INSTRUMENTATION, PERFORMANCE VISUALIZATION AND DEBUGGING TOOLS FOR MULTIPROCESSORS

Jerry C. Yan and Charles E. Fineman
Sterling Federal Systems Inc.
MS 244-4, NASA Ames Research Center, Moffett Field, CA 94035

Philip J. Hontalas
Computational Systems Research Branch
MS 244-4, NASA Ames Research Center, Moffett Field, CA 94035

ABSTRACT

The need for computing power has forced a migration from serial computation on a single processor to parallel processing on multiprocessor architectures. However, without effective means to monitor (and visualize) program execution, debugging and tuning parallel programs becomes intractably difficult as program complexity increases with the number of processors. Research on performance evaluation tools for multiprocessors is being carried out at NASA Ames Research Center. Besides investigating new techniques for instrumenting, monitoring and presenting the state of parallel program execution in a coherent and user-friendly manner, prototypes of software tools are being incorporated into the run-time environments of various hardware testbeds to evaluate their impact on user productivity. Our current tool set, the Ames InstruMentation System (or AIMS), incorporates features from various software systems developed in academia and industry. The execution of FORTRAN programs on the Intel iPSC/860 can be automatically instrumented and monitored. Performance data thus collected can be displayed graphically on workstations supporting X-Windows. We have successfully compared various parallel algorithms for CFD applications in collaboration with scientists from the Numerical Aerodynamic Simulation Systems Division. By performing these comparisons, we show that performance monitors and debuggers such as AIMS are practical and can illuminate the complex dynamics that occur within parallel programs.

1. INTRODUCTION

1.1. Motivation and Background

While parallel processing promises to deliver orders of magnitude speed-up in the near future, the actual speed-up obtained from parallel processing will always depend critically on three factors: i.) how the parallel application is formulated; ii.) the architecture of the multiprocessor and iii.) how well the application is mapped onto the machine. Although research in these areas has produced many interesting results based on simulation and theoretical considerations, their validity must be substantiated by data gathered from actual implementations. Such performance evaluation on multiprocessors presents many technical challenges.

A parallel program has many threads of control. Whether they are expressed as "parallel do-loops" or concurrent processes/objects, the completion time of the entire program depends on the order in which synchronization/communication events occur on different control threads. This "event-ordering" data is difficult to collect, analyze and present in a manner that relates performance with program structure and hardware architecture. Having accurate resource utilization information, for example, can be especially helpful for evaluating the effectiveness of the current program-to-machine mapping — whether there is proper trade-off between communication and concurrency as the computation is distributed over many processors.

In summary, whether a researcher is designing the "next parallel programming paradigm", another "scalable multiprocessor" or investigating resource allocation algorithms for multiprocessors, a facility that enables parallel program execution to be captured and displayed is invaluable. Careful analysis of such infor-

mation can help computer and software architects to detect, and therefore, exploit behavioral variations among/within parallel programs to take advantage of specific hardware characteristics.

1.2. Instrumentation Methodologies

Performance evaluation presumes some form of *instrumentation* — a mechanism whereby the execution of the program can be monitored. A variety of such mechanisms have been proposed to gather different information; these include *event sampling*, *passive event recorders*, and *inserted active event recorders*. A detailed survey of the various instrumentation methodologies for multiprocessors may be found in [1].

An event sampler, whether software or hardware, periodically examines and records the state of the executing software. For example, the UNIX *gprof* [2] has been used to collect statistics about the the distribution of work among the modules and statements of a sequential application. In a sequential environment, an external agent (usually another process in a multiprogramming environment) carries out the *sampling* by periodically interrupting the monitored process to record the value of its program counter. Based on the data collected, the time spent in various parts of a program can be determined. *Event sampling* techniques have been applied successfully on sequential programs for many years now. In a parallel processing environment however, *event sampling* might not be feasible because a *sampling process* can be highly intrusive. Even if the problem of intrusion is overcome through the use of specialized instrumentation hardware, the inter-process event dependencies often found in parallel programs cannot be reconstructed based on statistical data alone.

The use of passive event recorders requires specialized instrumentation hardware for implementation. The word "passive" implies that a monitored system does not do anything *extra* for performance data to be collected. Program state, therefore, must be deduced from low-level data gathered from various devices such as addresses/data placed on buses or values in registers. Even with simple sequential programs, a large amount of data has to be gathered. This implies that instrumentation hardware for parallel systems has to cope with even higher data rates and capacities. Furthermore, hardware monitors tend to be inflexible and vendor specific. The algorithms that relate collected data to program source code must take into account specific compilation strategies and operating system versions. It takes a lot of effort to build a single passive instrumentation system — not to mention building a suite across different software/hardware architectures for research and development.

