links collector.....	76
Figure 15 - AVX NInfo module	78
Figure 16 - Info format.....	80
Figure 17 - Info message receive.....	81
Figure 18 - Info download.....	81
Figure 19 - Interval timer event handler.....	83
Figure 20 - 4x4 routing map 1.....	89
Figure 21 - 4x4 routing map 2.....	90
Figure 22 - 8x8 congestion map.....	91
Figure 23 - I860 memory and process usage	93
Figure 24 - T805 memory and process usage.....	94
Figure 25 - Buffer queue usage	95
Figure 26 - Buffer worker usage	96
Figure 27 - File state usage.....	97
Figure 28 - Overloading host node.....	98
Figure 29 - NEMO system overview.....	101
Figure 30 - NEMO "1" file state data.....	103
Figure 31 - NEMO "1" buffer usage	104
Figure 32 - NEMO "2" buffer usage 1	105
Figure 33 - NEMO "2" buffer usage 2	106
Figure 34 - NEMO "2" buffer usage 3.....	107
Figure 35 - Cell driver.....	108
Figure 36 - RM1 live data	110
Figure 37 - RM2 live data	111
Figure 38 - RM2 stored data.....	113
Figure 39 - Separate link buffers.....	117
Figure 40 - Expanded Info Structure	118

1 - INTRODUCTION

1.1 - RESEARCH AREA

The area of research for this thesis is visualization of parallel programs. First, we discuss briefly parallel computing, and then why visualization is beneficial. In general, a parallel computer is a group of processors functioning together on a single computational task. Parallel computing provides more computational power than single processor computing. However, a parallel system design is more complex for this additional power. Serial computers have one execution unit, and temporal issues are the main concerns in program design and implementation. Whereas, parallel computers have multiple execution units and we have to handle communication and synchronization issues as well.

These topological issues add a significant amount of complexity in parallel programming. Items to consider in the design and implementation of parallel programs include effective processor usage (load balancing), efficient communication methods, and effective algorithm designs. These are difficult to accomplish without proper tools.

To help produce better parallel programs, one can collect execution data. Possible data collection items include memory usage, processes start/finish times, and communication between processors. While looking at the results in a textual form can provide insights, the quantity of text produced can be overwhelming. One alternative for better understanding converts the textual data into a graphical format. Graphical visualizations can effectively display millions of values in a way which is understandable by humans. This method can be used to better understand parallel execution data, and thus parallel program behavior.

The main benefit of visualization in parallel processing is that it allows spatial relationships to be naturally assimilated. Classical techniques for understanding dynamic program behavior in a purely temporal form can overload the human mind. However, through the use of graphical interfaces using both two and three dimensional displays, parallel program behavior can be more easily understood; resulting in more efficient and easier to produce parallel programs.

1.2 - SURVEY OF PREVIOUS WORK

The history of parallel visualization systems follows an interesting path. The beginnings of parallel visualizations started by deriving displays from other fields. Gantt charts and Kiviat diagrams were used in statistical graphics, and were adapted to represent parallel performance data. The importance of dynamic performance data and relationships between events from multiple nodes inspired the development of other displays [1]. Many of these displays were based on PERT charts and activity networks. It was not until Thomas LeBlanc, John Mellor Crummey, and Robert Fowlers [2] started to define a theory of performance visualization, that tools emerged that provided sophisticated multiple display environments. The most popular of these tools was Paragraph [3].

Many projects have advanced parallel visualization technology [4, 5, 6]. They have defined the components needed and effective system architecture for parallel visualization systems. The next section will describe the system types and their components.

1.2.1 - Parallel Visualization Systems

When designing a visualization system one must understand the components needed: data collection, storage, data analysis, and display [7]. One must consider the kind of information to collect and how to accomplish it. Visualization systems can produce a lot of trace data. It is important to understand the design issues for online storage and for long term storage. After collecting the data, the type of analysis must be considered; this can range from quite simple to complex. Visualization systems are constructed to view something. One must choose from many possible display types that can be used to convey the relationships of the data collected. The next subsections discuss the issues: data collection, storage, data analysis, and displays in more detail.

1.2.1.1 - Data Collection

There is a large amount of possible data to collect in the execution of parallel programs. It is important to collect data as efficiently as possible. Any gathering of information for the use in visualization will disturb the application being analyzed in some way. The extent of this perturbation, or probe effect, is determined by the volume of information collected and the monitoring hardware, and system software used [8]. In the design of any debugging, monitoring, or visualization system, one must try to understand the effect of information gathering, so it does not affect the order of events in the target system.

There are multiple ways to collect performance data, this can be accomplished by either software or hardware methods. One way that is similar to debugging, collects data

by inserting gathering functions into the source code. As with the methods to obtain data, the type of data collected can also vary; information can be collected either by sampling or tracing techniques. In sampling (a polling technique) a value is obtained by the request of the visualization system. While, tracing, captures all occurrences of specified events in the application.

Hardware Collection

Hardware collection has been studied at various sites to capture low level events from parallel machines. In general, it has less of an effect on the program in study than a software-driven system. However, most systems do not have the necessary monitoring equipment in hardware, and not all events can be captured at the hardware level. The most common information collected by hardware include processor speed, memory usage, context switching, and cache misses. For example, IBM designed a monitoring chip for their RP3 machine to collection this type of information [9]. This chip is useful for low level collection, but application level information needs higher level events, which are collected by software means.

Software Collection

Software collection can be accomplished at different functional levels. The operating system, runtime system, or application is modified by adding small bits of code to collect the information desired. This code functions as a sensor and reports the value of a system state item at that time.

When the operating system is changed for visualization information gathering,

lower level information can be collected. This allows the collection of information such as message sends / receives, process creation / termination, page faults, context switching, memory access, and system calls. When visualizing this type of information one can gain insight into the operating system itself; and how it interacts with the computer hardware. However, additional information is needed to create visualizations to show inefficiencies in function or performance at the application level.

By changing the runtime environment, one can collect information on various runtime created queues (message arrival and departure from processor nodes, and procedure calls and returns). Through the use of animated displays this information can show areas for application code bottlenecks.

The application program itself can be modified by producing wrappers for commonly used functions or by providing functions which allows the visualization software to gain access to the system's data structures. Visualization in this way allows the viewer to verify the correctness of a program.

Application source code modification for visualization can be automated to some extent. MPTrace [10], automatically modifies the application to produce trace data at runtime. IPS [11], similarly does not require the user to change the program for visualization, any modifications are done automatically at compile time. PPVT [12], allows one to view standard events without modification, but custom events require the user to modify the source code.

Another way to collect information at the application level automatically is to provide a common communication library. This is the method taken by Paragraph [3], and its reliance on the Portable Instrumented Communication Library (PICL). This

method can collect send and receive events, and from these can calculate computation and communication statistics.

Some tools collect information at the various different levels. PIE [13], can collect data both at the kernel and user level. While, Tapestry [14] provides software tracing at the operating system and user levels, as well as, a hardware monitoring system.

Systems can collect additional data to further study computation and help to reduce nondeterminism. Instant Replay [15] records accesses to shared objects, and Bugnet [16] captures interprocess communications and periodically saves globally consistent states (all message timestamps globally agree). These methods allow later runs to follow the same execution path helping to reduce program execution nondeterminism. Nondeterminism results when program execution behaves differently with the same input data, and causes problems when trying to diagnose the source of program errors.

Some systems take different methods to collect data. IVE [17] controls the execution of an application's execution. At certain sections in a program, the user places visualization statements. IVE [17] then, by debugger methods (a table of breakpoints) can stop program execution, obtain the data, and construct visualization displays. Pavane [18], an unique system, uses a formal proof system, and produces visualizations based on formal properties of computation. The visualizations are not based on the operational structure of the program, but are mappings from the program computation states to the display.

Event Ordering Issues

It is important that the events in the captured parallel program execution data are correctly ordered, and both the collection and the analysis phase can be used in this process. Collection of data from parallel programs and serial programs differ, because of the multiple event streams. The events in the parallel streams must be combined and ordered. This is commonly done by using timestamps, and must be accurate since the analysis and display stages rely on them. Without synchronized clocks additional processing is needed to ensure proper event ordering, and thus the analysis phase may need event ordering functionality as well.

1.2.1.2 - Storage

Data storage requirements differ depending on the data's intended future use and the type of visualization produced. We must consider both the short and long term storage needs. Short term storage is used for the internal data displayed in the execution of the visualization system. Internal data can be stored in a temporary structure, where any new data received overwrites the previous. This method reduces the amount of memory used, but only the most current data is accessible.

Long term storage is used to save the visualization data for later retrieval. Basic systems can store file data in unanalyzed text or binary format. Then, a visualization can be recreated by reading the stored data and repeating the same initial analysis. However, more sophisticated data access is useful. It is informative to be able to search and sort on various keys in the data. Belvedere [19] stores data in a relational database form, allowing

users to perform SQL queries on it.

Post processing systems can reduce the amount of data stored. Raw data can be compressed using calculations and filters, reducing the total storage amount significantly. Possible processing items include calculating time scale, counting processors, and ordering messages. Real time systems, however, are limited by time constraints and less of this type of analysis can be performed.

In general, the way of storing data depends on the analysis and display functions needed [20]. SHMAP [21] uses a hierarchical graph structure to represent work to be done during display. SIEVE.1 [22] stores data in a spreadsheet format recording both dependence and trace information. IPS-2 [23] stores trace information in a Program Activity Graph (PAG). The graph displays nodes (function) and arcs (timing). Each process is contained in its own sub graph, and by combining these, global state information can be constructed.

1.2.1.3 - Data Analysis

After data collection, the data is analyzed to produce some result. The analysis performed depends on the amount of user interaction and the type of visualization system. Possible user efforts include event specification, analysis specification, display specification, and display comprehension. There is a tradeoff between the complexity of each of these. The more user input in the analysis and display components, the less mental effort is needed to understand the displayed data. Standard analysis include ordering events, calculating statistics, and forming high level events by combining low

level ones.

Event ordering

An important part of the analysis phase is to ensure that the parallel execution data events are correctly ordered. This can be accomplished in multiple ways. The type of system, the data has been collected for is an important part of event ordering.

Performance events must be ordered by physical time, while concept events can be ordered by logical or phase time ordering.

Physical time is used to order events in performance systems [24]. If all processes have access to a global clock, a timestamp can easily be assigned to each event. Physical time allows for the ordering of events, but also the elapsed time between events. The system can use these timestamps to produce a correct event ordering.

Logical ordering can be used in systems without a synchronized clock. A consistent logical ordering, follows Lamport happened before rule [25], where relationships and observable event order is maintained. This can be achieved from partial event ordering on each local node and knowledge of interprocess communication between nodes. This consistent ordering can then be used to assign timestamps.

Phase time ordering [26], a less common method, can also be used in systems without access to a synchronized clock. A user constructs a phase by specifying which events are related. The events are then ordered by their phase number. Belvedere [19] uses this approach. It analyzes similar system wide events and assigns these the same time. For example, message traffic for all processes sending a value in the same direction. This allows for a logical time order to be constructed.

Analysis Type

The analysis performed on the collected events can range from simple to complex. A simple analysis might perform data format operations. Whereas, a more complex analysis might require proper event ordering, calculating statistics, or producing higher level events by combining simpler ones.

As well, different analysis is needed depending on system type. Two commonly used are for performance systems and debuggers. In general, performance systems display calculated statistics, and debuggers attempt to show program state. However, there are systems which do not fall into these categories and perform other analyses [27].

Performance systems collect data to determine system performance metrics and produce statistics [28]. Paragraph [3] collects information on message passing, and by matching sends and receives, can calculate total message volume and count. Through additional analysis higher level displays can be constructed, which shows processor utilization and communication, and information like load balancing, topology, and communication patterns.

Parallel debuggers are more complicated than serial ones. They must properly process global state information, event ordering, and remote value access. Usually, a complete global state analysis is not performed, forcing the user to handle many local views. Good distributed debuggers need to have coordinating control between multiple nodes, this gives the user a better understanding of the overall computation. EBBA [29] is a high level debugger, which provides functionality for the user, to combine low level events into more desired high level ones. It accomplishes this by using clustering and

filtering techniques. Clustering combines low level events into higher level ones, and filtering removes unimportant data from the event stream.

Debuggers can be used to help report and eliminate possible race conditions. A race condition can occur when inconsistent events are present. This can happen when the data has unordered events, concurrent events, or variable access conflicts. TraceViewer [30] produces an event history graph, storing synchronization events of shared objects. By analyzing this data application race conditions can be reported.

An analysis can be specific to a particular system. IPS-2 [23] produces a program execution graph by critical path analysis. Performance tuning information can then be obtained by examining the paths in the graph. Pablo [31] allows the user to change the analysis performed through the use data transformation modules. These modules have a graphical representation and control the analysis by their interconnections. Falcon [32] use metrics and statistics to improve real-time program execution. Possible improvements include more effective processor usage or communications.

User Interaction in Analysis

User interactions in the analysis phase can be divided into a few stages, and deals with, events to capture, analysis to perform, and display methods. There can be multiple levels of interaction. Data collection and data display can be accomplished with little user input. However, this can cause difficulties in the analysis phase, which can result in uninformative relationships to program functionality. Whereas, in systems where events and displays can be modified by the user, the computation will more closely correspond to the user mental model.

