AD-A262 277

[DOCUM]ENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		FINAL/01 DEC 89 TO 30 NOV 92

4. TITLE AND SUBTITLE

AN INTERGRATED ENVIRONMENT FOR THE DEVELOPMENT OF SCIENTIFIC & ENGINEERING APPLICATIONS (U)

5. FUNDING NUMBERS

3484/MS/URI
AFOSR-90-0044

6. AUTHOR(S)

Professor David J. Kuck

DTIC ELECTE APR 1 1993 S C D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Illinois
305 Talbot Laboratory
Urbana IL 61801-2932

8. PERFORMING ORGANIZATION REPORT NUMBER

AFOSR-TR- [illegible] 3

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
110 DUNCAN AVE, SUTE B115
BOLLING AFB DC 20332-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFOSR-90-0044

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

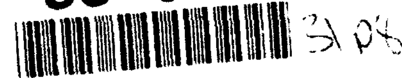
UL

13. ABSTRACT *(Maximum 200 words)*

A new environment has been provided to assist in the programming of high performance parallel and vector computers. The environment consists of a unified set of programming tools such as restructuring compilers, parallel debuggers, performance evaluation tools, and data visualization tools. This design takes into consideration the entire process of scientific /engineering project development.

93 [illegible] 060

93-06622



14. SUBJECT TERMS	15. NUMBER OF PAGES
	30
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR(SAME AS REPORT)

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

Final Report: AFOSR 90-0044 URI Grant
An Integrated Environment for the Development of Scientific and Engineering Applications

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

Principal Investigators
David Kuck, George Cybenko, Dennis Gannon, Allen Malony

DTIC QUALITY INSPECTED

1 Introduction

The overall objective of the URI project was to provide users of high-performance parallel and vector machines with an environment that makes the task of programming easier. The environment unifies a set of tools. Many of these tools are specific to the problem of parallel programming, such as restructuring compilers, parallel debuggers, performance evaluation tools, data visualization and computational steering tools.

The design takes into consideration the entire process of scientific/engineering project development. This process can be characterized as a cycle of distinct transformations applied to a practical problem. Beginning with a description or definition of a physical problem, the user must design and build a source program, debug it, optimize it, and finally execute it to review its results. Figure 1 illustrates the five phases of this cyclic process. The ultimate goal of the project was to provide tools to aid users in all steps along the way during this transformation process.

Referring to Figure 1, the user enters the loop at Phase 1, the physical problem domain. Here the user thinks about an application in the context of a particular problem domain. The first step in the solution process is the transformation of the user's problem from the physical problem domain to the source code domain (Phase 2). In the source code domain, the problem is expressed as a sequence of algorithms encoded in a particular programming language, frequently Fortran or C.

Once the problem has been mapped into a source code representation, the user can begin to evolve it toward a running program. This is the internal program domain (Phase 3). The transition to this internal space requires several tools. First, source language compilers are needed to correct and translate the source code "interpretations" of the physical problem into a form that coincides with the traditional programming paradigm. The internal program domain is still characterized by its machine independence. Although the original physical problem has been rendered into a form that can be executed on a specific hardware platform, no machine-dependent aspects of the implementation have yet entered the design.

In the internal program domain, the user can also investigate various machine-independent aspects of the application that pertain to parallelization. Using the data dependence and control dependence information that exists in this space, the user may apply interactive and automatic restructuring tools to the application for the purpose of improving the *potential* performance of the program. We provided several tools that operate in this domain. The user may invoke automatic restructuring compilers to operate on applications. Interactive tools such as the Sigma editor allow the user manually to peruse the application while querying the system about various aspects. The

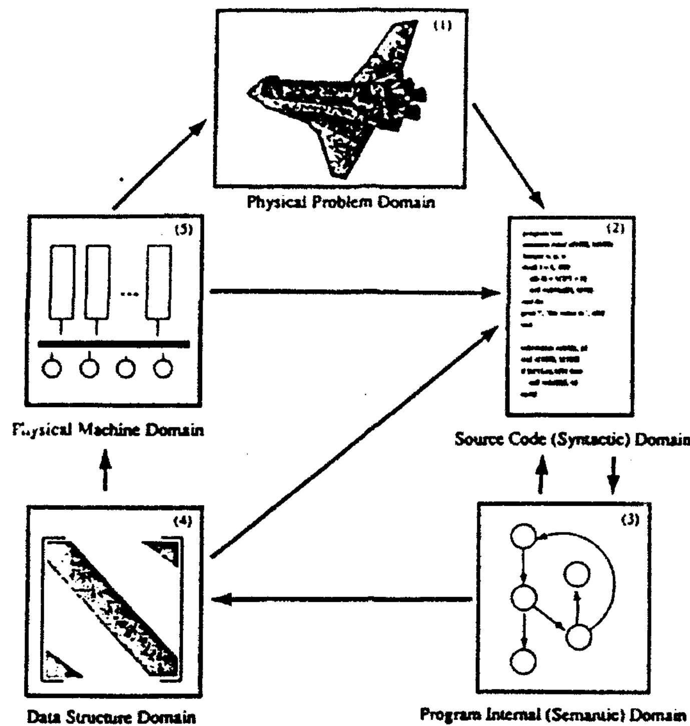


Figure 1: Program Development Cycle

compiler MIPRAC, in addition to generating parallel code for programs written in Fortran, C, and Lisp, can provide users with detailed insight into the dependence relationships within the code.

From the internal program domain, the application is transformed into the program data domain (Phase 4). This domain is characterized primarily by the existence of debugging tools that allow the user to validate the integrity of an application by verification of program state at various stages of its execution. This internal state is composed of data elements; that is, the condition of the program is evaluated with respect to the value of all of the internal data structures on which it is operating. The debugging tool MDB, an efficient machine-level debugger for the Cedar multiprocessor, was developed as part of this effort.

Once a program is running correctly, development moves into the physical machine domain (Phase 5), and performance improvement can begin. This involves not only gathering and presenting performance data, but also interpreting this data. The improvement process is characterized by program profiling, program tuning, and performance prediction. The user's conceptual view of a parallel program's operation often differs substantially from the actual execution behavior. To understand why a program might be running slowly, a global view of performance behavior is needed. New performance visualization tools help the user reconcile these differing views. The user may examine execution traces with TraceView or may peruse trace-based profiling information using CPROF and SUPERVU.

Program tuning requires finer control and additional performance detail (e.g., loop or statement flow). The coupling of performance tools with program restructuring software permits automatic instrumentation and display of performance data, to identify those portions of the code suitable for

additional performance tuning.

Once the program achieves an acceptable level of performance, the user can execute it and evaluate its results. Because the program was originally developed to help a scientist or engineer with a particular real-world problem, there is much current work on rendering the results in a manner that resembles the original problem domain. We developed a series of visualization tools for output rendering that close the loop in the program development process. VISTA, and its successor VASE, provide a basis for distributed execution and visualization of applications. VASE also provides a sophisticated framework for steering applications during execution, allowing the user to adjust problem parameters or examine data structures in depth without a tedious and expensive edit-compile-execute cycle.

The phases described above overlap fairly significantly; therefore, tools used in one phase are often useful in others. For example, the steering and visualization capabilities provided by VASE are valuable in achieving correct execution on a particular machine (Phase 4), optimizing its performance for a particular dataset (Phase 5), and experimenting with alternative algorithms (Phase 2).

The tools required to assist in the overall program development cycle can be categorized as follows:

- overall environment tools (phase 1 → 2)
- compilers and program structure analysis/query tools (phase 2 → 3)
- debugging and data visualization tools (phase 3 → 4)
- performance instrumentation and performance visualization tools (phase 4 → 5)

The work supported by this grant is an extension of the original Faust project, which was supported by AFOSR grant F49620-86-C-0136. With the support of the first grant, a set of X-based environment tools (including a graph manager, text manager, and a comprehensive project manager) were developed to form the foundation for a portable, integrated environment. However, during the same period outside research and development efforts, notably new user interface toolkits for version 11 of the X Window System, were gaining widespread acceptancce as portable standard interfaces. Recognizing that the goals of portability and integration could be met by adopting this industry-standard platform, the work supported by this follow-on grant focused principally upon tool development within an X11 framework. The remaining sections of this report describe the accomplishments in these areas.

2 Compiler and Program Structure Tools

Compilers transform source code into object code. Thus, referring back to Figure 1, compiler tools move the user from Phase 2 to Phase 3. Restructuring compilers for shared-memory parallel processors are the subject of considerable research at the Center for Supercomputing Research and Development. Several avenues of research were pursued as part of the this project.

2.1 Sigma *(D. Gannon)*

The Sigma system was one of the original components of the Faust project [1]. In its initial form it was an integrated tool for interactive restructuring of Fortran programs [2]. In the current phase of the AFOSR contract, the system was redesigned to provide basic compiler technology support for a wide variety of projects. More specifically, core elements of Sigma consisted of modules that could provide support the complete syntactic and semantic analysis of Fortran and C programs. It was decided that in the second version of Sigma, called Sigma II, that this support should be provided to a wider variety of tools that needed to understand C and Fortran programs. Sigma was then redesigned as a portable "tool kit" for building interactive and static program analysis systems.

Sigma II is intended use is for building end-user tools to aid in the process of program parallelization and performance analysis. Simply put, Sigma II is a data base for accessing and manipulating program control and data dependence information. The interface to the end-user tool consists of over 100 C functions that can be used to access and modify the data base.