Inserted active event recorders collect exactly what you want to measure — no more no less. Just like putting print-statements at various points in the program to trace its control-flow, "event records", which indicate event types and their times of occurrence, can be placed at various points of the source code. Program execution can then be easily reconstructed based on these records. The tedious task of instrumenting program source code can be automated, even across different parallel programming languages[1]. Furthermore, this approach is highly portable since the program is instrumented at the source code level. The performance of an instrumented parallel program can be studied on any machine without major modification. Because the event format can be standardized across different machines/languages, only one set of performance analysis tools is required to interpret the data gathered. Although the overhead of this approach is not negligible, it still can be accurately measured, characterized and factored out using various compensation techniques (e.g. [3]).

1.3. Outline of Paper

The goal of this paper is to present some of the techniques and methodologies employed in the instrumentation and performance debugging of applications executing on multiprocessors. To that end, this paper will present our current tool set, the Ames InstruMentation System (AIMS), as an example. Section 2 of the paper describes how AIMS monitors program execution. The *source code instrumentor* automatically inserts active event recorders (i.e. subroutine calls to the *run-time performance monitoring library*) into the source code before compilation. Performance data generated by these event recorders are gathered into a *trace file* from which the *visualization tool-set* reconstructs program execution. Section 3 contains a sample of views obtained

---

[1] For example, PIE [7] uses a source code instrumentor that handles parallel programs written in C, Ada, and FORTRAN.

using AIMS to measure the performance of a parallel version of ARC2D, a computational fluid dynamics application, on the Intel iPSC/860 at NASA Ames Research Center. In this case, AIMS helped the researcher to idenfify execution bottlenecks and room for improvement. Conclusions and directions for future research are discussed in section 4.

2. THE AMES INSTRUMENTATION SYSTEM

2.1. Structural Overview

AIMS is designed to facilitate performance evaluation of parallel applications on multiprocessors by capturing and visualizing execution data. AIMS has three major software components: a *source code instrumentor*, a *run-time performance monitoring library* and a *visualization tool-set*.

The instrumentor inserts active event recorders (i.e. function calls to the monitor library) directly into the application source code with little or no intervention by the user. AIMS provides a graphical interface for the researcher to selectively instrument his/her code. As shown in Figure 1, specific modules and procedure calls can be selected/deselected easily via the click of a mouse. Thus, the programmer is relieved of the tedious work of instrumentation by hand.

The monitor library provides a set of active event recorders to measure and record various aspects of program performance such as message passing overhead, processor synchronization overhead, and processor time spent in user defined areas of the application.

The visualization tool-set processes the execution data gathered and displays them using graphical views. Detailed information showing how the application interacted with the multiprocessor is presented using *animated views*, from which processor state, implementation bottlenecks and load imbalances can easily be observed. Performance statistics of the entire program execution can also be gathered and displayed via *statistical views* to provide insights into the general behavior of the program; these may yield valuable clues regarding where the animated views should be focused.