When data is collected solely for visualization some user analysis is required. The source must be instrumented in a logical fashion to ease the display requirements of the desired information. Voyeur [33] uses user instrumentation to create application specific displays. User controlled adapters filter data, which is then passed to modelers for display. SIEVE.1 [22] parses a source program, producing a dependency trace file. From this, user-defined functions can be inserted, and application information can be collected.

1.2.1.4 - Displays

The visualization displays needed depend on the type of system being used. Systems that deal mainly with performance issues display various statistics. Whereas, debuggers capture a state in the execution of a program. Higher level systems try to show some part of application execution architecture. Between systems types, there is a fair amount of common display formats.

Displays in performance systems include communication, memory, or file transfer information. Paragraph [3] has an extensive collection of displays to show communication patterns and processor utilization. Debuggers have displays which show program architecture, such as displays for program data structures and other abstract representations of the computation in progress.

The displays used in the various systems show their information through the use of common graphical methods. The next sections discusses the most common types: program graphs, communications graphs, space-time diagrams, and statistical displays, as well as, briefly describing less common and user displays.

- **Program graphs**

Many systems display their parallel execution data by the use of program graphs. This is commonly seen in debugging and program visualization systems. In this representation nodes of the graph corresponds to program entities, and edges represent a temporal or functional ordering. Often this type of display is animated showing the order of execution of program parts and time spent at various locations. HeNCE [34], a toolkit for PVM [35, 36], displays an animated view of program execution using a program graph. The nodes in the graph change color depending on the various trace events. Schedule [37], a FORTRAN program viewer shows subroutines as nodes, and various colors and patterns are used to convey program state change. Tapestry [14], for the iPSC/2 hypercube, provides a display which shows a hypercube topological view of function call graphs.

- **Communication graphs**

Visualization systems also display parallel execution data using communication graphs. In this type of graph, processor state information is shown as a visual node (usually as a box), and communication links as lines interconnecting the nodes. It is possible to arrange the visual nodes on the display corresponding to the underlying topology of the system. This can provide insight to issues and problems regarding topology and communication patterns. Bugnet [16] and Belvedere [19], provide this type of functionality. Radar [38], a debugger for message passing systems, labels nodes with input and output ports, allowing for the display of queued messages. Then

using animation various communication patterns can be shown.

- **Space-time diagram**

Another type of communication display is the space-time diagram seen in many performance and debugging visualization systems. Usually one axis represents time and the other processor number. Message traffic is shown as lines connecting send-receive pairs. A parallel FFT algorithm is shown below using Paragraph's [3] space-time diagram. One can see that communication send/receive patterns change as the program executes.

Figure 1 - Paragraph's space-time diagram

- **Statistical displays**

Performance visualization systems usually have displays to show statistical data. The type of displays include bars, dials, charts, matrix, and Kiviat diagrams. Paragraph [3] includes many of these display types. Paragraph [3] uses Kiviat diagrams (a wheel

like display where each spoke in the wheel corresponds to a processor, and the spoke length represents processor utilization) to convey processor utilization, and matrix diagrams to show message communication.

Figure 2 - Paragraph's Kiviat diagram

Figure 3 - Paragraph's matrix view

- **Less common displays**

There are displays which are useful, but less commonly used. These include XY plots, barscope views, and memory access displays. A XY scatter plot is formed by plotting assigned X and Y data pair values to their corresponding position in the graph. The plot can then be used to determine if any data dependency exists between the X and Y values. SIEVE.1 [22] creates a number of XY plot displays. PIE [13] uses barscope views to display execution and CPU information. Shared memory machines can show access to common memory. SHMAP [21] provides a graphical display of

memory access patterns and cache history. Memory is shown as a two dimensional array, and a cell is highlighted when a memory or cache block is accessed.

- **User Displays**

The displays seen so far are useful to show information common to all parallel programs. However, some systems have capabilities for users to develop their own displays. Paraglyph has basic functionality in this area. The task feature of PICL can be used to collect application specific data and display it in a window.

Some systems provide more advanced functionality for users to create their own displays. Paradise [39] provides a complete abstract object oriented environment in which a user can produce a tool to display an event stream. Whereas, Voyeur [33] allow users to define their own routines that can be called by the system. The IVE [17] system allows users to create application displays by using visualization templates and properties. The templates describe the objects to display, and the properties the values to collect.

1.3 - AIMS OF RESEARCH

It is well known that the design and implementation of parallel algorithms is typically non-trivial. Tools that aid in the process of developing parallel software are therefore important components of parallel development environments.

This thesis designs, implements, and validates a visualization system (NodeView). NodeView uses object-oriented methods for its fundamental design. At the same it uses client/server properties to efficiently process profiling data. It is modular and expandable to provide a complete visualization system.

NodeView functionality helps decrease the complexity of common parallel programming issues. It shows node topologies and routing patterns. And it is a profiling tool that can examine: process, memory, buffer, and file usage components of parallel execution data. The collected execution data can be examined real-time or post-mortem. The real-time method allows for visualizations during program execution. However, the post-mortem method can collect data at a higher clock resolution, and can save data for later retrieval and study.

After completing the initial tool, research will continue to find useful visualization displays in three and higher dimensions. Possible areas for new visualizations will include dimension expansion of previous displays, and original displays only possible in higher dimensions. It will be important to understand what makes a good higher dimensional display, and its best visual arrangement.

1.4 - THESIS CHAPTER STRUCTURE

This section gives an overview of the structure for the remainder of the thesis. For each of the remaining chapters a brief discussion is given.

- **Chapter 2 provides more background information on issues regarding parallel visualization systems. It also provides justification for the choices made.**
- **Chapter 3 introduces the NodeView system. Items covered include an explanation of NodeView, reasons to use it, and a description of its design.**
- **Chapter 4 discusses results obtained by using NodeView. The experiments used for validation are described. To conclude, an interpretation of the obtained results are given.**
- **Chapter 5 presents a summary of the results, the goals met, and possible future work.**
- **Chapter 6 describes the NodeView starters guide and source code.**

2 - BACKGROUND

There are many issues to consider regarding parallel visualization systems. This chapter provides additional background information to help understand the concepts used in these systems. As well, the research environment, an important component of any visualization system is discussed.

2.1 - CONCEPTS AND RESEARCH ENVIRONMENT

This section discusses issues important in parallel visualization systems. Items covered include: parallel computer design, parallel computer performance, parallel program visualization, and effective graphical display. As well, the AVX system (the research environment) and its current tracing system are discussed.

2.1.1 - Parallel Computers

Generally, parallel computing can be considered a form which efficiently emphasizes and exploits concurrent events in the computing process. These concurrent events can be exploited at many different processing levels [40]. Multiple processors are the fundamental component of these systems, but other components are needed for a system to be considered truly parallel.

A parallel computing system can have different concurrency levels, which can be described in terms of granularity. Parallel system granularity defines the size of the

functional units allocated to each processor, which then dictates the number, size and complexity of the processors used. Granularity is usually grouped into three levels.

- **Course-grain parallel computers typically consist of a small number of complex processors. These systems involve concurrency at the highest program control level, such as executing different jobs on different processors.**
- **Medium-grain parallel computers balance the number of processors and their complexity. Concurrency is higher than course-grain systems.**
- **Fine-grain parallel computers consist of a larger number of simple processors. They involve concurrency at the lowest program control level, such as program instructions. These systems are limited by algorithm complexity and communication overhead problems, but provide the most potential for performance enhancement.**

The organization and distribution of memory is another fundamental issue of parallel computer systems. The two types are: loosely-coupled and tightly-coupled. A loosely-coupled memory system is a distributed or dispersed local memories across processors. Distributed computers are often formed by loosely-coupled memory, which allows the hardware resources to be more physical dispersed. A tightly-coupled memory system is a centralized global memory shared and accessible to all processors.

Loosely-coupled parallel systems communicate between processors by message passing. This communication form consists of send / receive operations between processors. Whereas, tightly-coupled parallel systems communicate by sharing variables in a global memory system. Variable sharing in shared memory is performed by specifying common locations for the communication interactions.

The inter-connection network is fundamental component in the design of parallel systems. This subsystem links the hardware resources: processors, memory units, and I/O units of the system. Possible connection networks are numerous, and include shared buses, multistage networks, crossbar switches, switch lattices, processor arrays, hypercubes, and hierarchical connections.

Performance in the inter-connected network can be described by two metrics: throughput and latency. Throughput or bandwidth is the rate of data transfer from source to destination. Whereas, latency is the time required for a piece of information to travel from the source to the destination.

These fundamental issues form the choices made in the design of parallel systems. Choices to be made include fine-grain verses course grain, global verses local memory, centralized verses distributed, and variable sharing versus message passing. To categorize the possible parallel system architectures the Flynn taxonomy [41] is widely used. This classifies systems based on number of instruction and data streams present in the parallel machine.

Parallelism in the architecture of a control-driven machine is obtained by replicating the data stream, the instruction stream, or both [41]. This results in three general parallel architecture types: single instruction stream with multiple data stream (SIMD), multiple instruction stream with single data stream (MISD), and multiple instruction stream with multiple data stream (MIMD).

The SIMD and MIMD architectures are commonly used, while MISD is believed to be impractical [40]. According to DeCegama's [42] classification, SIMD machines are further subdivided into six classes and MIMD machines into seven.

The processing units in SIMD machines execute the same instructions at the same time on different data elements. SIMD machines are divided into six categories: array processors, multiple SIMD machines, pipelined SIMD machines, associative processors, orthogonal processors, and the Connection Machine.

- A SIMD array processor consists of a single control unit managing a synchronized array of processing elements. The elements can have access to global memory, its own local memory, or both. These machines use the control processor to handle control instructions, while using the processing elements for vector and matrix instructions.
- A multiple SIMD machine is an array processor which has multiple control units. The control units share the processing elements dynamically.
- A pipelined SIMD machine is also an extension of the array processor. This design gives control units exclusive access to global memory, and the processing elements are configured as a pipeline.
- Associated and orthogonal processors perform high-speed read and writes functions using associative or content-addressable memory. These systems can perform parallel search and compare functions by word or bit.
- The Connection Machine is a SIMD array processor with extra functionality. It is a fine-grained SIMD architecture with thousands of processors. It uses an hypercube interconnection network as its primary framework.

MIMD machines have processors that can execute independently of each other. Each processor in these machines can execute different instructions on different data elements. MIMD machines are further divided primarily depending on the

interconnection network used into seven categories: bus systems, multistage systems, crossbar systems, hypercubes, hierarchical systems, switch lattices, and processor arrays.

- A MIMD bus system connects independent processors by a common bus. This design is simple, but a single bus is a source for bottlenecks and failures.
- A MIMD multistage connects independent processors by a common multistage network. The network is connected by switches to provide at least one path from any source unit to any destination unit.
- A crossbar system connects independent processors by a common switch bar. These systems possess the most inherent processor connectivity, but adding new processors are very expensive.
- A hypercube system connects independent processors by a common hypercube network. The regularity and connectivity of this system makes it an attractive approach for many applications.
- A hierarchical system connects independent processors by a common hierarchical network.
- A switch lattice MIMD system consists of a collection of identical cells arranged as a two-dimensional lattice. Each cell consists of a processor and local memory. No global memory is used, and all communications use the perimeter of the lattice. The switches can be dynamically reconfigured to allow the lattice to represent different topologies.
- A MIMD processor array consists of loosely-coupled processors connected to their neighbors in precise patterns. Common implementations to form the processor array are systolic and wavefront. The systolic array method uses a global clock in a

synchronous mode. Whereas, the wavefront array uses a asynchronous handshaking protocol. The systolic array is used when processors are numerous and simple, and the wavefront array when processors are more complex.

2.1.1.1 - The AVX Series 2 System

The AVX Series 2 is a parallel computer with a MIMD distributed memory architecture from Alex Parallel Computers [43]. Its open architecture, allows for user defined topologies. It provides back end computing for users working from UNIX workstations and DOS PC host computers. The host systems can be connect directly to the AVX or indirectly through a network. The AVX 2 has a scaleable and flexible architecture, which can been used in various computational intensive applications.

The machine's open architecture is designed for multi-user parallel processing with up to eight simultaneous users. The system building blocks are the computing nodes, and up to sixty-four of them are configurable at once. A compute node is a processing element made up of one or more processors and memory. Usually each node contains a communication processor (INMOS T805) and a computation processor (Intel I860). The T805 is used mainly for communication, but can be used for basic computation.

Each node uses a shared memory MIMD architecture internally, while distributed memory MIMD is used externally. Each node has two memory types; one used for communication in the T805 processor, and the other shared by both the T805 and I860. The shared memory stores data used by both processors, providing an intranodal communication buffer, and holds program instructions for the I860.

The AVX architecture is divided into three layers: service, communication, and application. Each layer provides hardware structures for a set of activities performed by software. Items performed include topology control and maintenance, node to node interconnections for message passing, and parallel computations.

The service layer forms the base in the AVX architecture. At this level, hardware and software merge to build and maintain the topologies of the machine. To build a topology the software instructions manipulate control processors to change crossbar switch configuration to form the pathways between user's nodes.

The AVX allows users to construct variable topologies for their applications. This is provided by the T805 processor, which provides connection to the switch matrix by four configurable communication links. The topologies chosen depends on the requirements of the user's application. Topology configuration is software controlled and is simple to use. Examples of the type of topologies that can be form are: hypercube, mesh, pipeline, and binary trees.

The communication layer forms the backbone of the AVX architecture, and implements the interprocessor communication system. All node and host data transfers use the interconnections of this layer. The hardware structure of the communication layer consists of the switch matrix, links connection each node, and the elements which send and receive data.