Support for multiple source languages is supported in Sigma II. Input to the analysis data base can be either Fortran 77, Fortran 90, (some support for PCF Fortran is also provided), C, C++ and pC++ (which is extension to C++ for a variation on data parallel programming we call "object parallel" computation [3]). While this does not cover all the languages used for parallel programming, it does cover the majority of areas where large applications exist.

While there is little support for extensions to standard languages, Sigma II provides support for a very powerful annotation and directive system that lets users add source code comments that can be passed to the tool. These annotations can be used for data distribution directives, program instrumentation commands or restructuring directives.

The functions provided by the data base can be used by the end-user tool in one of three ways.

- *Extracting syntactic information.* This includes symbol and type table information, program control flow structure, user annotation and directive information, and access to the complete parse tree of the application.
- *Extracting semantic information.* This includes interprocedural definition and use summaries for each function and procedure, data dependence analysis and scalar propagation and symbolic analysis and simplification of scalar expressions.
- *Restructuring* the program by modifying the data base contents and to generating (unparsing) new versions of the source code based on the modifications.

Parallel programming tools are designed to work with user application program which are often very large and exist as multiple source files. In Sigma terms, each application program defines a *project* which consists of a set of source files, a set of dependence files and a project file. For each source file there is a corresponding dependence file generated by the parser. The dependence file for each source file contains a complete parse tree and symbol table for the program fragment in that file. In addition it contains a first-pass analysis of the interprocedural flow of for the functions and subroutines in that module.

The dependence files for an application constitute the data base. Each dependence file is a collection of graphs and tables corresponding to the structured parse tree for the program. The tool

builder can access and modify the information in the data base by means of the Sigma function library.

Examples of the types of tools that Sigma was designed to support include the following.

- *Interactive program restructuring tools to aid in exploiting parallelism.* A prototype of this application of Sigma was demonstrated in the SigmaCS system which was based on an extension of the popular EMACS program editor [4]. Another component of the program restructuring effort involved a details study of the way cache memory behavior can be improved by subtle program restructuring. This work was published in several conference proceedings but best summarized in [5] and the Ph. D. thesis [6].

- *Performance analysis tools that incorporated the structure and semantics of the application into the users view of the computation.* This has been a very fruitful area of Sigma application. Our initial experiments with extending Sigma for performance analysis focused on using the cache studies cited above to predict program behavior [7] which culminated in the Ph. D. thesis [8]. A second performance analysis project used Sigma as a component of a temporal relational algebra for a "Performance Spreadsheet". This project Sieve.1 [9, 10] is a Performance Evaluation "Spreadsheet " which allows users to interactively explore parallel program event trace files and ask "what if...?" questions about possible parallel optimization. Sekhar Sarukkai, who has just completed his Ph. D. thesis[11], directed this work. Another student, Suresh Srinivas, is continuing this work but with a greater focus on performance animation. Additional performance analysis work that was supported by sigma tools was a study of network traffic on the BBN series of machines [12].

- *Scientific visualization and computational steering tools* that provide a way for the user to see application data structures and to modify the progress of the computation without requiring the user to modify the code. This areas is the most advanced application of tool kits like Sigma. The interactive steering project described in this report represents the state of the art.

- *Parallel programming language design and implementation.* Our work in this area has focused on parallel extensions of the C and C++ programming language. The first effort was an experiment in vectorizing C [13]. Over the past two year we have focused on a Sigma based tool that translates a dialect of C++ to run on multiprocessor systems [3, 14–17] and a Ph.D. thesis [18]. This effort has grown into a separate DARPA sponsored endeavor to define a standard for "High Performance C++".

The first version of the Sigma system is now complete and has been distributed to many sites. The list of places where Sigma has been installed and used in research projects include: CSRD, Indiana University, Cornell University, Los Alamos, University of Colorado, University of California San Diego, Rice University, University of Wisconsin, University of Washington, Australian National University, University of Michigan, NASA Ames, University of Edinburgh, University of Rennes, University of Southampton, and the University of Delft. It is likely that it is in use in other institutions but this list contains only those sites that where we are engaged in direct and ongoing communications.

Sigma II has proven to be of substantial value in the construction of a number of parallel programming tools. By providing a toolkit which provides researchers an easy way to access

Fortran 90, C and C++ application syntax and semantics, we feel new ideas can be more quickly tested and production systems can be more easily built.

The complete source code, documentation and example applications are available by anonymous FTP. However, it must be stated that the system is still evolving. Our Fortran 90 front end has not been extensively tested because, at the time of this writing, there are still not very many programs that use the full set of features of Fortran 90 (such as modules, operator overloading and pointers.) As our test set becomes larger, we will eliminate existing bugs.

A summary of the project has now been published in [19]. A follow-up project to redesign Sigma to support High Performance Fortran and HPC++ has recently started and is supported by DARPA [20].

2.2 Restructuring Functional and Logic Programs *(D. Padua and D. Sehr)*

We have applied a collection of transformations developed for Fortran programs to the parallelization of functional and logic programs. The overall objective was the development of compiler techniques for the parallelization of Prolog programs. As functional programs comprise an interesting subset of logic programs, a number of our techniques were applied to functional programs. These transformations were then extended to the full Prolog language, and a complete framework for the parallelization of Prolog has been developed.

Our compilation techniques were shown to be capable of all the transformations of functional programs performed by Darlington's program synthesis technique, and are briefly described in a paper that appeared in the 1991 International Conference on Parallel Processing[21]. Moreover, our techniques are based upon the dependence graph framework developed for Fortran parallelization, and hence should be easily extensible to include new transformations, scheduling, and synchronization algorithms.

Having developed the techniques for functional programs, we extended them to logic languages, particularly Prolog. As the control flow of logic programs is very difficult to determine syntactically, our main effort was to develop a control flow framework in order to construct dependence graphs. Our techniques bring together a number of Prolog and Fortran techniques to expose loops in recursive procedures, and to analyze their dependences. In particular, abstract interpretation is used to obtain call/success modes and aliasing, which are used to refine the flow graph. Induction variable analysis and linear constraint solving expose loop control variables, which are then used to convert the program to a form containing loops. After such transformation, these programs can be processed by most Fortran techniques. A technical report describing these transformations was produced in October 1992[22], and a paper describing them will be submitted to the 1993 ACM Conference on Supercomputing.

To assess the effectiveness of these transformations a set of instrumentation programs was developed which measure inherent parallelism in Prolog programs. These programs measure critical path times according to two parallelism models, OR and AND/OR, and have been extended to include the parallel loop forms used in the above transformations. A paper describing these transformations appeared in the 1992 International Conference on Fifth Generation Computer Systems[23], and a revised and expanded version is to appear in New Generation Computing early in 1993.

2.3 Compilation of Symbolic Computations *(W. L. Harrison)*

We have investigated the problem of compiling non-numerical programs for parallel machines. This effort considered the problems that pointer manipulations (dynamic allocation, casting, pointer arithmetic, etc.) pose for a parallelizing compiling. Miprac is the vehicle for this research.

Miprac is an umbrella project, under which several smaller projects are being conducted. Miprac itself is a parallelizer and compiler that operates on a low-level intermediate form called MIL, and that has especially powerful interprocedural analysis of pointers and first-class procedures. The interprocedural analysis in Miprac is a whole-program abstract interpretation. It provides the compiler with precise side-effect, dependence and object lifetime information that is used for extraction of parallelism and management of memory. Miprac is described in [24].

Although work on the compiler itself has been discontinued, several projects are carrying forward the technology that was introduced in Miprac. We have created a software system called Z1 that is used to create program analyses automatically from high-level specifications. To build a compile-time analysis of reference counts, for example, one writes a operational semantics of MIL that counts references, and Z1 turns this semantics into a C program that performs the analysis. The analysis may be tuned (trading accuracy away to gain efficiency) by a simple mechanism we call *projection expressions*. Z1 is available by anonymous FTP from CSRD, and is described in [25,26].

A second project that is proceeding under the umbrella of Miprac is concerned with the compile-time analysis of explicitly parallel programs. Such programs can neither be analyzed nor optimized by traditional methods for the reason that they are nondeterministic. We have developed extensions to Miprac's interprocedural analyses that permit the analysis of side-effects, dependences and object lifetimes in such programs. Using these methods, we have successfully optimized some explicitly parallel programs that contain interesting synchronization. (The transformations that can be applied to such programs is often highly counter-intuitive.) This work is to be applied in compilers which are designed to accept a program that has been parallelized by a programmer, but which is to be further optimized, either for the extraction of additional parallelism, for the management of complex memory resources, or simply for the sake of improving the running time of sequential sections of the code (as mentioned above, even ordinary optimizations cannot be applied safely to such programs without such an analysis). This work is described in [27].

3 Debugging and Data Visualization Tools

After a program has been written down as source code and successfully compiled into object code, the behavior of the program for input datasets must be examined. Referring to Figure 1, the code has moved from Phase 3 to Phase 4. Two activities take place at this stage. First, the essential correctness of the source code must be verified, and second the essential correctness of the algorithm must be verified.

The first activity finds and eliminates errors that occured when the physical problem was mapped into source code (Phase 1 to Phase 2). Realistically, it sometimes also uncovers errors that crept in between Phase 2 and Phase 3 because of errors in the compiler software. This process requires interactive debugging tools.