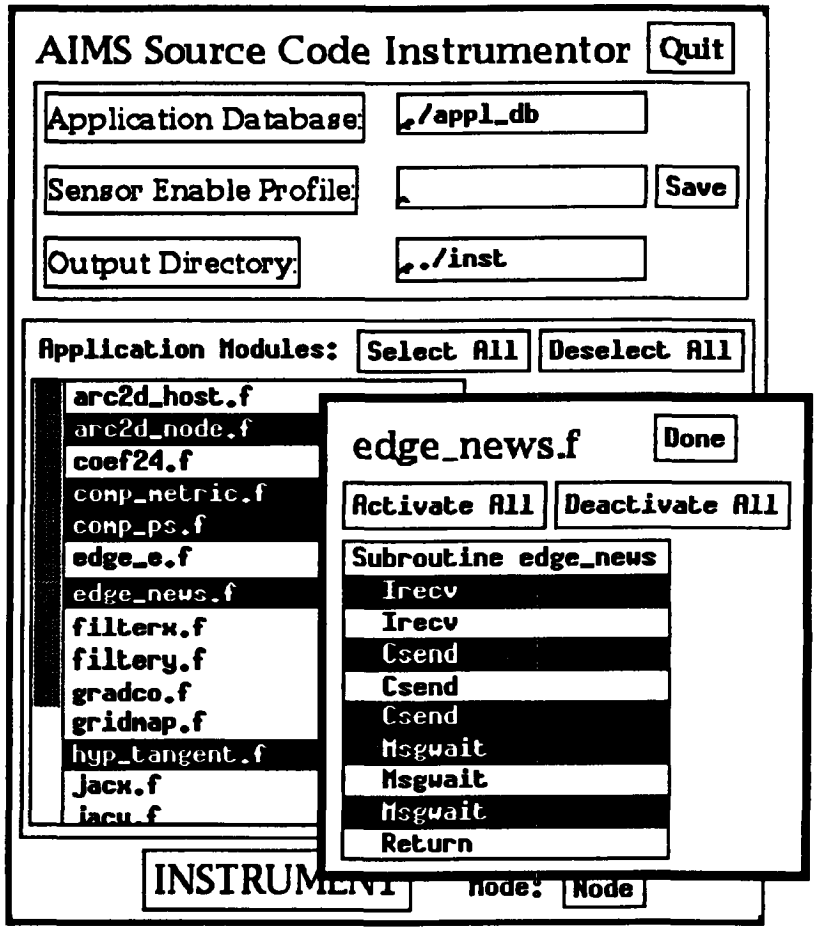

Figure 1. Graphical Interface to AIM's *Source Code Instrumentor*

2.2. Usage Overview

By applying each of the AIMS components sequentially, the performance of various parallel programs on a multiprocessor can be evaluated. As shown in Figure 2, the source code is first instrumented automatically by

AIMS's *instrumentor*. By default, points of interest include message sending, receiving and blocking as well as procedure entries and exits. The user may specify the procedures and code blocks to be monitored, as well as other instrumentation parameters, via a configuration file. Besides adding code at various points in the source code to generate event records, some system calls are replaced by monitor library calls when timing measurements have to be made within such calls[2]. After the source code is instrumented, it is compiled and linked with the run-time performance monitoring library.

The instrumented program is then loaded and run on the multiprocessor. Performance data is gathered during program execution and stored to local memory buffers. Periodically, these buffers fill up and the data is written out to a *trace file* on the file system. Event records generated include:

- *procedure events* — provide performance data on user selected subroutines;
- *blocking events* — indicate waiting time spent on synchronization;
- *message events* — records message transmission time, message size, destination and type; and
- *statistical event records* — summarizes cumulative performance statistics at specified points of program execution.

Finally, the *trace file*, which contains the event records for a monitored program execution, is collected and transferred to a graphic work station to be processed and displayed in various formats. The *visualization toolset* reads the performance data from the trace file and interprets that information on a variety of X-window based displays. With the aid of an example, we will illustrate how different displays can capture various aspects of system performance in section 3.