The highest layer in the AVX architecture is the application layer. This is where the application computation is performed. When executing programs, communication between other nodes is necessary. To do this, messages go from the application processor to the shared memory of the node. Then from the shared memory, the channels of the

T805 send the message out of the node to the topology. When considering the application layer one must consider the host machine. The host is included in the application layer by the software that forms the interface between it and the AVX machine.

The AVX uses Trollius for its operating system. It provides the basic needs for parallel computing, and was specifically designed for multicomputer applications.

Trollius uses a broad view of the multicomputer architecture. The software presents a uniform environment across both host and AVX systems. Trollius does not provide all the services of a general purpose operating system, and the host's operating system must provide the missing functionality.

A Trollius parallel system is formed by multitasking message passing nodes. There is a small kernel at each node that controls each local process. An event handle (32 bit integer) is used to synchronize two local processes. The synchronization points are used to control the message passing, and is accomplished by synchronizing events with equal handles.

2.1.1.2 - The AVX Series 2 Tracing System

The AVX provides a system for basic communication tracing. This allows one to capture send and receive events, and analysis them in a post-mortem fashion. However, this tracing allows only for capturing of high level network events. The system has to be expanded if lower layer communication tracing is required.

To use the tracing system source code modification is necessary. This can limit the tracing usefulness if all the source is not available for each component being traced.

As well, this tracing is controlled by command line programs, which are unreliable and difficult to use.

An example of application collected parallel execution data and its format is shown in Figure 4 - AVX trace. The application shows basic send / receive communication between two processes; the first process sends two messages and the second process receives them.

The trace is divided into four basic event types: start, stop, send, receive. The unique fields of each event is briefly described.

- **Start**
 Position 1 - Trace event type
 Position 2 - Time event was processed
 Position 3 - Node that produced event
 Position 5 - Process used to make event
- **Send**
 Position 4 - Node to send message
 Position 5 - Message event type
 Position 6 - Size of message
 Position 7 - Time spent in communication.
- **Receive**
 Position 4 - Node message received from

	Event Type	Time (us)	Node Id	Process Id					
Start	1	18132468	0	0	309837				
				To	Event	Size (Bytes)	Comm		
Send	6	18132528	0	1	12	80	6800	0	0
	6	18139328	0	1	12	80	64	0	0
Stop	2	18139904	0						
Start	1	18139124	1	1	309853				
				From					
Recv	5	18139185	1	0	12	80	192	0	0
	5	18139377	1	0	12	80	64	0	0
Stop	2	18139953	1						

Figure 4 - AVX trace

2.1.2 - Parallel Computer Performance Issues

It is becoming evident that parallel processing is critical in future computing systems [42]. Sequential computers are approaching the upper limit of their computational potential. This performance is determined by the maximum silicon signal transmission potential and is dictated by the speed of light ($3E7$ m/s). For example, a chip with a 3 centimeter diameter needs at least $1E-9$ seconds to transfer a signal, and therefore limiting performance to 1 GFLOP. Existing computers are approaching this limit.

Parallel computing can handle problems which are larger and process them faster. However, an extra level of complexity is added to the system to obtain this advantage. Serial computers only have one execution stream, but parallel computers have multiple. Therefore, in programming parallel computers one has to deal with standard serial temporal issues, but also topological issues, such as communication and synchronization between processors. This adds a significant amount of complexity in programming these systems.

There are many important issues in parallel program design and implementation stages. Items to consider include effective processor usage (load balancing), efficient communication methods, and effective algorithm designs. To effectively use parallel architectures developers must accomplish three important items.

- A good parallel algorithm must be chosen to solve the problem. The effectiveness of this algorithm dictates the upper bound of performance.
- An implementation of this algorithm needs to be written for a specific platform. It is

important to develop a machine specific implementation of the algorithm adapted to the features of the target hardware.

- The implementation must be evaluated and tuned to perform efficiently for varying numbers of processors and problem sizes.

Without proper tools the refinement stage of these items can be very tedious and difficult to accomplish.

The next generation of parallel computers will achieve more than 3 TFLOPS [44]. Unfortunately, several factors can limit users from exploiting this potential. Over the years, programmers have developed computer software and operating systems that exploit the use of one processor for a computational task. For parallel computing to be successful, software and operating systems need to be redeveloped using multiple processors working together. The lack of useful and accurate tools for measuring and analyzing program performance is distressing, since we are using parallel systems to increase computational performance.

2.1.3 - Parallel Program Visualization

A parallel program being a complex dynamic entity produces a lot of interrelated data; much like data collected from an experiment in a pure scientific field or a computer simulation of some process. This data can be analyzed in the hope of better understanding or producing a better system. While looking at the results in a textual form can provide insights, the quantity of text produced can be overwhelming. One alternative for better understanding is to convert the text into a graphical format. Humans are inherently better at understanding complex data in a graphical representation. In fact, according to

visualization in scientific computing [45], an estimated 50 percent of the brain's neurons are associated with vision.

In general, visualizations can effectively display millions of values, in a way, understandable by humans. Much insight is gained from seeing how components of a parallel program's execution profile visually interact. This allows for better understanding of parallel performance, and for better construction of parallel programs.

The main benefit of visualization in parallel processing is that it allows spatial relationships to be naturally assimilated. These spatial relationships distinguish serial and parallel programming. Again, serial programs require only temporal information, but parallel programs require both temporal and spatial information. Classical techniques for understanding dynamic program behavior in a purely temporal dimension can overload the human mind. However, through the use of graphical interfaces using both two and three dimensional displays, parallel program behavior can be more easily understood.

Visualization is an important component in the development and application of parallel performance tools. The positive results shown so far encourage us to continue research in this area. However, there are some challenges to overcome to produce more effective tools. To help focus new research and development areas, a recent theory has been formulated [46, 47]. The theory gives a framework of abstract principles to help guide visualization development.

One must always be careful in that the information being displayed correlates to the actual program's behavior. Completely non-intrusive monitoring tools are too expensive, but software based instrumentation is not effective unless the system is designed to be non-intrusive as possible.

Many visualization systems are fairly specialized, because the design of parallel systems are evolving and most tools are still experimental. This has reduced their success when compared to scientific visualization methods. When evaluating visualization data, a difficult challenge is to apply the correct mental models. Richard Hamming's maxim states that we use computers to heighten our understanding of various phenomenon and computation is truly only understood in some symbolic form.

There are some inherent difficulties to produce this symbolic form for parallel program visualization. Barton Miller [48], has stated that unlike physical scientists who can use reality for guidance, computer scientists work with more abstract models, and this can cause difficulties when mapping artificial constructs to a specific visualization.

A fundamental problem to overcome, is constructing useful representation of parallel data when ones needs to understand the computation that produced the data. The easiest solution is to ensure that visualization methods closely follow the model of parallel computation it represents. Even with these difficulties, concepts for effective visualizations are emerging, as well as, various systems have been through several generations.

More and more evidence is suggesting that application specific displays will be beneficial [46]. Recently, research work has been increasing to study the technology needed to generate application visualizations. The areas under research fall in three main areas.

- Use today's powerful computer systems to display sophisticated graphics.
- Provide more high level visualization methods through visualization libraries and environments.

- Allow users more control of the type of display given.

Traceview [49] is an early system that provided this type of functionality. Pablo [50] improves on these ideas and provides an environment for visualization prototyping; it allows users to develop tools for their visualization needs.

Work for the next generation of visualization is focusing on two important concerns; the design and evaluation of visualizations, and sophistication of graphical displays [51]. With the use of powerful scientific visualization packages, users can create complex 2D and 3D graphical displays [52]. These systems allow for more robust performance analysis and display capabilities.

An example of a next generation visualization is produced by extending the 2D Kiviat diagram into a 3D Kiviat tube [46]. This removes some of its inherent limitations. The 2D Kiviat display does not convey time history effectively. Whereas, the 3D Kiviat tube (Figure 5 - 3D Kiviat tube), formed by combining 2D Kiviat slices, uses the extra axis for time history purposes. When using a sophisticated package like IBM Data Explorer, a display like this can be made quickly, and a user has a great deal of control on how the data can be shown (the user can zoom into and rotate the tube as desired).

Figure 5 - 3D Kiviat tube

The main objective of visualizing parallel systems is to help improve the software development and execution of parallel programs. Obviously, writing software that executes correctly and efficiently is important. Many times, non uniform, or poorly defined communication or synchronization interfaces have made the creation of correctly functioning and efficiently executing software a very expensive process. It may not be possible to provide a language or system interface which makes the initial creation of software a lot easier, but a set of support tools can be a great help.

2.1.4 - Effective Graphical Displays

In developing parallel program visualizations we must understand the issues that make effective graphical displays: context, scaling, perception and interaction, comparison, and information extraction. These items have been studied in several books [53, 54] and are summarized.

- **Context**

One must understand the context of a visualization. This includes perspective; how the information is presented, and semantic context; the relationship between performance data and its constructs (its data structures). The subview mapping (how a portion of screen relates to program data) is an important consideration.

- **Scaling**

It is a difficult challenge visualizing a large number of processors for an extended period of time. Solutions to help include using multidimensional and multivariate representation, macroscopic and microscopic views, adaptive and controllable displays, and composite views.

- **Perception and interaction**

Understanding of a visualization tool's performance data is gained from a feedback loop. A tool must be flexible and controllable allowing users to observe spatial, temporal, and logical performance patterns.

- **Comparison**

Cross correlation between related views provide an informative method of understanding visualization data. Common methods include different perspectives viewing, successive frames (using series of images like a movie), and cross execution views (relating displays between multiple runs of a program).

- **Information extraction**

Raw data information extraction helps reduce and provide better understanding of visualization data. Commonly used methods include clustering and encoding.

Clustering of data groups related data points. Encoding conveys multidimensional data by using different color, size, shape, and spatial orientations.

3 - NODEVIEW DETAILS

3.1 - CHOSEN METHODOLOGIES

The methodologies used to produce NodeView are derivatives from various previous systems. The system collects information by software polling with consideration to application disturbance problems. It obtains information at the operating system and runtime levels. The operating system level collects lower level information, such as routing and topology data. The runtime level collects various runtime information such as dynamic message queues, memory, and process data.

The system records events in physical time ordering using timestamps for each event. Analysis is performed to reduce the total volume of data visualized, this allows the internal storage requirements to be simple and stored in the received format. The collected data, is then used to determine system performance metrics, and program state information.

NodeView uses an advance graphical display with mouse interaction capabilities. This allows one to view and select various components and obtain the desired information.

3.2 - AIMS FOR RESEARCH ENVIRONMENT

The goal of the thesis is the construction of a graphical profiling visualization system (NodeView) for the AVX parallel computer.

NodeView was developed to reduce the complexity of parallel application development on the AVX. Currently, the AVX does not have such a tool and many developers will benefit. In general, NodeView helps developers better understand parallelism issues. Specifically NodeView can be use to improve application design, implementation, and performance.

Developing NodeView for the AVX was not trivial. Presently, AVX's operating system is not completely stable and has inefficiencies. Special attention in the design of NodeView was required to properly handle these AVX idiosyncrasies.

After system design and implementation, various tests were performed to validate system functionality. The most significant test was with the NEMO system.

3.3 - NODEVIEW DESCRIPTION

NodeView is a profiler/debugger for parallel computers using message passing. NodeView uses object-oriented methods for its fundamental design. At the same it uses client/server properties to efficiently process profiling data. It is modular and expandable and uses an advance graphical interface to provide a complete visualization system. The implementation discussed here is written for the AVX parallel system. It has three main functionality levels: topology browsing, real-time profiling, and post-mortem profiling.

- **Topology browsing**

NodeView can display the topology of the nodes in the AVX system. This is useful to visualize the current topology and to ensure it is correctly formed. Topology browsing also provides routing information.

The routing information contains routing paths and congestion maps. A routing path shows the group of nodes a message must travel through going from a source to a destination node. A congestion map shows the routing pattern for the entire topology to a destination node. This information can help in understanding routing table inefficiencies.

- **Real-time profiling**

Real-time profiling can give immediate feedback on an application being studied. Items reported include time, memory, process, buffer, and file state information. This data gives insights for application improvements.

The real-time method provides a system view which relates to the behavior of an application. This allows NodeView to view dynamic application communication bottlenecks and to provide reasons for a crashed application.

- **Post-mortem profiling**

The post-mortem and real-time methods are similar, but the post-mortem method is more flexible and provides information at a finer detail level. After an application finishes executing one can download the information from each node. The execution profile can then be played back and controlled to discover congested points in the program execution.

3.4 - NODEVIEW USES

NodeView provides multilevel functionality for the AVX system in an easy to use graphical interface. Information obtain from NodeView can provide insight on how to improve the execution of a parallel program. This can include debugging a non-functional program, as well, to improve execution time of a working program.

Before the development of NodeView it was difficult to understand the current state information for the AVX system. Parallel systems are complex and there are many items to consider. When problems arise in developing an application it is difficult to pinpoint specific problems; reasons for faults are larger than in serial systems. Without help the discovery of these problems can be impossible or at least very time consuming.

NodeView helps reduce the guesswork required in finding parallel program problems.

- Basic system configuration problems can be quickly eliminated by using the topology and routing information.

Figure 6 - Topology view

A topology view is displayed by NodeView (see Figure 6 - Topology view). This can provide quick information why an application is not performing correctly. If an application is not functioning or performing slowly, a view of the topology can be useful. One can immediately see if the fault is application or system based. If the system topology is incorrectly formed the topology can simply be reinitialized. Then, the application can be reexamined to observe if the fault persists. If the fault is not topological based the real-time and post-mortem methods can be used.