The second activity finds errors at a higher level of abstraction. Perhaps the algorithms that the programmer chose in Phase 2 do not faithfully reflect the underlying physical process. Perhaps

some valid, but unexpected, result has occurred and the underlying cause must be uncovered. These tasks require a flexible facility for visualizing the program's data interactively, so that the user has the ability at runtime to peruse the program data in whatever fashion his or her investigations may dictate. A suite of powerful visualization methods is required. Further, it should be possible for the user to modify the state of the program at runtime, perhaps by adding some additional code to be executed at key places, in order to experiment with new or modified algorithms. These activites require interactive data visualization and application steering tools.

3.1 MDB — Xylem Parallel Debugger *(P. Emrath)*

For a machine to be useful users must be able to write and debug programs for the machine. Debugging parallel programs is harder than debugging sequential programs because parallel programs tend to be more complicated than their sequential counterparts. Parallel programs can consist of multiple tasks which run on multiple machines having both private and shared address spaces. Hence, the debugging of parallel programs requires additional capabilities that are not available with traditional debuggers for sequential programs.

MDB is a parallel debugger for Cedar[28] and its operating system Xylem[29] that provides facilities for examining the data and control structures of a Xylem process. As a prototype, MDB does not provide all the services one would find in a traditional sequential debugger, such as the ability to modify data or to set breakpoints. The debugger can show the machine registers for any processor, the status of any task, and memory locations within the private address space of a task, or in the shared address space. Even with such limited capabilities, it has proven to be a valuable tool for debugging actual programs on Cedar. We believe we have demonstrated the design approach with very positive results. From this point, it seems to be merely an exercise to add all the functionality, such as breakpoints, one has come to expect in a complete debugger.

The debugger is available in two versions. A reduced version of MDB allows for the examination of the memory and registers of a process and can display stack traces. The complete version of the debugger supports all the features of the reduced version with additional capabilities such as decoding (disassembly) of instructions and using the symbol table for symbolic addressing. In either version, MDB can be used interactively or it will dump error status when entered from a non-interactive program.

The design of MDB is different from many interactive debuggers in that rather than being a separate process that controls the target program through the operating system, MDB is a set of modules that are linked with the target program when creating an executable file. This means that MDB is not totally isolated from the target, but it has the advantages of being relatively simple to implement and can examine large amounts of program state very efficiently. Both these advantages stem from the fact that program state is directly accessible without requiring any operating system services.

The capabilites of MDB have evolved as it was used to debug itself during development. The user interface was repeatedly refined as continued use suggested changes to make debugging easier. Feedback from the user community has generally been favorable and a significant number of real bugs have been easily and quickly found once the faulty program was linked with the complete version of MDB. Even without the enhancements of being able to change values in a process or to set breakpoints interactively, MDB has proven to have a very practical design and to be a useful tool for debugging parallel programs.

A detailed description of MDB's user interface is contained in [30]. An overview of MDB was presented at the 1991 Supercomputing Debugging Workshop[31].

3.2 Visualization and Application Steering Environments *(J. Bruner, R. Haber, A. Tuchman, B. Bliss, D. Jablonowski, D. Hammerslag, G. Cybenko)*

Visualization allows the user of a scientific application to apply his or her innate visual pattern recognition skills to comprehend the application's data. The results generated by today's large scientific and engineering codes often comprise large datasets, perhaps several different arrays of values that must be interpreted together in order to be understood. Even for a single application, the datasets to be displayed, and the display method to be used, may vary considerably from one problem to the next. For example, some problems may call for simple displays such as a two-dimensional color-coded plot of a scalar field, while in other cases more sophisticated displays such as two- or three-dimensional vector fields may be required.

Further, in many cases, the user of a scientific or engineering application is not interested solely in the answer to a problem. Often, the reasons behind the answer are as interesting as, or even more interesting than, the answer itself. When confronted with a particular solution, the user may wish to play "what if" games. For example: "What if the time step were shortened?" "What if the electric field strength were increased?"

Thus, the user of a scientific application needs a visualization system that provides two additional capabilities. The first capability is "runtime visualization" — the ability to select datasets and their display methods interactively at runtime. This requires a suite of powerful display methods, together with a mechanism for constructing datasets at runtime. The second capability is "application steering" — an interactive means to modify the state of the program at runtime. This could include adding some additional code to the program to be executed at key places, in order to experiment with new or modified algorithms.

Further, the visualization system should be able to operate in a distributed environment. In a typical case, an engineering design code running on a large computational engine might be linked to a visualization program running on a rendering engine, and the entire system would be configured and controlled from a workstation. Such a distributed nature would allows the user to draw upon the strengths of different platforms (computation, rendering, user interaction) in a single environment.

CSRD developed two generations of runtime visualization and application steering software. Experience gained with the first system, Vista, was incorporated in the design of the second, VASE. The first version of the VASE system was completed in November 1992, and we are currently planning future extensions to the project.

3.2.1 Vista

Vista is an architecture and a framework for distributed run-time visualization of data. It provides a window into an application by showing program data automatically during execution. The system architecture is designed for a distributed or remotely executing application; however, the Vista model allows a data or trace file to replace the executing application, providing a visualization "data browser" for existing data or simulation runs. The data to be displayed and the type of display to be used are chosen interactively while the application is executing. It is not necessary to specify the data or graphics technique before compilation as with conventional

graphics tools. With minimal instrumentation, an application run in the Vista environment will have its data (variables and data structures) made available to a visualization system on a remote workstation. Any data display can be enabled or disabled at any time. The application may execute locally, on a remote supercomputer, on several clusters of a shared memory computer, or even across a network of distributed computers.

Vista's major user-visible modules are the Visualization Manager (VM) and the Application Executive (AE)[32]. The VM is the user interface and display manager. It runs as a separate process from the application. The AE is co-resident with the application program; therefore, it runs in the context of the application and has access to its entire address space. The application need not run on the same machine as the VM. Other modules, which are not directly visible to the user, provide data transfer and synchronization between the VM and the AE.

To use Vista, the programmer prepares the source code by defining *vis-points*, visualization breakpoints. These represent significant places within the program where graphical displays should be updated or where the user may wish to interact with the code. The programmer defines vis-points by inserting calls to the Vista routine `avbreak`.

The prepared source code is then compiled and linked with the Vista runtime library (which includes the AE). To execute the application, the user first starts the VM (typically on the local workstation) and then starts the application (typically on a remote supercomputer). When the application reaches the first call to `visbreak` it will initialize the AE and connect to the VM. As part of the initialization, the AE will read the symbol table for the application program, thereby obtaining information about all of the variables in the program.

At this time the user may ask for and display graphically any variables in the program. The form in which the variables are displayed is determined by the user's choice of *display method*. Vista provides a basic set of display methods. In addition, external display methods may be used. When an external display method is employed, the VM passes the appropriate data to the display method through a communication channel. This capability allows Vista to employ such powerful visualization systems as AVS[33] or ApE[34].

An initial version of Vista was completed and distributed. However, work on the Vista project was suspended when development of VASE began. VASE borrowed from Vista, both in its design concepts and its implementation (e.g., VASE employs a derivative of the Application Executive in Vista). The Vista project culminated with the presentation of two papers in 1991, one at the Fifth SIAM Conference on Parallel Processing for Scientific Computing[35] and the second at Visualization '91[36].

3.2.2 VASE

Overview

Work on VASE began at CSRD in 1991, and version 1.0 was completed in November 1992. The VASE project grew out of previous experience in high-performance distributed visualization, including the RIVERS project [37,38] at the National Center for Supercomputing Applications as well as the Vista project (Section 3.2.1). VASE also relies upon the program analysis and restructuring tools developed as part of the Sigma project (Section 2.1).

VASE is a collection of programming tools and system software that add runtime visualization and application steering capabilities to application codes. VASE supports the activities of three

different classes of users during an application's lifetime: the code developer, the distributed application configurer, and the end user. At each stage, VASE collects and organizes information about the program, which it then communicates to users in later stages.

VASE employs both control-flow and data-flow paradigms. Visualization systems typically are based upon the data-flow model, in which data passes from one functional module to another. Each module waits until sufficient data is present at its inputs, at which time it "fires," producing output data that is transmitted to other modules downstream. This paradigm is attractive because it naturally accomodates asynchronous distributed computing, and it more easily allows reconfiguration of a set of functional modules (e.g., to substitute one display method for another); therefore, VASE uses the data-flow paradigm to manipulate the interactions between a set of (possibly distributed) processes.

VASE communications are presently based upon DTM[39]. The endpoints are called *ports*. Ports are unidirectional, and can be associated with an entire process or only with a single breakpoint within that process. A VASE *connection* is a communication link that connects an output port to an input port. Reads and writes on ports can be defined as either blocking or non-blocking.

Although the data-flow model is well-suited to the interprocess communication used in distributed visualization, it is cumbersome when used to express the complex logic typically found in large-scale scientific applications. It also does not reflect the thought process of an developer who writes code in Fortran. By contrast, a control-flow model can readily represent an application's internal structure. The structure of the code can be defined by a control flow graph, where the nodes in the graph represent functional blocks and the arcs are transitions between those block. VASE uses the control-flow model to describe the internal structure of processes and identify appropriate locations for visualization and steering operations.

Building a VASE Application

The code developer writes the source code. Using VASE tools, he or she defines major functional blocks in the program; steering breakpoints, which are intermediate points at which interaction with the data in the application is meaningful; and, for each steering breakpoint, the variables that can be read or written.