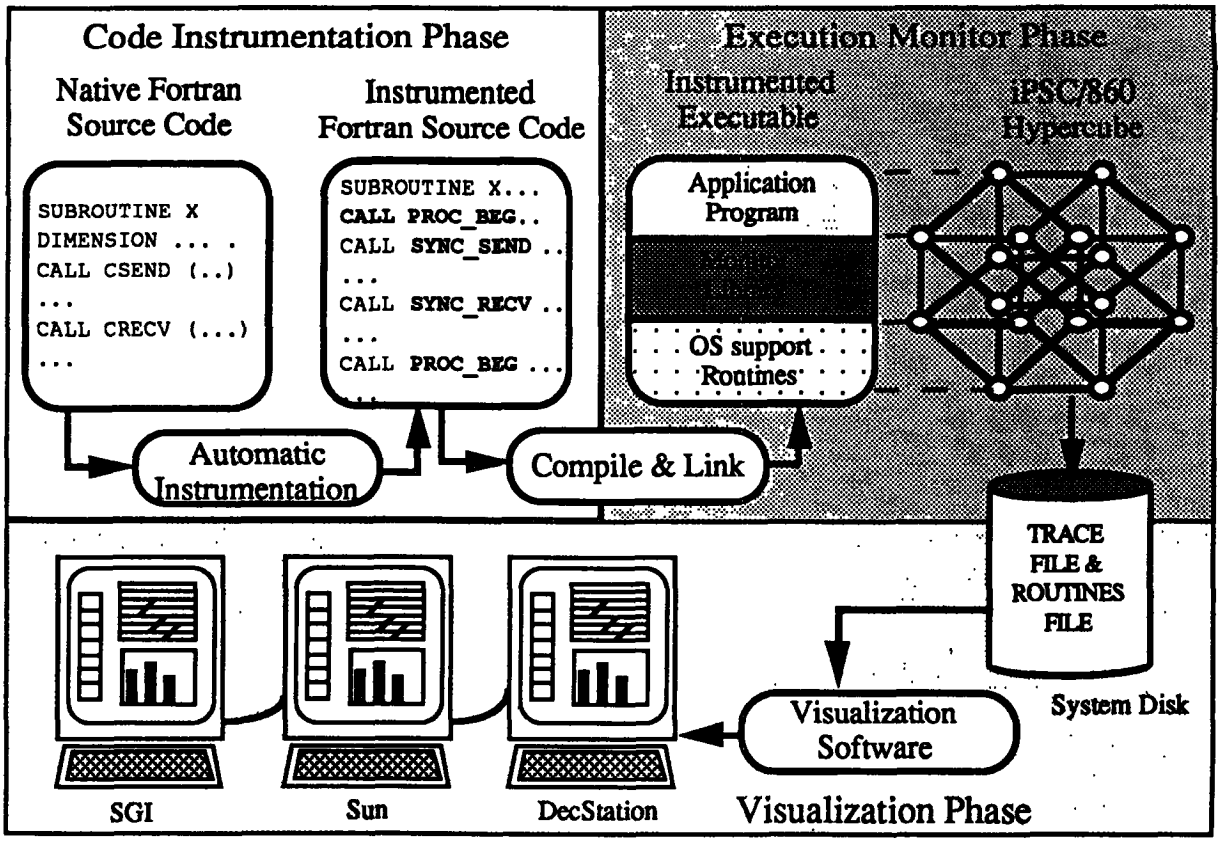

Figure 2. Using AIMS to Collect Performance Data

[2] For example, SYNC_SEND and SYNC_RECV replaces CSEND and CRECV on the Intel iPSC/860 while at the same time, providing timing data about this message transaction.

3. VISUALIZING PARALLEL PROGRAM EXECUTION

3.1. The Example Application

A grand challenge of NASA's High Performance Computing and Communications Program [4] involves the development of parallel Computational Fluid Dynamics (CFD) programs. CFD involves the numerical solution of a system of nonlinear partial differential (Navier-Stokes) equations — these represent the laws of conservation of mass, momentum and energy applied to a fluid medium in motion. One such FORTRAN program, ARC2D [5], which applies an implicit solution algorithm to a problem with two spatial dimensions, has been parallelized for the Intel iPSC/860 Hypercube (an MIMD multiprocessor) at NASA Ames Research Center.

3.2. A Few Examples of Animated Views

The AIMS *visualization toolset* was developed after a careful evaluation of the views provided by the *ParaGraph* [6] visualization toolset and PIE [7]. We selected those that we found useful for our applications and incorporated them into AIMS. In this paper, we only describe those views that are not provided by ParaGraph. The *OverVIEW Diagram* shown in Figure 3 animates program execution by scrolling from right to left. When a processing node (say #15) is busy, a colored bar is drawn (next to the label "15"). The bar is colored according to the subroutine currently executing. White space indicates that the processing node is idle, probably waiting for the arrival of a message. When a message is passed (say from #15 to #14), a (blue) line is drawn from the point (on the sender's time line) when the message was sent to the point (on the receiver's time line) when the message was removed from the queue. The *Aggregate Processor Utilization Chart* plots processor utilization as a function of time. The height of the curve denotes the number of processors currently busy. As shown in Figure 4, it is also color-coded according to subroutine name.