Figure 7 - Congestion map

The topological view also provides routing paths and congestion maps (see Figure 7 - Congestion map). The highlighted nodes show the route used to send messages from node 15 to node 0. The congestion map (numbers on links) shows the routing paths used to communicate to a specified node (node 0). The paths used for communication is important when the number of nodes used is large. The congestion map can show which pathways are overloaded and the application design can be modified using this information to improve performance.

- Application problems can be narrowed by using the program state and communication information.

The program and communication information is provided by two forms: real-time and post-mortem. The real-time method is less detailed data, but gives information as an

application executes, and provides useful information even if an application crashes.

Whereas, to obtain the post-mortem data an application needs to finish correctly. But, the data received is detailed and allows one to study the efficiency of the various program and communication information, allowing one to produce more efficient applications.

The real-time and post-mortem methods can be used for profiling and debugging. Program state information can determine if there is a memory leak or faulty process problems. Whereas, the communication information can provide if there is buffer or buffer worker problems. AVX programs can easily overload the communication channels in the system and one needs to be informed on peak buffer usage. The communication design of the AVX system depends on buffer workers to handle message passing between nodes. This can cause problems, because the total number of buffer workers is limited. If a buffer worker is required, but none is available the system will crash.

3.5 - NODEVIEW DESIGN

NodeView uses object-oriented methods for its fundamental design. At the same time it uses client/server properties to efficiently process real-time profiling data.

NodeView has two main functional components: the host portion, and the AVX portion.

The host portion of NodeView uses an object-oriented design and is written in C++ for

Solaris 2.x. As well, it follows an event-driven architecture for the graphical interface

based on the UIT class library. The AVX Trollius portion of NodeView was implemented using a procedural design in C.

3.5.1 - NodeView High Level Design

Figure 8 - Classes and interactions

To begin the design details of NodeView, the system classes and modules along with their interactions will be discussed. On initial inspection of Figure 8 - Classes and interactions we can observe some high level design details. Classes are shown as rectangles and groups of functions as circles. Communication paths are shown by the arcs connecting the classes. These paths show how classes can interact.

When looking at a more detailed level one can distinguish different class functional groups. Structured in this way, the system is modular. This allows for easier enhancements and porting to other systems. The groups fall into three categories: user-

interface classes, information processing and storage classes, and the Alex daemon module.

- The user-interface classes and functions include NodeViews, NodeView, BufferView, ProcessView, and UI stubs. These classes form and control the graphical components of the system. Functions performed include managing windows, graphical widgets interactions, and display of trace data.
- The information processing and storage classes include Nodes, Node, NodeInfo, and FileInfo. These classes interact with the Alex daemon module to receive, process, and store trace data.
- The AVX daemon (the NInfo module) controls information collection on the AVX nodes. It obtains and stores memory, process, and communication data from the AVX. This data is then sent to the host for visualization.

Figure 9 - Object creation order

One can gain insight on system relationships by examining the object creation diagram (see Figure 9 - Object creation order). This diagram shows the order objects are created by showing parent-child creation relationships. A parent object is on the left and its child on the right. Objects are created going from left to right.

On startup, the main function creates all the UI components and initializes their callbacks. When this is finished, a NodeViews object is created. This object then creates the Nodes, NodeInfo, and FileInfo objects. The Nodes object then finds the topology of the current system by sending the links collector to each AVX node. A Node object is created to store the information received from each link collector message. Then, each Node object is linked to a NodeView object, so each node in the topology can be displayed.

Figure 10 - NodeView's communication architecture

NodeView's communication architecture is shown in Figure 10 - NodeView's communication architecture. NodeView's host component uses a multiprocess architecture to receive information from the AVX nodes. There are three processes executing on the host: the main process, the NodeInfo receive process, and the FileInfo receive process. The main process handles analyzing, storing, and displaying the visualization data. Each NInfo daemon sends Info messages from the AVX nodes to the NodeInfo receive process. This information is stored in a shared memory segment (on the host), which is used by NodeView's host main process for real-time profile visualization. The FileInfo receive process handles the collection of file information for each node. This is obtained by polling the host node and checking for current file transfers.

3.5.2 - NodeView Host Components

At this point, a detailed description of NodeView Host's classes and functions will given. The Host components of NodeView are easily ported to other systems, they are written using GNU C++ and use a public domain windowing system which is available for many platforms. Each class and function group will be discussed, as well, class structures and functions prototypes will be shown.

3.5.2.1 - Main Module

The main module initializes the system various components. On execution the main function creates all of the graphical components in the system. This includes items, such as, windows, scroll bars, menus, buttons, and various other graphical widgets (for an explanation of the graphical components see section 3.6.1). When creating these widgets, a callback stub, is linked to each. These stubs are later used to perform the desired functionality in an event driven fashion.

Examples of widget creation is shown below. Two widgets are created: a BaseWindow, and a CommandMenu. A BaseWindow displays and contains all of the graphical components in the system. A CommandMenu forms a standard drop-down menu. There are three selectable items on this menu: Open, Save, and Exit. These items are linked to the stub functions: openHandler, saveHandler, and exitHandler.

BaseWindow widget

```
bw = new BaseWindow;  
bw->initUI (argc, argv);
```

```

bw->setDisplayFooter(FALSE);
bw->setWidth (560);
bw->setHeight (595);
bw->setLabel ("Alex Nodeview");

```

CommandMenu widget

```

fileMenu = new CommandMenu;
fileMenu->addItem("Open");
fileMenu->addItem("Save");
fileMenu->addItem("Exit");
fileMenu->setItemHandler("Open", openHandler);
fileMenu->setItemHandler("Save", saveHandler);
fileMenu->setItemHandler("Exit", exitHandler);

```

After the graphical widgets are made the main function creates an XWindow's [55] event timer. This timer is called periodically, and is used to update the visualization information in the system. The timer update speed is set from the linked drop down selection menu, and can be set, ranging from 100 ms to 10 s. For more information, on specific use, please refer to the interval timer event handler section (section 3.5.6).

```

// Called each time a timer event is produced.
Notify_value updater(Notify_client client, int which);

// Initializes how fast timer events are produced.
void initTimer();

// Contains all the functionality to produce the about window.
void makeAboutWindow(Button *aboutButton);

// Called when system is started.
main(int argc, char **argv);

```

3.5.2.2 - Stubs Functions

The stubs in the system link the graphical interface components to the functionality provided by the class hierarchy. When a graphical component is activated its

stub handler is called. When activated, information is extracted from the object given (this is usually mouse position or active widget selections). This then is passed to a method in the NodeViews object, which performs the desired function.

// Shutdown the system.

void exitHandler(UIObject *object);

// Load trace data.

// Makes a open file dialog box.

void openHandler(UIObject *object);

// Save trace data.

// Makes a save file dialog box.

void saveHandler(UIObject *object);

// Activated when text in visual node is single clicked.

// Changes position of route source node.

void nodeTextSCHandler(UIObject *object);

// Activated when text in visual node is double clicked.

// Changes position of route destination node.

void nodeTextDCHandler(UIObject *object);

// Changes time position of stored data.

void timeSliderHandler(UIObject *object);

// Live/stored trace data checkbox.

void liveMsgsCBHandler(UIObject *object);

// Routing information on/off checkbox.

void nCongestCBHandler(UIObject *object);

// Stat messages on/off checkbox.

void statMsgsCBHanlder(UIObject *object);

// Sys messages on/off checkbox.

void sysMsgsCBHandler(UIObject *object);

// Button handler to download stored trace data.

void dLoadBHandler(UIObject *object);

// Activated when button clicked in main window.

// Creates BufferView and ProcessView windows.

```

void CanvasEventHandler(UIObject *object);

// Name of file to load.
void fileChooserOpenHandler(UIObject *object);

// Name of file to save.
void fileChooserSaveHandler(UIObject *object);

```

3.5.2.3 · NodeViews Class

The NodeViews object can be consider the high level dispatcher/controller object. It is the first object created after the initialization phase of the system is completed. It is responsible for the creation and control of all of the other classes in the system. When an UI event is produced by manipulating an user-interface item, its corresponding UI stub callback function is triggered. These callbacks execute methods contained in the NodeViews object, which in turn call methods in the required object.

The NodeViews object provides this functionality by containing a pointer to the Nodes object and an array of NodeView objects. The Nodes object allows for information collection from each Node, NodeInfo, and FileInfo. The NodeView objects allows for graphical display and control for each Node object. This means each visual node can be individually controlled, allowing its format to be changed as desired.

```

class NodeViews
{
    // Pointer array of NodeView objects (visual nodes).
    NodeView *nView[128];

    // Pointer to Nodes (controller of each node object).
    Nodes *nodes;

    // Component which draws graphics to the screen.
    CanvasShellDisplay *canvas;

```

```

// Destination route node.
int routeDest;

// Routing on/off.
int routingState;

// Display Live/Stored trace data.
int liveMsgs;

public:
NodeViews(CanvasShellDisplay *canvasShell);
~NodeViews();

// Initializes the NodeInfo object: shared memory, semaphores, Alex demon modules.
void startNodeInfoServer();

// Cleanup the components of the NodeInfo object.
void stopNodeInfoServer();

// Set route destination.
void setRouteDest(int nodeId);

// Display route used between source and destination nodes.
void showRoute(int routeSrc);

// Calculate congestion map.
void setRouteCongestion();

// Display congestion map on visual node links.
void showRouteCongestion();

// Remove congestion map from visual node links.
void clearRouteCongestion();

// Called each time a timer event is produced.
// Displays new Live/Stored trace display.
void timerUpdateStates();

// Switch Sys/Stat message types.
void changeMsgType();

// Receive stored trace data from Alex demon.
void downLoad();

```

```

// View real-time trace data.
void startLiveMsgs();

// View post-mortem trace data.
void stopLiveMsgs();

// Change time position of stored data.
void changeTimePos(int val);

void loadStoredData(char *filePath);
void saveStoredData(char *filePath);
};

```

3.5.2.4 - Nodes Class

The Nodes object provides the functionality to control each Node object, and the NodeInfo and FileInfo objects. The system is built upon the information contained in each Node object. The Node objects are managed directly by the Nodes object through two arrays. The first array "node" contains pointers to each node in the system. This way Nodes can access the Node objects information through the use of their public member functions. The second array "nodeIds" contain the node id number stored in the same order as in the "node" array. The "nodeIds" array is used to access the correct Node object.

```

class Nodes
{
    // Number of nodes in user topology.
    int numNodes;

    // Array of pointers to each Node object.
    Node *node[128];

    // Ids of nodes used in user topology.
    int nodeIds[128];

```



```

// NodeInfo server on/off.
int nInfoServerRunning;

// Route used from source to destination node.
NRoute nRoute[128];

// Contains the number of nodes passing through each node's four links.
NConjest nConjest[128];

// Pointer to the NodeInfo server.
NodeInfo *nInfo;

// Pointer to the FileInfo server.
FileInfo *f;

public:
    Nodes();
    ~Nodes();
    int getNumNodes() {return numNodes;}
    int* getNodeIds() { return nodeIds; }
    int getNInfoServerRunning() { return nInfoServerRunning; }
    Node* getNode(int index) {return node[index];}
    NRoute* getNRoute() {return nRoute; }
    NConjest* getNConjest() { return nConjest; }
    NodeInfo* getNInfo() { return nInfo; }

// Send routing collector to nodes.
void sendRouting();

// Receive routing collector from nodes.
void collectRouting();

void startNodeInfoServer();
void stopNodeInfoServer();

// Get neighbor and routing information for the root node.
void rootLinks(int *routeInfo, int size);

// Use the index value of the nodeIds array to find the corresponding Node object.
int findNodeIndex(int nodeId);

void findRoute(int srcId, int destId);
void calcRouteCongestion(int nodeId);

// Switch Stat/Sys message types.

```

```
void changeMsgType();
```

```
// Download trace information stored in the Alex demon.
```

```
void downLoad();
```

```
};
```

3.5.2.5 - Info Class

The Info class is an abstract base class for the NodeInfo and FileInfo class. It provides the base functionality to control the semaphores and shared memory used in its inherited classes. This includes initializing, removing, and controlling the multiprocess aspects needed.

```
class Info
```

```
{
```

```
protected:
```

```
// Id of the spawn process.
```

```
int pid;
```

```
// Semaphore.
```

```
int sem;
```

```
// Shared memory id value.
```

```
int shmid;
```

```
int nodeIds[128];
```

```
int numNodes;
```

```
public:
```

```
Info();
```

```
~Info();
```

```
void semDown(int sem);
```

```
void semUp(int sem);
```

```
// Given a node id finds its index in the nodeIds array.
```

```
int findIndex(int nodeId);
```

```
// Wait until "val" time has passed.
```

```
void timer(float val);  
};
```

3.5.2.6 - NodeInfo Class

The NodeInfo class is somewhat more complicated than the other classes in the system. We must be able to display the most current node data in a timely fashion. As well, the user interface should be usable even during the heaviest load conditions. Each node is sending data to the host at a constant rate, but it might not be dependable. We can not expect data to always be present, therefore a special purpose process is used to collect data. The specific solution used is for NodeInfo to spawn ones of its methods, which is used as a message receive loop.