Figure 2 shows how a steerable application is built using VASE tools. The code developer inserts directives into the source code to identify major functional blocks in the program. (These directives, which take the form of "pseudo comments" are the only changes that must be made to the original source code). The VASE program graph builder, which is based upon Sigma, reads the annotated source code and constructs a hierarchical control-flow graph.

A second VASE tool, the breakpoint insertion tool, displays this graph visually. The developer may peruse the graph, expanding and collapsing hierarchically-nested blocks. By clicking upon the arcs between blocks, the developer can insert steering breakpoints — places in the program where steering operations can take place. (Steering breakpoints in VASE are roughly analogous to vis-points in Vista.) For each visualization breakpoint, the developer selects the variables from that portion of the program that should be made available for read access (e.g., visualization) and write access (steering).

Another Sigma based tool, the breakpoint code generator, combines the breakpoint information with the hierarchical control flow graph and generates source code consisting of the original program augmented with calls to the VASE breakpoint library. This source code is then compiled

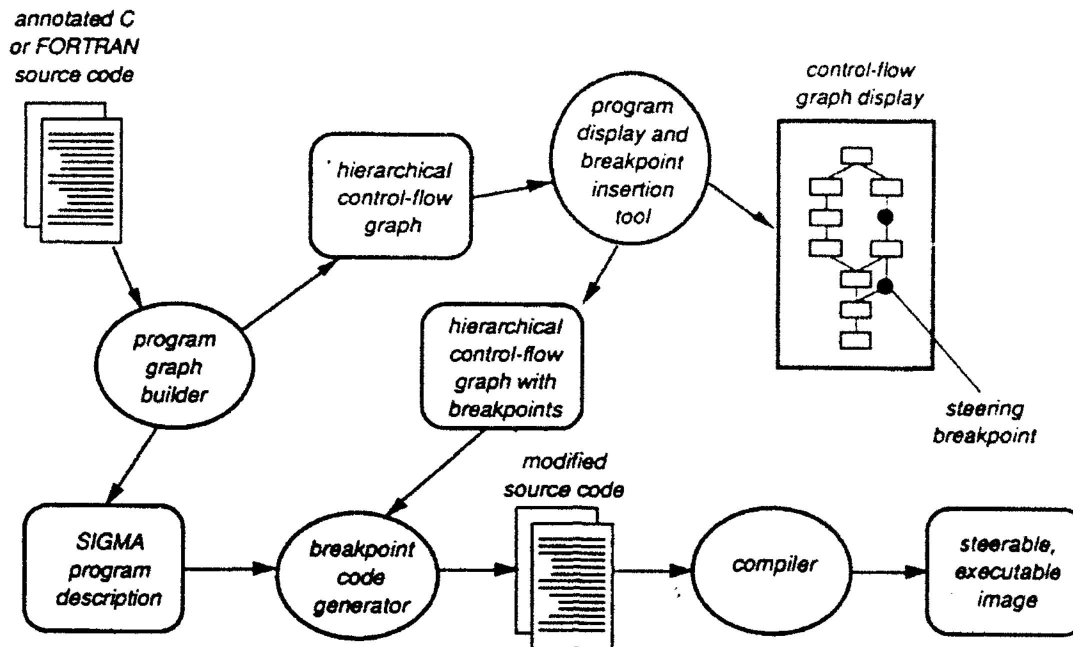


Figure 2: Building a Steerable Application with VASE

and linked with the VASE library to form a steerable executable image.

Application Configuration

The distributed application configurer combines a set of application programs and visualization programs into a distributed system. Based upon information created by the code developer, the configurer defines communication channels between the programs and defines how the individual codes will exchange data.

Compiled languages promote efficient execution, while interpreted languages encourage interaction and allow the program logic to be modified at run time. VASE combines the best features of the two approaches. VASE includes a breakpoint script interpreter, based upon the Application Executive[32], that is activated when a program encounters a steering breakpoint. The interpreter parses and executes breakpoint scripts, consisting of actions for inter-process communication and steering. In effect, the interpreter provides a high-level debugging facility. However, in constrast with conventional debuggers, the interpreter shares the application program's address space, giving it efficient access to large blocks of data.

VASE includes a graphical configuration and execution control tool. The configurer uses this tool to assign executable processes to host processors, to set up communication links between processes, and to configure steering breakpoints. Breakpoints may cause the process to pause and await interactive input, to execute a script, to execute a script and then pause, or to have no effect at all.

The configurer will typically define scripts for applications that access variables of interest in

the application code and write them to VASE output ports. In addition, he or she will define scripts for the visualization processes to read data from the VASE ports and display it.

Application Execution

The end user runs the configured distributed application. In many cases he or she will specify an input dataset and will observe the progress of the application through a set of visual displays defined by the configurer. In addition, the information created by the code developer and the configurer is available to the end user. The end user can exploit this information to change the visualization technique, examine different data sets, and even modify the behavior of the program.

Like the configurer, the end user employs the VASE configuration and execution tool. The end user can modify scripts created by the configurer, can add or delete processes from the configuration, can and can create or delete communication ports and connections.

Future Plans

VASE version 1.0, comprising the program graph builder, breakpoint insertion tool, breakpoint code generator, and configuration and execution tool was completed in November 1992. A distributed supercomputing application based on VASE for shape optimization of structures of variable topologies was created and reported in [40] and also formed the basis for a demonstration videotape[41].

3.3 Visibility Ordering of Meshed Polyhedra (*P. Williams*)

A *visibility ordering* of a set of objects from some viewpoint is an ordering such that if object *a* obstructs object *b*, then *b* preceeds *a* in the ordering. Certain visualization techniques, particularly direct volume rendering based on projective methods require a visibility ordering of the polyhedral cells of a mesh so the cells can be rendered using color and opacity compositing. Visibility ordering the cells of rectilinear meshes (or certain classes of regular meshes based on a decomposition of a rectilinear mesh) is straightforward[42]. However, for other types of meshes, such as curvilinear or unstructured meshes, it is not immediately obvious how to compute this ordering.

We have developed a simple and efficient algorithm for visibility ordering the cells of any acyclic convex set of meshed convex polyhedra. This algorithm, called the Meshed Polyhedra Visibility Ordering (MPVO) Algorithm, orders the cells of a mesh in linear time using linear storage. Preprocessing techniques and/or modifications to the MPVO Algorithm permit nonconvex cells, nonconvex meshes (meshes with cavities and/or voids), meshes with cycles, and sets of disconnected meshes to be ordered. The MPVO Algorithm can also be used for domain decomposition of finite element meshes for parallel processing. The data structures for the MPVO Algorithm can be used to solve the spatial point location problem.

The MPVO Algorithm was used to generate the picture of a scientific data set of approximately 70,000 tetrahedra which is shown on the cover of [43]. The basic ideas of the MPVO Algorithm were suggested by Herbert Edelsbrunner in a conversation regarding his paper on the acyclicity of cell complexes[44]. A similar algorithm to the MPVO Algorithm, also based on Edelsbrunner's suggestions, was developed independently by Max, Hanrahan, and Crawfis[45].

The Binary Space Partition (BSP) tree algorithm[46] is not suitable for visibility ordering large meshes because the algorithm uses splitting planes (even when not required to break cycles). Because the cells are meshed, a large number of cells could be split, resulting in a potential explosion in the total number of cells. Analysis of the BSP tree algorithm by Paterson and Yao[47] suggests that its performance could be $O(f^2)$, where f is the number of faces in the original mesh. An A-buffer[48] is also not suitable for visibility ordering large meshes because there are too many transparent cells at each pixel, making memory requirements prohibitive with current hardware. Goad[49] describes a special purpose program written in Lisp for visibility ordering polygons. This approach might be adapted for polyhedra; further investigation may be warranted. The *brute force* algorithm, for visibility ordering an acyclic mesh, calculates an obstructs relation for every pair of the n cells in the mesh, and requires at least $O(n^2)$ cycles.

Using the MPVO algorithm as a basis, we developed a parallel (MIMD) volume rendering system for scalar data defined on irregular or curvilinear meshes. In the process, the MPVO algorithm was parallelized. The particular platform utilized was a Silicon Graphics VGX 4D/360VGX with 6 CPUs. The focus of the system is to visualize 3D finite element and scattered data. (When scattered data is triangulated it results in an irregular data set.)

The volume renderer, based on cell projection rather than ray tracing, utilizes a hierarchy of rendering approximations, graphics hardware, preprocessing and filtering techniques[50, 51]. Using these techniques, volumetrically rendered images of curvilinear and irregular data sets with over 1,000,000 cells were generated in 15 seconds (without filtration)[50]. Using filtering, similar performance is possible for even larger data sets. Data sets with up to 100,000 cells were rendered in less than 2 seconds.

In addition, a volume density optical model was developed[52]. This model was developed as a theoretical basis for volume rendering finite element data as opposed to scanned data sets where material classification is required.

The National Center for Supercomputing Applications is currently productizing the renderer (constructing a user-friendly interface for it) so it can be made available to scientists.

4 Performance Instrumentation and Visualization Tools

Once a large code is running successfully on a target machine, the focus of a developer's attention moves from obtaining correct results to improving performance. Referring to Figure 1, the attention shifts from Phase 4 to Phase 5.

Supercomputer codes, by the very nature, are intensive consumers of computational resources. Therefore, it is essential that they run as efficiently as possible. However, it is also the nature of supercomputer codes to be large and complex. As a result, the accumulation of years of incremental improvements may produce a lengthy code with which no one person is expert.