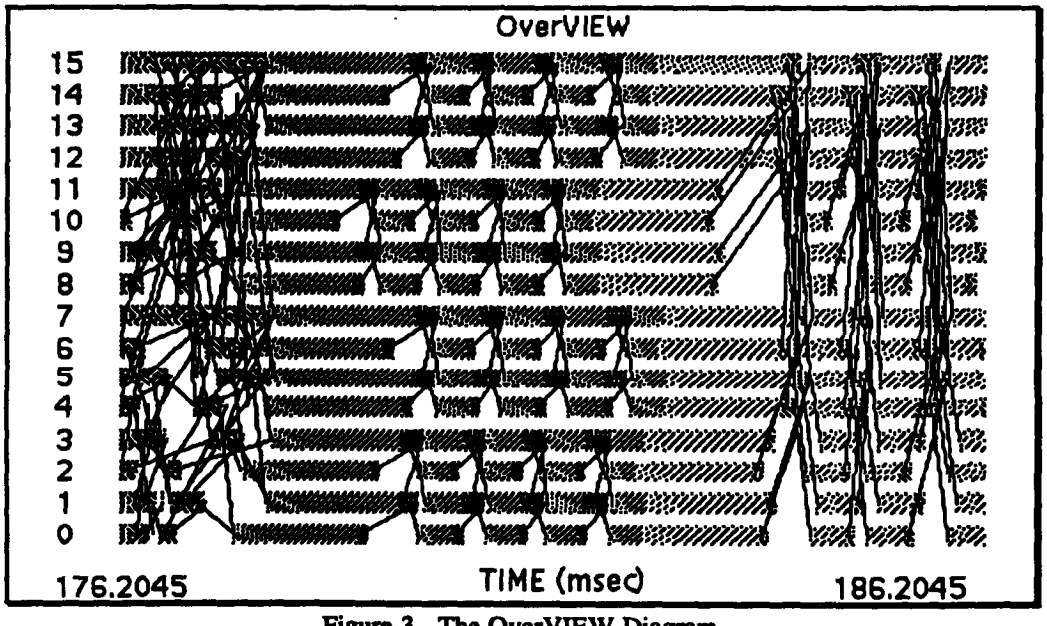

Figure 3. The OverVIEW Diagram

Besides providing views focused on the parallel program's flow of control, AIMS also provide views that display the state of each processor at particular points in time. The Grid view shown in Figure 5 is such an example. Each box of the Grid view displays the current state, subroutine being executed, message queue size and overall utilization for each processor. In addition, this view permits the developer to map the physical processors of the iPSC/860 onto a two dimension mesh. Many parallel applications (such as ARC2D) can be decomposed to topologies which may not conform exactly to the iPSC/860's hypercube.

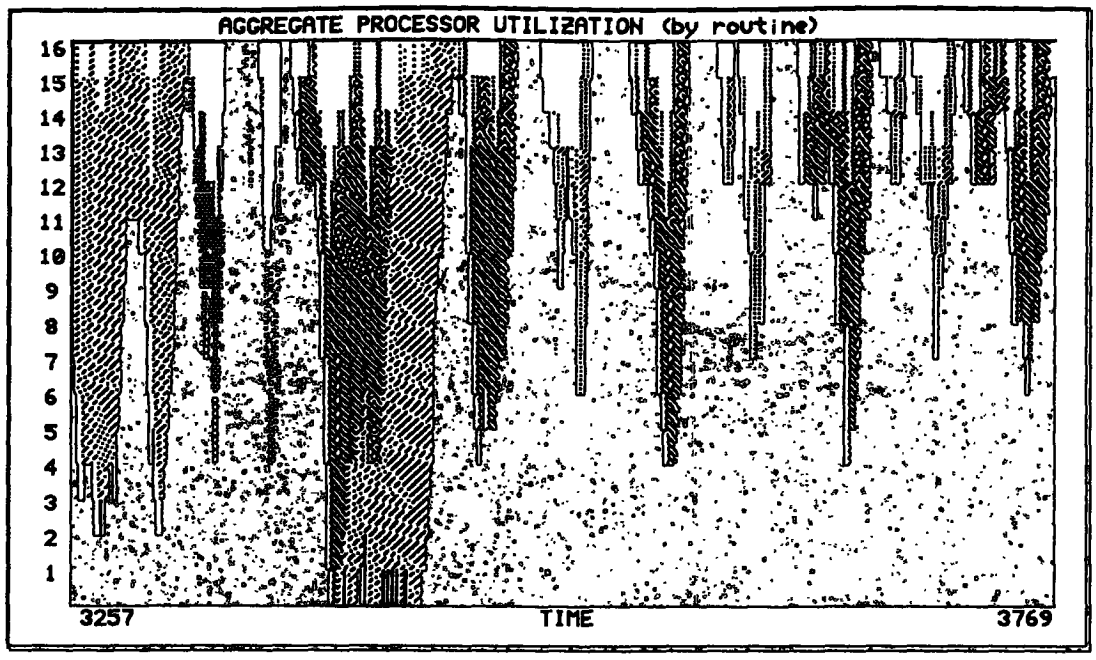

Figure 4. The *Aggregate Processor Utilization Chart*

The NCPU view summarizes the performance characteristics for the entire execution. As shown in Figure 6, a histogram plots the normalized[3] CPU usage of various subroutine. For example, `ypldge` spends most of its time executing when 12 processors are busy.

Based on these animation and statistical views, the programmer can identify the subroutines and message transactions associated with periods of idleness in his/her program. This, in turn, provides valuable insights about the parallelization strategy chosen and helps the programmer to reformulate the application if necessary.