The NodeInfo process controls each of Alex NInfo modules. There is functionality to start, stop, change state, receive, and download information from each Alex NInfo module. Most functionality of the Alex NInfo modules is control from the NodeView main process. This results since usually the access is quick and does not need the added complication associated with multiprocesses. However, as stated above, the receive message component needs the spawn process capabilities.

All visualization Info data is stored in this class. The live data is stored in the shared memory segment where both processes can access it. Each data message obtained is not stored and new information writes over the old. The stored Info data structure is very similar, but is expanded to handle multiple Info structures, stored in increasing time. The stored Info data is kept in standard memory array of NInfo structures each array entry has a pointer to the next, making it a link list stored by time. A current time position

is kept for each node, which is the data to used for display purposes.

```
class NodeInfo : public Info
{
    NInfo *nInfo;
    struct bfstate *bfstate;
    struct bfparms *parms;

    // Process Ids of each Alex demon.
    int tpids[128];

    // Stat/Sys message type.
    int msgType;

    // Stored trace data structure (array of link lists).
    NILink *niLink[128];

public:
    NodeInfo(int *nlds, int nNodes);
    ~NodeInfo();

    // Get real-time data for given node.
    NInfo* getLiveState(int nodeId);

    // Get post-mortem data for given node.
    NILink* getStoredState(int nodeId);

    // Set stored data time index to 0.
    void resetStoredState();

    // Set stored data time index to "val".
    void positionStoredState(float val);

    // Advance time position by one for given node.
    void advanceStoredState(int nodeId);

    // Erase all stored (post-mortem) data.
    void deleteNILinks();

    // Start receive process.
    void startUp();

    // Stop receive process.
    void shutDown();
}
```

```

// Initialize trace data values in shared memory segment.
void clearNInfo();

// Set Stat/Sys message type.
void changeMsgType();

// Start Alex demon on each T805 processor.
void loadT805Processes();

// Remove Alex demon from each T805 procesor.
void removeT805Processes();

// Get store trace data from Alex demon.
void downLoad();

// Get memory, process, and communication information for host node.
void hostInfo();

// Spawn message receive method.
void startLoop();

// Load post-mortem data in to system.
void loadNILinks(char *filePath);

// Save post-mortem data to file.
void saveNILinks(char *filePath);

void statMsgsOn();
void statMsgsOff();
};

```

3.5.2.7 - FileInfo Class

The FileInfo class is structured as seen in the NodeInfo class, it is inherited from the Info class, and uses a multiprocess structure to collect and process information. Processing file state information in the AVX system is handled slightly differently than in the NodeInfo class. It uses the multiprocess structure from the Info class, but the host node provides all the data needed for each node, that is, each node it not probed to obtain

the data. The host keeps a record of all current file transfer, and processing the information, each specific node transfer state can be calculated.

From a high level it is not apparent why a multiprocess structure is needed for this class. Because of the AVX system structure, file state information is not properly handled if the same initialized process is used for message receives as well. This forces file transfers to be controlled by a separate process. That is, the file information collector has its own kinit function to connect to the trollius kernel. In general, if file transfers and messages are handled in the same process (kinit) file transfer information is updated only when message transfers are not being handled, this reduces the file state information detail.

```
class FileInfo : public Info
{
    // Number of nodes transferring files.
    int ftableSize;

    // Link list of AVX file transfer information.
    struct fstate *ftable;

    NFil * nFil;

public:
    FileInfo(int *nlds, int nNodes);
    ~FileInfo();

    // Spawn file transfer probe process.
    void startLoop();

    // Get file transfer information from shared memory for the given node.
    NFil* getState(int nodeId);
};
```

3.5.2.8 - Node Class

The Node class handles the lowest level of the system's information. It holds the records for each node in the system. Each Node object stores neighbor link and routing information. The link information is used to produce the topology of the current system, and the routing information to produce the routing tables and congestion maps.

Each Node object contains a pointer to the NodeInfo and FileInfo objects. This allows direct access to the visualization data for that node. This information is then accessed when higher level objects request the current node to obtain visualization data.

```
class Node
{
    // Id of the node.
    int nodeId;

    // Ids of neighbor nodes.
    int links[4];

    // Routing table.
    int routes[128];

    // Node has been position in topology true/false.
    int processed;

    // Pointer to the NodeInfo server.
    NodeInfo *nInfo;

    // Pointer to the FileInfo server.
    FileInfo *f;
public:
    Node(int nId, int *lnks, int *routes);
    int getId() {return nodeId;}
    int* getLinks() {return links;}
    int* getRoutes() { return routes; }
    int getProcessed() {return processed;}
```

```

NodeInfo* getNodeInfo() { return nInfo;}
FileInfo* getFileInfo() {return f;}
void setProcessed(int val) {processed = val;}
void setNodeInfo (NodeInfo *ni) { nInfo = ni; }
void setFileInfo(FileInfo *fil) {f = fil;}
};

```

3.5.2.9 - NodeView Class

Figure 11 - Node visual format

NodeView has many variables (pointers to UIT class widgets) to form the graphical look for each node. This includes arrays for the background and foreground boxes, an array for the node text, and an array for the hot regions. The background boxes forms the empty node view, whereas the foreground boxes displays visualization data,

and changes whenever there is an update. The node text is used to show total buffer usage, and congestion map information. The hot regions are used to select either a process or buffer view pop up window.

The layout of each NodeView is as follows (see Figure 11 - Node visual format). On the top left, outside the NodeView area is the total number of buffers processed on that node. This is the total number of buffers polled from the buffer demon. Since this is a polled value, there is the possibility to count messages more than once (the same message may stay in the buffer for more than one poll cycle), but the total number is not important as the relative total. From this we can see which nodes are under the heaviest communications. Usually, this corresponds to the nodes that are on the main routes. This is important because, these nodes have a higher chance of crashing (when buffers are exhausted).

The first bar (top left) contains information for the T805 and the second bar (top right) for the I860. Two items are displayed in these areas. A green bar which goes from right to left shows the percentage of free memory. The display shows that currently the T805 has 90% free memory, while the I860 has 85% free memory. Going from left to right each vertical smaller blue line shows the number of processes running on that processor. This can many for the T805, but only one for the I860 (a square is used for easier viewing).

The third bar shows buffer information. Going from right to left shows the percentage of free buffer demon memory. The display shows that currently the buffer demon has 45% free memory. Going from left to right each vertical line shows the number of buffers currently in the demon. Buffers in blue are blocked messages, which

are being handled by a buffer worker. Buffers in red are being held in the buffer demons memory. The display shows that there is a total of 30 buffer messages present. The bar is setup this way to allow one to determine the size of the messages without looking at any other displays. Buffer bars with a lot of red and green means smaller messages, but bars with a lot of red and little green means large messages.

The fourth bar shows buffer workers. Going from right to left shows the percentage of free buffer workers. The display shows that currently 90% of the buffer workers are free. To understand the buffer work bar ones must understand how messages are processed in the AVX system. Each message type in the AVX must be handled by a different buffer worker. Applications with many messages types, could possibly run out of buffer workers, which can cause the system to crash. This is important for all nodes (usually 32 buffer workers), but is most important on the host (only has 8 buffer workers), because it has fewer buffer workers.

The fifth bar shows file state information. Going from left to right shows the total amount of bytes transferred by the currently open file. The display shows that 256 kilobytes has been transferred.

```
class NodeView
{
    // Contains all graphical widgets.
    CanvasShellDisplay *canvas;

    // Pointer to the corresponding node object.
    Node *node;

    // Outer node position value.
    int ncBaseX, ncBaseY;
```

```

// Rectangle for node.
CSDrawRect *nodeRect;

// Text for buffer total and congestion map.
CSDrawText *nodeText[7];

// Background value bars.
CSDrawRect *bgRects[5];

// Clickable region.
// Makes ProcessView and BufferView for node.
HotRegion *hr[2];

// Foreground value bars.
// Displays visualization data.
CSDrawRect *valRects[9];

// Link lines connecting nodes.
CSDrawLine *links[4];

// Inner node position value.
int nvBaseX, nvBaseY;

// Determines location of text and bars in node.
int nvSize, unvSize, linkSize;
int bSpacing, tHeight;
int rectsStartY;
int unvX, unvY;
int bgRectHeight1, bgRectHeight2, bgRectWidth1;
public:
NodeView(CanvasShellDisplay *canvasShell, Node *n, int x, int y);
int getX() { return ncBaseX; }
int getY() { return ncBaseY; }
CSDrawText* getNodeText(int val) { return nodeText[val]; }
CSDrawRect* getNodeRect() { return nodeRect; }
void setXY(int x, int y) { ncBaseX = x; ncBaseY = y; }

// Calculate node text and bars positions.
void calculatePosVals();

// Draw node.
void showNode();

// Draw link lines.
void showConnections();

```

```
// Display visualization data.
void showLiveBufferState();
void showLiveProcessState();
void showStoredBufferState();
void showStoredProcessState();
};
```

3.5.2.10 - ProcessView Class

Processes for Node: 2				
Index	Pid	KPri	KState	Program
19	257533	0	8	<unknown>
20	422545	0	8	r5
21	11945985	0	8	r5
22	14964897	0	8	r5
23	16483713	0	8	r5

Figure 12 - Process view

By clicking on the T805 process and memory bar, a current process view is created. This view allows one to view all application processes running on that node. Processes are stored by their pid value (a 32-bit integer). The I860 always uses a pid of 860, whereas the T805 can be any other integer value. Various other information is displayed here, such as, index, priority, state, and program name.

```
class ProcessView
{
    // Id of node.
    int nodeId;

    // Number of processes on node.
    int tableSize;
```

```

// Link list of processes for node.
struct pstate *ptable;

// Window for process view.
PopUpWindow *proWin;
public:
    ProcessView(int nld);
    ~ProcessView();
    void setNodeId(int nld) { nodeId = nld; }

// Make window and show processes.
void displayProcess();
};

```

3.5.2.11 - BufferView Class

Node: 1 Number: 10 Mem Used: 40k			
Event	Type	Length	State
14	0	4096	blocked
14	0	4096	waiting
14	0	4096	waiting
14	0	4096	waiting
14	0	4096	waiting
14	0	4096	waiting
14	0	4096	waiting
14	0	4096	waiting
14	0	4096	waiting
14	0	4096	waiting

Figure 13 - Buffer view

By clicking a node's buffer bar, a current buffer view is created. This view displays information relating to the current queued buffers on that node. The message event value, its size, and state are given. On the title bar the total number of messages and the total buffer memory usage is shown. Messages waiting in a buffer worker are shown

as blocked, and messages in the buffer demon's memory are shown as waiting.

```
class BufferView
{
    // Id of node.
    int nodeId;

    // Number of messages in buffer.
    int bftableSize;

    // Link list of buffer information.
    struct bfstate *bftable;
    struct bfparms *parms;

    // Window for buffer display.
    PopUpWindow *bufWin;
public:
    BufferView(int nId);
    ~BufferView();
    void setNodeId(int nId) { nodeId = nId; }

    // Make window and show buffers.
    void display();
};
```

3.5.3 - NodeView AVX Components

NodeView contains two specific AVX modules: the Links and the NInfo modules. These modules handle the collection of AVX information for NodeView. These are the main modules that have to be rewritten when porting NodeView to other computers, and this is easily done because they contain only about 500 lines of code.

3.5.3.1 - AVX Links Module

Figure 14 - AVX links collector

The Alex links collector (see Figure 14 - AVX links collector) executed on each node (sent by the Nodes object), collects information on the current user topology. On execution, each node finds, and then records neighbor links and routing information. This

information is stored into an array of integers. When finished this array is sent back to the host. Position 0 in the array contains the node id and is used as in the NodeInfo receive process.

The neighbor node ids are stored in this array at positions 1 to 4. Each node in the AVX system can have up to four neighbors. The position entries uses a square layout: position 1 (bottom), position 2 (top), position 3 (right), position 4 (left). Each node neighbor link is checked and the connecting node id is stored. If a link is not connected to another node a unconnected value (-1) is stored in that position. For example, for node 2 in a 2x2 mesh has [-1, 0, 3, -1]. These values show that node 2 is on the lower left hand corner of a square.

From the neighbors information one can construct the user's topology. The algorithm functions as follows. It starts with node 0, and places it at the first empty node view position. Then, for each node in the system, it loops through each of its links obtaining the corresponding neighbor node, and calculates its correct node view position. As each node is processed, it is marked, so it will not be processed again. This continues until all nodes have been processed. It would be possible to display a 3-D topology with this method, but the system would have to be expanded in some areas to support it.

The node's routing information is stored from array position 5 to the end of the array. The number of positions stored is not fixed, but depends on how many nodes are in the system. The routing tables in the AVX system are not complicated. Each node contains an array with a position for each node in the system. At each node position the link to send the message through is given. Transferring through this link will move the message one node closer to its destination. Then the same procedure is used at the new

node until the message reaches the final destination. For example, for node 2 in a 2x2 mesh has the routing list (to node: link number) [0:1, 1:2, 2:-1, 3:2, 100:1, 101:1] stored in array positions 5-9.

3.5.3.2 - AVX NInfo Daemon

Figure 15 - AVX NInfo module

A fundamental part of the NodeView system is the NInfo daemon running on each node of the AVX System. This daemon executes on the T805 processor, collecting and processing information for each node.

When executing, the daemon performs its functionality in a cyclic nature. After initialization the system enters the timer loop. The system runs this loop until the module is stopped or a signal request is received. The loop visualization information is obtained from four functions: time, memory, process, buffer. Each function collects information related to its name. For example the time function gets the current time on the local node.