Users commonly employ profiling tools to assist in this task. A profiling tool can provide high-level, general statistical information on which code segments consume the most time. A major advantage of most profiling tools is that they are relatively automatic, requiring little user intervention. They are also available for a wide variety of machines.

Unfortunately, most profiling tools are limited to providing a list of the most well-optimized subroutines. For many parallel machines or vector supercomputers, the most interesting data falls at the loop level and is invisible to the profiler.

Another problem with existing profilers is that they are machine-specific. In order to make meaningful direct comparisons between machines, users need results that are commensurable. This is possibly the most outstanding weakness of existing performance evaluation tools. Table 1 shows the fifteen most time-consuming routines as measured by GPROF on an Alliant FX/80, an Alliant FX/2800, and a Sun Sparcstation-1. It is difficult to recognize from those results that the same code is being run on the three machines. The user is required to divine the purpose of implementation-specific routines with cryptic names such as `mcount` and `__unlock_wo_prof`, which are not routines in his code. Furthermore, users want the time spent in library functions (`_dexp_fortran_val_`, etc.) attributed to the routines using those functions, not to the library routines. While performance data for system functions like `mcount` does provide information, most users want information only about the routines in their code over which they exercise control.

Sparcstation-1		Alliant FX/2800		Alliant FX/80	
% Time	Routine	% Time	Routine	% Time	Routine
14.6	_ckrat_	14.8	__cond_lock_wo_prof	21.9	_ckrat_
12.3	_exp	10.1	_ckcpms_	12.2	_daxpy_
11.5	_pow_di	9.1	_ckwyp_	9.3	_ckwyp_
10.1	_daxpy_	7.2	_ckrat_	8.0	_mceval_
7.3	mcount	6.8	_daxpy_	7.1	_mcedif_
4.9	_ckwyp_	6.3	_ckhml_	6.2	_ckcpms_
4.0	_fun_	5.9	_cksmh_	5.8	_djpwr_fortran_
3.8	_ckcpms_	5.7	_mcedif_	3.6	_fun_
3.5	_mcedif_	5.6	__unlock_wo_prof	3.1	_ckhml_
3.1	_mceval_	3.8	_fun_	3.1	_cksmh_
2.5	_ckhml_	3.7	_getrusage	2.9	_mdifv_
2.2	_mdifv_	2.4	_mceval_	1.5	_dgbsl_
2.1	_cksmh_	2.3	_dexp_fortran_val_	1.4	_ckytx_
1.5	_ckytx_	2.0	_mdifv_	1.0	_copy_
1.2	.mul	1.7	_ckytx_	1.0	_ckrhoy_

Table 1: GPROF Results of One Code on Three Machines

Many widely-used profilers (e.g., the GPROF utility on many Unix systems) collect data by sampling the program counter at regular intervals. The resulting data represents the average amount of time spent in each portion of the code during the sampling interval, but much detail is lost. If a routine performs identical work on each invocation but these invocations have drastically differing execution times, this information can indicate important effects such as paging from virtual memory or cache usage. Sampling, which averages over the entire execution of the program, misses these differences. Further, sampling cannot easily account for varying amounts of work performed in different invocations. Consider the program shown in Figure 3. This program was executed and profiled on an RS/6000 with the sampling-based tool GPROF, and the resulting performance data is shown in Figure 4.

Figure 4 indicates that 100% of the time was spent in C, which was called 200 times. Further, it states that C was called 100 times from A and spent 1.27 seconds when C was invoked from A (second row). Similarly, the profile states that C spent 1.27 seconds when C was invoked from B.

```
      DO 100 I = 1,10
         CALL A
         CALL B
100      CONTINUE
      STOP
      END

      SUBROUTINE A
      DO 20  I = 1, 10
         CALL C(10)
 20      CONTINUE
      RETURN
      END

      SUBROUTINE B
      DO 30  I = 1, 10
         CALL C(100)
 30      CONTINUE
      RETURN
      END

      SUBROUTINE C(K)
      DO 300 J = 1, K
         DO 100 I = 1, K
100         CONTINUE
300      CONTINUE
      RETURN
      END
```

Figure 3: An Example Program

```
                                     called/total       parents
index  %time      self descendents  called+self        name      index
                                     called/total       children
                  1.27        0.00     100/200           .A [4]
                  1.27        0.00     100/200           .B [5]
[1]    100.0      2.54        0.00         200           .C [1]
------------------------------------------------
                  0.00        2.54         1/1     .__start [3]
[2]    100.0      0.00        2.54         1          .main [2]
                  0.00        1.27       10/10           .A [4]
                  0.00        1.27       10/10           .B [5]
------------------------------------------------
                                                  <spontaneous>
[3]    100.0      0.00        2.54                 .__start [3]
                  0.00        2.54         1/1        .main [2]
------------------------------------------------
                  0.00        1.27       10/10        .main [2]
[4]     50.0      0.00        1.27          10           .A [4]
                  1.27        0.00      100/20           .C [1]
```

Figure 4: Dynamic Call Graph for the Example Program

(The rest of the figure describes statistics for the other sections of the program.) However, it is clear from inspection of the source code that this sampling-based analysis is incorrect: most of the time is being spent in subroutine C when C is being called from B.

For these reasons, we have developed a set of performance tools based upon source-level instrumentation and trace collection. Section 4.1.1 describes TraceView, a tool for examining these traces. Section 4.1.2 describes the tools CPROF and SUPERVU, which employ structured traces to provide both summary profiling and trace perusal capabilities.

Measurements of a code on a machine provide an accurate view of the performance currently being obtained, as well as the areas where the code performs well or fails to perform well. However, when faced with the task of improving a code's performance, a designer often needs to know not only the achieved performance but also the potential for performance improvement. In effect, he or she needs a tool that can give some quantitative assessment of what the performance could be. Section 4.2 describes how we modified the MaxPar performance simulator[53], written at CSRD, to fill this role.

4.1 Trace-Based Performance Tools *(J. Bruner, G. Cybenko, D. Hammerslag, D. Jablonowski, A. Malony, S. Sharma)*

4.1.1 TraceView

TraceView is an event-trace display and analysis tool. It supports the viewing of timeline displays of event occurrences together with performance metrics.

The TraceView architecture is based on the concept of a *trace visualization session*. A session consists of files, views, and displays. The topmost level of a session specifies a set of trace files to visualize. For each file, a set of *views* can be constructed. A view is a template that defines a region of the trace, including the beginning location, the ending location, and event filtering. Then, for each view, a set of displays can be created. Although views and displays can only be defined within a single trace file, (therefore disallowing displays that combine data from multiple trace files), multiple trace files can be displayed simultaneously in the same session.

TraceView operates upon trace files that contain a series of event records, where an event represents a transition from one state to another. (TraceView assigns no meaning to the states themselves. Typically, the user will instrument an application code with subroutine calls to an trace-generation library. Each such call will use a unique identifier. In this case the trace file describes the movement of the program between the trace points.) In addition to encoding the current and previous state, the event record contains an event type and a time stamp. Events may also include other data; for instance, on the Cray Y-MP event records may contain data from the hardware performance monitors.

TraceView displays events as a set of timelines that show the event transitions and the associated other information (e.g., HPM performance metrics). The display methods and user interface are based upon X11 and Motif. TraceView supports many user interface functions, including the ability to save session configurations between invocations.

TraceView's capabilities and user interface are described in detail in [54].

4.1.2 CPROF and SUPERVU

CPROF is a trace-based profiler for serial and shared-memory parallel computers. An automatic

instrumentation tool augments the application source code with calls to a tracing library. This approach is similar to Cray Research's FlowTrace[55]; however, unlike Flowtrace, CPROF employs source-level instrumentation and is therefore widely portable.

The tracing routines reflect the application's structure. The execution of the application is divided into regions delimited by a pair of instrumentation points where a routine in the tracing library is invoked. An automatic instrumentation tool which is based on the Cedar Fortran[56] preprocessor marks subroutines and loop nests in Fortran programs automatically. Users may manually insert additional instrumentation points to provide additional detail in some areas of the program.

The instrumented program is linked with the tracing library and executed. At link time, the use may choose a tracing library that employs special hardware (such as the tracing hardware available for Cedar[28, 57]), or a portable software-based tracing library that does not require any special hardware.

As the program runs it produces a set of trace records. Each trace record includes the identity of the trace point and other relevant performance information, such as elapsed or CPU time. In addition, the structure of traces allows for machine-specific extensions on platforms that provide additional information (such as directly-accessible hardware performance counters).

CPROF is an analysis tool that reads traces and prints summary statistics. It profiles the application using wall-clock time and computes statistics based upon the exclusive timings of code regions (the time spent in each region minus the time spent in its children). CPROF provides information on

- the amount of time the machine spent executing code in the named code segment (subroutine, basic block, loop).
- the number of times the code segment (subroutine, basic block, loop) was executed.
- statistical variations in the timing of the code segment, the minimum , maximum, average, standard deviation times and the instances where these times have occurred.
- detailed execution profile of each code segment for performance tuning.
- a dynamic calling tree of all the code segments.
- input data for further theoretical performance analysis by other tools.

SUPERVU is an interactive performance visualization tool. SUPERVU reads the same event trace files that CPROF reads, but unlike CPROF it allows the user to browse through the data interactively. SUPERVU runs under X11/Motif, and we have ported it to multiple platforms including Sparcstations, Sun 3's and the Alliant FX/2800.