Besides providing graphical data for performance tuning, AIMS also provide an important feature known as *source code click-back*. A mouse click in the OverVIEW will bring up

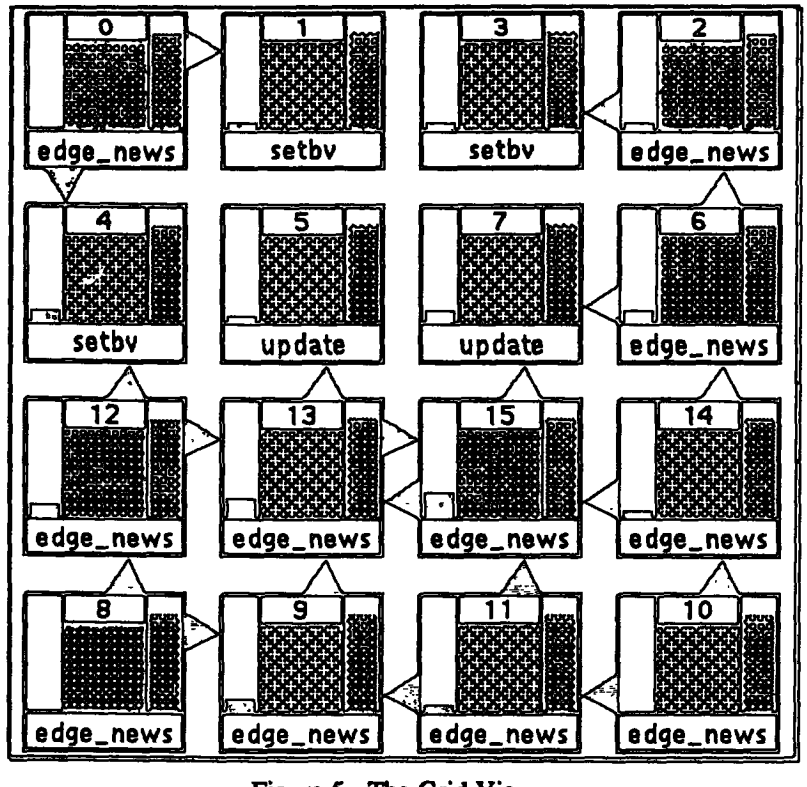

Figure 5. The Grid View

[3] The normalized CPU usage of a subroutine is the total amount of CPU time it used divided by k where k processors were active simultaneously.

a text window depending on the location of the cursor in the view. If the cursor was pointing to a message line, the text file containing the `send` command will be opened and the corresponding program line will be highlighted (as shown in Figure 7). If the cursor is pointing to an idle period of the processor and this idling was caused by the late arrival of a message, the exact `msgwait` call responsible will also be identified. Finally, if the mouse is clicked over a color bar, the code for that subroutine will be retrieved.