On each cycle of the timer loop the data collected is stored in an Info linked list structure. The timer resolution of this data is changeable, but usually the data is stored at a 200 ms timer value. After storing the current data, the Info message variable is checked to determine if a message should be sent to the host. An Info message is usually only sent for every 10 timer loops. This method allows detailed post-mortem data, but also real-time information that does not effect the AVX application being studied much.

The NodeInfo object sends AVX signals to control the daemon. The AVX signals are similar to standard UNIX signals, and allow for user defined signal handlers. When the daemon receives an AVX signal the timer loop stops, causing a signal specific function to be executed. The signals used, allow for message and timer changes, and for downloading of the node visualization data. When the signal code finishes the previous execution position in the timer loop is continued.

3.5.4 - NodeView Info Message Structure

Figure 16 - Info format

The messages received from the AVX nodes are 16 double words (64 bytes), each position in the array contains specific node information (node file state information is handled differently see FileInfo class description). The first position in this structure contains the node id for that message. Each message is stored in the host shared memory segment depending on this node id. When NodeView's host NodeInfo receive process, receives a message it examines the node id value and stores the rest of the data structure in the corresponding shared memory location. The node id is not a direct index into the shared memory array. Instead the index used is the node count. For example, in a 4x4 mesh the host node id is 100, but its count id is 16, and node id 101 is count id 17.

3.5.5 - Message Transfers

3.5.5.1 - Info Message Receive (real-time profiling)

Figure 17 - Info message receive

The Info message receive procedure is used to send and received visualization data for the real-time profiling method on each node. An Info message is sent from an Alex NInfo process and is received by a NodeInfo receive process. Each time a message is received, it is put into an array of shared memory NInfo structures by the NodeInfo receive process. This allows the NodeInfo main process to probe the shared memory at any time without waiting. The NodeInfo main process can always read the most recent data obtain, even if the system has crashed or stopped sending messages. In this way, the system is guaranteed to operated properly.

3.5.5.2 - Info Downloading (post-mortem profiling)

Figure i8 - Info download

The Info downloading transfer method is used to receive visualization data for the post-mortem profiling method. It is similar to the Info message receive procedure, but has been modified to handle large data volumes more efficiently. When the user clicks on the download button on the graphical display, the download function in the NodeInfo main process is executed. This causes all nodes to stop collecting data, and wait for downloading. Each node is sequentially requested to download its Info structures to the host node.

In the AVX system the number of messages processed is more of a problem than its size. To provide further stability, messages are sent batched in 4096 byte packets (64-64 byte messages). This method decreases the number of messages in the system by a factor of 64. This was necessary, because if all nodes sent large trace files at the same time, the system can become overloaded and crash. When received, each packet of Info structures is decoded on the host and saved in the stored Info list for that node. A side benefit of this method is that it increases the download transfer rate from about 10 KB/sec to about 150 KB/sec.

Once all the nodes have finished the download process, one can playback the stored node information. When using the stored information one can use the slider on the NodeView main window to position execution time. On play back, each timer event causes the current Info structure position to be advanced to the next in the link list.

One deficiency with the post-mortem Info playback method, occurs with nodes under heavy loads losing time synchronization. This happens because the functions called in the Alex NInfo module can not return quick enough. This is most noticeable at the 100 ms resolution, and disappears at the 500 ms resolution.

3.5.6 - Interval Timer Event Handler

Figure 19 - Interval timer event handler

The profile visualization display procedure is performed by NodeView's interval timer event handler (see Figure 19 - Interval timer event handler). This procedure is used to display the visualization data for each node. This execution path is called each time a timer event is produced from the UI timer component. On execution the timerUpdateState method in the NodeViews object is executed. This in turn executes the showBuffer and showProcess methods in each NodeView object. The NodeView object then calls its corresponding Node object to obtain the visualization data from the NodeInfo and FileInfo objects.

The way the data is obtained from the NodeInfo and FileInfo objects differs depending on the current display method. When using real-time display, the Info objects obtain the real-time information from the receive process through the use of the shared

memory segment. When using the post-mortem display the data is obtained from the stored Info list using the current time Info pointer.

3.6 - DEVELOPMENT IMPLEMENTATION CONSIDERATIONS

3.6.1 - UIT Graphical Library

NodeView is constructed using an event-driven architecture based on UIT. UIT is a C++ class library (UIC) and a utility (guic) to translate DevGuide files into C++ code that uses UIC. However, it is possible to integrate UIC functionality into C++, without using DevGuide. The UIC set of C++ classes, define objects that produce the graphical user-interface of an application. It allows for easy component extensibility and the creation of new components, while providing the look-and-feel conforming to the OPENLOOK standard.

There are many reasons why the UIT system was chosen for the user-interface library. Foremost it is an easy to use high level C++ application class library. It greatly reduces the complexity of developing a sophisticated graphical interface, when comparing with standard C methods. At the same time it is the most sophisticated freely available application class library; which is an important issue when porting to other platforms.

Many important high level application functional aspects are provided by UIT. It provides an easier interface for UI function callbacks. Normally, when you define callback functions you must use a sequence of parameters. UIC removes this need by

allowing for simple event callback functions, which pass only one parameter (a pointer to the object on which the event occurred). Then any access to the needed data can be performed through the object's member functions.

As well, UIT simplifies the use of color. UIT allocates a ColorMap object for each application and for each canvas in an application. These ColorMaps can be isolated from the color tables of other display elements of the application. Once you load colors into the application's ColorMap object, the colors can be used by name throughout the application.

Other useful features include HotRegion objects and simplified Drag and Drop functionality. HotRegion objects can be rectangular or irregularly shaped regions and each allows for an event handler function. Drag and Drop functionality is provided by member functions and are available for all visible objects.

The UIC classes fall into five functional categories, each will be described briefly.

- The first group of objects form the basis or framework for any graphical user-interface application, such as a window.
- The second group of objects form the surface within the framework upon which individual screen images appear, such as a component display area.
- The third group of objects give users an interface to the application, such as buttons, sliders, or text messages.
- The fourth group of objects provide for branching and choices, such as menus.
- The fifth group are miscellaneous objects, such as color, character fonts, and images.

When using these classes you can follow a standard method to produce your application interface. You begin by creating a main window, on this window you place a

surface, once the surface is created buttons and text messages can be positioned as desired, finally you associate functions with any objects that you created.

3.6.2 - System Design Considerations

To understand the total requirements for NodeView, a study of the AVX tracing system was undertaken. It was discovered that the included tracing does not have the flexibility needed to make a robust system. Thus, NodeView is designed as a completely new system.

Many design issues were considered in the design of the system. There were two methods to display data: the host, or graphical node. Upon choosing the display system (the host), it was important to choose the correct graphical libraries. The graphical libraries chosen (UIT) is simple to use and portable to other systems.

After choosing these higher level design issues, more specific design items needed to be considered. The best way to obtain the desired information from the AVX nodes for real-time and post-mortem systems needed to be chosen. As well, the methods on how the nodes would communicate their information to the graphical system. And finally, the layout and interaction of the graphical nodes themselves needed to be considered.

In the first design of the system, the host would probe each node and obtain all the information needed. This worked, but there were a few problems. Structured in this way, the system causes a larger disturbance than desired. First, more data than necessary was sent back. As well, the functions used to get node information had a higher priority than application user messages. This forced the system to process NodeView messages first,

which in turn slowed the application program down considerably. An undesirable side effect of this design is that the user interface would freeze if a node was busy or crashed.

NodeView's current system design reduces application disturbance by producing message traffic at a very small level. Before information is sent to the host node it has been processed locally decreasing its size to 64 bytes. This information only has to be sent back once per second. The combination of these two items reduces application disturbance quite effectively. More advance methods have been included in the system to help reduce application disturbance. One can turn off update messages: no live data is sent back to the host, but information is still collected and stored at each node.

The information collected at each node can be captured as fast as 100 ms, but 200 ms is a more reasonable timer value. This results, because the functions called to get the information on each local node demon can not return faster than 200 ms on heavy communication loads. As well, the total amount of data stored can become quite large at the 100 ms resolution. For example, 64 nodes at 100ms resolution running for 10 min produces 15 megabytes of data. Overall, the 200 ms resolution is a good value to use, since it still shows fine details, and it reduces the total amount of information stored.

4 - NODEVIEW EXPERIMENTS

Various functional levels are considered in examining NodeView results. At the basic level, system functionality and validity is examined and reported. Then, the tool is used in a variety of experiments to show its uses and at the same time provide more insight to the AVX system.

4.1 - EXPERIMENT DESIGNS

The type of experiments performed fall into two groups: functional and production. The functional experiments show basic system functionality. This includes routing, congestion maps, memory, process, buffer, and file data for each node. The production experiments show tool use in user applications. These experiments show how the tool can be used to help to fix a nonfunctional application or improve application performance. The functional experiments are given first followed by the production experiments. Each experiment is contained in its own subsection.

4.1.1 - Functional Experiments

4.1.1.1 - 4x4 Routing Map 1

Purpose: To show the message route from node 15 to node 0 in a 4x4 mesh.

Figure 20 - 4x4 routing map 1

The route used for message passing from node 15 to node 0 in a 4x4 mesh is shown by the highlighted nodes, i.e., nodes 15, 14, 10, 6, 5, 1, 0 (6 hops). This route is constant for all 4x4 meshes passing messages from node 15 to node 0. This is expected since the AVX uses static routing. The numbers on the links between nodes form the congestion map and will be discussed shortly.

4.1.1.2 - 4x4 Routing Map 2

Purpose: To show the message route from node 0 to node 15 in a 4x4 mesh.

Figure 21 - 4x4 routing map 2

The route used to pass messages from "node 0 to node 15" and "node 15 to node 0" is not identical. The AVX uses a constant seed random static route. This will produce the same route each time a topology is made, but does not force route pairs to be identical. The given example contains the same number of hops, but this is not always true. The routes used to communicate between node 7 and node 8 differ by 2 hops. The route used from node 7 to node 8 uses 6 hops (7, 11, 15, 14, 13, 12, 8). While, the route used from node 8 to node 7 uses 4 hops (8, 9, 10, 11, 7). If a route has less hops, information can be transferred quicker. It is important to consider this result when timing

is important between pairs of nodes. Time synchronization issues can result in various parallel programming problems.

4.1.1.3 - 8x8 Congestion Map

Purpose: To show a large congestion map and demonstrate current routing inefficiency.

Figure 22 - 8x8 congestion map

This experiment shows a 8x8 congestion map for the AVX system. The map shows the routing links used to transfer messages from each node in the system to a

destination node (here node 0). There are inefficiencies to the present routing algorithm design. Each route is short as possible, but some links see more traffic than others. The major pathways in the map are where most messages pass through, and these pathways can overload the AVX communication system.

These pathways are the result of the AVX routing algorithm. It tries to minimize overall communication loads, but it is not optimal for individual nodes. A better design would try to spread the links more evenly throughout the mesh. For example, node 0 shows that 62 nodes use one of its links and only 1 node uses the other. It would be better to use each link equally. If done in this manner the system bandwidth can be increased, and at the same time put less of a burden on a node's links. This will decrease buffer overruns and therefore increase system stability.

4.1.1.4 - Intel 860 Memory and Process Usage

Purpose: To validate NodeView's I860 memory and process usage.

Figure 23 - I860 memory and process usage

For this experiment, an application that consumes variable memory amounts on the I860 nodes was constructed. This application was executed on nodes 0-15 (this is confirmed by the squares present in the I860 process bars). Each node's current execution time can be seen right of its id value. These times are accurate within one second. The reason for a possible one second variance results on how NodeView processes real-time visualization messages and was explained in chapter 3.

The application consumes a percentage of memory depending on the row it is

executing on: nodes "0-3" 75% used, nodes "4-7" 50% used, nodes "8-11" 25% used, and nodes "12-15" 0% used (see "Figure 11 - Node visual format" to understand the details).

4.1.1.5 - INMOS 805 Memory and Process Usage

Purpose: To validate NodeView's T805 memory and process usage.

Figure 24 - T805 memory and process usage

This experiment demonstrates memory and process usage for the T805 processor. To achieve this, a memory consumer program was constructed to use 25 % of a node's T805 memory. The number of programs executed on a node depends on which row a

node exists on: nodes "0-3" 1 consumer, nodes "4-7" 2 consumers, nodes "8-11" 3 consumers, nodes "12-15" 4 consumers. The display shows: "row 1" 25% memory usage, "row 2" 50% memory usage, "row 3" 75% memory usage, "row 4" 100% memory usage.

4.1.1.6 - Buffer Queue Usage

Purpose: To validate NodeView's buffer queue usage.

Figure 25 - Buffer queue usage

To show NodeView's buffer usage, an application was constructed to produce messages of variable size and number. The message size used varied depending on the row a node was on: nodes "0-3" 64 bytes, nodes "4-7" 512 bytes, nodes "8-11" 1024 bytes, and nodes "12-15" 4096 bytes. The application then produced an increasing

2

OF/DE

2

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

number of messages for a given row: node "1" 11, node "2" 21, node "3" 31, node "4" 41. This application shows that NodeView displays buffers correctly and how the buffer bar can be used to estimate message size.

4.1.1.7 - Buffer Worker Usage

Purpose: To validate NodeView's buffer worker usage.