SUPERVU provides a different set of displays than TraceView (Section 4.1.1). While TraceView does not associate any particular meaning to events, SUPERVU's analysis is based upon the knowledge that events come in entry-exit pairs. Thus, SUPERVU is able to refine the raw event data to display derived values such as the amount of time spent within each code region, the number of times each region was executed, etc.

The SUPERVU user may choose from a variety of performance data displays for each process in the application. Their diversity allows the user to analyze the application at multiple levels. The

displays include flat profiles, detailed profiles, dynamic calling trees, and overall process behavior. These are displayed in a visually informative manner.

SUPERVU is able to process information about the behavior of the computer system while the application is running. SUPERVU treats the context switches as a code segment within the program and treats this code segment the same as other code segments. Among other things it can subtract the time spent "switched out".from the execution time of the program.

A more detailed description of CPROF and SUPERVU, together with a case study for ARC2D, one of the Perfect Benchmarks®, has been submitted for publication[58]. Additional information about the tools can also be found in [59] and [60].

4.2 MaxPar *(S. Ho)*

Overview

The MaxPar performance simulator[53, 61], written at CSRD, employs execution-driven simulation to estimate parallel performance. It was inspired by Fortran parallelism estimation techniques[62], and their first implementation, as found in COMET[63].

MaxPar's input is a sequential Fortran 77 program. MaxPar defines simulated time in programs in terms of an abstract machine, where operations require a predefined amount of time. The abstract machine may be deined with a finite or finite number of processors. For each variable in the original program, MaxPar creates *shadow variables*. The *read shadow* and *write shadow* represent the last point in simulated time at which the associated variable was read or written, respectively. MaxPar then identifies all of the operations upon variables (e.g., add, multiply) and augments the program with code that computes the new values for the associated shadows. By scheduling operations according to the shadow values that it computes (effectively computing all data dependences at runtime), MaxPar is able to compute a histogram of the available parallelism for each simulated time unit. It also can generate a trace file containing all of the operations as it has scheduled them.

Although the primary development of MaxPar, which is ongoing, was supported by other agencies, we made some modifications to MaxPar[64] to enable it to be used together with our trace-based performance tools to identify performance bottlenecks.

Extensions for TraceView

We extended MaxPar to generate trace records in the format used by TraceView (Section 4.1.1). MaxPar predicts the time when each routine in an application is entered and exited, and writes these to the trace file. It also is capable of writing its histogram of parallelism in TraceView-compatible format. These modifications allow side-by-side comparisons of MaxPar predicted results and actual measured results. Such comparisons may, for example, compare how well the actual execution takes advantage of parallelism, compared to the parallelism figures reported by MaxPar.

The Parallel Performance Advisor

MaxPar was also modified to work more closely with the trace-based performance tools CPROF and SUPERVU (Section 4.1.2) to combine measured performance data with predictions. This project, called the Parallel Performance Advisor (PPA) brought together these two types of analysis to find

the portions of the code with the greatest time consumption *and* unexploited parallelism. An overview of this approach was published in [65]. Additional examples are described in [64].

The PPA consists of two major parts. First, the trace-based performance tool CPROF is used to create a profile of the application's actual execution time. Based upon this information, CPROF generates a list of directives to MaxPar specifying the routines in each of the most time-consuming subtrees of the call graph.

We modified MaxPar to restrict its analysis to the specified subtree of routines, assuming each invocation of the routines is disjoint. In general there are a small number of these subtrees, and each is processed by a separate invocation of MaxPar. These MaxPar processes are independent from one another and can be run in parallel.

To collect data within a subtree of routines, we extended MaxPar to permit gathering data at the module level, instead of only computing program-level summaries. In addition, a barrier is inserted at the entry and exit points of the highest-level routine to be instrumented.

MaxPar's output summarizes the parallelism available within each routine and its descendants. The interactive tool SUPERVU reads MaxPar's output and combines it with the measured performance data, which it can then display in the form of a dynamic calling tree. Figure 5 shows an example for FLO52, one of the Perfect Benchmarks®. Each box depicts performance information for one routine in the calling tree, giving the name of the routine, the time consumed by the routine (in seconds), and the number of calls to the routine. In addition, the measured data is augmented with the MaxPar-computed parallelism (which is shown in parentheses), giving the percentage of total time attributable to the routine and its average parallelism.

By examining the combined MaxPar and measured data, the PPA user can identify likely candidates for performance improvement. In the example in Figure 5, for example, over 70% of the total CPU time is consumed by the four routines `dflux`, `eflux`, `psmoo`, and `dfluxc`, and they also show good parallelism values. Thus, a good start would be to parallelize these routines first.

MaxPar also was modified to perform its analysis only for a specified interval of an application's total execution time. To use this facility, the user first runs MaxPar on the entire application to get an overall parallelism histogram and identify the major phases in the computation, and then "zooms in" on intervals of interest to determine how the performance is affected by changes in the parallelism, synchronization costs, etc.

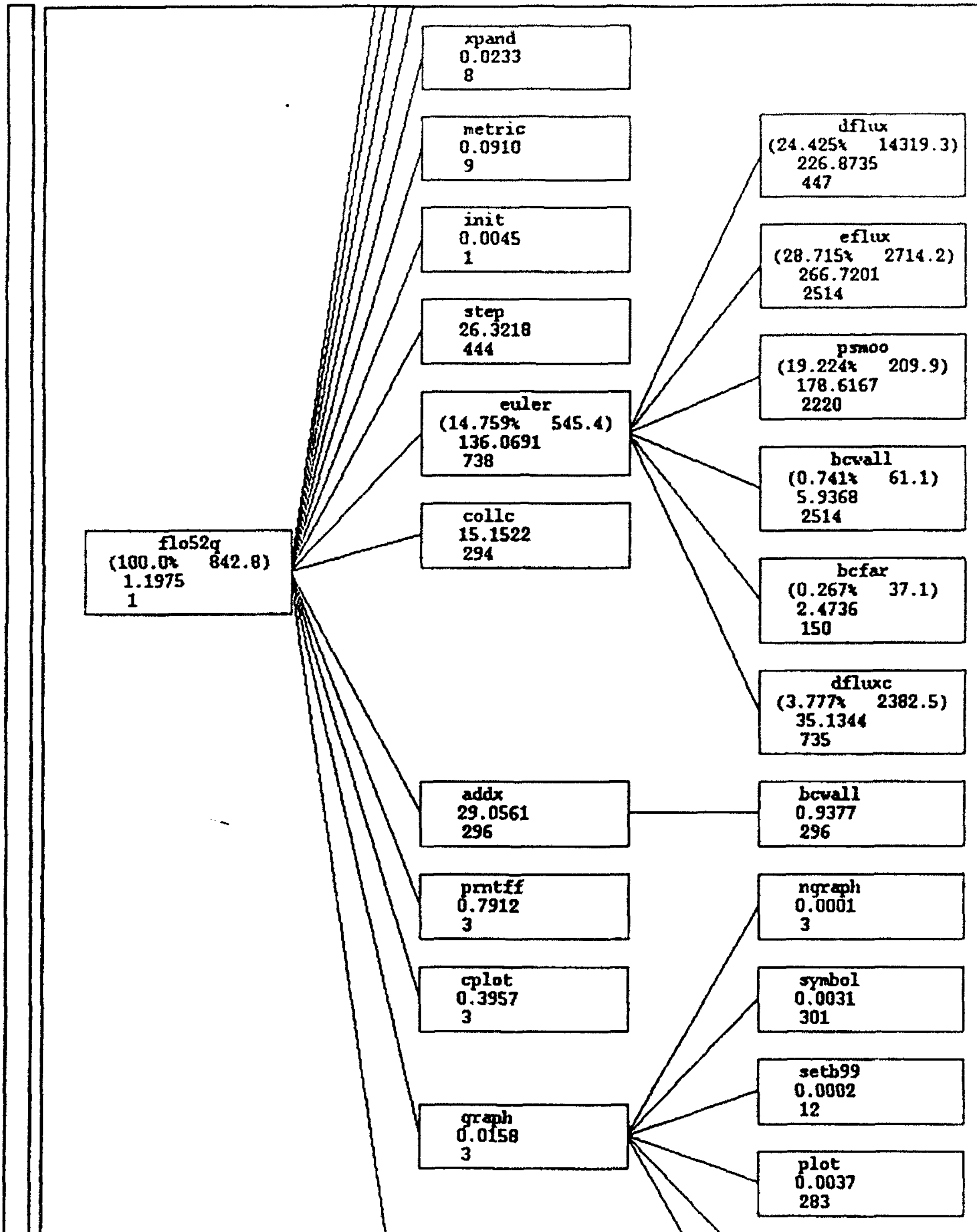


Figure 5: FLO52 Dynamic Calling Tree from CPROF and MaxPar

5 Bibliography

[1] J. Vincent Guarna, D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur, "Faust: An Integrated Environment for the Development of Parallel Programs," *IEEE Software*, pp. 20–27, July 1989.

[2] D. Gannon, D. Atapattu, B. Shei, and M. Lee, "A Software Tool for Programming Parallel Systems," in *Parallel Computations and Their Impact on Mechanics* (A. Noor, ed.), pp. 81–92, The American Society of Mechanical Engineers, 1987.

[3] J. K. Lee and D. Gannon, "Object Oriented Parallel Programming: Experiments and Results," in *Proceedings of Supercomputing '91*, pp. 273–282, November 1991.