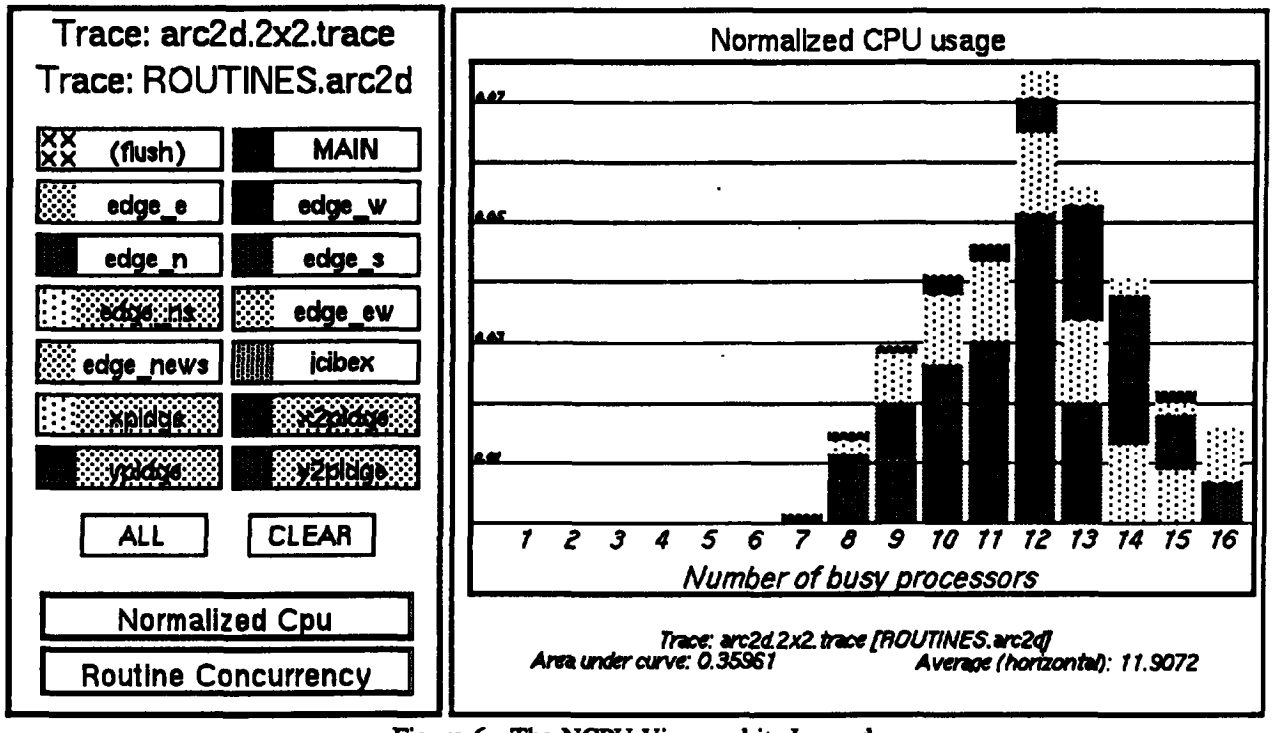

Figure 6. The NCPU View and its Legend

4. CONCLUSIONS AND FUTURE RESEARCH

In summary, the Ames InstruMentation System provides a suite of software tools to facilitate the tuning and debugging of parallel applications. FORTRAN source code is instrumented automatically. Performance data gathered from the execution of instrumented code can be displayed on a variety of workstations. These displays may provide researchers a means for observing the behavior of their programs as well as tracing the sequence of operations via "source code click-back". Thus the performance and correctness of parallel algorithms on hypercubes may be evaluated easily.

Although we have shown that AIMS can be a powerful tool for the development of parallel applications, it is not without pitfalls. One major obstacle to be overcome is data size. Programs running on parallel processors tend to produce an enormous amount of performance data using the techniques described here. Furthermore, data written to disk asynchronously from each processor must be sorted by execution time before it can be read by the visualization toolset. If the data set is particularly large, the overhead of processing this data could render the tools described here impractical. Our current research efforts are addressing the data size and sorting problem from several directions. These solutions include:

- refining the instrumentor to be more selective about which portions of the program to monitor. In future versions of the AIMS, the researcher will be able to enable and disable monitoring according to time and processor parameters. This approach has the potential of greatly reducing the data size.
- integrating a merge sort of the raw data from the multiprocessor at the time of visualization. This technique will eliminate the time consuming pre-sorting process by performing a merge sort of the raw data streams coming from each processor.

- developing "course grain" monitoring tools to compliment the fine grain monitoring capabilities of AIMS. The development of these tools will permit the developer to get a coarse grain view of an application's performance behavior for sampled time periods. Such an approach should have lower overhead in terms of data collection. Based on these coarse grain views, the researcher may identify problem spots quickly which can then be examined more closely be the fine grain performance monitoring facilities of AIMS.

Finally, all performance monitoring systems must deal (to one extent or another) with the problem of perturbance. Instrumentation overhead may re-order events in different control threads of a parallel program and, therefore, obscure the actual data collected. Future versions of AIMS will produce statistics that help determine the level of perturbance within the monitoring process and compensate the performance displays accordingly.