Figure 26 - Buffer worker usage

To show NodeView's buffer workers usage, an application was constructed to send messages of different event types to each node in AVX. This application sends a variable number of messages (each message a different event type) to a node depending

on is row number row "1" 8 messages, row "2" 16 messages, row "3" 24 messages, row "4" 32 messages. The above display shows the percentage of buffer workers used on each row: row "1" 25% used, row "2" 50% used, row "3" 75% used, row "4" 100% used.

4.1.1.8 - File State Usage

Purpose: To validate NodeViews' file state usage.

Figure 27 - File state usage

This experiment shows NodeView processing file state information. An application was constructed to read a 512 kilobyte file on each node. The time file transfer started on each node depended on its row number. Row 4 started transferring first

followed by row 3, then row 2, and finally row 1. Each row's nodes started to read the file after the previous row's nodes have read 128 kilobytes of the file. The above display shows how far the transfer has progressed for each row: row "4" 75% finished, row "3" 50% finished, row "2" 25% finished, row "1" 0% finished.

4.1.1.9 - Overloading the Host Node

Purpose: To demonstrate that the host node can be a communication limitation

Figure 28 - Overloading host node

This experiment shows that the host node can become a communication bottleneck. For this experiment each node in a 8x8 mesh sends visualization data to the host node once every 800 milliseconds. Eventually the host node could not process the number of messages required and the system crashed.

Information about the crash can be obtained by examining the buffer view window. This shows that there are 1509 messages using approximately 1 gigabyte of buffer memory. This amount of buffer memory is incorrect and shows some internal error. Normally only 256 kilobytes of memory is allocated to the buffer demon.

This is a surprising result, since the number of messages present is still rather small. It shows that the host node needs a fix amount of time to receive a message, and if messages are sent quicker than this the host node can become overwhelmed.

4.1.2 - Production Experiments

4.1.2.1 - NEMO

NEMO System

The following discussion of the NEMO system has been extracted from a paper entitled "Parallel Neighbourhood Modeling: Research Summary" [56]. The description gives background reasons for NEMO development and its present design and functionality.

"In recent years the demand on Geographical Information System (GIS) technology from application areas has sharply increased. In particular, with the EOS (Earth Observation System), terabytes of spatial information will be collected daily. The increase in speed in sequential computing cannot keep up with the demand placed by many systems handling spatial data. Therefore a need for parallel computing in this area is widely recognized.

We are in the third year of a project funded in part by industry and the Canadian Government (NSERC) to develop a parallel GIS. The funding for the project is approximately \$4,000,000 (CAN). Here, we describe our first major milestone, the design and features of a parallel system for NEighbourhood MOdeling (called NEMO) of raster data. The mandate of our system is to provide users with a fast engine for processing a variety of time-consuming tasks for raster based data. The system is designed to be platform independent and is currently implemented on the AVX series II parallel computer manufactured by Alex Informatique, a Canadian supercomputer manufacturer.

NEMO is capable of processing three types of time consuming raster neighbourhood models: Cellular Automata, Propagation, and Neighbourhood Analysis. NEMO is not tailored to any specific application. Rather, it is designed to support applications falling under the umbrella of the above raster neighbourhood modeling families. NEMO achieves this flexibility by having five components: three application drivers, namely, Cellular Automata Driver, Propagation Driver and Neighbourhood Analysis Driver components, and two client-server components, namely, Display Manager and Raster Database Manager" (see Figure 29 - NEMO system overview)

Figure 29 - NEMO system overview

NodeView use of NEMO

The NEMO system provided NodeView with a very useful feedback mechanism during development. Early in NodeView development NEMO's executions were visualized. This proved useful for both systems. NodeView functionality expanded to provide information useful to NEMO.

This feedback between systems went through multiple iterations. This resulted in NodeView changing its fundamental design (see section 3.6.2) and including file transfer functionality and total message and total buffer memory used. This in turn provided additional insights for the NEMO application, as well as, the AVX system.

The following displays show execution profile snapshots of the NEMO system. Two versions of NEMO were profiled. NEMO "1" was released on January 8th 1996 and

NEMO "2" was released on February 15th 1996. The two NEMO versions use different methods to transfer raster files from/to the host node, and show how design changes can effect the communication bandwidth used by an AVX application.

NEMO Profiles

Purpose Examining NEMO "I" file transfers

Figure 30 - NEMO "I" file state data

Initially NodeView did not consider file transfer information. After visualizing NEMO it was discovered that communication was present but not being shown. After some research it was discovered that file transfers are handled differently than message passing and NodeView was enhanced to include this functionality.

This experiment demonstrates NodeView file transfer functionality. The above real-time display shows NEMO "I" loading a 300k raster file (mountain) to each ITB node, and shows an interesting result with node 10. All nodes in the AVX except node 10

are currently loading the raster file at the same time. When watching the real time display one can see file transfers on node 10 start and stop. After consulting with the NEMO developer it was discovered that node 10 was responsible to send update messages to the host. This caused node 10 to start and stop file transfers and its apparent irregular behavior.

This type of inconsistency can cause errors which are very difficult to discover. NodeView allowed one to see that a file transfer irregularity occurred, and then proper methods can be taken to solve the problem.

Purpose: Examining NEMO "I" buffer usage for a 300 kilobyte raster file.

Figure 31 - NEMO "I" buffer usage

This experiment using the post-mortem method shows NEMO "1" sending messages to the host for display after finishing with its raster operations on a 300 kilobyte raster file (mountain). One can see that buffer usage is high on nodes 0, 4, and 8. These results will be compared with NEMO "2" buffer usage.

Also shown by this experiment is that the buffer workers on the host node (node 100) are almost completely used. If the application requests more buffer workers, but none are available the application will deadlock. One must pay attention to the total amount of buffer workers being used on the host node.

Purpose: Examining NEMO "2" buffer usage for a 300 kilobyte raster file.

Figure 32 - NEMO "2" buffer usage 1

This experiment shows NEMO "2" buffer usage for the same 300 kilobyte raster file (mountain) used in NEMO "1" buffer usage experiment. One can see that buffer usage is higher than in NEMO "1". This increase in buffer usage becomes important as the size of the raster file increases. The next experiments will show buffer usage for larger raster files.

Purpose: Examining NEMO "2" buffer usage for a 500 kilobyte raster file.

Figure 33 - NEMO "2" buffer usage 2

This experiment shows buffer usage in NEMO "2" for a 500 kilobyte raster file (medTropic). One can see that buffers are used much more heavily. For this version of NEMO, a 500 kilobyte raster file is largest that can be correctly processed. The next experiment shows this result.

Purpose Examining NEMO "2" buffer usage for a 4 megabyte raster file.

Figure 34 - NEMO "2" buffer usage 3

This experiment shows NEMO "2" buffer usage of a 4 megabyte raster file (bigForest). Using a raster file of this size NEMO crashes before all raster messages can be sent to the host for display. Using the buffer view window provides reasons why this happened. The buffer view of node 4 shows that it currently has 593 messages using around 400 megabytes of buffer demon memory. Again, this amount of used buffer memory is incorrect and shows an internal error.

By examining the results from the routing experiments we can see a reason why node 4 has crashed. Since, node 4 is a major pathway in a 4x4 mesh it is possible to use all of its allotted buffer memory, and when this happens the node will crash. This shows

the importance of the correct usage and understanding of communication in the AVX system.

4.1.2.2 - Cell Driver

Purpose: Examination of the Cell driver application

Figure 35 - Cell driver

This experiment profiles the Cell driver [57] application. The Cell driver component of the NEMO application provides a framework on which general cellular automata applications can be easily designed.

The cellular automata computational method decomposes an area into multiple identical cells, and is used to model many problems in geography. It is particularly useful

for raster based GIS models because of its uniform cellular decomposition of a section.

Modeling in this way, is very computationally intensive and using multiple processors is very useful. The neighborhood effects tend to be localized, which keeps the amount of interprocessor communication small. This suggests that good performance can be obtained from a coarse grained parallel machine such as the AVX.

Usually this application functions correctly. However, on this run the application crashed. The application was allowed to continue for 90 seconds to observe the effect. An interesting point is that all nodes have the correct time except for node 11 (time is displayed next to the node id in seconds). For some reason node 11 at 152 seconds has crashed and this has caused the entire application to fail. Later examination of a correctly executing version, showed node 11 was also the last node to finish. NodeView does not always provide the exact reasons for a problem, but using the given information can help limit the areas to explore.

4.1.2.3 - RM1 Live Data

Purpose: Examination of the RM1 application using real-time profiling.

Figure 36 - RM1 live data

This experiment captures data from the RM1 [58] application. This application was designed to construct a more efficient method for message transferring in the AVX. The application designed a new method for sending and splitting a large number of variable size messages. The program transfers 400 megabytes of data from the host node to each node in the mesh.

We can see some interesting results from this program. One can see a large queue of messages on the root's buffer display. When watching the real-time display this queue keeps becoming larger. The application is putting a heavy burden on the root node. If not

careful with other message passing in the system, the root node will not be able to store all the required messages and the system will crash. This display shows the importance of the proper understanding of the root communication component in the AVX system.

4.1.2.4 - RM2 Live Data

Purpose: Examination of the RM2 application using real-time profiling.

Figure 37 - RM2 live data

This experiment profiles the RM2 [59] application. This application is very similar to RM1, but instead of sending 400 megabytes of messages from the host to each node, it sends 400 megabytes of messages from each node to the host. The above real-time display shows how heavy communication to the host can effect an application.

In the real-time method both application messages and visualization messages are handled in the same manner. Therefore, we can infer that application and visualization messages are received at the host nearly at the same time. By examining time on each node we can obtain an estimate on how application messages are being processed.

From this we can see that nodes which are closer to the root can send messages more regularly than further ones. This results because closer nodes have fewer hops to reach the root. This is an important issue when communication in the system is heavy. When nodes must synchronize the delay differences between close and far nodes can cause application functional and performance problems.

4.1.2.5 - RM2 Stored Data

Purpose: Examination of the RM2 application using post-mortem profiling.

Figure 38 - RM2 stored data

This experiment profiles the RM2 [59] application using the post-mortem method. By examining the real-time and post-mortem displays one can see that there are differences. The post-mortem display shows all nodes have similar time. This results from how the data is collected and visualized. In the post-mortem method all data is collected on its local node, and overall communication does not effect data collection and hence visualization.

The above post-mortem display allows one to see inefficiencies with the AVX's static routing. Once the system is initialized message routes are fixed. This decreases the

overall communication bandwidth in the AVX. At the basic level, the profile shows that nodes 0-5 have finished executing, and nodes 6-15 are still running and producing heavy message communications.

Examining the profile in more detail shows that the static messages routes are row ordered. Messages are passed from right to left on a row, and then passed up to the host using the left most node on each row. This can be proved by examining the profile's communication patterns. Nodes 0-3 are finished executing with no buffer queues present. Nodes 5 is finished, but is still handling messages from 6 and 7. As well, examining the profile's playback to completion shows row 2 finishes next followed by row 3 and 4.

This shows an inefficiency in the static routing used in the AVX system. The communication bandwidth could be increase if the messages could be sent through the nodes which have the least communication load.

4.2 - INTERPRETATION AND SUMMARY

The reported results have demonstrated NodeView functionality as well as providing insight into application and AVX system behavior. The three functional areas: topological view, real-time profiling, and post-mortem profiling have been demonstrated and shown to be useful.

NodeView's topological view, providing routes and congestion maps have shown limitations in the static routing used in the AVX communication design. The static routes are formed using a constant seed value random allocation method. This design tries to increase overall communication bandwidth, however this proves to be inefficient.

- The routes used between node pairs are different. This causes messages to be transferred using different nodes resulting in different transfer rates. This can cause application performance and functional problems.
- Nodes which are close to the root node have heavy communication loads and must be used efficiently. If routes are not optimized for these nodes application performance and stability is greatly effected.

NodeView's real-time and post-mortem methods have helped in defining functional and performance problems in the experiment applications. This in turn has given additional insight into the AVX system.

Profiling the NEMO application has shown a communication bandwidth limitation in the AVX system. NodeView results show that major pathways are formed in the used static routes. This can cause application faults when the bandwidth of message transfers pass a threshold value.

This limitation can cause scaling difficulties when NEMO needs to process larger raster files using a large mesh. The maximum raster file used was 4 megabytes and it did not finish correctly, but NEMO will eventually process 100 megabyte files. Without understanding of the AVX routing system, this size of raster file is difficult process.

The NEMO profiles examined used a 4x4 mesh, however an 8x8 mesh provides larger processing performance. When using an 8x8 mesh the communication pathways become a significant problem. Transferring 100 megabyte raster files in a 8x8 mesh is difficult without proper understanding of the AVX routing system.

5 - CONCLUDING REMARKS

5.1 - SUMMARY OF RESULTS

NodeView has been shown to be functional and stable providing insights for applications and the AVX system itself. NodeView's functionality: topological viewing, real-time profiling, and post-mortem profiling each provide useful information.

The topological view has shown limitations in the static routing used in the AVX communication design; while the real-time and post-mortem profiling give application insights in functional and performance areas. This in turn has given an additional insight that communication bandwidth is a limitation of the AVX system

5.2 - EVALUATION OF WORK

NodeView has accomplished the main research goals. It provides a modular and expandable system that provides parallel program visualization functionality that is easy to use. This helps reduce some of the complexity of the AVX application development cycle, which in turn allows one to produce better AVX programs. As well, it has uncovered various inefficiencies in the AVX system software design.

The problems uncovered allow for future research. Eventually this research will produce better AVX system software, which in turn allows for applications that have increased performance and stability. In the short term the results can be used at the application level to directly improve AVX system software inefficiencies.