[4] B. Shei and D. Gannon, "Sigmacs: A Programmable Program Restructuring Tool," in *1990 Workshop on Compilers for Parallel Systems*, MIT Press, August 1990.

[5] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation," *Journal Parallel and Distributed Computing*, vol. 5, pp. 587–616, October 1988.

[6] M.-H. Lee, *Data Localization in Parallel Computer Systems*. PhD thesis, Computer Science Department, Indiana University, December 1990.

[7] D. Atapattu and D. Gannon, "Building Analytical Models into an Interactive Performance Prediction Tool," in *Proceedings of Supercomputing '89*, pp. 521–530, November 1989.

[8] D. Atapattu, *Performance Prediction of Supercomputer Programs*. PhD thesis, Computer Science Department, Indiana University, June 1991.

[9] S. Sarukkai and D. Gannon, "SIEVE: A Performance Debugging Environment for Parallel Programs," *Journal of Parallel and Distributed Computing*, 1993. to appear in the Special Issue on Parallel Systems Performance Tools.

[10] S. Sarukkai and D. Gannon, "Parallel Program Visualization using SIEVE.1," in *Proceedings of International Conference on Supercomputing*, pp. 157–166, 1992.

[11] S. Sarukkai, *Interactive Tools for Performance Analysis*. PhD thesis, Computer Science Department, Indiana University, December 1992.

[12] F. Bodin, D. Windheiser, W. Jalby, D. Gannon, D. Atapattu, and M. Lee, "Performance Evaluationand Prediction for Parallel Algorithms on the BBN GP1000," in *Proceedings of the 1990 ACM International Conference on Supercomputing*, June 1990.

[13] D. Gannon, J. Vincent A. Guarna, and J. K. Lee, "Static Analysis and Runtime Support for Parallel Execution," in *Monographs in Parallel and Distributed Computing* (C. Jesshope and D. Klappholz, eds.), Pitman Publishing, 1989.

[14] D. Gannon and J. K. Lee, "Object Oriented Parallelism: pC++ Ideas and Experiments," in *Proceedings of Japan Society for Parallel Processing*, pp. 13–23, 1991.

[15] D. Gannon and J. K. Lee, "On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions," in *Proceedings Conpar-Vap*, September 1992.

[16] D. Gannon, "Libraries and Tools for Object Parallel Programming," in *Proceedings, CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing*, September 1992. To appear: Springer-Verlag *Lectures on Computer Science*.

[17] S. X. Yang, J. K. Lee, and S. P. Narayana, "Programming an Astrophysics Application in an Object-Oriented Parallel Language," in *Proceedings of SHPCC-92 Scalable High Performance Computing Conference*, pp. 236–239, IEEE Computer Society Press, 1992.

[18] J. K. Lee, *Object Oriented Parallel Programming Paradigms and Environments For Supercomputers*. PhD thesis, Computer Science Department, Indiana University, June 1992.

[19] D. Gannin, J. K. Lee, B. Shei, S. Sarukkai, S. Narayana, N. Sundaresan, D. Attapatu, and F. Bodin, "Sigma II: A Tool Kit for Building Parallelizing Compilers and Performance Analysis Systems," in *Proceedings 1992, IFIP Edinburgh Workshop on Parallel Programming Environments*, April 1992. Also in *Programming Environments for Parallel Computing*, IFIP Transactions A-11, N. Topham, R. Ibbet, and T. Bemmerl, eds., North-Holland Press, pp. 17–36.

[20] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Srinivas, "Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools." Indiana University technical report.

[21] D. C. Sehr, L. V. Kale, and D. A. Padua, "Fortran-Style Transformations for Functional Programs (Extended Abstract)," in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 282–283, August 12–16 1991.

[22] D. C. Sehr, *Automatic Parallelization of Prolog Programs*. PhD thesis, University of Illinois at Urbana-Champaign, October 1992.

[23] D. C. Sehr and L. V. Kale, "Estimating the Inherent Parallelism in Prolog Programs," in *Proceedings of the International Conference on Fifth Generation Computer Systems*, June 1–5 1992.

[24] W. Harrison III and Z. Ammarguellat, "A Program's Eye View of Miprac," in *Languages and Compilers for Parallel Computing* (D. Gelernter, N. Nicolau, and D. Padua, eds.), MIT Press, August 1992.

[25] K. Yi and L. Harrison, "System Z1 Programming Manual," Technical Report 1283, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, February 1993.

[26] K. Yi and W. L. Harrison III, "Automatic Generation and Management of Interprocedural Program Analyses," in *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pp. 246–259, January 1993.

[27] J. Chow and W. Harrison III, "Compile-time Analysis of Parallel Programs that Share Memory," in *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, 1991.

[28] J. Konicek, T. Tilton, A. Veidenbaum, C. Z. Zhu, D. Kuck, P. C. Yew, D. Lavery, R. Lindsey, D. Pointer, T. Murphy, J. Andrews, and S. Turner, "The Organization of the Cedar System," in *Proceedings of the International Conference on Parallel Processing*, vol. I, pp. 49–56, August 12–16 1991.

[29] P. Emrath, M. Anderson, R. Barton, and R. McGrath, "The Xylem Operating System," in *Proceedings of the International Conference on Parallel Processing*, vol. I, pp. 67–70, August 12–16 1991.

[30] P. Emrath and B. Marsolf, "mdb — Xylem Parallel Debugger User's Guide," Technical Report 1180, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1991.

[31] P. Emrath and B. Marsolf, "mdb — A Parallel Debugger for Cedar," in *Proceedings of the Supercomputer Debugging Workshop '91*, November 1991.

[32] B. Bliss, "Interactive Steering Using the Application Executive," in *Proceedings of Supercomputing Debugging Workshop*, November 14–16 1991.

[33] C. Upson *et al.*, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, vol. 9, pp. 30–42, July 1989.

[34] D. S. Dyer, "A Dataflow Toolkit for Visualization," *IEEE Computer Graphics and Applications*, vol. 10, pp. 60–69, July 1990.

[35] A. Tuchman, G. Cybenko, D. Jablonowski, B. Bliss, and S. Sharma, "VISTA: A System for Remote Data Visualization," in *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, March 25–27 1991.

[36] A. Tuchman, D. Jablonowski, and G. Cybenko, "Run-Time Visualization of Program Data," in *Proceedings of Visualization '91*, October 25 1991.

[37] R. Haber, "Scientific Visualization and the RIVERS Project at the National Center for Supercomputing Applications," in *Scientific Visualization*, pp. 231–236, IEEE Computer Society Press, 1990.

[38] R. Haber, D. A. McNabb, and R. Ellis, "Eliminating Distance in Scientific Computing: An Experiment in Televisualization," *International Journal of Supercomputing Applications*, vol. 4, pp. 71–89, Winter 1990.

[39] J. Terstriep, "DTM: Data Transfer Mechanism." User documentation, National Center for Supercomputing Applications, December 1990.

[40] R. Haber, B. Bliss, D. Jablonowski, and C. Jog, "A Distributed Environment for Run-Time Visualization and Application Steering in Computational Mechanics," *Computing Systems in Engineering*, vol. 3, pp. 501–515, December 1992.

[41] J. Bruner, R. Haber, B. Bliss, D. Jablonowski, and C. Jog, "Visualization and Application Steering Environment." Center for Supercomputing Research and Development videotape, 1992.

[42] G. Frieder, D. Gordon, and R. A. Reynolds, "Back-to-Front Display of Voxel-Based Objects," *IEEE Computer Graphic Applications*, vol. 5, pp. 52–60, January 1985.

[43] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," in *Proceedings of the San Diego Workshop on Volume Visualization*, pp. 63–70, December 10–11 1990.

[44] H. Edelsbrunner, "An Acylicity Theorem for Cell Complexes in *d* Dimension," *Combinatorica*, vol. 10, no. 3, pp. 251–260, 1990.

[45] N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume COherence for Efficient Visualization of 3D Scalar Functions," in *Proceedings of the San Diego Workshop on Volume Visualization*, pp. 27–33, December 10–11 1990.

[46] H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by A Priori Tree Structures," *ACM SIGGRAPH Computer Graphics*, vol. 14, pp. 124–133, July 1980.

[47] M. S. Paterson and F. F. Yao, "Binary Partitions with Applications to Hidden-Surface Removal and Solid Modeling," in *Proceedings of the Fifth Annual Symposium on Comput. Geometry*, pp. 23–32, June 1989.

[48] L. Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 103–108, 1984.

[49] C. Goad, "Special Purpose Automatic Programming for Hidden Surface Elimination," *ACM SIGGRAPH Computer Graphics*, vol. 16, pp. 167–178, July 1982.

[50] P. L. Williams, "Interactive Splatting of Nonrectilinear Volumes," in *Proceedings of Visualization '92*, pp. 37–44, October 1992.

[51] P. L. Williams, *Interactive Direct Volume Rendering of Data Defined on Nonrectilinear Meshes*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1992.

[52] P. L. Williams and N. Max, "A Volume Density Optical Model," in *Proceedings of the ACM SIGGRAPH Volume Visualization Workshop '92*, pp. 61–68, October 1992.

[53] D.-K. Chen, "MaxPar: An Execution Driven Simulator for Studying Parallel Systems," Master's thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, October 1989.

[54] A. Malony, D. Hammerslag, and D. Jablonowski, "TraceView: A Trace Visualization Tool," *IEEE Software*, vol. 8, pp. 19–28, September 1991.