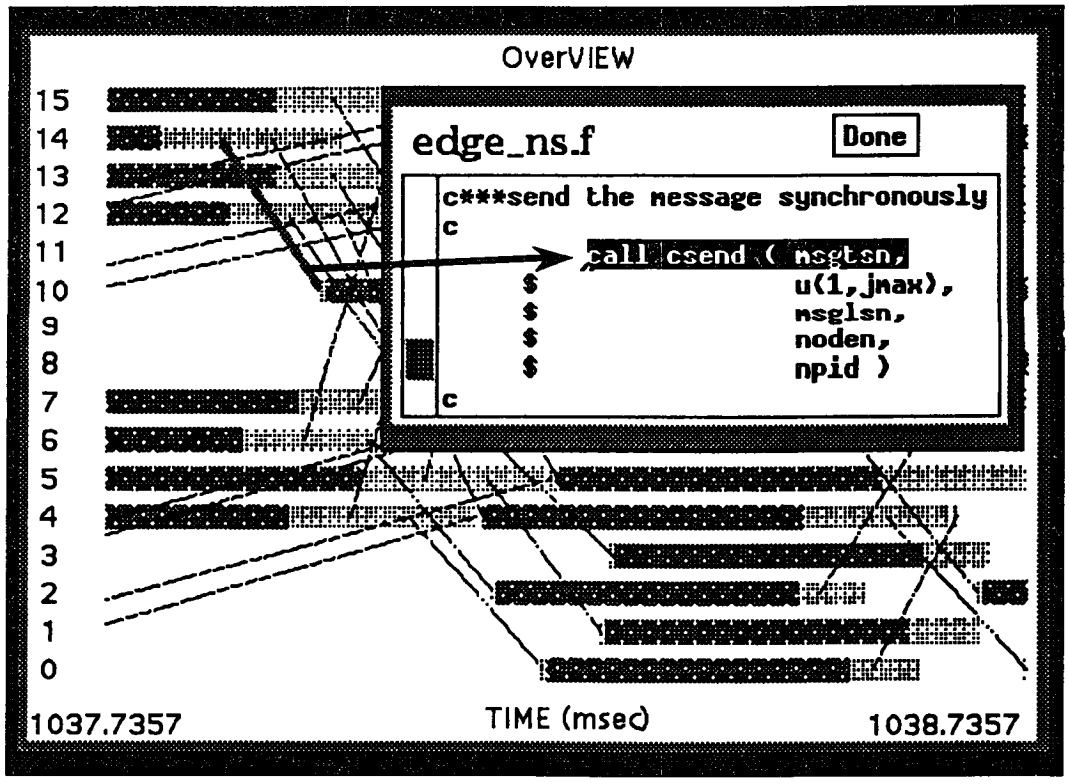

Figure 7. AIMS' *Source-code Click-back* Feature

ACKNOWLEDGEMENTS

AIMS was put together after careful evaluation of a few software prototypes from the research community as well as published ideas on performance visualization. We would like to acknowledge their help/support in letting us adopt, adapt and augment their research prototypes for the parallel processing environment here at NASA Ames Research Center. We also want to thank two summer students, Chris Hanson from Santa Clara University and Sheralyn Listgarter from Stanford University who spent many hours implementing (and watching) scrolling and flickering windows on our workstation screens.

AIMS' instrumentor is implemented on top of a parser developed for PIE [7] which we obtained through the CMU Affiliates' Program. AIMS's monitor library adopted many of the interface conventions established by PICL — a Portable Instrumented Communication Library [8] developed at Oak Ridge National Laboratory. AIMS's visualization toolset incorporated display concepts use by AXE [9] from NASA Ames Research Center, ParaGraph [6] from Oak Ridge National Laboratory and *Quartz* [10] from University of Washington.

REFERENCES

[1] M. H. Reilly, *A Performance Monitor for Parallel Programs*. Academic Press Inc. 1990.

[2] S. L. Graham, P. B. Kessler, M. K. McKusick. "gprof: a Call Graph Execution Profiler". In *Proceedings of SIGPLAN '82 Symposium on Compiler Construction*, June 1982.

[3] T. Lehr. *Compensating for Perturbation by Software Performance Monitors in Asynchronous Computations*. Ph.D. Dissertation, Department of Electrical Engineering, Carnegie Mellon University. 1990.

[4] *Grand Challenges: High Performance Computing and Communications*. A Report by the Committee on Physical, Mathematical, and Engineering Sciences, Federal Coordinating Council for Science, Engineering, and Technology, Office of Science and Technology, Executive Office of the President. 1991.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon Ed., "The NAS Parallel Benchmarks" Report RNR-91-002, NAS Systems Division, NASA Ames Research Center. January 91.

[6] M. Heath and J. Ethridge. "Visualizing the Performance of Parallel Programs". *IEEE Software*, Vol. 8, No. 5, Sept. 1991, pp. 29-39.

[7] Z. Segall, L. Rodolph, "PIE — A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, Vol. 2, No. 6, Nov. 1985, pp. 22-37.

[8] G. A. Geist, M. T. Heath, B. W. Peyton, P. H. Worley "PICL — A Portable Instrumented Communication Library" Tech Report ORNL/TM-11130, Oak Ridge National Laboratory. May 1990.

[9] J. C. Yan. "Axe — An Experimentation Environment for Concurrent Systems". *IEEE Software*, page 25, May 1990.

[10] T. E. Anderson and E. D. Lazowska. "Quartz: A Tool for Tuning Parallel Program Performance". In *Proceedings of SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, May 1990, pp. 115-125.