5.3 - POTENTIAL FUTURE WORK

There are number of items for future research using NodeView. Possible items to consider include:

- Visualizing individual buffer queues

Currently NodeView combines each node's four buffer links and displays them together using a single buffer bar. NodeView could be enhanced to display each link's buffer queue separately. Three components of NodeView's design need to be modified: the NodeView class, the NodeInfo class, and the Alex demon module.

The NodeView class needs to be updated to display link buffer separately. A possible format to use would display each link's buffer on the visual links connecting the nodes.

Figure 39 - Separate link buffers

The NodeInfo and the Alex demon components need to be changed to send and receive individual link buffer information. This requires adding six new fields to the Info structure.

Figure 40 - Expanded Info Structure

- Visualizing message progression

An unfortunate design limitation in the AVX, is that, messages only contain destination information, no originating information is kept. If origin message information was available, message progression through the system could be displayed. One possible way to accomplish this is to add origin information to the data pouch in the message header. The user would be responsible at the application level to include the origin for every message sent. NodeView could then use this information to see how messages are transferred throughout the AVX system.

- User function callbacks in the buffer view window

The buffer view window lists all messages for the selected node. A useful feature would allow a user to click on a message in the buffer view and display its contents using an user defined function. This can be accomplished by assigning each event type to a predefined function relating to its number. For example, event 11 type 0 could call a function called `bvEvent11Type0()`. This feature can be used to check for message packet problems, such as corruption.

- Integration of NodeView with the AVX tracing system

In developing NodeView, the AVX tracing system was studied. The AVX tracing system allows capturing of all high level message send/receives. This provides additional information not currently provided by NodeView. The AVX tracing system uses a signal mechanism to retrieve the message events in the trace buffer demon. NodeView uses this same design to control the Alex demon module.

NodeView can be expanded to control the AVX tracing system, this will increase the AVX tracing system's easy of use, performance, and stability. NodeView's Alex module can receive and process the AVX tracing data for each local node. As well, NodeView transfers data from the nodes to the host in a more sophisticated and efficient method. These will improve the AVX tracing system. Less tracing data has to be sent and the data sent will be transferred by more reliable means.

- Higher dimensional displays.

There is much opportunity to increase understanding of parallel performance data by using three or higher dimensional displays. The 3D Kiviat tube has already shown that using three dimensional displays can enhance previous two dimensional ones. Many two dimensional displays can be extended into a three dimensional version. A short list of possible displays include: 3D Space-time diagrams, 3D XY scatter plots, and 3D matrix views.

Research is needed to discover new useful higher dimensional displays. There are many possibilities to find displays only possible in three and higher dimensions. Parallel performance data has multiple variant relationships; when shown in higher dimensions additional insight not possible in two dimensions can be gained.

NodeView has the flexibility to allow integration of higher dimensional displays. Higher dimensional displays can be integrated directly into NodeView's modular class hierarchy. This would allow real-time higher dimensional viewing of performance data. As well, NodeView's data could be imported to sophisticated visualization systems (IBM data explorer) for their manipulation. This allows for quick and easy production of complex higher dimensional display.

6 - APPENDIX

6.1 - NODEVIEW STARTERS GUIDE

The following steps are needed to begin using NodeView.

- Obtain the NodeView111.tar.gz file.
- Uncompress the NodeView111.tar.gz file into a directory called "nv".
- Add the "nv" directory to the shared library path in the .cshrc file.
- Make an AVX topology in the "nv" directory.
- Execute the "nv" program in the "nv" directory. After executing "nv", one can control NodeView's functionality by its graphical interface.
- Read the nv-tutorial.txt file in the "nv" directory for a short tutorial.

6.2 - NODEVIEW SOURCE CODE

The current NodeView implementation is for Solaris 2.x for Sun's Sparc Station and the AVX Series 2. NodeView's source code is available as a tarred gzip file. It can be obtained by e-mailing to gis@scs.carleton.ca.

REFERENCES

-
- [1] Diane T. Rover, Charles T. Wright, Visualizing the Performance of SPMD and Data-Parallel Programs, Journal of Parallel and Distributed Computing 18, 129-146, 1993.
- [2] T. LeBlanc, J. Mellor-Crummey, and R. Fowler, Analyzing Parallel Program Execution Using Multiple Views, J. Parallel & Distributed Computing, Vol 9, No. 2, June 1990, 203-217.
- [3] M. T. Health, and J. A. Etheridge, Visualizing the performance of parallel programs, IEEE Software 8, Sept. 1991, 29-39.
- [4] Thomas L. Casavant, James Arthur Kohl, Yiannis E. Papelis, Practical Use of Visualization for Parallel Systems, Parallel Processing Laboratory, University of Iowa.
- [5] Eileen Kraemer, John T. Stasko, The Visualization of Parallel Systems: An Overview, Journal of Parallel and Distributed Computer 18, 105-117, 1993.
- [6] James Arthur Kohl, Visual Techniques for Parallel Processing, Ph.D. Comprehensive Examination, 1991.
- [7] Cherri M. Pancake, Margaret L. Simmons, Jerry C. Yan, Performance Evaluation Tools for Parallel and Distributed Systems, Computer, Nov. 1995, 16-19.
- [8] A. D. Malony, D. A. Reed, An integrated performance data collection, analysis visualization system, Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications, Monterey, CA, Mar. 1989.
- [9] D. Kimelman, T. Ngo, Program visualization for RP3: An overview, Technical report, IBM Research Divisionm T. J. Watson Research Center, 1990.
- [10] S. J. Eggers, D. R Keppel, E. J. Koldinger, H. M. Levy, Techniques for efficient inline tracing on a shared-memory multiprocessor, Performance Evaluation Rev. 18, May 1990, 37-47.
- [11] B. P. Miller, C. Q. Yan, IPS: An interactive and automatic performance measurement tool for parallel and distributed program, Proceedings of Seventh International Conference on Distributed Computing Systems, University of Wisconsin, Sept. 1987, 482-489.
- [12] S.R Sarukkai, Performance visualization and prediction of parallel supercomputer programs: An interim report, Technical Report 318, Indiana University, Bloomington, IN, Nov. 1990.

-
- [13] T. Lehr, et al. Visualizing performance debugging, Computer 22, Oct. 1989, 38-51.
- [14] A. D. Malony, D. A. Reed, Visualizing parallel computer system performance, In Simmons, Parallel Computer Systems: Performance Instrumentation and Visualization. Assoc. Comput. Mach., New York, 1990.
- [15] T. J. LeBlanc, A. D. Robbins, Debugging parallel programs with Instant Replay, IEEE Trans. Comput. C-36, Apr. 1987, 471-482.
- [16] L. D. Wittie, Debugging distributed C programs by real time replay, SIGPLAN Notices 24, Jan. 1989, 57-67.
- [17] M. Friedall, M. LaPoll, et al, Visualization the behavior of massively parallel programs, Proceedings of Supercomputing 1991. Albuquerque, NM, Nov. 1991, 472-480.
- [18] C. G. Roman, K. C. Wilcox, Pavane: a system for declarative visualization of concurrent computations, J. Visual Languages Comput. 3, June 1992, 161-193.
- [19] A. A. Hough, J. E. Cuny, Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation, Proceedings of the 1987 International Conference on Parallel Processing, Aug. 1987, 735-738.
- [20] Barton P. Miller, Jonathan M. Cargille, R. Bruce Irvin, The Paradyn Parallel Performance Measurement Tools, University of Wisconsin-Madison.
- [21] J. Dongarra, O. Brewer, J. A. Kohl, S. Fineberg, A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors, J. Parallel Distrib. Comput. 9, June 1990, 185-202.
- [22] S. R. Sarukkai, D. Gannon, Performance visualization of parallel programs using SIEVE.1, Proceedings of the 1992 International Conference on Supercomputing, Washington, D. C., July 1992, 157-166
- [23] B. Miller, et al., IPS-2: The second generation of a parallel program measurement system, IEEE Trans. Parallel Distrib. Systems 1, Apr. 1990, 206-217.
- [24] Thomas Fahringer, Estimating and Optimizing Performance for Parallel Programs, Computer, Nov. 1995, 47-56
- [25] L. Lamport, Time, clocks, and ordering of events in a distributed systems, Comm. ACM 21, July 1978, 558-565.
- [26] T. J. LeBlanc, J. M. Mellor-Crummey, et al., Analyzing parallel program execution

using multiple views, J. Parallel Distrib. Comput. 9, June 1990, 203-217.

[27] Mark E. Crovella, Thomas J. LeBlanc, The Search for Lost Cycles: A New Approach to Parallel Program Performance Evaluation, University of Rochester, Technical Report 479, 1994.

[28] Hank Jakiela, Performance Visualization of a Distributed System: A Case Study, Computer, Nov. 1995, 30-36

[29] P. Bates, Debugging heterogeneous distributed systems using event-based models of behaviour, SIGPLAN Notices 21, Jan. 1989, 11-22.

[30] D.P. Helmhold, C.W. McDowell, J. Z. Wang, TracerViewer: a graphical browser for trace analysis, Technical Report UCSC-CRL-90-59, University of California at Santa Cruz, Oct. 1990.

[31] D. A. Reed, R. D. Olson, R. A. Aydt, et al., Scaleable performance environment for parallel systems, Proceedings of the Sixth Distributed Memory Computing Conference, Apr. 1991, 562-569.

[32] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Falcon: Online Monitoring and Steering of Large-Scale Parallel Programs, Technical Report GIT-CC-94-21, 1994.

[33] D. Bailey, M. L., D. Notkin, Voyeur: Graphical views of parallel programs, SIGPLANS Notices 24, Jan. 1989, 206-215.

[34] A. Beguelin, J. J., Dongarra, G. Geist, et al., Graphical development tools for network-based concurrent supercomputing, Proceedings of Supercomputer 1991, Albuquerque, New Mexico, Nov. 1991, 435-444.

[35] V. Sunderam, PVM: A framework for parallel distributed computing, Concurrency Practice Experience 2, Dec. 1990, 315-339.

[36] Adam Louis Beguelin, Xab: A Tool for Monitoring PVM Programs, Pittsburgh Supercomputer Center.

[37] J. Dongarra, D. Sorenson, SCHEDULE: Tools for developing and analyzing parallel Fortran programs, The Characteristics of Parallel Algorithms, MIT Press, Cambridge, MA, 1987.

[38] R. J. LeBlanc, A. D. Robbins, Event-driven monitoring of distributed programs, Proceedings of the Fifth International Conference on Distributed Computing Systems, May 1985, 515-522.

-
- [39] J. A. Kohl, T. L. Casvant, Use of PARADISE: A meta-tool for visualizing parallel systems Proceedings of the Fifth International Parallel Processing Symposium, Anaheim, CA, May 1991, 561-567
- [40] K. Hwang, F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, New York, NY, 1984.
- [41] J. J. Flynn, Very High-Speed Computing Systems, Proceedings of the IEEE, 1901-1909, Dec, 1966.
- [42] A. L. DeCegma, Parallel Processing Architectures and VLSI Hardware, Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [43] AVX Series 2 Parallel Computer System: System Description, Alex Informatic Inc., Rev. Oct. 1995.
- [44] Subrata Dasgupta, Computer Architecture: A Modern Synthesis Volume 2: Advanced Topics, John Wiley & Sons, 1989.
- [45] National Science Foundation, Visualization in Scientific Computing, report 1987.
- [46] Michael T. Heath, Allen D. Malony, Diane T. Rover, Parallel Performance Visualization: From Practice to Theory, IEEE Parallel & Distributed Technology, Winter 1995, 44-59.
- [47] Micheal T. Health, Allen D. Malony, Diane T. Rover, The Visual Display of Parallel Performance Data, Computer, Nov. 1995, 21-28.
- [48] B. Miller, What to Draw? When to Draw? An Essay on Parallel Program Visualization, J. Parallel & Distributed Computing, Vol. 18, No. 2, June 1993, 265-269.
- [49] A. D. Malony, D. H. Hammerslag, Traceview: A trace visualization tool, IEEE Software 8, Sept. 1991, 29-38.
- [50] D. Reed, A. Olson, et al, Scaleable performance environments for parallel systems, Processings of Sixth Distributed Memory Computing Conference, Apr. 1991, 562-569.
- [51] Colin Ware, Glenn Franck, Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions, University of New Brunswick, 1994.
- [52] Daniel A. Reed, Keith A. Shields, Will H. Scullin, Luis F. Tavera, Christopher L. Elford, Virtual Reality and Parallel System Performance Analysis, Computer, Nov. 1995, 57-67

[53] E. Tufte, The Visual Display of Quantitative Information, Graphics Press, Cheshire, Conn., 1983.

[54] P. Keller and M. Keller, Visual Cues: Practical Data Visualization, IEEE Press, Piscataway, N.J., 1993.

[55] E. F. Johnson, K. Reichard, Advanced X Window Application Programming, M & T Books, Dec. 1994.

[56] D. Hutchinson, L. Kuttner, M. Lanthier, A. Maheshwari, D. Nussbaum, D. Roytenberg, J.-R Sack, Parallel Neighbourhood Modeling: Research Summary, to be presented at SPAA 96.

[57] D. Roytenberg, Cellular Automata Driver, Carleton University, M. C. S. thesis, in preparation, May 1996.

[58] D. Nussbaum, RM1, Carleton University, 1996.

[59] D. Nussbaum, RM2, Carleton Univeristy, 1996.

END

1 810 6-9 7

FIN