[55] Cray Research, Inc., "UNICOS Performance Utilities Reference Manual," May 1989.

[56] M. Guzzi, "Cedar Fortran Programmer's Handbook," Technical Report 601, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1987.

[57] S. Sharma and A. Malony, "The Tracing Facility in the Cedar System," Technical Report 1015, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1990.

[58] S. Sharma, R. Bramley, P. Sinvahl-Sharma, J. Bruner, and G. Cybenko, "P3S: Portable, Parallel Program Performance Evaluation System." Submitted for publication, September 1992.

[59] S. Sharma, R. Bramley, P. Sinvahl-Sharma, and G. Cybenko, "Evaluating, Visualizing and Analysing the Parallel Program Performance," Technical Report 1169, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1991.

[60] P. Sinvhal-Sharma and S. Sharma, "CPROF: A Trace Based Profiler for Shared Memory Multiprocessor Systems," Technical Report 1016, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1990.

[61] D.-K. Chen and P.-C. Yew, "An Empirical Study of DOACROSS Loops," in *Proceedings of Supercomputing '91*, pp. 630–632, November 1991.

[62] D. Kuck and A. Sameh, "A Supercomputing Performance Evaluation Plan," in *Lecture Notes in Computer Science No. 297: Proc. of First Int'l. Conf. on Supercomputing, Athens, Greece* (C. P. E.N. Houstis, T.S. Papatheodorou, ed.), (New York, NY), pp. 1–17, Springer-Verlag, June 1987.

[63] M. Kumar, "Measuring Parallelism in Computation-Intensive Scientific Engineering Applications," *IEEE Transactions on Computers*, pp. 1088–1098, 1988.

[64] S. Ho, "MaxPar Extensions for Isolating Performance Problems," Technical Report 1240, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1992.

[65] G. Cybenko, J. Bruner, S. Ho, and S. Sharma, "Parallel Computing and the Perfect Benchmarks™," *International Symposium on Supercomputing*, November 6–8 1991.

Complete Bibliography Supported by AFOSR 90-0044

This list, which names all publications related to support from AFOSR 90-0044, is ordered first by year (beginning with the most recent publications) and then alphabetically by the name of the first author.

[BHBJ92] J. Bruner, R. Haber, B. Bliss, D. Jablonowski, and C. Jog, "Visualization and Application Steering Environment." Center for Supercomputing Research and Development videotape, 1992.

[GaSa92] E. Gallopoulos and Y. Saad, "Efficient Solution of Parabolic Equations By Krylov Approximation Methods," *SIAM Journal for Scientific Statistical Computing*, vol. 13, pp. 1236–1264, September 1992.

[Ghos92] S. Ghosh, *Automatic Detection of Nondeterminacy and Scalar Optimization in Parallel Programs*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, October 1992.

[HBJJ92] R. Haber, B. Bliss, D. Jablonowski, and C. Jog, "A Distributed Environment for Run-Time Visualization and Application Steering in Computational Mechanics," *Computing Systems in Engineering*, vol. 3, pp. 501–515, December 1992.

[Ho92] S. Ho, "MaxPar Extensions for Isolating Performance Problems," Technical Report 1240, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1992.

[NeGh92] R. H. B. Netzer and S. Ghosh, "Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization," in *Proceedings of the International Conference on Parallel Processing*, August 17-21, 1992.

[SeKa92] D. C. Sehr and L. V. Kale, "Estimating the Inherent Parallelism in Prolog Programs," in *Proceedings of the International Conference on Fifth Generation Computer Systems*, June 1–5 1992.

[Seme92] B. D. Semeraro, *Operator Splitting Methods for the Navier Stokes Equations*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, March 1992.

[ViHa92] C. A. Vidal and R. B. Haber, "Design Sensitivity Analysis for Rate-Independent Elastoplasticity." May 1992.

[Will92a] P. L. Williams, "Visibility Ordering Meshed Polyhedra," *ACM Transactions on Graphics*, vol. 11, pp. 103–126, April 1992.

[Will92b] P. L. Williams, "Is Interactive Direct Volume Rendering Feasible for Nonrectilinear Volumes?," Technical Report 1177, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1992.

[Will92e] P. L. Williams, *Interactive Direct Volume Rendering of Data Defined on Nonrectilinear Meshes*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1992.

[WiMa92] P. L. Williams and N. Max, "A Volume Density Optical Model," in *Proceedings of the ACM SIGGRAPH Volume Visualization Workshop '92*, pp. 61–68, October 1992.

[Blis91] B. Bliss, "Interactive Steering Using the Application Executive," in *Proceedings of Supercomputing Debugging Workshop*, November 14–16 1991.

[CBHS91] G. Cybenko, J. Bruner, S. Ho, and S. Sharma, "Parallel Computing and the Perfect Benchmarks™," *International Symposium on Supercomputing*, November 6–8 1991.

[Cybe91] G. Cybenko, "Supercomputer Performance Trends and the Perfect Benchmarks," *Supercomputing Review*, pp. 53–60, April 1991.

[EmMa91a] P. Emrath and B. Marsolf, "mdb — A Parallel Debugger for Cedar," in *Proceedings of the Supercomputer Debugging Workshop '91*, November 1991.

[EmMa91b] P. Emrath and B. Marsolf, "mdb — Xylem Parallel Debugger User's Guide," Technical Report 1180, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1991.

[JaTu91] D. Jablonowski and A. Tuchman, "Vista Users Manual," Technical Report 1068, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1991.

[MaHJ91] A. Malony, D. Hammerslag, and D. Jablonowski, "TraceView: A Trace Visualization Tool," *IEEE Software*, vol. 8, pp. 19–28, September 1991.

[SeKP91] D. C. Sehr, L. V. Kale, and D. A. Padua, "Fortran-Style Transformations for Functional Programs (Extended Abstract)," in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 282–283, August 12–16 1991.

[Staf91] C. Staff, "The Cedar Project," in *Proceedings of the International Conference on Parallel Processing*, August 12-16, 1991.

[TCJB91] A. Tuchman, G. Cybenko, D. Jablonowski, B. Bliss, and S. Sharma, "VISTA: A System for Remote Data Visualization," in *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, March 25–27 1991.

[TuJC91a] A. Tuchman, D. Jablonowski, and G. Cybenko, "A System for Remote Data Visualization," Technical Report 1067, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1991.

[TuJC91b] A. Tuchman, D. Jablonowski, and G. Cybenko, "Run-Time Visualization of Program Data," in *Proceedings of Visualization '91*, October 25 1991.

[Will91] P. L. Williams, "Applications of Computational Geometry to Volume Visualization," in *Proceedings of the Third Canadian Conference on Computational Geometry*, August 1991.

[Berr90] M. W. Berry, *Multiprocessor Sparse SVD Algorithms and Applications*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, November 1990.

[GaBe90] H. Gao and M. Berry, "Performance Studies of LAPACK on Alliant FX/80 and 1 Cedar Cluster," Technical Report 1001, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1990.

[HaAm90] W. L. Harrison and Z. Ammarguellat, "PARCEL and MIPRAC: Parallelizers for Symbolic and Numeric Programs," in *Proceedings of the International Workshop on Compilers for Parallel Computers*, December 1990.

[Hamm90] D. Hammerslag, "Faust Library Browser User's Manual," Technical Report 961, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, January 1990.

[Jabl90b] D. Jablonowski, "GMB: Graph Manager/Browser," Technical Report 968, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, February 1990.

[JaGu90] D. Jablonowski and J. Vincent Guarna, "A Tool Integration Facility for Programming Environments," in *Proceedings of First International Conference on Systems Integration*, vol. 1, pp. 162–170, April 23-26, 1990.

[MiPa90] S. Midkiff and D. Padua, "Issues in the Compile-Time Optimization of Parallel Programs," in *Proceedings of International Conference on Parallel Processing*, vol. II, pp. 105–113, August 1990.

[Neem90] H. J. Neeman, "Visualization Techniques for Three-Dimensional Flow Fields," Master's thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1990.

[SeKP90] D. C. Sehr, L. V. Kale, and D. A. Padua, "OR Parallel Prolog with Side Effects: the Results of Several Benchmarks," Technical Report 981, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, November 1990.

[ShTu90] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," in *Proceedings of the San Diego Workshop on Volume Visualization*, pp. 63–70, December 10–11 1990.

[SMBS90] S. Sharma, A. Malony, M. Berry, and P. Sinvhal-Sharma, "Run-Time Monitoring and Performance Visualization of Concurrent Programs for Shared Memory Multiprocessors," in *Proceedings of Supercomputing '90*, pp. 784–793, November 12–16, 1990.

[TuBe90] A. Tuchman and M. Berry, "Matrix Visualization in the Design of Numerical Algorithms," *ORSA Journal of Computing*, vol. 2, no. 1, 1990.

[Will90] P. L. Williams, "Issues in Interactive Direct Projection Volume Rendering of Nonrectlinear Meshed Data Sets," in *Progress Report for San Diego Workshop on Volume Visualization*, December 1990.

[WiSh90] P. Williams and P. Shirley, "An A Priori Depth Ordering Algorithm For Meshed Polyhedra," Technical Report 1018, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1990.

[GaFM89] E. Gallopoulos, G. Frank, and U. Meier, "Experiments with Elliptic Problem Solvers on the Cedar Multicluster," in *Proceedings of Fourth SIAM Conference Parallel Processing Scientific Computing*, pp. 245–250, December 1989.